Flat Assembler 1.66

Manuel du Programmeur

Thomas Grysztar

version Française

Nicolas Djurovic

1er avril 2017

Table des matières

Chapitre 1

Introduction

Ce chapitre contient toutes les informations importantes dont vous aurez besoin pour commencer à utiliser flat assembler. Si vous êtes un programmeur expérimenté en assembleur, vous devriez lire au moins ce chapitre avant d'utiliser ce compilateur.

1.1 Survol du Compilateur

Flat assembler est un compilateur rapide générant du langage machine pour les processeurs x86, faisant une multitude de passages pour optimiser la taille du code généré. Il est auto-compilable et des versions pour différentes plates-formes sont fournies. Ces versions ont été conçues pour être utilisé à partir de la ligne de commande et leur fonctionnement sur les différentes plates-formes devrait être identique.

Ce document décrit aussi la version IDE conçue pour le système Windows qui utilise une interface graphique, et qui a un éditeur intégré, au lieu de la version console. Mais d'un point de vue compilation, il a exactement les mêmes fonctionnalités que la version console, et les parties suivantes (à partir du chapitre 1.2 de ce document) seront identiques à toutes les versions de flat assembler. L'exécutable de la version IDE est `fasmw.exe`, alors que la version ligne de commande est `fasm.exe`

1.1.1 Pré-requis système

Toutes les versions utilisent un processeur x86/32 bits (un 80386 minimum), cependant ils peuvent aussi concevoir des programmes pour les architectures x86/16 bits. La version console de Windows utilise n'importe quel système d'exploitation Windows 32 bits, alors que la version avec interface graphique requiert une version

4.0 ou plus du système, il fonctionnera donc sur tous systèmes compatibles avec Windows 95 et supérieure.

Les exemples fournies avec cette version implique que la variable d'environnement `INCLUDE` soit définie par le chemin du répertoire `include` de flat assembler. Si cette variable existe déjà dans votre système et contient des chemins utilisés par d'autres programmes, vous n'avez qu'à lui ajouter le nouveau chemin (les différents chemins doivent être séparés par des points-virgule). Si vous ne voulez pas définir ce type de variable, ou si vous ne savez pas comment, vous pouvez le définir pour la version avec interface graphique de flat assembler en éditant le fichier `fasmw.ini` qui est dans le même répertoire, ce fichier est créé automatiquement dés le 1^{er} lancement de `fasmw.exe`, mais vous pouvez aussi le créer par vous-même, dans ce cas vous devrez ajouter la valeur `Include` dans la section `Environment`. Par exemple, si vous avez installé les fichiers flat assembler dans le répertoire `c:\fasmw`, vous devrez insérer ces deux lignes suivantes dans votre fichier `c:\fasmw\fasmw.ini` :

```
[Environment]
Include = c:\fasmw\include
```

Si vous ne définissez pas la variable d'environnement `INCLUDE` correctement, vous devrez mettre manuellement le chemin complet des includes Windows dans chaque programme que vous allez compiler.

1.1.2 Utilisation du compilateur

Pour commencer à utiliser flat assembler, double-cliquer sur l'icône du fichier `fasmw.exe`, ou déplacez l'icône de votre fichier source dessus (drag & drop). Plus tard vous pourrez aussi ouvrir des fichiers sources avec la commande *Open* du menu *File*, ou en déplaçant les fichiers dans l'éditeur. Vous pouvez avoir plusieurs fichiers ouverts simultanément, chacun étant représenté par un onglet en haut de l'éditeur. Pour sélectionner le fichier à éditer vous n'avez qu'à cliquer sur l'onglet correspondant avec le clique gauche de votre souris. Par défaut le compilateur gère le fichier que vous êtes en train d'éditer, mais vous pouvez forcer le compilateur à gérer un fichier en particulier en cliquant sur l'onglet approprié avec le bouton droit de votre souris et en sélectionnant *Assign*. Un seul fichier peut être assigné à la fois.

Quand votre fichier est prêt à être compilé, sélectionnez la commande *Compile* dans le menu *Run*. Un résumé de la compilation sera affiché si celle-ci est réussie, sinon l'erreur sera affiché. Dans ce résumé se trouvera une information sur le nombre de passages effectué, combien de temps cela a pris, et combien d'octets ont été écrits dans le fichier final. Dans ce résumé vous avez aussi un emplacement appelé *Display* dans lequel vous trouverez tous les messages affichés grâce à la directive `display` (voir section 2.2.5). Si l'erreur est liée à une ligne spécifique de

votre code source, elle s'affichera alors dans le champ texte *Instruction* contenant l'instruction qui a causé l'erreur et uniquement si une erreur apparaît lors de cette phase du pré-processeur (autrement ce champ reste vide). Ce résumé contient aussi le champ *Source*, qui est une liste affichant les lignes de toutes les sources qui se réfèrent à cette erreur, lorsque l'on clique sur l'une des lignes de cette liste, la ligne ou se trouve l'erreur sera automatiquement sélectionnée dans l'éditeur (si le fichier n'est pas ouvert dans l'éditeur, il sera ouvert automatiquement).

La commande *Run* lance aussi le compilateur et, en cas de réussite de la compilation, il lancera la programme compilé si celui-ci est un format qui peut-être lancé dans un environnement Windows, autrement vous aurez un message vous indiquant que ce fichier ne peux pas être lancé. Si une erreur apparaît, le compilateur affichera une information identique à celle que l'on trouve avec la commande *Compile* que l'on a utilisé.

Si le compilateur vous indique qu'il n'y a pas assez de mémoire, vous pouvez accroître la mémoire normalement alloué via la boîte de dialogue *Compiler setup*, dans le menu *Options*. Dans ce menu vous pouvez donner le nombre de kilo-octets que le compilateur devra utiliser ainsi que la priorité du compilateur dans les processus système.

1.1.3 Exécuter le compilateur via la ligne de commande

Pour compiler à partir de la ligne de commande vous devez exécuter l'executable `fasm.exe` avec deux paramètres - le nom de votre ficher source et le nom du fichier destination. Si aucun second paramètre n'est donné, le nom sera généré automatiquement (en fonction du nom du fichier source). Après l'affichage de quelques courtes informations à propos du nom du programme et de sa version, le compilateur lira les données du fichier source et lancera la compilation. Quand il aura fini, le compilateur écrira le code généré dans le fichier final et affichera un résumé du processus de compilation ; sinon il affichera les erreurs qui sont apparues.

Le fichier source devra être un fichier texte et peut être créé avec n'importe quel éditeur texte. Les caractères de fin de ligne sont aussi bien acceptés par les standards DOS et Unix, les tabulations sont considérées comme des caractères "espace".

Vous pouvez aussi inclure dans la ligne de commande le paramètre `-m` suivi par un numéro, cela permettra à flat assembler de connaître le nombre de kilo-octets maximal qu'il doit utiliser. Pour la génération du fichier DOS seulement, cette option limite le nombre d'utilisation de la mémoire étendue. L'option `-p` suivi par un numéro peut être utilisé pour spécifier le nombre maximum de passages de l'assembleur. Si le code ne peut être généré dans le montant de passages spécifiés, l'assemblage se terminera avec un message d'erreur. La valeur maximum de cette option est 65536, alors que par défaut, si cette option n'est pas dans la ligne de

commande, elle est de 100.

Il n'y a aucune option qui pourrait affecter la sortie du compilateur, flat assembler demande seulement le code source comme information vraiment utile. Par exemple, pour spécifier un format de sortie différent, vous l'indiquez en utilisant la directive `format` au début de votre code source.

1.1.4 Messages du compilateur en ligne de commande

Après le succès de la compilation le compilateur affiche donc un résumé. Il nous donne le nombre de passages effectué, combien de temps cela a pris, et combien d'octets ont été écrits dans le fichier final. Voici un exemple d'un résumé de la compilation :

```
flat assembler  version 1.66
38 passes, 5.3 seconds, 77824 bytes.
```

En cas d'erreur pendant la phase de compilation, le programme affiche un message d'erreur. Par exemple, quand le compilateur ne trouve pas le fichier source, il affichera le message suivant :

```
flat assembler  version 1.66
error: source file not found.
```

Si l'erreur se trouve à un endroit particulier dans le code source, la ligne qui a causé l'erreur sera alors affichée. Le numéro de la ligne vous est donné pour que vous puissiez trouver l'erreur, par exemple :

```
flat assembler  version 1.66
example.asm [3]:
        mob     ax,1
error: illegal instruction.
```

A la 3^{eme} ligne du fichier `example.asm`, le compilateur a rencontré une instruction inconnue. Lorsque la ligne qui a causé une erreur contient une macro-instruction, alors la ligne définissant cette macro et ayant générée une instruction incorrect est affiché :

```
flat assembler  version 1.66
example.asm [6]:
        stoschar 7
example.asm [3] stoschar [1]:
        mob     al,char
error: illegal instruction.
```

A la 6^{eme} ligne du fichier `example.asm`, la macro-instruction a généré une instruction inconnue dans la 1^{ere} ligne de sa définition.

1.1.5 Formats de sortie

Par défaut, quand aucune directive `format` n'est donné dans le code source, flat assembler génère des codes d'instructions sur la sortie, créant ainsi un fichier binaire. Il génère par défaut du code 16 bits, mais vous pouvez le mettre en mode 16 bit ou en mode 32 bits simplement en utilisant les directives `use16` ou `use32`. Quelques formats de sortie, lorsqu'ils sont sélectionnés, se mettent en mode 32 bit - vous trouverez plus d'informations sur les formats utilisable dans la section 2.4.

En fonction du format de sortie sélectionné, le compilateur choisi automatiquement le type d'extension du fichier de destination.

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

1.2 Syntaxe de l'assembleur

L'information fournie ci-dessous est prévue principalement pour les programmeurs qui ont utilisés d'autres compilateurs auparavant. Si vous êtes débutant, vous devriez jeter un oeil sur les tutoriaux de la programmation en assembleur.

Flat assembler utilise par défaut la syntaxe Intel pour les instructions d'assemblage, cependant vous pouvez l'adapter en utilisant les possibilités du préprocesseur (macro-instructions et constantes symboliques). Aussi il a son propre ensemble de directives - les instructions pour le compilateur.

Tous les symboles définis dans les sources respectent la différence entre majuscule/minuscule.

Opérateur	Bits	Octets
`byte`	8	1
`word`	16	2
`dword`	32	4
`fword`	48	6
`pword`	48	6
`qword`	64	8
`tbyte`	80	10
`tword`	80	10
`dqword`	128	16

TABLE 1.1 – Taille des opérateurs

Types	Bits								
General	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		
Control	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

TABLE 1.2 – Les registres

1.2.1 Syntaxe des Instructions

En assembleur, les instructions sont séparées par des sauts de ligne et il n'y a qu'une instruction par ligne. Si cette ligne contient un point-virgule, sauf si elle se trouve dans un texte délimité par des guillemets, le reste de la ligne est considéré comme un commentaire et ne sera donc pas pris en compte par le compilateur. Si une ligne se termine avec le caractère \ (un commentaire peut suivre ce caractère), la ligne suivante se rattache à celle-ci.

Chaque ligne dans le source est une suite d'éléments qui peut être un des trois types connus. Il y a les caractères symboliques, ce sont des caractères spéciaux qui sont uniques même lorsqu'ils ne sont pas espacés des autres. Chaque `+-*/=<>[]{}:,|&~#‘` est un caractère symbolique. Pour les autres caractères, séparés des autres éléments par des espaces ou par des caractères symboliques est un caractère symbolique. Si le 1^{er} caractère est un «'» ou un «"», tous les caractères suivants, même les caractères spéciaux, deviennent une chaîne qui devra se terminer par le même caractère utilisé au début (apostrophe ou guillemet) - cependant si on trouve deux de ces caractères identiques qui se suivent (sans aucun caractère entre eux), ils sont intégrés dans la chaîne de caractères sans arrêter la délimitation de celle-ci. Les symboles autres que les caractères symboliques ou que les chaînes de caractères peuvent être utilisés comme étant des noms, aussi définies comme noms symboliques.

Chaque instruction est représentée par un mnémonique (nom symbolique) et un nombre varié d'opérandes (constante, symbolique ou pas, ou une variable) séparé par des virgules. L'opérande peut être un registre, une valeur immédiate ou une donnée adressée en mémoire, il peut aussi être précédé par un opérateur de taille pour définir ou augmenter sa taille (Tableau 1.1). Les noms des registres disponibles sont dans le Tableau 1.2, leur taille ne peut être augmentée. Les valeurs immédiates

peuvent être indiquées par une expression numérique.

Quand l'opérande est une donnée en mémoire, l'adresse de cette donnée (ou expression numérique mais dans un registre) doit être mise entre crochets (`[ ]`) ou est précédé par l'opérateur `ptr`. Par exemple l'instruction `mov eax,3` met la valeur immédiate 3 dans le registre `eax`, l'instruction `mov eax,[7]` met la valeur (32 bit) se trouvant à l'adresse 7 dans le registre `eax` et l'instruction `mov byte [7],3` met la valeur immédiate 3 dans l'octet se trouvant à l'adresse 7, on peut aussi l'écrire `mov byte ptr 7,3`. Pour indiquer quel registre de segment doit être utilisé pour l'adressage, son nom suivi par deux points (`:`) devra être mis juste avant la valeur de l'adresse (entre les crochets ou après l'opérateur `ptr`) :

```
mov word [es:bx],ax
```

1.2.2 Définitions des données

Pour définir une donnée ou réserver de la place pour elle, il faut utiliser une des directives donnée dans le Tableau 1.3. La directive de définition de donnée doit être suivie par une ou plusieurs expressions numériques séparées par des virgules. Suivant la directive utilisée ces expressions définissent la taille des valeurs pour les données. Par exemple `db 1,2,3` défini trois octets de valeur 1, 2 et 3.

Les directives `db` et `du` acceptent aussi les chaînes de caractères de n'importe quelle taille, qui seront converties en octet (8 bits) quand `db` sera utilisé et en word (16 bits) avec la valeur haute égale à zéro (0) quand `du` est utilisé. Par exemple `db 'abc'` définira les 3 octets de valeurs 61, 62 et 63.

Les directives `dp` et `df` acceptent les valeurs définies comme étant deux expressions numériques séparés par (`:`), la première valeur deviendra un mot (`word`) de poids fort, tandis que la seconde valeur sera un double word de poids faible de la valeur du pointeur en-dehors du segment courant. La directive `dd` accepte aussi ce type de pointeur composé de deux valeurs word séparées par (`:`) et `dt` accepte comme valeur un word et un quadruple word séparées par (`:`), le quad word est stocké en premier. La directive `dt` avec une seule expression comme paramètre accepte les valeurs en virgule flottante et créé les données en FPU (Floating Point Unit/Unité de calcul en virgule flottante) format étendue double précision.

Chacune de ces directives permet l'utilisation de l'opérateur spécial `dup` permet de faire des copies multiples de valeur. Le multiplicateur doit précédé cette directive et les valeurs à dupliquer sont après celle-ci - on peut mettre plusieurs valeurs, tant qu'elles sont séparées par des virgules et délimitées par des parenthèses, comme `db 5 dup (1,2)`, on défini 5 copies des deux octets soit, après la phase de pré-processus, `db 1,2,1,2,1,2,1,2,1,2`.

`file` est une directive spéciale et sa syntaxe est différente. Cette directive inclut les octets d'un fichier et doit être suivi par le nom du fichier entre apostrophes,

en option on peut ajouter une expression numérique indiquant l'offset du fichier précédé par (:), et on peut encore ajouter une dernière option permettant d'indiquer le nombre d'octets à inclure, mettre une virgule avant la valeur (s'il n'y a pas de valeur, toutes les données lues à partir de l'offset jusqu'à la fin du fichier seront inclus). Par exemple `file 'data.bin'` insérera toutes les données du fichier binaire et `file 'data.bin':10h,4` insérera seulement 4 octets à partir de l'offset/adresse 10h.

Taille (octets)	Définir les données	Réserver les données
1	`db` `file`	`rb`
2	`dw` `du`	`rw`
4	`dd`	`rd`
6	`dp` `df`	`rp` `rf`
8	`dq`	`rq`
10	`dt`	`rt`

TABLE 1.3 – Directives des données

La directive de réservation de donnée doit être suivi par une seule expression numérique, et cette valeur définie combien de zones mémoires, d'une taille spécifiée, doit être réservée. Toutes les directives de définition des données acceptent aussi la valeur `?` qui permet d'initialiser une zone mémoire avec n'importe quelle valeur, comme `label db ?` et l'effet est identique à la directive de réservation de données. La donnée non initialisée ne sera pas incluse dans le fichier final, donc ces valeurs devront toujours être considérées comme inconnues.

1.2.3 Constantes et labels

Dans les expressions numériques vous pouvez aussi utiliser des constantes ou des labels à la place des nombres. Pour définir une constante ou un label vous devrez utiliser des directives spécifiques. Chaque label doit être défini une seule fois et sera accessible de n'importe quel endroit du source (même avant qu'il soit défini). Une constante peut être redéfinie plusieurs fois, mais sera accessible uniquement après sa définition et sera toujours égale à la dernière valeur redéfinie avant l'endroit ou elle est utilisée. Quand une constante est définie une seule fois dans le source il est, comme pour le label, accessible de n'importe ou.

La définition d'une constante se compose du nom de la constante suivi par le caractère "=" et d'une expression numérique, qui après calcul deviendra la valeur de cette 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`, et ainsi l'utiliser avec les instructions de l'assembleur comme `mov cv,count` - qui deviendra `mov cx,17` pendant le processus de compilation.

Il y a différente façon de définir des labels. Le plus simple est de mettre le nom du label suivi par (:), cette directive peut être suivie par d'autres instructions sur la même ligne. Elle donne une valeur qui est son adresse. Cette méthode est utilisée pour définir un emplacement dans le code. L'autre façon est de faire suivre le nom du label (sans les (:)) par une directive de donnée. Cela défini le label avec une valeur égale à l'adresse/offset de début de définition des données, et un label de donnée avec une taille comme il est spécifié dans le tableau 1.3.

Le label peut être considéré comme une constante de valeur égale à l'offset du code ou de la donnée fixé. Par exemple quand vous définissez une donnée utilisant cette directive `char db 224`, pour mettre l'addresse de cette valeur dans le registre `bx` vous devrez utiliser l'instruction `mov bx,char`, et pour mettre la valeur d'un octet adressé par le label `char` dans le registre `dl`, vous devrez utiliser `mov dl,[char]` (ou `mov dl,ptr char`). Mais si vous essayez d'assembler `mov ax,[char]` cela donnera une erreur, car fasm compare la taille des opérandes qui doit toujours être égale. Vous pouvez forcer l'assemblage de cette instruction en utilisant un dépassement de taille : `mov ax,word [char]`, mais rappelez-vous que cette instruction lira les deux octets se trouvant à l'adresse `char` alors qu'il n'y à qu'un octet défini.

La dernière et la plus flexible des manières de définir un label est d'utiliser la directive `label`. Cette directive doit être suivi par le nom du label puis, en option, de l'opérateur de taille et, encore en option, de l'opérateur `at` et de l'expression numérique indiquant l'adresse d'ou ce label devra être défini. Par exemple `label wchar word at char` définira un nouveau label (`wchar`) pour une donnée de 16 bits (`word`) à l'adresse `char`. L'instruction `mov ax,[wchar]` deviendra après compilation la même chose que `mov ax,word [char]`. Si aucune adresse n'est indiquée, la directive `label` défini l'adresse du label à sa position courante. Ainsi `mov [wchar],57568` copiera deux octets tandis que `mov [char],224` copiera un seul octet à la même adresse.

Le label qui a un nom commençant par un (.) sera considéré comme un label local, et son nom est attaché au nom du dernier label global (qui commence par n'importe quoi sauf un ".") pour faire le nom complet du label. Donc vous pouvez utilisez le nom court (commençant par un ".") de ce label n'importe ou avant la définition du prochain label global, et dans un autre endroit de votre source vous devez utilisez le nom complet. Il y a une exception avec les labels commençant par

deux "." - ils sont comme des labels globaux mais ne servent pas de préfixes pour les labels locaux.

Le label anonyme est défini par `@@`, vous pouvez l'utiliser plusieurs fois dans votre source. Le symbole `@b` (et son équivalent `@r`) fait référence au précédent label anonyme, alors que le symbole `@r` fait référence au label anonyme suivant. Ces symboles spéciaux ne font pas de distinctions entre minuscule et majuscule.

1.2.4 Expressions numériques

Dans les exemples que nous venons de voir, toutes les expressions numériques ne sont que de simples nombres, constantes ou labels. Mais elles peuvent être plus complexes en utilisant des opérateurs arithmétiques ou logiques pour des calculs au moment de la compilation. Les priorités des ces opérateurs se trouvent dans le tableau 1.4. Les opérations avec une priorité élevée seront calculées en premier, bien sur vous pouvez changer le comportement du ces calculs en mettant certaines instructions entre-parenthèses. `+`, `-`, `*` et `/` sont les opérateurs arithmétiques standards, `mod` donne le reste d'une division. `and`, `or`, `xor`, `shl`, `shr` et `not` ont la même fonction que les instructions assembleur de même nom. `rva` permet la conversion d'une adresse en un offset readressable et est spécifique à certain des formats de sortie (voir section 2.4).

Priorité	Opérateurs
0	`+` `-`
1	`*` `/`
2	`mod`

3	`and`
	`or`
	`xor`

4	`shl` `shr`
5	`not`
6	`rva`

TABLE 1.4 – Opérateurs arithmétiques et logiques par priorité

Les nombre dans chaque expression est considéré comme un chiffre décimal, les nombres binaires doivent avoir la lettre `b` ajouté à la fin, un nombre octal se termine par la lettre `o`, les nombres hexadécimaux commencent avec `0x` (comme

avec le langage C) ou avec le caractère `$` (comme en Pascal) ou bien se termine avec la lettre `h`. De même pour les chaînes, lorsqu'on les rencontre dans une expression, seront converties en nombre - le premier caractère deviendra l'octet le moins significatif du nombre.

L'expression numérique utilisée comme une adresse peut aussi contenir n'importe quels registres généraux utilisés pour l'adressage, ils peuvent être additionnés et multipliés par des valeurs appropriées, comme il est autorisé pour les instructions x86.

Il y a aussi quelques symboles spéciaux qui peuvent être utilisés à l'intérieur de l'expression numérique. Premièrement il y a `$`, qui est toujours égale à la valeur de l'offset courant, tandis que `$$` est égale à l'adresse de base de l'espace de l'adresse courante. Puis il y a `%`, qui est le nombre de répétition courante dans les parties du code qui sont répétées en utilisant quelques directives spéciales (voir section 2.2). Il y a aussi le symbole `%t`, qui est toujours égal à l'heure courante.

Toute expressions numériques peuvent aussi être composées d'une simple valeur en virgule flottante en notation scientifique (flat assembler n'autorise pas toute opération en virgule flottante au moment de la compilation), elles peuvent se terminer par le caractère `f` pour être reconnues, autrement elles doivent contenir au moins le caractère `.` ou le caractère `E`. Donc `1.0`, `1E0` et `1f` exprime la même valeur en virgule flottante, alors qu'un simple `1` défini une valeur entière.

1.2.5 Sauts et Appels

L'opérande de toute instruction de saut ou d'appel peut être précédé pas seulement par l'opérateur de taille, mais aussi par les opérateurs indiquant le type du saut : `short`, `near` ou `far` (*ndt : respectivement : court, proche ou loin*). Par exemple, lorsque l'assembleur est en mode 16 bits, l'instruction `jmp dword [0]` deviendra un saut lointain (far : en dehors du segment courant) et quand l'assembleur est en mode 32 bits, il deviendra un saut proche (near : dans le segment courant). Pour forcer l'instruction a être traitée différement, on utilise `jmp near dword [0]` ou `jmp far dword [0]`.

Quand l'opérande d'un saut proche (near jump) est une valeur immédiate, l'assembleur générera une variante de cette instruction de saut plus courte, si cela lui est possible (mais ne créera pas d'instruction 32 bits dans un mode 16 ni d'instruction 16 bits dans un mode 32, à moins d'utiliser un opérateur de taille avec). En indiquant le type de saut vous pouvez le forcer à toujours générer une variante longue (par exemple `jmp near 0`) ou pour générer une variante courte qui se termine avec une erreur lorsque cela devient impossible (par exemple `jmp short 0`).

1.2.6 Réglage de la taille

Quand une instruction utilise quelque adressage de mémoire, la plus petite instruction est généré par défaut en utilisant le déplacement court si la valeur de l'adresse s'ajuste à la distance. Cela peut être dépassé en utilisant l'opérateur `word` ou `dword` avant l'adresse entre-crochets (ou après l'opérateur `ptr`), qui force le long déplacement d'une taille appropriée à être faite. Au cas où l'adresse n'est relatif à aucun registre, ces opérateurs autorisent de choisir le mode approprié d'adressage absolu.

Les instructions `adc`, `add`, `and`, `cmp`, `or`, `sbb`, `sub` et `xor` avec un premier opérande étant en mode 16 ou 32 bits sont, par défaut, générées sous la forme courte 8 bits quand le deuxième opérande est ajusté en une valeur signé de 8 bits. Il peut être augmenter en mettant l'opérande `word` ou `dword` avant la valeur immédiate. Les règles similaires s'appliquent à l'instruction `imul` avec la dernière opérande étant une valeur immédiate.

Une valeur immédiate comme opérande pour l'instruction `push` sans opérateur de taille est considéré, par défaut, comme valeur "mot" (word), si l'assembleur est en mode 16 bits, et en double mot (dword) si l'assembleur est en mode 32 bits, si cela est possible on utilise la forme 8 bits de cette instruction, l'opérateur de taille `word` ou `dword` force l'instruction `push` d'être généré sous une forme longue de la taille spécifiée. Les mnémoniques `pushw` et `pushd` force l'assembleur a générer du code 16 ou 32 bits sans le forcer à utiliser une forme d'instruction longue.

Chapitre 2

Jeu d'instructions

Ce chapitre fourni les informations détaillées sur les instructions et les directives supportées par flat assembler. Les directives pour définir les labels ont été vues dans la sous-section 1.2.3, toutes les autres directives seront décrites plus tard dans ce chapitre.

2.1 Les instructions de l'architecture x86

Dans cette section vous trouverez les informations sur la syntaxe et le but des instructions du langage assembleur. Si vous avez besoin de plus d'informations techniques, veuillez vous reporter au Manuel du Développeur de Logiciel sous Architecture Intel sur le site

`http://www.intel.com/design/pentium4/manuals/index_new.htm`

Les instructions de l'assembleur se composent de mnémoniques (nom de l'instruction) et de zéro à trois opérandes. S'il y a deux opérandes ou plus, le premier est la destination et le second est la source. Chaque opérandes peut être un registre, une zone mémoire ou une valeur immédiate (voir section 1.2 pour avoir plus de détails sur la syntaxe des opérandes). Dans cette section il y a des exemples des différentes combinaisons d'opérandes (si l'instruction en est composée) après leur description.

Quelques instructions se comportent comme un préfixe et peuvent être suivies par d'autres instructions sur la même ligne, et il peut y avoir plus d'un préfixe sur une ligne. Chaque nom de registre de segment est aussi un mnémonique d'un préfixe d'instruction, cependant il est recommandé d'utilisé un dépassement de segment que l'on définit à l'intérieur de crochets, à la place de ces prefixes.

2.1.1 Déplacement de données

`mov`*(destination,source)* transfert un octet, mot ou double-mot de l'opérande source vers l'opérande destination, de cette façon :

```
mov destination,source
```

Il peut transférer des données entre des registres, d'un registre vers une zone mémoire, ou d'une zone mémoire vers un registre, mais il ne peut pas faire de déplacements de zone mémoire à zone mémoire. Il peut aussi transférer une valeur immédiate vers une zone mémoire ou une registre, d'un registre de segment vers un registre général ou une zone mémoire, registre général ou zone mémoire vers un registre de segment, d'un registre de contrôle ou de debug vers un registre général et réciproquement. L'assemblage de `mov` ne pourra s'effectuer que si la taille de l'opérande source et destination sont identiques. Voici les exemples de toutes les combinaisons possibles :

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

`xchg`*(destination,source)* échange le contenu de deux opérandes. Il peut échanger deux opérandes de type octets, deux mots ou deux double-mot. L'ordre des opérandes n'est pas important. Les opérandes peuvent être des registres généraux ou des zones mémoires. Par exemple :

```
xchg ax,bx          ; échange deux registres généraux
xchg al,[char]      ; échange un registre avec une zone mémoire
```

`push` *destination* décrémente l'indicateur du pointeur de pile (registre `esp`) et transfert l'opérande en haut de la pile désigné par `esp`. L'opérande peut être une zone mémoire, un registre général, un registre de segment ou une valeur immédiate de type octet, mot ou double-mot. Si l'opérande est une valeur immédiate et qu'aucune taille n'est indiquée alors il est considéré, par défaut, comme un mot (word) si l'assembleur est en mode 16 bits et en double-mot sir l'assembleur est en

mode 32 bits. Les mnémoniques `pushw` et `pushd` sont les variantes de l'instruction qui stocke les valeurs d'un mot ou d'un double-mot. Si il y plusieurs opérandes sur la même ligne (séparées par des espaces et pas par des virgules), le compilateur assemblera la suite de ces instructions. Exemple avec des opérandes uniques :

```
push ax             ; Stocke un registre général
push es             ; Stocke une registre de segment
push [bx]           ; Stocke une zone mémoire
push 1000h          ; Stocke une valeur immédiate

push ecx edx eax    ; Stocke plusieurs registres généraux
```

`pusha` sauvegarde le contenu des huit registres généraux sur la pile. Cette instruction n'a aucune opérande. Il y a deux versions de cette instruction, une version 16 bits et un version 32 bits, l'assembleur génère automatiquement la bonne version pour le mode courant, mais il peut être dépassé en utilisant les mnémoniques `pushaw` ou `pushad` pour toujours avoir la version 16 ou 32 bits. La version 16 bits empile les registres généraux sur la pile suivant cet ordre : `ax`, `cx`, `dx`, `bx`, la valeur de `sp` avant que `ax` soit empilé, `bp`, `si` et `di`. Même chose pour la version 32 bits. Si on représente cette instruction comme un programme, nous avons :

```
IF Taille_Operand = 16   ; instruction PUSHA
   Temp = (SP)
   Push(AX)
   Push(CX)
   Push(DX)
   Push(BX)
   Push(Temp)
   Push(BP)
   Push(SI)
   Push(DI)
ELSE                     ; Taille_Operande = 32 => instruction PUSHAD
   Temp = (ESP)
   Push(EAX)
   Push(ECX)
   Push(EDX)
   Push(EBX)
   Push(Temp)
   Push(EBP)
   Push(ESI)
   Push(EDI)
ENDIF
```

`pop` *destination* transfère le mot ou double-mot du haut de la pile vers l'opérande de destination puis incrémente `esp` pour pointer vers la nouvelle valeur du haut de la pile. L'opérande peut être une zone mémoire, un registre général ou un registre de segment. Les mnémoniques `popw` et `popd` sont les variantes de l'instruction qui récupère les valeurs d'un mot ou d'un double-mot. Si plusieurs opérandes sont sur la même ligne et séparées par des espaces, l'assembleur compilera la suite des instructions.

```
pop bx              ; Restaure un registre général
pop ds              ; Restaure le segment de registre
popw [si]           ; Restaure une zone mémoire

pop eax edx ecx     ; Restaure plusieurs registres généraux
                    ; Notez l'ordre des registres empilés par le push
                    ; et l'ordre des registres dépilés par le pop
```

`popa` restaure les registres sauvegardés sur la pile par l'instruction `pusha` à part la valeur de `sp` (ou `esp`), qui est ignorée. Cette instruction n'a pas d'opérandes. Pour forcer l'assemblage de cette instruction en mode 16 ou 32 bits, il faut utiliser les mnémoniques `popaw` ou `popad`.

2.1.2 Conversion de type

Les instructions de conversions de types convertissent les octets en mots, mots en doubles-mot et doubles-mot en quadruples-mot. Ces conversions peuvent se faire en ajoutant une extension de signe ou une extension par zéro. L'extension de signe remplit les bits supplémentaires du plus grand élément avec la valeur du bit de signe du plus petit élément, l'extension de zéro remplit les bits supplémentaires par des zéros.

`cwd` et `cdq` doublent la taille des valeurs des registres `ax` et `eax` et sauvegardent les bits supplémentaires dans les registres `dx` ou `edx`. La conversion s'effectue en utilisant l'extension de signe. Ces instructions n'ont pas d'opérandes.

`cbw` étend le signe de l'octet du registre `al` faisant partie de `ax`, et `cwde` étend le signe du mot `ax` faisant partie de `eax`. Ces instructions n'ont pas d'opérandes.

`movsx` converti un octet en mot ou double-mot et un mot en double-mot avec l'extension de signe. `movzx` fait la même chose, mais il utilise l'extension par zéro. L'opérande source peut être un registre général ou une zone mémoire, alors que l'opérande de destination doit être un registre général. Par exemple :

```
movsx ax,al         ; registre octet vers registre mot
movsx edx,dl        ; registre octet vers registre double-mot
movsx eax,ax        ; registre mot vers registre double-mot
```

```
movsx ax,byte [bx]  ; zone mémoire octet vers registre mot
movsx edx,byte [bx] ; zone mémoire octet vers registre double-mot
movsx eax,word [bx] ; zone mémoire mot vers registre double-mot
```

2.1.3 Les instructions de l'arithmétique binaire

`add`*(destination,source)* remplace l'opérande de destination avec la somme des opérandes source et destination et met l'indicateur `CF` (Carry Flag : le drapeau de retenue) à 1 si un débordement se produit. Les opérandes peuvent être des octets, mots ou doubles-mot. L'opérande destination peut être un registre général ou une zone mémoire, l'opérande source un registre général ou une valeur immédiate, il peut aussi être une zone mémoire si la destination est un registre.

```
add ax,bx           ; Additionne un registre à un registre
add ax,[si]         ; Additionne une zone mémoire à un registre
add [di],al         ; Additionne un registre à une zone mémoire
add al,48           ; Additionne une valeur immédiate à un registre
add [char],48       ; Additionne une valeur immédiate à une zone mémoire
```

`adc`*(destination,source)* fait la somme des opérandes, ajoute 1 si l'indicateur `CF` est positionné, et remplace l'opérande de destination par le résultat. Cette instruction suit les mêmes règles que l'instruction `add`. `add` suivit par plusieurs instructions `adc` peut être utilisé pour additionner des nombres supérieurs à 32 bits.

`inc` *destination* ajoute 1 à l'opérande, il n'affecte pas `CF`. L'opérande peut être un registre général ou une zone mémoire, et la taille de celle-ci peut être un octet, un mot ou un double-mot.

```
inc ax              ; Incrémente le registre par 1
inc byte [bx]       ; Incrémente la zone mémoire par 1
```

`sub`*(destination,source)* soustrait l'opérande source de l'opérande destination et remplace la destination par le résultat. L'indicateur de retenue `CF` est posé si le résultat ne tient pas dans la destination. Cette instruction suit les mêmes règles que l'instruction `add`.

`sbb`*(destination,source)* soustrait l'opérande source de l'opérande destination et soustrait 1 si l'indicateur `CF` est posé, et stocke le résultat dans l'opérande de destination. Cette instruction suit les mêmes règles que l'instruction `add`. `sub` suivit par plusieurs instructions `sbb` peut être utilisé pour soustraire des nombres plus grand que 32 bits.

`dec` *destination* soustrait 1 à l'opérande, il n'affecte pas l'indicateur `CF`. Cette instruction suit les même règles que l'instruction `inc`.

`cmp`*(destination,source)* soustrait l'opérande source de l'opérande destination. Il met à jour les indicateurs comme pour l'instruction `sub`, mais ne modifie pas la source et la destination. Cette instruction suit les même règles que l'instruction `sub`.

`neg` *destination* soustrait un entier signé de zéro. L'effet de cette instruction est de transformer le signe de l'opérande de négatif à positif ou inversement. Cette instruction suit les même règles que l'instruction `inc`.

`xadd`*(destination,source)* échange l'opérande de destination avec l'opérande source, puis il charge la somme de ces deux valeurs dans l'opérande de destination. Cette instruction suit les même règles que l'instruction `add`. Exemple :

```
mov bx,1200          ; bx = 1200
mov dx,2500          ; dx = 2500
xadd bx,dx           ; bx = 3700 - dx = 1200
```

Toutes ces instructions arithmétiques binaires modifient les indicateurs `SF` *(Sign Flag : indicateur de signe - bit 7)*, `ZF` *(Zero Flag : indicateur de zéro - bit 6)*, `PF` *(Parity Flag : indicateur de parité - bit 2)* et `OF` *(Overflow Flag : indicateur de débordement - bit 11)*. L'indicateur `SF` a la même valeur que le résultat du bit de signe, `ZF` est mis à 1 quand le résultat d'une opération donne 0 (zéro), `PF` est mis à 1 quand les 8 bits de poids faibles du résultat d'une opération contient un nombre pair de bits égaux à 1, `OF` est mis à 1 si le résultat est trop grand pour un nombre positif ou trop petit pour un nombre positif (bit de signe exclu).

`mul` *source* exécute une multiplication non-signée de l'opérande source et de l'accumulateur. Si l'opérande est un octet, le processeur le multiplie par le contenu du registre `al` et retourne le résultat au format 16 bits dans `ah` et `al`. Si l'opérande est un mot, le processeur le multiplie le contenu de `ax` et retourne le résultat au format 32 bits dans les registres `dx` et `ax`. Si l'opérande est un double-mot, le processeur le multiplie par le contenu de `eax` et retourne le résultat au format 64 bits dans les registres `edx` et `eax`. `mul` met à 1 les indicateurs `CF` et `OF` quand la moitié haute du résultat est différent de 0, autrement ces indicateurs sont mis à 0. Cette instruction suit les même règles que l'instruction `inc`.

`imul` effectue une multiplication signée. Cette instruction a trois variantes. La première n'a qu'une seule opérande (`imul` *source*) et se comporte de la même façon que l'instruction `mul`. La seconde a deux opérandes (`imul`*(destination,source)*), l'opérande destination est multiplié à l'opérande source et le résultat est mis dans la destination. L'opérande destination doit être un registre général, un mot ou un double-mot, l'opérande source est un registre général, une zone mémoire ou une valeur immédiate. La troisième variante a trois opérandes (`imul`*(destination, source1, source2)*), l'opérande destination doit être un registre général, un mot ou un double-mot, l'opérande source1 est un registre général ou une zone mémoire, et le dernier opérande source2 doit être une valeur immédiate. L'opérande source1 est

multiplié par l'opérande source2 (une valeur immédiate) et le résultat est stocké dans l'opérande destination. Ces trois formes calculent le produit de la taille des opérandes par deux et mettent `CF` et `OF` à 1 quand la moitié haute du résultat est différent de zéro, mais la seconde et troisième variante de cette instruction tronque le produit de la taille des opérandes, ces variantes peuvent donc aussi être utilisées pour des opérandes non-signées car, que les opérandes soient signées ou non, la moitié basse du produit est identique. Voici des exemples de ces trois variantes :

```
imul bl          ; Accumulateur par registre
imul word [si]   ; Accumulateur par zone mémoire
imul bx,cx       ; Registre par registre
imul bx,[si]     ; Registre par zone mémoire
imul bx,10       ; Registre par valeur immédiate
imul ax,bx,10    ; bx * 10 => registre ax
imul ax,[si],10 ; Zone mémoire par valeur immédiate vers registre
```

`div` *source* effectue une division non-signée de l'accumulateur par l'opérande. Le dividende (accumulateur) est deux fois la taille du diviseur (opérande source), le quotient et le reste de la division ont la même taille que le diviseur. Si le diviseur est un octet, le dividende sera alors le registre `ax`, le quotient est stocké dans `al` et le reste dans `ah`. Si le diviser est un mot, la moitié haute du dividende est pris du registre `dx`, la moitié basse du dividende est pris du registre `ax`, le quotient est stocké dans `ax` et le reste dans `dx`. Si le diviseur est une double-mot, la moitié haute du dividende est pris du registre `edx`, la partie basse du dividende est pris du registre `eax`, le quotient est stocké dans `eax` et le reste dans `edx`. Cette instruction suit les même règles que l'instruction `mul`.

`idiv` *source* effectue une division signée de l'accumulateur par l'opérande. Il utilise les mêmes registres que l'instruction `div` et suit les mêmes règles que cette opérande.

2.1.4 Les instructions de l'arithmétique décimale

L'arithmétique décimal s'effectue en combinant l'arithmétique binaire (décrite dans la section précédente) avec les instructions de l'arithmétique décimal. Ces instructions sont utilisées pour ajuster une précédente opération arithmétique binaire pour produire un résultat décimal compacté ou non compacté valide, ceci afin d'ajuster les prochaines entrées d'une opération arithmétique binaire, l'opération produira un résultat décimal compacté ou non compacté valide.

`daa` ajuste en `al` le résultat d'une addition de deux nombres décimaux compactés, `daa` doit toujours suivre l'addition de deux nombres décimaux compactés (un chiffre pour chaque demi-octet) pour obtenir comme résultat deux nombre décimaux compactés valides. L'indicateur de retenue est mis à 0 si le contenu

est correct (entre 0 et 99 car les chiffres sont représentés en BCD *Binary Coded Decimal - Décimal Codé en Binaire*). Cette instruction est sans opérande.

`das` ajuste en `al` le résultat d'une soustraction de deux nombres décimaux compactés, `das` doit toujours suivre la soustraction de deux nombres décimaux compactés (un chiffre pour chaque demi-octet) pour obtenir comme résultat deux nombre décimaux compactés valides. L'indicateur de retenue `CF` est mis à 1 s'il y a dépassement (> 99). Cette instruction est sans opérande.

`aaa` change le contenu du registre `al` en un nombre décimal non compacté valide, et rempli de zéros les quatre bits (ou quartet) de poids fort. `aaa` doit toujours suivre l'addition des deux opérandes décimales non compactées dans `al`. L'indicateur de retenue est posé et `ah` est incrémenté si une retenue est nécessaire. Cette instruction est sans opérande. Exemple :

```
mov ah,8
mov al,4
add al,ah       ; al = al + ah --> al = $0C
xor ah,ah       ; ah = 0
                ; on ajuste au format BCD
aaa             ; ah = 1 et al = 2
```

`aas` change le contenu du registre `al` en un nombre décimal non compacté valide, et rempli de zéros les quatre bits (ou quartet) de poids fort. `aas` doit toujours suivre la soustraction des deux opérandes décimales non compactées dans `al`. Si la soustraction produit une retenue, le registre `ah` est incrémenté, et les indicateurs `CF` et `AF` *(AF : Auxiliary Flag - Indicateur de retenue auxiliaire)* sont mis à 1. Dans le cas contraire `CF` et `AF` sont mis à 0 et `ah` reste inchangé. Cette instruction est sans opérande.

`aam` corrige le résultat d'une multiplication de deux nombres décimaux non compactés valides. `aam` doit toujours suivre une multiplication de deux nombres décimaux pour produire un résultat décimal valide. Le chiffre de poids fort est stocké dans `ah`, le chiffre de poids faible dans `al`. La version généralisée de cette instruction permet à l'ajustement du contenu du registre `ax` de créer deux chiffres non compactées. `aam` décompacte le résultat dans `al` en divisant al par 10, laissant le quotient (chiffre de poids fort) dans `ah` et le reste (chiffre de poids faible) dans `al`. Cette instruction n'a pas d'opérandes. Exemple :

```
mov al,8
mov cl,4
mul cl          ; ax = al * cl --> ax = 32 ($20)
                ; on ajuste le résultat
aam             ; ah = 3 et al = 2
```

`aad` prépare deux valeurs décimales non compactés à être traité par une division et modifie le numérateur dans `al` et `ah`, ainsi le quotient produit par la division sera un nombre décimal non compacté. `ah` contient le chiffre de poids fort et `al` le chiffre de poids faible. Cette instruction ajuste la valeur et met le résultat dans `al` tandis que `ah` sera égal à zéro. Cette instruction suit les même règles que l'instruction `aam`. Exemple :

```
mov al,8
mov ah,4
aad             ; prépare ax à une division
                ; après cette instruction al = 48 et ah = 0
```

2.1.5 Instructions logiques

L'instruction `not` inverse les bits de l'opérande, ce qui permet de faire le complément à un de cette opérande. Il n'a aucun effet sur les indicateurs (flags). Cette instruction suit les même règles que l'instruction `inc`.

Les instructions `and`, `or` et `xor` exécutent les opérations standards logiques. Ils modifient les indicateurs `SF` *(Indicateur de Signe-Bit 7)*, `ZF` *(Indicateur de Zéro-Bit 6)* et `PF` *(Indicateur de Parité-Bit 2)*. Cette instruction suit les même règles que l'instruction `add`.

Les instructions `bt`*(destination,source)*, `bts`, `btr` et `btc` opèrent sur un seul bit d'une zone mémoire ou d'un registre général. Le rang du bit à tester est indiqué par la position du bit de l'opérande source (deuxième opérande), qui peut être une valeur immédiate ou un registre. Ces instructions extrait le bit de l'opérande destination et le met dans l'indicateur `CF`. L'instruction `bt` ne fait rien d'autre, `bts` met le bit sélectionné à 1, `btr` fait l'inverse, il met le bit sélectionné à 0 et `btc` inverse le bit sélectionné. Le premier opérande peut être un mot ou un double-mot. Exemple :

```
bt  ax,15           ; Teste le bit 15 du registre ax
bts word [bx],15    ; Test et met le bit 15 de la mémoire à 1
btr ax,cx           ; Test et met le bit de cx dans ax à 0
btc word [bx],cx    ; Test et inverse le bit de cx dans la mémoire

mov ax,1000000000000000b    ; ax = 32768
bt  ax,1000000000000000b    ; test le bit 15 de ax
```

Les instructions `bsf`*(destination,source)* et `bsr` recherchent le premier bit d'un mot ou d'un double-mot et stocke l'index de ce bit dans l'opérande destination qui est un registre général. Le bit recherché est indiqué par l'opérande source qui est un registre ou une zone mémoire. Si la recherche aboutit l'indicateur `ZF` est mis à

1 et l'opérande source contient l'index du bit. Si aucun bit n'est trouvé, la source et l'indicateur `ZF` sont mis à 0. `bsf` fait une recherche en partant du bit 0 et en remontant vers le bit de poids fort. `bsr` fait une recherche en partant du bit de poids fort (bit 31 pour un double-mot, bit 15 pour un mot) et redescend vers le bit 0.

```
bsf ax,bx           ; Recherche 1er bit à 1 dans ax
                    ; met l'index dans bx en partant du bit 0
bsr ax,[si]         ; Recherche le 1er bit à 1 dans ax
                    ; met l'index dans zone mémoire [si] en
                    ; partant du bit 15 vers bit 0
```

`shl`*(destination,source)* décale vers la gauche l'opérande destination par le nombre de bits indiqué par l'opérande source. L'opérande destination peut être un octet, un mot, un double mot d'un registre général ou une zone mémoire. L'opérande source peut être une valeur immédiate ou le registre `cl`. `shl` rempli de zéro à partir de la droite de l'opérande (poids faible) et les bits disparaissent à gauche. Le dernier bit décalé vers la gauche est stocké dans l'indicateur `CF`. L'instruction `sal` est identique à l'instruction `shl`.

```
shl al,1            ; Décale le registre al d'un bit vers la gauche
                    ; => Correspond à une multiplication par 2
shl byte [bx],1     ; Décale la zone mémoire d'un bit vers la gauche
shl ax,cl           ; Décale vers la gauche le registre ax
                    ; du nombre de bits indiqué dans cl
shl word [bx],cl    ; Décale vers la gauche la zone mémoire
                    ; du nombre de bits indiqué dans cl
```

`shr`*(destination,source)*, et `sar`, décale vers la droite la destination par le nombre de bits indiqué par la source. Cette instruction suit les même règles que l'instruction `shl`. `shr` rempli de zéro à partir de la gauche de l'opérande et les bits disparaissent à droite. Le dernier bit décalé vers la droite est stocké dans l'indicateur `CF`. Par contre l'instruction `sar` préserve le signe de l'opérande en le décalant par des zéros si la valeur est positive et par des uns si la valeur est négative.

```
shr al,1            ; Décale le registre al d'un bit vers la droite
                    ; => Correspond à une division par 2
shr al,2            ; Décale le registre al de deux bits vers la droite
                    ; => Correspond à une division par 4
```

`shld`*(destination,source,nombre)* décale vers la gauche les bits de l'opérande source du nombre de positions donné par le troisième opérande, le résultat est mis dans la destination sans modification de l'opérande source. La destination peut

être de type Mot ou Double-Mot, un registre général ou une zone mémoire, la source doit être de type registre général et le troisième opérande peut être une valeur immédiate ou le registre `cl`.

```
shld ax,bx,1        ; décale le registre bx de 1 bit et le met dans ax
shld [di],bx,1      ; décale bx de 1 bit
                    ; et met le résultat dans la zone mémoire [di]
shld ax,bx,cl       ; décale bx du nombre donné par cl :
                    ; résultat dans ax
shld [di],bx,cl     ; décale bx du nombre donné par cl
                    ; et met le résultat dans [di]
```

`shrd`*(destination,source,nombre)* décale vers la droite les bits de l'opérande source du nombre de positions donné par l'opérande nombre, le résultat est mis dans la destination. L'opérande source n'est pas modifié. Les règles sont les mêmes que l'instruction `shld`.

`rol` et `rcl`*(destination,source)* décale, l'Octet, Mot ou Double-Mot, vers la gauche les bits de la destination par le nombre indiqué par l'opérande source. Pour `rol` : à chaque rotation le bit de poids fort (bit le plus à gauche) qui sort par la gauche est placé à droite dans le bit de poids faible (bit '0'). `rcl` effectue la même chose sauf que le premier bit est placé dans l'indicateur `CF` après que le contenu de cet indicateur ait été placé dans le bit '0' de la destination. Ces instructions suivent les mêmes règles que l'instruction `shl`.

`ror` et `rcr`*(destination,source)* décale, l'Octet, Mot ou Double-Mot, vers la droite les bits de la destination par le nombre indiqué par l'opérande source. Pour `ror` : à chaque rotation le bit de poids faible (bit le plus à droite) qui sort par la droite est placé à gauche dans le bit de poids fort. `rcr` effectue la même chose sauf que le dernier bit est placé dans l'indicateur `CF` aprés que le contenu de cet indicateur ait été placé dans le bit de poids fort de la destination. Ces instructions suivent les mêmes règles que l'instruction `shl`.

`test`*(destination,source)* agit de la même façon que l'instruction `and`, à la seule différence qu'il n'altère pas la l'opérande de destination, il met juste les flags à jour. Cette instruction suit la même règle que l'instruction `and`.

`bswap`*(destination)* permute l'ordre des octets d'un registre général de 32 bits. Il permute les bits 0 à 7 avec les bits 24 à 31 et les bits 8 à 15 avec les bits 16 à 23. Cette instruction est mise à disposition pour la conversion des valeurs de type "little-endian" vers un format de type "big-endian" et réciproquement.

```
bswap edx           ; permute les octets du registre edx
```

2.1.6 Les instructions de contrôle de transfert

`jmp`*(destination)* transfert le contrôle, d'une façon inconditionnelle, à la destination. L'adresse de destination peut être spécifiée directement dans l'instruction ou indirectement par un registre ou une zone mémoire, la taille de cette adresse dépend si le saut est proche ou lointain *(appelé aussi appel inter-segment)*(on peut le définir en ajoutant les opérateurs `'near'` ou `'far'` avant l'opérande) et si l'instruction est en 16 ou 32 bits. L'opérande a utiliser pour les sauts `'near'` doit être `word` pour les instructions 16 bits et `dword` pour les instructions 32 bits. Pour les sauts `'far'`, on utilise `dword` pour les instructions en 16 bits ou `pword` pour les instructions 32 bits. Une instruction directe `jmp` a obligatoirement l'adresse de destination (on peut la précéder de l'opérateur `short`, `near` ou `far`), l'opérande indiquant l'adresse doit être une expression numérique pour les sauts de type `near` ou `short`, ou deux expressions numériques séparées par `':'` pour les sauts `far`, la première expression indique le segment et le deuxième est l'offset à l'intérieur de ce segment. L'opérateur `pword` peut être utilisé pour forcer un saut de type `far` 32 bits, et `dword` pour un saut `far` 16 bits. Une instruction indirecte `jmp` nous donne l'adresse de destination indirectement par un registre ou une variable pointeur, l'opérande utilisé doit être un registre ou une zone mémoire. Voir section 1.2.5 pour plus de détails.

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

`call`*(destination)* appelle une procédure ou sous-programme, l'adresse de l'instruction suivant le `call` est sauvegardée sur la pile, l'adresse de retour sera alors lue lors de l'utilisation de l'instruction `ret` (return). Cette instruction suit la même règle que l'instruction `jmp`, sauf que `call` n'a aucune version courte pour une instruction directe et n'est donc pas optimisé.

`ret`, `retn` et `retf` mettent fin au sous-programme et redonnent le contrôle au programme appelant en utilisant l'adresse sauvegardée sur la pile par l'instruction `call`. `ret` est identique à `retn`, elle revient de la procédure qui a été executée par un `call` de type `'near'`, alors que `retf` revient d'un call de type `'far'`. Ces instructions ont une taille par défaut en fonction des paramètres du code, la taille de l'adresse peut être forcée à 16 bits en utilisant les instructions `retw`, `retnw` et `retfw`, et à 32 bits par l'utilisation des instructions `retd`, `retnd` et `retfd`. En option, on peut ajouter une opérande immédiate à ces instructions, en ajoutant cette constante au pointeur de la pile, tous les arguments mis sur la pile, que le programme appelant a passé, sont effacés avant l'execution de l'instruction `call`.

`iret` redonne le contrôle d'une procédure d'interruption. Elle diffère de `ret` car elle retourne aussi les 'flags' de la pile dans le registre des flags. Ces flags sont sauvegardés sur la pile par le mécanisme de l'interruption. La taille de l'adresse de retour est définie en fonction des paramètres du code, mais peut-être forcée à 16 ou 32 bits par l'utilisation des instructions `iretw` ou `iretd`.

Les instructions de transferts conditionnels sont des sauts qui transfèrent ou non le contrôle en fonction du statut des flags du CPU quand l'instruction s'exécute. Les mnémoniques pour les sauts conditionnels peuvent être obtenus en ajoutant le mnémonique de condition adéquat (voir tableau 2.1) au mnémonique `'j'`, par exemple l'instruction `jc` transfert le contrôle si le flag `CF` est défini. Les sauts conditionnels peuvent être de type `short` ou `near`, uniquement directs, et peuvent être optimisés (voir 1.2.5), l'opérande doit être une valeur immédiate indiquant l'adresse de destination.

Les instructions `loop`*(destination)* sont des sauts conditionnels utilisant une valeur placée dans `cx` (ou `ecx`), celle-ci donne le nombre de répétition à effectuer dans la boucle du programme. Toutes les instructions `loop` décrémentent automatiquement la valeur du registre `cx` (ou `ecx`) et termine la boucle (sans donner le contrôle) quand `cx` (ou `ecx`) est égal à zéro. Ils utilisent `cx` ou `ecx` si les paramétres du code actuel est en 16 ou 32 bits, mais on peut forcer l'utilisation de `cx` avec l'instruction `loopw` *(cx : 16 bits et loopw : word)* ou l'utilisation de `ecx` avec `loopd`. `loope` et `loopz` sont similaires à l'instruction `loop` mais termine aussi la boucle quand le flag `ZF` est défini. `loopew` et `loopzw` forcent l'utilisation du registre `cx` alors que `looped` et `loopzd` force l'utilisation de `ecx`. `loopne` et `loopnz` sont identiques à l'instruction `loop` sauf que la boucle peut aussi se terminer quand le flag `ZF` n'est pas défini. Les mnémoniques `loopnew` et `loopnzw` forcent l'utilisation du registre `cx` alors que `loopned` et `loopnzd` forcent l'utilisation du registre `ecx`. Chaque instruction `loop` requiert une valeur immédiate indiquant l'adresse de destination qui ne peut être qu'un saut court (saut maximum de 128 octets avant et 127 octets après l'adresse suivant l'instruction `loop`).

`jcxz`*(destination)* saute au label indiqué dans la destination si la valeur du registre `cx` est égale à zéro, `jecxz` fait la même chose mais vérifie la valeur dans le registre `ecx`. Ces opérandes suivent les mêmes règles que l'instruction `loop`

`int`*(valeur)* déclenche une interruption donnée pas le numéro indiqué dans l'opérande valeur, celui-ci doit être compris entre 0 et 255. La routine d'interruption se termine par l'instruction `iret` et redonne le contrôle à l'instruction suivant le `int`. Le code `int3` appelle l'interruption 3. `into` appelle l'interruption 4 si le flag `OF` est défini.

`bound`*(destination,source)* vérifie que la valeur destination se trouve dans les limites définies par la source. Si la valeur contenue dans le registre est plus petit que la borne inférieure ou plus grand que la borne supérieure, on déclenche l'inter-

ruption 5. L'opérande destination est le registre à tester, la source est une adresse mémoire des deux valeurs limites signées. La taille des valeurs peut être de type `word` ou `dword`.

```
bound ax,[bx]        ; Vérifie les bornes pour un Mot
bound eax,[esi]      ; Vérifie les bornes pour un Double Mot
```

Mnémonique	Conditions testées	Description
o	OF = 1	débordement
no	OF = 0	aucun débordement
c b nae	CF = 1	retenue inférieur ni supérieur ni égal
nc ae nb	CF = 0	aucune retenue supérieur ou égal pas inférieur
e z	ZF = 1	égal zéro
ne nz	ZF = 0	pas égal différent de zéro
be na	CF or ZF = 1	inférieur ou égal pas supérieur
a nbe	CF or ZF = 0	supérieur différent de zéro
s	SF = 1	signé
ns	SF = 0	non signé
p pe	PF = 1	parité parité paire
np po	PF = 0	aucune parité parité impaire
l nge	SF xor OF = 1	inférieur ni supérieur ni égal
ge nl	SF xor OF = 0	supérieur ou égal pas inférieur
le ng	(SF xor OF) or ZF = 1	inférieur ou égal pas supérieur
g nle	(SF xor OF) or ZF = 0	supérieur ni inférieur ni égal

FIGURE 2.1 – Conditions

2.1.7 Les instructions d'E/S (Entrées/Sorties)

`in`*(destination,source)* transfert un octet, mot ou double mot d'un port d'entrée vers le registre `al`, `ax` ou `eax`. Les ports d'entrée/sortie peuvent être adressées directement, avec une valeur immédiate codée en octet dans l'instruction, soit une valeur immédiate comprise entre 0 et 255 ou indirectement via le registre `dx`. La destination doit être un des registres suivant : `al`, `ax` ou `eax`.

```
in al,20h           ; Récupère un octet du port 20h
in ax,dx            ; Récupère un mot du port adressé par dx
```

`out`*(destination,source)* écrit un octet, mot ou double-mot dans le port de destination, soit `al`, `ax` ou `eax`. On peut indiquer le numéro du port en utilisant les mêmes méthodes que l'instruction `in`.

```
out 20h,ax          ; Ecrit un mot dans le port 20h
out dx,al           ; Ecrit un octet dans le port adressé par dx
```

2.1.8 Opérations sur les chaînes

Les opérations sur chaîne de caractères ne se font que sur un seul élément de la chaîne. Un élément de chaîne peut être un octet, un mot ou un double-mot. Chaque élément de la chaîne est adressé par les registres `si` et `di` (ou `esi` et `edi`). Aprés chaque opération sur la chaîne, `si` et/ou `di` (ou `esi` et/ou `edi`) sont automatiquement mis à jour en pointant sur le prochain élément 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. La valeur de l'incrémentation ou de la décrémentation dépend de la taille de l'élément de la chaîne, soit 1, 2 ou 4 (octet, mot ou double-mot). Chaque instruction d'opération de chaîne a un format court qui n'a aucune opérande et utilise `si` et/ou `di` quand le code est en 16 bits ou `esi` et/ou `edi` quand le code est en 32 bits. `si` et `esi` adressent par défaut les données 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 ajoutant au mnémonique une lettre indiquant la taille de l'élément de la chaîne, `b` pour un élément de type octet, `w` pour un élément de type mot, et `d` pour un élément de type double-mot. Le format complet veut que les opérandes indique la taille de l'opérateur ainsi que les adresses mémoire, cela peut être `si` ou `esi` avec n'importe quel prefix de segment, `di` ou `edi` avec toujours comme prefix de segment `es`. `movs`*(destination,source)* transfère le contenu de la mémoire adressée par exemple par `[ds:si]` (ou `[ds:esi]`), ou tout autre segment, dans la mémoire adressée par `[es:di]` (ou `[es:edi]`) avec toujours le segment `es`. La taille des opérandes peut être de type octet, mot ou double mot. La destination doit être

une mémoire adressée par `di` ou `edi` et la source est une mémoire adressée par `si` ou `esi`.

```
movs byte [di],[si]          ; Transfert d'octet de [si] vers [di]
movs word [es:di],[ss:si]    ; Transfert d'un mot
movsd                        ; Transfert d'un double-mot
```

`cmps`*(destination,source)* soustrait l'élément de chaîne 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 deux éléments sont égaux, `ZF` est défini (= 1), autrement il est effacé (mis à zéro). Il compare donc l'octet, le mot ou le double-mot `[ds:si]` (ou `[ds:esi]`), ou tout autre segment, à la mémoire adressée par `[es:di]` (ou `[es:edi]`) avec toujours le segment `es`.

```
cmpsb                        ; Compare des octets
cmps word [ds:si],[es:di]    ; Compare des mots
cmps dword [fs:esi],[edi]    ; Compare des doubles mots
```

`scas`*(destination)* soustrait l'élément de chaîne destination du registre `al`, `ax` ou `eax` (en fonction de la taille de l'élément de la chaîne de caractères) et met à jour les flags `AF`, `SF`, `ZF`, `PF`, `CF` et `OF`. Si les valeurs sont identiques, `ZF` est défini, autrement il est effacé. La destination doit être une mémoire adressée par `di` ou `edi`.

```
scas byte [es:di]            ; Scanne un octet
scasw                        ; Scanne un mot
scas dword [es:edi]          ; Scanne un double mot
```

`lods`*(source)* met dans `al`, `ax` ou `eax` le contenu de la mémoire indiqué dans la source. La source doit être une mémoire adressée par `si` ou `esi` avec n'importe quel segment.

```
lods byte [ds:si]            ; Charge un octet
lods word [cs:si]            ; Charge un mot
lodsd                        ; Charge un double mot
```

`stos`*(destination)* met la valeur de `al`, `ax` ou `eax` dans la mémoire indiquée dans la destination. Cette instruction suit les même règles que l'instruction `scas`.

`ins`*(destination,source)* transfère un octet, mot ou double mot d'un port d'entrée adressé par le registre `dx` à l'élément chaîne dans la destination. La destination doit être une mémoire adressée par `di` ou `edi`, la source doit être le registre `dx`.

```
insb                         ; Lit un octet
ins word [es:di],dx          ; Lit un mot
ins dword [edi],dx           ; Lit un double mot
```

`outs`*(destination,source)* transfère un élément chaîne de caractère de la source vers le port de sortie adressé par le registre `dx`. La destination doit être le registre `dx` et la source doit être une mémoire adressée par `si` ou `esi` avec n'importe quel segment.

```
outs dx,byte [si]           ; Ecrit un octet
outsw                       ; Écrit un mot
outs dx,dword [gs:esi]      ; Écrit un double mot
```

Les préfixes de répétition `rep`, `repe`/`repz` et `repne`/ `repnz` permettent des opérations de répétitions sur des chaînes. Quand une instruction a le préfixe de répétition, celle-ci est répétée chaque fois en utilisant un élément différent de la chaîne de caractères. Cette répétition se termine quand une des conditions spécifiques au préfixe est satisfaisante. Après chaque opération, ces trois préfixes décrémentent automatiquement le registre `cx` ou `ecx` (en fonction de l'adressage en 16 ou 32 bits) et se répète jusqu'à ce que `cx` ou `ecx` soit égal à zéro. `repe`/`repz` et `repne`/ `repnz` sont exclusivement utilisés avec les instructions `scas` et `cmps` (décrites plus haut). Quand ces préfixes sont utilisés, la répétition de l'instruction suivante dépend aussi du flag zéro (`ZF`), `repe` et `repz` arrête l'execution du code quand `ZF` est égal à zéro, `repne` et `repnz` stoppe l'exécution quand `ZF` est défini (mis à un).

```
rep  movsb        ; Transfert de plusieurs double mots
repe cmpsb        ; Compare des octets tant qu'ils sont égaux
```

2.1.9 Instructions de contrôle du Flag

Les instructions de contrôle du flag fournissent une méthode pour changer directement l'état d'un bit dans le registre flag. Toutes les instructions décrites dans cette section n'ont pas d'opérandes.

Chapitre 3

Programmation Windows

3.1 Entêtes de base

3.1.1 Structures

3.1.2 Imports

3.1.3 Procédures

3.1.4 Exports

3.1.5 Component Object Model : COM

3.1.6 Ressources

3.1.7 Encodage du texte

3.2 Entêtes étendues

3.2.1 Paramètres de procédure

3.2.2 Structurer la source