fasm

flat assembler 1.73
Manuel de programmation

par Tomasz Grysztar

Édition du 4 janvier 2024

Traduction française

By Calendos
Forum https://flatassembler.net/
Texte original prélevé le 9 mai 2024
Traduction v2.2 (23/05/2024 à 8:31)

Table des matières

Chapitre 1 - Introduction

Ce chapitre contient toutes les informations les plus importantes dont vous avez besoin pour commencer à utiliser l'assembleur *flat*. Si vous êtes un programmeur expérimenté en langage assembleur, vous devez lire au moins ce chapitre avant d'utiliser ce compilateur.

1.1 Présentation du compilateur

Flat Assembler est un compilateur de langage d'assemblage rapide pour les processeurs d'architecture x86, qui effectue plusieurs passes afin d'optimiser la taille du code machine généré. Il est auto-compilable et des versions pour différents systèmes d'exploitation sont fournies. Toutes les versions sont conçues pour être utilisées à partir de la ligne de commande du système et leur comportement ne doit pas différer.

1.1.1 Configuration système requise

Toutes les versions nécessitent le processeur 32 bits de l'architecture x86 (au moins 80386), bien qu'elles puissent également produire des programmes pour les processeurs 16 bits de l'architecture x86. La version DOS nécessite un système d'exploitation compatible avec MS DOS 2.0 et un véritable environnement en mode réel ou DPMI. La version Windows nécessite une console Win32 compatible avec la version 3.1.

1.1.2 Exécution du compilateur à partir de la ligne de commande

Pour lancer l'exécution du *Flat assembler* à partir de la ligne de commande, vous devez fournir deux paramètres : le premier doit être le nom du fichier source, le second doit être le nom du fichier destination. Si le deuxième paramètre est omis, le nom du fichier de sortie sera automatiquement constitué. Après avoir affiché de brèves informations sur le nom et la version du programme, le compilateur lit les données du fichier source et les compile. Lorsque la compilation réussit, le compilateur écrit le code généré dans le fichier destination et propose un résumé du processus de compilation ; sinon, il affiche les informations sur la ou les erreur(s) survenue(s).

Dans la ligne de commande, vous pouvez également inclure l'option `-m` suivie d'un nombre, qui spécifie le nombre de kilo-octets de mémoire que l'assembleur doit utiliser au maximum. Dans le cas de la version DOS, cette option limite uniquement l'utilisation de la mémoire étendue. L'option `-p` suivie d'un nombre peut être utilisée pour spécifier la limite du nombre de passes effectuées par l'assembleur. Si le code ne peut pas être généré dans le nombre de passes spécifié, l'assemblage se termine avec un message d'erreur. La valeur maximale de ce paramètre est 65 536, tandis que la limite par défaut, utilisée lorsqu'aucune option de ce type n'est incluse dans la ligne de commande, est de 100.

Le fichier source doit être un fichier texte et peut être créé dans n'importe quel éditeur de texte. Les sauts de ligne sont acceptés dans les standards DOS et Unix. Les tabulations sont traitées comme des espaces.

Il n'y a aucune option de ligne de commande susceptible d'affecter la sortie du compilateur. *Flat assembler* nécessite uniquement que le code source inclue les informations dont il a réellement besoin. Par exemple, vous pouvez spécifier le format de sortie en utilisant la directive `format` au début du script source.

1.1.3 Messages du compilateur

Comme indiqué ci-dessus, après une compilation réussie, le compilateur propose un résumé de la compilation. Il comprend des informations sur le nombre de passes effectuées, le temps nécessaire et le nombre d'octets écrits dans le fichier destination. Voici un exemple de résumé de compilation :

```
flat assembler  version 1.73 (16384 kilobytes memory)
38 passes, 5.3 seconds, 77824 bytes.
```

En cas d'erreur lors du processus de compilation, le programme affiche un message d'erreur. Par exemple, lorsque le compilateur ne trouve pas le fichier d'entrée, il affiche le message suivant :

```
flat assembler  version 1.73 (16384 kilobytes memory)
error: source file not found.
```

Si l'erreur est liée à une partie spécifique du code source, la ligne source à l'origine de l'erreur est affichée. L'emplacement de cette ligne dans le script source est également indiqué pour vous aider à trouver cette erreur, par exemple :

```
flat assembler  version 1.73 (16384 kilobytes memory)
example.asm [3]:
        mob     ax,1
error: illegal instruction.
```

Cela signifie que, dans la troisième ligne du fichier *example.asm*, le compilateur a rencontré une instruction non reconnue. Lorsque la ligne qui a provoqué l'erreur contient une macro-instruction, la ligne dans la définition de la macro-instruction qui a généré l'instruction erronée est également affichée :

```
flat assembler  version 1.73 (16384 kilobytes memory)
example.asm [6]:
        stoschar 7
example.asm [3] stoschar [1]:
        mob     al,char
error: illegal instruction.
```

Cela signifie que la macro-instruction de la sixième ligne du fichier *exemple.asm* a généré une instruction non reconnue avec la première ligne de sa définition.

1.1.4 Formats de sortie

Par défaut, lorsqu'il n'y a pas de directive **FORMAT** dans le script source, l'assembleur met simplement les codes d'instruction générés en sortie, créant ainsi un fichier binaire *flat*. Par défaut, il génère du code 16 bits, mais vous pouvez toujours le transformer en mode 16 bits ou 32 bits en utilisant les directives `use16` ou `use32`. Certains formats de sortie passent en mode 32 bits lorsqu'ils sont sélectionnés – plus d'informations sur les formats que vous pouvez choisir peuvent être trouvées dans le paragraphe 2.4 relatif aux *directives de formatage*.

Tout le code de sortie est toujours dans l'ordre dans lequel il a été saisi dans le fichier source.

1.2 Syntaxe de l'assembleur

Les informations fournies ci-dessous sont principalement destinées aux programmeurs en langage assembleur qui ont déjà utilisé d'autres compilateurs assembleur. Si vous êtes débutant, il vous est conseillé de consulter des didacticiels de programmation en assembleur.

Flat Assembler utilise par défaut la syntaxe Intel pour les instructions d'assemblage, bien que vous puissiez la personnaliser à l'aide des possibilités offertes par le préprocesseur (macro-instructions et constantes symboliques). Il possède également son propre ensemble de directives – les instructions pour le compilateur.

Tous les symboles définis dans les scripts source sont sensibles à la casse des caractères.

1.2.1 Syntaxe des instructions

Les instructions en langage assembleur sont séparées les unes des autres par des sauts de ligne et une instruction est censée se limiter à une ligne de texte. Si une ligne contient un point-virgule, à l'exception des points-virgules à l'intérieur des chaînes entre guillemets, le reste de cette ligne est le commentaire et le compilateur l'ignore en tant que tel. Si une ligne se termine par le caractère \ (éventuellement le point-virgule et un commentaire peuvent la suivre), la ligne suivante est considérée comme sa continuation.

Chaque ligne du script source est une séquence d'éléments, qui peuvent être l'un des trois types suivants. Un type correspond aux caractères symboliques, qui sont des caractères spéciaux et des éléments individuels même lorsqu'ils ne sont pas espacés les uns des autres. Ces symboles peuvent être `+-/*=<>()[]{}:,|&~#‘` . La séquence constituée d'autres caractères, séparés des autres éléments par des espaces ou des symboles, est un symbole. Si le premier caractère du symbole est un guillemet simple ou double, il intègre toute séquence de caractères qui le suit, même les plus spéciaux, dans une chaîne entre guillemets, qui doit se terminer par le même caractère par lequel il a commencé (le caractère guillemet simple ou double) – cependant, s'il y a deux de ces caractères dans une rangée (sans aucun autre caractère entre eux), ils sont intégrés dans la chaîne entre guillemets comme l'un d'entre eux et la chaîne entre guillemets continue alors. Les symboles autres que les caractères symboliques et les chaînes entre guillemets peuvent être utilisés comme noms ; ils sont donc également qualifiés de symboles de nom.

Chaque instruction se compose d'un mnémonique suivi d'un nombre variable d'opérandes, séparés par des virgules. L'opérande peut être un registre, une valeur immédiate ou une donnée adressée en mémoire. Il peut également être précédé d'un opérateur taille pour définir ou remplacer sa taille (tableau 1.1). Vous pouvez trouver les noms des registres disponibles dans le tableau 1.2. Leur taille ne peut pas être remplacée. La valeur immédiate peut être spécifiée par n'importe quelle expression numérique.

Lorsque l'opérande est une donnée en mémoire, l'adresse de cette donnée (également toute expression numérique, mais elle peut contenir des registres) doit être placée entre crochets ou précédée de l'opérateur PTR. Par exemple, l'instruction `mov eax,3` mettra la valeur immédiate 3 dans le registre EAX, l'instruction `mov eax,[7]` mettra la valeur 32 bits de l'adresse 7 dans EAX et l'instruction `mov byte [7],3` mettra la valeur immédiate 3 dans l'octet à l'adresse 7. Elle peut également être écrite sous la forme `mov byte ptr 7,3`. Pour spécifier quel registre de segment doit être utilisé pour l'adressage, le nom du registre de segment suivi de deux points doit être placé juste avant la valeur de l'adresse (à l'intérieur des crochets ou après l'opérateur PTR).

Tableau 1.1 – Taille des opérateurs

Opérateur	Bits	Octets
byte	8	1
word	16	2
dword	32	4
fword	48	6
pword	48	6
qword	64	8
tbyte	80	10

Opérateur	Bits	Octets
tword	80	10
dqword	128	16
xword	128	16
qqword	256	32
yword	256	32
dqqword	512	64
zword	512	64

Tableau 1.2 – Registres

Type	Bits							
Général	8	al	cl	dl	bl	ah	ch	dh	bh
	16	ax	cx	dx	bx	sp	bp	si	di
	32	eax	ecx	edx	ebx	esp	ebp	esi	edi
Segment	16	es	cs	ss	ds	fs	gs		
Contrôle	32	cr0		cr2	cr3	cr4			
Debug	32	dr0	dr1	dr2	dr3			dr6	dr7
FPU	80	st0	st1	st2	st3	st4	st5	st6	st7
MMX	64	mm0	mm1	mm2	mm3	mm4	mm5	mm6	mm7
SSE	128	xmm0	xmm1	xmm2	xmm3	xmm4	xmm5	xmm6	xmm7
AVX	256	ymm0	ymm1	ymm2	ymm3	ymm4	ymm5	ymm6	ymm7
AVX-512	512	zmm0	zmm1	zmm2	zmm3	zmm4	zmm5	zmm6	zmm7
Opmask	64	k0	k1	k2	k3	k4	k5	k6	k7
Bounds	128	bnd0	bnd1	bnd2	bnd3				

1.2.2 Définitions des données

Pour définir des données ou leur réserver un espace, utilisez l'une des directives répertoriées dans le tableau 1.3. La directive de définition des données doit être suivie d'une ou plusieurs expressions numériques, séparées par des virgules. Ces expressions définissent les valeurs des cellules de données selon une taille fonction de la directive utilisée. Par exemple, `db 1,2,3` définira les trois octets de valeur respective 1, 2 et 3.

Les directives **DB** et **DU** acceptent également les valeurs de chaîne entre guillemets de n'importe quelle longueur, qui seront converties en chaîne d'octets lorsque DB est utilisé et en chaîne de mots avec un octet de poids fort mis à zéro lorsque DU est utilisé. Par exemple, `db 'abc'` définira trois octets de valeurs respectives 61, 62 et 63.

La directive **DP** et son synonyme **DF** acceptent les valeurs constituées de deux expressions numériques séparées par deux points : la première valeur sera le mot de poids fort et la deuxième correspondra au double mot de poids faible de la valeur du pointeur lointain. La directive **DD** accepte également des pointeurs constitués de deux valeurs de mot séparées par deux points, et **DT** accepte la valeur de mot et de quadruple-mot séparée par deux points, le quadruple mot étant stocké en premier. La directive DT avec une expression unique comme paramètre n'accepte que les valeurs à virgule flottante et crée des données au format FPU double précision étendue.

N'importe laquelle des directives ci-dessus permet l'utilisation d'un opérateur **DUP** spécial pour faire plusieurs copies de valeurs données. Le nombre de copies doit précéder cet opérateur et la valeur à dupliquer doit suivre – il peut même s'agir d'une chaîne de valeurs séparées par des virgules, mais un tel ensemble de valeurs doit être placé entre parenthèses, dz la même manière que `db 5 dup (1,2)`, qui définit cinq copies de la séquence de deux octets donnée.

La directive **FILE** est une directive spéciale et sa syntaxe est différente. Elle consiste en une chaîne d'octets du fichier suivie du nom du fichier entre guillemets, puis éventuellement d'une expression numérique spécifiant le décalage dans le fichier précédé des deux points, puis – également facultativement – d'une virgule et d'une expression numérique spécifiant le nombre d'octets à inclure (si aucun décompte n'est spécifié, toutes les données jusqu'à la fin du fichier sont incluses). Par exemple, `file 'data.bin'` inclura l'intégralité du fichier sous forme de données binaires tandis que `file 'data.bin':10h, 4` n'inclura que quatre octets commençant au décalage 10h.

La directive de réservation de données doit être suivie d'une seule expression numérique, et cette valeur définit le nombre de cellules de la taille spécifiée qui doivent être réservées. Toutes les directives de définition de données acceptent également le caractère générique ? qui signifie que cette cellule ne doit être initialisée à aucune valeur et l'effet est le même qu'en utilisant la directive de réservation de données. Les données non initialisées peuvent ne pas être incluses dans le fichier de sortie, leurs valeurs doivent donc toujours être considérées comme inconnues.

Tableau 1.3 – Directives de données

Taille (octets)	Définition de donnée	Réservation de donnée
1	db file	rb
2	dw du	rw
4	dd	rd
6	dp df	rp rf
8	dq	rq
10	dt	rt

1.2.3 Constantes et labels

Dans les expressions numériques, vous pouvez également utiliser des constantes ou des labels en lieu et place de nombres. Pour définir la constante ou le label, vous devez utiliser les directives spécifiques. Chaque label ne peut être défini qu'une seule fois et il est accessible depuis n'importe quel endroit du scripy source (avant même sa définition). La constante peut être redéfinie plusieurs fois, mais dans ce cas, elle n'est accessible qu'après avoir été définie et est toujours égale à la valeur de la dernière définition avant l'endroit où elle est utilisée. Lorsqu'une constante n'est définie qu'une seule fois dans le script source, elle est –comme le label –accessible de n'importe où.

La définition de la constante se compose du nom de la constante suivi du caractère = et de l'expression numérique, qui après calcul deviendra la valeur de la constante. Cette valeur est toujours calculée au moment de la définition de la constante. Par exemple, vous pouvez définir la constante `count` en utilisant la directive `count = 17` puis l'utiliser dans des instructions d'assemblage, comme `mov cx, count` – qui deviendra `mov cx, 17` pendant le processus de compilation.

Il existe différentes manières de définir des labels. La plus simple est de faire suivre le nom du label par deux points. Cette directive peut même être suivie de l'autre instruction de la même ligne. Elle définit le label dont la valeur est égale au décalage du point où il est défini. Cette méthode est généralement utilisée pour étiqueter les lieux dans le code. L'autre méthode consiste à faire suivre le nom du label (sans deux-points) par une directive de données. Elle définit le label avec une valeur égale au décalage du début des données définies et est mémorisée en tant que label pour les données avec une taille de cellule telle que spécifiée pour cette directive de données dans le tableau 1.3.

Le label peut être traité comme une constante de valeur égale au décalage du code ou des données étiquetées. Par exemple, lorsque vous définissez des données à l'aide de la directive étiquetée `char db 224`, vous devez utiliser l'instruction `mov bx, char` pour mettre le décalage de ces données dans le registre BX, et vous devez utilisez `mov dl, [char]` (ou `mov dl, ptr char`) pour mettre la valeur de l'octet adressé par le label `char` dans le registre DL. Mais lorsque vous essayez d'assembler `mov ax, [char]`, cela provoquera une erreur, car *fasm* compare les tailles des opérandes, qui devraient être égales. Vous pouvez forcer l'assemblage de cette instruction en utilisant un modificateur de taille tel que `mov ax, word [char]`, mais rappelez-vous que cette instruction lira les deux octets commençant à l'adresse `char`, alors qu'elle a été définie comme un octet.

La dernière manière, et la plus flexible, de définir des labels consiste à utiliser la directive `label`. Cette directive doit être suivie du nom du label, puis éventuellement de l'opérateur `size` (elle peut être précédée de deux points) et enfin –également éventuellement de l'opérateur `at` et de l'expression numérique définissant l'adresse à laquelle ce label doit être défini. Par exemple, `label wchar word at char` définira un nouveau label pour les données 16 bits à l'adresse de `char`. Maintenant, l'instruction `mov ax, [wchar]` sera la même que `mov ax, word [char]` après compilation. Si aucune adresse n'est spécifiée, la directive `label` définit le label à son décalage actuel. Ainsi `mov [wchar], 57568` copiera deux octets tandis que `mov [char], 224` copiera un octet à la même adresse.

Le label dont le nom commence par un point est traité comme un label local et son nom est attaché au nom du dernier label global (dont le nom commence par autre chose qu'un point) pour former le nom complet de ce label. Vous pouvez donc utiliser le nom court (commençant par un point) de ce label n'importe où avant que le label global suivant ne soit défini. Aux autres endroits, vous devez utiliser le nom complet. Les labels commençant par deux points sont l'exception : ils sont comme globaux, mais ils ne deviennent pas le nouveau préfixe des labels locaux.

Le nom `@@` caractérise un label anonyme. Il vous est possible d'en définir plusieurs dans le script source. Le symbole `@b` (ou équivalent `@r`) fait référence au label anonyme en amont le plus proche. Le symbole `@f` fait référence au label anonyme aval le plus proche. Ces symboles spéciaux ne sont pas sensibles à la casse.

1.2.4 Expressions numérique

Dans les exemples ci-dessus, toutes les expressions numériques étaient constituées de simples nombres, constantes ou labels. Mais elles peuvent être plus complexes, utilisant, par exemple, des opérateurs arithmétiques ou logiques pour les calculs au moment de la compilation. Tous ces opérateurs avec leurs valeurs de priorité sont répertoriés dans le tableau 1.4. Les opérations avec une valeur de priorité plus élevée seront calculées en premier, tout en rappelant que vous pouvez bien sûr modifier ce comportement en mettant certaines parties de l'expression entre parenthèses. Les symboles +, -, * et / sont des opérations arithmétiques standards. L'opérateur `mod` calcule le reste d'une division. Les opérateurs `and`, `or`, `xor`, `shl`, `shr`, `bsf`, `bsr` et `not` n'effectuent pas les mêmes opérations binaires que les instructions d'assemblage portant les mêmes noms. `rva` et `plt` sont des opérateurs unaires spéciaux qui effectuent des conversions entre différents types d'adresses. Ils ne peuvent être utilisés qu'avec quelques formats de sortie et leur signification peut varier (voir le paragraphe 2.4 relatif aux directives de formatage).

Les calculs arithmétiques et logiques binaires sont généralement traités comme s'ils opéraient sur des nombres binaires de précision infinie, et l'assembleur signale une erreur de débordement si, en raison de ses limitations, il n'est pas en mesure d'effectuer le calcul requis, ou si le résultat est un nombre trop grand pour s'adapter à la plage signée ou non correspondant à la taille de l'unité de destination.

Les nombres dans l'expression sont traités par défaut comme des nombres décimaux. Les nombres binaires doivent avoir la lettre `b` attachée à la fin. Les nombres octaux doivent se terminer par la lettre `o`. Les nombres hexadécimaux doivent commencer par les caractères `0x` (comme en langage C) ou par le caractère `$` (comme en langage Pascal) ou doivent se terminer par la lettre `h`. La chaîne entre guillemets, lorsqu'elle est rencontrée dans une expression, sera convertie en nombre – le premier caractère deviendra l'octet le moins significatif du nombre.

L'expression numérique utilisée comme valeur d'adresse peut également contenir n'importe lequel des registres généraux utilisés pour l'adressage. Ils peuvent être ajoutés et multipliés par des valeurs appropriées, comme cela est autorisé pour les instructions de l'architecture x86. Les calculs numériques dans la définition d'adresse fonctionnent par défaut avec une taille cible supposée être la même que le nombre de bits actuel du code, même si le codage des instructions générées utilisera une taille différente.

Il existe également des symboles spéciaux qui peuvent être utilisés dans l'expression numérique. Le premier est `$`, qui est toujours égal à la valeur du décalage actuel, tandis que `$$` est égal à l'adresse de base de l'espace d'adressage actuel. L'autre est `%`, qui est le nombre de répétitions actuelles dans les parties de code qui sont répétées à l'aide de certaines directives spéciales (voir paragraphe 2.2) et zéro partout ailleurs. Il existe également le symbole `%t`, qui est toujours égal à l'horodatage actuel.

Toute expression numérique peut également être constituée d'une seule valeur à virgule flottante (*Flat Assembler* n'autorise aucune opération à virgule flottante au moment de la compilation) en notation scientifique. Elle peut se terminer par la lettre `f` pour être reconnue, sinon elle doit contenir au moins un point ou la lettre E. Ainsi, `1.0`, `1E0` et `1f` caractérisent-ils la même valeur à virgule flottante, tandis qu'un simple `1` définit une valeur entière.

Tableau 1.4 – Opérateurs arithmétiques et logiques binaires par priorité

Priorité	Opérateurs
0	`+ -`
1	`* /`
2	`mod`
3	`and or xor`
4	`shl shr`
5	`not`
6	`bsf bsr`
7	`rva plt`

1.2.5 Sauts (Jumps) et Appels (Calls)

L'opérande de toute instruction de saut ou d'appel peut être précédé non seulement de l'opérateur taille, mais également de l'un des opérateurs spécifiant le type de saut : `short`, `near` ou `far`. Par exemple, lorsque l'assembleur est en mode 16 bits, l'instruction `jmp dword [0]` deviendra un saut `far` et lorsque l'assembleur est en mode 32 bits, elle deviendra un saut `near`. Pour forcer la différence de traitement de cette instruction, utilisez les formes `jmp near dword [0]` ou `jmp far dword [0]`.

Lorsque l'opérande de saut `near` est une valeur immédiate, l'assembleur générera si possible la variante la plus courte de cette instruction de saut (mais ne créera pas d'instruction 32 bits en mode 16 bits ni d'instruction 16 bits en mode 32 bits, sauf s'il y a un opérateur de taille le précisant). En spécifiant le type de saut, vous pouvez le forcer à toujours générer une variante longue (par exemple `jmp near 0`) ou à toujours générer une variante courte et se terminer par une erreur lorsque c'est impossible (par exemple, `jmp short 0`).

1.2.6 Définitions de la taille

Lorsque l'instruction utilise un adressage mémoire, la plus petite forme d'instruction est générée par défaut en utilisant le déplacement court si seule la valeur de l'adresse rentre dans la plage. Ceci peut être remplacé en utilisant l'opérateur `word` ou `dword` avant l'adresse entre crochets (ou après l'opérateur `ptr`), ce qui force un déplacement long de taille appropriée. Dans le cas où l'adresse n'est relative à aucun registre, ces opérateurs permettent également de choisir le mode d'adressage absolu approprié.

Les instructions **ADC**, **ADD**, **AND**, **CMP**, **OR**, **SBB**, **SUB** et **XOR** dont le premier opérande est de 16 bits ou 32 bits sont générées par défaut sous une forme raccourcie de 8 bits lorsque le deuxième opérande correspond à une valeur immédiate tenant dans la plage assignée aux valeurs 8 bits signées. Il peut également être remplacé en spécifiant l'opérateur `word` ou `dword` avant la valeur immédiate. Des règles similaires s'appliquent à l'instruction `imul`, le dernier opérande étant une valeur immédiate.

La valeur immédiate en tant qu'opérande pour une instruction **PUSH** sans opérateur de taille est traitée par défaut comme une valeur `word` si l'assembleur est en mode 16 bits et comme une valeur `dword` si l'assembleur est en mode 32 bits, forme plus courte de 8 bits. L'instruction est utilisée si possible, l'opérateur de taille word ou dword force l'instruction push à être générée sous une forme plus longue pour la taille spécifiée. Les mnémoniques **PUSHW** et **PUSHD** forcent l'assembleur à générer du code 16 bits ou 32 bits sans l'obliger à utiliser la forme d'instruction la plus longue.

Chapter 2 Jeux d'instructions

2.1 Les instructions de l'architecture x86

Dans cette section, vous pouvez trouver des informations à la fois sur la syntaxe et la finalité des instructions du langage assembleur. Si vous avez besoin de plus d'informations techniques, consulter le manuel du développeur de logiciels d'architecture Intel (Intel Architecture Software Developer's Manual).

Les instructions d'assemblage se composent du mnémonique (nom de l'instruction) suivi, selon le cas, de zéro à trois opérandes. S'il y a deux opérandes ou plus, le premier est généralement l'opérande de destination et le second est l'opérande source. Chaque opérande peut être un registre, le contenu d'un emplacement mémoire ou une valeur immédiate (voir le paragraphe 1.2 pour plus de détails sur la syntaxe des opérandes). Après la description de chaque instruction se trouvent des exemples de différentes combinaisons d'opérandes, si l'instruction en comporte.

Certaines instructions agissent comme des préfixes et peuvent être suivies par d'autres instructions sur la même ligne, et il peut y avoir plusieurs préfixes dans une ligne. Chaque nom du registre de segments est également un mnémonique de préfixe d'instruction, bien qu'il soit recommandé d'utiliser des modificateurs de segment entre crochets au lieu de ces préfixes.

2.1.1 Instructions de mouvement de données

L'instruction **MOV** transfère un octet, un mot ou un double mot de l'opérande source vers l'opérande destination. Elle peut, au choix, transférer des données entre registres généraux, du registre général à la mémoire ou de la mémoire au registre général, mais ne peut effectuer de déplacement de mémoire à mémoire. Elle peut également transférer une valeur immédiate vers un registre général ou une mémoire, un registre de segments vers un registre général ou une mémoire, un registre général ou une mémoire vers un registre de segments, un registre de contrôle ou de débogage vers un registre général et un registre général vers un registre de contrôle ou de débogage. Le MOV ne peut être assemblé que si la taille de l'opérande source et celle de l'opérande de destination sont les mêmes. Vous trouverez ci-dessous les exemples pour chacune des combinaisons autorisées :

```
mov bx, ax          ; registre général à registre général
mov [char], al      ; registre général à mémoire
mov bl, [char]      ; mémoire à registre général
mov dl, 32          ; valeur immédiate à registre général
mov [char], 32      ; valeur immédiate à mémoire
mov ax, ds          ; registre de segment à registre général
mov [bx], ds        ; registre de segment à mémoire
mov ds, ax          ; registre général à registre de segment
mov ds, [bx]        ; mémoire à registre de segment
mov eax, cr0        ; registre de contrôle à registre général
mov cr3, ebx        ; registre général à registre de contrôle
```

L'instruction **XCHG** permute le contenu de deux opérandes. Elle peut échanger 2 opérandes d'un octet, 2 opérandes d'un mot ou 2 opérandes d'un double-mot. L'ordre des opérandes n'a pas d'importance. Ils peuvent être deux registres généraux, ou un registre général et la mémoire. Par exemple :

```
xchg ax, bx         ; permute 2 registres généraux
xchg al, [char]     ; permute un registre avec la mémoire
```

L'instruction **PUSH** décrémente le pointeur de trame de pile (registre ESP), puis transfère l'opérande vers le haut de la pile pointé par ESP. L'opérande peut être une mémoire, un registre général, un registre de segments ou une valeur immédiate de la taille d'un mot ou d'un double-mot. Si l'opérande est une valeur immédiate et qu'aucune taille n'est spécifiée, elle est traitée par défaut comme une valeur `word` si l'assembleur est en mode 16 bits et comme une valeur `dword` si l'assembleur est en mode 32 bits. Les mnémoniques **PUSHW** et **PUSHD** sont des variantes de cette instruction qui stockent respectivement les valeurs de la taille d'un mot ou d'un double-mot. Si plusieurs opérandes suivent sur la même ligne (séparés uniquement par des espaces et non par des virgules), le compilateur assemblera la chaîne des instructions PUSH avec ces opérandes. Les exemples ci-dessous sont proposés avec des opérandes simples :

```
push ax             ; stocke en pile un registre général
push es             ; stocke en pile un registre de segment
pushw [bx]          ; stocke en pile le contenu d'un espace mémoire
push 1000h          ; stocke en pile une valeur immédiate
```

L'instruction **PUSHA** pousse en pile le contenu des huit registres généraux. Elle n'a pas d'opérande. Il existe deux versions de cette instruction : une 16 bits et une 32 bits. L'assembleur génère automatiquement la version appropriée au mode courant, mais elle peut être remplacée en utilisant les mnémoniques **PUSHAW** ou **PUSHAD** pour

forcer la version 16 ou 32 bits. La version 16 bits de cette instruction pousse les registres généraux sur la pile dans l'ordre suivant : AX, CX, DX, BX, la valeur initiale de SP avant que AX ne soit poussé, BP, SI et DI. La version 32 bits pousse les registres généraux équivalents 32 bits dans le même ordre.

L'instruction **POP** transfère le mot ou le double mot du haut de la pile courante vers l'opérande destination, puis incrémente ESP pour pointer vers le nouveau sommet de la pile. L'opérande peut être une mémoire, un registre général ou un registre de segment. Les mnémoniques **POPW** et **POPD** sont des variantes de cette instruction permettant de forcer respectivement les valeurs de la taille d'un mot ou double-mot. Si plusieurs opérandes séparés par des espaces suivent dans la même ligne, le compilateur assemblera la chaîne des instructions POP avec ces opérandes.

```
pop bx          ; restitue le registre général
pop ds          ; restitue le registre segment
popw [si]       ; restitue la mémoire
```

L'instruction **POPA** restaure les registres enregistrés sur la pile par l'instruction **PUSHA**, à l'exception de la valeur enregistrée de SP (ou ESP), qui est ignorée. Cette instruction n'a pas d'opérande. Pour forcer l'assemblage de la version 16 bits ou 32 bits de cette instruction, utilisez respectivement les mnémoniques **POPAW** ou **POPAD**.

2.1.2 Instructions de conversion de type

Les instructions de conversion de type convertissent les octets en mots, les mots en doubles-mots et les doubles-mots en quadruples-mots. Ces conversions peuvent être effectuées en utilisant l'extension de signe ou l'extension à zéro. L'extension de signe remplit les bits supplémentaires de l'élément le plus grand avec la valeur du bit de signe de l'élément plus petit. L'extension zéro les remplit simplement avec des zéros.

Les instructions **CWD** et **CDQ** doublent respectivement la taille du registre de valeur AX ou EAX et stockent les bits supplémentaires dans le registre DX ou EDX. La conversion se fait avec extension de signe. Ces instructions n'ont pas d'opérande.

L'instruction **CBW** étend le signe de l'octet dans AL dans tout AX, et **CWDE** étend le signe du mot dans AX dans tout EAX. Ces instructions n'ont pas non plus d'opérande.

L'instruction **MOVSX** convertit un octet en un mot ou un double-mot et un mot en un double-mot en utilisant l'extension de signe. L'instruction **MOVZX** fait la même chose, mais il utilise l'extension zéro. L'opérande source peut être un registre général ou un emplacement mémoire, tandis que l'opérande destination doit être un registre général. Par exemple :

```
movsx ax, al            ; registre 8 bits vers registre 16 bits
movsx edx, dl           ; registre 8 bits vers registre 32 bits
movsx eax, ax           ; registre 16 bits vers registre 32 bits
movsx ax, byte [bx]     ; mémoire 8 bits vers registre 16 bits
movsx edx, byte [bx]    ; mémoire 8 bits vers registre 32 bits
movsx eax, word [bx]    ; mémoire 16 bits vers registre 32 bits
```

2.1.3 Instructions arithmétiques binaires

L'instruction **ADD** effectue une addition en ce sens qu'elle remplace l'opérande destination par la somme des opérandes source et destination et définit CF (*Carry Flag*) en cas de débordement. Les opérandes peuvent être des octets, des mots ou des doubles-mots. L'opérande destination peut être un registre général ou une mémoire. L'opérande source peut être un registre général ou une valeur immédiate. Il peut également être une mémoire si l'opérande destination est un registre.

```
add ax, bx        ; additionne le contenu d'un registre à celui d'un autre registre
add ax, [si]      ; additionne le contenu d'un emplacement mémoire à celui d'un registre
add [di], al      ; additionne le contenu d'un registre à celui d'un emplacement mémoire
add al, 48        ; additionne une valeur immédiate au contenu d'un registre
add [char], 48    ; additionne une valeur immédiate au contenu d'un emplacement mémoire
```

L'instruction **ADC** additionne les opérandes et ajoute 1 au résultat si CF est défini. Ce résultat est stocké dans l'opérande destination. Les règles pour les opérandes sont les mêmes que pour l'instruction ADD. Une instruction ADD suivie de plusieurs instructions ADC peut être utilisée pour additionner des nombres de plus de 32 bits.

L'instruction **INC** ajoute 1 à l'opérande sans affecter CF. L'opérande peut être un registre général ou un emplacement mémoire, et sa taille peut être d'un octet, d'un mot ou d'un double-mot.

```
inc ax          ; incrémentation du contenu d'un registre d'une unité
inc byte [bx]   ; incrémentation du contenu d'un espace mémoire d'une unité
```

L'instruction **SUB** soustrait le contenu de l'opérande source à celui de l'opérande destination et remplace l'opérande destination par le résultat obtenu. Si une retenue est nécessaire, le flag CF est mis à 1. Les règles pour les opérandes sont les mêmes que pour l'instruction ADD.

L'instruction **SBB** soustrait le contenu de l'opérande source à celui de l'opérande destination, diminue le résultat

d’une unité si CF est défini et stocke le résultat dans l'opérande destination. Les règles pour les opérandes sont les mêmes que pour l'instruction ADD. Une instruction SUB suivie de plusieurs instructions SBB peut être utilisée pour soustraire des nombres de plus de 32 bits.

L’instruction **DEC** soustrait une unité de l'opérande sans affecter le flag CF. Les règles pour l'opérande sont les mêmes que pour l'instruction INC.

L’instruction **CMP** soustrait l'opérande source de l'opérande de destination. Il met à jour les indicateurs en tant que sous-instruction, mais ne modifie pas les opérandes source et destination. Les règles pour les opérandes sont les mêmes que pour l’instruction SUB.

L’instruction **NEG** soustrait un opérande entier signé de zéro. L'effet de cette instruction est d'inverser le signe de l'opérande du positif au négatif ou du négatif au positif, selon le cas. Les règles pour l'opérande sont les mêmes que pour l'instruction INC.

L’instruction **XADD** échange l'opérande destination avec l'opérande source, puis charge la somme des deux valeurs dans l'opérande destination. L'opérande destination peut être un registre général ou un emplacement mémoire. L'opérande source doit être un registre général.

Toutes les instructions arithmétiques binaires ci-dessus agissent sur les flags SF, ZF, PF et OF. Le flag SF est toujours défini sur la même valeur que le bit de signe du résultat, ZF est défini lorsque tous les bits du résultat sont nuls, PF est défini lorsque huit bits de poids faible du résultat contiennent un nombre pair de bits définis, OF est défini si le résultat est trop grand pour un nombre positif ou trop petit pour qu'un nombre négatif (à l'exclusion du bit de signe) rentre dans l'opérande de destination.

L’instruction **MUL** effectue une multiplication non signée de l'opérande et du registre accumulateur. Si l'opérande est un octet, le processeur le multiplie par le contenu de AL et renvoie le résultat 16 bits à AH et AL. Si l'opérande est un mot, le processeur le multiplie par le contenu de AX et renvoie le résultat 32 bits réparti sur DX et AX. Si l'opérande est un double mot, le processeur le multiplie par le contenu d'EAX et renvoie le résultat 64 bits réparti sur EDX et EAX. L’instruction MUL définit les flags CF et OF lorsque la moitié supérieure du résultat est différente de zéro, sinon ils sont effacés. Les règles pour l'opérande sont les mêmes que pour l'instruction INC.

L’instruction **IMUL** effectue une opération de multiplication signée. Cette instruction offre trois variantes : dans la première, elle n’a qu’un opérande spécifié (le deuxième opérande étant implicitement le contenu de l’accumulateur) et se comporte de la même manière que l'instruction MUL. La seconde comporte 2 opérandes explicites, de sorte que l'opérande destination est multiplié par l'opérande source et le résultat est placé dans l'opérande destination. L'opérande destination doit être un registre général. Il peut s'agir d'un mot ou d'un double-mot. L'opérande source peut être un registre général, un emplacement mémoire ou une valeur immédiate. La troisième forme comporte trois opérandes : l'opérande destination doit être un registre général, de la taille d’un mot ou d’un double mot, l'opérande source peut être un registre général ou un emplacement mémoire, et le troisième opérande doit être une valeur immédiate. L'opérande source est multiplié par la valeur immédiate et le résultat est stocké dans le registre destination. Les trois formes calculent un produit dont la taille est le double de celle des opérandes et définissent CF et OF lorsque la moitié supérieure du résultat est différente de zéro. Notez cependant que les deuxième et troisième formes tronquent le produit à la taille des opérandes. Ainsi, les deuxième et troisième formes peuvent également être utilisées pour les opérandes non signés car, que les opérandes soient signés ou non, la moitié inférieure du produit est la même. Vous trouverez ci-dessous les exemples des 3 formes qui viennent d’être décrites :

```
imul bl           ; produit accumulateur par registre
imul word [si]    ; produit accumulateur par emplacement mémoire
imul bx, cx       ; produit registre par registre
imul bx, [si]     ; produit registre par emplacement mémoire
imul bx, 10       ; produit registre par valeur immédiate
imul ax, bx, 10   ; produit registre par valeur immédiate par registre
imul ax, [si], 10 ; produit emplacement mémoire par valeur immédiate par registre
```

L’instruction **DIV** effectue une division non signée de l'accumulateur par l'opérande. Le dividende (l'accumulateur) est deux fois plus grand que le diviseur (l'opérande), le quotient et le reste ont la même taille que le diviseur. Si le diviseur est un octet, le dividende est extrait du registre AX. Le quotient est stocké dans AL et le reste, dans AH. Si le diviseur est un mot, la moitié supérieure du dividende est extraite de DX, la moitié inférieure du dividende est extraite de AX. Le quotient est stocké dans AX et le reste est stocké dans DX. Si le diviseur est un double-mot, la moitié supérieure du dividende provient d'EDX, la moitié inférieure du dividende provient d'EAX, le quotient est stocké dans EAX et le reste est stocké dans EDX. Les règles pour l'opérande sont les mêmes que pour l'instruction MUL.

L’instruction **IDIV** effectue une division signée de l'accumulateur par l'opérande. Il utilise les mêmes registres que l'instruction DIV et les règles pour l'opérande sont les mêmes.

2.1.4 Instructions arithmétiques décimales

L'arithmétique décimale est effectuée en combinant les instructions arithmétiques binaires (déjà décrites dans la section précédente) avec les instructions arithmétiques décimales. Les instructions arithmétiques décimales sont utilisées pour ajuster les résultats d'une opération arithmétique binaire précédente afin de produire un résultat décimal compacté ou non compacté valide, ou pour ajuster les entrées d'une opération arithmétique binaire ultérieure afin que l'opération produise un résultat décimal compacté ou non compacté valide.

L'instruction **DAA** ajuste le résultat de l'ajout de deux opérandes décimaux compressés valides dans AL. L'instruction DAA doit toujours suivre l'addition de deux paires de nombres décimaux compactés (un chiffre dans chaque demi-octet) pour obtenir une paire de chiffres décimaux compactés valides comme résultats. Le flag CF est défini si une retenue est nécessaire. Cette instruction n'a pas d'opérandes.

L'instruction **DAS** corrige le résultat de la soustraction de 2 nombres BCD compactés présent dans AL sous réserve que le résultat à traiter soit inférieur à 100. Si le résultat à traiter est supérieur ou égal à 100, les flags CF et AF sont mis à 1. L'instruction DAS doit toujours suivre la soustraction entre 2 paires de nombres BCD compactés (c'est-à-dire comportant un chiffre dans chaque demi-octet) en vue d'obtenir une paire de chiffres BCD compactés valides comme résultat. Le flag CF est défini si une retenue est nécessaire. Cette instruction n'a pas d'opérande.

```
mov al,74h      ; au format BCD compacté (soit 116 en décimal)
mov ah,18h      ; au format BCD compacté (soit 24 en décimal)
sub al,ah       ; résultat = 5Ch alors que le résultat attendu en BCD compacté est 92h
das             ; convertit al=5Ch en al=92h (format BCD compacté)
```

L'instruction **AAA** modifie le contenu du registre AL en un nombre décimal non compacté valide et met à zéro les quatre premiers bits. L'instruction AAA doit toujours suivre l'addition de deux opérandes décimaux non compactés dans AL. L'indicateur de retenue CF est défini et AH est incrémenté si une retenue est nécessaire. Cette instruction n'a pas d'opérande.

L'instruction **AAS** modifie le contenu du registre AL en un nombre décimal non compacté valide et met à zéro les quatre premiers bits. L'instruction AAS doit toujours suivre la soustraction d'un opérande décimal non compacté à un autre dans AL. L'indicateur de retenue est activé et AH décrémenté si un retenue est nécessaire. Cette instruction n'a pas d'opérande.

L'instruction **AAM** corrige le résultat de la multiplication de deux nombres BCD valides non compactés. Elle doit toujours suivre la multiplication de deux nombres BCD pour produire un résultat BCD valide. Le chiffre de poids fort est laissé dans AH, le chiffre de poids faible dans AL. La version généralisée de cette instruction permet d'ajuster le contenu de AX pour créer deux chiffres BCD non compactés de n'importe quelle base numérique. La version standard de cette instruction n'a pas d'opérande. Sa version généralisée a un opérande consistant en une valeur immédiate spécifiant la base numérique des chiffres créés.

L'instruction **AAD** modifie le numérateur dans AH et AL pour préparer la division de deux opérandes BCD non compactés valides afin que le quotient produit par la division soit un nombre BCD non compacté valide. AH doit contenir le chiffre de poids fort et AL le chiffre de poids faible. Cette instruction ajuste la valeur et place le résultat dans AL, tandis que AH contiendra zéro. La version généralisée de cette instruction permet l'ajustement de deux chiffres non compactés de n'importe quelle base numérique. Les règles pour l'opérande sont les mêmes que pour l'instruction AAM.

2.1.5 Instructions logiques

L'instruction **NOT** inverse les bits de l'opérande spécifié pour former un complément à un de l'opérande. Cela n'a aucun effet sur les flags. Les règles pour l'opérande sont les mêmes que pour l'instruction INC.

Les instructions **AND**, **OR** et **XOR** effectuent les opérations logiques standard. Elles mettent à jour les flags SF, ZF et PF. Les règles pour les opérandes sont les mêmes que pour l'instruction ADD.

Les instructions **BT**, **BTS**, **BTR** et **BTC** agissent sur un seul bit d'une valeur qui peut être en mémoire ou dans un registre général. L'emplacement du bit est spécifié comme un décalage par rapport à l'extrémité de poids faible de l'opérande. La valeur du décalage est celle du deuxième opérande, il peut s'agir soit d'un octet immédiat, soit d'un registre général. Ces instructions attribuent d'abord la valeur du bit sélectionné à CF. L'instruction BT ne fait rien de plus, BTS met le bit sélectionné à 1, BTR réinitialise le bit sélectionné à 0 et BTC change le bit en son complément. Le premier opérande peut être un mot, un double mot ou un quadruple mot.

```
bt  ax,15          ; teste le bit 15 dans le registre
bts word [bx],15   ; teste et met à 1 le bit 15 du mot en mémoire
btr ax,cx          ; teste et met à 0 le bit dont le rang est donné par le registre cx
btc word [bx],cx   ; teste et complémente le bit du mot en mémoire dont le rang
                   ; est donné par le registre cx
```

Les instructions **BSF** et **BSR** recherchent dans un mot, un double ou quadruple mot le premier bit à 1 et stockent l'index de ce bit dans l'opérande de destination, qui doit être un registre général. La chaîne de bits en cours d'ana-

lyse est spécifiée par l'opérande source. Il peut s'agir soit d'un registre général, soit d'un emplacement mémoire. Le flag ZF est mis à 1 si la chaîne entière est nulle (aucun bit à 1 n'est trouvé) ; sinon, il est mis à zéro. Si aucun bit à 1 n'est trouvé, la valeur du registre destination n'est pas définie. L'instruction BSF effectue sa recherche du bit de poids faible au bit de poids fort (en commençant à partir de l'index binaire zéro). L'instruction BSR scanne du bit de poids fort au bit de poids faible (à partir de l'index binaire 15 d'un mot, de l'index 31 d'un double mot ou de l'index 63 d'un quadruple mot).

```
bsf ax,bx       ; recherché le premier bit à 1 dans ax à partir du bit 0 jusqu'au bit 15.
                ; le rang de ce bit est stocké dans bx.
bsr ax,[si]     ; recherché le premier bit à 1 dans ax à partir du bit 15 jusqu'au bit 0.
                ; le rang de ce bit est stocké dans l'emplacement mémoire pointé par si.
```

L'instruction **SHL** décale l'opérande destination vers la gauche du nombre de bits spécifié dans le deuxième opérande. L'opérande destination peut être un registre général ou une mémoire d'octet, de mot, de double ou quadruple mot. Le deuxième opérande peut être une valeur immédiate ou le contenu du registre CL. Le processeur décale les bits vers la gauche. Le dernier bit sorti est stocké dans CF. Tout bit de poids faible est remplacé par 0 après son décalage. L'instruction SAL est un synonyme de SHL.

```
shl al,1            ; décalage d'un bit à gauche du registre al
shl byte [bx],1     ; décalage d'un bit à gauche de l'octet mémoire pointé par bx
shl ax,cl           ; décalage à gauche du registre ax du nombre de bits spécifié par cl
shl word [bx],cl    ; décalage à gauche du mot en mémoire pointé par bx du nombre de bits
                    ; spécifié par cl
```

Les instructions **SHR** et **SAR** décalent l'opérande destination vers la droite du nombre de bits spécifié dans le deuxième opérande. Les règles pour les opérandes sont les mêmes que pour SHL. L'instruction SHR décale les bits de l'opérande vers la droite. Le dernier bit sorti à droite est stocké dans CF. Le décalage du bit de plus fort poids est remplacé par un zéro. L'instruction sar préserve le signe de l'opérande en décalant les zéros sur le côté gauche si la valeur est positive ou en décalant les 1 si la valeur est négative.

L'instruction **SHLD** décale les bits de l'opérande destination vers la gauche du nombre de bits spécifié dans la troisième opérande, tout en décalant les bits de poids fort de l'opérande source vers l'opérande destination sur la droite. L'opérande source demeure inchangé. L'opérande destination peut être un registre général ou une mémoire de la taille d'un mot, d'un double ou quadruple-mot. L'opérande source doit être un registre général, le troisième opérande peut être une valeur immédiate ou le contenu du registre CL.

```
shld ax, bx, 1      ; décale AX de 1 bit vers la gauche et remplace le bit de poids faible
                    ;   de ce registre par celui de plus fort poids de BX
shld [di], bx, 1    ; décale la mémoire d'un bit vers la gauche et remplace le bit de poids
                    ;   faible de cet emplacement par celui de plus fort poids de BX
shld ax, bx, cl     ; décale AX de CL bits vers la gauche et remplace les bit de poids faible
                    ;   de ce registre par les CL bits de plus fort poids de BX
shld [di], bx, cl   ; décale la mémoire de CL bits vers la gauche et remplace les bit de poids
                    ;   faible de cet emplacement par les CL bits de plus fort poids de BX
```

L'instruction **SHRD** décale les bits de l'opérande destination vers la droite, tout en décalant, à quantité équivalente, les bits de poids faible de l'opérande source sur la gauche de l'opérande destination. L'opérande source demeure inchangé. Les règles pour les opérandes sont les mêmes que pour l'instruction SHLD.

Les instructions **ROL** et **RCL** effectuent une rotation vers la gauche de l'octet, du mot ou double-mot constituant l'opérande destination du nombre de bits spécifié dans le deuxième opérande. Pour chaque rotation spécifiée, le bit de poids fort qui sort de la gauche de l'opérande revient à droite pour devenir le nouveau bit de poids faible. RCL met en outre dans CF chaque bit de poids fort qui sort du côté gauche de l'opérande avant qu'il ne revienne à l'opérande en tant que bit de poids faible lors du prochain cycle de rotation. Les règles pour les opérandes sont les mêmes que pour l'instruction SHL.

Les instructions **ROR** et **RCR** effectuent une rotation vers la droite de l'octet, du mot ou double-mot constituant l'opérande destination du nombre de bits spécifié dans le deuxième opérande. Pour chaque rotation spécifiée, le bit de poids faible qui sort de la droite de l'opérande revient à gauche pour devenir le nouveau bit de poids fort. RCR met en outre dans CF chaque bit de poids faible qui sort du côté droit de l'opérande avant qu'il ne revienne à l'opérande en tant que bit de poids fort lors du prochain cycle de rotation. Les règles pour les opérandes sont les mêmes que pour l'instruction SHL.

L'instruction **TEST** effectue la même action que l'instruction AND, mais ne modifie pas l'opérande destination et ne met à jour que les flags. Les règles pour les opérandes sont les mêmes que pour l'instruction AND.

L'instruction **BSWAP** inverse l'ordre des octets d'un registre général de 32 bits : les bits 0 à 7 sont échangés avec les bits 24 à 31, et les bits 8 à 15 sont échangés avec les bits 16 à 23. Cette instruction est fournie pour convertir des valeurs du format *little-endian* au format *big-endian* et réciproquement.

```
bswap edx       ; permute les octets dans le registre edx
```

2.1.6 – Instructions de transfert de contrôle

L'instruction **JMP** transfère inconditionnellement le contrôle à l'emplacement cible. L'adresse de destination peut être spécifiée directement dans l'instruction ou indirectement via un registre ou une mémoire. La taille acceptable de cette adresse dépend du fait que le saut est *near* ou *far* (il peut être spécifié en précédant l'opérande avec, selon le cas, l'opérateur `near` ou `far`) et si l'instruction est de 16 bits ou 32 bits. L'opérande d'un saut *near* doit être de taille `word` pour une instruction 16 bits ou de taille `dword` pour une instruction 32 bits. L'opérande pour le saut *far* doit être de taille `dword` pour une instruction 16 bits ou de taille `pword` pour une instruction 32 bits. Une instruction JMP directe inclut l'adresse de destination dans le cadre de l'instruction (et peut être précédée de *short*, *near* ou *far*). L'opérande spécifiant l'adresse doit être l'expression numérique pour un saut *near* ou *short*, ou deux expressions numériques séparées par deux points pour un saut *far*. La première spécifie le sélecteur de segment, la seconde, le décalage dans le segment. L'opérateur `pword` peut être utilisé pour forcer l'appel *far* 32 bits et `dword` pour forcer l'appel distant 16 bits. Une instruction `jmp` indirecte obtient l'adresse de destination indirectement via un registre ou une variable de pointeur. L'opérande doit être un registre général ou une mémoire. Voir également le paragraphe 1.2.5 pour plus de détails.

```
jmp 100h          ; saut near direct
jmp 0FFFFh: 0     ; saut far direct
jmp ax            ; saut near indirect
jmp pword [ebx]   ; saut far indirect
```

L'instruction **CALL** transfère le contrôle à la procédure, en sauvegardant sur la pile l'adresse de l'instruction suivant CALL pour une utilisation ultérieure par une instruction RET (de retour). Les règles pour les opérandes sont les mêmes que pour l'instruction JMP, mais le CALL n'a pas de variante *near* de l'instruction directe et donc il n'est pas optimisé.

Les instructions **RET**, **RETN** et **RETF** terminent l'exécution d'une procédure et transfèrent le contrôle au programme qui a initialement appelé la procédure en utilisant l'adresse qui a été stockée sur la pile par l'instruction CALL. RET est l'équivalent de RETN, qui renvoie de la procédure exécutée à l'aide de l'appel proche, tandis que RETF renvoie de la procédure exécutée à l'aide de l'appel distant. Ces instructions par défaut à la taille appropriée d'adresse pour le réglage du code actuel, mais la taille de l'adresse peut être forcée à 16 bits en utilisant les mnémoniques **RETW**, **RETNW** et **RETFW** et à 32 bits en utilisant les mnémoniques **RETD**, **RETND** et **RETF**. Toutes ces instructions peuvent éventuellement spécifier un opérande immédiat, en ajoutant cette constante au pointeur de pile, elles suppriment effectivement tous les arguments que le programme appelant a poussé sur la pile avant l'exécution de l'instruction CALL.

L'instruction **IRET** renvoie le contrôle à une procédure interrompue. Elle en diffère de RET en ce qu'elle "poppe" également les flags en pile à destination du registre des flags. Les flags sont stockés sur la pile par le mécanisme d'interruption. Elle prend par défaut la taille de l'adresse de retour appropriée au code courant, mais IRET peut être forcée d'utiliser une adresse 16 bits ou 32 bits à l'aide des mnémoniques IRETW ou IRETD.

Les instructions de transfert conditionnel sont des sauts qui peuvent transférer ou non le contrôle, en fonction de l'état des flags de la CPU lorsque l'instruction s'exécute. Les mnémoniques pour les sauts conditionnels peuvent être obtenus en attachant le mnémonique de condition (voir tableau 2.1) au mnémonique `j`. Par exemple l'instruction **JC** transférera le contrôle lorsque le flag CF est à 1. Les sauts conditionnels peuvent être courts ou proches, et directs uniquement, et peuvent être optimisés (voir paragraphe 1.2.5). L'opérande doit être une valeur immédiate spécifiant l'adresse cible.

Tableau 2.1 – Conditions

Mnémonique	Condition testée	Description	
o	OF = 1	débordement	overflow
no	OF = 0	pas de débordement	not overflow
c	CF = 1	retenue	carry
b		au dessous de	below
nae		pas au-dessus ni égal à	not above nor equal
nc	CF = 0	ne pas porter	not carry
ae		supérieur ou égal	above or equal
nb		pas en dessous	not below
e	ZF = 1	égal	equal
z		zéro	zero
ne	ZF = 0	non égal	not equal
nz		différent de zéro	not zero
be	CF `ou` ZF = 1	inférieur ou égal	below or equal
na		non au-dessus de	not above
a	CF `ou` ZF = 0	au-dessus de	above
nbe		non au-dessous, ni égal	not below nor equal

Mnémonique	Condition testée	Description	
s	SF = 1	signe	sign
ns	SF = 0	non signe	not sign
p pe	PF = 1	parité	parity
		parité paire	parity even
np po	PF = 0	pas de parité	not parity
		parité impaire	parity odd
l nge	SF xor OF = 1	moins que	less
		non plus grand, ni égal	not greater nor equal
ge nl	SF xor OF = 0	plus grand ou égal	greater or equal
		non inférieur	not less
le ng	(SF xor OF) or ZF = 1	moins que ou égal	less or equal
		non plus grand	not greater
g nle	(SF xor OF) or ZF = 0	plus grand que	greater
		non moins que, ni égal	not less nor equal

Les instructions **LOOP** sont des sauts conditionnels utilisant une valeur placée dans CX (ou ECX) pour spécifier le nombre de répétitions d'une boucle logicielle. Toutes les instructions de boucle décrémentent automatiquement CX (ou ECX) et terminent la boucle (ne transférez pas le contrôle) lorsque CX (ou ECX) est égal à zéro. Il utilise CX ou ECX que le paramètre de code actuel soit 16 bits ou 32 bits, mais il peut nous être forcé CX avec le mnémonique **LOOPW** ou d'utiliser ECX avec le mnémonique **LOOPD**. Les instructions `loope` et `loopz` sont les synonymes de la même instruction, qui agit comme la boucle standard, mais termine également la boucle lorsque le flag ZF est à 1. Les mnémoniques **LOOPEW** et **LOOPZW** les forcent à utiliser le registre CX lorsqu'ils sont en boucle et `loopzd` les forcent à utiliser le registre ECX. Les instructions **LOOPNE** et **LOOPNZ** sont les synonymes des mêmes instructions, qui agissent comme la boucle standard, mais terminent également la boucle lorsque le flag ZF n'est pas défini. Les mnémoniques **LOOPNEW** et **LOOPNZW** les forcent à utiliser le registre CX tandis que **LOOPNZD** et **LOOPNZD** les forcent à utiliser le registre ECX. Chaque instruction de boucle a besoin d'un opérande qui est une valeur immédiate spécifiant l'adresse cible, il ne peut s'agir que d'un saut court (dans la plage de 128 octets en arrière et 127 octets en avant à partir de l'adresse de l'instruction suivant l'instruction LOOP).

L'instruction **JCXZ** se branche sur l'étiquette spécifiée dans l'instruction s'il trouve une valeur de zéro dans CX, `jecxz` fait de même, mais vérifie la valeur d'ECX au lieu de CX. Les règles pour les opérandes sont les mêmes que pour l'instruction de boucle.

L'instruction **INT** active la routine de service d'interruption qui correspond au nombre spécifié comme opérande de l'instruction. Le nombre attaché à INT doit être compris entre 0 et 255. La routine de service d'interruption se terminent par une instruction `iret` qui retourne le contrôle à l'instruction qui suit `int`. Le mnémonique **INT3** code le trap court (un octet) qui invoque l'interruption 3. L'instruction **INT0** invoque l'interruption 4 si le flag OF est à 1.

L'instruction **BOUND** vérifie que la valeur signée contenue dans le registre spécifié se situe dans les limites spécifiées. Une interruption 5 se produit si la valeur contenue dans le registre est inférieure à la borne inférieure ou supérieure à la borne supérieure. Elle a besoin de deux opérandes : le premier spécifie le registre testé, le deuxième doit être l'adresse mémoire pour les deux valeurs limites signées. Les opérandes peuvent être de taille word ou dword.

```
bound ax,[bx]   ; vérifie les limites du mot
bound eax,[esi] ; vérifie les limites du double-mot
```

2.1.7 Instructions E/S

L'instruction **IN** transfère un octet, un mot ou un double-mot d'un port d'entrée vers AL, AX ou EAX. Les ports d'E/S peuvent être adressés soit directement, avec la valeur d'octet immédiate codée en instruction, soit indirectement via le registre DX. L'opérande destination doit être le registre AL, AX ou EAX. L'opérande source doit être une valeur immédiate comprise entre 0 et 255, ou un registre DX.

```
in al,20h       ; extraction en AL d'un octet du port 20h
in ax,dx        ; extraction en AX d'un mot du port dont l'adresse est donnée par DX
```

L'instruction **OUT** transfère un octet, un mot ou un double-mot vers un port de sortie depuis AL, AX ou EAX. Le programme peut spécifier le numéro du port en utilisant les mêmes méthodes que l'instruction `in`. L'opérande destination doit être une valeur immédiate comprise entre 0 et 255, ou un registre DX. L'opérande source doit être le registre AL, AX ou EAX.

```
out 20h,ax      ; récupère un octet du port 20h
out dx,al       ; récupère un mot du port adressé par DX
```

2.1.8 Opérations sur les chaînes

Les opérations de chaîne agissent sur un élément d'une chaîne. Un élément de chaîne peut être un octet, un mot ou un double-mot. Les éléments de chaîne sont adressés par les registres SI et DI (ou ESI et EDI). Après chaque opération de chaîne, SI et/ou DI (ou ESI et/ou EDI) sont automatiquement mis à jour pour pointer vers l'élément suivant de la chaîne. Si DF (flag de direction) est égal à zéro, les registres d'index sont incrémentés, si DF est égal à un, ils sont décrémentés. Le montant de l'incrémentation ou du décrément est de 1, 2 ou 4 selon la taille de l'élément de chaîne. Chaque instruction d'opération de chaîne a des formes courtes qui n'ont pas d'opérande et utilisent SI et/ou DI lorsque le type de code est 16 bits, et ESI et/ou EDI lorsque le type de code est 32 bits. SI et ESI pointent par défaut les données adressées dans le segment sélectionné par DS. DI et EDI adressent toujours les données dans le segment sélectionné par ES. La forme courte est obtenue en attachant au mnémonique de la lettre d'opération de chaîne spécifiant la taille de l'élément de chaîne, elle doit être `b` pour l'élément d'octet, `w` pour l'élément mot et `d` pour l'élément double-mot. La forme complète d'opération de chaîne nécessite des opérandes fournissant l'opérateur de taille et les adresses mémoire, qui peuvent être SI ou ESI avec n'importe quel préfixe de segment, DI ou EDI toujours avec le préfixe de segment ES.

L'instruction **MOVS** transfère l'élément de chaîne pointé par SI (ou ESI) à l'emplacement pointé par DI (ou EDI). La taille des opérandes peut être un octet, un mot ou un double mot. L'opérande destination doit être adressé en mémoire par DI ou EDI, l'opérande source doit être adressé en mémoire par SI ou ESI avec n'importe quel préfixe de segment.

```
movs byte [di],[si]          ; copie l'octet [SI] dans [DI] et ajoute 1 à SI et DI
movs word [es:di],[ss:si]    ; copie de mot dans SS:[SI] dans ES:[DI] et ajoute 2 à SI et DI
movsd                        ; copie le double-mot [ESI] dans [EDI] et ajoute 4 à ESI et EDI
```

L'instruction **CMPS** soustrait l'élément de chaîne de destination de l'élément de chaîne source et met à jour les flags AF, SF, PF, CF et OF, mais ne modifie aucun des éléments comparés. Si les éléments de chaîne sont égaux, ZF est mis à 1, sinon il est mis à 0. Le premier opérande de cette instruction doit être l'élément de chaîne source adressé par SI ou ESI avec n'importe quel préfixe de segment, le second opérande doit être l'élément de chaîne de destination adressé par DI ou EDI.

```
cmpsb                        ; compare les octets [SI] et [DI] et ajoute 1 à SI et DI
cmps word [ds:si],[es:di]    ; compare les mot dans DS:[SI] et ES:[DI] et ajoute 2 à SI et DI
cmps dword [fs:esi],[edi]    ; copie les double-mots FS:[ESI] et [EDI] et ajoute 4 à ESI et EDI
```

L'instruction **SCAS** soustrait l'élément de chaîne de destination de AL, AX ou EAX (en fonction de la taille de l'élément de chaîne) et positionne les flags AF, SF, ZF, PF, CF et OF selon le résultat de l'opération. Si les valeurs sont égales, ZF est à 1, sinon il est mis à 0. L'opérande doit être l'élément de chaîne destination pointé par DI ou EDI.

```
scas byte [es:di]            ; compare AL à ES:[DI] puis incrémente DI d'une unité
scasw                        ; compare AX à [DI] puis incrémente DI de 2 unités
scas dword [es:edi]          ; compare EAX à ES:[EDI] puis incrémente EDI de 4 unités
```

L'instruction **STOS** place la valeur de AL, AX ou EAX dans l'élément de chaîne de destination. Les règles pour l'opérande sont les mêmes que pour l'instruction `scas`.

L'instruction **LODS** place l'élément de chaîne source dans AL, AX ou EAX. L'opérande doit être l'élément de chaîne source pointé par SI ou ESI avec n'importe quel préfixe de segment.

```
lods byte [ds:si]            ; charge dans AL l'octet à l'adresse DS:[SI]
lods word [cs:si]            ; charge dans AX le mot à l'adresse CS:[SI]
lodsd                        ; charge dans EAX le double-mot à l'adresse DS:[SI]
```

L'instruction **INS** transfère un octet, un mot ou un double-mot d'un port d'entrée adressé par le registre DX vers l'élément de chaîne de destination. L'opérande destination doit être adressé en mémoire par DI ou EDI, l'opérande source doit être le registre DX.

```
insb                         ; input byte
ins word [es:di],dx          ; input word
ins dword [edi],dx           ; input double word
```

L'instruction **OUTS** transfère l'élément de chaîne source vers un port de sortie adressé par le registre DX. L'opérande destination doit être le registre DX et l'opérande source doit être adressé en mémoire par SI ou ESI avec n'importe quel préfixe de segment.

```
outs dx,byte [si]            ; envoie l'octet de l'adresse [SI] vers le port de numéro DX
outsw                        ; envoie le mot de l'adresse [SI] vers le port spécifié par DX
outs dx,dword [gs:esi]       ; envoie le double-mot en GS:[ESI] vers le port de numéro DX
```

Les préfixes de répétition **REP**, **REPE/REPZ** et **REPNE/REPNZ** spécifient une opération de chaîne répétée. Lorsqu'une instruction d'opération de chaîne a un préfixe de répétition, l'opération est exécutée à plusieurs reprises, à chaque fois en utilisant un élément différent de la chaîne. La répétition se termine lorsque l'une des conditions

spécifiées par le préfixe est remplie. Les trois préfixes décrémentent automatiquement le registre CX ou ECX (selon que l'instruction d'opération de chaîne utilise l'adressage 16 bits ou 32 bits) après chaque opération et répètent l'opération associée jusqu'à ce que CX ou ECX soit égal à zéro. Les instructions REPE/REPZ et **REPNE/REPNZ** sont utilisés exclusivement avec les instructions SCAS et CMPS (décrites ci-dessous). Lorsque ces préfixes sont utilisés, la répétition de l'instruction suivante dépend également du flag de zéro (ZF). Les préfixes REPE et REPZ terminent l'exécution lorsque ZF est à zéro. REPNE et REPNZ terminent l'exécution lorsque ZF est à 1.

```
rep  movsd                  ; copie des double-mots avec incrémentation ou décrémentation
                            ;   automatique de ESI et EDI
repe cmpsb                  ; compare les octets en [SI] et [DI] jusqu'à ce qu'une
                            ;   inégalité soit rencontrée
```

2.1.9 Instructions de contrôle des flags

Les instructions de contrôle des flags offrent la possibilité de changer directement l'état des bits dans le registre des flags. Toutes les instructions décrites dans cette section n'ont pas d'opérande.

L'instruction **STC** met le CF (flag de retenue) à 1, CLC le met à zéro, tandis que `cmc` le complémente. **STD** met le DF (flag de direction) à 1, **CLD** le met à zéro. L'instruction **STI** met IF (flag d'interruption) à 1 et par conséquent active les interruptions. **CLI** met IF à zéro et désactive, de ce fait, les interruptions.

L'instruction **LAHF** copie les flags SF, ZF, AF, PF et CF sur les bits 7, 6, 4, 2 et 0 du registre AH. Le contenu des bits restants n'est pas défini. Les flags demeurent inchangés au terme de cette opération.

L'instruction **SAHF** transfère les bits 7, 6, 4, 2 et 0 du registre AH vers SF, ZF, AF, PF et CF.

L'instruction **PUSHF** décrémente ESP de 2 ou 4 unités et stocke le mot bas ou le double-mot du registre des flags en haut de la pile. La taille des données stockées dépend du paramètre de code courant. La variante `pushfw` force le stockage d'un mot et **PUSHFD** force le stockage d'un double-mot.

L'instruction **POPF** transfère des bits spécifiques du mot ou du double-mot en haut de la pile, puis incrémente ESP de 2 ou 4 unités selon le cas, cette valeur dépendant du réglage actuel du code. La variante **POPFW** force la restauration à partir du mot et **POPFD** force la restauration à partir du double mot.

2.1.10 Opérations conditionnelles

mettent un octet à un si la condition est vraie et mettent l'octet à zéro sinon. L'opérande doit être un registre général de 8 bits ou l'octet en mémoire.

```
setne al            ; met AL à 1 si le flag de zéro est nul
seto byte [bx]      ; met l'octet pointé par BX à 1 si dépassement (overflow)
```

L'instruction **SALC** met à 1 tous les bits du registre AL lorsque le flag de retenue est positionné et le met à zéro le registre AL dans le cas contraire. Cette instruction n'a pas d'arguments.

Les instructions obtenues en attachant la condition mnémonique à **CMOV** transfèrent le mot ou double mot du registre général ou de la mémoire vers le registre général uniquement lorsque la condition est vraie. L'opérande destination doit être un registre général. L'opérande source peut être un registre général ou une mémoire.

```
cmove ax,bx         ; copie BX dans AX si le flag de zéro est à 1
cmovnc eax,[ebx]    ; copie le double-mot pointé par EBX dans EAX
```

L'instruction **CMPXCHG** compare la valeur du registre AL, AX ou EAX avec l'opérande destination. Si les deux valeurs sont égales, l'opérande source est chargé dans l'opérande destination. Sinon, l'opérande destination est chargé dans le registre AL, AX ou EAX. L'opérande destination peut être un registre général ou une mémoire, l'opérande source doit être un registre général.

```
cmpxchg dl,bl       ; si AL = DL -> DL = BL. Si AL ≠ DL -> AL = DL
cmpxchg [bx],dx     ; si AX = [BX] -> [BX] = DX. Si AX ≠ DX -> AX = [BX]
```

L'instruction **CMPXCHG8B** compare la valeur 64 bits dans les registres EDX et EAX avec l'opérande destination. Si les valeurs sont égales, la valeur 64 bits des registres ECX et EBX est stockée dans l'opérande destination. Sinon, la valeur de l'opérande destination est chargée dans les registres EDX et EAX. L'opérande destination doit être un quadruple mot en mémoire.

```
cmpxchg8b [bx]      ; comparaison et échange de 8 octets
```

2.1.11 Instructions diverses

L'instruction **NOP** occupe un octet mais n'affecte rien d'autre que le pointeur d'instruction. Cette instruction n'a pas d'opérande et n'effectue aucune opération.

L'instruction **UD2** génère une exception d'*opcode* non valide. Cette instruction est fournie pour les tests logiciels afin de générer explicitement un *opcode* non valide. Cette instruction n'a pas d'opérande.

L'instruction **XLAT** remplace un octet dans le registre AL par un octet indexé par sa valeur dans une table de traduction adressée par BX ou EBX. L'opérande doit être une mémoire d'octets adressée par BX ou EBX avec

n'importe quel préfixe de segment. Cette instruction a également une forme abrégée **XLATB** qui n'a pas d'opérande et utilise l'adresse BX ou EBX dans le segment sélectionné par DS en fonction du paramètre de code courant.

L'instruction **LDS** transfère une variable de pointeur de l'opérande source vers DS et le registre de destination. L'opérande source doit être un opérande mémoire et l'opérande destination doit être un registre général. Le registre DS reçoit le sélecteur de segment du pointeur tandis que le registre destination reçoit la partie décalée du pointeur. Les instructions **LFS**, **LGS** et **LSS** fonctionnent de la même manière que LDS sauf que les registres DS, ES, FS, GS et SS sont utilisés respectivement.

```
lds bx,[si]          ; charge le pointeur dans DS:BX
```

L'instruction **LEA** transfère le décalage de l'opérande source (plutôt que sa valeur) à l'opérande destination. L'opérande source doit être un opérande mémoire et l'opérande destination doit être un registre général.

```
lea dx,[bx+si+1]     ; charge l'adresse effective dans DX
```

L'instruction **CPUID** renvoie l'identification du processeur et les informations sur les fonctionnalités dans les registres EAX, EBX, ECX et EDX. Les informations retournées sont sélectionnées en entrant une valeur dans le registre EAX avant que l'instruction ne soit exécutée. Cette instruction n'a pas d'opérande.

L'instruction **PAUSE** retarde l'exécution de l'instruction suivante d'un laps de temps spécifique à l'implémentation. Il peut être utilisé pour améliorer les performances des boucles d'attente de rotation. Cette instruction n'a pas d'opérandes.

L'instruction **ENTER** crée une trame de pile qui peut être utilisée pour implémenter les règles de portée des langages de haut niveau structurés par blocs. Une instruction **LEAVE** à la fin d'une procédure complète le ENTER au début de la procédure pour simplifier la gestion de la pile et contrôler l'accès aux variables pour les procédures imbriquées. L'instruction ENTER comprend deux paramètres. Le premier spécifie le nombre d'octets de stockage dynamique à allouer sur la pile pour la routine en cours de saisie. Le deuxième paramètre correspond au niveau d'imbrication lexical de la routine, qui peut être compris entre 0 et 31. Le niveau lexical spécifié détermine le nombre d'ensembles de pointeurs de trame de pile que la CPU copie dans la nouvelle trame de pile de la trame précédente. Cette liste de pointeurs de trame de pile est parfois appelée affichage. Le premier mot (ou double-mot lorsque le code est de 32 bits) de l'affichage est un pointeur vers la dernière trame de pile. Ce pointeur permet à une instruction de LEAVE d'inverser l'action de l'instruction **ENTER** précédente en supprimant efficacement la dernière trame de pile. Une fois que **ENTER** crée le nouvel affichage pour une procédure, il alloue l'espace de stockage dynamique pour cette procédure en décrémentant ESP du nombre d'octets spécifié dans le premier paramètre. Pour permettre à une procédure d'adresser son affichage, ENTER laisse BP (ou EBP) pointer le début de la nouvelle trame de pile. Si le niveau lexical est nul, ENTER pousse BP (ou EBP) en pile, copie SP vers BP (ou ESP vers EBP) puis soustrait le premier opérande de ESP. Pour les niveaux d'imbrication supérieurs à zéro, le processeur pousse des pointeurs de trame supplémentaires sur la pile avant d'ajuster le pointeur de pile.

```
enter 2048,0    ; création d'une trame de pile et allocation de 2048 octets sur la pile
```

2.1.12 Instructions système

L'instruction **LMSW** charge l'opérande dans le mot d'état de la machine (bits 0 à 15 du registre CR0), tandis que **SMSW** stocke le mot d'état de la machine dans l'opérande destination. L'opérande de ces deux instructions peut être un registre général de 16 bits ou une mémoire. Pour SMSW, il peut également s'agir d'un registre général de 32 bits.

```
lmsw ax      ; chargement de l'état de la machine à partir du registre
smsw [bx]    ; stockage de l'état de la machine dans la mémoire
```

Les instructions **LGDT** et **LIDT** chargent les valeurs de l'opérande respectivement dans le registre de table de descripteurs global ou dans le registre de table de descripteurs d'interruption Les instructions **SGDT** et **SIDT** stockent le contenu du registre de table de descripteurs global ou du registre de table de descripteurs d'interruption dans l'opérande destination. L'opérande doit être un ensemble de 6 octets contigus en mémoire.

```
lgdt [ebx]   ; chargement de la table des descripteurs globaux
```

L'instruction **LLDT** charge l'opérande dans le champ de sélecteur de segment du registre de table de descripteur local et sldt stocke le sélecteur de segment du registre de table de descripteur local dans l'opérande. L'instruction **LTRC** harge l'opérande dans le champ de sélecteur de segment du registre de tâche et **STR** stocke le sélecteur de segment du registre de tâche dans l'opérande. Les règles pour l'opérande sont les mêmes que pour les instructions **LMSW** et **SMSW**.

L'instruction **LAR** charge les droits d'accès du descripteur de segment spécifié par le sélecteur dans l'opérande source dans l'opérande destination et définit le flag ZF. L'opérande destination peut être un registre général de 16 bits ou 32 bits. L'opérande source doit être un registre général ou un emplacement mémoire 16 bits.

```
lar ax, [bx]    ; chargement des droits d'accès dans un mot
lar eax, dx     ; chargement des droits d'accès dans un double mot
```

L'instruction **LSL** charge la limite de segment du descripteur de segment spécifié par le sélecteur dans l'opérande source dans l'opérande destination et définit le flag ZF. Les règles pour l'opérande sont les mêmes que pour l'instruction LAR.
Les instructions **VERR** et **VERW** vérifient si le code ou le segment de données spécifié avec l'opérande est lisible ou inscriptible à partir du niveau de privilège actuel. L'opérande doit être un mot, ce peut être un registre général ou une mémoire. Si le segment est accessible et lisible (pour VERR) ou inscriptible (pour VERW), le flag ZF est défini, sinon il est effacé. Les règles pour l'opérande sont les mêmes que pour l'instruction LLDT.

L'instruction **ARPL** compare les champs RPL (niveau de privilège du demandeur) de deux sélecteurs de segment. Le premier opérande contient un sélecteur de segment et le second opérande contient l'autre. Si le champ RPL de l'opérande destination est inférieur au champ RPL de l'opérande source, le flag ZF est positionné et le champ RPL de l'opérande destination est augmenté pour correspondre à celui de l'opérande source. Sinon, le flag ZF est effacé et aucune modification n'est apportée à l'opérande destination. L'opérande destination peut être un registre général de mot ou une mémoire, l'opérande source doit être un registre général.

```
arpl bx, ax     ; ajustement du RPL du sélecteur dans le registre
arpl [bx], ax   ; ajustement du RPL du sélecteur en mémoire
```

L'instruction **CLTS** efface le flag TS (tâche commutée) dans le registre CR0. Cette instruction n'a pas d'opérande.

Le préfixe **LOCK** provoque l'affirmation du signal de verrouillage de bus du processeur pendant l'exécution de l'instruction d'accompagnement. Dans un environnement multiprocesseur, le signal de verrouillage de bus garantit que le processeur a une utilisation exclusive de toute mémoire partagée pendant que le signal est affirmé. Le préfixe LOCK peut être préfixé uniquement aux instructions et qu'aux formes des instructions suivantes où l'opérande destination est une mémoire opérande : ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB SUB, XOR, XADD et XCHG. Si le préfixe LOCK est utilisé avec l'une de ces instructions et l'opérande source est un opérande mémoire, une exception de code d'opération non définie peut être générée. Une exception d'opcode non définie sera également générée si le LOCK préfixe est utilisé avec une instruction ne figurant pas dans la liste ci-dessus. L'instruction XCHG affirme toujours le signal de verrouillage de bus indépendamment de la présence ou de l'absence du préfixe LOCK.

L'instruction **HLT** arrête l'exécution de l'instruction et place le processeur dans un état arrêté. Une interruption activée, une exception de débogage, le signal **BINIT**, **INIT** ou **RESET** reprendra l'exécution. Cette instruction n'a pas d'opérande.

L'instruction **INVLPG** invalide (vide) l'entrée TLB (Translation Lookaside Buffer) spécifiée avec l'opérande, qui devrait être une mémoire. Le processeur détermine la page qui contient cette adresse et vide l'entrée TLB pour cette page.

L'instruction **RDMSR** charge le contenu d'un MSR 64 bits (registre spécifique au modèle) de l'adresse spécifiée dans le registre ECX dans les registres EDX et EAX. **WRMSR** écrit le contenu des registres EDX et EAX dans le MSR 64 bits de l'adresse spécifiée dans le registre ECX. **RDTSC** charge la valeur courante du compteur d'horodatage du processeur du MSR 64 bits dans les registres EDX et EAX. Le processeur incrémente le compteur d'horodatage MSR à chaque cycle d'horloge et le remet à 0 chaque fois que le processeur est réinitialisé. RDPMCcharge le contenu du compteur de surveillance des performances 40 bits spécifié dans le registre ECX dans les registres EDX et EAX. Ces instructions n'ont pas d'opérande.

L'instruction **WBINVD** réécrit toutes les lignes de cache modifiées dans le cache interne du processeur dans la mémoire principale et invalide (vide) les caches internes. L'instruction émet ensuite un cycle de bus de fonction spécial qui ordonne aux caches externes de réécrire également les données modifiées et un autre cycle de bus pour indiquer que les caches externes doivent être invalidés. Cette instruction n'a pas d'opérande.

L'instruction **RSM** retourne le contrôle du programme du mode de gestion du système au programme qui a été interrompu lorsque le processeur a reçu une interruption **SMM**. Cette instruction n'a pas d'opérande.

L'instruction **SYSENTER** exécute un appel rapide à une procédure système de niveau 0. L'instruction **SYSEXIT** exécute un retour rapide au code utilisateur de niveau 3. Les adresses utilisées par ces instructions sont stockées dans les MSR. Ces instructions n'ont pas d'opérande.

2.1.13 Instructions FPU

Les instructions FPU (Floating-Point Unit) fonctionnent avec des valeurs en virgule flottante selon trois formats : simple précision (32 bits), double précision (64 bits) et double précision étendue (80 bits). Les registres FPU forment la pile et chacun d'eux contient une valeur à virgule flottante à double précision étendue. Lorsque certaines valeurs sont poussées sur la pile ou sont supprimées du haut de cette pile, les registres FPU sont décalés. Il en résulte que ST0 est toujours la valeur en haut de la pile FPU, ST1 est la première valeur en dessous du haut, etc. Le nom ST0 a également le synonyme ST.

L'instruction **FLD** pousse la valeur à virgule flottante sur la pile de registres FPU. L'opérande peut être un emplacement mémoire 32 bits, 64 bits ou 80 bits ou le registre FPU. Sa valeur est ensuite chargée en haut de la pile de registres FPU (le registre ST0) et est automatiquement convertie au format double précision étendue.

```
fld dword [bx]    ; chargement d' une valeur en simple précision localisée en mémoire
fld st2           ; push de la valeur de st2 dans la pile de registres
```

Les instructions **FLD1**, **FLDZ**, **FLDL2T**, **FLDL2E**, **FLDPI**, **FLDLG2** et **FLDLN2** chargent les contantes couramment utilisés sur la pile de registres FPU. Les constantes chargées sont respectivement +1,0, +0,0, log210, log2e, π, log102 et ln 2. Ces instructions n'ont pas d'opérande.

L'instruction **FILD** convertit l'opérande source d'entier signé au format à virgule flottante à double précision étendue et pousse le résultat sur la pile de registres FPU. L'opérande source peut être un emplacement de mémoire 16 bits, 32 bits ou 64 bits.

```
fild qword [bx]    ; chargement d'une valeur entière 64-bits à partir de la mémoire
```

L'instruction **FST** copie la valeur du registre ST0 dans l'opérande destination, qui peut être un emplacement de mémoire 32 bits ou 64 bits ou un autre registre FPU. FSTP effectue la même opération que **FST**, puis fait apparaître la pile de registres, éliminant ST0. L'instruction **FSTP** accepte les mêmes opérandes que la première instruction et peut également stocker la valeur dans la mémoire 80 bits.

```
fst st3           ; copie de la valeur dans st0 dans le registre st3
fstp tword [bx]   ; stockage d'une valeur en mémoire et dépilage
```

L'instruction **FIST** convertit la valeur de ST0 en entier signé et stocke le résultat dans l'opérande destination. L'opérande peut être un emplacement de mémoire 16 bits ou 32 bits. FISTP effectue la même opération et fait apparaître la pile de registres, il accepte les mêmes opérandes que la première instruction et peut également stocker une valeur entière dans la mémoire 64 bits, de sorte qu'il a les mêmes règles pour les opérandes que l'instruction FILD.

L'instruction **FBLD** convertit l'entier BCD compressé en format à virgule flottante à double précision étendue et pousse cette valeur sur la pile FPU. L'instruction **FBSTP** convertit la valeur de ST0 en un entier BCD compressé à 18 chiffres, stocke le résultat dans l'opérande destination et fait apparaître la pile de registres. L'opérande doit être un emplacement mémoire de 80 bits.

L'instruction **FADD** ajoute la destination et l'opérande source et stocke la somme dans l'emplacement de destination. L'opérande destination est toujours un registre FPU, si la source est un emplacement mémoire, la destination est le registre ST0 et seul l'opérande source doit être spécifié. Si les deux opérandes sont des registres FPU, au moins l'un d'entre eux doit être un registre ST0. Un opérande en mémoire peut être une valeur de 32 bits ou 64 bits.

```
fadd qword [bx]    ; addition de la valeur en double précision dans [BX] à st0
fadd st2, st0      ; addition de st0 à st2
```

L'instruction **FADDP** ajoute l'opérande destination et source, stocke la somme dans l'emplacement de destination, puis fait apparaître la pile de registres. L'opérande destination doit être un registre FPU et l'opérande source doit être le ST0. Lorsqu'aucun opérande n'est spécifié, ST1 est utilisé comme opérande destination.

```
faddp             ; addition de st0 à st1 et dépilage
faddp st2, st0    ; addition de st0 à st2 et dépilage
```

L'instruction **FIADD** convertit un opérande source entier en une valeur à virgule flottante à double précision étendue et l'ajoute à l'opérande destination. L'opérande doit être un emplacement mémoire 16 bits ou 32 bits.

```
fiadd word [bx]    ; addition du mot entier en [BX] dans st0
```

Les instructions **FSUB**, **FSUBR**, **FMUL**, **FDIV**, et **FDIVR** sont similaires à FADD, ont les mêmes règles pour les opérandes et ne diffèrent que dans le calcul effectué. L'instruction FSUB soustrait l'opérande source de l'opérande destination. FSUBR soustrait l'opérande destination de l'opérande source, FMUL multiplie les opérandes de destination et source, FDIV divise l'opérande destination par l'opérande source et FDIVR divise l'opérande source par l'opérande destination. Les instructions FSUBP, FSUBRP, FMULP, FDIVP et FDIVRP effectuent les mêmes opérations et popent la pile de registres. Les règles pour l'opérande sont les mêmes que pour FADDP. Les instructions **FISUB**, **FISUBR**, **FIMUL**, **FIDIV** et **FIDIVR** effectuent ces opérations après avoir converti l'opérande source entier en valeur à virgule flottante. Elles ont les mêmes règles pour les opérandes que l'instruction **FIADD**.

L'instruction **FSQRT** calcule la racine carrée de la valeur dans le registre ST0, **FSIN** calcule le sinus de cette valeur, **FCOS** en calcule le cosinus, **FCHS** complète son bit de signe, **FABS** met son signe à zéro pour créer la valeur absolue, **FRNDINT** l'arrondit à la valeur entière la plus proche, selon le mode d'arrondi courant. L'instruction **F2XM1** calcule la valeur exponentielle de 2 à la puissance de ST0 et en soustrait 1,0, la valeur de ST0 devant être comprise entre -1,0 et +1,0. Toutes ces instructions stockent le résultat dans ST0 et n'ont pas d'opérande.

L'instruction **FSINCOS** calcule à la fois le sinus et le cosinus de la valeur dans le registre ST0. Elle stocke le sinus dans ST0 et pousse le cosinus en haut de la pile de registres FPU. L'instruction **FPTAN** calcule la tangente de la valeur dans ST0, stocke le résultat dans ST0 et pousse un 1.0 sur la pile de registres FPU. **FPATAN** calcule l'arc tangente de la valeur dans ST1 divisée par la valeur dans ST0, stocke le résultat dans ST1 et fait apparaître la pile de registres FPU. L'instruction **FYL2X** calcule le logarithme binaire de ST0, le multiplie par ST1, stocke le résultat dans ST1 et fait apparaître la pile de registres FPU. L'instruction **FYL2XP1** effectue la même opération mais ajoute 1.0 à ST0 avant de calculer le logarithme. L'instruction **FPREM** calcule le reste obtenu en divisant la

valeur de ST0 par la valeur de ST1 et stocke le résultat dans ST0. L'instruction **FPREM1** effectue la même opération que fprem, mais il calcule le reste de la manière spécifiée par la norme IEEE 754. **FSCALE** tronque la valeur de ST1 et augmente l'exposant de ST0 de cette valeur. **FXTRACT** sépare la valeur de ST0 en son exposant et sa mantisse, stocke l'exposant dans ST0 et pousse la mantisse sur la pile de registres. **FNOP** n'effectue aucune opération. Ces instructions n'ont pas d'opérande.

L'instruction **FXCH** échange le contenu de ST0 avec un autre registre FPU. L'opérande doit être un registre FPU. Si aucun opérande n'est spécifié, le contenu de ST0 et ST1 est échangé.

Les instructions **FCOM** et **FCOMP** comparent le contenu de ST0 et de l'opérande source et définissent des flags dans le mot d'état FPU en fonction des résultats. L'instruction **FCOMP** affiche également la pile de registres après avoir effectué la comparaison. L'opérande peut être une valeur de précision simple ou double en mémoire ou dans le registre FPU. Lorsqu'aucun opérande n'est spécifié, ST1 est utilisé comme opérande source.

```
fcom       ; comparaison de st0 avec st1
fcomp st2  ; comparaison de st0 avec st2 et dépilage
```

L'instruction **FCOMPP** compare le contenu de ST0 et ST1, définit les flags dans le mot d'état FPU en fonction des résultats et fait apparaître deux fois la pile de registres. Cette instruction n'a pas d'opérande.

Les instructions **FUCOM**, **FUCOMP** et **FUCOMPP** effectuent une comparaison non ordonnée de deux registres FPU. Les règles pour les opérandes sont les mêmes que pour les instructions **FCOM**, **FCOMP** et **FCOMPP**, mais l'opérande source doit être un registre FPU.

Les instructions **FICOM** et **FICOMP** comparent la valeur dans ST0 avec un opérande source entier et définissez les flags dans le mot d'état FPU en fonction des résultats. FICOMP fait apparaître en outre la pile de registres après avoir effectué la comparaison. La valeur entière est convertie au format à virgule flottante à double précision étendue avant que la comparaison ne soit effectuée. L'opérande doit être un emplacement mémoire 16 bits ou 32 bits.

```
ficom word [bx]     ; comparaison du contenu de st0 avec un entier 16 bits
```

Les instructions **FCOMI**, **FCOMIP**, **FUCOMI** et **FUCOMIP** effectuent la comparaison de ST0 avec un autre registre FPU et positionne les flags ZF, PF et CF en fonction du résultat. FCOMIP et FUCOMIP effectuent en outre un dépilage de registres après avoir effectué la comparaison. Les instructions obtenues en attachant le mnémonique de condition FPU (voir tableau 2.2) à la suite du mnémonique **FCMOV** transfèrent le registre FPU spécifié dans le registre ST0 si la condition de test donnée est vraie. Ces instructions autorisent deux syntaxes différentes, une avec un seul opérande spécifiant le registre FPU source, et une avec deux opérandes, dans ce cas l'opérande destination doit être le registre ST0 et le second opérande spécifie le registre FPU source.

```
fcomi st2          ; comparaison du contenu de st0 avec celui de st2 et définition des flags
fcmovb st0, st2    ; transfert de st2 à st0 si inférieur
```

Tableau 2.2 – Conditions FPU

Mnémonique	Condition testée	Description	
b	CF = 1	au dessous de	below
e	ZF = 1	égal	equal
be	CF ou ZF = 1	égal	equal
u	PF = 1	désordonné	unordered
nb	CF = 0	pas en dessous	not below
ne	ZF = 0	inégal	not equal
nbe	CF ou ZF = 0	inégal	not equal
nu	PF = 0	pas sans ordre	not unordered

L'instruction **FTST** compare la valeur de ST0 à 0,0 et définit les flags dans le mot d'état FPU en fonction des résultats. L'instruction FXAM examine le contenu du ST0 et définit les flags dans le mot d'état FPU pour indiquer la classe de valeur dans le registre. Ces instructions n'ont pas d'opérande.

Les instructions **FSTSW** et **FNSTSW** stockent la valeur courante du mot d'état FPU dans l'emplacement destination. L'opérande destination peut être une mémoire 16 bits ou le registre AX. FSTSW vérifie les exceptions FPU non masquées en attente avant de stocker le mot d'état, ce que FNSTSW ne fait pas.

Les instructions **FSTCW** et **FNSTCW** stockent la valeur courante du mot de commande FPU à la destination spécifiée en mémoire. FSTCW vérifie les exceptions FPU non masquées en attente avant de stocker le mot de contrôle, ce que FNSTCW ne fait pas. L'instruction **FLDCW** charge l'opérande dans le mot de contrôle FPU. L'opérande doit être un emplacement de mémoire 16 bits.

Les instructions **FSTENV** et **FNSTENV** stockent l'environnement d'exploitation FPU courant à l'emplacement de mémoire spécifié avec l'opérande destination, puis masquent toutes les exceptions FPU. FSTENV vérifie les exceptions FPU non masquées en attente avant de continuer, ce que FNSTENV ne fait pas. FLDENV charge l'environnement d'exploitation complet de la mémoire dans la FPU.

Les instructions **FSAVE** et **FNSAVE** stockent l'état courant de la FPU (environnement d'exploitation et pile de

registres) à la destination spécifiée dans la mémoire et réinitialisent la FPU. FSAVE vérifie les exceptions FPU non masquées en attente avant de continuer, ce que FNSAVE ne fait pas.

L'instruction **FRSTOR** charge l'état de la FPU à partir de l'emplacement mémoire spécifié. Toutes ces instructions nécessitent un opérande comme emplacement mémoire. Pour chacune de ces instructions existent deux mnémoniques supplémentaires qui permettent de sélectionner précisément le type d'opération. Les mnémoniques **FSTENVW**, **FNSTENVW**, **FLDENVW**, **FSAVEW**, **FNSAVEW** et **FRSTORW** forcent l'instruction à effectuer l'instruction opération comme dans le mode 16 bits, tandis que **FSTENVD**, **FNSTENVD**, **FLDENVD**, **FSAVED**, **FNSAVED** et **FRSTORD** forcent l'opération comme dans le mode 32 bits.

Les instructions **FINIT** et **FNINIT** définissent l'environnement d'exploitation FPU dans son état par défaut. FINIT vérifie les exceptions FPU non masquées en attente avant de continuer, ce que FNINIT ne fait pas. Les instructions **FCLEX** et **FNCLEX** effacent les flags d'exception FPU dans le mot d'état FPU. FCLEX vérifie les exceptions FPU non masquées en attente avant de continuer, ce que FNCLEX ne fait pas. Les instructions **WAIT** et **FWAIT** sont des synonymes de la même instruction, ce qui oblige le processeur à vérifier les exceptions FPU non masquées en attente et à les traiter avant de continuer. Ces instructions n'ont pas d'opérande.

L'instruction **FFREE** définit la balise associée au registre FPU spécifié sur vide. L'opérande doit être un registre FPU.

Les instructions **FFINCSTP** et **FDECSTP** font pivoter la pile FPU d'une unité en ajoutant ou en soustrayant un au pointeur du haut de la pile. Ces instructions n'ont pas d'opérande.

2.1.14 Instructions MMX

Les instructions MMX fonctionnent sur les types entiers condensés et utilisent les registres MMX, qui sont les parties 64 bits de faible niveau des registres FPU 80 bits. Pour cette raison, les instructions MMX ne peuvent pas être utilisées en même temps que les instructions FPU. Elles peuvent fonctionner sous le format d'octets multiples (huit entiers de 8 bits), de mots multiples (quatre entiers de 16 bits) ou de double-mots multiples (deux entiers de 32 bits). L'utilisation de formats multiples autorise un parallélisme de fonctionnement permettant d'effectuer des opérations sur plusieurs données à la fois.

L'instruction **MOVQ** copie un quadruple mot de l'opérande source vers l'opérande destination. Au moins l'un des opérandes doit être un registre MMX. Le second peut également être un registre MMX ou un emplacement mémoire 64 bits.

```
movq mm0, mm1     ; déplacer le mot quadruple d'un registre à un autre
movq mm2, [ebx]   ; déplacer le mot quadruple de la mémoire vers le registre
```

L'instruction **MOVD** copie un double mot de l'opérande source vers l'opérande destination. L'un des opérandes doit être un registre MMX. Le second peut être un registre général ou un emplacement mémoire 32 bits. Seul le double mot bas du registre MMX est utilisé.

Toutes les opérations MMX générales ont deux opérandes. L'opérande destination doit être un registre MMX. L'opérande source peut être un registre MMX ou un emplacement mémoire 64 bits. L'opération est effectuée sur les éléments de données correspondants de l'opérande source et destination et stockée dans les éléments de données de l'opérande destination.

Les instructions **PADDB**, **PADDW** et **PADDD** effectuent l'addition d'octets, de mots ou de double-mots multiples.

Les instructions **PSUBB**, **PSUBW** et **PSUBD** effectuent la soustraction des types appropriés.

Les instructions **PADDSB**, **PADDSW**, **PSUBSB** et **PSUBSW** effectuent l'addition ou la soustraction d'octets ou ou mots multiples avec saturation signée. Les instructions **PADDUSB**, **PADDUSW**, **PSUBUSB** et **PSUBUSW** sont analogues mais avec saturation non signée.

Les instructions **PMULHW** et **PMULLW** effectuent une multiplication signée des mots multiples et stockent les mots haut ou bas des résultats dans l'opérande destination. L'instruction **PMADDWD** effectue une multiplication des mots multiples et additionne les quatre produits de mots doubles intermédiaires par paire pour produire le résultat sous forme de doubles mots multiples.

Les instructions **PAND**, **POR** et **PXOR** effectuent des opérations logiques sur des quadruples mots. **PANDN** effectue également une négation logique de l'opérande destination avant d'effectuer l'opération AND.

Les instructions **PCMPEQB**, **PCMPEQW** et **PCMPEQD** testent l'égalité d'octets, de mots ou de doubles mots multiples. Si une paire d'éléments de données est égale, l'élément de donnée correspondant dans l'opérande destination est rempli de bits de valeur 1, sinon il est mis à 0.

Les instructions **PCMPGTB**, **PCMPGTW** et **PCMPGTD** exécutent l'opération similaire, mais ils vérifient si les éléments de données dans l'opérande destination sont supérieurs aux éléments de données correspondants dans l'opérande source.

L'instruction **PACKSSWB** convertit les mots signés multiples en octets signés multiples, **PACKSSDW** convertit les doubles mots signés multiples en mots signés multiples en utilisant la saturation pour gérer les conditions de débordement. **PACKUSWB** convertit les mots signés multiples en octets non signés multiples. Les éléments de

données convertis de l'opérande source sont stockés dans la partie haute de l'opérande destination, tandis que les éléments de données convertis de l'opérande destination sont stockés dans la partie basse.

Les instructions **PUNPCKHBW**, **PUNPCKHWD** et **PUNPCKHDQ** entrelacent les éléments de données des parties hautes des opérandes de source et de destination et stockent le résultat dans l'opérande destination.

Les instructions **PUNPCKLBW**, **PUNPCKLWD** et **PUNPCKLDQ** effectuent la même opération, mais les parties basses de l'opérande source et destination sont utilisées.

```
paddsb mm0, [esi]  ; addition d'octets multiples avec saturation signée
pcmpeqw mm3, mm7   ; comparaison de mots multiples sur le critère d'égalité
```

Les instructions **PSLLW**, **PSLLD** et **PSLLQ** effectuent un décalage logique à gauche de mots ou de doubles mots multiples ou d'un quadruple mot unique dans l'opérande destination de la quantité de bits spécifiée dans l'opérande source.

Les instructions **PSRLW**, **PSRLD** et **PSRLQ** effectuent un décalage logique à droite de mots ou de doubles mots multiples ou d'un quadruple mot unique dans l'opérande destination de la quantité de bits spécifiée dans l'opérande source.

Les instructions **PSRAW** et **PSRAD** effectuent un décalage arithmétique des mots ou doubles mots multiples. L'opérande destination doit être un registre MMX, tandis que l'opérande source peut être un registre MMX, un emplacement mémoire 64 bits ou une valeur immédiate de 8 bits.

```
psllw mm2, mm4     ; décalage logique de mots vers la gauche
psrad mm4, [ebx]   ; décalage arithmétique de doubles mots vers la droite
```

L'instruction **EMMS** rend les registres FPU utilisables pour les instructions FPU. Elle doit être utilisée avant de mettre en œuvre des instructions FPU si des instructions MMX ont été utilisées.

2.1.15 Instructions SSE

L'extension SSE étend le jeu d'instructions MMX et introduit également les opérations sur les valeurs à virgule flottante simple précision multiples. Le format simple précision multiple de 128 bits juxtapose quatre valeurs virgule flottante simple précision. Les registres SSE 128 bits sont conçus pour effectuer des opérations sur ce type de données.

Les instructions **MOVAPS** et **MOVUPS** transfèrent un opérande source composé de deux quadruples-mots contenant des valeurs simple précision multiples à l'opérande destination. Au moins l'un des opérandes doit être un registre SSE, le second peut également être un registre SSE ou un emplacement mémoire de 128 bits. Les opérandes de mémoire pour l'instruction MOVAPS instruction doivent être alignés sur une limite de 16 octets, contrairement aux opérandes pour l'instruction MOVUPS qui n'ont pas besoin d'être alignés.

```
movups xmm0, [ebx]    ; déplacement d'un quadruple-mot non aligné
```

L'instruction **MOVLPS** déplace deux valeurs simple-précision entre la mémoire et le quadruple mot bas du registre SSE. L'instruction **MOVHPS** déplace deux valeurs simple-précision entre la mémoire et le quadruple mot haut du registre SSE. L'un des opérandes doit être un registre SSE ; l'autre doit être un emplacement de mémoire 64 bits.

```
movlps xmm0, [ebx]  ; déplacement de la mémoire vers le quadruple-mot bas de xmm0
movhps [esi], xmm7  ; déplacement du quadruple-mot haut de xmm7 vers la mémoire
```

L'instruction **MOVLHPS** déplace deux valeurs en simple-précision du quadruple-mot bas du registre source vers le quadruple-mot haut du registre de destination. L'instruction **MOVHLPS** déplace deux valeurs en simple-précision du quadruple-mot haut du registre source vers le quadruple-mot bas du registre de destination. Les deux opérandes doivent être des registres SSE.

L'instruction **MOVMSKPS** transfère le bit le plus significatif de chacune des quatre valeurs en simple-précision dans le registre SSE dans les quatre bits de poids faible d'un registre général. L'opérande source doit être un registre SSE. L'opérande destination doit être un registre général.

L'instruction **MOVSS** transfère une valeur simple-précision entre l'opérande source et l'opérande destination (seul le double-mot bas est transféré). Au moins l'un des opérandes doit être un registre SSE. Le second peut également être un registre SSE ou un emplacement mémoire de 32 bits.

```
movss [edi], xmm3  ; déplacement du double mot bas de xmm3 vers la mémoire
```

Chacune des opérations arithmétiques SSE a deux variantes. Lorsque le mnémonique se termine par PS, l'opérande source peut être un emplacement de mémoire de 128 bits ou un registre SSE, l'opérande destination doit être un registre SSE et l'opération est effectuée sur quatre valeurs de précision simples compactées, pour chaque paire d'éléments de données correspondants séparément, le résultat est stocké dans le registre de destination. Lorsque le mnémonique se termine par SS, l'opérande source peut être un emplacement de mémoire 32 bits ou un registre SSE. L'opérande destination doit être un registre SSE et l'opération est effectuée sur des valeurs en simple-précision. Seuls les doubles-mots de poids faible des registres SSE sont utilisés dans ce cas. Le résultat est stocké dans le double-mot bas du registre destination.

Les instructions **ADDPS** et **ADDSS** additionnent les valeurs. Les instructions **SUBPS** et **SUBSS** soustraient la valeur de la source de la valeur de la destination.

```
addps xmm3, xmm7   ; addition de valeurs multiples simple-précision
```

Les instructions **MULPS** et **MULSS** multiplient les valeurs

```
mulss xmm0, [ebx]  ; multiplication de valeurs simple précision
```

Les instructions **DIVPS** et **DIVSS** divisent la valeur cible par la valeur de la source.

Les instructions **RCPPS** et **RCPSS** calculer l'inverse approché de la valeur de la source.

Les instructions **SQRTPS** et **SQRTSS** calculent la racine carrée de la valeur de la source.

Les instructions **RSQRTPS** et **RSQRTSS** calculent l'inverse approchée de la racine carrée de la valeur de la source.

Les instructions **MAXPS** et **MAXSS** comparent les valeurs source et destination et retournent la plus grande.

Les instructions **MINPS** et **MINSS** comparent les valeurs source et destination et retournent la plus petite.

Les instructions **ANDPS**, **ANDNPS**, **ORPS** et **XORPS** effectuent des opérations logiques sur des valeurs multiples simple précision. L'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE, l'opérande destination doit être un registre SSE.

L'instruction **CMPPS** compare les valeurs multiples en simple précision et renvoie un résultat de masque dans l'opérande destination, qui doit être un registre SSE. L'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE. Le troisième opérande doit être un code de sélection d'opérande immédiat de l'une des huit conditions de comparaison (cf. tableau 2.3). L'instruction **CMPSS** effectue la même opération sur des valeurs en simple précision. Seul, le double mot de poids faible du registre destination est affecté. Dans ce cas, l'opérande source peut être un emplacement mémoire de 32 bits ou un registre SSE. Ces deux instructions ont également des variantes avec seulement deux opérandes et la condition est codée dans le mnémonique. Ce mnémonique est obtenu en attachant le mnémonique du tableau 2.3 au mnémonique CMP puis en attachant les suffixes PS ou SS à la fin.

```
cmpps xmm2, xmm4, 0    ; comparaison de valeurs en simple précision multiples
cmpltss xmm0, [ebx]    ; comparaison de valeurs en simple précision
```

Tableau 2.3 – Conditions SSE

Code	Mnémonique	Description	
0	eq	Egal à	equal
1	lt	moins que	less than
2	le	moins que ou égal	less than or equal
3	unord	non-ordonné	unordered
4	neq	non-égal à	not equal
5	nlt	non-moins que	not less than
6	nle	non-moins que ni égal	not less than nor equal
7	ord	ordonné	ordered

Les instructions **COMISS** et **UCOMISS** comparent des valeurs de précision simple et définissent les flags ZF, PF et CF pour afficher le résultat. L'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement de mémoire 32 bits ou un registre SSE.

L'instruction **SHUFPS** déplace deux des quatre valeurs de précision simple de l'opérande destination dans le mot quadruple bas de l'opérande destination et deux des quatre valeurs quelconques de l'opérande source dans le mot quadruple haut de l'opérande destination. L'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement de mémoire de 128 bits ou un registre SSE, le troisième opérande doit être une valeur immédiate de 8 bits sélectionnant les valeurs qui seront déplacées dans l'opérande destination. Les bits 0 et 1 sélectionnent la valeur à déplacer de l'opérande destination vers le double mot bas du résultat, les bits 2 et 3 sélectionnent la valeur à déplacer de l'opérande destination vers le deuxième double-mot. Les bits 4 et 5 sélectionnent la valeur à être déplacé de l'opérande source vers le troisième double-mot. Les bits 6 et 7 sélectionnent la valeur à déplacer de l'opérande source vers le double mot haut du résultat.

```
shufps xmm0, xmm0, 10010011b ; mélange des doubles-mots
```

L'instruction **UNPCKHPS** effectue un démultiplexage entrelacé des valeurs des parties hautes des opérandes source et destination et stocke le résultat dans l'opérande destination, qui doit être un registre SSE. L'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE. L'instruction UNPCKLPS effectue un démultiplexage entrelacé des valeurs des parties basses de l'opérande source et de destination et stocke le résultat dans l'opérande destination. Les règles concernant les opérandes sont les mêmes.

L'instruction **CVTPI2PS** convertit deux doubles-mots entiers multiples en deux valeurs à virgule flottante simple précision multiples et stocke le résultat dans le quadruple-mot bas de l'opérande destination, qui doit être un registre SSE. L'opérande source peut être un emplacement de mémoire 64 bits ou un registre MMX.

```
cvtpi2ps xmm0, mm0   ; conversion d'entiers en valeurs simple-précision
```

L'instruction **CVTSI2SS** convertit un double-mot entier en une valeur à virgule flottante simple précision et stocke le résultat dans le double-mot bas de l'opérande destination, qui doit être un registre SSE. L'opérande source peut être un emplacement mémoire 32 bits ou un registre général 32 bits.

```
cvtsi2ss xmm0, eax   ; conversion d'un entier en une valeur simple-précision
```

L'instruction **CVTPS2PI** convertit deux valeurs multiples en virgule flottante simple précision en deux doubles-mots entiers multiples et stocke le résultat dans l'opérande destination, qui doit être un registre MMX. L'opérande source peut être un emplacement de mémoire 64 bits ou un registre SSE. Seul le quadruple-mot bas du registre SSE est utilisé. L'instruction **CVTTPS2PI** exécute une opération similaire, sauf que la troncature est utilisée pour arrondir les valeurs d'une source à des entiers. Les règles concernant les opérandes sont les mêmes.

```
cvtps2pi mm0, xmm0   ; conversion de valeurs simple-précision en entiers
```

L'instruction **CVTSS2SI** convertit une valeur en virgule flottante simple précision en un double-mot entier et stocke le résultat dans l'opérande destination qui doit être un registre général de 32 bits. L'opérande source peut être un emplacement mémoire de 32 bits ou un registre SSE. Seul le double mot bas du registre SSE est utilisé. L'instruction **CVTTSS2SI** exécute une opération similaire, sauf que la troncature est utilisée pour arrondir une valeur source à un entier. Les règles concernant les opérandes sont les mêmes.

```
cvtss2si eax, xmm0   ; conversion d'une valeur simple-précision en entier
```

L'instruction **PEXTRW** copie le mot de l'opérande source spécifié par le troisième opérande dans l'opérande destination. L'opérande source doit être un registre MMX, l'opérande destination doit être un registre général de 32 bits (le mot haut de la destination est effacé), le troisième opérande doit être une valeur immédiate de 8 bits.

```
pextrw eax, mm0, 1    ; extraction du mot récupéré dans eax
```

L'instruction **PINSRW** insère un mot de l'opérande source dans l'opérande destination à l'emplacement désigné par le troisième opérande qui doit être une valeur immédiate de 8 bits. L'opérande destination doit être un registre MMX. L'opérande source peut être un emplacement de mémoire 16 bits ou un registre général 32 bits (seul le mot bas du registre est utilisé).

```
pinsrw mm1, ebx, 2   ; insérertion d'un mot depuis ebx
```

Les instructions **PAVGB** et **PAVGW** calculent la moyenne arithmétique des octets ou mots multiples.

L'instruction **PMAXUB** renvoie la valeur maximale des octets non signés multiples. L'instruction **PMINUB** renvoie la valeur minimals des octets non signés multiples. L'instruction **PMAXSW** renvoie les valeurs maximales des mots signés condensés. L'instruction **PMINSW** renvoie les valeurs minimales des mots signés condensés. **PMULHUW** effectue une multiplication non signée des mots condensés et stocke les mots hauts des résultats dans l'opérande destination. L'instruction **PSADBW** calcule les différences absolues des octets non signés multiples, additionne les différences et stocke la somme dans le mot bas de l'opérande destination. Toutes ces instructions suivent les mêmes règles pour les opérandes que les opérations MMX générales décrites dans la section précédente.

L'instruction **PMOVMSKB** crée un masque composé du bit le plus significatif de chaque octet de l'opérande source et stocke le résultat dans l'octet de poids faible de l'opérande destination. L'opérande source doit être un registre MMX, l'opérande destination doit être un registre général de 32 bits.

L'instruction **PSHUFW** insère les mots de l'opérande source dans l'opérande destination à partir des emplacements spécifiés avec le troisième opérande. L'opérande destination doit être un registre MMX, l'opérande source peut être un emplacement mémoire 64 bits ou un registre MMX, le troisième opérande doit être une valeur immédiate de 8 bits sélectionnant les valeurs qui seront déplacées dans l'opérande destination, de la même manière que le troisième opérande de l'instruction SHUFPS.

L'instruction **MOVNTQ** déplace le quadruple-mot de l'opérande source vers la mémoire en utilisant un indice intemporel pour minimiser la pollution de l'antémémoire. L'opérande source doit être un registre MMX, l'opérande destination doit être un emplacement mémoire 64 bits. L'instruction **MOVNTPS** stocke les valeurs simple précision multiples du registre SSE dans la mémoire à l'aide d'un indice intemporel. L'opérande source doit être un registre SSE. L'opérande destination doit être un emplacement mémoire de 128 bits.

L'instruction **MASKMOVQ** stocke les octets sélectionnés du premier opérande dans un emplacement mémoire de 64 bits à l'aide d'un indice intemporel. Les deux opérandes doivent être des registres MMX, le second opérande sélectionne les octets de l'opérande source qui sont écrits en mémoire. L'emplacement mémoire est pointé par le registre DI (ou EDI) dans le segment sélectionné par DS.

Les instructions **PREFETCHT0**, **PREFETCHT1**, **PREFETCHT2** et **PREFETCHNTA** récupèrent la ligne de données de la mémoire qui contient l'octet spécifié par l'opérande à l'emplacement spécifié dans la hiérarchie. L'opérande doit être un emplacement mémoire de 8 bits.

L'instruction **SFENCE** effectue une opération de sérialisation sur toutes les instructions stockées dans la mémoire qui ont été émises avant cela. Cette instruction n'a pas d'opérande.

L'instruction **LDMXCSR** charge l'opérande de mémoire 32 bits dans le registre MXCSR. L'instruction **STMXCSR** stocke le contenu de MXCSR dans un opérande de mémoire 32 bits.

L'instruction **FXSAVE** enregistre l'état actuel de la FPU, du registre MXCSR et de tous les registres FPU et SSE dans un emplacement mémoire de 512 octets spécifié dans l'opérande destination. L'instruction **FXRSTOR** recharge les données précédemment stockées avec l'instruction FXSAVE à partir de l'emplacement de mémoire de 512 octets spécifié. L'opérande mémoire pour ces deux instructions doit être aligné sur une limite de 16 octets, il doit déclarer l'opérande sans taille spécifiée.

2.1.16 Instructions SSE2

The SSE2 extension introduces the operations on packed double precision floating point values, extends the syntax of MMX instructions, and adds also some new instructions.

Les instructions **MOVAPD** et **MOVUPD** transférent un opérande constitué d'un double quadruple-mot contenant des valeurs double-précision multiples de l'opérande source à l'opérande destination. Ces instructions sont analogues à **MOVAPS** et **MOVUPS** et ont les mêmes règles pour les opérandes.

L'instruction **MOVLPD** la valeur double précision entre la mémoire et le quadruple-mot bas du registre SSE. L'instruction **MOVHPD** déplace la valeur double précision entre la mémoire et le quadruple-mot haut du registre SSE. Ces instructions sont analogues à **MOVLPS** et **MOVHPS** et ont les mêmes règles pour les opérandes.

L'instruction **MOVMSKPD** transfère le bit le plus significatif de chacune des deux valeurs double précision dans le registre SSE en deux bits de poids faible d'un registre général. Cette instruction est analogue **MOVMSKPS** et a les mêmes règles pour les opérandes.

L'instruction **MOVSD** transfère une valeur double précision entre l'opérande source et destination (seul le mot quadruple bas est transféré). Au moins l'un des opérandes doit être un registre SSE, le second peut également être un registre SSE ou un emplacement mémoire de 64 bits.

Les opérations arithmétiques sur les valeurs en double précision sont : **ADDPD**, **ADDSD**, **SUBPD**, **SUBSD**, **MULPD**, **MULSD**, **DIVPD**, **DIVSD**, **SQRTPD**, **SQRTSD**, **MAXPD**, **MAXSD**, **MINPD**, **MINSD** et sont analogues aux opérations arithmétiques sur les valeurs simple précision décrites dans la section précédente. Lorsque le mnémonique se termine par PD au lieu de PS, l'opération est effectuée sur deux valeurs double précision multiples, mais les règles pour les opérandes sont les mêmes. Lorsque le mnémonique se termine par SD au lieu de SS, l'opérande source peut être un emplacement mémoire de 64 bits ou un registre SSE. L'opérande destination doit être un registre SSE et l'opération est effectuée sur des valeurs double-précision. Seuls les quadruples-mots de poids faible des registres SSE sont utilisés dans ce cas.

Les instructions **ANDPD**, **ANDNPD**, **ORPD** et **XORPD** effectuent des opérations logiques sur des valeurs doubles précision multiples. Elles sont analogues aux opérations logiques SSE sur des valeurs simple précision et ont les mêmes règles pour les opérandes.

L'instruction **CMPPD** compare les valeurs double-précision multiples et retourne un résultat de masque dans l'opérande destination. Cette instruction est analogue à **CMPPS** et a les mêmes règles pour les opérandes. L'instruction **CMPSD** effectue la même opération sur des valeurs double-précision. Seul le quadruple-mot de poids faible du registre destination est affecté. Dans ce cas, l'opérande source peut être une mémoire 64 bits ou un registre SSE. Une variante avec seulement deux opérandes est obtenue en attachant le mnémonique de condition du tableau 2.3 au mnémonique CMP puis en attachant le suffixe PD ou SD à la fin.

Les instructions **COMISD** et **UCOMISD** comparent les valeurs double-précision et positionnent les flags ZF, PF et CF pour afficher le résultat. L'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE.

L'instruction **SHUFPD** déplace l'une des deux valeurs double précision de l'opérande destination dans le quadruple-mot bas de l'opérande destination et l'une des deux valeurs de l'opérande source dans le quadruple-mot haut de l'opérande destination. Cette instruction est analogue à **SHUFPS** en adopte les règles pour l'opérande. Le bit 0 du troisième opérande sélectionne la valeur à déplacer de l'opérande destination, le bit 1 sélectionne la valeur à déplacer de l'opérande source, le reste des bits est réservé et doit être mis à zéro.

L'instruction **UNPCKHPD** exécute un démultiplage des quadruples-mots de poids fort à partir des opérandes source et de destination. L'instruction **UNPCKLPD** effectue un démultiplage des quadruples-mots bas des opérandes source et destination. Elles sont analogues à **UNPCKHPS** et **UNPCKLPS** et ont les mêmes règles pour les opérandes.

L'instruction **CVTPS2PD** convertit les deux valeurs à virgule flottante simple précision compactées en deux valeurs à virgule flottante double précision compactées, l'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement de mémoire 64 bits ou un registre SSE. L'instruction **CVTPD2PS** convertit les deux valeurs à virgule flottante double précision multiples en deux valeurs virgule flottante simple précision, l'opérande destination doit être un registre SSE. L'opérande source peut être un emplacement mémoire de 128

bits ou un registre SSE. L'instruction **CVTSS2SD** convertit la valeur à virgule flottante simple précision en valeur à virgule flottante double précision. L'opérande destination doit être un registre SSE. L'opérande source peut être un emplacement de mémoire 32 bits ou un registre SSE. L'instruction **CVTSD2SS** convertit la valeur à virgule flottante double précision en valeur à virgule flottante simple précision, l'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement de mémoire 64 bits ou un registre SSE.

L'instruction **CVTPI2PD** convertit deux doubles-mots entiers multiples en valeurs à virgule flottante double précision multiples. L'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement de mémoire 64 bits ou un registre MMX. L'instruction **CVTSI2SD** convertit un double-mot entier en une valeur virgule flottante double précision. L'opérande destination doit être un registre SSE, l'opérande source peut être un emplacement mémoire 32 bits ou un registre général 32 bits. L'instruction **CVTPD2PI** convertit des valeurs à virgule flottante double précision multiples en deux doubles-mots entiers multiples. L'opérande destination doit être un registre MMX, l'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE. L'instruction **CVTTPD2PI** exécute l'opération similaire, sauf que la troncature est utilisée pour arrondir les valeurs d'une source à des entiers, les règles pour les opérandes sont les mêmes. L'instruction **CVTSD2SI** convertit une valeur à virgule flottante double précision en un double-mot entier, l'opérande destination doit être un registre général de 32 bits, l'opérande source peut être un emplacement mémoire de 64 bits ou un registre SSE. L'instruction CVTTSD2SI exécute l'opération similaire, sauf que la troncature est utilisée pour arrondir une valeur source à un entier. Les règles pour les opérandes sont les mêmes.

Les instructions **CVTPS2DQ** et **CVTTPS2DQ** convertir des valeurs virgule flottante simple précision multiples en quatre doubles-mots entiers multiples, en les stockant dans l'opérande destination. Les instructions **CVTPD2DQ** et **CVTTPD2DQ** convertissent des valeurs à virgule flottante double précision multiples en deux doubles-mots entiers multiples, en stockant le résultat dans le quadruple-mot bas de l'opérande destination. L'instruction **CVTDQ2PS** convertit quatre doubles-mots entiers multiples en valeurs à virgule flottante simple précision multiples. Pour toutes ces instructions, l'opérande destination doit être un registre SSE. L'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE. L'instruction **CVTDQ2PD** convertit deux doubles-mots entiers multiples du quadruple-mot bas de l'opérande source en valeurs virgule flottante double précision multiples. La source peut être un emplacement de mémoire 64 bits ou un registre SSE. La destination doit être un registre SSE.

Les instructions **MOVDQA** et **MOVDQU** transférent un opérande sous forme de double quadruple-mot contenant des entiers multiples de l'opérande source à l'opérande destination. Au moins l'un des opérandes doit être un registre SSE. Le second peut également être un registre SSE ou un emplacement mémoire de 128 bits. Les opérandes mémoire pour l'instruction **MOVDQA** doivent être alignés sur une limite de 16 octets. Les opérandes pour l'instruction **MOVDQU** n'ont pas besoin d'être alignés.

L'instruction **MOVQ2DQ** déplace le contenu du registre source MMX vers le quadruple-mot bas du registre SSE destination. L'instruction **MOVDQ2Q** déplace le quadruple-mot bas du registre SSE source vers le registre MMX destination.

```
movq2dq xmm0, mm1    ; déplacement du registre MMX au registre SSE
movdq2q mm0, xmm1    ; déplacement du registre SSE au registre MMX
```

Toutes les instructions MMX fonctionnant sur les entiers multiples 64 bits (celles avec des mnémoniques commençant par P) sont étendues pour fonctionner sur des entiers multiples 128 bits situés dans les registres SSE. Une syntaxe supplémentaire pour ces instructions nécessite un registre SSE là où un registre MMX était nécessaire, et l'emplacement mémoire 128 bits ou registre SSE là où un emplacement mémoire 64 bits ou un registre MMX était nécessaire. L'exception est l'instruction **PSHUFW**, qui n'autorise pas la syntaxe étendue, mais a deux nouvelles variantes: **PSHUFHW** et **PSHUFLW**, qui n'autorisent que la syntaxe étendue, et effectuent la même opération que **PSHUFW** sur les mots quadruples hauts ou bas des opérandes respectivement. La nouvelle instruction **PSHUFD** est également introduite. Elle effectue la même opération que **PSHUFW**, mais sur des doubles-mots au lieu de mots et n'autorise que la syntaxe étendue.

```
psubb xmm0, [esi]    ; soustraction de 16 octets multiples
pextrw eax, xmm0,7   ; extraction du mot de plus fort poids dans eax
```

L'instruction **PADDQ** effectue l'addition de quadruples-mots multiples. L'instruction **PSUBQ** effectue la soustraction de quadruples-mots multiples. L'instruction **PMULUDQ** effectue la multiplication non signée des doubles-mots bas à partir de chaque quadruple-mot correspondant et renvoie les résultats en quadruples-mots multiples. Ces instructions suivent les mêmes règles pour les opérandes que les opérations MMX générales décrites au paragraphe 2.1.14.

Les instructions **PSLLDQ** et **PSRLDQ** effectuent un décalage logique à gauche ou à droite du double quadruple-mot dans l'opérande destination de la quantité d'octets spécifiée dans l'opérande source. L'opérande destination doit être un registre SSE. L'opérande source doit être une valeur immédiate de 8 bits.

L'instruction **PUNPCKHQDQ** entrelace le quadruple-mot haut de l'opérande source et le quadruple-mot haut de l'opérande destination et les écrit dans le registre SSE destination. L'instruction **PUNPCKLQDQ** entrelace le

quadruple-mot bas de l'opérande source et le quadruple-mot bas de l'opérande destination et les écrit dans le registre SSE destination. L'opérande source peut être un emplacement mémoire de 128 bits ou un registre SSE.

L'instruction **MOVNTDQ** stocke les données entières multiples du registre SSE dans la mémoire en utilisant un indice non temporel. L'opérande source doit être un registre SSE. L'opérande destination doit être un emplacement mémoire de 128 bits. L'instruction **MOVNTPD** stocke les valeurs double précision multiples du registre SSE dans la mémoire à l'aide d'un indice non-temporel. Les règles pour l'opérande sont les mêmes. L'instruction MOVNTI stocke l'entier d'un registre général en mémoire en utilisant un indice non-temporel. L'opérande source doit être un registre général de 32 bits. L'opérande destination doit être un emplacement mémoire de 32 bits. L'instruction **MASKMOVDQU** stocke les octets sélectionnés du premier opérande dans un emplacement mémoire de 128 bits en utilisant un indice non temporel. Les deux opérandes doivent être des registres SSE. Le deuxième opérande sélectionne les octets de l'opérande source qui sont écrits en mémoire. L'emplacement mémoire est pointé par le registre DI (ou EDI) dans le segment sélectionné par DS et n'a pas besoin d'être aligné.

L'instruction **CLFLUSH** écrit et invalide la ligne de cache associée à l'adresse de l'octet spécifié avec l'opérande, qui doit être un emplacement mémoire de 8 bits.

L'instruction **LFENCE** effectue une opération de sérialisation sur toutes les instructions de chargement à partir de la mémoire qui ont été émises avant cela. L'instruction **MFENCE** exécute une opération de sérialisation sur toutes les instructions accédant à la mémoire qui ont été émises avant elle, et combine donc les fonctions de **SFENCE** (décrites dans la section précédente) et les instructions LFENCE. Ces instructions n'ont pas d'opérande.

2.1.17 Instructions SSE3

Prescott technology introduced some new instructions to improve the performance of SSE and SSE2 - this extension is called SSE3.

L'instruction **FISTTP** se comporte comme l'instruction FISTP et accepte les mêmes opérandes. La seule différence est qu'elle utilise toujours la troncature, quel que soit le mode d'arrondi.

L'instruction **MOVSHDUP** charge dans l'opérande destination la valeur de 128 bits obtenue à partir de la valeur source de même taille en remplissant chaque quadruple-mot avec deux duplications de la valeur de son double-mot haut. L'instruction **MOVSLDUP** effectue la même action, sauf qu'elle duplique les valeurs des doubles-mots de faible poids. L'opérande destination doit être un registre SSE. L'opérande source peut être un registre SSE ou un emplacement mémoire 128 bits.

L'instruction **MOVDDUP** charge la valeur source 64 bits et la duplique en quatre mots haut et bas de l'opérande destination. L'opérande destination doit être un registre SSE, l'opérande source peut être un registre SSE ou un emplacement de mémoire 64 bits.

L'instruction **LDDQU** est fonctionnellement équivalente à MOVDQU avec la mémoire comme opérande source, mais elle peut améliorer les performances lorsque l'opérande source franchit une limite de ligne de cache. L'opérande destination doit être le registre SSE. L'opérande source doit être un emplacement mémoire de 128 bits.

L'instruction **ADDSUBPS** effectue une addition simple-précision des deuxième et quatrième paires et une soustraction simple précision des première et troisième paires de valeurs à virgule flottante dans les opérandes. L'instruction **ADDSUBPD** effectue une addition double précision de la deuxième paire et une soustraction double précision de la première paire de valeurs à virgule flottante dans l'opérande. L'instruction **HADDPS** effectue l'addition de deux valeurs simple-précision dans chaque quadruple-mot des opérandes source et destination, et stocke les résultats de cette addition horizontale de valeurs de l'opérande destination dans le quadruple-mot bas de l'opérande destination, et les résultats de l'opérande source dans le quadruple-mot haut de l'opérande destination. L'instruction **HADDPD** effectue l'addition de deux valeurs double-précision dans chaque opérande, et stocke le résultat de l'opérande destination dans le quadruple-mot bas de l'opérande destination, et le résultat de l'opérande source dans le quadruple-mot haut de l'opérande destination. Toutes ces instructions nécessitent que l'opérande destination soit un registre SSE. L'opérande source peut être un registre SSE ou un emplacement mémoire de 128 bits.

L'instruction **MONITOR** définit une plage d'adresses pour la surveillance des magasins de réécriture. Il faut que ses trois opérandes soient des registres EAX, ECX et EDX dans cet ordre. L'instruction **MWAIT** attend une réécriture dans la plage d'adresses définie par l'instruction MONITOR. Elle utilise deux opérandes avec des paramètres supplémentaires, le premier étant le registre EAX et le second, le registre ECX.

La fonctionnalité de SSE3 est encore étendue par l'ensemble d'instructions supplémentaires SSE3 (SSSE3). Ils suivent généralement les mêmes règles pour les opérandes que toutes les opérations MMX multiples par SSE.

Les instructions **PHADDW** et **PHADDD** effectuer l'addition horizontale des paires de valeurs adjacentes à la fois à partir de l'opérande source et de la destination, et stocker les sommes dans la destination (les sommes de l'opérande source vont dans la partie supérieure du registre destination). Ils fonctionnent respectivement sur des blocs de 16 bits ou 32 bits. L'instruction **PHADDSW** effectue la même opération sur les valeurs compactées 16 bits signées, mais le résultat de chaque addition est saturé. Les instructions **PHSUBW** et **PHSUBD** effectuent de manière analogue la soustraction horizontale de la valeur multiple 16 bits ou 32 bits, et **PHSUBSW** effectue la soustraction horizontale des valeurs multiples 16 bits signées avec saturation.

Les instructions **PABSB**, **PABSW** et **PABSD** calculent la valeur absolue de chaque valeur signée multiple dans l'opérande source et les stocke dans le registre destination. Ils opèrent respectivement sur des éléments 8 bits, 16 bits et 32 bits.

L'instruction **PMADDUBSW** multiplie les valeurs 8 bits signées de l'opérande source avec les valeurs 8 bits non signées correspondantes de l'opérande destination pour produire des valeurs 16 bits intermédiaires, et chaque paire adjacente de ces valeurs intermédiaires est ensuite ajoutée horizontalement et ces sommes 16 bits sont stockées dans l'opérande destination.

L'instruction **PMULHRSW** multiplie les entiers 16 bits correspondants de l'opérande source et de destination pour produire des valeurs 32 bits intermédiaires, et les 16 bits à côté du bit le plus élevé de chacune de ces valeurs sont ensuite arrondis et multiplexées dans l'opérande destination.

L'instruction **PSHUFB** mélange les octets de l'opérande destination en fonction du masque fourni par l'opérande source - chacun des octets de l'opérande source est un index de la position cible pour l'octet correspondant dans la destination.

Les instructions **PSIGNB**, **PSIGNW** et **PSIGND** effectuent l'opération sur 8, 16 ou 32 bits des nombres entiers dans l'instruction opérande destination, en fonction des signes des valeurs de la source. Si la valeur dans la source est négative, la valeur correspondante dans le registre de destination est annulée, si la valeur dans la source est positive, aucune opération n'est effectuée sur la valeur correspondante n'est effectuée, et si la valeur dans la source est zéro, la valeur dans la destination est également remis à zéro.

L'instruction **PALIGNR** ajoute l'opérande source à l'opérande destination pour former la valeur intermédiaire de taille double, puis extrait dans le registre de destination les 64 ou 128 bits alignés à droite sur le décalage d'octet spécifié par le troisième opérande, qui doit être une valeur immédiate 8 bits. C'est la seule instruction SSSE3 qui comporte trois arguments.

2.1.18 Instructions AMD 3DNow!

L'extension 3DNow! ajoute de nouvelles instructions MMX à celles décrites dans le paragraphe 2.1.14, et introduit une opération sur les valeurs à virgule flottante 64 bits, chacune constituée de deux valeurs virgule flottante simple précision.

Ces instructions suivent les mêmes règles que les opérations MMX générales. L'opérande destination doit être un registre MMX. L'opérande source peut être un registre MMX ou un emplacement mémoire 64 bits.

L'instruction **PAVGUSB** calcule les moyennes arrondies des octets non signés multiples. L'instruction **PMULHRW** effectue une multiplication signée des mots multiples, arrondit le mot haut de chaque double-mot de résultat et stocke ce double-mot dans l'opérande destination. L'instruction **PI2FD** convertit les doubles-mots entiers multiples en valeurs à virgule flottante multiples. L'instruction **PF2ID** convertit les valeurs à virgule flottante multiples en doubles-mots entiers multiples en utilisant la troncature. L'instruction **PI2FW** convertit les mots entiers multiples en valeurs à virgule flottante multiples. Seuls les mots bas de chaque double-mot de l'opérande source sont utilisés. L'instruction **PF2IW** convertit les valeurs à virgule flottante mutiples en mots entiers multiples, les résultats sont étendus aux doubles-mots en utilisant l'extension de signe. L'instruction **PFADD** ajoute des valeurs à virgule flottante compactées.

Les instructions **PFSUB** et **PFSUBR** soustraient des valeurs à virgule flottante multiples. La première soustrait les valeurs source des valeurs destination. La seconde soustrait les valeurs destination des valeurs source.

L'instruction **PFMUL** multiplie des valeurs à virgule flottante multiples. L'instruction **PFACC** ajoute les valeurs en virgule flottante basse et haute de l'opérande destination, en stockant le résultat dans le double-mot bas de destination, et ajoute les valeurs en virgule flottante basse et haute de l'opérande source, en stockant le résultat dans le double-mot haut destination. L'instruction **PFNACC** soustrait la valeur en virgule flottante haute de l'opérande destination de la valeur basse, en stockant le résultat dans le double-mot bas de destination, et soustrait la valeur en virgule flottante haute de l'opérande source de la valeur basse, en stockant le résultat dans le double-mot haut de destination. L'instruction **PFPNACC** soustrait la valeur en virgule flottante haute de l'opérande destination de la valeur basse, en stockant le résultat dans le double mot bas de destination, et ajoute les valeurs en virgule flottante basse et haute de l'opérande source, en stockant le résultat dans le double mot haut de destination.

Les instructions **PFMAX** et **PFMIN** calculent le maximum et le minimum des valeurs en virgule flottante. L'instruction PSWAPD inverse le double mot haut et bas de l'opérande source. L'instruction **PFRCP** renvoie une estimation des réciproques des valeurs à virgule flottante à partir de l'opérande source. L'instruction **PFRSQRT** renvoie une estimation des racines carrées réciproques des valeurs à virgule flottante à partir de l'opérande source, **PFRCPIT1** effectue la première étape de l'itération Newton-Raphson pour affiner l'approximation réciproque produite par l'instruction PFRCP, **PFRSQIT1** effectue la première étape de l'itération Newton-Raphson pour affiner la réciproque l'approximation de racine carrée produite par l'instruction **PFRSQRT**. L'instruction PFRCPIT2 effectue la deuxième étape finale de l'itération Newton-Raphson pour affiner l'approximation réciproque ou l'approximation réciproque de la racine carrée. Les instructions **PFCMPEQ**, **PFCMPGE** et **PFCMPGT** comparent les valeurs à virgule flottante multiples et mettent à zéro tous les bits ou remettent à zéro tous les bits

de l'élément de données correspondant dans l'opérande destination en fonction du résultat de la comparaison, vérifie d'abord si les valeurs sont égales, puis vérifie si la valeur de destination est supérieure ou égale à la valeur source, le troisième vérifie si la valeur de destination est supérieure à la valeur source.

Les instructions **PREFETCH** et **PREFETCHW** chargent la ligne de données de la mémoire qui contient l'octet spécifié avec l'opérande dans le cache de données. L'instruction **PREFETCHW** doit être utilisée lorsque les données de la ligne de cache sont censées être modifiées, sinon PREFETCH doit être utilisée. L'opérande doit être un emplacement mémoire de 8 bits.

L'instruction **FEMMS** effectue un effacement rapide de l'état MMX. Cette instruction n'a pas d'opérande.

2.1.19 Les instructions du mode long x86-64

Les architectures AMD64 et EM64T (nous utiliserons le nom commun x86-64 pour les deux) étendent le jeu d'instructions x86 pour le traitement 64 bits. Alors que les modes hérités et de compatibilité utilisent le même ensemble de registres et d'instructions, le nouveau mode long étend les opérations x86 à 64 bits et introduit plusieurs nouveaux registres. Vous pouvez activer la génération du code pour ce mode avec la directive **USE64**.

Chacun des registres à usage général est étendu à 64 bits et les huit nouveaux registres à usage général ainsi que huit nouveaux registres SSE sont ajoutés. Voir le tableau 2.4 pour le résumé des nouveaux registres (uniquement ceux qui ne figuraient pas dans le tableau 1.2). Les registres à usage général de taille plus petite sont les parties d'ordre inférieur des plus grandes. Vous pouvez toujours accéder aux registres AH, BH, CH et DH en mode long, mais vous ne pouvez pas les utiliser dans la même instruction avec l'un des nouveaux registres.

Tableau 2.4 – Nouveaux registres en mode long

Type	**Général**				**SSE**	**AVX**
Bits	8	16	32	64	128	256
				rax		
				rcx		
				rdx		
				rbx		
	spl			rsp		
	bpl			rbp		
	sil			rsi		
	dil			rdi		
	r8b	r8w	r8d	r8	xmm8	ymm8
	r9b	r9w	r9d	r9	xmm9	ymm9
	r10b	r10w	r10d	r10	xmm10	ymm10
	r11b	r11w	r11d	r11	xmm11	ymm11
	r12b	r12w	r12d	r12	xmm12	ymm12
	r13b	r13w	r13d	r13	xmm13	ymm13
	r14b	r14w	r14d	r14	xmm14	ymm14
	r15b	r15w	r15d	r15	xmm15	ymm15

En général, toute instruction de l'architecture x86, qui permettait des tailles d'opérandes de 16 bits ou 32 bits, en mode long autorise également les opérandes de 64 bits. Les registres 64 bits doivent être utilisés pour l'adressage en mode long. L'adressage 32 bits est également autorisé, mais il n'est pas possible d'utiliser les adresses basées sur des registres 16 bits. Voici les exemples de nouvelles opérations possibles en mode long sur l'exemple d'instruction `mov` :

```
mov rax, r8     ; transfert de registre général 64 bits
mov al, [rbx]   ; transfert en al d'un emplacement mémoire adressé par registre 64 bits
```

Le mode Long utilise également les adresses basées sur le pointeur d'instruction. Vous pouvez le spécifier manuellement avec le symbole de registre RIP spécial, mais un tel adressage est également généré automatiquement par *Flat Assembler*, car il n'y a pas d'adressage absolu 64 bits en mode long. Vous pouvez toujours forcer l'assembleur à utiliser l'adressage absolu 32 bits en plaçant le modificateur de taille DWORD pour l'adresse entre les crochets. Il y a aussi une exception, où l'adressage absolu 64 bits est possible, c'est l'instruction MOV avec l'un des opérandes étant le registre accumulateur et le second étant l'opérande mémoire. Pour forcer l'assembleur à utiliser l'adressage absolu 64 bits, utilisez l'opérateur de taille QWORD pour l'adresse entre crochets. Lorsqu'aucun opérateur de taille n'est appliqué à l'adresse, l'assembleur génère automatiquement la forme optimale.

```
mov [qword 0], rax          ; adressage absolu 64 bits
mov [dword 0], r15d         ; adressage absolu 32 bits
mov [0], rsi                ; adressage automatique relatif au RIP
mov [déchirure + 3], sil    ; adressage relatif au RIP manuel
```

De même, comme les opérandes immédiats pour les opérations 64 bits, seules les valeurs 32 bits signées sont possibles, la seule exception étant l'instruction `mov` dont l'opérande destination est un registre à usage général de 64 bits. Essayer de forcer le 64 bits immédiat avec n'importe quelle autre instruction provoquera une erreur.

Si une opération est effectuée sur les registres généraux 32 bits en mode long, les 32 bits supérieurs des registres 64 bits les contenant sont remplis de zéros. Ceci est différent des opérations sur les parties 16 bits ou 8 bits de ces registres, qui préservent les bits supérieurs.

Trois nouvelles instructions de conversion de type sont disponibles. L'instruction **CDQE** étend le signe du double-mot dans EAX en un quadruple-mot et stocke le résultat dans le registre RAX. L'instruction **CQO** étend le signe du quadruple-mot dans RAX en un double quadruple-mot et stocke les bits excédentaires dans le registre RDX. Ces instructions n'ont pas d'opérande. L'instruction **MOVSXD** étend le signe du double-mot de l'opérande source, qui est soit un registre 32 bits, soit la mémoire, en un opérande destination 64 bits, qui doit être un registre. Aucune instruction analogue n'est nécessaire pour l'extension à zéro, car elle est effectuée automatiquement par toutes les opérations sur les registres 32 bits, comme indiqué dans le paragraphe précédent. Et les instructions **MOVZX** et **MOVSX**, conformément à la règle générale, peuvent être utilisées avec l'opérande destination 64 bits, permettant l'extension des valeurs d'octet ou de mot en quadruples-mots.

Toutes les instructions arithmétiques et logiques binaires sont aménagées pour autoriser les opérandes 64 bits en mode long. L'utilisation d'instructions arithmétiques décimales en mode long est interdite.

Les opérations de pile, comme **PUSH** et **POP** en mode long, utilisent par défaut des opérandes 64 bits et il n'est pas possible d'utiliser des opérandes 32 bits avec elles. Les **PUSHA** et **POPA** sont interdits en mode long.

Les sauts de proximité indirects et les appels en mode long utilisent par défaut des opérandes 64 bits et il n'est pas possible d'utiliser les opérandes 32 bits avec eux. D'autre part, les sauts et les appels distants indirects autorisent tous les opérandes permis par l'architecture x86 et l'opérande de mémoire 80 bits est autorisé (bien que seul EM64T semble implémenter cette variante), les huit premiers octets définissant le décalage et deux derniers octets spécifiant le sélecteur. Les sauts distants directs et les appels ne sont pas autorisés en mode long.

Les instructions d'E/S **IN**, **OUT**, **INS** et **OUTS** sont les instructions exceptionnelles qui ne sont pas habilitées à accepter les opérandes quadruple-mot en mode long. Mais toutes les autres opérations de chaîne le sont, en revanche. Et il y a de nouvelles formes courtes **MOVSQ**, **CMPSQ**, **SCASQ**, **LODSQ** et **STOSQ** mises en place pour les variantes des opérations de chaîne pour les éléments de chaîne 64 bits. Les registres RSI et RDI sont utilisés par défaut pour adresser les éléments de chaîne.

Les instructions **LFS**, **LGS** et **LSS** sont aménagées pour accepter l'opérande de mémoire source de 80 bits avec un registre de destination de 64 bits (bien que seul EM64T semble implémenter une telle variante). Les instructions **LDS** et **LES** sont interdites en mode long.

Les instructions système telles que **LGDT** qui nécessitait un opérande de mémoire de 48 bits, en mode long, nécessite un opérande de mémoire de 80 bits.

L'instruction **CMPXCHG16B** est l'équivalent 64 bits de l'instruction **CMPXCHG8B**. Elle utilise un opérande mémoire en forme de double quadruple mot et des registres 64 bits pour effectuer l'opération analogue.

Les instructions **FXSAVE64** et **FXRSTOR64** sont de nouvelles variantes des instructions **FXSAVE** et **FXRSTOR**, disponibles uniquement en mode long, qui utilisent un format de zone de stockage différent afin de stocker certains pointeurs en pleine taille 64 bits.

L'instruction **SWAPGS** est la nouvelle instruction, qui permute le contenu du registre GS et du registre spécifique au modèle KernelGSbase (adresse MSR 0C0000102h).

Les instructions **SYSCALL** et **SYSRET** est la paire d'instructions nouvelles qui fournissent la fonctionnalité similaire à **SYSENTER** et **SYSEXIT** en mode long, où cette dernière paire est interdite. Les mnémoniques **SYSEXITQ** et **SYSRETQ** fournissent les versions 64 bits des instructions **SYSEXIT** et **SYSRET**.

Les mnémoniques **RDMSRQ** et **WRMSRQ** sont les variantes 64 bits des instructions **RDMSR** et **WRMSR**.

2.1.20 Instructions SSE4

Il existe en fait trois ensembles d'instructions différents sous le nom SSE4. Intel a conçu deux d'entre eux, SSE4.1 et SSE4.2, le dernier étendant le premier à l'ensemble SSE4 d'Intel. D'autre part, l'implémentation par AMD ne comprend que quelques instructions de cet ensemble, mais contient également des instructions supplémentaires, appelées ensemble SSE4a.

Les instructions SSE4.1 suivent la plupart du temps les mêmes règles pour les opérandes, que les opérations SSE de base, de sorte qu'elles nécessitent que l'opérande destination soit le registre SSE et l'opérande source soit un emplacement mémoire 128 bits ou un registre SSE, et certaines opérations nécessitent un troisième opérande,, sous forme de valeur immédiate 8 bits.

L'instruction **PMULLD** effectue une multiplication signée des doubles-mots multiples et stocke les doubles-mots bas des résultats dans l'opérande destination. L'instruction **PMULDQ** multiplie des doubles-mots entiers signés multiples dans les éléments pairs (référence de base zéro) du premier opérande source avec des doubles-mots

entiers signés multiples dans les éléments correspondants du deuxième opérande source et stocke les résultats de doubles-mots signés multiples dans l'opérande destination..

Les instructions **PMINSB** et **PMAXSB** retournent le minimum ou les valeurs maximales d'octets multiples signés. Les instructions **PMINUW** et **PMAXUW** renvoient les valeurs minimales et maximales des mots non signés multiples. Les instructions **PMINUD**, **PMAXUD**, **PMINSD** et **PMAXSD** renvoient les valeurs minimales et maximales de doubles-mots non signés ou signés multiples. Ces instructions complètent les instructions calculant le minimum ou le maximum multiple introduites par SSE.

L'instruction **PTEST** définit le flag ZF sur un lorsque le résultat du AND bit-à-bit des deux opérandes est égal à zéro, et remet à zéro le ZF dans le cas contraire. Elle définit également le flag CF sur un, lorsque le résultat du AND bit-à-bit de l'opérande destination avec le NOT bit-à-bit de l'opérande source est égal à zéro, et met à zéro le CF dans le cas contraire. L'instruction **PCMPEQQ** compare les quadruples-mots multiples sur le critère d''égalité et remplit les éléments correspondants de l'opérande destination avec des uns ou des zéros, selon le résultat de la comparaison.

L'instruction **PACKUSDW** convertit les doubles-mots signés multiples à la fois des opérandes source et destination en mots non signés appliquant la saturation, et stocke les huit valeurs de mot résultantes dans le registre destination.

L'instruction **PHMINPOSUW** trouve la valeur minimale du mot non signé dans l'opérande source et la place dans le mot le plus bas de l'opérande destination, en mettant à zéro les bits supérieurs restants de la destination.

Les instructions **ROUNDPS**, **ROUNDSS**, **ROUNDPD** et **ROUNDSD** effectuent l'arrondi des valeurs en virgule flottante simple ou double précision multiples ou individuels, en utilisant le mode d'arrondi spécifié dans le troisième opérande.

```
roundsd xmm0, xmm1,0011b   ; arrondi vers zéro
```

L'instruction **DPPS** calcule le produit scalaire de valeurs à virgule flottante simple précision multiples, c'est-à-dire qu'il multiplie les paires de valeurs correspondantes de l'opérande source et de l'opérande destination, puis additionne les produits. Les quatre bits supérieurs du troisième opérande immédiat de 8 bits contrôlent quels produits sont calculés et amenés à la somme, et les quatre bits inférieurs contrôlent, dans quels éléments de destination le produit scalaire résultant est copié (les autres éléments sont remplis de zéro) . L'instruction **DPPD** calcule le produit scalaire des valeurs à virgule flottante double précision multiples. Les bits 4 et 5 du troisième opérande contrôlent, quels produits sont calculés et ajoutés, et les bits 0 et 1 de cette valeur contrôlent quels éléments du registre de destination doivent être remplis avec le résultat.

L'instruction **MPSADBW** calcule plusieurs sommes de différences absolues d'octets non signés. Le troisième opérande contrôle, avec une valeur dans les bits 0-1, lequel des blocs de quatre octets de l'opérande source est utilisé pour calculer les différences absolues, et avec une valeur dans le bit 2, auquel des deux premiers blocs de quatre octets de destination l'opérande commence à calculer plusieurs sommes. La somme est calculée à partir de quatre différences absolues entre les octets non signés correspondants dans le bloc source et de destination, et chaque somme suivante est calculée de la même manière, mais en prenant les quatre octets de la destination à la position un octet après la position du bloc précédent. Les quatre octets de la source restent les mêmes à chaque fois. De cette façon, huit sommes de différences absolues sont calculées et stockées sous forme de valeurs de mot condensées dans l'opérande destination. De la même manière que l'instruction **ROUNDPS**.

Les instructions **BLENDPS**, **BLENDVPS**, **BLENDPD** et **BLENDVPD** copient conditionnellement les valeurs de source de l'opérande dans l'opérande destination, en fonction des bits du masque fourni par le troisième opérande. Si un bit de masque est défini, l'élément correspondant de la source est copié au même endroit dans la destination. Sinon cette position est la destination est laissée inchangée. Les règles pour les deux premiers opérandes sont les mêmes que pour les instructions SSE générales. Les instructions **BLENDPS** et **BLENDPD** nécessitent que le troisième opérande soit immédiat sur 8 bits, et elles fonctionnent respectivement sur des valeurs de simple ou double précision. Les instructions **BLENDVPS** et **BLENDVPD** imposent que le troisième opérande soit le registre XMM0.

```
blendvps xmm3, xmm7, xmm0   ; mélange selon le masque
```

L'instruction **PBLENDW** copie conditionnellement les éléments de mot de l'opérande source dans la destination, en fonction des bits de masque fournis par le troisième opérande, qui doit être une valeur immédiate de 8 bits. L'instruction **PBLENDVB** copie conditionnellement les éléments octets des opérandes source dans la destination, en fonction du masque défini par le troisième opérande, qui doit être le registre XMM0. Ces instructions suivent les mêmes règles pour les opérandes que les instructions **BLENDPS** et **BLENDVPS**, respectivement.

L'instruction **INSERTPS** insère une valeur à virgule flottante simple précision prise à partir de la position dans l'opérande source spécifiée par les bits 6-7 du troisième opérande dans l'emplacement du registre destination sélectionné par les bits 4-5 du troisième opérande. De plus, les quatre bits de poids faible du troisième opérande contrôlent, quels éléments du registre de destination seront mis à zéro. Les deux premiers opérandes suivent les mêmes règles que pour l'opération SSE générale. Le troisième opérande doit être immédiat de 8 bits.

L'instruction **EXTRACTPS** extrait une valeur à virgule flottante simple précision prise à partir de l'emplacement de l'opérande source spécifié par deux bits bas du troisième opérande, et la stocke dans l'opérande destination. La destination peut être une valeur de mémoire 32 bits ou un registre à usage général. L'opérande source doit être un registre SSE et le troisième opérande doit être une valeur immédiate 8 bits.

```
extractps edx, xmm3,3     ; extraction de la valeur la plus élevée
```

Les instructions **PINSRB**, **PINSRD** et **PINSRQ** copient un octet, un double-mot ou un quadruple-mot de l'opérande source vers l'emplacement de l'opérande destination désigné par le troisième opérande. L'opérande destination doit être un registre SSE. L'opérande source peut être un emplacement mémoire de taille appropriée ou un registre à usage général 32 bits (mais un registre à usage général 64 bits pour **PINSRQ**, qui n'est disponible qu'en mode long). Le troisième opérande doit être une valeur immédiate 8 bits. Ces instructions complètent l'instruction **PINSRW** opérant sur le registre destination SSE, qui a été introduit par SSE2.

```
pinsrd xmm4, eax, 1   ; insertion d'un double mot en deuxième position
```

Les instructions **PEXTRB**, **PEXTRW**, **PEXTRD** et **PEXTRQ** copient un octet, mot, double-mot ou quadruple-mot de l'emplacement dans l'opérande source spécifié par le troisième opérande, dans la destination. L'opérande source doit être un registre SSE. Le troisième opérande doit être une valeur immédiate 8 bits et l'opérande destination peut être un emplacement mémoire de taille appropriée, ou un registre à usage général 32 bits (mais un registre à usage général 64 bits pour **PEXTRQ**, qui est disponible uniquement en mode long). L'instruction **PEXTRW** avec le registre SSE comme source a déjà été introduite par SSE2, mais SSE4 l'étend pour autoriser l'opérande mémoire comme destination.

```
pextrw [ebx], xmm3,7    ; extraction du mot le plus élevé en mémoire
```

Les instructions **PMOVSXBW** et **PMOVZXBW** effectuer une extension de signe ou une extension zéro de huit valeurs d'octets à partir de l'opérande source en valeurs de mot condensées dans l'opérande destination, qui doit être le registre SSE. La source peut être une mémoire 64 bits ou un registre SSE - lorsqu'elle est enregistrée, seule sa partie basse est utilisée. Les instructions **PMOVSXBD** et **PMOVZXBD** effectuent une extension de signe ou une extension zéro des valeurs à quatre octets de l'opérande source en valeurs de double-mot compactées dans l'opérande destination, la source peut être une mémoire de 32 bits ou un registre SSE. Les instructions **PMOVSXBQ** et **PMOVZXBQ** effectuent une extension de signe ou une extension zéro des deux valeurs d'octet de l'opérande source en valeurs de quadruple-mot multiples dans l'opérande destination, la source peut être une mémoire de 16 bits ou un registre SSE. Les instructions **PMOVSXWD** et **PMOVZXWD** effectuent une extension de signe ou une extension à zéro des quatre valeurs de mot de l'opérande source en doubles-mots multiples dans l'opérande destination. La source peut être un emplacement mémoire 64 bits ou un registre SSE. Les instructions **PMOVSXWQ** et **PMOVZXWQ** effectuent une extension de signe ou une extension à zéro des deux valeurs de mot de l'opérande source en quadruples-mots multiples dans l'opérande destination, la source peut être un emplacement mémoire de 32 bits ou un registre SSE. Les instructions **PMOVSXDQ** et **PMOVZXDQ** effectuent une extension de signe ou une extension à zéro des deux valeurs de double-mot de l'opérande source en quadruples-mots multiples dans l'opérande destination. La source peut être une mémoire de 64 bits ou un registre SSE.

```
pmovzxbq xmm0, mot [si]  ; octets étendus à zéro à quatre mots
pmovsxwq xmm0, xmm1      ; mots en extension de signe à quatre mots
```

L'instruction **MOVNTDQA** charge le quadruple-mot de l'opérande source vers la destination à l'aide d'un indice non temporel. L'opérande destination doit être un registre SSE et l'opérande source doit être un emplacement de mémoire de 128 bits.

Le SSE4.2, décrit ci-dessous, ajoute non seulement de nouvelles opérations sur les registres SSE, mais introduit également des instructions complètement nouvelles fonctionnant uniquement sur des registres à usage général.

L'instruction **PCMPISTRI** compare deux chaînes terminées par zéro (longueur implicite) fournies par les opérandes source et destination et génère un index stocké dans ECX. L'instruction **PCMPISTRM** effectue la même comparaison et génère un masque stocké dans XMM0. L'instruction **PCMPESTRI** compare deux chaînes de longueur explicite, avec une longueur fournie par EAX pour l'opérande destination et par EDX pour l'opérande source, et génère un index stocké dans ECX; L'instruction **PCMPESTRM** effectue la même comparaison et génère un masque stocké dans XMM0. Les opérandes source et destination suivent les mêmes règles que pour les instructions SSE générales. Le troisième opérande doit être une valeur immédiate de 8 bits déterminant les détails de l'opération effectuée. Reportez-vous à la documentation Intel pour plus d'informations sur ces détails.

L'instruction **PCMPGTQ** compare des quadruples-mots multiples et remplit les éléments correspondants de l'opérande destination avec des uns ou des zéros, selon que la valeur de la destination est supérieure ou non à celle de la source. Cette instruction suit les mêmes règles pour les opérandes que l'instruction **PCMPEQQ**.

L'instruction **CRC32** accumule une valeur CRC32 pour l'opérande source en commençant par la valeur initiale fournie par l'opérande destination et stocke le résultat dans la destination. Sauf en mode long, l'opérande destination doit être un registre à usage général de 32 bits, et l'opérande source peut être un registre d'octet, de mot, de double-mot ou un emplacement mémoire. En mode long, l'opérande destination peut également être un registre à usage général de 64 bits, et l'opérande source dans ce cas peut être un registre d'octets, de quadruple-mot ou un emplacement de mémoire.

```
crc32 eax, dl            ; accumulation du CRC32 sur la valeur d'octet
crc32 eax, mot [ebx]     ; accumulation du CRC32 sur la valeur du mot
crc32 rax, qword [rbx]   ; accumulation du CRC32 sur la valeur quadruple mot
```

L'instruction **POPCNT** calcule le nombre de bits définis dans l'opérande source, qui peut être un registre à usage général ou un emplacement mémoire de 16 bits, 32 bits ou 64 bits, et stocke ce compte dans l'opérande destination, qui doit être le même registre size comme opérande source. La variante 64 bits n'est disponible qu'en mode long.

```
popcnt ecx, eax    ; nombre de bits mis à 1
```

L'extension SSE4a, qui comprend également l'instruction POPCNT introduite par SSE4.2, ajoute en même temps l'instruction **LZCNT**, qui suit la même syntaxe, et calcule le décompte des bits zéro en tête dans l'opérande source (si l'opérande source est tous à zéro bits, le nombre total de bits dans l'opérande source est stocké dans la destination).

L'instruction **EXTRQ** extraire la séquence de bits du quadruple-mot bas du registre SSE fourni comme premier opérande et les stocke à l'extrémité inférieure de ce registre, en remplissant les bits restants dans le quadruple-mot bas avec des zéros. La position de la chaîne de bits et sa longueur peuvent être fournies avec deux valeurs immédiates de 8 bits comme deuxième et troisième opérande, ou par le registre SSE comme deuxième opérande (et il n'y a pas de troisième opérande dans ce cas), qui doit contenir la valeur de position dans bits 8-13 et longueur de la chaîne de bits en bits 0-5.

```
extrq xmm0, 8, 7     ; extraction de 8 bits de la position 7
extrq xmm0, xmm5     ; extraction des bits définis par le registre
```

L'instruction **INSERTQ** écrit la séquence de bits du quadruple-mot bas de l'opérande source dans la position spécifiée dans le quadruple-mot bas de l'opérande destination, laissant les autres bits du quadruple-mot bas de destination intacts. La position où les bits doivent être écrits et la longueur de la chaîne de bits peut être fournie avec deux valeurs immédiates de 8 bits comme troisième et quatrième opérande, ou par les champs de bits dans l'opérande source (et il n'y a que deux opérandes dans ce cas), qui doit contenir la valeur de position dans les bits 72-77 et la longueur de la chaîne de bits dans les bits 64-69.

```
insertq xmm1, xmm0, 4, 2    ; insértion de 4 bits à la position 2
insertq xmm1, xmm0          ; insértion de bits définis par le registre
```

Les instructions **MOVNTSS** et **MOVNTSD** stockent une valeur virgule flottante simple ou double précision du registre SSE source dans un emplacement mémoire destination de 32 bits ou 64 bits respectivement, en utilisant une indication non temporelle.

2.1.21 Instructions AVX

Les extensions vectorielles avancées (*Advanced Vector Extensions*) introduisent des instructions qui sont de nouvelles variantes d'instructions SSE, avec un nouveau schéma de codage qui permet une syntaxe étendue ayant un opérande destination séparé de tous les opérandes source. Il introduit également des registres AVX 256 bits, qui étendent les anciens registres SSE 128 bits. Toute instruction AVX qui met un résultat dans le registre SSE, met à zéro les bits dans la partie haute du registre AVX qui le contient.

On obtient le mnémonique de la version AVX d'une instruction SSE en faisant précéder le nom de l'instruction SSE par `V`. Pour toute instruction arithmétique SSE qui avait un opérande destination également utilisé comme l'une des valeurs source, la variante AVX a une nouvelle syntaxe avec trois opérandes : la destination et deux sources. La destination et la première source peuvent être des registres SSE. La seconde source peut être un registre ou une mémoire SSE. Si l'opération est effectuée sur une seule paire de valeurs, les bits restants du premier registre SSE source sont copiés dans le registre destination.

```
vsubss xmm0, xmm2, xmm3          ; soustraction de 2 flottants 32 bits
vmulsd xmm0, xmm7, qword [esi]   ; multiplication de 2 flottants 64 bits
```

Dans le cas d'opérations multiples, chaque instruction peut également fonctionner sur la taille de données de 256 bits lorsque les registres AVX sont spécifiés au lieu des registres SSE, et la taille de l'opérande mémoire est également doublée alors.

```
vaddps ymm1, ymm5, yword [esi]   ; 8 sommes de paires de flottants 32 bits
```

Les instructions qui opèrent sur des types entiers multiples (en particulier celles qui auparavant avaient été promues de MMX à SSE) ont également acquis la nouvelle syntaxe avec trois opérandes, mais elles ne sont autorisées à fonctionner que sur des types multiples 128 bits et ne peuvent donc pas utiliser les registres AVX entiers.

```
vpavgw xmm3, xmm0, xmm2  ; moyenne d'entiers 16 bits
vpslld xmm1, xmm0, 1     ; décalage de mots doubles vers la gauche
```

Si la version SSE de l'instruction avait une syntaxe avec trois opérandes, le troisième étant une valeur immédiate, la version AVX de cette instruction prend quatre opérandes, le dernier restant immédiat.

```
vshufpd ymm0, ymm1, ymm2, 10010011b  ; mélange de flottants 64 bits
```

```
vpalignr xmm0, xmm4, xmm2,3          ; extraction de la valeur alignée sur l'octet
```

La promotion vers la nouvelle syntaxe selon les règles décrites ci-dessus a été appliquée à toutes les instructions des extensions SSE jusqu'à SSE4, avec les exceptions décrites ci-dessous.

L'instruction **VDPPD** a une syntaxe étendue à quatre opérandes, mais n'a pas de version 256 bits.

Il y a quelques instructions, telles que **VSQRTPD**, **VSQRTPS**, **VRCPPS** et **VRSQRTPS**, qui peuvent fonctionner sur la taille des données 256 bits, tout en conservant une syntaxe avec seulement deux opérandes, car elles utilisent des données à partir d'une seule source :

```
vsqrtpd ymm1, ymm0       ; mettre des racines carrées dans un autre registre
```

De la même manière, les instructions **VROUNDPD** et **VROUNDPS** ont conservé la syntaxe avec trois opérandes, le dernier étant une valeur immédiate.

```
vroundps ymm0, ymm1,0011b      ; arrondit vers zéro
```

De plus, certaines des opérations sur les entiers multiples ont conservé leur syntaxe à deux ou trois opérandes tout en étant promues vers la version AVX. Dans ce cas, ces instructions suivent exactement les mêmes règles pour les opérandes que leurs homologues SSE (puisque les opérations sur les entiers multiples n'ont pas de variante 256 bits dans l'extension AVX). Celles-ci comprennent **VPCMPESTRI**, **VPCMPESTRM**, **VPCMPISTRI**, **VPCMPISTRM**, **VPHMINPOSUW**, **VPSHUFD**, **VPSHUFHW** et **VPSHUFLW**. Et il y a plus d' instructions que dans les versions AVX conservent exactement la même syntaxe pour les opérandes que celles de la SSE, sans options supplémentaires : **VCOMISS**, **VCOMISD**, **VCVTSS2SI**, **VCVTSD2SI**, **VCVTTSS2SI**, **VCVTTSD2SI**, **VEXTRACTPS**, **VPEXTRB**, **VPEXTRW**, **VPEXTRD**, **VPEXTRQ**, **VMOVD**, **VMOVQ**, **VMOVNTDQA**, **VMASKMOVDQU**, **VPMOVMSKB**,**VPMOVSXBW**, **VPMOVSXBD**, **VPMOVSXBQ**, **VPMOVSXWD**, **VPMOVSXWQ**, **VPMOVSXDQ**, **VPMOVZXBW**, **VPMOVZXBD**, **VPMOVZXBQ**, **VPMOVZXWD**, **VPMOVZXWQ** et **VPMOVZXDQ**.

Les instructions de déplacement et de conversion ont principalement été promues pour autoriser des opérandes de taille 256 bits en plus de la variante 128 bits avec une syntaxe identique à celle de la version SSE de la même instruction. Chacun des **VCVTDQ2PS**, **VCVTPS2DQ** et **VCVTTPS2DQ**, **VMOVAPS**, **VMOVAPD**, **VMOVUPS**, **VMOVUPD**, **VMOVDQA**, **VMOVDQU**, **VLDDQU**, **VMOVNTPS**, **VMOVNTPD**, **VMOVNTDQ**, **VMOVSLDUP**, **VMOVSHDUP**, **VMOVMSKPS** et **VMOVMSKPD** hérite de la syntaxe 128 bits de SSE sans aucune modification, et permet également une nouvelle forme avec opérandes 256 bits au lieu de 128 bits.

```
vmovups [edi],ymm6     ; stockage de données 256-bits non-alignées
```

L'instruction **VMOVDDUP** a la même syntaxe 128 bits que sa version SSE, et il a également une version 256 bits, qui stocke les doublons du quadruple-mot le plus bas de l'opérande source dans la moitié inférieure de l'opérande destination et dans la moitié supérieure de la destination les doublons du quadruple-mot bas de la moitié supérieure de la source. Les opérandes source et destination doivent alors être des valeurs de 256 bits.

Les instructions **VMOVLHPS** et **VMOVHLPS** n'ont que des versions 128 bits, et chacun prend trois opérandes, qui doivent tous être des registres SSE. L'instruction **VMOVLHPS** copie deux valeurs de précision simple du quadruple-mot bas du deuxième registre source vers le quadruple-mot haut du registre de destination, et copie le quadruple-mot bas du premier registre source dans le quadruple-mot bas du registre de destination. L'instruction VMOVHLPS copie deux valeurs de précision simple du quadruple-mot haut du deuxième registre source vers le quadruple-mot bas du registre de destination, et copie le quadruple-mot haut du premier registre source dans le quadruple-mot haut du registre de destination.

Les instructions **VMOVLPS**, **VMOVHPS**, **VMOVLPD** et **VMOVHPD** ont seulement des versions 128 bits et leur syntaxe varie selon que la mémoire opérande est une destination ou la source. Lorsque la mémoire est la destination, la syntaxe est identique à celle de l'instruction SSE équivalente, et lorsque la mémoire est la source, l'instruction nécessite trois opérandes, les deux premiers étant des registres SSE et le troisième une mémoire de 64 bits. La valeur mise en destination est alors la valeur copiée à partir de la première source avec le quadruple-mot bas ou haut remplacé par la valeur de la seconde source (l'opérande mémoire).

```
vmovhps [esi], xmm7          ; stockage de la moitié supérieure en mémoire
vmovlps xmm0, xmm7, [ebx]    ; partie de la mémoire de poids faible, reste du registre
```

Les instructions **VCVTSS2SD**, **VCVTSD2SS**, **VCVTSI2SS** et **VCVTSI2D** utilisent une syntaxe à trois opérandes, où la source et la destination première sont toujours des registres SSE, et la deuxième source suit les mêmes règles et la source dans la syntaxe d'instruction SSE équivalent. La valeur stockée dans la destination est alors la valeur copiée de la première source avec l'élément de données le plus bas remplacé par le résultat de la conversion.

```
vcvtsi2sd xmm4, xmm4, ecx    ; entier 32 bits à flottant 64 bits
vcvtsi2ss xmm0, xmm0, rax    ; entier 64 bits à flottant 32 bits
```

Les instructions **VCVTDQ2PD** et **VCVTPS2PD** autorisent la même syntaxe que leurs équivalentes SSE, ainsi que les nouvelles variantes avec registre AVX comme destination et registre SSE ou mémoire 128 bits comme source. De manière analogue **VCVTPD2DQ**, **VCVTTPD2DQ** et **VCVTPD2PS**, en plus de la variante avec une

syntaxe identique à la version SSE, autorisent une variante avec registre SSE comme destination et registre AVX ou mémoire 256 bits comme source.

Les instructions **VINSERTPS**, **VPINSRB**, **VPINSRW**, **VPINSRD**, **VPINSRQ** et **VPBLENDW** utilisent une syntaxe avec quatre opérandes, où destination et première source doivent être des registres SSE. Le troisième et le quatrième opérande suivent les mêmes règles que les deuxième et troisième opérandes dans la syntaxe d'instruction SSE équivalente. La valeur stockée dans la destination est la valeur copiée à partir de la première source avec certains éléments de données remplacés par des valeurs extraites de la deuxième source, de manière analogue à l'opération de l'instruction SSE correspondante.

```
vpinsrd xmm0, xmm0, eax, 3  ; insertion d'un double mot
```

Les instructions **VBLENDVPS**, **VBLENDVPD** et **VPBLENDVB** utilisent une nouvelle syntaxe avec quatre opérandes de registre : destination, deux sources et un masque, où la deuxième source peut également être un opérande mémoire. Les instructions **VBLENDVPS** et **VBLENDVPD** ont une variante de 256 bits, où les opérandes sont des registres AVX ou une mémoire de 256 bits, ainsi qu'une variante de 128 bits, dont les opérandes sont des registres SSE ou une mémoire de 128 bits. L'instruction **VPBLENDVB** n'a qu'une variante de 128 bits. La valeur stockée dans la destination est la valeur copiée à partir de la première source avec certains éléments de données remplacés, selon le masque, par des valeurs de la deuxième source.

```
vblendvps ymm3, ymm1, ymm2, ymm7  ; mélange selon le masque
```

L'instruction **VPTEST** autorise la même syntaxe que sa version SSE et dispose également d'une version 256 bits, avec les deux opérandes doublés de taille. Il existe également deux nouvelles instructions, **VTESTPS** et **VTESTPD**, qui effectuent des tests analogues, mais uniquement des bits de signe des valeurs de simple précision ou de double précision correspondantes, et définissent le ZF et le CF en conséquence. Ils suivent les mêmes règles de syntaxe que **VPTEST**.

```
vptest ymm0, yword [ebx]  ; test des valeurs 256 bits
vtestpd xmm0, xmm1        ; test des bits de signe des flottants 64 bits
```

Les instructions **VBROADCASTSS**, **VBROADCASTSD** et **VBROADCASTF128** sont de nouvelles instructions, qui diffusent l'élément de données défini par l'opérande source dans tous les éléments de taille correspondante dans le registre de destination. L'instruction **VBROADCASTSS** nécessite que la source soit une mémoire 32 bits et que la destination soit un registre SSE ou AVX. L'instruction **VBROADCASTSD** nécessite une mémoire 64 bits comme source et un registre AVX comme destination. L'instruction **VBROADCASTF128** nécessite une mémoire de 128 bits comme source et un registre AVX comme destination.

```
vbroadcastss ymm0, dword [eax]  ; obtention de 8 exemplaires d'une valeur
```

L'instruction **VINSERTF128** est la nouvelle instruction, qui prend quatre opérandes. La destination et la première source doivent être des registres AVX, la deuxième source peut être un registre SSE ou un emplacement mémoire de 128 bits et le quatrième opérande doit être une valeur immédiate. Il stocke dans la destination la valeur obtenue en prenant le contenu de la première source et en remplaçant l'une de ses unités de 128 bits par la valeur de la deuxième source. Le bit le plus bas du quatrième opérande spécifie à quelle position ce remplacement est effectué (0 ou 1).

L'instruction **VEXTRACTF128** est la nouvelle instruction avec trois opérandes. La destination doit être un registre SSE ou un emplacement mémoire de 128 bits, la source doit être un registre AVX et le troisième opérande doit être une valeur immédiate. Il extrait dans la destination l'une des unités 128 bits de la source. Le bit le plus bas du troisième opérande spécifie l'unité extraite.

Les instructions **VMASKMOVPS** et **VMASKMOVPD** sont les nouvelles instructions à trois opérandes qui stockent sélectivement dans la destination les éléments de la seconde source en fonction des bits de signe des éléments correspondants de la première source. Ces instructions peuvent fonctionner sur des données 128 bits (registres SSE) ou sur des données 256 bits (registres AVX). La destination ou la seconde source doit être un emplacement mémoire de taille appropriée, les deux autres opérandes doivent être des registres.

```
vmaskmovps [edi], xmm0, xmm5   ; stockage sous condition
vmaskmovpd ymm5, ymm0, [esi]   ; chargement conditionnel
```

Les instructions **VPERMILPD** et **VPERMILPS** sont les nouvelles instructions avec trois opérandes qui permutent les valeurs de la première source en fonction des champs de contrôle de la seconde source et placent le résultat dans l'opérande destination. Il permet d'utiliser soit trois registres SSE, soit trois registres AVX comme opérandes, la deuxième source peut être une mémoire de taille égale aux registres utilisés. En variante, la seconde source peut être une valeur immédiate et ensuite la première source peut être un emplacement mémoire de taille égale au registre de destination.

L'instruction **VPERM2F128** est la nouvelle instruction avec quatre opérandes, qui sélectionne des blocs de 128 bits de données à virgule flottante de la première et de la deuxième source en fonction des champs de bits du quatrième opérande, et les stocke dans la destination. La destination et la première source doivent être des registres AVX, la deuxième source peut être un registre AVX ou une zone de mémoire 256 bits et le quatrième opérande doit être une valeur immédiate.

```
vperm2f128 ymm0, ymm6, ymm7,12h    ; permute des blocs de 128 bits
```

L'instruction **VZEROALL** met tous les registres AVX à zéro. L'instruction **VZEROUPPER** définit les parties supérieures de 128 bits de tous les registres AVX à zéro, laissant les registres SSE intacts. Ces nouvelles instructions ne prennent aucun opérande.

Les instructions **VLDMXCSR** et **VSTMXCSR** sont respectivement les variantes AVX des instructions **LDMXCSR** et **STMXCSR**. Les règles de leurs opérandes restent inchangées.

2.1.22 AVX2 instructions

L'extension AVX2 permet à toutes les instructions AVX fonctionnant sur des entiers multiples d'utiliser des types de données 256 bits et introduit également de nouvelles instructions.

Les instructions AVX qui fonctionnent sur des entiers multiples et n'avaient que des variantes de 128 bits, ont été complétées par des variantes de 256 bits et ainsi leurs règles de syntaxe sont devenues analogues aux instructions AVX fonctionnant sur des types à virgule flottante multiples.

```
vpsubb ymm0, ymm0, [esi]    ; soustraction de 32 octets multiples
vpavgw ymm3, ymm0, ymm2     ; moyenne d'entiers 16 bits
```

Cependant, certaines instructions n'ont pas été équipées des variantes 256 bits. Les instructions **VPCMPESTRI**, **VPCMPESTRM**, **VPCMPISTRI**, **VPCMPISTRM**, **VPEXTRB**, **VPEXTRW**, **VPEXTRD**, **VPEXTRQ**, **VPINSRB**, **VPINSRW**, **VPINSRD**, **VPINSRQ** et **VPHMINPOSUW** ne sont pas affectés par AVX2 et autorisent uniquement les opérandes 128 bits.

Les instructions de décalage multiples, qui permettaient au troisième opérande spécifiant la quantité d'être un registre SSE ou un emplacement mémoire de 128 bits, utilisent les mêmes règles pour le troisième opérande dans leur variante de 256 bits.

```
vpsllw ymm2, ymm2, xmm4          ; décalage des mots vers la gauche
vpsrad ymm0, ymm3, xword [ebx]   ; décalage des mots doubles vers la droite
```

Il existe également de nouvelles instructions de décalage compact avec la syntaxe AVX standard à trois opérandes, qui décalent chaque élément de la première source de la quantité spécifiée dans l'élément correspondant de la deuxième source et stockent les résultats dans la destination. L'instruction VPSLLVD décale les éléments 32 bits vers la gauche. L'instruction **VPSLLVQ** décale les éléments 64 bits vers la gauche. L'instruction **VPSRLVD** décale les éléments 32 bits vers la droite logiquement. L'instruction **VPSRLVQ** décale les éléments 64 bits vers la droite logiquement et **VPSRAVD** décalent arithmétiquement les éléments 32 bits vers la droite.

Les instructions d'extension de signe et d'extension zéro, qui dans les versions AVX permettaient à l'opérande source d'être un registre SSE ou une mémoire de taille spécifique, dans la nouvelle variante de 256 bits, il faut une mémoire de cette taille doublée ou un registre SSE comme source et un registre AVX comme destination.

```
vpmovzxbq ymm0, dword [esi] ; octets vers quatre mots
```

L'instruction **VMOVNTDQA** a également été mise à niveau avec une variante 256 bits, ce qui permet de transférer une valeur de 256 bits de la mémoire vers le registre AVX. Elle a besoin d'une adresse mémoire qui soit alignée sur 32 octets.

Les instructions **VPMASKMOVD** et **VPMASKMOVQ** sont les nouvelles instructions avec une syntaxe identique à **VMASKMOVPS** ou **VMASKMOVPD**. Elles effectuent une opération analogue sur des valeurs multiples de 32 ou 64 bits.

Les instructions **VINSERTI128**, **VEXTRACTI128**, **VBROADCASTI128** et **VPERM2I128** sont de nouvelles instructions de syntaxe identique à **VINSERTF128**, **VEXTRACTF128**, **VBROADCASTF128** et **VPERM2F128** respectivement. Elles effectuent des opérations analogues sur des blocs de 128 bits de données de nombres entiers.

Les instructions **VBROADCASTSS** et **VBROADCASTSD** ont été étendues pour permettre le registre SSE comme opérande source (qui, dans AVX, ne pouvait être qu'un emplacement mémoire).

Les nouvelles instructions **VPBROADCASTB**, **VPBROADCASTW**, **VPBROADCASTD** et **VPBROADCASTQ** diffusent l'octet, le mot, le double-mot ou le quadruple-mot de l'opérande de source vers tous les éléments de taille correspondante dans le registre de destination. L'opérande destination peut être un registre SSE ou AVX, et l'opérande source peut être un registre SSE ou une mémoire de taille égale à la taille de l'élément de données.

```
vpbroadcastb ymm0, octet [ebx]; obtention de 32 octets identiques
```

Les instructions **VPERMD** et **VPERMPS** sont de nouvelles instructions à trois opérandes, qui utilisent chaque élément 32 bits de la première source comme index d'élément dans la seconde source qui est copié dans la destination à la position correspondant à l'élément contenant l'index. La destination et la première source doivent être des registres AVX, et la deuxième source peut être un registre AVX ou une mémoire 256 bits.

Les instructions **VPERMQ** et **VPERMPD** sont de nouvelles instructions à trois opérandes, qui utilisent des index de 2 bits à partir de la valeur immédiate spécifiée comme troisième opérande pour déterminer quel élément de la mémoire source à une position donnée dans la destination. La destination doit être un registre AVX. La source

peut être un registre AVX ou une mémoire de 256 bits et le troisième opérande doit être une valeur immédiate de 8 bits.

La famille de nouvelles instructions effectuant une opération GATHER (rassemblement) a une syntaxe particulière, car dans leur opérande mémoire, ces instructions utilisent un mode d'adressage qui leur est propre. La base d'adresse peut être un registre à usage général de 32 ou 64 bits (ce dernier uniquement en mode long), et l'index (éventuellement multiplié par la valeur d'échelle, comme dans l'adressage standard) est spécifié par le registre SSE ou AVX. . Il est possible d'utiliser uniquement un index sans base et tout déplacement numérique peut être ajouté à l'adresse. Chacune de ces instructions prend trois opérandes. Le premier opérande est le registre destination, le deuxième opérande est la mémoire adressée avec un index vectoriel et le troisième opérande est le registre contenant un masque. Le bit le plus significatif de chaque élément du masque détermine si une valeur sera chargée depuis la mémoire dans l'élément correspondant en destination. L'adresse de chaque élément à charger est déterminée en utilisant l'élément correspondant du registre d'index dans l'opérande mémoire pour calculer l'adresse finale avec une base et un déplacement donnés. Lorsque le registre d'index contient moins d'éléments que les registres de destination et de masque, les éléments de destination supérieurs sont mis à zéro. Une fois la valeur chargée avec succès, l'élément correspondant dans le registre de masque est mis à zéro. La destination, l'index et le masque doivent tous être des registres distincts. Il n'est pas permis d'utiliser le même registre dans deux rôles différents.

L'instruction **VGATHERDPS** charge les valeurs virgule flottante simple précision adressées par des index 32 bits. La destination, l'index et le masque doivent tous être des registres du même type, SSE ou AVX. Les données adressées par l'opérande mémoire ont une taille de 32 bits.

```
vgatherdps xmm0, [eax + xmm1], xmm3       ; rassemblement de 4 flottants
vgatherdps ymm0, [ebx + ymm7 * 4], ymm3   ; rassemblement de 8 huit flottants
```

L'instruction **VGATHERQPS** charge les valeurs virgule flottante simple précision adressées par des index 64 bits. La destination et le masque doivent toujours être des registres SSE, tandis que le registre d'index peut être un registre SSE ou AVX. Les données adressées par l'opérande mémoire ont une taille de 32 bits.

```
vgatherqps xmm0, [xmm2], xmm3         ; rassemblement de 2 flottants
vgatherqps xmm0, [ymm2 + 64], xmm3    ; rassemblement de 4 flottants
```

L'instruction **VGATHERDPD** charge des valeurs virgule flottante double précision adressées par des index 32 bits. Le registre d'index doit toujours être le registre SSE, la destination et le masque doivent être deux registres du même type, SSE ou AVX. Les données adressées par l'opérande mémoire ont une taille de 64 bits.

```
vgatherdpd xmm0, [ebp + xmm1], xmm3   ; rassemblement de 2 double-précision
vgatherdpd ymm0, [xmm3 * 8], ymm5     ; rassemblement de 4 double-précision
```

L'instruction **VGATHERQPD** charge des valeurs virgule flottante double précision adressées par des index 64 bits. La destination, l'index et le masque doivent tous être des registres du même type, SSE ou AVX. Les données adressées par l'opérande mémoire ont une taille de 64 bits.

Les instructions **VPGATHERDD** et **VPGATHERQD** chargent des valeurs 32 bits adressées par des index 32 bits ou 64 bits. Elles suivent les mêmes règles que **VGATHERDPS** et **VGATHERQPS** respectivement.

Les instructions **VPGATHERDQ** et **VPGATHERQQ** chargent des valeurs 64 bits adressées par des index 32 bits ou 64 bits. Elles suivent les mêmes règles que **VGATHERDPD** et **VGATHERQPD** respectivement.

2.1.23 Jeux d'instructions auxiliaires de calcul

Il existe un certain nombre d'extensions de jeu d'instructions supplémentaires liées à AVX. Elles introduisent de nouvelles instructions vectorielles (et parfois aussi leurs équivalents SSE qui utilisent le codage d'instructions classique), et même de nouvelles instructions opérant sur des registres généraux qui utilisent le codage de type AVX permettant la syntaxe étendue avec des opérandes de destination et de source séparés. La prise en charge du processeur pour chacun de ces jeux d'instructions doit être déterminée séparément.

Les instructions **AESENC** et **AESENCLAST** effectuent un seul tour de cryptage AES sur les données de la première source avec la clé fournie dans la seconde source et stockent le résultat dans la destination. La destination et la première source sont des registres SSE. La seconde source peut être un registre SSE ou une mémoire de 128 bits.

Les variantes AVX de ces instructions, **VAESENC** et **VAESENCLAST**, utilisent la syntaxe à trois opérandes, alors que la version de type SSE n'en a que deux, le premier opérande étant à la fois la destination et la première source.

Les instructions **AESDEC** et **AESDECLAST** effectuent un seul tour de décryptage AES sur les données de la première source avec la clé fournie dans la seconde source. Les règles de syntaxe pour ces instructions et leurs variantes AVX sont les mêmes que pour **AESENC**.

L'instruction **AESIMC** effectue la transformation *InvMixColumns* de l'opérande source et stocke le résultat dans la destination. Les instructions **AESIMC** et **VAESIMC** n'utilisent que deux opérandes, la destination étant le registre SSE et la source étant le registre SSE ou un emplacement mémoire de 128 bits.

L'instruction **AESKEYGENASSIST** est une instruction d'aide pour générer la clé de tour. Elle a besoin de trois opérandes : la destination étant le registre SSE, la source étant le registre SSE ou la mémoire 128 bits et le troisième opérande étant une valeur immédiate 8 bits. La version AVX de cette instruction utilise la même syntaxe.

L'extension **CLMUL** n'affiche qu'une seule instruction **PCLMULQDQ**, ainsi que sa version AVX. Cette instruction effectue une multiplication sans retenue de deux valeurs 64 bits sélectionnées parmi la première et la deuxième source en fonction des champs de bits en valeur immédiate. La destination et la première source sont des registres SSE. La seconde source est un registre SSE ou une mémoire de 128 bits. Une valeur immédiate est fournie comme dernier opérande. L'instruction **VPCLMULQDQ** requiert quatre opérandes, tandis que PCLMULQDQ prend seulement trois opérandes, le premier servant à la fois le rôle de destination et de première source.

L'extension **FMA** (Fused Multiply-Add) introduit des instructions AVX supplémentaires qui effectuent la multiplication et la sommation en une seule opération. Chacune prend trois opérandes, le premier servant à la fois de destination et de première source, les suivantes étant la deuxième et la troisième source. Le mnémonique de l'instruction FMA est obtenu en faisant suivre le préfixe VF : d'abord de M ou NM pour sélectionner si le résultat de la multiplication doit être pris tel quel ou annulé, puis soit ADD ou SUB pour sélectionner si une troisième valeur sera additionnée ou soustraite au produit, puis soit 132, 213 ou 231 pour sélectionner quels opérandes sources sont multipliés et lesquels sont ajoutés ou soustraits, et enfin le type de données sur lesquelles l'instruction opère, soit PS, PD, SS ou SD. Comme c'était le cas avec les instructions SSE promues en AVX, les instructions fonctionnant sur des valeurs à virgule flottante multiples autorisent une syntaxe de 128 ou 256 bits, dans l'ancien, tous les opérandes sont des registres SSE, mais le troisième peut également être une mémoire de 128 bits, en dernier, les opérandes sont des registres AVX et le troisième peut également être une mémoire 256 bits. Les instructions qui calculent un seul résultat en virgule flottante ont besoin d'opérandes pour être des registres SSE, et le troisième opérande peut également être une mémoire, soit 32 bits pour une simple-précision, soit 64 bits pour une double-précision.

```
vfmsub231ps ymm1, ymm2, ymm3      ; multiplication puis soustraction
vfnmadd132sd xmm0, xmm5, [ebx]    ; multiplication, négation et addition
```

En plus des instructions créées selon la règle édictée ci-dessus, il existe des familles d'instructions avec des mnémoniques commençant par **VFMADDSUB** ou **VFMSUBADD**, suivis de 132, 213 ou 231 puis de PS ou PD (l'opération doit toujours être sur des valeurs multiples dans ce cas). Elles s'additionnent ou se soustraient au résultat de la multiplication en fonction de la position de la valeur dans les données multiples : les instructions du groupe VFMADDSUB s'ajoutent lorsque la position est impaire et se soustraient lorsque la position est paire ; les instructions du groupe VFMSUBADD s'additionnent lorsque la position est paire et se soustraient lorsque la position est impaire. Les règles pour les opérandes sont les mêmes que pour les autres instructions FMA.

Les instructions FMA4 sont similaires à FMA, mais utilisent une syntaxe avec quatre opérandes et permettent ainsi à la destination d'être différente de toutes les sources. Leurs mnémoniques sont identiques aux instructions FMA avec le 132, 213 ou 231 non-présent, car le fait d'avoir des opérandes de destination séparés rend une telle sélection d'opérandes superflue. La multiplication est toujours effectuée sur les valeurs de la première et de la deuxième source, puis la valeur de la troisième source est ajoutée ou soustraite. La deuxième ou la troisième source peut être un opérande de mémoire, et les règles concernant les tailles des opérandes sont les mêmes que pour les instructions FMA.

```
vfmaddpd ymm0, ymm1, [esi], ymm2    ; multiplication puis addition
vfmsubss xmm0, xmm1, xmm2, [ebx]    ; multiplication puis addition
```

L'extension F16C se compose de deux instructions, **VCVTPS2PH** et **VCVTPH2PS**, qui convertissent les valeurs à virgule flottante entre une simple-précision et demi-précision (le format à virgule flottante 16 bits). L'instruction VCVTPS2PH gère trois opérandes : la destination, la source et les modes d'arrondi. Le troisième opérande est toujours immédiat, la source est un registre SSE ou AVX contenant des valeurs en simple-précision, et la destination est un registre ou une mémoire SSE. La taille de la mémoire est de 64 bits lorsque la source est le registre SSE et de 128 bits lorsque la source est un registre AVX. L'instruction VCVTPH2PS prend deux opérandes, la destination qui peut être un registre SSE ou AVX, et la source qui est un registre SSE ou la mémoire avec une taille de moitié de celle de l'opérande destination.

L'extension AMD **XOP** introduit un certain nombre de nouvelles instructions vectorielles avec un codage et une syntaxe analogues aux instructions AVX. Les instructions **VFRCZPS**, **VFRCZSS**, **VFRCZPD** et **VFRCZSD** extraient les parties fractionnaires de valeurs simple ou double-précision. Elles prennent tous deux opérandes. Les opérations multiples autorisent les registres SSE ou AVX comme destination. Pour les deux autres il doit s'agir d'un registre SSE. La source peut être un registre du même type que la destination, ou une mémoire de taille appropriée (256 bits pour la destination étant un registre AVX, 128 bits pour une opération groupée avec destination étant un registre SSE, 64 bits pour une opération sur une valeur double-précision isolée et 32 bits pour un fonctionnement sur une valeur simple-précision unique).

```
vfrczps ymm0, [esi]       ; chargement des parties fractionnaires
```

L'instruction **VPCMOV** copie les bits de la première ou de la deuxième source vers la destination en fonction des valeurs des bits correspondants dans le quatrième opérande (le sélecteur). Si le bit dans le sélecteur est activé, le bit correspondant de la première source est copié dans la même position dans la destination, sinon le bit de la seconde source est copié. La deuxième source ou le sélecteur peut être un emplacement de mémoire, 128 bits ou 256 bits selon que les registres SSE ou AVX sont spécifiés comme les autres opérandes.

```
vpcmov xmm0, xmm1, xmm2, [ebx]     ; sélecteur en mémoire
vpcmov ymm0, ymm5, [esi], ymm2     ; source en mémoire
```

La famille d'instructions de comparaison multiples prend quatre opérandes : la destination et la première source sont un registre SSE, la deuxième source est un registre SSE ou la mémoire 128 bits et le quatrième opérande est une valeur immédiate définissant le type de comparaison. Le mnémonique ou l'instruction est créé en ajoutant au préfixe **VPCOM**, soit B ou UB pour comparer des octets signés ou non signés, soit W ou UW pour comparer des mots signés ou non signés, soit D ou UD pour comparer des mots doubles signés ou non signés, soit Q ou UQ pour comparer des quadruples-mots signés ou non signés. Les valeurs respectives de la première et de la seconde source sont comparées et l'élément de données correspondant dans la destination est mis à tous ou à tous les zéros en fonction du résultat de la comparaison. Le quatrième opérande doit spécifier l'un des huit types de comparaison (tableau 2.5). Toutes ces instructions ont également des variantes avec seulement trois opérandes et le type de comparaison codé dans le nom de l'instruction en insérant le mnémonique de comparaison après VPCOM.

```
vpcomb xmm0, xmm1, xmm2,4      ; test d'égalité d'octets
vpcomgew xmm0, xmm1, [ebx]     ; comparaison de mots signés
```

Tableau 2.5 – Comparaisons XOP (AMD)

Code	Mnémonique	Description	
0	lt	moins que	less than
1	le	inférieur ou égal	less than or equal
2	gt	plus grand que	greater than
3	ge	plus grand que ou égal	greater than or equal
4	eq	égal	equal
5	neq	inégal	not equal
6	false	faux	false
7	true	vrai	true

Les instructions **VPERMIL2PS** et **VPERMIL2PD** définissent les éléments du registre de destination à zéro ou à une valeur sélectionnée parmi la première ou la seconde source en fonction des champs de bits correspondants du quatrième opérande (le sélecteur) et de la valeur immédiate fournie dans le cinquième opérande. Reportez-vous aux manuels AMD pour une explication détaillée de l'opération effectuée par ces instructions. Chacun des quatre premiers opérandes peut être un registre, et la deuxième source ou le sélecteur peuvent être un emplacement mémoire, 128 bits ou 256 bits selon que les registres SSE ou AVX sont utilisés pour les autres opérandes.

```
vpermil2ps ymm0, ymm3, ymm7, ymm2,0   ; permute à partir de deux sources
```

L'instruction **VPHADDBW** additionne des paires d'octets signés adjacents pour former des valeurs 16 bits et les stocke aux mêmes positions dans la destination. L'instruction **VPHADDUBW** fait de même mais traite les octets comme non signés. Les instructions **VPHADDBD** et **VPHADDUBD** additionnent tous les octets (signés ou non) de chaque bloc de quatre octets sous forme de résultats 32 bits. Les instructions **VPHADDBQ** et **VPHADDUBQ** additionnent tous les octets de chaque bloc de huit octets sous forme de résultats 64 bits. Les instructions **VPHADDWD** et **VPHADDUWD** additionnent des paires de mots sous forme de résultats 32 bits.

Les instructions **VPHADDWQ** et **VPHADDUWQ** additionnent tous les mots de chaque bloc de quatre mots pour former des résultats 64 bits. Les instructions **VPHADDDQ** et **VPHADDUDQ** additionnent des paires de doubles-mots pour former des résultats 64 bits. L'instruction **VPHSUBBW** soustrait, dans chaque bloc de deux octets, l'octet à la position supérieure de celui à la position inférieure, et stocke le résultat sous la forme d'une valeur signée de 16 bits à la position correspondante dans la destination. L'instruction **VPHSUBWD** soustrait dans chaque bloc de deux mots le mot en position supérieure de celui en position inférieure et produit des résultats signés de 32 bits. L'instruction **VPHSUBDQ** soustrait, dans chaque bloc de deux doubles-mots, celui en position supérieure de celui en position inférieure et produit des résultats 64 bits signés. Chacune de ces instructions prend deux opérandes, la destination étant le registre SSE et la source étant le registre SSE ou la mémoire 128 bits.

```
vphadduwq xmm0, xmm1          ; somme des quadruplés de mots
```

Les instructions **VPMACSWW** et **VPMACSSWW** multiplient les valeurs 16 bits signées correspondantes de la première et de la deuxième source, puis ajoutent les produits aux valeurs parallèles de la troisième source, puis **VPMACSWW** prend les 16 bits les plus bas du résultat et **VPMACSSWW** sature le résultat jusqu'à la valeur de 16 bits, et ils stockent les résultats finaux 16 bits dans la destination. Les instructions **VPMACSDD** et

VPMACSSDD effectuent l'opération analogue sur des valeurs 32 bits. Les instructions **VPMACSWD** et **VPMACSSWD** effectuent le même calcul uniquement sur les valeurs 16 bits de poids faible de chaque bloc 32 bits et forment les résultats 32 bits. Les instructions **VPMACSDQL** et **VPMACSSDQL** effectuent une telle opération sur les faibles valeurs de 32 bits à partir de chaque bloc de 64 bits et forme les résultats de 64 bits, tandis que **VPMACSDQH** et **VPMACSSDQH** font la même chose sur les valeurs hautes de 32 bits à partir de chaque bloc de 64 bits, formant également des résultats 64 bits. Les instructions **VPMADCSWD** et **VPMADCSSWD** multiplient les valeurs 16 bits signées correspondantes à partir de la première et de la deuxième source, puis additionnent les quatre produits et ajoutent cette somme à chaque élément de 16 bits de la troisième source, en stockant le résultat tronqué ou saturé dans la destination. Toutes ces instructions prennent quatre opérandes. La deuxième source peut être une mémoire de 128 bits ou un registre SSE. Tous les autres opérandes doivent être des registres SSE.

```
vpmacsdd xmm6, xmm1, [ebx], xmm6  ; accumulation du produit
```

L'instruction **VPPERM** sélectionne les octets de la première et de la deuxième source, applique éventuellement une transformation distincte à chacun d'eux et les stocke dans la destination. Les champs de bits du quatrième opérande (le sélecteur) spécifient pour chaque position dans la destination quel octet de quelle source est prélevé et quelle opération lui est appliquée avant d'y être stocké. Reportez-vous aux manuels AMD pour obtenir des informations détaillées sur ces champs de bits. Cette instruction prend quatre opérandes. La deuxième source ou le sélecteur peuvent être une mémoire de 128 bits (ou ils peuvent être des registres SSE tous les deux). Tous les autres opérandes doivent être des registres SSE.

Les instructions **VPSHLB**, **VPSHLW**, **VPSHLD** et **VPSHLQ** effectuent un décalage logique respectivement d'octets, de mots, de doubles ou quadruples-mots. La quantité de bits à décaler est spécifiée pour chaque élément séparément par l'octet signé placé à la position correspondante dans le troisième opérande. La source contenant les éléments à décaler est fournie en tant que deuxième opérande. Le deuxième ou le troisième opérande peut être une mémoire de 128 bits (ou il peut s'agir de registres SSE tous les deux) et les autres opérandes doivent être des registres SSE.

```
vpshld xmm3, xmm1, [ebx]    ; décalage des octets de xmm1
```

Les instructions **VPSHAB**, **VPSHAW**, **VPSHAD** et **VPSHAQ** effectuent un décalage arithmétique, respectivement sur des octets, des mots, des doubles ou quadruples-mots. Ces instructions suivent les mêmes règles que les décalages logiques décrits ci-dessus.

Les instructions **VPROTB**, **VPROTW**, **VPROTD** et **VPROTQ** effectuent une rotation de bits, respectivement sur des octets, des mots, des doubles ou quadruples-mots. Elles suivent les mêmes règles que les décalages, mais permettent en outre au troisième opérande d'être une valeur immédiate, auquel cas la même quantité de rotation est spécifiée pour tous les éléments de la source.

```
vprotb xmm0, [esi], 3       ; rotation des octets vers la gauche
```

L'extension MOVBE introduit une seule nouvelle instruction, **MOVBE** qui inverse l'ordre des octets de la valeur de la source avant de stocker cette opération dans la destination. Elle peut donc être utilisée pour charger et stocker des valeurs du format *little endian* au format *big endian* et réciproquement. Il faut deux opérandes. La destination ou la source doit être une mémoire 16 bits, 32 bits ou 64 bits (cette dernière n'étant autorisée qu'en mode long). L'autre opérande doit être un registre général de la même taille.

L'extension BMI, composée de deux sous-ensembles - BMI1 et BMI2, introduit de nouvelles instructions opérant sur des registres généraux, qui utilisent le même codage que les instructions AVX et permettent ainsi la syntaxe étendue. Toutes ces instructions utilisent des opérandes 32 bits, et en mode long. Elles autorisent également les formes avec des opérandes 64 bits.

L'instruction **ANDN** calcule le AND bit-à-bit de la deuxième source avec les bits inversés de la première source et stocke le résultat dans la destination. La destination et la première source doivent être des registres généraux. La deuxième source peut être un registre général ou une mémoire.

```
andn edx, eax, [ebx]        ; eax inversé multiplié par bits avec mémoire
```

L'instruction **BEXTR** extrait de la première source la séquence de bits utilisant un index et une longueur spécifiés par des champs de bits dans le deuxième opérande source et la stocke dans la destination. Les 8 bits les plus bas de la seconde source spécifient la position de la séquence de bits à extraire et les 8 bits suivants de la seconde source spécifient la longueur de la séquence. La première source peut être un registre général ou une mémoire, les deux autres opérandes doivent être des registres généraux.

```
bextr eax, [esi], ecx       ; extraction du champ de bits de la mémoire
```

L'instruction **BLSI** extrait le bit à 1 le plus bas de la source, mettant à zéro tous les autres bits de la destination. La destination doit être un registre général. La source peut être un registre général ou une mémoire.

```
blsi rax, r11               ; isolement du bit à 1 le plus bas
```

L'instruction **BLSMSK** met à 1 tous les bits de la destination jusqu'au bit le plus bas de la source, y compris ce bit. L'instruction **BLSR** copie tous les bits de la source vers la destination à l'exception du bit le plus bas défini, qui est remplacé par zéro. Ces instructions suivent les mêmes règles pour les opérandes que BLSI.

L'instruction **TZCNT** compte le nombre de bits de fin à zéro, c'est-à-dire les bits à zéro, du plus bas jusqu'au premier bit à 1 en remontant vers les bits de plus fort poids de la valeur source. Cette instruction est analogue LZCNT et suit les mêmes règles pour les opérandes, elle a donc également une version 16 bits, contrairement aux autres instructions BMI.

L'instruction **BZHI** est l'instruction BMI2, qui copie les bits de la première source à la destination, mettant à zéro tous les bits à partir de la position spécifiée par la deuxième source. Elle suit les mêmes règles pour les opérandes que BEXTR.

L'instruction **PEXT** utilise un masque dans le second opérande source pour sélectionner des bits à partir des premiers opérandes et place les bits sélectionnés comme une séquence continue dans la destination. L'instruction **PDEP** effectue l'opération inverse : elle prend une séquence de bits de la première source et les place consécutivement aux positions où les bits de la seconde source sont définis, mettant à zéro tous les autres bits de la destination. Ces instructions BMI2 suivent les mêmes règles pour les opérandes que l'instruction **ANDN**.

L'instruction **MULX** est une instruction BMI2 qui effectue une multiplication non signée de la valeur du registre EDX ou RDX (selon la taille des opérandes spécifiés) par la valeur du troisième opérande, et stocke la moitié inférieure du résultat dans le deuxième opérande et la moitié supérieure du résultat dans le premier opérande, et ceci, sans affecter les flags. Le troisième opérande peut être un registre général ou une mémoire. Les deux opérandes destination doivent être des registres généraux.

```
mulx edx, eax, ecx          ; multiplication de edx par ecx en edx:eax
```

Les instructions **SHLX**, **SHRX** et **SARX** sont des instructions BMI2, qui effectuent des décalages logiques ou arithmétiques de la valeur de la première source par la quantité de bits spécifiée par seconde source, et stockent le résultat dans la destination sans affecter les flags. Ces instructions ont les mêmes règles pour les opérandes que pour l'instruction BZHI.

L'instruction **RORX** est une instruction BMI2 qui fait pivoter à droite la valeur de l'opérande source de la quantité constante spécifiée dans le troisième opérande et stocke le résultat dans la destination sans affecter les flags. L'opérande destination doit être un registre général, l'opérande source peut être un registre général ou une mémoire, et le troisième opérande doit être une valeur immédiate.

```
rorx eax, edx, 7            ; rotation sans affectation des flags
```

Le TBM est une extension conçue par AMD pour compléter l'ensemble BMI. L'instruction **BEXTR** est étendue avec une nouvelle forme, dans laquelle la deuxième source est une valeur immédiate de 32 bits. L'instruction **BLSIC** est une nouvelle instruction qui effectue la même opération que BLSI, mais avec les bits de résultat inversés. Il utilise les mêmes règles pour les opérandes que BLSI. L'instruction **BLSFILL** est une nouvelle instruction, qui prend la valeur de la source, définit tous les bits sous le bit le plus bas et stocke le résultat dans la destination, elle utilise également les mêmes règles pour les opérandes que BLSI.

Les instructions **BLCI**, **BLCIC**, **BLCS**, **BLCMSK** et **BLCFILL** sont des instructions analogues à BLSI, BLSIC, BLSR, BLSMSK et BLSFILL respectivement, mais qui exécutent les versions de bit inversé des mêmes opérations. Elles suivent les mêmes règles pour les opérandes que les instructions qu'ils reflètent.

L'instruction **TZMSK** trouve le bit à 1 le plus bas dans la valeur de l'opérande source, définit tous les bits inférieurs à 1 et tous les bits restants à zéro, puis écrit le résultat dans la destination. L'instruction **T1MSKC** trouve le bit zéro le moins significatif dans la valeur de l'opérande source, définit les bits inférieurs à zéro et tous les autres bits à 1 et écrit le résultat dans la destination. Ces instructions ont les mêmes règles pour les opérandes que BLSI.

2.1.24 Instructions AVX-512

L'AVX-512 introduit des registres vectoriels de 512 bits, qui étendent les registres de 256 bits utilisés par AVX et AVX2. Il étend également la quantité de registres vectoriels de 16 à 32, avec les registres supplémentaires `zmm16` à `zmm31`, incluant leur partie basse de 256 bits `ymm16` à `ymm31` et leur partie basse de 128 bits `xmm16` à `xmm31`. Ces registres supplémentaires ne sont accessibles qu'en mode long.

Tableau 2.6 – Nouveaux registres disponibles en mode long avec AVX-512

Taille	Registres							
128 bits	xmm16	xmm17	xmm18	xmm19	xmm20	xmm21	xmm22	xmm23
	xmm24	xmm25	xmm26	xmm27	xmm28	xmm29	xmm30	xmm31
256 bits	ymm16	ymm17	ymm18	ymm19	ymm20	ymm21	ymm22	ymm23
	ymm24	ymm25	ymm26	ymm27	ymm28	ymm29	ymm30	ymm31
512 bits	zmm16	zmm17	zmm18	zmm19	zmm20	zmm21	zmm22	zmm23
	zmm24	zmm25	zmm26	zmm27	zmm28	zmm29	zmm30	zmm31

En plus des nouvelles tailles de registres et d'opérandes, l'AVX-512 introduit un certain nombre de paramètres supplémentaires qui peuvent être inclus dans les opérandes des instructions AVX.

L'opérande destination de la plupart des instructions AVX peut être suivi du nom d'un registre *opmask* encadré d'accolades. Ce modificateur spécifie un masque qui décide quelles unités de données dans l'opérande destination vont être mises à jour. Le registre k0 ne peut pas être utilisé comme masque de destination. Ce paramètre peut être suivi d'un modificateur {z} pour choisir de remettre à zéro les unités de données non sélectionnées par masque au lieu de les laisser inchangées.

```
vaddpd zmm1 {k1}, zmm5, zword [rsi]   ; mise à jour des flottants sélectionnés
vaddps ymm6 {k1} {z}, ymm12, ymm24    ; mise à jour sélectionnée, mise à zéro des autres
```

Lorsqu'une instruction qui opère sur des données multiples a un opérande source chargé à partir d'une mémoire, l'emplacement mémoire peut être juste une seule unité de données et la source utilisée pour l'opération est créée en diffusant cette valeur dans toutes les unités dans la taille requise. Pour indiquer que cette méthode de diffusion est utilisée, la mémoire opérande doit être suivie par l'un des modificateurs {1to2}, {1to4}, {1to8}, {1to16}, {1to32}et {1to64}, la multiplication appropriée d'une unité.

```
vsubps zmm1, zmm2, dword [rsi] {1to16}   ; soustraire de tous les flottants
```

Lorsqu'une instruction n'utilise pas d'opérande mémoire, un opérande supplémentaire peut souvent suivre les opérandes source, contenant le spécificateur de mode d'arrondi. Lorsqu'une instruction a des variantes qui fonctionnent sur différentes tailles de données, le mode d'arrondi ne peut être spécifié que lorsque les opérandes de registre sont de 512 bits.

```
vdivps zmm2, zmm3, zmm5, {ru-sae}        ; arrondi des résultats
```

Tableau 2.7 – Modes d'arrondi de l'AVX-512

Opérande	Description
{rn-sae}	arrondi au plus proche et suppression de toutes les exceptions
{rd-sae}	Arrondi à la valeur inférieure et suppression de toutes les exceptions
{ru-sae}	arrondi à la valeur supérieure et suppression de toutes les exceptions
{rz-sae}	Arrondi vers zéro et suppression de toutes les exceptions

Certaines instructions n'utilisent pas de mode d'arrondi mais permettent tout de même de spécifier l'option de suppression d'exception avec un modificateur supplémentaire {sae} dans l'opérande.

```
vmaxpd zmm0, zmm1, zmm2, {sae}        ; suppression de toutes les exceptions
```

La famille d'instructions gather (rassemblement) dans leurs variantes AVX-512 utilise une nouvelle syntaxe avec seulement deux opérandes. Le registre *opmask* prend le rôle qui était joué par le troisième opérande dans la syntaxe AVX2 et il est obligatoire dans ce cas.

```
vgatherdps xmm0 {k1}, [eax + xmm1]    ; rassemblement de 4 flottants
vgatherdpd zmm0 {k3}, [ymm3 * 8]      ; rassemblement de 8 doubles
```

La nouvelle famille d'instructions scatter (dispersion) effectue une opération inverse de celle de gather. Elles prennent également deux opérandes. La destination est une mémoire avec indexation vectorielle et modificateur *opmask*, et la source est un registre vectoriel.

```
vscatterdps [eax + xmm1] {k1}, xmm0   ; dispersion de 4 flottants
vscatterdpd [ymm3 * 8] {k3}, zmm0     ; dispersion de 8 doubles
```

L'extension AVX512_4VNNI introduit des instructions avec une autre variante de syntaxe inhabituelle. Le premier opérande source des instructions **VP4DPWSSD** ou **VP4DPWSSDS** fait référence à un bloc aligné de quatre registres de 512 bits, contenant le registre de base spécifié par l'opérande. Cela peut être indiqué en ajoutant +3 au nom du registre, bien que ce soit facultatif.

```
vp4dpwssd zmm1 {k1} {z}, zmm2+3, xword [rbx]
```

2.1.25 Autres extensions du jeu d'instructions

Il existe un certain nombre d'extensions de jeu d'instructions supplémentaires reconnues par l'assembleur *flat*, et des exemples de syntaxe des instructions introduites par ces extensions sont fournis ici. Pour obtenir des informations détaillées sur les opérations effectuées par elles, consultez les manuels d'Intel ou d'AMD.

Les extensions de machine virtuelle (VMX) fournissent un ensemble d'instructions pour la gestion des machines virtuelles. L'instruction **VMXON**, qui entre dans l'opération VMX, nécessite un seul opérande de mémoire de 64 bits, qui doit être une adresse physique de la région de mémoire, que le processeur logique peut utiliser pour prendre en charge l'opération VMX. L'instruction **VMXOFF**, qui quitte l'opération VMX, n'a pas d'opérande. Les instructions **VMLAUNCH** et **VMRESUME**, dont le lancent ou reprennent les machines virtuelles, et l'instruction VMCALL qui permet au logiciel invité d'appeler le moniteur VM, n'utilisent pas d'opérande non plus.

L'instruction **VMPTRLD** charge l'adresse physique de la structure de contrôle de machine virtuelle (VMCS) actuelle à partir de son opérande de mémoire. L'instruction **VMPTRST** stocke le pointeur vers le VMCS actuel dans l'adresse spécifiée par son opérande de mémoire et **VMCLEAR** définit l'état de lancement du VMCS réfé-

rencé par son opérande de mémoire à effacer. Ces trois instructions nécessitent toutes un opérande de mémoire 64 bits unique.

L'instruction **VMREAD** lit dans VCMS un champ spécifié par l'opérande source et le stocke dans l'opérande destination. L'opérande source doit être un registre à usage général et l'opérande destination peut être un registre de mémoire. L'instruction **VMWRITE** écrit dans un champ VMCS spécifié par l'opérande destination la valeur fournie par l'opérande source. L'opérande source peut être un registre ou une mémoire à usage général, et l'opérande destination doit être un registre. La taille des opérandes pour ces instructions doit être de 64 bits en mode long et de 32 bits dans le cas contraire.

Es 'instructions **INVEPT** et **INVVPID** invalident les mémoires tampons de traduction (*lookaside* de TLB) et des caches de structure de pagination, soit dérivées de tables de pages étendues (EPT), ou sur la base de l'identificateur de processeur virtuel (VPID). Ces instructions nécessitent deux opérandes, le premier étant le registre à usage général spécifiant le type d'invalidation, le second étant un opérande mémoire de 128 bits fournissant le descripteur d'invalidation. Le premier opérande doit être un registre 64 bits en mode long et un registre 32 bits dans le cas contraire.

Les extensions de mode le plus sûr (SMX) fournissent les fonctionnalités disponibles via l'instruction **GETSEC**. Cette instruction ne prend pas d'opérandes et la fonction qui est exécutée est déterminée par le contenu du registre EAX lors de l'exécution de cette instruction.

La machine virtuelle sécurisée (SVM) est une variante de l'extension de machine virtuelle utilisée par AMD. L'instruction **SKINIT** réinitialise en toute sécurité le processeur permettant le démarrage d'un logiciel de confiance, tel que le moniteur de machine virtuelle (VMM). Cette instruction prend un seul opérande, qui doit être EAX, et fournit une adresse physique du bloc chargeur sécurisé (SLB).

L'instruction **VMRUN** permet de démarrer une machine virtuelle invitée. Son seul opérande doit être un registre accumulateur (AX, EAX ou RAX, le dernier disponible uniquement en mode long) fournissant l'adresse physique du bloc de contrôle de la machine virtuelle (VMCB). L'instruction **VMSAVE** stocke un sous-ensemble de l'état du processeur dans le VMCB spécifié par son opérande et VMLOAD charge le même sous-ensemble de l'état du processeur à partir d'un VMCB spécifié. Les mêmes règles d'opérande que pour l'instruction **VMRUN** s'appliquent à ces deux instructions.

L'instruction **VMMCALL** permet au logiciel invité d'appeler le VMM. Elle ne prend pas d'opérande.

L'instruction STGI définissez le flag d'interruption globale sur 1 et le CLGI le remet à zéro. Ces instructions ne prennent aucun opérande.

L'instruction **INVLPGA** invalide le mappage TLB pour une page virtuelle spécifiée par le premier opérande (qui doit être le registre accumulateur) et l'identificateur d'espace d'adressage spécifié par le deuxième opérande (qui doit être le registre ECX).

L'ensemble d'instructions XSAVE permet de sauvegarder et de restaurer les composants de l'état du processeur. Les instructions **XSAVE** et **XSAVEOPT** stockent les composants de l'état du processeur défini par le masque de bits dans les registres EDX et EAX dans la zone définie par l'opérande mémoire. L'instruction **XRSTOR** restaure à partir de la zone spécifiée par l'opérande mémoire les composants de l'état du processeur défini par masque dans EDX et EAX. Les instructions **XSAVE64**, **XSAVEOPT64** et **XRSTOR64** sont les versions de ces instructions 64 bits, autorisées uniquement en mode long.

L'instruction **XGETBV** lit le contenu du XCR 64 bits (registre de contrôle étendu) spécifié dans le registre ECX dans les registres EDX et EAX. L'instruction **XSETBV** écrit le contenu de EDX et EAX dans le XCR 64 bits spécifié par le registre ECX. Ces instructions n'ont pas d'opérande.

L'extension **RDRAND** introduit une nouvelle instruction, **RDRAND** qui charge la valeur aléatoire générée par le matériel dans le registre général. Elle prend un opérande, qui peut être un registre 16 bits, 32 bits ou 64 bits (ce dernier n’étant autorisé qu’uniquement en mode long).

L'extension FSGSBASE ajoute des instructions en mode long qui permettent de lire et d'écrire les registres de base de segment pour les segments FS et GS. Les instructions **RDFSBASE** et **RDGSBASE** lisent les registres de base de segment correspondants dans l'opérande, tandis que les instructions **WRFSBASE** et **WRGSBASE** écrivent la valeur de l'opérande dans ces registres. Toutes ces instructions prennent un opérande, qui peut être un registre général 32 bits ou 64 bits.

L'extension INVPCID ajoute une instruction **INVPCID**, qui invalide le mappage dans les TLB et les caches de pagination en fonction du type d'invalidation spécifié dans le premier opérande et du descripteur d'invalidation PCID spécifié dans le deuxième opérande. Les premiers opérandes doivent être un registre général de 32 bits lorsqu'il n'est pas en mode long, ou un registre général de 64 bits lorsqu'il est en mode long. Le deuxième opérande doit être un emplacement mémoire de 128 bits.

Les extensions HLE et RTM fournissent un ensemble d'instructions pour la gestion transactionnelle. Les instructions **XACQUIRE** et **XRELEASE** sont en fait de nouveaux préfixes qui peuvent être utilisés avec certaines des instructions pour démarrer ou terminer le verrouillage de l'élision sur l'adresse mémoire spécifiée par l'instruction préfixée. L'instruction **XBEGIN** démarre l'exécution transactionnelle, son opérande est l'adresse d'une rou-

tine de secours qui s'exécute en cas d'abandon de transaction, spécifiée comme l'opérande pour l'instruction de saut proche. L'instruction **XEND** marque la fin de la région d'exécution transactionnelle. Elle ne prend aucun opérande. L'instruction **XABORT** force l'abandon de la transaction. Elle prend une valeur immédiate de 8 bits comme seul **opérande**. Cette valeur est transmise dans les bits les plus élevés d'EAX à la routine de secours. L'instruction XTEST vérifie si une exécution transactionnelle est en cours, cette instruction ne prend aucun opérande.

L'extension MPX ajoute des instructions qui fonctionnent sur de nouveaux registres de limites et aident à vérifier les références mémoire. Pour certaines de ces instructions, les assembleurs *flat* autorisent une syntaxe spéciale qui permet un contrôle précis de leur fonctionnement, où une adresse d'un opérande de mémoire est séparée en deux parties par une virgule. Avec l'instruction **BNDMK**, la première partie de cette adresse spécifie la borne inférieure et la seconde la borne supérieure. La limite inférieure peut être zéro ou un registre, la limite supérieure peut être toute adresse qui n'utilise pas plus d'un registre (multipliée par 1, 2, 4 ou 8). Les registres d'adressage doivent être 64 bits en mode long et 32 bits dans le cas contraire.

```
bndmk bnd0, [rbx, 100 000h]    ; borne inférieure du registre, supérieure directement
bndmk bnd1, [0, rbx]           ; borne inférieure zéro, supérieure du registre
```

Dans le cas des instructions **BNDLDX** et **BNDSTX**, la première partie de l'opérande mémoire spécifie une adresse utilisée pour accéder à une entrée de table liée, tandis que la seconde partie est soit zéro, soit un registre qui joue le rôle d'un opérande supplémentaire pour une telle instruction. L'adresse de la première partie ne peut utiliser plus d'un registre et le registre ne peut pas être multiplié par un nombre autre que 1.

```
bndstx [rcx, rsi], bnd3        ; stockage de bnd3 et rsi en rcx dans la table de limites
bndldx bnd2, [rcx, rsi]        ; chargement à partir de la table liée si l'entrée correspond à rsi
```

2.2 Directives de contrôle

Cette section décrit les directives qui contrôlent le processus d'assemblage. Elles sont traitées pendant l'assemblage et peuvent entraîner l'assemblage de certains blocs d'instructions différemment ou ne pas être assemblés du tout.

2.2.1 Constantes numériques

La directive = permet de définir la constante numérique. Elle doit être précédée du nom de la constante et suivie de l'expression numérique fournissant la valeur. La valeur de ces constantes peut être un nombre ou une adresse, mais – contrairement aux étiquettes – les constantes numériques ne sont pas autorisées à contenir les adresses basées sur les registres. Outre cette différence, dans leur variante de base, les constantes numériques se comportent presque comme des étiquettes et vous pouvez même les référencer en aval (accéder à leurs valeurs avant qu'elles ne soient réellement définies).

Il existe cependant une deuxième variante des constantes numériques, qui est reconnue par l'assembleur lorsque vous essayez de définir le nom de la constante, sous laquelle il y avait déjà une constante numérique définie. Dans ce cas, l'assembleur traite cette constante comme une variable au moment de l'assemblage et lui permet de lui être assignée avec une nouvelle valeur, mais interdit de la référencer avant (pour des raisons évidentes). Voyons à la fois la variante des constantes numériques dans un exemple :

```
dd sum
x = 1
x = x+2
sum = x
```

Ici, `x` est une variable au moment de l'assemblage, et à chaque fois qu'on y accède, la valeur qui lui a été assignée le plus récemment est utilisée. Ainsi, si nous essayions d'accéder au `x` avant qu'il ne soit défini la première fois, comme si nous écrivions `dd x` à la place de l'instruction `dd sum`, cela provoquerait une erreur. Et quand elle est redéfinie avec la directive `x = x+2`, la valeur précédente de `x` est utilisée pour calculer la nouvelle. Ainsi, lorsque la constante `sum` est définie, la valeur `x` a la valeur 3, et cette valeur est affectée au `sum`. Comme celle-ci n'est définie qu'une seule fois dans la source, il s'agit de la constante numérique standard et peut être référencée en aval. Donc, le `dd sum` est assemblé comme `dd 3`. Pour en savoir plus sur la façon dont l'assembleur est capable de résoudre ce problème, consultez la section 2.2.6.

La valeur de la constante numérique peut être précédée de l'opérateur de taille, qui peut garantir que la valeur correspondra à la plage de la taille spécifiée et peut également affecter la manière dont certains des calculs à l'intérieur de l'expression numérique sont effectués. Cet exemple :

```
c8 = octet -1
c32 = dword -1
```

définit deux constantes différentes, la première tient sur 8 bits, la seconde sur 32 bits.

Lorsque vous devez définir une constante avec la valeur de l'adresse, qui peut être basée sur un registre (et que vous ne pouvez donc pas utiliser de constante numérique à cette fin), vous pouvez utiliser la syntaxe étendue de la directive `label` (déjà décrite dans la section 1.2.3), comme :

```
label myaddr à ebp + 4
```

qui déclare l'étiquette placée à l'adresse `ebp+4`. Cependant, rappelez-vous que les étiquettes, contrairement aux constantes numériques, ne peuvent pas devenir des variables de montage.

2.2.2 Assemblage conditionnel

La directive **IF** permet de faire en sorte qu'un bloc d'instructions puisse être assemblé uniquement sous certaines conditions. Cette directive doit être suivie d'une expression logique spécifiant la condition. Les instructions des lignes suivantes ne seront assemblées que lorsque cette condition est remplie, sinon elles seront ignorées. La directive facultative **ELSE IF** suivie d'une expression logique spécifiant une condition supplémentaire commence le bloc d'instructions suivant qui sera assemblé si les conditions précédentes n'étaient pas remplies et que la condition supplémentaire est remplie. La directive **ELSE** optionnelle commence le bloc d'instructions qui sera assemblé si toutes les conditions n'étaient pas remplies. La directive **END IF** termine le dernier bloc d'instructions.

Vous devez noter que la directive **IF** est traitée au stade de l'assemblage et qu'elle n'affecte donc pas les directives de préprocesseur, comme les définitions des constantes symboliques et des macro-instructions - lorsque l'assembleur reconnaît la directive **IF**, tout le prétraitement est déjà terminé.

L'expression logique se compose de valeurs logiques et d'opérateurs logiques. Les opérateurs logiques sont ~ pour la négation logique, & pour un AND logique et | pour un OU logique. La négation a la priorité la plus élevée. La valeur logique peut être une expression numérique. Elle sera fausse si elle est égale à zéro, sinon elle sera vraie. Deux expressions numériques peuvent être comparées à l'aide de l'un des opérateurs suivants pour créer la valeur logique : =(égal), <(inférieur), >(supérieur), <=(inférieur ou égal), >=(supérieur ou égal), <>(non égal).

L'opérateur **USED** suivi d'un nom de symbole, est la valeur logique qui vérifie si le symbole donné est utilisé quelque part (il retourne un résultat correct même si le symbole n'est utilisé qu'après cette vérification). L'opérateur **DEFINED** peut être suivi par n'importe quelle expression, généralement juste par un seul nom de symbole ; il vérifie si l'expression donnée contient uniquement des symboles définis dans la source et accessibles à partir de la position actuelle. L'opérateur **DEFINITE** effectue une vérification similaire avec restriction aux symboles définis avant la position actuelle dans la source.

Avec l'opérateur **RELATIVETO**, il est possible de vérifier si les valeurs de deux expressions ne diffèrent que par une quantité constante. La syntaxe valide est une expression numérique suivie d'un **RELATIVETO** une autre expression (éventuellement basée sur un registre). Les étiquettes qui n'ont pas de valeur numérique simple peuvent être testées de cette façon pour déterminer le type d'opérations possibles avec elles.

L'exemple simple suivant utilise la constante `count` qui doit être définie quelque part dans la source :

```
if count>0
    mov cx, count
    rep movsb
end if
```

Ces deux instructions d'assemblage ne seront assemblées que si la constante `count` est supérieure à 0. L'exemple suivant montre une structure conditionnelle plus complexe :

```
if count & ~ count mod 4
    mov cx, count/4
    rep movsd
else if count>4
    mov cx, count/4
    rep movsd
    mov cx, count mod 4
    rep movsb
else
    mov cx, count
    rep movsb
end if
```

Le premier bloc d'instructions est assemblé lorsque `count` est différent de zéro et divisible par quatre. Si cette condition n'est pas remplie, la deuxième expression logique, qui suit `else if`, est évaluée et si c'est vrai, le deuxième bloc d'instructions est assemblé, sinon le dernier bloc d'instructions, qui suit la ligne contenant seulement `else`, est assemblé.

Il existe également des opérateurs qui permettent de comparer les valeurs de n'importe quelle chaîne de symboles. L'opérateur **EQ** compare ces deux valeurs sur un critère d'égalité. L'opérateur IN vérifie si la valeur donnée est membre de la liste de valeurs suivant cet opérateur, la liste doit être entourée de caractères < et >, ses

membres doivent être séparés par des virgules. Les symboles sont considérés comme identiques lorsqu'ils ont la même signification pour l'assembleur – par exemple `pword` et `fword` pour l'assembleur sont les mêmes et ne sont donc pas distingués par les opérateurs ci-dessus. De la même manière `16 eq 10h` est la vraie condition, mais `16 eq 10+4` ne l'est pas.

L'opérateur **EQTYPE** vérifie si les deux valeurs comparées ont la même structure et si les éléments structurels sont du même type. Les types distinctifs incluent les expressions numériques, les chaînes individuelles entre guillemets, les nombres à virgule flottante, les expressions d'adresse (les expressions entre crochets ou précédées de l'opérateur PTR), mnémoniques d'instructions, registres, opérateurs de taille, opérateurs de type de saut et de type de code. Et chacun des caractères spéciaux qui agissent comme séparateurs, comme une virgule ou deux points, est le type distinct lui-même. Par exemple, deux valeurs, chacune consistant en un nom de registre suivi d'une virgule et d'une expression numérique, seront considérées comme du même type, quel que soit le type de registre et la complexité de l'expression numérique utilisée; à l'exception des chaînes entre guillemets et des valeurs à virgule flottante, qui sont les types spéciaux d'expressions numériques et sont traités comme des types différents. Ainsi, la condition `eax,16 eqtype fs,3+7` est-elle vraie, alors que `eax,16 eqtype eax,1.6` est fausse.

2.2.3 Répétition de blocs d'instructions

La directive **TIMES** répète une instruction d'un nombre de fois spécifié. Elle doit être suivie d'une expression numérique spécifiant le nombre de répétitions et l'instruction à répéter (facultativement deux-points peuvent être utilisés pour séparer le nombre et l'instruction). Lorsqu'un symbole spécial % est utilisé dans l'instruction, il est égal au nombre de répétition en cours. Par exemple `times 5 db %`, définira cinq octets avec les valeurs 1, 2, 3, 4, 5. L'utilisation récursive de la directive TIMES est également autorisée, donc `times 3 times % db %` définira six octets avec les valeurs 1, 1, 2, 1, 2, 3.

La directive **REPEAT** répète tout le bloc d'instructions. Il doit être suivi d'une expression numérique spécifiant le nombre de répétitions. Les instructions à répéter sont attendues dans les lignes suivantes, terminées par la directive END REPEAT, par exemple :

```
repeat 8
    mov byte [bx],%
    inc bx
end repeat
```

Le code généré stockera des valeurs d'octets de un à huit dans la mémoire adressée par le registre BX.

Le nombre de répétitions peut être nul, dans ce cas les instructions ne sont pas du tout assemblées.

La directive **BREAK** permet d'arrêter de répéter plus tôt et de continuer l'assemblage à partir de la première ligne après le END REPEAT. Combiné avec la directive **IF**, il permet d'arrêter de se répéter sous certaines conditions spéciales, comme :

```
s = x/2
repeat 100
    if x/s = s
        break
    end if
    s = (s+x/s)/2
end repeat
```

La directive **WHILE** répète le bloc d'instructions tant que la condition spécifiée par l'expression logique qui la suit est vraie. Le bloc d'instructions à répéter doit se terminer par la directive **END WHILE**. Avant chaque répétition, l'expression logique est évaluée et lorsque sa valeur est false, l'assemblage se poursuit à partir de la première ligne après le END WHILE. Dans ce cas également, le symbole % contient le nombre de répétitions en cours. La directive BREAK peut être utilisée pour arrêter ce type de boucle de la même manière qu'avec la directive REPEAT. L'exemple précédent peut être réécrit pour utiliser WHILE au lieu de REPEAT cette manière :

```
s = x/2
while x/s <> s
    s = (s+x/s)/2
    if % = 100
        break
    end if
end while
```

Les blocs définis avec **IF**, **REPEAT** et **WHILE** peuvent être imbriqués dans l'ordre, mais ils doivent être fermés dans le même ordre dans lequel ils ont commencé. La directive **BREAK** arrête toujours le traitement du bloc qui a été démarré en dernier avec la directive REPEAT ou WHILE.

2.2.4 Espaces d'adressage

La directive **ORG** définit l'adresse à laquelle le code suivant doit apparaître en mémoire. Il doit être suivi d'une expression numérique spécifiant l'adresse. Cette directive commence le nouvel espace d'adressage, le code suivant lui-même n'est déplacé d'aucune façon, mais toutes les étiquettes définies à l'intérieur et la valeur de $symbole sont affectées comme s'il était placé à l'adresse donnée. Cependant, il est de la responsabilité du programmeur de mettre le code à la bonne adresse au moment de l'exécution.

La directive **LOAD** permet de définir une constante avec une valeur binaire chargée à partir du code déjà assemblé. Cette directive doit être suivie du nom de la constante, puis éventuellement de l'opérateur **SIZE**, puis de l'opérateur FROM et d'une expression numérique spécifiant une adresse valide dans l'espace d'adressage courant. L'opérateur de taille a une signification inhabituelle dans ce cas – il indique combien d'octets (jusqu'à 8) doivent être chargés pour former la valeur binaire de constante. Si aucun opérateur de taille n'est spécifié, un octet est chargé (la valeur est donc comprise entre 0 et 255). Les données chargées ne peuvent pas dépasser le décalage actuel.

La directive **STORE** peut modifier le code déjà généré en remplaçant certaines des données précédemment générées par la valeur définie par l'expression numérique donnée, qui suit. L'expression peut être précédée de l'opérateur de taille facultatif pour spécifier la taille de la valeur définie par l'expression, et par conséquent la quantité d'octets stockée, s'il n'y a pas d'opérateur de taille, la taille d'un octet est supposée. Ensuite, l'opérateur **AT** et l'expression numérique définissant l'adresse valide dans l'espace de code d'adressage actuel, à laquelle la valeur donnée doit être stockée doivent suivre. Il s'agit d'une directive pour les appareils avancés et doit être utilisée avec précaution.

Les deux directives **LOAD** et **STORE** dans leur variante de base (définie ci-dessus) sont limitées pour fonctionner sur des emplacements dans l'espace d'adressage actuel. Le symbole **$$** est toujours égal à l'adresse de base de l'espace d'adressage actuel, et le symbole **$** correspond à l'adresse de la position actuelle dans cet espace d'adressage. Par conséquent, ces deux valeurs définissent les limites de la zone, où LOAD et STORE peuvent fonctionner.

La combinaison des directives LOAD et STORE permet de faire des choses comme encoder une partie du code déjà généré. Par exemple, pour encoder tout le code généré dans l'espace d'adressage actuel, vous pouvez utiliser un tel bloc de directives :

```
repeat $-$$
    load a byte from $$+%-1
    store byte a xor c at $$+%-1
end repeat
```

et chaque octet de code sera “Xoré” avec la valeur définie par la constante `c`.

La directive **VIRTUAL** définit les données virtuelles à l'adresse spécifiée. Ces données ne seront pas incluses dans le fichier de sortie, mais les étiquettes qui y sont définies peuvent être utilisées dans d'autres parties de la source. Cette directive peut être suivie de l'opérateur **AT** et de l'expression numérique spécifiant l'adresse des données virtuelles, sinon elle utilise l'adresse actuelle, identique à `virtual at $`. Les instructions définissant les données sont attendues dans les lignes suivantes, se terminant par une directive **END VIRTUAL**. Le bloc d'instructions virtuelles lui-même est un espace d'adressage indépendant, une fois terminé. Le contexte de l'espace d'adressage précédent est restauré.

La directive **VIRTUAL** peut être utilisée pour créer l'union de certaines variables, par exemple :

```
GDTR dp ?
virtual at GDTR
    GDT_limit dw ?
    GDT_address dd ?
end virtual
```

Elle définit deux étiquettes pour les parties de la variable 48 bits à l'adresse **GDTR**.

Elle peut également être utilisée pour définir des étiquettes pour certaines structures adressées par un registre, par exemple :

```
virtual at bx
    LDT_limit dw ?
    LDT_address dd ?
end virtual
```

Avec une telle définition, l'instruction `mov ax,[LDT_limit]` sera assemblée selon la même instruction que `mov ax,[bx]`.

Déclarer des valeurs de données définies ou des instructions à l'intérieur du bloc virtuel peut également être utile, car la directive **LOAD** peut être utilisée pour charger les valeurs du code généré virtuellement dans des cons-

tantes. Cette directive dans sa version de base doit être utilisée après le chargement du code mais avant la fin du bloc virtuel, car elle ne peut charger les valeurs qu'à partir du même espace d'adressage. Par exemple :

```
virtual at 0
    xor eax,eax
    and edx,eax
    load zeroq dword from 0
end virtual
```

Le morceau de code ci-dessus définira la constante `zeroq` contenant quatre octets du code machine des instructions définies à l'intérieur du bloc virtuel. Cette méthode peut également être utilisée pour charger une valeur binaire à partir d'un fichier externe. Par exemple ce code :

```
virtual at 0
    file 'a.txt':10h,1
    load char from 0
end virtual
```

charge l'octet unique à partir du décalage 10h du fichier `a.txt` dans la constante `char`.

Au lieu ou en plus d'un argument **AT**, **VIRTUAL** peut également être suivi d'un mot-clé **AS** et d'une chaîne définissant une extension de fichier supplémentaire où le contenu initialisé de l'espace d'adressage commencé par VIRTUAL va être stocké à la fin d'un assemblage réussi.

```
virtual at 0 as 'asc'
    times 256 db %-1
end virtual
```

Toutes les directives `section` décrites en 2.4 commencent également un nouvel espace d'adressage.

Il est possible de déclarer un type spécial d'étiquette qui marque l'espace d'adressage courant, en ajoutant un double deux-points au lieu d'un seul après un nom d'étiquette. Ce symbole ne peut alors pas être utilisé dans les expressions numériques. Le seul endroit où il est autorisé à l'utiliser est la syntaxe étendue des directives LOAD et STORE. Il est possible de faire fonctionner ces directives sur un espace d'adressage différent du courant, en spécifiant l'adresse avec les deux composants : d'abord le nom d'une étiquette spéciale qui marque l'espace d'adressage, suivi du caractère deux-points et d'une expression numérique définissant une adresse valide à l'intérieur de cet espace d'adressage. Dans l'exemple suivant, cette syntaxe étendue est utilisée pour charger la valeur d'un bloc après sa fermeture :

```
virtual at 0
    hex_digits::
    db '0123456789ABCDEF'
end virtual
load a byte from hex_digits:10
```

De cette façon, il est possible d'opérer sur des valeurs à l'intérieur de n'importe quel bloc de code, y compris toutes celles définies avec **VIRTUAL**. Cependant, il n'est pas autorisé de spécifier un espace d'adressage qui n'a pas encore été assemblé, tout comme il n'est pas autorisé de spécifier une adresse dans l'espace d'adressage actuel qui dépasse le décalage actuel. Les adresses dans tout autre espace d'adressage sont également limitées par les limites du bloc.

La directive `"virtual"` peut avoir une étiquette d'espace d'adressage préalablement définie comme seul argument. Cette variante permet d'étendre un bloc préalablement défini et fermé avec des données supplémentaires. Toute définition de données dans un bloc d'extension aura le même effet que si cette définition était présente dans le bloc `"virtual"` d'origine.

```
virtual at 0 as 'log'
    Log::
end virtual

virtual Log
    db 'Hello!',13,10
end virtual
```

2.2.5 Autres directives

La directive **ALIGN** aligne le code ou les données sur la limite spécifiée. Il doit être suivi d'une expression numérique spécifiant le nombre d'octets, au multiplicateur duquel l'adresse courante doit être alignée. La valeur limite doit être la puissance de deux.

La directive ALIGN remplit les octets qui ont dû être ignorés pour effectuer l'alignement avec les instructions NOP et marque en même temps cette zone comme des données non initialisées, donc si elle est placée parmi d'autres données non initialisées qui ne prendraient pas de place dans le fichier de sortie, les octets d'alignement

agiront de la même manière. Si vous avez besoin de remplir la zone d'alignement avec d'autres valeurs, vous pouvez combiner ALIGN avec VIRTUAL pour obtenir la taille d'alignement nécessaire, puis créer l'alignement vous-même, comme :

```
virtual
    align 16
    a = $ - $$
end virtual
db a dup 0
```

La constante a est définie comme étant la différence entre l'adresse après alignement et l'adresse du bloc `virtual` (voir la section précédente). Elle est donc égale à la taille de l'espace d'alignement nécessaire.

La directive **DISPLAY** affiche le message au moment de l'assemblage. Elle doit être suivie des chaînes ou des valeurs d'octets entre guillemets, séparés par des virgules. Elle peut être utilisée pour afficher les valeurs de certaines constantes, par exemple :

```
bits = 16
display 'Current offset is 0x'
repeat bits/4
    d = '0' + $ shr (bits-%*4) and 0Fh
    if d > '9'
        d = d + 'A'-'9'-1
    end if
    display d
end repeat
display 13,10
```

Ce bloc de directives calcule les quatre chiffres hexadécimaux de la valeur 16 bits et les convertit en caractères à afficher. Notez que cela ne fonctionnera pas si les adresses dans l'espace d'adressage actuel sont déplaçables (comme cela peut arriver avec les formats de sortie PE ou objet), car seules des valeurs absolues peuvent être utilisées de cette façon. La valeur absolue peut être obtenue en calculant l'adresse relative, comme `$-$$`, ou `rva $` en cas de format PE.

La directive **ERR** met immédiatement fin au processus d'assemblage lorsqu'il est rencontré par l'assembleur.

La directive **ASSERT** teste si l'expression logique qui la suit est vraie, et sinon, elle signale l'erreur.

2.2.6 Passes multiples

Puisque l'assembleur permet de référencer certaines des étiquettes ou des constantes avant qu'elles ne soient réellement définies, il lui incombe de prédire les valeurs de ces étiquettes et s'il y a même un soupçon que la prédiction a échoué dans au moins un cas, il effectue une passe supplémentaire, procédant cette fois à l'assemblage de toute la source en faisant une meilleure prédiction basée sur les valeurs obtenues par les étiquettes lors de la passe précédente.

Les valeurs changeantes des étiquettes peuvent amener certaines instructions à avoir des codages de longueur différente, ce qui peut provoquer à nouveau la modification des valeurs des étiquettes. Et comme les étiquettes et les constantes peuvent également être utilisées à l'intérieur des expressions qui affectent le comportement des directives de contrôle, l'ensemble du bloc source peut être traité de manière complètement différente lors de la nouvelle passe. Ainsi l'assembleur fait-il de plus en plus de passes, essayant à chaque fois de faire de meilleures prédictions pour approcher la solution finale, lorsque toutes les valeurs sont correctement prédites. Il utilise diverses méthodes de prédiction des valeurs, qui ont été choisies pour permettre de trouver en quelques passes la solution de longueur éventuellement la plus petite pour la plupart des programmes.

Certaines des erreurs, comme les valeurs qui ne correspondent pas aux limites requises, ne sont pas signalées pendant ces passes intermédiaires, car il peut arriver que lorsque certaines des valeurs sont mieux prédites, ces erreurs disparaissent. Cependant, si l'assembleur rencontre une construction de syntaxe illégale ou une instruction inconnue, il s'arrête toujours immédiatement. Définir également une étiquette plus d'une fois provoque une telle erreur, car cela rend les prédictions sans fondement.

Seuls les messages créés avec la directive **DISPLAY** lors du dernier passage effectué sont réellement affichés. Dans le cas où l'assemblage a été arrêté en raison d'une erreur, ces messages peuvent refléter les valeurs prédites qui ne sont pas encore résolues correctement.

La solution peut parfois ne pas exister et dans de tels cas, l'assembleur ne parviendra jamais à faire des prédictions correctes. Pour cette raison, il y a une limite du nombre de passes, et lorsque l'assembleur atteint cette limite, il s'arrête et affiche le message indiquant qu'il n'est pas capable de générer la sortie correcte. Prenons l'exemple suivant :

```
if ~ defined alpha
    alpha:
```

```
end if
```

L'opérateur **DEFINED** donne la valeur vraie lorsque l'expression qui la suit pourrait être calculée à cet endroit, ce qui dans ce cas signifie que l'étiquette `alpha` est définie quelque part. Mais le bloc ci-dessus fait que cette étiquette n'est définie que lorsque la valeur donnée par l'opérateur DEFINED est fausse, ce qui conduit à une antynomie et rend impossible la résolution d'un tel code. Lors du traitement de la directive `if`, l'assembleur doit prédire si l'étiquette `alpha` sera définie quelque part (il n'aurait pas à prédire uniquement si l'étiquette a déjà été définie plus tôt dans cette passe), et quelle que soit la prédiction, le contraire se produit toujours. Ainsi l'assemblage échouera, à moins que le label `alpha` ne soit défini quelque part dans la source précédant le bloc d'instructions ci-dessus. Dans ce cas, comme cela a déjà été noté, la prédiction n'est pas nécessaire et le bloc sera simplement ignoré.

L'exemple ci-dessus a peut-être été écrit pour tenter de définir l'étiquette uniquement lorsqu'elle n'était pas encore définie. Il échoue, car l'opérateur DEFINED vérifie si l'étiquette est définie n'importe où, et cela inclut la définition à l'intérieur de ce bloc traité conditionnellement. Il pourrait être facilement corrigé en utilisant l'opérateur **DEFINITE** au lieu de DEFINED. Mais il y a aussi une autre modification qui pourrait le résoudre:

```
if ~ defined alpha | defined @f
    alpha:
    @@:
end if
```

Le `@f` est toujours le même libellé que le symbole `@@` le plus proche dans la source qui le suit. Donc l'exemple ci-dessus aurait la même signification si un nom unique était utilisé à la place de l'étiquette anonyme. Lorsque `alpha` n'est défini à aucun autre endroit dans la source, la seule solution possible est lorsque ce bloc est défini, et cette fois cela ne conduit pas à l'antinomie, à cause de l'étiquette anonyme qui rend ce bloc auto-établi. Pour mieux comprendre cela, regardez les blocs qui n'ont rien de plus que cet auto-établissement :

```
if defined @f
    @@:
end if
```

Ceci est un exemple de source qui peut avoir plus d'une solution, car les deux cas où ce bloc est traité ou non sont également corrects. Laquelle de ces deux solutions que nous obtenons dépend de l'algorithme de l'assembleur, dans le cas de l'assembleur *flat* - de l'algorithme de prédictions. De retour à l'exemple précédent, lorsque `alpha` n'est défini nulle part ailleurs, la condition du bloc IF ne peut pas être fausse. Il ne nous reste donc qu'une seule solution possible, et nous pouvons espérer que l'assembleur y arrivera. D'un autre côté, quand `alpha` est défini à un autre endroit, nous avons à nouveau deux solutions possibles, mais l'une d'elles provoque la double définition de `alpha`, et une telle erreur amène l'assembleur à abandonner immédiatement l'assemblage, car c'est le genre d'erreur qui perturbe profondément le processus de résolution. Nous pouvons donc obtenir une telle source soit correctement résolue ou provoque une erreur, et ce que nous obtenons peut dépendre des choix internes effectués par l'assembleur.

Cependant, certains faits sur ces choix sont certains. Lorsque l'assembleur doit vérifier si le symbole donné est défini et s'il a déjà été défini dans la passe en cours, aucune prédiction n'est nécessaire. Cela a déjà été noté ci-dessus. Et lorsque le symbole donné n'a jamais été défini auparavant, y compris toutes les passes déjà terminées, l'assembleur prédit qu'il ne sera pas défini. Sachant cela, nous pouvons nous attendre à ce que le simple bloc auto-établissant montré ci-dessus ne soit pas assemblé du tout et que l'exemple précédent se résolve correctement lorsque `alpha` est défini quelque part avant notre bloc conditionnel, alors qu'il définira lui-même `alpha` quand il n'est pas déjà défini précédemment, causant ainsi potentiellement l'erreur en raison de la double définition si le `alpha` est également définie quelque part plus tard.

L'opérateur **USED** peut s'attendre à ce que l'opérateur se comporte de la même manière dans des cas analogues, mais tout autre type de prédiction peut ne pas être aussi simple et vous ne devriez jamais vous y fier de cette façon.

La directive **ERR**, généralement utilisée pour arrêter l'assemblage lorsqu'une condition est remplie, arrête l'assemblage immédiatement, que la passe en cours soit la dernière ou intermédiaire. Ainsi, même lorsque la condition qui a entraîné l'interprétation de cette directive est temporaire et finirait par disparaître dans les passes ultérieures, l'assemblage est arrêté de toute façon.

La directive **ASSERT** signale l'erreur uniquement si son expression est fausse après que tous les symboles ont été résolus. Vous pouvez utiliser `assert 0` à la place de `err` lorsque vous ne souhaitez pas que l'assemblage soit arrêté pendant les passes intermédiaires.

2.3 Directives du préprocesseur

Toutes les directives de préprocesseur sont traitées avant le processus d'assemblage principal et ne sont donc pas affectées par les directives de contrôle. À ce moment également, tous les commentaires sont supprimés.

2.3.1 Inclusion des fichiers source

La directive **INCLUDE** inclut le fichier source spécifié à la position où il est utilisé. Il doit être suivi du nom entre guillemets du fichier à inclure, par exemple :

```
include 'macros.inc'
```

L'ensemble du fichier inclus est pré-traité avant le pré-traitement des lignes suivant celle contenant la directive INCLUDE. Il n'y a pas de limite au nombre de fichiers inclus tant qu'ils tiennent en mémoire.

Le chemin d'accès entre guillemets peut contenir des variables d'environnement entourées de caractères %. Elles seront remplacées par leurs valeurs à l'intérieur du chemin. Les caractères \ et / sont autorisés comme séparateurs de chemin d'accès. Le fichier est d'abord recherché dans le répertoire contenant le fichier qui l'a inclus et lorsqu'il n'y est pas trouvé, la recherche se poursuit dans les répertoires spécifiés dans la variable d'environnement appelée INCLUDE (des chemins multiples séparés par des points-virgules peuvent y être définis ; ils seront recherchés dans le même ordre que celui spécifié). Si le fichier n'a été trouvé à aucun de ces endroits, le préprocesseur le recherche dans le répertoire contenant le fichier source principal (celui spécifié dans la ligne de commande). Ces règles concernent également les chemins donnés avec la directive **FILE**.

2.3.2 Constantes symboliques

Les constantes symboliques diffèrent des constantes numériques. Avant le processus d'assemblage, elles sont remplacées par leurs valeurs partout dans les lignes source après leurs définitions, et tout peut devenir leurs valeurs.

La définition d'une constante symbolique se compose du nom de ladite constante suivi de la directive **EQU**. Tout ce qui suit cette directive prendra la valeur de la constante. Si la valeur de la constante symbolique contient d'autres constantes symboliques, elles sont remplacées par leurs valeurs avant d'affecter cette valeur à la nouvelle constante. Par exemple :

```
d equ dword
NULL equ d 0
d equ edx
```

Après ces trois définitions, la valeur de la constante NULL est `dword 0` et la valeur de `d` est `edx`. Ainsi, par exemple, `push NULL` sera assemblé en tant que `push dword 0` et `push d` sera assemblé en tant que `push edx`. Et si alors la ligne suivante était mise :

```
d equ d,eax
```

la constante `d` obtiendrait la nouvelle valeur de `edx,eax`. De cette façon, les listes croissantes de symboles peuvent être définies.

La directive **RESTORE** permet de récupérer la valeur précédente de la constante symbolique redéfinie. Elle doit être suivie d'un ou plusieurs autres noms de constantes symboliques, séparés par des virgules. Ainsi, `restore d` après les définitions ci-dessus rendra-t-elle à `d` la valeur constante `edx`, la seconde la restaurera à sa valeur `dword`, et une autre rendra à `d` sa signification d'origine comme si aucune constante de ce type n'avait été définie. S'il n'y avait pas de constante définie d'un nom donné, `restore` ne causerait pas d'erreur, elle serait simplement ignorée.

La constante symbolique peut être utilisée pour adapter la syntaxe de l'assembleur aux préférences personnelles. Par exemple, l'ensemble de définitions suivant fournit les raccourcis pratiques pour tous les opérateurs de taille :

```
b equ byte
w equ word
d equ dword
p equ pword
f equ fword
q equ qword
t equ tword
x equ dqword
y equ qqword
```

Étant donné que la constante symbolique peut également avoir une valeur vide, elle peut être utilisée pour autoriser la syntaxe avec le mot OFFSET avant toute valeur d'adresse :

```
offset equ
```

Après cette définition `mov ax,offset char` sera une construction valide pour copier le décalage de la variable `char` dans le registre `ax`, car `offset` est remplacé par une valeur vide, et donc ignoré.

La directive **DEFINE** suivie du nom de la constante, puis de la valeur, est la manière alternative de définir une constante symbolique. La seule différence entre DEFINE et EQU réside dans le fait que DEFINE affecte la valeur telle quelle. Elle ne remplace pas les constantes symboliques par leurs valeurs à l'intérieur.

Les constantes symboliques peuvent également être définies avec la directive **FIX**, qui a la même syntaxe que EQU, mais définit des constantes de haute priorité, c'est-à-dire qu'elles sont remplacées par leurs valeurs symbo-

liques avant même de traiter les directives et les macro-instructions du préprocesseur. La seule exception est la directive FIX elle-même, qui a la priorité la plus élevée possible, ce qui permet de redéfinir les constantes définies de cette manière.

La directive FIX peut être utilisée pour les ajustements de syntaxe liés aux directives du préprocesseur, ce qui ne peut pas être fait avec la directive EQU. Par exemple :

```
incl fix include
```

définit un nom court pour la directive INCLUDE, alors que la définition similaire faite avec la directive EQU ne donnerait pas un tel résultat, car les constantes symboliques standard sont remplacées par leurs valeurs après avoir recherché la ligne des directives du préprocesseur.

2.3.3 Macro-instructions

La directive **MACRO** vous permet de définir vos propres instructions complexes, appelées macro-instructions, qui peuvent grandement simplifier le processus de programmation. Dans sa forme la plus simple, il est similaire à la définition de constante symbolique. Par exemple, la définition suivante définit un raccourci pour l'instruction `test al,0xFF` :

```
macro tst {test al,0xFF}
```

Après la directive MACRO, il y a un nom de macro-instruction, puis son contenu entre les caractères { et }. Vous pouvez utiliser l'instruction TST n'importe où après cette définition et elle sera assemblée comme `test al,0xFF`. La définition de la constante symbolique `tst` de cette valeur donnerait le même résultat, mais la différence est que le nom de la macro-instruction n'est reconnu que comme un mnémonique d'instruction. De plus, les macro-instructions sont remplacées par le code correspondant avant même que les constantes symboliques ne soient remplacées par leurs valeurs. Donc, si vous définissez une macro-instruction et une constante symbolique du même nom, et que vous utilisez ce nom comme une instruction mnémonique, il sera remplacé par le contenu de la macro-instruction, mais il sera remplacé par value si une constante symbolique si elle est utilisée quelque part à l'intérieur des opérandes.

La définition de macro-instruction peut se composer de plusieurs lignes, car les caractères { et } ne doivent pas nécessairement être dans la même ligne que la directive MACRO. Par exemple :

```
macro stos0
 {
    xor al,al
    stosb
 }
```

La macro-instruction `stos0` sera remplacée par ces deux instructions d'assemblage partout où elle est utilisée.

Comme les instructions qui nécessitent un certain nombre d'opérandes, la macro-instruction peut être définie comme nécessitant un certain nombre d'arguments séparés par des virgules. Les noms des arguments nécessaires doivent suivre le nom de la macro-instruction dans la ligne de la directive **MACRO** et doivent être séparés par des virgules s'il y en a plusieurs. Partout où l'un de ces noms apparaît dans le contenu de la macro-instruction, il sera remplacé par la valeur correspondante, fournie lorsque la macro-instruction est utilisée. Voici un exemple de macro-instruction qui effectuera l'alignement des données pour le format de sortie binaire :

```
macro align value { rb (value-1)-($+value-1) mod value }
```

Lorsque l'instruction `align 4` est trouvée après la définition de cette macro-instruction, elle est remplacée par le contenu de cette macro-instruction, et `value` deviendra alors 4. Donc, le résultat sera `rb (4-1)-($+4-1) mod 4`.

Si une macro-instruction est définie qui utilise une instruction avec le même nom dans sa définition, la signification précédente de ce nom est utilisée. Une redéfinition utile des macro-instructions peut être effectuée de cette manière, par exemple :

```
macro mov op1,op2
 {
  if op1 in <ds,es,fs,gs,ss> & op2 in <cs,ds,es,fs,gs,ss>
    push  op2
    pop   op1
  else
    mov   op1,op2
  end if
 }
```

Cette macro-instruction étend la syntaxe de l'instruction **MOV**, permettant aux deux opérandes d'être des registres de segment. Par exemple `mov ds,es` sera assemblé comme `push es` suivi de `pop ds`. Dans tous les autres cas, l'instruction MOV standard sera utilisée. La syntaxe de MOV peut être étendue en définissant la macro-instruction suivante de ce nom, qui utilisera la macro-instruction précédente :

```
macro mov op1,op2,op3
 {
  if op3 eq
    mov   op1,op2
  else
    mov   op1,op2
    mov   op2,op3
  end if
 }
```

Elle permet à l'instruction **MOV** d'avoir trois opérandes, tout en sachant qu'elle ne peut toujours avoir que deux opérandes, car lorsque la macro-instruction reçoit moins d'arguments que nécessaire, le reste des arguments aura des valeurs vides. Lorsque trois opérandes sont donnés, cette macro-instruction deviendra deux macro-instructions de la définition précédente. Donc, l'instruction `mov es, ds, dx` sera assemblée comme `push ds`, `pop es` suivi de `mov ds, dx`.

En plaçant le caractère * après le nom de l'argument, vous pouvez marquer l'argument comme requis. Le préprocesseur ne lui permettra pas d'avoir une valeur vide. Par exemple, la macro-instruction ci-dessus pourrait être déclarée `macro mov op1*, op2*, op3` pour s'assurer que les deux premiers arguments devront toujours avoir des valeurs non vides.

Vous pouvez également fournir la valeur par défaut de l'argument, en plaçant le symbole `=` suivi de la valeur après le nom de l'argument. Ensuite, si l'argument a une valeur vide, la valeur par défaut sera utilisée à la place.

Lorsqu'il est nécessaire de fournir une macro-instruction avec un argument contenant des virgules, un tel argument doit être placé entre les caractères < et >. S'il contient plus d'un caractère `<`, le même nombre de caractères `>` doit être utilisé pour indiquer que la valeur de l'argument se termine.

Lorsque le nom du dernier argument de macro-instruction est suivi d'un caractère `&`, cet argument consomme tout jusqu'à la fin de la ligne, y compris les virgules.

La directive **PURGE** permet de supprimer la dernière définition de la macro-instruction spécifiée. Elle doit être suivie d'un ou plusieurs noms de macro-instructions, séparés par des virgules. Si une telle macro-instruction n'a pas été définie, vous n'obtiendrez aucune erreur. Par exemple, après avoir étendu la syntaxe de `mov` avec les macro-instructions définies ci-dessus, vous pouvez désactiver la syntaxe avec trois opérandes en utilisant la directive PURGE MOV. La directive PURGE MOV suivante désactivera également la syntaxe pour deux opérandes étant des registres de segment, et toutes les directives suivantes de ce type ne feront rien.

Si, après la directive **MACRO**, vous placez un groupe de déclarations d'arguments entre crochets, cela permettra de donner plus de valeurs pour ce groupe d'arguments lors de l'utilisation de cette macro-instruction. Tout argument supplémentaire suivant le dernier argument d'un tel groupe démarrera le nouveau groupe et deviendra le premier argument de celui-ci. Pour cette raison, après le crochet fermant, aucun autre nom d'argument ne peut suivre. Le contenu de la macro-instruction sera traité pour chacun de ces groupes d'arguments séparément. L'exemple le plus simple consiste à mettre un nom d'argument entre crochets :

```
macro stoschar [char]
 {
    mov al,char
    stosb
 }
```

Cette macro-instruction accepte un nombre illimité d'arguments, et chacun sera traité séparément dans ces deux instructions. Par exemple `stoschar 1, 2, 3` sera assemblé avec les instructions suivantes :

```
mov al,1
stosb
mov al,2
stosb
mov al,3
stosb
```

Certaines directives spéciales ne sont disponibles que dans les définitions de macro-instructions. La directive LOCAL définit les noms locaux qui seront remplacés par des valeurs uniques à chaque fois que la macro-instruction est utilisée. Elle doit être suivie de noms séparés par des virgules. Si le nom donné en paramètre à la directive `local` commence par un point ou deux points, les étiquettes uniques générées par chaque évaluation de macro-instruction auront les mêmes propriétés. Cette directive est généralement nécessaire pour les constantes ou les étiquettes que la macro-instruction définit et utilise en interne. Par exemple :

```
macro movstr
 {
    local move
```

```
 move:
   lodsb
   stosb
   test al,al
   jnz move
}
```

Chaque fois que cette macro-instruction est utilisée, `move` deviendra un autre nom unique dans ses instructions, de sorte que vous n'obtiendrez pas une erreur que vous obtenez normalement lorsqu'une étiquette est définie plus d'une fois.

Les directives **FORWARD**, **REVERSE** et **COMMON** divisent la macro-instruction en blocs, chacun étant traité après la fin du traitement de la précédente. Leur comportement diffère uniquement si la macro-instruction autorise plusieurs groupes d'arguments. Le bloc d'instructions qui suit la directive FORWARD est traité pour chaque groupe d'arguments, du premier au dernier - exactement comme le bloc par défaut (non précédé par aucune de ces directives). Le bloc qui suit la directive REVERSE est traité pour chaque groupe d'arguments dans l'ordre inverse – du dernier au premier. Le bloc qui suit la directive COMMON n'est traité qu'une seule fois, généralement pour tous les groupes d'arguments. Le nom local défini dans l'un des blocs est disponible dans tous les blocs suivants lors du traitement du même groupe d'arguments que lors de sa définition, et lorsqu'il est défini dans un bloc commun, il est disponible dans tous les blocs suivants sans dépendre de la manière dont le groupe d'arguments est traité.

Voici un exemple de macro-instruction qui créera la table d'adresses aux chaînes suivies de ces chaînes:

```
macro strtbl name,[string]
 {
  common
    label name dword
  forward
    local label
    dd label
  forward
    label db string,0
 }
```

Le premier argument donné à cette macro-instruction deviendra l'étiquette de la table d'adresses. Les arguments suivants devraient être les chaînes. Le premier bloc est traité une seule fois et définit l'étiquette. Le second bloc pour chaque chaîne déclare son nom local et définit l'entrée de table contenant l'adresse de cette chaîne. Le troisième bloc définit les données de chaque chaîne avec l'étiquette correspondante.

La directive commençant le bloc dans la macro-instruction peut être suivie de la première instruction de ce bloc dans la même ligne, comme dans l'exemple suivant :

```
macro stdcall proc,[arg]
 {
  reverse push arg
  common call proc
 }
```

Cette macro-instruction peut être utilisée pour appeler les procédures en utilisant la convention STDCALL, qui a tous les arguments poussés sur la pile dans l'ordre inverse. Par exemple `stdcall foo,1,2,3` sera assemblé comme :

```
push 3
push 2
push 1
call foo
```

Si un nom à l'intérieur d'une macro-instruction a plusieurs valeurs (c'est, soit l'un des arguments entre crochets, soit un nom local défini dans le bloc suivant les directives FORWARD ou REVERSE) et est utilisé dans le bloc suivant la directive COMMON, il sera remplacé par toutes ses valeurs, séparées par des virgules. Par exemple, la macro-instruction suivante transmettra tous les arguments supplémentaires à la macro-instruction STDCALL précédemment définie :

```
macro invoke proc,[arg]
 { common stdcall [proc],arg }
```

Elle peut être utilisée pour appeler indirectement (par le pointeur stocké en mémoire) la procédure utilisant la convention STDCALL.

À l'intérieur de la macro-instruction, un opérateur spécial `#` peut également être utilisé. Cet opérateur provoque la concaténation de deux noms en un seul nom. Cela peut être utile, car il est effectué après le remplacement des

arguments et des noms locaux par leurs valeurs. La macro-instruction suivante générera le saut conditionnel selon l'argument cond :

```
macro jif op1,cond,op2,label
 {
    cmp op1,op2
    j#cond label
 }
```

Par exemple `jif ax,ae,10h,exit` sera assemblé comme s'il s'agissait des instructions `cmp ax,10h` et `jae exit`.

L'opérateur # peut également être utilisé pour concaténer deux chaînes entre guillemets en une seule. La conversion du nom en chaîne entre guillemets est également possible, avec l'opérateur ‘ , qui peut également être utilisé dans la macro-instruction. Il convertit le nom qui le suit en une chaîne entre guillemets - mais notez que lorsqu'il est suivi d'un argument de macro qui est remplacé par une valeur contenant plus d'un symbole, seul le premier d'entre eux sera converti, car l'opérateur ‘ ne convertit qu'un symbole qui le suit immédiatement. Voici un exemple d'utilisation de ces deux fonctionnalités :

```
macro label name
 {
    label name
    if ~ used name
      display ‘name # " is defined but not used.",13,10
    end if
 }
```

Lorsque l'étiquette définie avec une telle macro n'est pas utilisée dans la source, la macro vous en avertit avec un message, en précisant de quelle étiquette il s'agit.

Pour que la macro-instruction se comporte différemment lorsque certains des arguments sont d'un type spécial, par exemple une chaîne entre guillemets, vous pouvez utiliser l'opérateur de comparaison **EQTYPE**. Voici un exemple de l'utilisation de cet opérateur pour distinguer une chaîne entre guillemets d'un autre argument :

```
macro message arg
 {
  if arg eqtype ""
    local str
    jmp   @f
    str   db arg,0Dh,0Ah,24h
    @@:
    mov   dx,str
  else
    mov   dx,arg
  end if
    mov   ah,9
    int   21h
 }
```

La macro ci-dessus est conçue pour afficher des messages dans les programmes DOS. Lorsque l'argument de cette macro est un nombre, une étiquette ou une variable, la chaîne de cette adresse est affichée, mais lorsque l'argument est une chaîne entre guillemets, le code créé affichera cette chaîne suivie du retour chariot et du saut de ligne.

Il est également possible de mettre une déclaration de macro-instruction à l'intérieur d'une autre macro-instruction, afin qu'une macro puisse en définir une autre, mais il y a un problème avec de telles définitions résultant du fait que le caractère } ne peut pas apparaître dans la macro-instruction, car cela signifie toujours la fin de la définition. Pour surmonter ce problème, l'échappement des symboles dans la macro-instruction peut être utilisé. Cela se fait en plaçant une ou plusieurs barres obliques inverses devant tout autre symbole (même le caractère spécial). Le préprocesseur voit une telle séquence comme un symbole unique, mais chaque fois qu'il rencontre un tel symbole pendant le traitement de la macro-instruction, il coupe le caractère de barre oblique inverse à l'avant de celui-ci. Par exemple, \} est traité comme un symbole unique, mais pendant le traitement de la macro-instruction, il devient le symbole }. Cela permet de mettre une définition de macro-instruction dans une autre :

```
macro ext instr
 {
  macro instr op1,op2,op3
   \{
    if op3 eq
```

```
      instr op1,op2
    else
      instr op1,op2
      instr op2,op3
    end if
   \}
 }

ext add
ext sub
```

La macro `ext` est définie correctement, mais lorsqu'elle est utilisée, les symboles `\{` et `\}` deviennent les symboles `{` et `}`. Ainsi, lorsque la ligne `ext add` est traitée, le contenu de la macro devient une définition valide d'une macro-instruction et de cette façon la macro **ADD** devient définie. De la même manière, la ligne `ext sub` définit la macro **SUB**. L'utilisation du symbole `\{` n'était pas vraiment nécessaire ici, mais c'est la bonne façon de procéder pour rendre la définition plus claire.

Si certaines directives spécifiques aux macro-instructions, comme LOCAL ou COMMON sont nécessaires dans une macro intégrée de cette façon, elles peuvent être échappées de la même manière. Échapper le symbole avec plus d'une barre oblique inverse est également autorisé, ce qui permet plusieurs niveaux d'imbrication des définitions de macro-instructions.

L'autre technique pour définir une macro-instruction par une autre est d'utiliser la directive FIX, qui devient utile lorsqu'une macro-instruction ne fait que commencer la définition d'une autre, sans la fermer. Par exemple :

```
macro tmacro [params]
 {
  common macro params {
 }

MACRO fix tmacro
ENDM fix }
```

définit une syntaxe alternative pour définir les macro-instructions, qui ressemble à :

```
MACRO stoschar char
    mov al,char
    stosb
ENDM
```

Notez que le symbole qui a une telle définition personnalisée doit être défini avec une directive FIX, car seules les constantes symboliques prioritaires sont traitées avant que le préprocesseur ne recherche le caractère `}` lors de la définition de la macro. Cela peut poser un problème si l'on doit effectuer des tâches supplémentaires à la fin d'une telle définition, mais il existe une fonctionnalité supplémentaire qui aide dans de tels cas. À savoir, qu'il est possible de placer n'importe quelle directive, instruction ou macro-instruction juste après le caractère `}` qui termine la macro-instruction et elle sera traitée de la même manière que si elle était placée dans la ligne suivante. La directive `"postpone"` peut être utilisée pour définir un type spécial de macro-instruction qui n'a pas de nom ou d'arguments et sera automatiquement appelée lorsque le préprocesseur atteindra la fin de la source:

```
postpone
 {
  code_size = $
 }
```

C'est une sorte de macro-instruction très simplifiée et elle délègue simplement un bloc d'instructions à mettre à la fin.

2.3.4 Structures

La directive **STRUC** est une variante spéciale de directive **MACRO** utilisée pour définir les structures de données. La macro-instruction définie à l'aide de la directive STRUC doit être précédée d'une étiquette (comme la directive de définition de données) lorsqu'elle est utilisée. Cette étiquette sera également attachée au début de chaque nom commençant par un point dans le contenu de la macro-instruction. La macro-instruction définie à l'aide de la directive STRUC peut avoir le même nom qu'une autre macro-instruction définie à l'aide de la directive MACRO, la macro-instruction de structure n'empêchera pas la macro-instruction standard d'être traitée lorsqu'il n'y a pas d'étiquette devant elle et vice versa. Toutes les règles et fonctionnalités concernant les macro-instructions standard s'appliquent aux macro-instructions de structure.

Voici l'exemple de macro-instruction de structure :

```
struc point x,y
```

```
{
   .x dw x
   .y dw y
}
```

Par exemple `my point 7,11`, définira une structure étiquetée `my`, composée de deux variables : `my.x` avec la valeur 7 et `my.y` avec la valeur 11.

Si quelque part dans la définition de la structure le nom se compose d'un seul point qu'il a trouvé, il est remplacé par le nom de l'étiquette pour l'instance de structure donnée et cette étiquette ne sera pas définie automatiquement dans ce cas, permettant de personnaliser complètement la définition. L'exemple suivant utilise cette fonctionnalité pour étendre la directive de définition de données `db` avec la possibilité de calculer la taille des données définies :

```
struc db [data]
 {
   common
    . db data
    .size = $ - .
 }
```

Avec une telle définition, `msg db 'Hello!',13,10` définira également `msg.size` constante, égale à la taille des données définies en octets.

La définition des structures de données adressées par des registres ou des valeurs absolues doit être effectuée à l'aide de la directive **VIRTUAL** avec macro-instruction de structure (voir paragraphe 2.2.4).

La directive **RESTRUC** supprime la dernière définition de la structure, tout comme le fait **PURGE** avec les macro-instructions et **RESTORE** avec les constantes symboliques. Il a également la même syntaxe - doit être suivi d'un ou plusieurs noms de macro-instructions de structure, séparés par des virgules.

2.3.5 Répétition des macro-instructions

La directive **REPT** est un type spécial de macro-instruction qui crée une quantité donnée de doublons du bloc entre accolades. La syntaxe de base est une directive REPT suivie d'un nombre, puis bloc source entre les caractères { et }. L'exemple le plus simple :

```
rept 5 { in al,dx }
```

produira cinq doublons de la ligne `in al,dx`. Le bloc d'instructions est défini de la même manière que pour la macro-instruction standard et tous les opérateurs et directives spéciaux qui ne peuvent être utilisés que dans des macro-instructions sont également autorisés ici. Lorsque le nombre donné est nul, le bloc est simplement ignoré, comme si vous aviez défini une macro-instruction mais que vous ne l'auriez jamais utilisée. Le nombre de répétitions peut être suivi du nom du symbole du compteur, qui sera remplacé symboliquement par le nombre de doublons actuellement générés. Donc, cette formulation :

```
rept 3 counter
 {
    byte#counter db counter
 }
```

génèrera les lignes :

```
byte1 db 1
byte2 db 2
byte3 db 3
```

Le mécanisme de répétition appliqué aux blocs REPT est le même que celui utilisé pour traiter plusieurs groupes d'arguments pour les macro-instructions, donc les directives telles que `forward`, `common` et `reverse` peuvent être utilisées dans leur sens habituel. Ainsi une telle macro-instruction :

```
rept 7 num { reverse display `num }
```

affichera les chiffres de 7 à 1 sous forme de texte. La directive `local` se comporte de la même manière que dans une macro-instruction avec plusieurs groupes d'arguments, donc :

```
rept 21
 {
   local label
   label: loop label
 }
```

générera une étiquette unique pour chaque duplicata.

Le symbole du compteur par défaut compte à partir de 1, mais vous pouvez déclarer une valeur de base différente en plaçant le nombre précédé de deux points immédiatement après le nom du compteur. Par exemple :

```
rept 8 n:0 { pxor xmm#n, xmm#n }
```

générera du code qui effacera le contenu de huit registres SSE. Vous pouvez définir plusieurs compteurs séparés par des virgules, et chacun peut avoir une base différente.

Le nombre de répétitions et les valeurs de base des compteurs peuvent être spécifiés à l'aide d'expressions numériques avec des règles d'opérateur identiques à celles de l'assembleur. Cependant, chaque valeur utilisée dans une telle expression doit être soit un nombre directement spécifié, soit une constante symbolique dont la valeur est également une expression qui peut être calculée par le préprocesseur (dans ce cas, la valeur de l'expression associée à la constante symbolique est calculée en premier, puis remplacée dans l'expression extérieure à la place de cette constante). Si vous avez besoin de répétitions basées sur des valeurs qui ne peuvent être calculées qu'au moment de l'assemblage, utilisez l'une des directives de répétition de code qui sont traitées par l'assembleur. Voir la section 2.2.3.

La directive **IRP** itère l'argument unique dans la liste de paramètres donnée. La syntaxe est `irp` suivi du nom de l'argument, puis de la virgule et enfin de la liste des paramètres. Les paramètres sont spécifiés de la même manière que lors de l'invocation d'une macro-instruction standard. Ils doivent donc être séparés par des virgules et chacun peut être entouré des caractères < et >. Le nom de l'argument peut également être suivi de * pour indiquer qu'il ne peut pas obtenir une valeur vide. Un tel bloc :

```
irp value, 2,3,5
 { db value }
```

générera des lignes:

```
db 2
db 3
db 5
```

La directive **IRPS** parcourt la liste de symboles donnée. Elle doit être suivie du nom de l'argument, puis de la virgule et enfin, de la séquence de tous les symboles. Chaque symbole de cette séquence, qu'il s'agisse du symbole de nom, du caractère de symbole ou de la chaîne entre guillemets, devient une valeur d'argument pour une itération. S'il n'y a aucun symbole après la virgule, aucune itération n'est effectuée. Cet exemple :

```
irps reg, al bx ecx
 { xor reg,reg }
```

générera les lignes:

```
xor al,al
xor bx,bx
xor ecx,ecx
```

La directive **IRPV** parcourt toutes les valeurs affectées à la variable symbolique donnée. Elle doit être suivie du nom de l'argument et du nom de la variable symbolique, séparés par une virgule. Lorsque la variable symbolique est traitée avec une directive **RESTORE** pour supprimer sa dernière valeur, cette valeur est supprimée de la liste des valeurs accessibles par IRPV. Mais toute modification apportée à cette liste lors des itérations effectuées par IRPV (soit en définissant une nouvelle valeur pour la variable symbolique, soit en détruisant la valeur avec la directive RESTORE) n'affecte pas l'opération effectuée par cette directive. La liste qui est itérée reflète l'état de symbolique variable au moment où la directive IRPV a été rencontrée. Par exemple, cet extrait restaure une variable symbolique appelée `d` à son état initial, avant que des valeurs ne lui soient attribuées :

```
irpv value, d
 { restore d }
```

Il génère simplement autant de copies de la directive RESTORE, autant de valeurs à supprimer.

Les blocs définis par les directives IRP, IRPS et IRPV sont également traités de la même manière que toutes les macro-instructions, de sorte que les opérateurs et les directives spécifiques aux macro-instructions peuvent également être librement utilisés dans ce cas.

2.3.6 Prétraitement conditionnel

La directive **MATCH** fait qu'un bloc source est prétraité et transmis à l'assembleur uniquement lorsque la séquence de symboles donnée correspond au modèle spécifié. Le modèle vient en premier, terminé par une virgule, puis les symboles qui doivent être mis en correspondance avec le modèle, et enfin le bloc source, entouré d'accolades comme macro-instruction. Il y a les quelques règles pour construire l'expression pour la correspondance. La première est que tous les caractères symboliques et toute chaîne entre guillemets doivent correspondre exactement tels quels. Dans cet exemple :

```
match +,+ { include 'first.inc' }
match +,- { include 'second.inc' }
```

le premier fichier sera inclus, car + après la virgule correspond au + dans le modèle, et le second fichier ne sera pas inclus, car il n'y a pas de correspondance.

Pour correspondre littéralement à tout autre symbole, il doit être précédé d'un caractère = dans le modèle. Aussi pour correspondre au caractère = lui-même, ou à la virgule, les constructions == et =doivent être utilisées. Par exemple, le motif =a== correspondra à la séquence a=.

Si un symbole de nom est placé dans le modèle, il correspond à toute séquence composée d'au moins un symbole, puis ce nom est remplacé par la séquence correspondante partout à l'intérieur du bloc suivant, de manière analogue aux paramètres de macro-instruction. Par exemple :

```
match a-b, 0-7
 { dw a,b-a }
```

génèrera l'instruction dw 0,7-0. Chaque nom est toujours associé à aussi peu de symboles que possible, laissant le reste pour les suivants, donc dans ce cas :

```
match a b, 1+2+3 { db a }
```

le nom a correspondra au symbole 1, laissant la séquence +2+3 à associer à b. Mais dans ce cas :

```
match a b, 1 { db a }
```

il ne restera plus rien pour correspondre à b. Donc le bloc ne sera pas traité du tout.

Le bloc source défini par MATCH est traité de la même manière que toute macro-instruction. Donc, tous les opérateurs spécifiques aux macro-instructions peuvent également être utilisés dans ce cas.

Ce qui rend la directive MATCH plus utile est le fait qu'elle remplace les constantes symboliques par leurs valeurs dans la séquence de symboles correspondante (c'est-à-dire partout après la virgule jusqu'au début du bloc source) avant d'effectuer la correspondance. Grâce à cela, il peut être utilisé par exemple pour traiter un bloc source à condition qu'une constante symbolique ait la valeur donnée, comme :

```
match =TRUE, DEBUG { include 'debug.inc' }
```

qui n'inclura le fichier que lorsque la constante symbolique DEBUG a été définie avec valeur TRUE.

2.3.7 Ordre de traitement

Lors de la combinaison de diverses fonctionnalités du préprocesseur, il est important de connaître l'ordre dans lequel ces fonctionnalités sont traitées. Comme il a déjà été noté, la plus haute priorité va à la directive FIX et aux remplacements définis avec elle. Ce traitement est fait complètement avant de faire tout autre prétraitement. Il en résulte que ce morceau de script source :

```
V fix {
  macro empty
   V
V fix }
   V
```

devient une définition valide d'une macro-instruction empty. On peut considérer que la directive **FIX** et les constantes symboliques prioritaires sont traitées dans une étape distincte, et que tous les autres prétraitements sont effectués après sur la source résultante.

Le prétraitement standard qui vient après, sur chaque ligne commence par la reconnaissance du premier symbole. Il commence par vérifier les directives du préprocesseur, et lorsqu'aucune d'entre elles n'est détectée, le préprocesseur vérifie si le premier symbole est une macro-instruction. Si aucune macro-instruction n'est trouvée, elle se déplace vers le deuxième symbole de ligne, et recommence par la vérification des directives, qui dans ce cas n'est que la directive **EQU**, car c'est la seule qui apparaît comme deuxième symbole de la ligne. S'il n'y a pas de directive, le deuxième symbole est vérifié pour le cas d'une macro-instruction de structure et lorsqu'aucune de ces vérifications ne donne le résultat positif, les constantes symboliques sont remplacées par leurs valeurs et cette ligne est passée à l'assembleur.

Pour le voir sur l'exemple, considérons la macro-instruction appelée foo et la macro-instruction de structure appelée bar. Ces lignes :

```
foo equ
foo bar
```

seront alors tous deux interprétés comme des invocations de macro foo, puisque la signification du premier symbole l'emporte sur celle du second.

Lorsque la macro-instruction génère les nouvelles lignes à partir de son bloc de définition, dans chaque ligne, elle recherche d'abord les directives de macro-instruction et les interprète en conséquence. Tout le reste du contenu du bloc de définition est utilisé pour créer les nouvelles lignes, en remplaçant les paramètres par leurs valeurs, puis en traitant le symbole échappant aux opérateurs et # et ‘ . L'opérateur de conversion a la priorité plus élevée que la concaténation et si l'un d'entre eux opère sur le symbole échappé, l'échappement est annulé avant de terminer l'opération. Une fois cette opération terminée, la ligne nouvellement générée passe par le prétraitement standard, comme décrit ci-dessus.

Bien que les constantes symboliques ne soient généralement remplacées que dans les lignes, où aucune directive de préprocesseur ni macro-instruction n'a été trouvée, il existe des cas particuliers où ces remplacements sont effectués dans les parties de lignes contenant des directives. La première est la définition de la constante symbolique, où les remplacements sont effectués partout après le mot-clé EQU et la valeur résultante est ensuite affectée à la nouvelle constante (voir le paragraphe 2.3.2). Le deuxième cas est la directive MATCH, où les remplacements sont effectués dans les symboles après la virgule avant de les faire correspondre avec le motif. Ces fonctionnalités peuvent être utilisées par exemple pour maintenir les listes, comme dans cet ensemble de définitions :

```
list equ

macro append item
 {
   match any, list \{ list equ list,item \}
   match , list \{ list equ item \}
 }
```

La constante `list` est initialisée ici avec une valeur vide, et la macro-instruction `append` peut être utilisée pour ajouter les nouveaux éléments dans cette liste en les séparant par des virgules. La première correspondance dans cette macro-instruction se produit uniquement lorsque la valeur de list n'est pas vide (voir le paragraphe 2.3.6). Dans ce cas, la nouvelle valeur de la liste est la précédente avec la virgule et le nouvel élément ajouté à la fin. La deuxième correspondance se produit uniquement lorsque la liste est encore vide, et dans ce cas, la liste est définie pour ne contenir que le nouvel élément. Donc, en commençant par la liste vide, le `append 1` définirait `list equ 1` et les `append 2` suivants, il définirait `list equ 1,2`. On pourrait alors avoir besoin d'utiliser cette liste comme paramètres d'une macro-instruction. Mais cela ne peut pas être fait directement. Si `foo` est la macro-instruction, alors `foo list` passerait simplement le symbole `list` comme paramètre à la macro, puisque les constantes symboliques ne sont pas déroulées à ce stade. À cette fin, la directive MATCH est à nouveau utile :

```
match params, list { foo params }
```

La valeur de `list`, si elle n'est pas vide, correspond au mot-clé `params`, qui est ensuite remplacé par la valeur correspondante lors de la génération des nouvelles lignes définies par le bloc entre accolades. Donc, si la valeur `list` avait `1,2`, la ligne ci-dessus générerait la ligne contenant `foo 1,2`, qui passerait ensuite par le prétraitement standard.

L'autre cas particulier se situe dans les paramètres de directive REPT. Le nombre de répétitions et la valeur de base du compteur peuvent être spécifiés à l'aide d'expressions numériques, et s'il existe une constante symbolique avec un nom non numérique utilisé dans une telle expression, le préprocesseur essaie d'évaluer sa valeur en tant qu'expression numérique et s'il réussit, il remplace la constante symbolique par le résultat de ce calcul et continue d'évaluer l'expression principale. Si l'expression à l'intérieur de ces constantes symboliques contient également des constantes symboliques, le préprocesseur essaiera de calculer toutes les valeurs nécessaires de manière récursive.

Cela permet d'effectuer certains calculs au moment du prétraitement, à condition que toutes les valeurs utilisées soient les nombres connus au stade du prétraitement. Une seule répétition avec REPT peut être utilisée dans le seul but de calculer une valeur, comme dans cet exemple :

```
define a b+4
define b 3
rept 1 result:a*b+2 { define c result }
```

Pour calculer la valeur de base pour le `result` compteur, le préprocesseur remplace le `b` à sa valeur , et calcule de manière récursive la valeur de `a`, l'obtention de 7 à la suite, puis il calcule l'expression principale, le résultat étant 23. Le cobtient alors défini avec la première valeur de compteur (car le bloc n'est traité qu'une seule fois), qui est le résultat du calcul, donc la valeur de `c` est un symbole simple `23`. Notez que si `b` est redéfinie ultérieurement avec une autre valeur numérique, la prochaine fois et l'expression contenant `a` est calculée, la valeur de `a` reflétera la nouvelle valeur de `b`, car la constante symbolique contient uniquement le texte de l'expression.

Il y a un autre cas particulier - lorsque le préprocesseur va vérifier le deuxième symbole de la ligne et qu'il se trouve qu'il s'agit du caractère deux-points (ce qui est ensuite interprété par l'assembleur comme la définition d'une étiquette), il s'arrête à cet endroit et termine le prétraitement de le premier symbole (donc si c'est la constante symbolique qu'il se déroule) et s'il semble toujours être l'étiquette, il effectue le prétraitement standard à partir de l'endroit après l'étiquette. Cela permet de placer des directives de préprocesseur et des macro-instructions après les étiquettes, de manière analogue aux instructions et directives traitées par l'assembleur, comme :

```
start: include 'start.inc'
```

Cependant, si l'étiquette se casse pendant le prétraitement (par exemple lorsqu'il s'agit de la constante symbolique à valeur vide), seul le remplacement des constantes symboliques est poursuivi pour le reste de la ligne.

Il faut se rappeler que les tâches effectuées par le préprocesseur sont les opérations préliminaires sur les sym-

boles de textes, qui sont effectuées en un seul passage avant le processus principal d'assemblage. Le texte qui est le résultat du prétraitement est passé à l'assembleur, et il effectue ensuite ses multiples passages dessus. Ainsi les directives de contrôle, qui ne sont reconnues et traitées que par l'assembleur – car elles dépendent des valeurs numériques qui peuvent même varier entre les passes – ne sont en aucun cas reconnues par le préprocesseur et n'ont aucun effet sur le prétraitement. Considérons cet exemple de source :

```
if 0
a = 1
b equ 2
end if
dd b
```

Lorsqu'il est prétraité, la seule directive reconnue par le préprocesseur est EQU, qui définit la constante symbolique b, donc plus tard dans la source, le symbole b est remplacé par la valeur 2. Hormis ce remplacement, les autres lignes sont transmises sans modification à l'assembleur. Ainsi, après le prétraitement, la source ci-dessus devient:

```
if 0
a = 1
end if
dd 2
```

Désormais, lorsque l'assembleur le traite, la condition pour le IF est fausse et la constante a n'est pas définie. Cependant, la constante symbolique b est traitée normalement, même si sa définition est placée juste à côté de celle de a. Donc, en raison de la confusion possible, vous devez être très prudent à chaque fois que vous mélangez les fonctionnalités du préprocesseur et de l'assembleur. Dans de tels cas, il est important de réaliser ce que deviendra la source après le prétraitement, et donc ce que l'assembleur verra et fera lors de ses multiples passes.

2.4 Directives de formatage

Ces directives sont en fait aussi, en quelque sorte, des directives de contrôle, dans le but de contrôler le format du code généré.

La directive **FORMAT** suivie de l'identifiant de format permet de sélectionner le format de sortie. Cette directive doit être placée au début de la source. Il peut toujours être suivi dans la même ligne du mot- clé **AS** et de la chaîne entre guillemets spécifiant l'extension de fichier par défaut pour le fichier de sortie. À moins que le nom du fichier de sortie n'ait été spécifié à partir de la ligne de commande, l'assembleur utilisera cette extension lors de la génération du fichier de sortie.

Les directives **USE16** et les **USE32** forcent l'assembleur à générer du code 16 bits ou 32 bits, en omettant le paramètre par défaut pour le format de sortie sélectionné. La directive USE64 permet de générer le code pour le mode long des processeurs x86-64.

Le format de sortie par défaut est un fichier binaire plat, il peut également être sélectionné à l'aide de la directive **FORMAT BINARY**. Lorsque ce format est choisi, un symbole spécial $% peut être utilisé pour obtenir le décalage actuel dans la sortie et $%% peut être utilisé pour obtenir le décalage réel dans le fichier de sortie, en omettant toutes les données non définies qui seraient ignorées si la sortie était terminée à ce stade. De plus, pour ce format, les directives **LOAD** et **STORE** autorisent l'accès à toutes les données dans la sortie déjà générée en suivant les mots-clé from ou at avec un caractère : puis une expression spécifiant le décalage dans la sortie.

Ci-dessous sont décrits les différents formats de sortie avec les directives spécifiques à ces formats.

2.4.1 Éxécutable MZ

Pour sélectionner le format de sortie MZ, utilisez la directive **FORMAT MZ**. Le paramètre de code par défaut pour ce format est 16 bits.

La directive **SEGMENT** définit un nouveau segment. Elle doit être suivie de label, dont la valeur sera le numéro de segment défini, éventuellement le mot **USE16** ou **USE32** peut suivre pour spécifier si le code de ce segment doit être 16 bits ou 32 bits. L'origine du segment est alignée sur le paragraphe (16 octets). Toutes les étiquettes définies alors auront des valeurs relatives au début de ce segment.

La directive **ENTRY** définit le point d'entrée de l'exécutable MZ. Elle doit être suivie de l'adresse distante (nom du segment, deux-points et le décalage à l'intérieur du segment) du point d'entrée souhaité.

La directive **STACK** configure la pile pour l'exécutable MZ. Il peut être suivi d'une expression numérique spécifiant la taille de la pile à créer automatiquement ou de l'adresse éloignée du cadre de pile initial lorsque vous souhaitez configurer la pile manuellement. Lorsqu'aucune pile n'est définie, la pile de taille par défaut 4096 octets sera créée.

La directive **HEAP** doit être suivie d'une valeur 16 bits définissant la taille maximale du tas supplémentaire dans les paragraphes (il s'agit du tas en plus de la pile et des données non définies). Utilisez **HEAP 0** pour toujours allouer uniquement la mémoire dont le programme a vraiment besoin. La taille par défaut du tas est 65535.

2.4.2 Format Portable Executable

Pour sélectionner le format de sortie Portable Executable, utilisez la directive **FORMAT PE** qui peut être suivie de paramètres de format supplémentaires : d'abord le paramètre de sous-système cible, qui peut être CONSOLE ou GUI pour les applications Windows, NATIVE pour les pilotes Windows EFI, EFIboot ou EFIruntime pour l'UEFI. Il peut être suivi de la version minimale du système vers laquelle l'exécutable est ciblé (spécifiée sous forme de valeur à virgule flottante). Les mots-clés facultatifs DLL et WDM marquent le fichier de sortie comme une bibliothèque de liens dynamiques et un pilote WDM respectivement. Le mot-clé large marque l'exécutable comme étant capable de gérer des adresses supérieures à 2 Go et le mot-clé NX signale que l'exécutable est conforme à la restriction de ne pas exécuter de code résident dans des sections exécutables.

Une fois que ces paramètres peuvent suivre l'opérateur at et l'expression numérique spécifiant la base de l'image PE, puis éventuellement l'opérateur on suivi de la chaîne entre guillemets contenant le nom du fichier, sélectionne l'entête MZ personnalisé pour le programme PE (lorsque le fichier spécifié n'est pas un exécutable MZ, il est traité comme un fichier exécutable binaire *flat* et converti au format MZ). Le paramètre de code par défaut pour ce format est 32 bits. Voici un exemple de déclaration de format PE complet :

```
format PE GUI 4.0 DLL at 7000000h on 'stub.exe'
```

Pour créer un fichier PE pour l'architecture x86-64, utilisez le mot-clé PE64 au lieu de PE dans la déclaration de format. Dans ce cas, le code de mode long est généré par défaut.

La directive **SECTION** définit une nouvelle section. Elle doit être suivie d'une chaîne entre guillemets définissant le nom de la section, puis d'un ou plusieurs flags de section. Les flags disponibles sont : code, data, readable, writeable, executable, shareable, discardable, notpageable. L'origine de la section est alignée sur la page (4096 octets). Exemple de déclaration de section PE :

```
section '.text' code readable executable
```

Parmi les flags avec aussi l'un des identifiants de données PE spéciaux peuvent être spécifiés pour marquer toute la section en tant que données spéciales. Les identifiants possibles sont export, import, resource et fixups. Si la section est marquée pour contenir des corrections, elles sont générées automatiquement et aucune donnée supplémentaire ne doit être définie dans cette section. Les données de ressources peuvent également être générées automatiquement à partir du fichier de ressources. Cela peut être obtenu en écrivant l'opérateur FROM et le nom de fichier cité après l'identifiant RESOURCE. Vous trouverez ci-dessous des exemples de sections contenant des données PE spéciales :

```
section '.reloc' data readable discardable fixups
section '.rsrc' data readable resource from 'my.res'
```

La directive **ENTRY** définit le point d'entrée pour le format Portable Executable. La valeur du point d'entrée doit suivre.

La directive **STACK** définit la taille de la pile pour le format Portable Executable, la valeur de la taille de la réserve de pile doit suivre, éventuellement la valeur de la validation de pile séparée par une virgule peut suivre. Lorsque la pile n'est pas définie, elle est définie par défaut sur une taille de 4096 octets.

La directive **HEAP** choisit la taille du tas pour le format Portable Executable, la valeur de la taille de la réserve du tas doit suivre, éventuellement la valeur de la validation du tas séparée par une virgule peut suivre. Lorsqu'aucun segment de mémoire n'est défini, il est défini par défaut sur une taille de 65536 octets. Lorsque la taille de la validation de segment de mémoire n'est pas spécifiée, elle est définie par défaut sur zéro.

La directive **DATA** commence la définition des données de PE spéciales. Elle doit être suivie par l'un des identifiants de données (export, import, resource ou fixups) ou par le nombre d'entrée de données dans l'entête PE. Les données doivent être définies dans les lignes suivantes, terminées par une directive end data. Lorsque la définition des données de correction est choisie, elles sont générées automatiquement et aucune donnée supplémentaire ne doit y être définie. Il en va de même pour les données de ressource lorsque l'identifiant **RESOURCE** est suivi de l'opérateur **FROM** et du nom de fichier cité. Dans ce cas, les données sont extraites du fichier de ressources donné.

L'opérateur **RVA** peut être utilisé à l'intérieur des expressions numériques pour obtenir la RVA de l'élément adressé par la valeur à laquelle il est appliqué, c'est-à-dire le décalage par rapport à la base de l'image PE.

2.4.3 Common Object File Format (format COFF)

Pour sélectionner le format COFF (format de fichier objet commun), utilisez la directive format COFF ou format MS COFF, selon que vous souhaitez créer une variante classique (DJGPP) ou Microsoft du fichier COFF. Le paramètre de code par défaut pour ce format est 32 bits. Pour créer le fichier au format COFF de Microsoft pour l'architecture x86-64, utilisez le paramètre format MS64 COFF, dans ce cas, le code en mode long est généré par défaut.

La directive **SECTION** définit une nouvelle section. Elle doit être suivie d'une chaîne entre guillemets définissant le nom de la section, puis, éventuellement, d'un ou plusieurs flags de section. Les flags de section disponibles pour les deux variantes de COFF sont code et data, tandis que les flags readable, writeable, executable, sha-

reable, discardable, notpageable, linkremove et linkinfo ne sont disponibles qu'avec la variante COFF de Microsoft.

Par défaut, la section est alignée sur un double-mot (quatre octets). Dans le cas de la variante Microsoft COFF, un autre alignement peut être spécifié en fournissant l'opérateur ALIGN suivi de la valeur d'alignement (toute puissance de deux jusqu'à 8192) parmi les flags de section.

La directive **EXTRN** définit un symbole comme étant externe. Elle doit être suivie du nom du symbole et éventuellement de l'opérateur de taille spécifiant la taille des données étiquetées par ce symbole. Le nom du symbole peut également être précédé d'une chaîne entre guillemets contenant le nom du symbole externe et l'opérateur AS. Voici quelques exemples de déclarations de symboles externes:

```
extrn exit
extrn '__imp__MessageBoxA@16' as MessageBox:dword
```

La directive **PUBLIC** déclare le symbole existant comme public. Elle doit être suivi du nom du symbole, éventuellement suivi de l'opérateur AS et de la chaîne entre guillemets contenant le nom sous lequel le symbole doit être disponible en tant que public. Voici quelques exemples de déclarations de symboles publics :

```
public main
public start as '_start'
```

De plus, avec le format COFF, il est possible de spécifier le symbole exporté comme statique. Cela se fait en faisant précéder le nom du symbole du mot-clé **STATIC**.

Lors de l'utilisation du format COFF de Microsoft, l'opérateur **RVA** peut être utilisé dans les expressions numériques pour obtenir le RVA de l'élément adressé par la valeur à laquelle il est appliqué.

2.4.4 Format exécutable et liable

Pour sélectionner le format de sortie ELF, utilisez la directive format ELF. Le paramètre de code par défaut pour ce format est 32 bits. Pour créer un fichier ELF pour l'architecture x86-64, utilisez la directive format ELF64, dans ce cas, le code de mode long est généré par défaut.

La directive SECTION définit une nouvelle section, elle doit être suivie d'une chaîne entre guillemets définissant le nom de la section, puis peut suivre l'un ou les deux des indicateurs executable et writeable, éventuellement également l'opérateur ALIGN suivi du nombre spécifiant l'alignement de la section (il doit être la puissance de deux), si aucun alignement n'est spécifié, la valeur par défaut est utilisée, qui est 4 ou 8, selon la variante de format choisie.

Les directives EXTRN et PUBLIC ont la même signification et la même syntaxe que lorsque le format de sortie COFF est sélectionné (voir section précédente).

L'opérateur RVA peut également être utilisé dans le cas de ce format (mais pas lorsque l'architecture cible est x86-64), il convertit l'adresse en décalage par rapport à la table GOT. Il peut donc être utile de créer du code indépendant de la position. Il existe également un opérateur PLT spécial, qui permet d'appeler les fonctions externes via la table de liaison de procédure. Vous pouvez même créer un alias pour une fonction externe qui la fera toujours être appelée via PLT, avec le code comme :

```
extrn 'printf' as _printf
printf = PLT _printf
```

Pour créer un fichier exécutable, suivez la directive de choix de format avec le mot-clé executable ou dynamic et éventuellement le numéro spécifiant la marque du système d'exploitation cible (par exemple, la valeur 3 marquerait l'exécutable pour le système Linux). Avec ce format sélectionné, il est permis d'utiliser la directive ENTRY suivie de la valeur à définir comme point d'entrée du programme. D'autre part , il rend les directives EXTRN et PUBLIC indisponibles, et au lieu de SECTION, la directive SEGMENT utilisée, suivie par un ou plusieurs flags d'autorisation de segments et éventuellement un marqueur du segment exécutable spécial ELF, qui peut être interpreter, dynamic, note, gnuehframe, gnustack ou gnurelro. Les flags d'autorisation disponibles sont : readable, writeable et executable. L'origine d'un segment non spécial est alignée sur une page (4096 octets).