



# Jeu d'instruction des processeurs x86

Partie II – les préfixes

Ce document traite spécialement des préfixes composant  
les instructions des processeurs x86.

**Neitsa**  
**1/5/2008**  
**Version 1.0**

## Table des matières

<b>INTRODUCTION</b> .....	<b>3</b>
<b>LES PRÉFIXES D'HÉRITAGE</b> .....	<b>3</b>
<b>PRÉFIXES DU GROUPE 1</b> .....	<b>7</b>
PRÉFIXE LOCK .....	7
PRÉFIXE REPNE / REPNZ .....	9
PRÉFIXE REP / REPE / REPZ .....	10
<i>Généralités sur les préfixes de répétition</i> .....	11
<b>PRÉFIXES DU GROUPE 2</b> .....	<b>11</b>
PRÉFIXES DE SEGMENTS.....	12
<i>Qu'est-ce qu'un segment ?</i> .....	12
<i>Les registres de segment</i> .....	13
<i>Utilisation implicite des registres de segment</i> .....	13
<i>Utilisation forcée des segments mémoire grâce aux préfixes de segment</i> .....	14
<i>Cas du double opérande mémoire</i> .....	15
<i>Cas de la mémoire plate</i> .....	16
<i>Utilité dans le désassemblage</i> .....	17
<i>Cas du mode 64 bits</i> .....	18
PRÉFIXES D'INDICATION DE BRANCHE .....	19
<i>Branche non prise</i> .....	19
<i>Branche prise</i> .....	20
<b>PRÉFIXE DU GROUPE 3</b> .....	<b>22</b>
CAS DU MODE 64 BITS .....	25
CAS DE LA TAILLE D'OPÉRANDE 64 BITS PAR DÉFAUT .....	26
CHOIX DE LA TAILLE D'OPÉRANDE PAR DÉFAUT .....	29
<b>PRÉFIXE DU GROUPE 4</b> .....	<b>30</b>
INSTRUCTIONS UTILISANT IMPLICITEMENT UNE ADRESSE MÉMOIRE OU UN COMPTEUR .....	32
<b>NOTES FINALES SUR LES PRÉFIXES</b> .....	<b>35</b>
PRÉFIXES REQUIS .....	35
PRÉFIXES ET CODE « POUBELLE » .....	35
PRÉFIXES SURNUMÉRAIRES DANS LE CADRE D'UN DÉSASSEMBLEUR .....	37
EXÉCUTABLES DE DÉMONSTRATION .....	37
<b>REMERCIEMENTS</b> .....	<b>37</b>

Aucune reproduction, même partielle, ne peut être faite de ce document et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur

## Introduction

Nous continuons avec ce présent document à disséquer les constituants fondamentaux des instructions pour les processeurs de la famille x86 / x86-64 (x64)


Dans le premier document de cette série nous avons fait un rapide tour d'horizon des différents constituants d'une instruction. Cette deuxième partie explore plus en profondeur les préfixes d'héritage (hérités des modes 16 et 32 bits).

Notez que dans ce document, les instructions sont vues du côté d'un hypothétique désassembleur et non du point de vue habituel d'un programmeur, qui dans l'écrasante majorité des cas, n'a pas à se soucier de la mise en œuvre des instructions et, plus particulièrement pour le cas qui nous intéresse dans ce document, des préfixes.


## Les préfixes d'héritage

Nous aborderons ici l'utilisation générale des préfixes d'héritage, sans entrer plus avant dans les détails qui seront abordés par la suite.

Le tableau ci-dessous fournit un aperçu général du rôle et du caractère de ces mêmes préfixes :

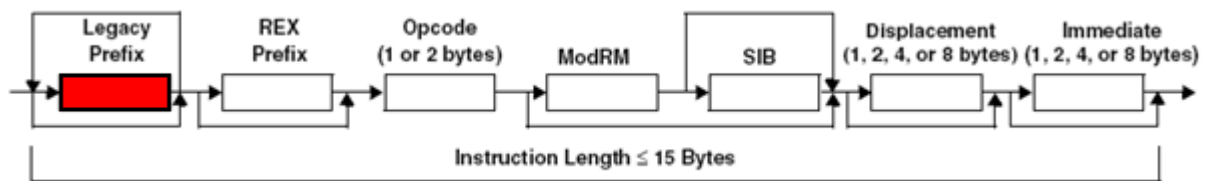
	<b>Rôle</b>	Les préfixes d'héritage permettent de changer le comportement par défaut d'une seule et unique instruction.
	<b>Caractère</b>	Tous les préfixes d'héritage sont optionnels.
	<b>Mode d'opération</b>	Les préfixes d'héritage sont utilisables dans tous les modes d'opération.

L'utilisation de ces préfixes est soumise à des règles et contraintes qui sont détaillées dans le tableau suivant :

Règles et contraintes d'utilisation	
	<ul style="list-style-type: none"> <li>Le ou les octets de préfixes sont les premiers à composer l'instruction.</li> </ul>
	<ul style="list-style-type: none"> <li>Les préfixes d'héritage ont tous une taille d'un octet.</li> </ul>
	<ul style="list-style-type: none"> <li>Il peut y avoir de 0 à 4 préfixes d'héritage (caractère optionnel).</li> </ul>
	<ul style="list-style-type: none"> <li>On ne peut utiliser qu'un seul préfixe par groupe.</li> </ul>

Revenons plus en détail sur les différentes règles énoncées dans le tableau ci-dessus.

Le ou les octets de préfixe (1 préfixe = 1 octet), s'ils sont présents (le préfixe d'héritage est toujours optionnel), sont toujours en première place dans la composition d'une instruction, comme le montre l'image ci-dessous :



En effet, chaque préfixe ayant une valeur bien précise, ils ne peuvent être confondus avec un autre octet, qui aurait alors une signification différente.

Une instruction peut avoir de 0 à 4 préfixes d'héritage. Si une instruction doit avoir plus de 4 préfixes, le résultat de cette instruction est considéré comme imprédictible<sup>1</sup>.

Les préfixes d'héritage sont aussi divisés en différents groupes distincts, réglementant leur utilisation. On ne peut, pour une même instruction, utiliser qu'un seul préfixe d'un même groupe. L'utilisation de plusieurs préfixes d'un même groupe rend le résultat de cette instruction imprédictible.

<sup>1</sup> Dans le meilleur des cas, l'instruction se comporte tout à fait normalement. Dans le pire des cas, le comportement de l'instruction est aberrant ou génère une exception.

Notez que l'ordre des préfixes, si plusieurs sont présents, n'a aucune incidence particulière sur l'instruction.

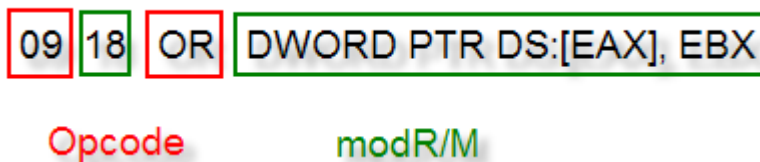
Une dernière note à propos de la convention de nommage utilisée : Les préfixes d'héritage (« *Legacy prefixes* »), comme nous l'avons vu dans le premier document, doivent leur nom au fait qu'ils sont « hérités » des précédents modes d'exécution des processeurs x86 (16 et 32 bits).

Ils sont nommés comme étant « d'héritage » lors de leur utilisation dans un contexte d'exécution 64 bits (de façon à éviter toute confusion avec les préfixes REX – que nous verrons dans un autre document – disponibles uniquement en mode 64 bits).

Dans un contexte d'exécution en 16 ou 32 bits, on les nomme tout simplement « préfixes ».

Avant d'aller plus loin et en guise d'introduction aux différents préfixes d'héritage, voici deux images permettant de voir les modifications d'une instruction avec l'apport de plusieurs préfixes.

L'image ci-dessous nous montre l'instruction suivante : OR DWORD [EAX], EBX, sans aucun préfixe. Les deux seuls constituants de l'instruction étant l'opcode et le modR/M.



L'image ci-dessous nous permet quant à elle de voir les modifications apportées à cette même instruction en y adjoignant quatre préfixes différents :



Ne vous inquiétez pas si l'image ci-dessus vous paraît encore obscure, tout devrait devenir clair au fur et à mesure.


Nous allons maintenant nous attacher à voir les différents groupes de préfixes d'héritage et les préfixes qui les composent.

## Préfixes du Groupe 1

Le groupe 1 des préfixes comprend le préfixe LOCK et les préfixes de répétition :

- 0xF0 — LOCK
- 0xF2 — REPNE / REPNZ
- 0xF3 — REP ou REPE / REPZ

## Préfixe LOCK

	<b>Rôle</b>	<p>Le préfixe LOCK est utilisé pour s'assurer que l'instruction qu'il accompagne possède un accès exclusif à la mémoire dans un environnement multiprocesseur.</p> <p>Ce préfixe n'est bien sûr utile que dans le cas où plusieurs processeurs seraient présents sur le système, mais il peut sans problème être utilisé sur un système monoprocesseur.</p>
	<b>Valeur</b>	0xF0.

Rôle du préfixe LOCK

Lorsque le préfixe LOCK est utilisé, l'opération effectuée par l'instruction dotée de ce préfixe dispose d'un accès privilégié à une ressource en mémoire partagée.

Ainsi, lorsqu'un processeur rencontre un préfixe LOCK attaché à une instruction, un signal est envoyé aux autres processeurs du système disant que le processeur où s'exécute l'instruction avec le préfixe LOCK possède un accès exclusif à la mémoire utilisée dans l'instruction, cette dernière opérant alors de manière atomique.


Prenons un exemple plus parlant avec une instruction disposant d'un préfixe LOCK, comme suit :

FO 011D 00204000 LOCK ADD DWORD PTR [0x402000], EBX

L'instruction ci-dessus ajoute le contenu du registre EBX au double mot en 0x402000. Le préfixe LOCK (émis sur un processeur particulier) envoie un signal vers les autres processeurs disant qu'il possède un accès privilégié sur le double mot situé à l'adresse 0x402000.


Les autres processeurs ne peuvent pas accéder à ce double mot pendant l'exécution de l'instruction.

Le préfixe LOCK a des conditions d'utilisation strictes détaillées dans le tableau suivant :

Règles et contraintes d'utilisation du préfixe LOCK	
	<ul style="list-style-type: none"> <li>Le préfixe LOCK ne peut être utilisé qu'avec les instructions suivantes :                             <ul style="list-style-type: none"> <li>ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCH16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, et XCHG.</li> </ul> </li> </ul>
	<ul style="list-style-type: none"> <li>Lorsqu'une des instructions ci-dessus est utilisée conjointement avec LOCK, l'opérande de destination est <u>obligatoirement</u> un opérande mémoire.</li> </ul>

Règles d'utilisation du préfixe LOCK


Enfin, le préfixe LOCK génère des erreurs si les règles du tableau ci-dessus ne sont pas appliquées :

Erreurs générées par le préfixe LOCK	
	<ul style="list-style-type: none"> <li>Dans le cas où le préfixe LOCK est utilisé avec une autre instruction que celles listées dans le tableau ci-dessus, le processeur génère alors une exception de type #UD (<i>Undefined Opcode</i> : Opcode non défini).</li> </ul>
	<ul style="list-style-type: none"> <li>Même si l'instruction utilisée avec le préfixe LOCK est une des instructions permise, dans le cas où l'opérande de destination ne serait pas opérande mémoire le processeur génère alors une exception de type #UD (<i>Undefined Opcode</i> : Opcode non défini).</li> </ul>

Erreurs générées par le préfixe LOCK




## Préfixe REPNE / REPNZ

	<b>Rôle</b>	Le préfixe REPNE est utilisé en conjonction avec les instructions opérant sur des chaînes de caractères. Il permet de répéter un certain nombre de fois, suivant une certaine condition, l'instruction à laquelle il est adjoint.
	<b>Alias</b>	REPZ est un alias pour REPNE.
	<b>Valeur</b>	0xF2.

Rôle du préfixe REPNE /REPZ

Le préfixe REPNE (pour *Repeat while Not Equal*) ou REPZ (pour *Repeat while Not Zero* - qui est un alias de REPNE) est contraint par les règles d'utilisation suivantes :

	<b>Règles et contraintes d'utilisation du préfixe REPNE</b>	
	<ul style="list-style-type: none"> <li>▪ Le préfixe REPNE (ou son alias) ne peut être utilisé qu'avec les instructions suivantes :                             <ul style="list-style-type: none"> <li>○ INS, MOVS, OUTS, LODS, STOS et SCAS.</li> </ul> </li> <li>▪ Les conditions d'arrêt de la répétition engendrée par le préfixe REPNE (ou son alias) sont listées dans le tableau suivant.</li> <li>▪ L'utilisation superflue des préfixes de répétitions (REP / REPE / REPNE) avec une toute autre instruction parmi celles « permises » rend le comportement de l'instruction indéfini sans toutefois générer d'exception (le préfixe est dit « superflu »).</li> </ul>	

Règles d'utilisation du préfixe REPNE /REPZ

Le préfixe REPNE sert uniquement à répéter l'instruction qu'il précède. Les deux seules conditions d'arrêt possibles de la répétition sont :

Préfixe	Condition d'arrêt n°1	Condition d'arrêt n°2
REPNE / REPZ	(R)CX = 0	ZF = 1

Condition d'arrêt du préfixe REPNE

Lorsqu'une des deux conditions d'arrêt est remplie, la répétition s'arrête.

Voici un exemple d'utilisation du préfixe REPNE qui met à 0 les 16 octets en partant de l'adresse 0x403000 :

32C0	XOR	AL, AL
BF 00304000	MOV	EDI, 0x403000
B9 10000000	MOV	ECX, 0x10
F2:AA	REPNE	STOS BYTE PTR ES:[EDI]

C'est le préfixe lui-même qui permet la décrémentation du registre (R)CX et la vérification de la condition.

REPNE n'est donc, au final, qu'une commodité permettant d'éviter l'utilisation d'une boucle, ce qui permet un code plus compact.

## Préfixe REP / REPE / REPZ

Le préfixe REP / REPE (REPZ est un alias de REPE : *Repeat while Equal*), dont l'octet est 0xF3, fonctionne avec les mêmes instructions que le préfixe REPNE.

Ses conditions d'utilisation sont les mêmes que celles du préfixe REPNE / REPNZ et seules les conditions d'arrêt diffèrent :

Préfixe	Condition d'arrêt n°1	Condition d'arrêt n°2
REP	(R)CX = 0	Aucune
REPE / REPZ	(R)CX = 0	ZF = 0

Condition d'arrêt du préfixe REP / REPE / REPZ

On notera simplement que REP n'a qu'une seule condition d'arrêt.

La forme REP est utilisée avec les instructions INS, OUTS, MOVSB, LODSB et STOS tandis que la forme REPE / REPZ est utilisée avec les instructions CMPS et SCAS.

## Généralités sur les préfixes de répétition

Notons que l'utilisation superflue des préfixes de répétitions (REP / REPE / REPNE) avec une toute autre instruction que celles qui sont « permises » rend le comportement de l'instruction indéfini sans toutefois générer d'exception (le préfixe est dit « superflu »).

Enfin, notons que les octets 0xF2 et 0xF3 sont aussi utilisés pour sélectionner certaines instructions SIMD (MMX ou SSE) quand ils sont utilisés avant l'opcode 0x0F et que dans ce cas précis uniquement, ils n'agissent pas en tant que préfixe mais comme un simple octet constituant l'instruction (qui s'étend alors à deux ou trois octets d'opcodes).

Nous reviendrons sur ces cas lorsque nous aborderons la carte des opcodes en détail.


## Préfixes du groupe 2

Les préfixes du groupe 2 comprennent les préfixes de segments et les *branch hints* (indications de branche).

- 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*).

## Préfixes de segments

	<b>Rôle</b>	Les préfixes de segments sont utilisés avec les instructions utilisant des opérandes mémoires afin de spécifier explicitement quel est le segment à utiliser.
	<b>Valeur</b>	Suivant le segment. Voir tableau ci-dessous.

Rôle des préfixes de segments

Préfixe de segment	Valeur du préfixe
CS	0x2E
SS	0x36
DS	0x3E
ES	0x26
FS	0x64
GS	0x65

Valeurs des préfixes de segments

Les segments sont utilisés avec les instructions utilisant des opérandes mémoires. Avant d'aller plus loin dans l'utilisation des préfixes de segment, voyons tout d'abord, de manière simple, ce qu'est un segment.

### Qu'est-ce qu'un segment ?

D'un point de vue général, le processeur ne « voit » que des séquences d'octets réparties tout au long de la mémoire RAM du système. L'organisation de cette mémoire est un facteur important pour le processeur, de façon à ce qu'il puisse distinguer notamment ce qui est du code de ce qui peut être des données.

Quand une instruction – et incidemment le processeur – nécessite d'accéder à un octet (ou plusieurs) en mémoire, le processeur a besoin d'une adresse pour localiser les données voulues. Tous les endroits adressables – c'est-à-dire accessibles – de la mémoire font partie de l'espace d'adressage global du processeur.

Afin de simplifier l'organisation de ce vaste espace qu'est la mémoire système, les fondateurs de processeurs ont pris le parti de scinder cette dernière en plusieurs tranches. Chacune de ces tranches devient alors à son tour un espace d'adressage propre, ce qui permet à un programme d'avoir

plusieurs espaces d'adressage autonomes et indépendants. Ce sont ces espaces (ou tranches de mémoire) que l'on nomme segments ou, afin d'être plus précis, segments de mémoire.

## Les registres de segment

Les processeurs x86 disposent de 6 registres de segment qui indiquent chacun une « tranche » de mémoire particulière :

Nom du segment	Signification	Nommage anglophone
CS	Segment de code	<i>Code Segment</i>
DS	Segment de donnée	<i>Data Segment</i>
SS	Segment de pile	<i>Stack Segment</i>
ES, FS, GS	Segment de donnée supplémentaire.	<i>Extra Segment</i>

Signification des segments

Une instruction faisant référence à la mémoire via un ou plusieurs de ses opérandes fait donc implicitement ou explicitement référence à une « tranche » mémoire particulière.

Ainsi, lorsqu'on lit l'instruction suivante :

```
8B03    MOV    EAX, DWORD PTR [EBX]
```

Il faut en réalité lire :

```
8B03    MOV    EAX, DWORD PTR DS:[EBX] ; utilisation implicite du registre de segment DS
```

On sait ainsi que le registre EBX est opérande mémoire qui pointe vers le segment de donnée, puisqu'il y a dans cet exemple utilisation du segment DS.

Cet exemple pose toutefois la question de la présence du préfixe indiquant le segment DS qui n'est tout simplement pas présent... (0x8B est l'octet d'opcode et 0x03 est l'octet de modR/M). C'est en fait parce que le processeur utilise un encodage « par défaut » du registre de segment !

## Utilisation implicite des registres de segment

Comme nous venons de le voir, certaines instructions utilisent par défaut un encodage de segment précis (dans l'exemple vu auparavant, le segment DS était utilisé par défaut).

Le tableau ci-après donne les règles d'utilisation implicite des registres de segment dans les instructions.

Type de référence	Registre utilisé	Segment utilisé	Règle de sélection par défaut
<b>Instructions</b>	CS	Segment de code	Toute référence à une instruction ( <i>instruction fetching</i> ).
<b>Pile</b>	SS	Segment de pile	<ul style="list-style-type: none"> <li>• PUSH et POP.</li> <li>• Référence mémoire utilisant (E)SP ou (E)BP comme registre de base.</li> </ul>
<b>Donnée locale</b>	DS	Segment de donnée	Toute référence vers une donnée, sauf si la pile est utilisée ou s'il s'agit de la destination d'une chaîne.
<b>Destination de chaînes</b>	ES	Segment de donnée pointé par le registre ES	Destination des instructions de chaîne.

Le tableau ci-dessus nous indique donc que le registre de segment :

- CS est utilisé implicitement si un opérande mémoire référence du code.
- SS est utilisé implicitement si un opérande mémoire référence la pile.
- DS est utilisé implicitement si un opérande mémoire référence une donnée.
- ES est utilisé implicitement si un opérande mémoire référence la destination d'une instruction de chaîne.

Les autres segments quant à eux ne sont pas présumés être utilisés par défaut.

Quelques exemples d'utilisation implicite des registres de segment :

```

A1 10104000  MOV EAX,DWORD PTR CS:[401010] ; CS implicite si 0x401010 est bien du code.
8B4424 04    MOV EAX, DWORD PTR SS:[ESP+4] ; SS implicite car utilisation du registre de pile ESP.
0305 50030400 ADD EAX,DWORD PTR DS:[40350] ; DS implicite si 0x403050 est bien dans un segment de donnée
AE      SCAS BYTE PTR ES:[EDI] ; ES implicite car [EDI] est la destination de l'instruction SCAS.
    
```

## Utilisation forcée des segments mémoire grâce aux préfixes de segment

L'utilisation forcée des segments fait justement appel aux préfixes dits de « *segment overriding* » (littéralement « forçage de segment »). Si nous reprenons l'exemple précédent où l'utilisation du segment DS était sous-entendue :

**8B03**      MOV    EAX, DWORD PTR **DS**: [EBX] ; utilisation implicite du registre de segment DS

Nous pouvons forcer l'utilisation d'un autre préfixe en adjoignant un préfixe de segment comme ceci :

**2E:8B03**      MOV    EAX, DWORD PTR **CS**: [EBX] ; utilisation explicite du registre de segment CS

Notez que l'utilisation « superflue » et forcée du préfixe DS, bien qu'il soit déjà sous entendu, est possible :

**3E:8B03**      MOV    EAX, DWORD PTR **DS**: [EBX] ; utilisation explicite du registre de segment DS

Notez aussi que dans le cas où une instruction référence (R)SP ou (R)BP explicitement ou implicitement, le forçage vers un autre segment rend l'instruction fautive<sup>2</sup>:

**64:8B0424**      MOV    EAX, DWORD PTR **FS**: [ESP] ; utilisation fautive !

## Cas du double opérande mémoire

Certaines instructions utilisent deux opérandes mémoire et donc deux segments. En voici un exemple :

**A5**      MOVS    DWORD PTR ES: [EDI], DWORD PTR DS: [ESI]

Si un préfixe de segment devait être utilisé avec une telle instruction, seul l'opérande source serait alors affecté par le préfixe de segment :

**64:A5**      MOVS    DWORD PTR ES: [EDI], DWORD PTR **FS**: [ESI]

Notez que seules les instructions CMPS et MOVS utilisent deux opérandes mémoires.

<sup>2</sup> En mode 64 bit, ce type de forçage provoque une faute de type #GP (General protection). En mode 32 ou 16 bits, le comportement de l'instruction est indéterminé.

## Cas de la mémoire plate



Ce chapitre aborde un concept relativement avancé.

Le mode d'adressage dit « protégé » est un mode d'adressage particulier des processeurs x86. L'un de ses avantages est notamment de permettre (c'est une possibilité offerte mais pas obligatoire) un modèle de mémoire dit « plat » (*flat memory model*).

La principale particularité du modèle de mémoire plat est que les segments de mémoire ne partitionnent plus la mémoire en tranches mais au contraire forment un espace mémoire unifié. Cela sous-entend que la base des segments (l'adresse mémoire où commence une tranche de mémoire) et la taille des segments mémoire sont les mêmes pour tous les segments de mémoire.

Pour clarifier le sujet, on peut alors dire que ce qui suit est vrai, en ce qui concerne la base et la taille des segments :

CS = DS = SS = ES

Tous les systèmes d'exploitation reconnus comme « modernes » (Windows 95 et supérieurs, Linux et dérivés, BSD, Solaris, etc.) utilisent ce système d'adressage particulier car il présente de nombreux avantages, le principal étant qu'il n'y a aucune ambiguïté lorsque l'on référence une adresse mémoire et que cette dernière est toujours la même quel que soit le segment utilisé.

Sous un de ces systèmes d'exploitation, l'instruction suivante :

```
A1 00304000  MOV  EAX, DWORD PTR DS:[0x403000]
```

Vaut donc strictement celle-ci :

```
2E :A1 00304000  MOV  EAX, DWORD PTR CS:[0x403000]
```



Puisque DS et CS disposent de la même base (adresse 0), l'adresse 0x403000 est la même dans les deux cas.

Pour simplifier le mécanisme des segments, on peut imaginer qu'ils pointent (grâce au registre de segment) vers des adresses spécifiques et que l'adresse donnée en paramètre dans l'instruction est ajoutée à l'adresse du segment (adresse dite d'« offset »).

Si l'on peut dire que les deux instructions vues ci dessus sont égales c'est parce que CS et DS pointent tous les deux vers l'adresse 0 et que l'adresse donnée (ici 0x403000) est ajoutée. C'est pour cela que l'on dit que la mémoire est plate, parce que la base des segments (l'adresse 0x0) et leur taille sont les mêmes. Ils donnent donc tous accès aux mêmes adresses.

Dans un modèle de mémoire segmenté, la base des segments est différente (leur taille peut être égale ou différente). Ils ne pointent donc pas vers les mêmes adresses, même si l'instruction semble, à la lecture, être la même.

## Utilité dans le désassemblage

On peut alors se poser la question de la nécessité du décodage, notamment dans le cadre de l'implémentation d'un désassembleur, des préfixes de segments si ces derniers ont la même base et la même limite. On peut toutefois dégager plusieurs réponses :

(Notez que les cas 2 à 5 se rapportent à un désassemblage intervenant sur du code 32 bits. Le cas du code 16 bits est abordé dans le premier point et le cas du code 64 bits est abordé dans le chapitre suivant).

1) Le cas du code fonctionnant en 16 bits est bien connu. Les systèmes 16 bits fonctionnent en mode segmenté. Si vous décidez de supporter le désassemblage de code 16 bits, vous devrez nécessairement spécifier les segments dans les instructions.

2) Même si tous les systèmes d'exploitations majeurs utilisent conjointement le mode protégé et la mémoire plate, on peut imaginer l'existence hypothétique d'un système utilisant le mode protégé, mais avec un modèle de mémoire segmenté. Ce cas reste un cas d'école, et un tel système n'existe pas du fait des pénalités engendrées par un tel choix.

3) En réalité, sous les systèmes d'exploitations modernes, on peut considérer que le système de mémoire plate nous donne le résultat suivant : CS = DS = SS. Mais il faut alors noter que les autres segments (les segments de données supplémentaires ES, FS, GS) peuvent pointer ailleurs. D'où la nécessité de supporter la présence, dans le désassemblage, des segments.

4) Les utilisateurs de désassembleurs sont habitués à la lecture des segments qui permet de savoir à quel type de portion de la mémoire se réfère une adresse, même si elles pointent toutes vers les mêmes adresses. L'utilisateur détermine d'un seul coup d'œil si l'instruction se réfère à des données (via la présence du segment DS) ou à la pile (via SS).

5) Certains programmeurs en assembleur spécifient parfois la présence du segment CS pour bien notifier au lecteur que leur code impacte spécifiquement une portion de mémoire contenant du code.

## Cas du mode 64 bits

Le mode 64 bits a introduit un changement dans la manière d'utiliser les segments. En effet, après ce que nous venons de voir, il semble que si la base et la limite des segments CS, DS, SS et ES est la même, l'utilité d'utiliser spécifiquement et délibérément des segments devient nulle.

Le mode 64 bits voit donc la disparition des quatre registres de segment suivants :


- CS, DS, SS, ES

Leurs préfixes sont toujours utilisables mais sont ignorés<sup>3</sup> et il est donc devenu inutile de les décoder (désassembler) en mode 64 bits. Notez toutefois la nécessité de décoder explicitement les préfixes FS et GS qui servent sous certains systèmes, aussi bien en 32 qu'en 64 bits, à accéder à des régions spéciales de la mémoire.

---

<sup>3</sup> L'adjonction d'un préfixe de segment en mode 64 bits ne rend pas l'instruction indéfinie et ne provoque pas de faute particulière, sauf dans le cas d'un forçage autre que le registre de segment SS pour une instruction faisant implicitement ou explicitement référence à la pile.

## Préfixes d'indication de branche

	<b>Rôle</b>	Les préfixes d'indication de branche sont utilisés pour donner une indication au processeur sur la nature d'un saut.
	<b>Valeur</b>	Voir ci-dessous.

Rôle des préfixes d'indication de branchement

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

Les préfixes d'indication de branche (0x2E et 0x3E) utilisent les mêmes octets que les préfixes de segments CS et DS. Il faut ici comprendre « branche » comme « branchement » conditionnel dans le flux de code.

Leur intérêt est de donner une indication (*hint*) au processeur sur la probabilité qu'un branchement se fasse afin que celui-ci adapte au mieux sa vitesse d'exécution via la mise en cache.

Si les préfixes de segments ne sont utilisables qu'avec des instructions utilisant au moins un opérande mémoire, les préfixes de branche ne sont utilisables qu'avec les sauts conditionnels. On ne doit donc les décoder comme tels uniquement si ils accompagnent un Jcc (*Jmp Conditional Code*) tel que les instructions JE, JNZ, JO, etc.

Il est à noter que les préfixes de branches ne sont valables que pour les processeurs Intel de dernière génération. Les processeurs AMD ne semblent pas décoder ces préfixes (ils sont simplement ignorés).

### Branche non prise

Le préfixe de branche non prise est là pour indiquer au processeur la probabilité élevée qu'un saut conditionnel ne soit pas pris.

Imaginons le code suivant :

00401000	A0 00304000	MOV AL, BYTE PTR DS:[0x403000]
00401005	3C 01	CMP AL, 1
00401007	74 F7	JE 0x401000

Si la probabilité que l'octet en 0x403000 soit de 1 est relativement basse on peut gager que le saut aura le même pourcentage de chance d'être pris que la probabilité qu'un 1 apparaisse.

En indiquant au processeur cet état de fait, celui-ci aura tendance à sérialiser les instructions suivant le saut conditionnel (JE) plus facilement dans son cache et donc à les exécuter plus rapidement, évitant les problèmes d'exécution dits « *out of order* ».

Pour ce faire on utilise le préfixe 0x2E :

00401007	<b>2E</b> :74 F7	JE 0x401000 [ <i>Branch Not Taken Hint</i> ]
----------	------------------	--

## Branche prise

Le raisonnement inverse est applicable au préfixe de branche prise. Voyons le code ci-dessous :

00401000	83 C9 FF	OR ECX, 0xFFFFFFFF
00401003	33 C8	XOR ECX, EAX
00401005	49	DEC ECX
00401005	85 C9	TEST ECX, ECX ; ECX = 0 ?
00401007	75 F7	JNZ 0x401000 ; si = 0 alors saute en 0x401000

On voit nettement dans cette boucle qu'à une seule occasion sur plus de 4 milliards, la comparaison est vraie.

On indiquera alors au processeur que dans la majorité des cas la branche conditionnelle sera prise grâce au préfixe 0x3E :

00401007	<b>3E</b> :75 F7	JNZ 0x401000 [ <i>Branch Taken Hint</i> ]
----------	------------------	---

Notez que les préfixes de *Branch Hints* ne sont jamais générés par des compilateurs (et n'existent même pas en *intrinsic*). Seuls les programmeurs assembleurs peuvent les incorporer dans leur code.


Notez aussi qu'aucun désassembleur ne semble prendre en compte correctement, à l'heure actuelle, les préfixes de branchement. Cela est notamment dû au fait que ces préfixes sont relativement récent, qu'ils sont rarement implémentés et que les compilateurs actuels ne les mettent jamais en jeu...

## Préfixe du groupe 3

Le groupe 3 ne comporte qu'un seul préfixe, le préfixe de taille d'opérande, 0x66 :

- Group 3

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

	<b>Rôle</b>	Le préfixe de taille d'opérande influe sur la taille des opérandes d'une instruction. Il permet de choisir entre une taille d'opérande de 32 ou 16 bits, suivant laquelle est la taille par défaut.
	<b>Valeur</b>	0x66

Rôle du préfixe de taille d'opérande

Comme les autres préfixes, celui-ci influe sur l'instruction à laquelle il est adjoint. Sa particularité est d'agir sur la taille des opérandes et sa mise en action est très simple à comprendre.

Le tableau ci-dessus donne un aperçu global de la taille d'opérande suivant le mode d'opération et les cas où le préfixe de taille d'opérande (0x66) doit être utilisé.

Mode d'opération		Taille d'opérande par défaut (en bits)	Taille d'opérande effective (en bits)	Préfixe utilisé ?	
				0x66	REX.W
<b>Mode Long</b>	<b>Mode 64 bits</b>	32 <sup>4</sup>	64	Inutile	Oui
			32	Non	Non
			16	Oui	Non
	<b>Mode compatibilité</b>	32	32	Non	N/A
			16	Oui	
			16	Oui	
<b>Mode d'héritage (Protégé, Réel, Virtuel 8086, etc.)</b>	32	32	Non		
		16	Oui		
	16	32	Oui		
		16	Non		

Sélection de la taille d'opérande

<sup>4</sup> En mode 64 bits la taille d'opérande par défaut est de 32 bits sauf pour les instructions référant le pointeur de pile (RSP) et les instructions de branchement de type NEAR où la taille d'opérande est d'office 64 bits (sans utilisation implicite du préfixe REX.W).

Explications :

- La colonne de gauche donne le mode d'opération.
- La deuxième colonne (en partant de la gauche) donne la taille d'opérande par défaut pour le mode d'opération courant.
- La troisième colonne donne la taille d'opérande en vigueur si un préfixe de taille d'opérande (0x66) est utilisé (quatrième colonne).


Même si le tableau semble à première vue complexe, sa lecture en est relativement aisée. En voici un exemple :

- Sur un processeur 64 bits avec un système d'exploitation 64 bits :  
Mode d'opération → Mode Long et 64 bits. La taille d'opérande par défaut est de 32 bits. Si on veut utiliser des opérandes 16 bits, il faut obligatoirement un préfixe de taille d'opérande.

Notez que le passage en 32 bits (processeur 32 bits mais système d'exploitation 16 bits) pour une taille d'opérande par défaut de 16 bits est possible via un préfixe de taille d'opérande, mais cela est dépendant du système d'exploitation.

Notez aussi que le passage de la taille des opérandes en 64 bits (processeur et système d'exploitation 64 bits), n'est pas dépendant du préfixe de taille d'opérande, mais d'un préfixe « spécial » nommé REX.W que nous verrons en détail dans un prochain document.

Le préfixe de taille d'opérande est restreint par les quelques règles suivantes :

<b>Règles et contraintes d'utilisation du préfixe de taille d'opérande</b>	
	<ul style="list-style-type: none"> <li>▪ Le préfixe de taille d'opérande n'influe que sur : <ul style="list-style-type: none"> <li>○ Les opérandes ne référant pas la mémoire.</li> </ul> </li> <li>▪ Le préfixe de taille d'opérande ne peut être utilisé qu'avec : <ul style="list-style-type: none"> <li>○ Les instructions généralistes de L'ALU.</li> <li>○ Les instructions à virgule flottante (FPU / x87) suivantes : FLDENV, FNSTENV, FNSAVE et FRSTOR.</li> </ul> </li> </ul>

- Toute utilisation de ce préfixe avec un autre type d'instruction (MMX / SSE / SSE2 / SSE3 / SSSE3 / SSE4) rend le comportement de l'instruction indéfini.<sup>5</sup>
- Le préfixe de taille d'opérande est rendu inopérant par le préfixe 64 bits REX.W.

## Règles d'utilisation du préfixe de taille d'opérande

Prenons un exemple d'une instruction (en mode compatibilité ou d'héritage 32 bits ou en mode 64 bits, puisque la taille d'opérande est toujours 32 bits par défaut) avec deux opérandes, sans ce préfixe :

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

Ajoutons le préfixe de taille d'opérande :

```
66:8B C3    MOV    AX, BX
```

L'exemple ci-dessus nous montre que les deux opérandes sont passés d'une taille de 32 bits à une taille de 16 bits !

Il est important de comprendre que ce préfixe n'influe que (et uniquement !) sur les opérandes qui ne référencent pas la mémoire. Il n'influe en aucune manière sur la taille des pointeurs (en tant qu'opérande mémoire).

Ci-dessous un exemple avec EAX en opérande destination et un opérande mémoire en source :

```
8B 03    MOV    EAX, DWORD PTR DS:[EBX]
```

<sup>5</sup> Les instructions 3DNow! des processeurs AMD manipulant les entiers (les instructions 3DNow! à virgule flottante n'entrent pas dans ce cas) acceptent un préfixe de taille d'opérande. Celui-ci permet alors à l'instruction de manipuler des registres 128 bits (XMM) plutôt que 64 bits (MMX). Notez que ce comportement est un cas exceptionnel.



On y applique le préfixe de taille d'opérande :

```
66:8B 03    MOV    AX, WORD PTR DS:[EBX]
```

Dans l'exemple ci-dessus le préfixe influe uniquement sur EAX (qui devient AX) et sur la taille du pointeur qui de DWORD passe à WORD (il est en effet impossible de mettre les 32 bits pointés par EBX dans AX qui ne fait que 16 bits).

Souvenez vous qu'en aucun cas EBX ne devient BX car EBX est dans cet exemple un opérande mémoire !

Notez que le changement de taille d'un registre s'accompagne forcément d'un changement de taille des valeurs immédiates :

```
B8 78563412  MOV    EAX, 0x12345678
66:B8 7856    MOV    AX, 0x5678
```

Dans l'exemple ci-dessus, le registre EAX peut contenir jusqu'à 32 bits, mais l'adjonction d'un préfixe de taille d'opérande le « transformant » en registre AX, celui-ci ne peut contenir qu'un maximum de 16 bits.

Notez finalement que l'*endianess* des processeurs est particulièrement visible dans cet exemple. Même si la valeur immédiate commence par les mêmes octets (0x7856...) dans les deux cas, le décodage qui est fait de ces mêmes valeurs immédiates est totalement différent, ceci étant dû à la taille des opérandes (32 bits dans le premier cas et 16 bits dans le second).

## Cas du mode 64 bits

Comme nous l'avons vu dans le premier document de cette série (chapitre « Préfixe REX »), le mode 64 bits ne fait pas usage par défaut des registres généraux étendus à 64 bits (tel que EAX à RAX) ni des registres supplémentaires (R8 à R15 par exemple).

Le code suivant est tout aussi valable pour un processeur fonctionnant en mode 64 bits ou 32 bits :

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

Pour utiliser pleinement les registres étendus, l'adjonction d'un préfixe REX - dans l'exemple, l'octet 0x48 est le préfixe REX.W - est obligatoire :

```
48 8B C3   MOV    RAX, RBX
```

Il faut noter que l'ajout du préfixe de taille d'opérande est complètement ignoré si un préfixe REX.W est présent dans l'instruction :

```
66 48 8B C3      MOV    RAX, RBX      ; préfixe de taille d'opérande ignoré !
```

Notez qu'il n'y a pas d'effet de bord à l'instruction ci-dessus. Le préfixe 0x66 est ignoré et le comportement de l'instruction reste entièrement défini.

## Cas de la taille d'opérande 64 bits par défaut



Ce chapitre aborde un concept relativement avancé.

Le tableau que nous avons vu précédemment montre que les instructions en mode long ont une taille par défaut de 32 bits. Nous avons aussi vu dans divers exemples que la taille d'opérande 64 bits nécessite un préfixe nommé REX.W dont l'octet est 0x48.

Afin d'être complet, il faut noter que certaines instructions ont toutefois une taille d'opérande par défaut de 64 bits. Ces instructions sont :

- Les instructions référant implicitement le pointeur de pile RSP.
- Les instructions de branchement de type « proche » (NEAR)<sup>6</sup>.

Voyons un exemple en détail (code s'exécutant en mode Long 64 bits) :

B8 01 00 00 00	mov	eax, 1	; pas besoin de préfixe de taille d'opérande
48 B8 01 00 00 00 00 00 00	mov	rax, 1	; promotion 64 bits par préfixe REX
50	push	rax	; par défaut en 64 bits ! (aucun préfixe)
66 50	push	ax	; préfixe de taille d'opérande = opérande 16 bits

On voit dans l'exemple ci-dessus que le passage du registre EAX à RAX dans l'instruction MOV nécessite un préfixe REX.W (0x48).

Toutefois l'instruction PUSH n'utilise pas de préfixe REX pour être en 64 bits, parce que PUSH utilise implicitement le registre RSP !

Notez en plus que le préfixe de taille d'opérande (0x66) fait passer directement l'opérande en 16 bits.

On peut tirer une règle de ce constat :

- L'adjonction d'un préfixe de taille d'opérande pour une instruction étant par défaut en 64 bits donne à l'opérande une taille de 16 bits.

Notez ainsi qu'il est impossible pour une instruction ayant par défaut une taille d'opérande de 64 bits d'avoir un opérande de 32 bits.

Voici une liste des instructions référant implicitement le pointeur de pile :


Mnémonique	Opcodé (hex)	Description
ENTER	C8	Crée un cadre de pile pour procédure.
LEAVE	C9	Efface un cadre de pile de procédure.
POP reg/mem	8F/0	« Pop » de la pile (registre ou mémoire).
POP reg	58-5F	« Pop » de la pile (registre).
POP FS	0F A1	« Pop » de la pile dans le registre de segment FS.

<sup>6</sup> Ceci n'est valable que (et uniquement) pour les processeurs AMD 64 bits. Sur les processeurs Intel 64 bits, les instructions de branchements ne sont absolument pas affectées par les préfixes de taille d'opérande.

C'est une des disparités majeures entre les implémentations 64 bits des deux fondeurs de processeurs.

Mnémonique	Opcode (hex)	Description
<b>POP GS</b>	0F A9	« Pop » de la pile dans le registre de segment GS.
<b>POPF, POPFD, POPFQ</b>	9D	« Pop » de la pile un mot, double-mot ou quadruple-mot vers le registre (R)FLAGS.
<b>PUSH imm32</b>	68	Pousse sur la pile (double mot à signe étendu).
<b>PUSH imm8</b>	6A	Pousse sur la pile (octet à signe étendu).
<b>PUSH reg/mem</b>	FF/6	Pousse sur la pile (registre ou mémoire).
<b>PUSH reg</b>	50-57	Pousse sur la pile (registre).
<b>PUSH FS</b>	0F A0	Pousse le registre de segment FS sur la pile.
<b>PUSH GS</b>	0F A8	Pousse le registre de segment GS sur la pile.
<b>PUSHF, PUSHFD, PUSHFQ</b>	9C	Pousse un mot, double-mot ou quadruple-mot du registre (R)FLAG sur la pile.

Instructions référençant implicitement le pointeur de pile



Attention, ce qui suit ne concerne que les processeurs AMD 64 bits.

Ci-dessous le tableau référençant les instructions de branchement ayant une taille d'opérande par défaut de 64 bits :

Mnémonique	Opcode (hex)	Description
<b>CALL</b>	E8, FF/2	Appel de procédure proche.
<b>Jcc</b>	70 à 7F, 0F80 à 0F8F	Saut conditionnel court et proche.
<b>JMP</b>	E9, EB, FF/4	Saut inconditionnel court et proche
<b>LOOP</b>	E2	Boucle
<b>LOOPcc</b>	E0, E1	Boucle conditionnelle
<b>RET</b>	C3, C2	Retour d'appel (proche)
<b>JCXZ, JECXZ, JRCXZ</b>	E3	Saut si CX, ECX ou RCX vaut zéro.

Instructions de branchement

Notez qu'une instruction de branchement est dite de type « proche » (*near*) si la cible de l'instruction reste confinée au même segment de code.

Voici un exemple avec une instruction de branchement :

FF E0	jmp rax	; opérande 64 bits par défaut
<b>66</b> FF E0	jmp <b>ax</b>	; opérande 16 bits grâce au préfixe de taille d'opérande

On voit ici que l'instruction utilise une taille d'opérande par défaut de 64 bits (sans préfixe REX.W, 0x48). L'adjonction d'un préfixe de taille d'opérande passe directement la taille d'opérande en 16 bits. Il est impossible d'encoder une taille d'opérande de 32 bits.

## Choix de la taille d'opérande par défaut



Ce chapitre aborde un concept relativement avancé.

Dans les exemples que nous avons vus jusqu'à maintenant, la taille d'opérande par défaut semble être toujours de 32 bits, c'est-à-dire que sans préfixe ce sont les registres ayant une taille de 32 bits qui sont utilisés.

Le choix de la taille par défaut est dû au système d'exploitation, mais c'est en réalité le processeur qui contrôle ce comportement.

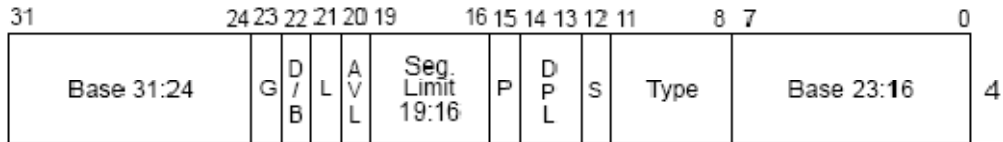
Le choix de la taille d'opérande par défaut est en effet dû au segment !

Plus exactement cela est dû au descripteur de segment (*segment descriptor*) et à son drapeau D (*D flag*, bit 22 – D voulant dire « défaut ») lorsque le processeur est un processeur 32 bits ou est en mode héritage (« *legacy* ») pour un processeur 64 bits<sup>7</sup>.

Si le processeur est en mode long<sup>8</sup> (« *long mode* », uniquement pour processeur 64 bits) ça n'est plus le drapeau D mais le drapeau L (bit 21) qui est responsable de la taille d'opérande.

<sup>7</sup> Le mode héritage est activé si un système d'exploitation 16 ou 32 bits s'exécute sur un processeur 64 bits [cf. « Décodage du jeu d'instruction des processeurs x86 - Partie I » chap. « Mode d'opération »].

<sup>8</sup> Le mode long est activé si un système d'exploitation 64 bits s'exécute sur un processeur 64 bits. [voir confer supra.]




Second double mot d'un descripteur de segment

Vous n'avez heureusement pas à vous inquiéter de ces particularités, le système d'exploitation gère tout seul la taille d'opérande.

## Préfixe du groupe 4

Le groupe 4 ne comporte qu'un seul préfixe, le préfixe de taille d'adresse, 0x67 :

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

	<b>Rôle</b>	Le préfixe de taille d'adresse influe sur la taille d'un opérande mémoire d'une instruction. Il permet de basculer entre une taille d'adresse de 32 ou 16 bits, suivant la taille par défaut.
	<b>Valeur</b>	0x67

Rôle du préfixe de taille d'adresse

Comme les autres préfixes, celui-ci influe sur l'instruction à laquelle il est adjoint. Sa particularité est d'agir sur uniquement sur la taille des adresses (opérande mémoire) d'une instruction. Notez que ce préfixe est n'est quasiment jamais utilisé en mode protégé ou compatibilité 32 bits.

La taille d'adresse par défaut étant dictée par le mode d'opération en cours, le tableau ci-dessous indique la mise en action du préfixe de taille d'adresse.

- La colonne de gauche indique le mode d'opération.
- La deuxième colonne (en partant de la gauche) indique la taille d'adresse par défaut dans le mode approprié.
- La troisième colonne indique la taille effective si un préfixe de taille d'adresse est présent ou non (quatrième colonne).

Mode d'opération		Taille d'adresse par défaut (en bits)	Taille effective (en bits)	Préfixe de taille d'adresse (0x67)
Mode Long	Mode 64 bits	64	64	Non
			32	Oui
	Mode compatibilité	32	32	Non
			16	Oui
			16	Oui
			16	Non
Mode d'héritage (Protégé, Virtuel 8086, Réel, etc.)	32	32	Non	
		16	Oui	
	16	32	Oui	
		16	Non	

Sélection de la taille d'adresse

Voici un exemple d'instruction en mode 64 bits :

```
48 8B 03    MOV    RAX, QWORD PTR DS:[RBX]
```

Nous ajoutons un préfixe de taille d'adresse (0x67) à cette même instruction :

```
67 48 8B 03    MOV    RAX, QWORD PTR DS:[EBX]
```

Notez ici que seul l'opérande qui est de type d'adresse est affecté : RBX (registre 64 bits), devient EBX (registre 32 bits). Ni RAX, qui n'est pas un opérande mémoire, ni la taille du pointeur (QWORD : pointeur de quadruple mot) n'en sont affectés.

Remarquez également – à l'aide du tableau – qu'en mode 64 bits, il est impossible d'atteindre une taille d'adresse égale à 16 bits.

Ci dessous la même instruction, mais cette fois ci avec deux opérandes 32 bits (notez simplement la disparition du préfixe de promotion 64 bits, REX.W dont la valeur est 0x48) :

```
8B 03    MOV    EAX, DWORD PTR DS:[EBX]
```

Si on lui adjoint un préfixe de taille d'adresse (0x67) :

```
67:8B 03    MOV    EAX, DWORD PTR DS:[BP+DI]
```


L'adresse a effectivement changé de 32 bits vers 16 bits ; toutefois, si on pouvait logiquement s'attendre à un pointeur sur BX, il n'en est rien !

Sachez simplement que si le préfixe de taille d'opérande permet de passer la taille d'un opérande mémoire vers la taille qui n'est pas celle par défaut – dans la première instruction, de 64 à 32 bits et dans la deuxième de 32 à 16 bits – la correspondance entre les registres (celui sans préfixe et celui avec préfixe) n'est pas le fait du préfixe de taille d'opérande mais le fait de l'octet de modR/M.

Dans les instructions que nous venons de voir, l'octet de modR/M est l'octet 0x03 et c'est grâce à lui que l'on peut établir les relations que nous avons vues :

- [RBX] devient [EBX] si un préfixe de taille d'adresse est présent.
- [EBX] devient [BP+DI] si un préfixe de taille d'adresse est présent.

Nous traiterons spécifiquement de l'octet de modR/M dans un prochain document.

<b>Règles et contraintes d'utilisation du préfixe de taille d'adresse</b>	
	<ul style="list-style-type: none"> <li>▪ Le préfixe de taille d'adresse n'influe que sur : <ul style="list-style-type: none"> <li>○ Les opérandes référant uniquement la mémoire.</li> </ul> </li> <li>▪ Toute utilisation de ce préfixe avec une instruction ne référant pas la mémoire rend le comportement de l'instruction indéfini.</li> <li>▪ Le préfixe de taille d'adresse influe sur les instructions utilisant implicitement une adresse mémoire ou un registre de compteur.</li> </ul>

Règles d'utilisation du préfixe de taille d'adresse

Le chapitre suivant s'attarde sur la dernière règle de ce tableau.

## Instructions utilisant implicitement une adresse mémoire ou un compteur

Il est dit qu'une instruction utilise implicitement un registre ou un pointeur lorsque son utilisation (ou son décodage) ne nécessite pas forcément l'utilisation d'opérandes. Ces derniers existent bel et bien mais sont sous-entendus.

Ainsi, on peut très bien écrire, dans un code source :

MOVSD



Qu'il faut en réalité lire (ici en mode 32 bits) :

```
MOVS  DWORD PTR ES:[EDI], DWORD PTR DS:[ESI]
```

Ou encore (mode 64 bits) :

```
MOVS  DWORD PTR [RDI], DWORD PTR [RSI]
```

Les opérandes sont dit implicites parce qu'ils sont directement « codés » dans l'instruction et ne peuvent être remplacés par d'autres opérandes :

```
A5    MOVS  DWORD PTR ES:[EDI], DWORD PTR DS:[ESI]
```

Dans l'exemple ci-dessus en 32 bits, l'opcode de l'instruction encode non seulement l'instruction elle-même (MOVS) mais aussi la taille des pointeurs (DWORD) ainsi que les deux opérandes mémoire (ESI et EDI) et les segments (ES et DS, qui ne sont pas utilisés en 64 bits).

De plus, ce type d'instruction peut utiliser le registre ECX en tant que compteur si un préfixe de répétition est utilisé :

```
F3:A5  REP  MOVS DWORD PTR ES:[EDI], DWORD PTR DS:[ESI] ; répète tant que ECX != 0
```

Dans l'instruction ci-dessus, le registre ECX est aussi utilisé implicitement.

L'adjonction d'un préfixe de taille d'adresse va automatiquement influencer sur les opérandes mémoire et sur le registre de compteur s'il est utilisé :

```
67:F3:A5  REP  MOVS DWORD PTR ES:[DI], DWORD PTR DS:[SI] ; répète tant que CX != 0
```

Notez ici que le préfixe de taille d'adresse « transforme » les deux opérandes mémoire mais aussi le registre de compteur.

Le tableau ci-dessous dresse la liste des instructions et de leurs opérandes implicites suivant la taille d'adresse. Entre parenthèses, le registre de compteur si un préfixe de répétition est utilisé.

Instruction	Pointeur ou Registre de compteur		
	Taille d'adresse : 16 bits	Taille d'adresse : 32 bits	Taille d'adresse : 64 bits
CMPS, CMPSB, CMPSW, CMPSD, CMPSQ— <b>Compare Strings</b>	SI, DI (CX)	ESI, EDI (ECX)	RSI, RDI (RCX)
INS, INSB, INSW, INSD— <b>Input String</b>	DI (CX)	EDI, ECX	RDI (RCX)
JCXZ, JECXZ, JRCXZ— <b>Saut si CX/ECX/RCX vaut zéro</b>	CX	ECX	RCX
LODS, LODSB, LODSW, LODSD, LODSQ— <b>Load String</b>	SI (CX)	ESI (ECX)	RSI (RCX)
LOOP, LOOPE, LOOPNZ, LOOPNE, LOOPZ— <b>Loop</b>	CX	ECX	RCX
MOVS, MOVSB, MOVSW, MOVSD, MOVSQ— <b>Move String</b>	SI, DI (CX)	ESI, EDI (ECX)	RSI, RDI (RCX)
OUTS, OUTSB, OUTSW, OUTSD— <b>Output String</b>	SI (CX)	ESI (ECX)	RSI (RCX)
REP, REPE, REPNE, REPNZ, REPZ— <b>Repeat Prefixes</b>	CX	ECX	RCX
SCAS, SCASB, SCASW, SCASD, SCASQ— <b>Scan String</b>	DI (CX)	EDI (ECX)	RDI (RCX)
STOS, STOSB, STOSW, STOSD, STOSQ— <b>Store String</b>	DI (CX)	EDI (ECX)	RDI (RCX)
XLAT, XLATB— <b>Table Look-up Translation</b>	BX	EBX	RBX

Instructions et opérandes implicites



Les paragraphes suivants abordent un concept relativement avancé.

Finalement, notez que le choix de la taille d'adresse par défaut est conditionné par les mêmes bits du descripteur de segment que ceux dictant la taille d'opérande par défaut (cf. chapitre « Choix de la taille d'opérande par défaut »).

Dans tous les cas le bit D choisit une taille par défaut (Bit D à 0 : taille d'adresse par défaut de 16 bits, bit D à 1 : taille d'adresse par défaut de 32 bits), sauf en mode Long 64 bits où c'est le bit L (Bit L à 1 : bit D obligatoirement à 0, taille d'adresse 64 bits).

Ainsi, on voit que la taille d'opérande et la taille d'adresse sont très fortement liées.

## Notes finales sur les préfixes

### Préfixes requis

Dans certaines circonstances, les octets normalement dévolus aux préfixes jouent le rôle d'opcode et perdent alors leur rôle de préfixes. Ces octets sont alors nommés « *mandatory prefixes* » (préfixes obligatoires).

Ces octets sont les octets suivants :

- 0x66, 0xF2 et 0xF3.

Ces octets ne sont considérés comme étant des « *mandatory prefixes* » que si (et uniquement si !) ils sont suivis de l'octet 0x0F (ce dernier étant nommé « *escape opcode* »). Ils servent dans ce cas à encoder les opcodes de certaines instructions SIMD (MMX ou SSE).

Lorsqu'ils sont utilisés dans ce cas précis il convient de ne pas appliquer les effets qu'ils ont « en temps normal » en tant que préfixes.

Voici trois exemples d'instructions utilisant ces octets en tant que préfixes requis :

F3 0F 2D C0	CVTSS2SI	EAX, XMM0
F2 0F 5E C1	DIVSD	XMM0, XMM1
66 0F 7D C1	HSUBPD	XMM0, XMM1

### Préfixes et code « poubelle »

Nous avons vu que les octets de préfixes sont contraints par des règles strictes et que leur utilisation en dehors de ces règles entraîne un comportement indéfini de l'instruction voire une faute.

Si dans les règles ce fait est établi, dans la pratique cela est toutefois moins vrai.

Il existe une pratique de programmation assembleur qui a pour but de retarder l'analyse d'un binaire en gênant un désassembleur ou un opérateur humain en adjoignant des préfixes superflus aux instructions (pratique dite de « *junk code* »).

D'après les règles, les instructions devraient avoir un comportement indéterminé, toutefois dans la pratique on constate qu'il n'en est pas toujours ainsi.

Ci-dessous un code relativement simple rempli de code « poubelle » - c'est-à-dire des octets de préfixes - entre chaque instruction (en bleu) :

```
start:
    db 0f2h, 0f3h
    xor eax, eax
    db 67h, 2eh, 36h, 3eh
    xor edx, edx
    jz @saut1
    db 67h, 66h, 0f0h
@saut1:
    mov ecx, 1
    db 3eh, 26h
    inc cx
    db 67h
    mov eax, 4
    db 36h, 3eh, 64h, 65h
    idiv ecx
    db 0f2h, 0f3h
    xchg eax, ebx
    db 66h, 67h, 3eh, 36h, 64h, 65h
    xor eax, eax

    ret
```

Une fois compilé, voici la vue que l'on a de ce code sous un débogueur :

004010C9	. F3:	PREFIX REP:	Superfluous prefix
004010CA	. 33C0	XOR EAX,EAX	
004010CC	. 67:	PREFIX ADDRESSIZE:	Superfluous prefix
004010CD	. 2E:	PREFIX CS:	Superfluous prefix
004010CE	. 36:	PREFIX SS:	Superfluous prefix
004010CF	. 3E:33D2	XOR EDX,EDX	Superfluous prefix
004010D2	~ 74 03	JE SHORT Prefixes.004010D7	
004010D4	. 67	DB 67	CHAR 'g'
004010D5	. 66	DB 66	CHAR 'f'
004010D6	. F0	DB F0	
004010D7	> B9 01000000	MOV ECX,1	
004010DC	. 3E:	PREFIX DS:	Superfluous prefix
004010DD	. 26:66:41	INC CX	Superfluous prefix
004010E0	. 67:B8 04000000	MOV EAX,4	Superfluous prefix
004010E6	. 36:	PREFIX SS:	Superfluous prefix
004010E7	. 3E:	PREFIX DS:	Superfluous prefix
004010E8	. 64:	PREFIX FS:	Superfluous prefix
004010E9	. 65:F7F9	IDIV ECX	Superfluous prefix
004010EC	. F2:	PREFIX REPNE:	Superfluous prefix
004010ED	. F3:	PREFIX REP:	Superfluous prefix
004010EE	. 93	XCHG EAX,EBX	
004010EF	. 66:	PREFIX DATASIZE:	Superfluous prefix
004010F0	. 67:	PREFIX ADDRESSIZE:	Superfluous prefix
004010F1	. 3E:	PREFIX DS:	Superfluous prefix
004010F2	. 36:	PREFIX SS:	Superfluous prefix
004010F3	. 64:	PREFIX FS:	Superfluous prefix
004010F4	. 65:33C0	XOR EAX,EAX	Superfluous prefix
004010F7	. C3	RETN	

Le code devient relativement difficile à lire... Il convient de noter que le code s'exécute sans problème pour certaines instructions, alors que le comportement d'autres instructions est « parfois » complètement indéfini.

Il faut ici simplement retenir que le comportement d'une instruction noté comme étant indéfini dans la documentation peut se révéler défini dans la pratique. Mais cela dépend d'une part de l'implémentation du jeu x86 par le fondeur (Intel ou AMD) mais aussi du type de processeur, de son *stepping* ou même de sa série.

## Préfixes surnuméraires dans le cadre d'un désassembleur

Dans le cadre de l'implémentation d'un désassembleur, les préfixes surnuméraires ou le *junk code* peuvent être problématiques. La règle la plus simple est de s'en tenir à la documentation :

- Rejetez tous les préfixes surnuméraires (une instruction ne peut avoir que 4 préfixes au maximum).
- Rejetez les préfixes ne pouvant s'appliquer à une instruction (par exemple, lorsqu'un préfixe de taille d'adresse est appliqué à une instruction n'ayant aucun opérande mémoire).
- Appliquez uniquement les préfixes applicables.
- Lorsqu'un préfixe n'a pu être appliqué, il est définitivement « perdu ». Ainsi, n'appliquez pas les préfixes n'ayant pas été appliqués sur une autre instruction.

Le rejet d'un préfixe se traduit par une sortie directe de l'octet du préfixe, sans application à l'instruction et séparé de celle-ci :

36:	PREFIX SS:	; Préfixe superflu
3E:	PREFIX DS :	; Préfixe superflu
64:	PREFIX FS:	; Préfixe superflu
65:	PREFIX GS:	; Préfixe superflu
F7F9	IDIV ECX	

## Exécutables de démonstration

Vous trouverez avec ce document des exécutables et leurs code sources :

- Un exécutable donnant quelques informations sur les segments.
- Un exécutable - et son code assembleur commenté - implémentant les préfixes.
- Une démonstration sur les préfixes en C# (*Runtime* .NET 3.0 requis).

## Remerciements

Un grand merci à Alcatiz et Juju\_41 pour les relectures, corrections et conseils.

Merci à tous les membres de la FRET pour leur amitié et leur support indéfectibles.