Décodage du jeu d'instruction des processeurs x86

Première partie

Par Neitsa [FRET]

Version 1.1

Table des matières

Table des illustrations

Avant propos

Ce document traite de la manière de décoder le jeu d'instruction des processeurs appartenant à la famille des processeurs x86 et traite aussi bien du mode x86 32 bits que du mode 64 bits.

Pour la liste des téléchargements (manuels et programmes), veuillez vous reporter à l'addendum situé à la fin de ce document.

Une connaissance minimale de l'assembleur est requise pour une bonne compréhension de ce document.

Mode d'opération

Le mode d'opération d'un processeur est avant tout le fait du système d'exploitation s'exécutant sur ce même processeur. Lors de l'allumage de l'ordinateur, le système d'exploitation commute rapidement le processeur dans un des modes d'exploitation nécessaire à son fonctionnement. Les modes d'exploitation des processeurs de la famille x86 sont :

- Mode réel (*Real mode*)
- Mode Non-réel[1] (*Unreal mode*)
- Mode Virtuel 8086[2] (*Virtual 8086 mode*)
- Mode protégé (*Protected mode*)
- Mode long (*Long mode*)

Notez que les systèmes d'exploitation modernes 32 bits utilisent le mode protégé, tandis que les systèmes d'exploitation 64 bits utilisent le mode long. Les autres modes sont utilisés par les anciens systèmes (DOS, Windows 3.1, etc.).

[1] Le mode non-réel (*Unreal mode*) n'est pas à proprement parler un mode spécifique des processeurs de la famille x86, mais plutôt un effet de bord du mode réel.
[2] Le mode Virtuel 8086 est un attribut du mode protégé et non pas un mode distinct d'opération.

Il faut aussi garder à l'esprit que les processeurs de la famille x86 sont rétro compatibles, cela signifie qu'un processeur 16 bits ne peut exécuter que du code 16 bits, qu'un processeur 32 bits ne peut exécuter que du code 16 ou 32 bits, et que finalement un processeur 64 bits est capable d'exécuter du code 64, 32 et 16 bits.

Particularité des processeurs x86-64

Les processeurs 64 bits de la famille x86 proposent un nouveau mode d'opération nommés *IA-32e* chez Intel et *Long Mode* chez AMD.

Les processeurs x86 64 bits de génération récente permettent ainsi de faire fonctionner un système d'exploitation soit en mode « héritage » (*legacy*), soit en mode long (*long mode*).

Le tableau ci-dessous liste les différents modes d'opération des processeurs x86-64 :

<table>
<tr><th colspan="2">Mode d'opération</th><th>Système d'exploitation requis</th><th>Recompilation d'application requise</th></tr>
<tr><td rowspan="2">Mode Long</td><td>Mode 64 bits</td><td rowspan="2">Système d'exploitation 64 bits.</td><td>Oui</td></tr>
<tr><td>Mode compatibilité</td><td>Non</td></tr>
<tr><td rowspan="3">Mode d'héritage</td><td>Mode protégé</td><td rowspan="2">Système d'exploitation 32 bits.</td><td rowspan="3">Non</td></tr>
<tr><td>Mode Virtuel 8086</td></tr>
<tr><td>Mode réel</td><td>Système d'exploitation 16 bits.</td></tr>
</table>

- **Mode d'héritage** :
  Le processeur 64 bits se comporte alors comme un processeur 32 bits si le système d'exploitation est 32 bits ou 16 bits dans le cas d'un système 16 bits, sans aucune distinction. L'accès au mode long (et à ses sous modes) est alors impossible.

Mais cette nouvelle architecture 64 bits permet aussi l'exécution du code dans deux sous modes non exclusifs, à la seule condition que le système d'exploitation soit un système 64 bits.

Ces deux sous modes sont descendants du **mode long** :

- **Mode compatibilité** : Ce mode permet à un système d'exploitation 64 bits d'exécuter du code 32 bits ou 16 bits sans toutefois opérer de changement de mode. Le code s'exécute alors comme si le système d'exploitation était en fait en 32 ou 16 bits (suivant l'application). Les applications 32 bits restent cantonnées au 32 bits, et les applications 16 bits fonctionnent, elles, en mode 16 bits. D'un point de vue applicatif, le mode compatibilité est semblable au mode protégé.

- **Mode 64 bits** : Seul un système d'exploitation 64 bits peut avoir accès à ce mode et seules les applications 64 bits peuvent y fonctionner. Ce mode apporte quelques nouveautés que nous allons voir dans le prochain chapitre et tout au long de ce document.

Le mode long permet donc à la fois de faire fonctionner des applications 64 bits et des applications 32 et / ou 16 bits.

Notez que sous Windows, le mode compatibilité est fourni au travers de WOW64 (Windows On Windows64) qui permet uniquement de faire fonctionner des applications 32 bits. Les applications 16 bits ne fonctionne pas sous Windows 64 bits.

Notez finalement que la compréhension des modes d'exécution est très importante pour décoder avec succès des instructions.

Les nouveautés du mode 64-bit

À l'heure actuelle, la part de marché des processeurs 64 bits devient de plus en plus grande, aussi un chapitre sur les nouveautés introduites par le mode 64 bits s'impose.

Contrairement à ce que l'on pourrait penser, le mode 64 bits des processeurs x86 n'apporte quasiment aucune instruction générale (seule une instruction d'ordre générale [MOVSXD] fait son apparition !). La quasi totalité des nouvelles instructions[3] sont donc des instructions système, inexploitables en mode protégé.

Outre un espace d'adressage théorique plus grand (2^{64} octets de mémoire adressable) les principales innovations du mode 64 bits sont l'extension des registres généraux à 64 bits et l'apparition de nouveaux registres généraux (ainsi que de nouveaux registres SSE et système).

Les registres généraux sont étendus à 64 bits et gardent leurs « parties » héritées des modes 16 et 32 bits. D'une manière générale, ces nouveaux registres étendus en 64 bits sont préfixés avec un « R » (et abandonne le « E » qui voulait justement dire « étendu »). Voici l'exemple du registre EAX (32 bits) qui étendu en 64 bits devient donc RAX :

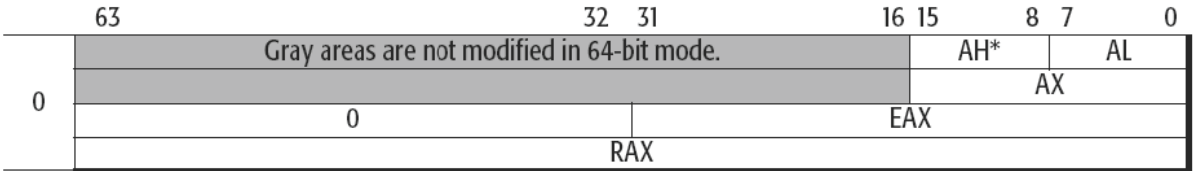

Figure 1 : Composition d'un registre 64 bits

En mode 64 bits, on peut donc toujours accéder à :

- EAX : partie basse 32 bits (bits 0 à 31) de RAX.
- AX : partie basse 16 bits (bits 0 à 15) de EAX.
- AH : partie haute 8 bits (bits 8 à 15) de AX.

[3] De nouvelles instructions SSE ont fait leur apparition sur les dernières générations de processeurs mais ne sont pas directement liées au mode 64 bits.

- AL : partie basse 8 bits (bit 0 à 7) de AX.

On ne peut donc ni accéder directement à la partie haute 16 bits de EAX, ni à la partie haute 32 bits de RAX.

Les nouveaux registres généraux 64 bits, au nombre de 8, sont nommés de R8 à R15 et sont utilisables comme les autres registres généraux. On dénombre donc à présent les registres suivants :

• 16 registres 8 bits « bas » : AL, BL, CL, DL, SIL, DIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B. [Avec 'B' pour BYTE].

On notera ici que les parties 8 bits de RSI (SIL), RDI (DIL), RBP (BPL) et RSP (SPL) sont désormais accessibles directement (ce qui n'était pas le cas en 16 ou 32 bits où, par exemple, la partie 8 bits du registre ESI n'était pas accessible directement).

• 4 registres 8 bits « haut » : AH, BH, CH, DH (utilisable seulement dans un cas bien précis : quand un préfixe REX est présent).

• 16 registres 16 bits : AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W. [Avec 'W' pour WORD]

• 16 registres 32 bits : EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D. [Avec 'D' pour DWORD].

• 16 registres 64 bits : RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15.

On notera également :

- L'arrivée de 8 nouveaux registres SSE : XMM8 à XMM15.
- Le passage d'EFLAG (registre de drapeaux) en 64 bits : RFLAG.
- Le pointeur d'instruction (EIP en 32 bits) passe lui aussi en 64 bits : RIP.

- 8 nouveaux registres système de contrôle : Cr8 à Cr15.
- 8 nouveaux registres de déboguage : Dr8 à Dr15.

Qu'est-ce qu'une instruction ?

Une instruction est une suite d'octets contenue dans un fichier exécutable. Le processeur lit chaque instruction et l'exécute, changeant ainsi d'état à chacune d'entre elles.

Le but de ce document traitant avant tout de la manière dont sont constituées les instructions et comment on peut décoder celles-ci, les changements d'états du processeur ne seront pas traités.

Les processeurs de la famille x86 sont des processeurs à technologie CISC (***complex instruction set computer***) ce qui sous-entend que ces processeurs possèdent de nombreuses instructions, aptes à faire chacune des opérations complexes. Chacune de ces opérations prend plus ou moins de temps pour s'exécuter suivant sa complexité.

L'architecture CISC est le contraire de l'architecture RISC (R pour « *Reduced* ») qui ne comporte qu'un nombre réduit d'instructions, chacune d'entre elles ne faisant qu'une opération primaire. Toutes les instructions d'un processeur RISC mettent le même temps à s'exécuter.

À titre d'information, les derniers nés des processeurs x86 comportent environ 1000 instructions.

Mais comment sait-on que tel octet (par exemple 0x90) correspond à telle instruction (NOP dans notre exemple) ?

Pour répondre à cette question, les fondeurs de processeurs mettent à notre disposition une « *opcode map* » (carte des opcodes) qui est un gros tableau où

chaque instruction est dûment référencée. Il suffit alors d'y retrouver le numéro d'opcode et d'y lire le mnémonique s'y rapportant (on notera dans un premier temps que tous les octets de la carte des opcodes ne référencent pas forcément une instruction).

Mais le travail de décomposition d'une instruction est cependant beaucoup plus complexe que cela et ne s'arrête pas à la simple lecture d'un tableau…

Composition générale d'une instruction

Une instruction d'un processeur x86 obéit à des règles strictes, sans quoi le processeur ne saurait exécuter celle-ci convenablement. La taille minimale d'une instruction est de 1 octet, tandis que sa taille maximale est de 15 octets.

Voyons maintenant la composition d'une instruction :

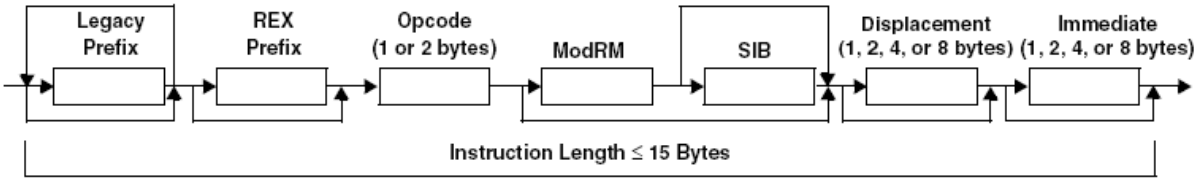

Figure 2 : Composition d'une instruction

Tous les constituants d'une instruction, exposés ci-dessus, sont optionnels sauf bien sûr, l'octet (ou les octets) d'opcode.

Nous allons voir à présent dans les différents chapitres suivants ce que font les différents octets d'une instruction et comment ils affectent l'instruction qu'ils composent. Tout d'abord nous allons voir rapidement chacun de ces groupes.

Legacy Prefix

Les « *legacy prefixes* » (préfixes d'héritage) sont nommés comme tels en mode 64 bits. Dans les autres modes on les nomme tout simplement « *prefixes* ».

Les préfixes sont des octets particuliers et ne peuvent pas être confondus avec d'autres octets (comme ceux d'une instruction). Nous verrons spécifiquement chaque octet de préfixe séparément.

Le rôle des préfixes est d'influer d'une certaine manière (différente suivant le préfixe mis en jeu) sur l'instruction qu'ils précèdent. Voilà les règles de base concernant les préfixes :

- Les préfixes ont tous une taille d'un octet.
- L'octet de préfixe, s'il est présent, est le premier à composer à l'instruction.
- Il peut y avoir de 0 à 4 préfixes.

Les préfixes étant divisés en 4 groupes différents, ils doivent aussi respecter la règle suivante :

- Il ne peut y avoir qu'un seul préfixe appartenant à un groupe spécifique.

Les préfixes supplémentaires (plus de 4 préfixes pour une instruction) sont simplement ignorés et dans le cas où 2 ou plusieurs préfixes d'un même groupe seraient présents, le résultat de l'instruction est dit « indéterminé ».

Notons finalement que l'ordre dans lequel sont placés les préfixes n'a pas d'importance. Il arrive aussi qu'une instruction ne nécessite aucun préfixe ou que les préfixes qui lui soient ajoutés n'aient aucune influence sur l'instruction. Dans ce cas là les préfixes sont dits superflus et sont tout simplement ignorés par le processeur.

Les groupes de préfixes :

• Group 1 : Lock et préfixes de répétition :
  — • 0xF0 — LOCK
  — • 0xF2 — REPNE / REPNZ
  — • 0xF3 — REP ou REPE / REPZ

• Group 2 : Préfixes de "*segment override*" :
  — • 0x2E — Préfixe CS « *segment override* ».
  — • 0x36 — Préfixe SS « *segment override* ».
  — • 0x3E — Préfixe DS « *segment override* ».
  — • 0x26 — Préfixe ES « *segment override* ».
  — • 0x64 — Préfixe FS « *segment override* ».
  — • 0x65 — Préfixe GS « *segment override* ».

  — Branch hints :
  — • 0x2E — Branche non prise (*Branch not taken*).
  — • 0x3E — Branche prise (*Branch taken*).

• Group 3
  — • 0x66 — Préfixe de taille d'opérande (*Operand-size override prefix*).

• Group 4
  — • 0x67 — Préfixe de taille d'adresse (*Address-size override prefix*).

Préfixes REX

Les préfixes REX sont une nouveauté puisqu'ils sont apparus avec le mode 64 bits et ne sont utilisables que dans ce mode (ce qui, comme nous l'avions vu dans le

chapitre sur les modes d'exécution, nécessite un processeur 64 bits et un système d'exploitation 64 bits).

Les préfixes REX s'étendent de 0x40 à 0x4F (ils remplacent les instructions INC et DEC disponibles en 16 et 32 bits qui se retrouvent repoussées plus loin dans la carte des opcodes et sont donc finalement toujours accessibles).

Les préfixes REX permettent :

- De spécifier une taille d'opérande de 64 bits (par ex. RAX au lieu d'EAX).
- De sélectionner les nouveaux registres généraux disponibles en mode 64 bits.
- De sélectionner les nouveaux registres SSE disponibles en mode 64 bits.
- De sélectionner les nouveaux registres système disponibles en mode 64 bits.

Comme nous l'avons vu dans le chapitre dédié aux nouveautés du mode 64 bits, les registres généraux ont été étendus à 64 bits. Le tableau ci-dessous dresse un rapide aperçu des possibilités offerte par le préfixe REX.

Type de registre	Sans REX	Avec REX
Registres d'octet (BYTE)	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L
Registres de mot (WORD)	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Registres de double mot. (DWORD)	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Registres de quadruple mot. (QUADWORD)	Non Applicable	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

On voit donc qu'en mode 64 bits, les registres étendus à 64 bits ne sont pas utilisés par défaut, il faut ajouter un préfixe REX pour les sélectionner !

Opcode

Nous allons voir dans ce chapitre, de manière basique, l'octet le plus important d'une instruction : l'opcode.

Opcode est la contraction d'« *operation code* ». C'est l'octet responsable de la signification même de l'instruction. Il existe donc un opcode précis pour une instruction précise.

Ainsi, par exemple, l'octet 0x90 est l'opcode signifiant que le processeur n'effectuera rien. Comme il est particulièrement difficile de se remémorer chaque opcode et sa signification, on établit une relation entre l'opcode et un nom qui peut être aisément retenu et qui donne un sens à l'opcode. Ce nom est le mnémonique de l'instruction.

NOP (pour « ***No Op****eration* ») est ainsi le mnémonique de l'opcode 0x90.

Le mnémonique est utile pour la programmation assembleur ou le désassemblage afin qu'il soit humainement possible de comprendre un listing désassemblé. Le processeur, lui, ne comprend que des suites de 0 et 1 et ne sait en aucun cas ce qu'est le mnémonique d'une instruction comme « PUSH EAX ». Il comprend seulement la suite de bits qui compose les différents octets de l'instruction.

La conversion du mnémonique (et dans un sens plus général, de l'instruction) vers l'opcode est le fait d'un compilateur. La conversion de l'opcode vers le mnémonique est l'apanage du désassembleur. Cette opération est donc possible dans les deux sens :

Mnémonique ➔ compilation ➔ Opcode

Opcode ➔ Désassemblage ➔ Mnémonique

Soit :

NOP ➔ Compilation ➔ 0x90

0x90 ➔ Désassemblage ➔ NOP

Vue en détail, une instruction humainement lisible se compose ainsi :

- 1 mnémonique qui est le nom de l'instruction et indique sa signification.
- 1 premier opérande optionnel nommé « première source ou destination ».
- 1 deuxième opérande optionnel nommé « source ».

Voyons cela en détail :

Ci-dessous sont présentées trois instructions tirées d'un désassemblage et qui illustrent le détail d'une instruction vue auparavant :

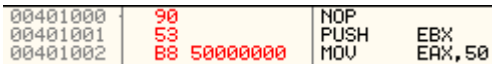

Figure 3 : Instruction et opérandes

- La première instruction, NOP ne comporte aucun opérande et se constitue uniquement d'un mnémonique.

- La deuxième instruction, PUSH EBX, se constitue du mnémonique PUSH et d'un seul opérande. Cet opérande (EBX dans notre cas) est appelé « première source » car il n'y a pas de deuxième opérande.

- La troisième instruction MOV EAX, 50 se constitue du mnémonique MOV, de l'opérande « destination » EAX et de l'opérande « source » 50.

Ainsi d'une manière générale, le mnémonique indique ce que fait l'instruction et les opérandes nous indiquent ce que manipule l'instruction.

Expérimentons à présent un peu avec les opcodes. Pour cela nous utiliserons le débogueur OllyDbg.

Ouvrez le programme 10NOP.exe, qui se compose de 10 fois l'instruction NOP suivi de l'instruction RETN, avec OllyDbg.

```
00401000 <>/$  90      NOP ; Point d'entrée du programme
00401001   |.  90      NOP
00401002   |.  90      NOP
00401003   |.  90      NOP
00401004   |.  90      NOP
00401005   |.  90      NOP
00401006   |.  90      NOP
00401007   |.  90      NOP
00401008   |.  90      NOP
00401009   |.  90      NOP
0040100A   \.  C3      RETN
```

La première ligne du programme à être exécutée est nommée « point d'entrée ». Nous allons changer cette ligne en une autre instruction car OllyDbg dispose en interne d'un moteur d'assemblage. Pour cela double cliquez sur la première ligne (ou sélectionnez la première ligne et faites « espace »). La fenêtre suivante apparaît :

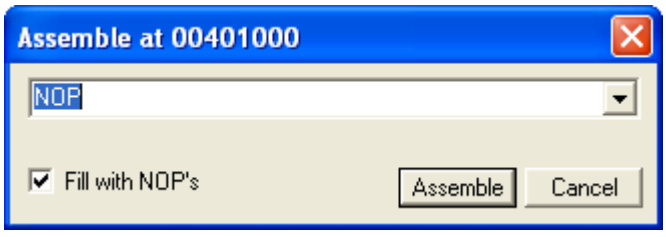

Figure 4 : Assemblage d'une instruction n°1

Nous allons entrer dans cette fenêtre, à la place du NOP surligné, l'instruction « INC EAX » qui **inc**rémente le registre EAX de 1.

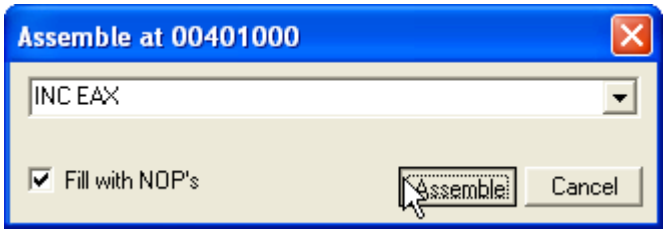

Figure 5 : Assemblage d'une instruction n°2

Nous n'avons plus ensuite qu'à appuyer sur le bouton « *Assemble* ». La fenêtre principale reflète alors le changement :

```
00401000 <>    40     INC     EAX
```

Nous devinons ainsi que l'opcode 0x40 correspond au mnémonique INC et plus précisément à l'instruction INC EAX.

À présent tentons de faire le cheminement inverse qui consistera à écrire l'opcode pour voir le mnémonique de l'instruction. Placez vous sur la première ligne et faite CTRL + E (ou click droit sur la première ligne et *Binary* > *Edit*). La fenêtre suivante apparaît :

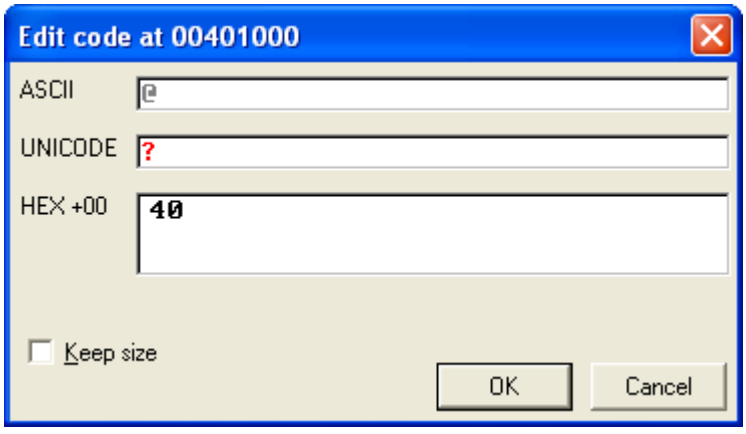

Figure 6 : Édition binaire n°1

Nous voyons l'opcode de l'instruction que nous avions assemblée précédemment. Nous pouvons par exemple le remplacer par 0x90 :

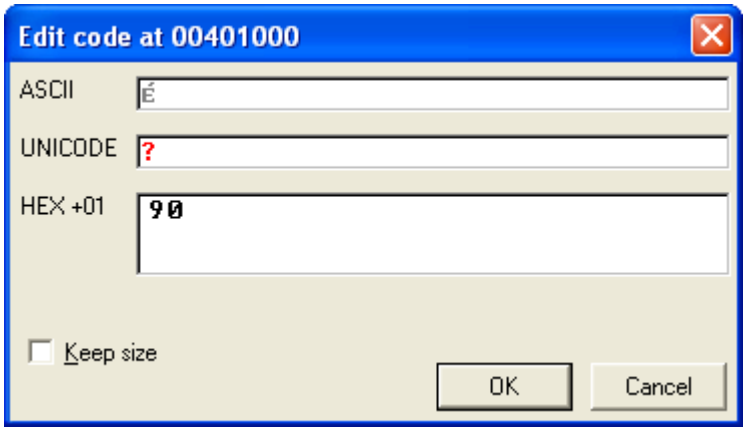

Figure 7 : Édition binaire n°2

En cliquant sur OK, nous voyons l'instruction désassemblée dans la fenêtre principale. L'instruction NOP est revenue.

Amusez-vous à expérimenter, à écrire des instructions et à les assembler ou encore à éditer directement les opcodes pour voir le résultat obtenu !

Nous pouvons aller encore plus loin dans l'étude basique des opcodes. Comme nous l'avons vu il semblerait au premier abord qu'un mnémonique ou une instruction corresponde à un seul opcode et qu'inversement, un opcode corresponde à un seul mnémonique ou instruction. La réalité est toute autre…

Si on essaye d'assembler l'instruction suivante :

```
XCHG EAX, EAX
```

Nous voyons en fait écrit :

```
90 NOP
```

Ce qui nous permet de dire que plusieurs mnémoniques peuvent partager le même opcode. C'est en réalité un comportement normal appelé « *aliasing* ». NOP est en fait un alias pour XCHG EAX, EAX.

Mais, comportement encore plus troublant, des opcodes différents peuvent avoir le même mnémonique et représenter ainsi exactement la même instruction. En voici quelques exemples :

```
03C0          ADD    EAX, EAX
01C0          ADD    EAX, EAX

8000 01       ADD    BYTE PTR [EAX], 1
8200 01       ADD    BYTE PTR [EAX], 1
800420 01     ADD    BYTE PTR [EAX], 1

68 01000000 PUSH   1
6A 01         PUSH   1

8BC0          MOV    EAX, EAX
89C0          MOV    EAX, EAX

8B40 01             MOV    EAX, DWORD PTR [EAX+1]
8B80 01000000       MOV    EAX, DWORD PTR [EAX+1]
```

On notera finalement qu'une instruction peut avoir de 1 à 3 opcodes. Les instructions les plus anciennes (datant des débuts de la famille des processeurs x86) n'ont dans leur grande majorité qu'un seul opcode, tandis que les plus récentes (famille des instructions FPU, 3DNow !, SSE) ont 2 voir 3 opcodes.

ModR/M

Nous venons de voir que l'octet d'opcode nous donnait la signification ou le sens général de l'instruction. Il faut ainsi noter qu'habituellement l'opcode ne renferme que ce sens général et ne comprend pas d'opérandes associés.

L'octet de modR/M (modR/M pour **Mod**e **R**egister / **M**emory), s'il est présent (c'est un octet optionnel), suit directement l'octet d'opcode. Il permet notamment de définir :

- Une référence registre.
- Une référence mémoire.
- Si une forme complexe d'adressage est requise.

Le modR/M sert donc à encoder les registres et / ou les opérandes mémoires utilisés avec une instruction.

Cas de la forme non fixée

Avant de parler d'une forme non fixée, intéressons nous à ce qu'est une forme fixe d'instruction.

Une forme fixe d'instruction est le cas où un (ou deux) opérande (la plupart du temps un registre mais il peut s'agir d'un opérande mémoire) est directement « inscrit » dans l'opcode. Par exemple, l'opcode 0x40 indique qu'il s'agit non seulement de l'instruction INC mais en plus que le registre utilisé est EAX. Il n'y a aucun octet nécessaire à l'encodage d'EAX, l'opcode comprend dans ce cas, le registre utilisé.

```
40          INC EAX
```

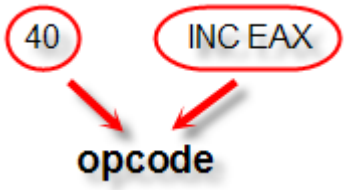

Figure 8 : Forme fixe d'une instruction

La forme non fixée d'une instruction adjoint donc un octet de modR/M à l'instruction, comme c'est le cas pour l'instruction suivante, utilisant deux registres, dont le mnémonique est MOV :

```
8B C3          MOV   EAX, EBX
```

Figure 9 : Instruction et modR/M

C'est l'octet de modR/M 0xC3 qui est responsable de la combinaison EAX en tant que destination et EBX en tant que source. Notons que 0xC3 devient alors une forme « universelle » pour cette combinaison, utilisable avec toutes les autres instructions pouvant utiliser cette combinaison de registres :

```
2B C3          SUB   EAX, EBX
```

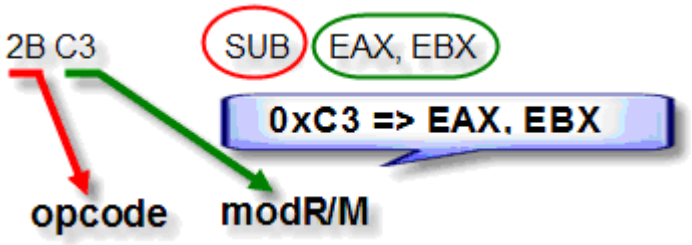

Figure 10 : Le ModR/M est invariable

Cas de l'adressage mémoire

Une instruction utilisant un adressage mémoire peut se composer ainsi :

```
8B 1D 00104000    MOV EBX, DWORD PTR [0x401000]
```

L'exemple ci-dessus veut dire : Place le double mot situé à l'adresse (ou pointé par) 0x401000 dans le registre EBX.

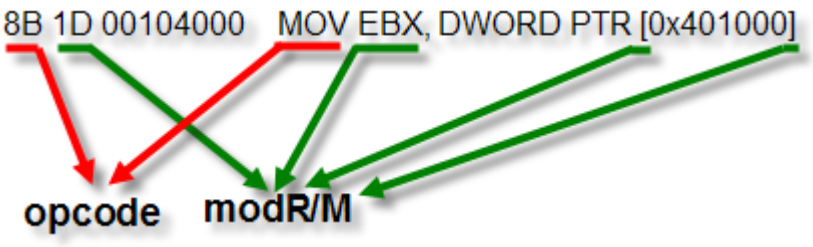

Figure 11 : ModR/M et adressage mémoire

Comme l'instruction utilise un opérande mémoire (ici : [0x401000]), un octet de modR/M est absolument nécessaire. Notez que la valeur de l'adresse en elle-même (0x401000) est donnée par un champ nommé « déplacement » que nous verrons bientôt.

L'octet de ModR/M (dans l'exemple ci-dessus : 0x1D) nous donne donc à la fois le registre de destination (EBX) et le fait qu'un opérande mémoire est utilisé en tant que source.

Notez enfin que le modR/M peut référencer une forme complexe d'adressage, qui nécessite alors un octet supplémentaire, l'octet de SIB.

SIB

L'octet de SIB (octet optionnel) suit directement l'octet de modR/M. Il est à noter que l'octet de SIB ne peut être présent que si et seulement si un octet de modR/M est déjà présent dans l'instruction. Par contre l'octet de SIB, s'il nécessite forcément un octet de modR/M n'est pas forcément présent si un octet de modR/M l'est.

En effet l'octet de SIB (SIB pour **S**cale **I**ndex **B**ase) remplit une fonction particulière dans un cas d'adressage bien précis où le modR/M doit référencer au moins un opérande mémoire.

Cette fonction particulière est de pouvoir permettre un adressage complexe, dont voici un exemple :

```
8B 84 C2 00104000     MOV     EAX, DWORD PTR [EDX+EAX*8+401000]
```

Dans l'exemple ci-dessus, le modR/M indique l'opérande de destination (registre EAX) et le fait que l'opérande source soit un opérande mémoire. Notez qu'ici le modR/M n'indique pas les valeurs contenues dans l'opérande mémoire, c'est notamment le SIB qui donne ces valeurs ! (EDX+EAX*8)

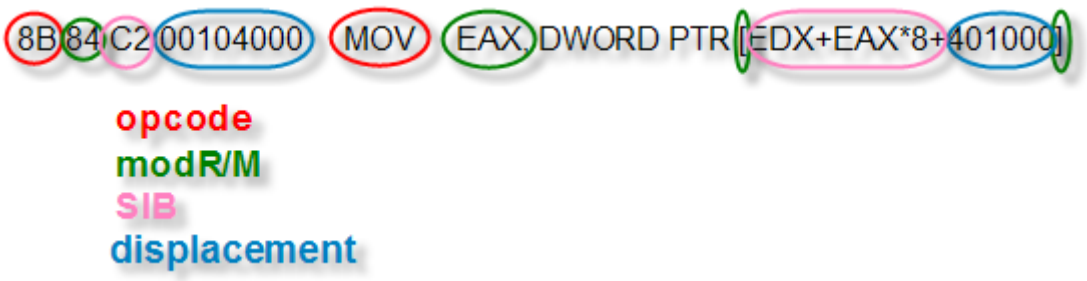

Figure 12 : Instruction et SIB

Dans l'exemple ci-dessus, il est à noter que le modR/M (0x84) dénote à la fois la présence du registre de destination EAX et aussi la présence d'un opérande mémoire. L'octet de modR/M indique que cet opérande mémoire est de la forme SIB + displacement sur 32 bits.

Nous allons voir ce qu'est exactement le displacement.

Displacement

Les octets de *displacement* (déplacement) ou *offset*, au nombre de 1, 2 ou 4 octets, sont ajoutés à l'instruction si celle ci comprend déjà un octet de modR/M. La place exacte de ces octets vient après le modR/M (s'il n'y a pas d'octet de SIB) ou le SIB si ce dernier est présent.

Le déplacement induit par ces octets d'offset est un déplacement signé (il peut être négatif) relatif soit au segment utilisé, soit au pointeur d'instruction (EIP ou RIP) suivant le mode d'adressage et n'est utilisé que pour les opérandes mémoire. En voici deux exemples :

```
8B 1D 00104000     MOV EBX, DWORD PTR [0x401000]
```

```
8B 84 C2 00104000     MOV EAX, DWORD PTR [EDX+EAX*8+401000]
```

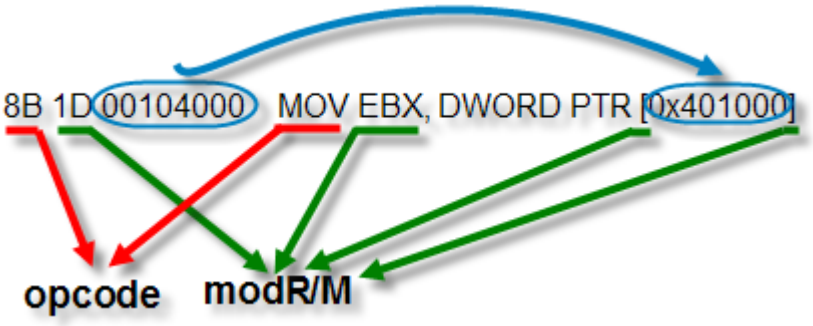

Figure 13 : Instruction et déplacement

Immediates

Les valeurs immédiates, optionnelles, sont encodées directement dans l'instruction, tout à la fin de celle-ci, après le modR/M, le SIB et le déplacement si l'un d'eux ou une combinaison de ceux-ci est présente.

Le nombre d'octets composant une valeur immédiate est de 1, 2, 4 ou 8 (la forme à 8 octets est disponible seulement en mode 64 bits).

```
83 C1 03              ADD     ECX, 3
C784C2 00104000 F1DEBC0A      MOV     DWORD PTR [ EDX+EAX*8+ 0x401000 ], 0x0ABCDEF1
```

Pour finir, voilà, avec une légende colorisée, le détail de l'instruction relativement complexe vue ci-dessus :

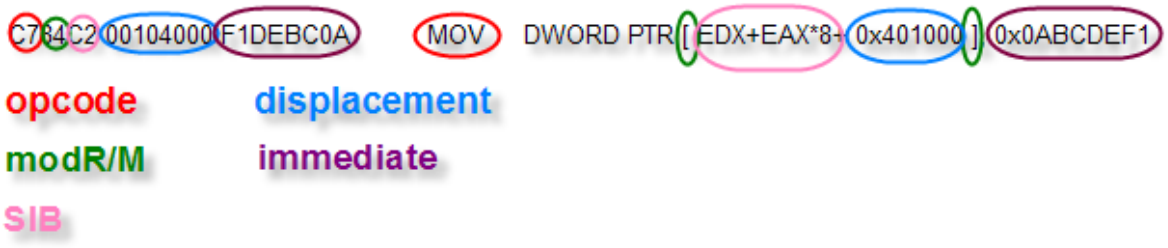

Figure 14 : Instruction et valeur immédiate

Endianness des processeurs x86

L'endianness d'un processeur (ou d'un système) est l'ordre dans lequel il lit certaines séquences de données. Les processeurs de la famille x86 ont une endianness dite « *little* » (petite endianness) ce qui implique que le processeur lit certaines données où les octets sont rangés suivant leur poids, le poids le plus fort venant en dernier et le poids le plus faible venant en premier.

Par exemple dans l'instruction suivante que nous avons déjà vue :

```
C784C2 00104000 F1DEBC0A        MOV     DWORD PTR [ EDX+EAX*8+ 0x401000 ], 0x0ABCDEF1
```

On voit que les octets de *displacement* et *d'immediate* semblent inversés par rapport à la lecture du code désassemblé.

```
00104000 => 0x00401000
F1DEBC0A => 0x0ABCDEF1
```

Pour le décodage des instructions, seul les données de types *immediate* et *displacement* sont inversées, car le reste des octets est lu octet par octet, il n'y a donc pas lieu de voir un inversement quelconque dans le flot des octets.

En effet, seules les données regroupées par paquets d'octets sont sujettes à l'endianness, que ce soient les mots (WORD), double mots (DWORD) ou quadruples mots (QUADWORD), etc.

Comme moyen mnémotechnique concernant l'endianness, rappelez-vous simplement que dans le système dit « *little endianness* », les petits (*little* / faibles) poids viennent en premier.

Fin de la première partie

Il reste sans aucun doute beaucoup de choses qui peuvent vous sembler encore obscures même à la lueur des quelques détails entraperçus jusqu'ici.

Cette première partie n'était qu'un avant goût des chapitres à venir dans lesquels nous verrons beaucoup plus en détails chacun des groupes constituant une instruction, ainsi que les différentes choses à savoir afin de pouvoir programmer correctement un désassembleur linéaire minimal.

N'hésitez pas à me faire parvenir vos questions (même si les prochains chapitres y répondront, je l'espère, dans une certaine mesure) afin que la suite soit la plus exhaustive possible.

Addendum

Vous pouvez télécharger les manuels des fondeurs de processeurs Intel et AMD aux adresses suivantes :

http://www.intel.com/products/processor/manuals/index.htm

http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_739_7044,00.html

(Pour les manuels des processeurs AMD, voir la partie nommée « *Datasheets, Design Guides & Technical Manuals* »).

Notez qu'il y a quelques différences entre les processeurs x86 Intel et AMD, les deux jeux de manuels ne sont donc pas complètement redondants, d'autant plus que les explications abordant un même point sont souvent exposées différemment.

La carte des opcodes (*opcode map*) qui permet de connaître l'opcode allant avec telle ou telle instruction est visible dans l'appendice A, volume 2B des manuels Intel et dans l'appendice A volume 3 des manuels AMD.

Le débogueur OllyDbg est téléchargeable à l'adresse suivante :

http://www.ollydbg.de/

Notez que nous n'utiliserons pas les capacités de déboguage d'OllyDbg mais uniquement son moteur de désassemblage.

Le principe de mise en fonctionnement de ce programme est réellement simple. Une fois téléchargé, dézippez l'archive du programme (aucune installation n'est nécessaire) dans un dossier sur votre disque dur.

Une fois OllyDbg démarré, il suffit d'ouvrir un programme (menu : File > open ou F3) pour voir son code désassemblé.

Remerciements

Je tiens, ici, à remercier particulièrement Alcatîz et Juju_41 pour leur aide et les corrections apportées à cet article. Je tiens aussi à remercier toute l'équipe de FRET ainsi que les lecteurs de la première version de ce document pour leurs commentaires.