Le Jeu d'instructions 80x86 Chapitre 6

Jusqu'à présent nous n'avons abordé que superficiellement le sujet des instructions disponibles dans les microprocesseurs 80x86. Ce chapitre rectifie la situation mais notez qu'il sert surtout de référence, il explique ce que chaque instruction fait, mais il n'indique pas comment combiner les instructions pour écrire des programmes complets. Ce sera dans le reste du livre que vous apprendrez à le faire.

6.0 Vue d'ensemble du chapitre

Ce chapitre décrit les instructions 80x86 du mode réel. Comme dans tout langage de programmation, il y a diverses instructions que vous utiliserez tout le temps, d'autres que vous n'utilisez qu'occasionnellement, et certaines autres encore que vous n'utiliserez que rarement ou jamais. Ce chapitre organise sa présentation par classes d'instructions et non par leur ordre d'importance. Puisque les programmeurs débutants n'ont pas à apprendre le jeu d'instructions en entier pour pouvoir écrire des programmes significatifs, vous n'aurez probablement pas à apprendre tout de suite comment chaque instruction fonctionne. La liste qui suit décrit les instructions qu'on verra dans ce chapitre. Le symbole "•" marque les instructions importantes de chaque groupe. En apprenant seulement ces instructions, vous serez déjà en mesure d'écrire les programmes assembleur que vous voudrez. Il y a d'autres instructions qui servent surtout à faciliter la programmation, spécialement pour des processeurs 80386. Mais le fait de ne pas les connaître ne vous empêche pas d'écrire des programmes.

De manière sommaire, les instructions 80x86 peuvent être rangées en huit classes différentes :

1) instructions de transfert de données
   • mov, lea, les, push, pop, pushf, popf
2) instructions de conversion
   • cwb, cwd, xlat
3) instructions arithmétiques
   • add, inc, sub, dec, cmp, neg, mul, imul, div, idiv
4) Opérations logiques, décalages, rotations et instructions de bits
   • and, or, xor, not, shl, shr, rcl, rcr
5) instructions d'entrées/sorties
   • in, out
6) instructions de manipulation de chaînes
   • movs, stos, lods
7) instructions de contrôle de flux
   • jmp, call, ret, *sauts conditionnels*
8) instructions diverses
   • clc, stc, cmc

Les sections suivantes décrivent toutes les instructions de ces groupes.

Si vos programmes sont destinés à des processeurs anciens ou bien s'ils doivent garder la compatibilité avec des programmes plus vieux, il ne faudra pas utiliser les instructions étendues du 80386, car leur usage limite le nombre de machines sur lesquelles votre code s'exécutera. Cependant, à l'heure d'écrire ce livre, le 80386 était déjà en train de disparaître. Par conséquent, vous pouvez tranquillement présumer que la plupart des systèmes auront au moins un processeur 80386sx. Ce livre utilise les instructions du 80386 pour des nombreux exemples de programmes, mais gardez à l'esprit que ceci est seulement par commodité ; il n'y a pas de programme qui ne pourrait être écrit en n'utilisant que des instructions 8088.

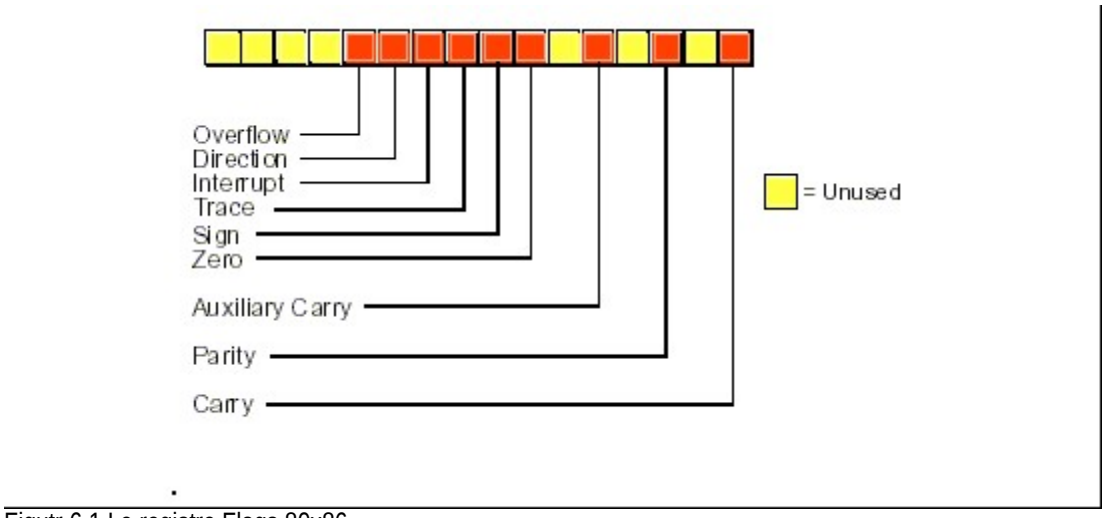
Une recommandation, surtout pour ceux qui apprennent seulement les instructions mentionnées ci-dessus : vous découvrirez que les instructions individuelles ne sont pas vraiment complexes et ont une sémantique simple. Cependant, à mesure que vous approcherez la fin de ce chapitre, vous constaterez que vous n'avez aucune indication sur comment mettre ces instructions ensemble pour former un programme complexe. Ne vous inquiétez pas, c'est un problème courant. Des chapitres ultérieurs combleront cette lacune.

Une dernière chose à noter : ce chapitre énumère diverses instructions avec le commentaire « disponible seulement sur des processeurs 80*n*86 et ultérieurs ». Beaucoup de ces instructions sont également disponibles

sur des 80186, mais puisque peu de PC emploient ces processeurs, on ignorera ce CPU. Néanmoins, prenez ce fait tout de même en compte.

6.1 Le registre d'état du processeur (Flags)

Le registre flags (*drapeaux* en français) conserve le mode de fonctionnement actuel du CPU et des informations sur l'état de certaines instructions. La figure 6.1 montre la disposition de ce registre.



Figutr 6.1 Le registre Flags 80x86

Les drapeaux *carry*, *parity*, *zero*, *sign* et *overflow*[1] sont spéciaux parce que vous pouvez tester leur état (zéro ou un) avec les instructions *setcc* et les instructions de saut (voir les paragraphes 6.6.5 et 6.9.4). Les CPU 80x86 utilisent ces bits (les *codes de condition*) pour prendre des décisions durant l'exécution d'un programme.

Plusieurs instructions arithmétiques, logiques etc., affectent le drapeau *overflow* (dépassement de capacité). Après une opération arithmétique, il peut contenir la valeur un si le résultat dépasse la capacité de la destination signée. Par exemple, si vous essayez d'additionner les nombres signés de 16 bits 7FFFh et 0001h, le résultat est trop grand (pour un nombre signé) et le CPU active le drapeau. Si le résultat d'une opération ne produit pas de dépassement, alors le drapeau est mis à zéro.

Puisque les opérations logiques s'appliquent généralement à des valeurs non signées, ce drapeau est automatiquement mis à zéro pour toute instruction impliquant ces opérations. D'autres instructions y laissent simplement une valeur arbitraire (autrement dit, ne le modifient pas).

Les instructions 80x86 de manipulation de chaînes se servent du *drapeau de direction* (*direction flag*). Quand ce drapeau est à zéro, les éléments d'une chaîne sont interprétés selon un ordre de stockage croissant (d'une adresse inférieure à une adresse supérieure) ; quand il est à 1, alors le CPU traite ces éléments en sens inverse. Pour plus de détails, voir le paragraphe 6.8.

Le drapeau d'activation des interruptions (*interrupt flag*) contrôle la capacité du CPU de répondre à des événements externes, connus comme « requêtes d'interruption ». Certains programmes peuvent contenir des séquences d'instructions que le CPU ne doit pas interrompre. Ce drapeau peut activer ou désactiver les interruptions pour garantir que le CPU n'interrompra pas ces sections de code critiques.

Le drapeau de trace (*trace flag*) active ou désactive le mode trace. Les débogueurs (comme CodeView) utilisent ce bit pour activer ou désactiver l'opération de trace pas à pas. Quand il est activé, le CPU s'interrompt après chaque instruction et passe le contrôle au logiciel de débogage, permettant au débogueur de faire la trace

[1]Afin de varier le style d'écriture et d'habituer le lecteur aux deux terminologies, les noms français et anglais des drapeaux individuels seront utilisés interchangeablement. Il en va de même pour ce qui concerne les termes "drapeaux" et "flags" ; le lecteur doit s'habituer dès le début à connaître la terminologie anglaise, car on la voit partout dans les codes sources et même dans d'autres ouvrages, n.d.t.

pas à pas à travers l'application. S'il est mis à zéro, alors les instructions seront exécutées sans interruption. Les CPU 80x86 ne fournissent aucune instruction permettant de manipuler directement ce flag. Pour le faire, vous devez :

- Placer le registre flags dans la pile 80x86
- Désempiler la valeur dans un autre registre
- Modifier la valeur
- Empiler le résultat, puis
- Désempiler flags

Si le résultat d'un calcul est négatif, alors le CPU active le *drapeau de signe* (*sign flag*). Vous pouvez tester ce drapeau après une opération arithmétique pour vérifier s'il y a un résultat négatif. Souvenez-vous qu'une valeur est négative si son bit le plus significatif est égal à 1. Par conséquent, les opérations sur des valeurs non signées activent ce drapeau si le résultat contient 1 dans le bit de poids fort.

Diverses instructions affectent le drapeau de zéro (*zero flag*) quand elles génèrent un résultat nul. Vous utiliserez souvent ce drapeau pour voir si deux valeurs sont égales (par exemple, deux nombres sont égaux si après leur soustraction le résultat est zéro). Ce drapeau est également utile après diverses opérations logiques pour voir si un bit spécifique dans un registre ou dans la mémoire contient zéro ou un.

Le drapeau de retenue auxiliaire (*auxiliary carry flag*) supporte les opérations spéciales du mode binaire codé décimal (BCD). Puisque la plupart des programmes n'ont rien à faire avec ces nombres, vous n'utiliserez que rarement ce drapeau, et même, vous ne pourrez pas non plus y accéder directement. Aucun des CPU 80x86 ne fournit d'instructions spécifiques pour manipuler ce bit, le tester, l'activer ou le désactiver. Seules les instructions BCD et les instructions add, adc, sub, sbb, mul, imul, div et idiv le manipulent.

Le drapeau de parité (*parity flag*) est activé par la parité des huit octets faibles de toute opération de données. Si une opération produit un nombre pair de bits ayant la valeur 1, le CPU active ce bit. Et il le désactive dans le cas contraire. Ce drapeau est utile dans certains programmes de communication de données, cependant Intel l'a inséré principalement pour des raisons de compatibilité avec les vieux processeurs 8080 μP.

Le drapeau de retenue (*carry flag*) a diverses fonctions. D'abord il indique un dépassement de capacité non signé (de la même manière que le drapeau de dépassement de capacité indique un dépassement signé). Vous l'utiliserez aussi lors d'opérations logiques et arithmétiques à précision multiple. Certaines instructions qui testent, positionnent, effacent ou inversent des bits, affectent directement ce drapeau. Finalement, puisque vous pouvez le manipuler facilement, il est utile pour diverses opérations booléennes. Ce drapeau a diverses fonctions et savoir quand l'utiliser, et pour quelle raison, peut embrouiller les programmeurs novices. Heureusement, pour toute instruction donnée, la signification du drapeau de retenue est claire.

Dans les sections et les chapitres qui suivent, l'usage de ces flags deviendra plus évident. Cette section est principalement une introduction formelle aux drapeaux individuels du registre flags au lieu d'une tentative d'expliquer l'exacte fonction de chacun d'eux. Pour plus de détails sur ceci, continuez à lire...

6.2 Encodage des instructions

Les processeurs 80x86 utilisent un encodage binaire pour chacune des opérations machine. Alors que c'est important d'avoir une compréhension générale sur comment un microprocesseur encode les instructions, vous n'avez pas à mémoriser l'encodage de chacune des instructions du jeu. Si vous deviez écrire un assembleur ou un désassembleur (débogueur), vous devriez définitivement connaître exactement les encodages. Pour la programmation en assembleur de tous les jours, vous n'avez pas besoin de connaître l'encodage exact.

Néanmoins, à mesure que vous devenez plus expert, vous voudrez probablement étudier plus à fond les encodages du jeu d'instructions du CPU. Certainement, vous devriez déjà être parfaitement au courant de termes comme *opcode*, *octet mod-reg-r/m*, *valeur de déplacement* et ainsi de suite. Bien que vous n'ayez pas besoin de mémoriser les paramètres de chaque instruction, c'est toujours une bonne idée de connaître les longueurs et les cycles d'horloge pour les instructions que vous utilisez le plus souvent, puisque ceci vous aidera à écrire de meilleurs programmes. Les chapitres trois et quatre ont offert une vue d'ensemble détaillée des encodages de diverses instructions (80x86 et x86) ; une telle discussion a été importante afin que vous compreniez comment le CPU les encode et les exécute. Ce chapitre ne se penchera pas sur ces détails, mais présentera une vue de haut niveau de chaque instruction en présumant que vous ne vous préoccupez pas de la

façon dont la machine traite les bits en mémoire. Pour ces quelques occasions où vous aurez besoin de connaître l'encodage binaire d'une instruction particulière, une liste complète des encodages apparaît à l'annexe D.

6.3 instructions de transfert de données

Les instructions de transfert de données copient des valeurs d'un emplacement à un autre. Ces instructions comprennent mov, xchg, lds, lea, les, lfs, lgs, lss, push, pusha, pushad, pushf, pushfd, pop, popa, popad, popf, popfd, lahf et sahf.

6.3.1 L'instruction MOV

L'instruction mov existe en différentes versions :

```
mov     reg, reg[2]
mov     mem, reg
mov     reg, mem
mov     mem, donnee_immediate
mov     reg, donnee_immediate
mov     ax/al, mem
mov     mem, ax/al
mov     regseg, mem16
mov     regseg, reg16
mov     mem16, regseg
mov     reg16, regseg
```

Le dernier chapitre a parlé de cette instruction en détail et seulement quelques aspects additionnels restent à mentionner ici. D'abord, il y a des variantes qui sont plus rapides et courtes que d'autres qui font le même travail. Par exemple, *mov ax, mem* et *mov reg, mem* chargent le registre AX avec le contenu d'un emplacement de mémoire. Dans tous les processeurs, la première instruction est plus courte et, dans les premiers membres de la famille 80x86, elle était également plus rapide.

Il y a deux détails très importants à noter à propos de l'instruction mov. Tout d'abord il n'y a pas d'instruction de transfert entre deux emplacements de mémoire. L'octet du mode d'adressage de l'encodage mod-reg-r/m permet seulement comme opérandes deux registres ou bien un registre et un emplacement de mémoire. Il n'existe pas de version de mov permettant d'encoder *deux* adresses de mémoire dans la même instruction. Deuxièmement, on ne peut pas transférer une donnée immédiate dans un registre de segment. Les seules instructions qui transfèrent des données dans ou depuis un registre de segment ont un octet mod-reg-r/m associé avec elles et il n'y a pas de format permettant de déplacer une constante dans ces registres. Effectuer un mov entre deux addresses de mémoire ou placer une constante dans un registre de segment, ce sond deux erreurs communes des programmeurs débutants.

Les opérandes d'une instruction mov peuvent être des octets, des mots ou des doubles-mots[3]. Les deux opérandes doivent avoir la même taille ou MASM émettra un message d'erreur. Ceci s'applique aussi bien aux opérandes de mémoire, qu'aux registres. Si vous déclarez une variable B en utilisant *byte* et que vous essayez de charger cette variable dans le registre ax, MASM se plaindra d'un conflit de types.

Le CPU élargit les données immédiates selon la taille de l'opérande de destination (sauf dans les cas où elles sont trop grandes pour leur destination ; dans ces cas, il s'agit d'une erreur). Notez que vous pouvez placer des valeurs immédiates dans un emplacement de mémoire et que la même règle concernant les tailles s'applique. Néanmoins, MASM ne peut déterminer la taille de certaines opérandes de mémoire. Par exemple, l'instruction mov [bx], 0 affecte-t-elle une valeur de huit, seize ou trente-deux bits ? MASM ne peut pas déterminer la taille exacte d'une constante, donc, dans un pareil cas, il émettra une erreur. Ce problème *n'existe pas* quand vous placez une valeur immédiate dans une variable que vous avez déjà déclarée dans votre programme. Par exemple, si vous avez déclaré B comme une variable de type *byte*, MASM sera en mesure de placer un zéro de huit bits dans B, pour l'instruction mov B, 0. Seules les opérandes concernant les pointeurs qui ne sont pas

[2]Ce chapitre utilise "reg" indifféremment pour indiquer un registre général de huit, seize ou (sur 80386 et ultérieur) 32 bits (AL/AX/EAX, BL/BX/EBX, SI/ESI, etc.).
[3]Sans oublier que les opérandes de 32 bits sont valides seulement sur des processeurs 80386 ou supérieurs.

référencés par une variable souffrent de ce problème. La solution est d'indiquer explicitement à l'assembleur quand une opérande est un octet, un mot ou un double-mot. Vous pouvez le faire comme suit :

```
mov     byte ptr [bx], 0
mov     word ptr [bx], 0
mov     dword ptr[bx], 0 ;sur des CPU 80386 ou supérieurs
```

Pour plus de détails à propos de l'opérateur *type ptr*, voir le Chapitre Huit.

Les instructions mov avec des registres de segment sont toujours de 16 bits ; l'opérande mod-reg-r/m doit être de cette taille ou MASM générera une erreur. Puisque vous ne pouvez pas placer une constante dans un de ces registres, une solution commune consiste en charger celle-ci dans un registre général et ensuite le copier dans un registre de segment. Par exemple, les deux instructions suivantes chargent le registre *es* avec la valeur 40h :

```
mov     ax, 40h
mov     es, ax
```

Remarquez que tout registre général pourrait faire l'affaire et qu'ici ax a été choisi arbitrairement.

Les instructions mov n'affectent aucunement le registre flags.

6.3.2 L'instruction XCHG

L'instruction xchg (*exchange*) permute deux valeurs. La forme générale est :

```
xchg    operande1, operande2
```

Et il y a quatre versions spécifiques de cette instruction sur le 80x86 :

```
xchg    reg, mem
xchg    reg, reg
xchg    ax, reg16
xchg    eax, reg32  ;seulement sur des CPU 80386 ou supérieurs
```

Les deux premières versions requièrent deux octets ou plus pour l'opcode et les octets mod-reg-r/m (un déplacement, si nécessaire, requiert encore plus d'octets). Les deux dernières sont des versions spéciales de la seconde et permutent des données entre le registre (e)ax et un autre registre de 16 ou 32 bits. Avec la version 16 bits, un seul octet d'opcode est utilisé, alors que les deux autres formes requièrent un octet mod-reg-r/m additionnel.

Vous pouvez déjà remarquer un schéma qui se développe : la famille 80x86 fournit souvent des versions plus courtes et plus rapides d'une instruction, par l'entremise du registre ax. Par conséquent, vous devriez arranger vos instructions de telle sorte que le registre (e)ax, y soit impliqué le plus possible. L'instruction xchg en est un exemple parfait, fournissant une version qui permute deux valeurs de 16 bits en employant seulement un octet.

Notez que l'ordre d'apparition des opérandes ne fait pas de différence, en ce sens que *xchg mem, reg* ou *xchg reg, mem* produisent le même résultat. Et si vous codez *xchg reg, ax*, la plupart des assembleurs modernes émettront automatiquement l'opcode de l'instruction plus courte *xchg ax, reg*.

Les deux opérandes doivent avoir la même taille. Les opérandes peuvent être de huit ou seize bits sur des processeurs antérieurs au 80386 et de huit, seize ou trente-deux bits sur des microprocesseurs supérieurs.

L'instruction xchg, ne modifie pas le registre flags.

6.3.3 Les instructions LDS, LES, LFS, LGS et LSS

Ces instructions vous permettent de charger un registre général et un registre de segment en une seule instruction. Sur des processeurs anciens, les instructions *lds* et *les* sont les seules permettant de manipuler directement des valeurs plus grandes que 32 bits. La forme générale est :

```
LxS     dest, source
```

Et ces instructions ont aussi, naturellement, leurs variations spécifiques :

```
lds     reg16, mem32
```

```
les     reg16, mem32
lfs     reg16, mem32 ;seulement sur des CPU 80386 ou supérieurs
lgs     reg16, mem32 ;idem
lss     reg16, mem32 ;idem
```

Reg_{16} est n'importe quel registre général de 16 bits et mem_{32} est un emplacement de mémoire d'un double-mot (déclaré avec l'instruction dword).

Ces instructions chargent le double-mot de l'adresse spécifiée par mem_{32} dans reg_{16} et l'un des registres ds, es, fs, gs ou ss. Le registre général recevra le mot le moins significatif de l'opérande de mémoire, tandis que le registre de segment se verra attribuer le mot le plus significatif de celle-ci. L'algorithme suivant en décrit le fonctionnement exact :

```
lds reg16, mem32 :
                reg16 := [mem32]
                ds := [mem32 + 2]
les reg16, mem32 :
                reg16 := [mem32]
                es := [mem32 + 2]
lfs reg16, mem32 :
                reg16 := [mem32]
                fs := [mem32 + 2]
lgs reg16, mem32 :
                reg16 := [mem32]
                gs := [mem32 + 2]
lss reg16, mem32 :
                reg16 := [mem32]
                ss := [mem32 + 2]
```

Puisque les instructions LxS chargent les registres de segment, vous ne devriez pas les utiliser pour n'importe quel but. Utilisez-les pour définir des pointeurs (éloignés) sur certains objets de données, comme il a été vu au chapitre 5[4]. Tout usage impropre peut causer des problèmes avec votre code, surtout si vous utilisez Windows, OS/2 ou UNIX.

Gardez à l'esprit que ces instructions chargent une valeur de 32 bits dans une paire de registres et *ne chargent pas* l'adresse d'une variable dans cette paire (c'est-à-dire que ces instructions n'ont pas de mode immédiat). Pour savoir comment charger une adresse dans une paire de registres, voir le Chapitre Huit.

Les instructions LxS n'affectent pas non plus le registre flags.

6.3.4 L'instruction LEA

L'instruction LEA (*Load Effective Address*) est une autre instruction utilisée pour préparer des valeurs de pointeur. Elle prend la forme :

```
lea     dest, source
```

Et les formes spécifiques de cette instruction sur le 80x86 sont :

```
lea     reg16, mem
lea     reg32, mem  ;seulement sur des CPU 80386 ou supérieurs
```

L'instruction charge le registre général de 16 ou de 32 bits avec l'*adresse réelle (effective)* de l'emplacement de mémoire spécifiée ; cette adresse est l'adresse finale obtenue après tous les calculs des modes d'adressage. Par exemple, lea ax, ds:[1234h] charge le registre ax avec l'adresse de l'emplacement 1234h ; ici, le registre ax est chargé simplement avec la valeur 1234h. Si vous y réfléchissez un moment, ce n'est pas une opération très excitante. Après tout, *mov ax, valeur_immediate* pourrait le faire également. Donc, pourquoi s'embêter avec l'instruction lea ? Parce qu'il y a aussi d'autres formes d'opérandes de mémoire en plus du déplacement seul. Considérez les instructions lea suivantes :

```
lea     ax, [bx]
```

[4]L'original cite le chapitre 4, mais, puisque ces sujets font en réalité l'objet du chapitre 5, il s'agit d'une autre des diverses erreurs apparaissant dans la version originale et jamais corrigées, n.d.t.

```
lea     bx, 3[bx]
lea     ax, 3[bx]
lea     bx, 4[bp+si]
lea     ax, -123[di]
```

La première de ces instructions copie l'adresse de l'expression [bx] dans le registre ax. Puisque l'adresse réelle est la valeur du registre bx, l'instruction copie la valeur de bx dans le registre ax. Encore une fois, l'instruction n'est pas très intéressante, parce que mov peut faire la même chose, même plus rapidement.

L'instruction lea bx, 3[bx] copie l'adresse effective de 3[bx] dans le registre bx. Puisque cette adresse est égale à la valeur courante de bx plus trois, alors lea additionne simplement trois au registre bx. L'instruction add permet de le faire, donc, lea est encore superflue.

La troisième instruction ci-dessus montre où lea commence à briller. lea ax, 3[bx] copie l'adresse de l'emplacement 3[bx] dans le registre ax ; c'est-à-dire, elle additionne trois à la valeur du registre bx et déplace la somme dans ax. Il s'agit d'un excellent exemple montrant comment vous pouvez effectuer une opération mov et une addition avec une seule instruction.

Les deux instructions finales, lea bx, 4[bp+si] et lea ax, -123[di] donnent des exemples supplémentaires d'instructions lea plus efficaces que leurs équivalents mov/add.

Sur des processeurs 80386 et supérieurs, vous pouvez utiliser le mode d'adressage scalaire indexé pour multiplier par deux, quatre ou huit et additionner des registres et des déplacements. Intel, suggère fortement l'usage d'instructions lea, parce que ces instructions sont beaucoup plus rapides qu'une séquence d'instructions produisant les mêmes résultats.

La fonction (réelle) de lea est de charger un registre avec une adresse de mémoire. Par exemple, lea bx, 128[bp+di] charge bx avec l'adresse de l'octet référencé par 128[BP+DI]. Il s'avère qu'une instruction de la forme mov al, [bx] s'exécute plus rapidement qu'une de la forme mov al, 128[bp+di]. Si cette instruction s'exécute plusieurs fois, c'est probablement plus efficace de charger l'adresse réelle de 128[bp+di] dans le registre bx et d'utiliser le mode d'adressage [bx]. Il s'agit d'une optimisation commune pour des programmes à hautes performances.

L'instruction lea n'affecte pas les drapeaux.

6.3.5 Les instructions PUSH et POP

Les instructions push (empiler) et pop (désempiler) manipulent les données dans la pile matérielle du 80x86. Il y a 19 variantes de ces instructions[5] :

```
push    reg16
pop     reg16
push    reg32                   ;(3)
pop     reg32                   ;(3)
push    regseg
pop     regseg                  ;sauf cs
push    mem
pop     mem
push    donnee_immediate        ;(2)
pusha                           ;(2)
popa                            ;(2)
pushad                          ;(3)
popad                           ;(3)
pushf
popf
pushfd                          ;(3)
popfd                           ;(3)
enter   imm, imm                ;(2)
leave                           ;(2)
```

(2) - Disponible seulement sur des CPU 80286 et supérieurs

[5]Et il faut ajouter aussi divers synonymes.

(3) - Disponible seulement sur des CPU 80386 et supérieurs

Les premières deux instructions empilent et désempilent un registre général de 16 bits. C'est une version compacte (d'un octet) conçue spécialement pour les registres. Notez qu'il y a une seconde version qui fournit un octet mod-reg-r/m permettant d'empiler des registres ; beaucoup d'assembleurs utilisent seulement celle-ci pour empiler une valeur stockée en mémoire.

La troisième et quatrième instruction empilent et désempilent un registre général de 32 bits et ce n'est rien de plus qu'une version de trente-deux bits des deux premières instructions, avec un octet de préfixe indiquant la taille.

La cinquième et sixième instruction permettent d'empiler et désempiler un registre de segment. Notez que les instructions qui empilent fs et gs sont plus longues que les instructions qui empilent cs, ds, es et ss ; voir l'annexe D pour des détails plus précis. Pour ce qui concerne le registre cs, vous pouvez uniquement l'empiler ; le désempiler donnerait lieu à des problèmes intéressants de contrôle de flux de programme.

La quatrième paire d'instructions push/pop vous permet d'empiler et de désempiler le contenu d'un emplacement de mémoire. Dans les processeurs 80286 et antérieurs, cette valeur devait être de 16 bits. Pour les opérations de mémoire sans type explicite (par exemple, [bx]), il faut utiliser soit la version pushw, soit indiquer explicitement la taille en se servant d'une instruction comme push word ptr [bx]. Dans les processeurs 80386 et supérieurs vous pouvez empiler et désempiler des données de 16 et de 32 bits[6]. Vous pouvez utiliser des opérandes de mémoire de type dword, utiliser la syntaxe pushd, ou vous servir de l'opérateur dword ptr pour imposer des opérations de 32 bits.

Exemples :

```
push    DblWordVar
push    dword ptr [bx]
pushd   dword
```

Les instructions pusha (*push all*) et popa (*pop all*) (disponibles sur un CPU 80286 ou supérieur) empilent et désempilent *tous* les registres généraux de 16 bits. pusha empile les registres dans l'ordre suivant : ax, cx, dx, bx, sp, bp, si et di. popa désempile les registres dans l'ordre inverse dans lequel ils ont été empilés. pushad et popad (disponibles sur un 80386 ou supérieur) font la même chose sur un jeu de registres de 32 bits du 80386 (ou ultérieur). Notez que les instructions pusha et popa ne *placent (ou déplacent) pas* dans la pile les registres de segment ou le registre flags.

Ce sont les instructions pushf et popf qui permettent d'empiler/désempiler le registre flags. Notez que ces instructions offrent un mécanisme pour modifier le drapeau de trace (voir la description de ce processus au début de ce chapitre). Bien sûr, vous pouvez aussi activer et désactiver tous les autres drapeaux de cette façon. Néanmoins, la plupart des autres drapeaux qu'on peut vouloir modifier (spécialement les codes de condition), fournissent des instructions spécifiques ou d'autres séquences d'instructions plus simples.

Les instructions enter et leave empilent et désempilent le registre bp et allouent de l'espace dans la pile pour des variables locales. Vous en saurez plus dans un chapitre ultérieur. Ici, on ne les abordera pas, puisqu'elles ne sont pas très utiles hors du contexte d'entrée ou de sortie d'une procédure.

« Et alors qu'est-ce que ces instructions font ? » vous demanderez-vous. Les instructions push placent des données dans la pile matérielle 80x86 et les instructions pop récupèrent les données de la pile et les placent dans la mémoire ou dans des registres. Ce qui suit, est une description algorithmique de chacune :

instructions push (16 bits) :

```
SP := SP - 2
[SS:SP] := Operande16Bits    ; stocke le résultat à l'emplacement SS:SP
```

instructions pop (16 bits) :

```
Operande16Bits := [SS:SP]
SP := SP + 2
```

instructions push de 32 bits :

```
SP := SP - 4
[SS:SP] := Operande32Bits
```

[6]Vous pouvez utiliser les syntaxes pushw et pushd pour indiquer les tailles des constantes de 16 ou de 32 bits.

instructions pop de 32 bits :

```
Operande32Bits := [SS:SP]
SP := SP + 4
```

Vous pouvez considérer aussi les instructions pusha/pushad et popa/popad de la même façon, c'est-à-dire correspondant à une séquence d'opération push/pop sur les registres généraux 16 ou 32 bits.

Trois choses à noter à propos de la pile matérielle : en premier lieu, elle se trouve toujours dans le segment de pile (peu importe où ss pointe). En second lieu, la pile croît vers le bas en mémoire, autrement dit, à mesure que l'on empile des valeurs, le CPU les place dans la pile à des adresses successives qui décroissent après chaque empilage. Et, finalement, le pointeur matériel de pile (ss:sp) contient toujours l'adresse du sommet de la pile (qui correspond à la dernière valeur empilée).

Vous pouvez utiliser la pile matérielle pour sauvegarder temporairement des registres et des variables, pour passer des paramètres à une procédure, pour allouer de l'espace pour des variables locales et pour d'autres usages. Les instructions push et pop ont une grande utilité, car elles permettent de manipuler ces objets dans la pile. Vous aurez l'occasion de voir comment les utiliser, plus tard dans ce livre.

La plupart des instructions push et pop n'affectent aucunement les drapeaux du registre flags. Les instructions popf/popfd, par leur nature intrinsèque, permettent de modifier tous les drapeaux. pushf/pushfd poussent flags dans la pile, mais ne changent aucun drapeau en le faisant.

Tous les empilages et les désempilages sont des opérations de 16 ou de 32 bits. Il n'y a pas de moyen (facile) de placer dans la pile une valeur de huit bits. Pour le faire, il faudrait la charger dans l'octet le plus significatif d'un registre de 16 bits, empiler le registre et puis additionner 1 au pointeur de pile. Dans tous les processeurs, sauf le 8088, ceci ralentirait les accès futurs à la pile, puisque la valeur actuelle de sp serait une adresse impaire, désalignant tous les empilages et les désempilages ultérieurs. Par conséquent, la plupart des programmes empilent et désempilent des valeurs de 16 bits, même s'ils sont en train de travailler avec des valeurs de huit bits.

Malgré que ce soit relativement sécuritaire d'empiler une variable de huit bits, il faut faire attention quand on désempile dans un emplacement de mémoire de cette taille. Empiler une variable de huit bits avec *push word ptr ByteBar* empile deux octets, à savoir l'octet de la variable ByteVar et l'octet qui la suit. Votre code pourra simplement ignorer l'octet en trop que l'instruction empile dans la pile. Mais désempiler de telles valeurs n'est pas aussi simple. Généralement, empiler ces deux octets ne devrait pas poser de problème. Cependant, il peut arriver un désastre si vous dépilez une valeur et que vous écrasez l'octet successif dans la mémoire. Il existe seulement deux solutions à ce problème. D'abord vous pouvez désempiler la valeur de 16 bits et la stocker dans un registre comme ax et ensuite sauver l'octet le moins significatif dans la variable byte. La seconde solution consiste à réserver un octet additionnel de remplissage après la variable byte pour contenir le mot entier que vous désempilez. Beaucoup de programmes utilisent cette approche.

6.3.6 Les instructions LAHF et SAHF

lahf (load ah from flags) et sahf (store ah into flags) sont des instructions archaïques, que le jeu 80x86 a inclus pour des questions de compatibilité avec les vieux programmes de la puce Intel 8080 µP. De ce fait, ces instructions sont très peu utilisées dans les programmes 80x86 d'aujourd'hui. L'instruction lahf n'affecte aucun des drapeaux du registre flags. L'instruction sahf, par sa nature même, modifie les bits S, Z, A, P et C de flags. Ces instructions ne requièrent aucune opérande et on les utilise de la manière suivante :

```
sahf
lahf
```

sahf affecte uniquement l'octet le moins significatif du registre flags, alors que lahf, charge uniquement l'octet le moins significatif de flags dans le registre AH. Ces instructions n'agissent pas sur les drapeaux *overflow*, *direction*, *interrupt disable* ou *trace*. Le seul fait que ces instructions ne peuvent pas agir sur le drapeau *overflow* est une limitation considérable.

Cependant, sahf a une une utilisation majeure : quand vous utilisez un coprocesseur à virgule flottante (8087, 80287, 80387, 80486, Pentium, etc.) vous pouvez vous en servir pour copier le registre flags d'état de virgule flottante dans le registre flags 80x86. Vous verrez cette utilisation dans le chapitre sur l'arithmétique à virgule flottante (chapitre 14).

6.4 Instructions de conversion

Le jeu d'instructions 80x86 fournit diverses instructions de conversion. Celles-ci incluent movzx, movsx, cbw, cwd, cwde, cdq, bswap et xlat. La plupart de ces instructions étendent des valeurs à zéro ou avec le signe, les deux dernières convertissent entre formats de stockage et traduisent les valeurs à l'aide d'une table de référence. Ces instructions prennent la forme générale :

```
movzx   dest, src ;la taille de dest doit être le double de src
movsx   dest, src ;la taille de dest doit être le double de src
cbw
cwd
cwde
cdq
bswap   reg32
xlat            ;une forme spéciale permet une opérande
```

6.4.1 Les instructions MOVZX, MOVSX, CBW, CWD, CWDE et CDQ

Ces instructions effectuent des extensions signées et non signées[7]. cwb et cwd sont disponibles sur tous les processeurs de la famille 80x86, alors que movzx, movsx, cwde et cdq sont disponibles seulement sur les CPU 80386 ou supérieurs.

L'instruction cbw (convert byte to word) effectue une extension signée de la valeur de al à ax, autrement dit, elle copie le bit sept de al dans tous les bits de 8 à 15 de ax. Cette instruction est particulièrement importante avant d'effectuer une division entre valeurs de huit bits (comme vous verrez au paragraphe 6.5). L'instruction ne requiert pas d'opérande et on l'utilise comme suit :

```
cbw
```

L'instruction cwd (convert word to double word) effectue une extension signée de la valeur de ax à une valeur de 32 bits qui sera stockée dans dx:ax. Cette extension se réalise en copiant le bit 15 de ax dans tous les bits de DX. Le fait que cette valeur ne soit pas placée dans eax en démontre la compatibilité dans tous les processeurs. Tout comme cbw, cette instruction est très importante pour les opérations de division. L'instruction ne requiert pas d'opérande et on l'utilise comme suit :

```
cwd
```

L'instruction cwde étend une valeur de 16 bits à une valeur de 32 bits, mais cette fois en la plaçant dans le registre eax, moyennant la copie du bit 15 de ax dans tous les bits de 16 à 31 de eax. Comme cbw et cwd, cette instruction n'a pas d'opérandes :

```
cwde
```

L'instruction cdq effectue une extension signée de la valeur de 32 bits de eax à une valeur de 64 bits et place le résultat dans edx:eax en copiant le bit 31 de eax dans tous les bits de edx. Cette instruction n'est disponible évidemment que sur les processeurs de 32 bits. On l'utilise normalement avant une opération de division de valeurs de type *long*. Comme cbw, cwd et cwde, elle n'a pas d'opérandes et on l'utilise comme suit :

```
cdq
```

Si vous voulez étendre une valeur de huit bits à une valeur de 32 ou de 64 bits, on pourra utiliser une séquence comme :

```
;Étendre al à dx:ax avec le signe
                cbw
                cwd

;Étendre al à eax avec le signe
                cbw
                cwde

;Étendre al à edx:eax
```

[7] Voir le chapitre 1 pour un rappel sur cette sorte d'extensions, n.d.t.

```
cbw
cwde
cdq
```

Vous pouvez utiliser également movsx pour effectuer des extensions signées de huit à seize ou trente-deux bits ; movsx est une forme généralisée de cbw, cwd et cwde. Comme elle peut étendre une valeur de huit bits à une de seize ou trente-deux bits, elle peut le faire aussi d'une valeur de seize bits à une de trente-deux bits. Cette instruction utilise un octet mod-reg-r/m pour spécifier les deux opérandes. Les formes disponibles sont :

```
movsx   reg16, mem8
movsx   reg16, reg8
movsx   reg32, mem8
movsx   reg32, reg8
movsx   reg32, mem16
movsx   reg32, reg16
```

Notez que tout ce que vous pouvez faire avec cbw et cwde, vous le pouvez aussi avec movsx :

```
movsx   ax, al          ;CBW
movsx   eax, ax         ;CWDE
movsx   eax, al         ;CBW suivi par CWDE
```

Cependant cbw et cwde sont plus courtes et parfois plus rapides. Cette instruction n'est disponible que sur le 80386 ou ultérieur. Notez qu'il n'y a pas d'équivalent direct de movsx pour cwd et cdq.

L'instruction movzx fonctionne exactement comme movsx, sauf que ses extensions sont non signées (les bits nouveaux sont remplis par des zéros). La syntaxe aussi est la même que movsx, mais il faut remplacer naturellement movsx par movzx.

Notez que si vous voulez effectuer une extension non signée d'une valeur de huit bits à seize (par exemple, de al à ax), un simple mov est plus rapide et plus court que movzx. Par exemple :

```
mov     bh, 0
```

est plus rapide et court que :

```
movzx   bx, bl
```

Sans doute, si vous convertissez les données entre différents registres (par exemple, movzx bx, al), alors movzx est mieux.

movsx et movzx, et toute autre instruction différente de cbw et cwd, sont disponibles seulement sur des processeurs de 32 bits. Aucune de ces instructions n'affecte les flags.

6.4.2 L'instruction BSWAP

Cette instruction, disponible seulement sur des processeurs 80486 (oui, 486) et supérieurs convertit les valeurs de 32 bits entre *little endian* et *big endian*. C'est-à-dire qu'elle accepte uniquement un registre de 32 bits comme opérande et qu'elle permute le premier octet avec le quatrième et le second avec le troisième. La syntaxe de l'instruction est :

```
bswap   reg32
```

où reg_{32} est un registre général de 32 bits.

La famille des processeurs Intel se sert d'une organisation de la mémoire appelée *little endian byte organization.* Selon cette organisation, l'octet le moins significatif d'une séquence multioctets apparaît à l'adresse mémoire la plus basse de la séquence. Par exemple, les bits 0 à 7 d'une valeur de 32 bits apparaissent à l'adresse la plus basse ; les octets 8 à 15 apparaissent à l'adresse du second octet ; les bits 16 à 23 forment le troisième octet et les bits 24 à 31 le dernier.

Une autre organisation populaire est appelée *big endian*. Dans ce schéma les bits 24 à 31 apparaissent dans le premier octet, les bits 16 à 23 figurent au second octet, les bits 8 à 15 au troisième et les bits 0 à 7 à l'octet le

plus haut. Les CPU comme la famille Motorola 68000 utilisés par Apple dans leur Macintosh, et aussi diverses puces RISC, emploient ce schéma[8].

Normalement, vous n'avez pas à vous inquiéter de l'organisation des octets dans la mémoire, car les programmes écrits par un processeur Intel, ne fonctionnent pas sur un processeur 68000. Cependant, il est très commun d'échanger des données entre machines employant des organisations différentes. Malheureusement, les valeurs de 16 et de 32 bits dans des machines de type big endian ne produisent pas des résultats corrects quand vous les utilisez dans une machine de type little endian. Et c'est en ceci que l'instruction bswap est utile. Elle vous permet de convertir facilement des valeurs big endian de 32 bits en des valeurs little endians de la même taille.

Un usage intéressant de bswap est de donner accès à un second jeu de registres généraux de 16 bits. Si, dans votre code, vous êtes en train d'utiliser uniquement des registres de 16 bits, vous pouvez doubler le nombre de registres disponibles en utilisant l'instruction bswap pour permuter les données d'un registre de 16 bits avec le mot le plus significatif d'un registre de 32 bits. Par exemple, vous pouvez garder deux valeurs de 16 bits dans eax et déplacer la valeur appropriée dans ax comme suit :

```
;Quelques opérations qui laissent leur résultat dans le registre ax
                bswap   eax

;Quelques opérations supplémentaires qui impliquent ax
                bswap   eax

;Quelques opérations impliquant la valeur originale de ax
                bswap   eax

;Opérations impliquant la seconde copie de ax ci-dessus
```

Vous pouvez utiliser cette technique pour obtenir deux copies de ax, bx, cx, dx, si, di et bp. Vous devez avoir une attention extrême en employant cette technique avec le registre sp !

Note : pour convertir une valeur big endian de 16 bits en une valeur little endian de la même taille, il vous suffira d'utiliser simplement l'instruction xchg conventionnelle. Par exemple, si ax contient une valeur big endian de 16 bits, vous pouvez la convertir en une valeur little endian (ou vice-versa) via :

```
                xchg    al, ah
```

L'instruction bswap n'affecte pas les drapeaux du registre flags.

6.4.3 L'instruction XLAT

L'instruction xlat traduit la valeur du registre al en se basant sur une table de correspondance en mémoire. Elle fait ce qui suit :

```
                temp := al+bx
                al := ds:[temp]
```

C'est-à-dire, bx pointe sur une table dans le segment de données courant. xlat remplace la valeur de al avec l'octet à l'offset qui était à l'origine dans al. Si al contient 4, xlat remplace la valeur de al avec le cinquième élément (l'offset quatre) à l'intérieur de la table pointée par ds:bx. Cette instruction a la syntaxe :

```
                xlat
```

Normalement elle n'a pas d'opérandes, vous pouvez en spécifier une, mais l'assembleur l'ignore virtuellement. Le seul intérêt du fait de spécifier une opérande est de pouvoir fournir un préfixe de surcharge du registre de segment :

```
                xlat    es:Table
```

[8]Évidemment, les adresses de mémoire dans ces machines sont organisées en ordre décroissant, ce qui résulte plus naturel. Si vous observez la mémoire de n'importe quel programme dans le système little endian vous verrez que tous les nombres que vous avez entrés dans vos variables, sont écrits à l'inverse, car les hommes lisent les nombres de gauche à droite alors que les emplacements de mémoire se lisent de droite à gauche. Le système big endian est donc plus lisible par un oeil humain, n.d.t.

Ceci indique à l'assembleur d'émettre un octet de préfixe es: avant l'instruction. Vous devez encore charger bx avec l'adresse de la table et la forme ci-dessus ne fournit pas l'adresse de la table. Seulement le segment de préfixe de surcharge de segment est significatif.

Cette instruction n'affecte pas les flags.

6.5 Instructions arithmétiques

Les processeurs 80x86 permettent de nombreuses opérations arithmétiques : addition, soustraction, négation, multiplication, division / modulo (reste), et comparaisons. Les instructions qui se chargent de ces opérations sont add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg et diverses instructions de conversion : aaa, aad, aam, aas, daa et das. La section suivante les décrit en détail.

Les formes génériques pour ces instructions sont :

```
add    dest, src               ;dest := dest + src
adc    dest, src               ;dest := dest + src + C
sub    dest, src               ;dest := dest - src
sbb    dest, src               ;dest := dest - src - C
mul    src                     ;acc  := acc * src
imul   src                     ;acc  := acc * src
imul   dest, src1, imm_src     ;dest := src1 * imm_src
imul   dest, imm_src           ;dest := dest * imm_src
imul   dest, src               ;dest := dest * src
div    src                     ;acc := xacc /-mod src
idiv   src                     ;acc := xacc /-mod src
cmp    dest, src               ;dest - src (et ajuster les flags)
neg    dest                    ;dest := -dest
inc    dest                    ;dest := dest + 1
dec    dest                    ;dest := dest - 1
xadd   dest, src               ;(voir plus loin)
cmpxchg operande1, operande2   ;idem
cmpxchg8 ax, operande          ;idem
aaa                            ;idem
aad                            ;idem
aam                            ;idem
aas                            ;idem
daa                            ;idem
das                            ;idem
```

6.5.1 Les instructions d'addition : ADD, ADC, INC, XADD, AAA et DAA

Ces instructions prennent les formes :

```
add     reg, reg
add     reg, mem
add     mem, reg
add     reg, donnee_immediate
add     mem, donnee_immediate
add     eax/ax/al, donnee_immediate

;les formes adc sont identiques à add

inc     reg
inc     mem
inc     reg16
xadd    mem, reg
xadd    reg, reg
aaa
daa
```

Notez que aaa et daa utilisent des modes d'adressage implicites et ne permettent pas d'opérandes.

6.5.1.1 Les instructions ADD et ADC

La syntaxe de add et adc (*add with carry*, (additionner avec retenue)), est semblable à mov. Comme pour mov, il y a des formes spéciales pour les registres ax et eax qui sont plus efficaces. Contrairement à mov, vous ne pouvez pas additionner une valeur et un registre de segment avec des instructions.

L'instruction add additionne le contenu de l'opérande source à l'opérande de destination. Par exemple, add ax, bx additionne bx à ax en laissant le résultat dans ax. add calcule dest:=dest+source, alors que adc calcule dest:=dest+source+C, où C représente la valeur du drapeau *carry* (retenue). Par conséquent si ce drapeau est désactivé avant l'exécution, adc se comporte exactement comme add.

Les deux instructions affectent les flags de manière identique. Elles les activent comme suit :

- Le flag *overflow* indique un dépassement de capacité signé
- Le flag *carry* indique un dépassement de capacité non signé
- Le flag *sign* indique un résultat négatif (c'est-à-dire que le bit le plus significatif du résultat est 1)
- Le flag *zero* est activé si le résultat de l'addition est zéro
- Le flag *auxiliary carry* contient 1 si un BCD dépasse la capacité du quartet[9] le moins significatif.
- Le flag *parity* est activé ou désactivé selon la parité des huit bits de l'octet le moins significatif du résultat. Si le résultat contient un nombre pair de bits, l'instruction add mettra le drapeau de parité à 1 (pour indiquer la parité). Dans le cas contraire, s'il y a un nombre impair de bits dans le résultat, l'instruction add met à zéro le flag (pour indiquer la non-parité).

Les instructions add et adc n'affectent pas les autres drapeaux.

Ces deux instructions admettent des opérandes de huit, seize et (sur des processeurs qui les supportent) trente-deux bits. Les deux opérandes de source et de destination doivent avoir la même taille. Voir le chapitre neuf pour savoir comment additionner des opérandes de tailles différentes.

Puisqu'il n'y a pas d'additions de mémoire à mémoire, pour effectuer la somme de deux variables, vous devez charger au moins un registre avec une des opérandes. L'exemple suivant, démontre des variations possibles :

```
;J := K + M
                mov     ax, K
                add     ax, M
                mov     J, ax
```

Pour additionner plusieurs valeurs, vous pouvez facilement calculer la somme dans un seul registre :

```
;J := K + M + N + P
                mov     ax, K
                add     ax, M
                add     ax, N
                add     ax, P
                mov     J, ax
```

Pour réduire le nombre d'effets de bord sur des processeurs 80486 et Pentium, vous pouvez utiliser un code comme le suivant :

```
                mov     bx, K
                mov     ax, M
                add     bx, N
                add     ax, P
                add     ax, bx
                mov     J, ax
```

L'une des choses que les débutants en assembleur oublient le plus souvent est qu'on peut additionner un registre à un emplacement de mémoire. Parfois, les débutants arrivent même à croire que les deux opérandes doivent se trouver dans des registres, en oubliant complètement la leçon du chapitre quatre. Les processeurs 80x86 sont des processeurs CISC permettant d'utiliser des modes d'adressage de mémoire avec diverses instructions, dont add. Et, souvent, c'est même plus efficace de profiter de ces avantages.

[9] *Nibble* en anglais, qui correspond à un groupe de 4 bits, soit la moitié d'un octet (voir chapitre 1), n.d.t.

```
J := K + J
                mov     ax, K   ;ceci fonctionne parce que l'addition est
                add     J, ax   ;commutative !

;Souvent, les débutants codent les opérations ci-dessus selon l'une ou l'autre
;des séquences suivantes. Ceci est inutile !
                mov     ax, J   ;Une très MAUVAISE manière d'effectuer J:=J+K
                mov     bx, K
                add     ax, bx
                mov     J, ax

                mov     ax, J   ;Un peu mieux, mais pas encore un bon moyen
                add     ax, K   ;d'effectuer J:=J+K
                mov     J, ax
```

Bien sûr, pour additionner une constante à un emplacement de mémoire, une seule instruction suffit. En effet, un CPU 80x86 vous permet d'effectuer directement une addition entre une constante et une variable :

```
J := J + 2
                add     J, 2
```

Il y a des versions spéciales des instructions add et adc qui permettent d'additionner une constante immédiate aux registres al, ax ou eax. Ces versions sont plus courtes que les instructions standard de type add reg, imm. D'autres instructions fournissent aussi des formes plus courtes quand vous utilisez ces registres ; par conséquent, essayez de garder les opérations dans le registre accumulateur (al, ax, eax) aussi souvent que possible.

```
                add     bx, 2   ;3 octets de long
                add     al, 2   ;2 octets de long
                add     bx, 2   ;4 octets de long
                add     ax, 2   ;3 octets de long
                etc.
```

Une autre optimisation avec les instructions add et adc consiste en l'utilisation de petites constantes signées. Si une valeur est dans la plage -128,+127, les instructions add et adc effectueront une extension signée pour toute constante de huit bits jusqu'à la taille nécessaire pour la destination (huit, seize ou trente-deux bits). Par conséquent, avec add et adc, utilisez de petites constantes (si possible).

6.5.1.2 L'instruction INC

L'instruction inc (incrément) additionne 1 à son opérande. Sauf pour le drapeau de retenue, inc affecte les flags de la même façon que le ferait *add operande, 1.*

Notez qu'il y a deux formes de *inc* pour les registres de 16 et de 32 bits. Il y a inc reg et inc reg_{16}. inc reg et inc mem sont la même chose. Cette instruction consiste en un opcode byte suivi par mod-reg-rém byte (voir l'annexe D pour les détails). L'instruction inc reg_{16} a un seul opcode byte ; par conséquent, elle est plus courte et souvent plus rapide.

L'opérande de l'instruction inc peut être un registre ou un emplacement de mémoire de 8, 16 ou (sur des processeurs qui le supportent) 32 bits.

L'instruction inc est plus compacte et souvent plus rapide que add reg, 1 ou add mem, 1. En fait, inc reg_{16} a seulement un octet de long, ce qui fait que deux de ces instructions sont plus courtes que l'instruction comparable add reg, 2 ; néanmoins, les deux instructions d'incrémentation sont moins rapides dans certains membres modernes de la famille 80x86.

L'instruction inc est très importante parce qu'elle ajoute 1 à un registre, est une opération très commune, comme l'incrémentation des variables de contrôle de boucle ou celle des index des tableaux, un cas parfait pour l'instruction inc. Et le fait qu'inc n'affecte pas le drapeau de retenue est aussi très important, car ceci permet d'incrémenter les index des tableaux sans affecter le résultat d'une opération arithmétique à précision multiple (voir le début du chapitre 9 pour plus de détails sur cette arithmétique).

6.5.1.3 L'instruction XADD

XADD (Exchange and Add) est une autre instruction disponible seulement à partir du microprocesseur 80486. Cette instruction additionne l'opérande de source et l'opérande de destination et garde le résultat dans la destination. Cependant, juste avant de stocker la somme, elle copie la valeur originale de l'opérande de destination dans l'opérande de source. L'algorithme suivant décrit l'opération :

```
xadd    dest, source

temp := dest
dest := dest + source
source := temp
```

XADD modifie les flags exactement comme ADD ; elle admet des opérandes de huit, seize et trente-deux bits, mais la source et la destination doivent avoir la même taille.

6.5.1.4 Les instructions AAA et DAA

AAA (*ASCII adjust after addition*) et DAA (*decimal adjust for addition*) supportent l'arithmétique *Décimal Codé Binaire*. Au-delà de ce chapitre, on ne couvrira pas les arithmétiques BCD et ASCII, puisque ces notations sont destinées à des applications de contrôle et non à des techniques générales de programmation. Les valeurs BCD sont des entiers décimaux codés en forme binaire, représentant un nombre décimal (0...9) par quartet. Les valeurs numériques ASCII contiennent un nombre décimal par octet et le quartet le plus significatif doit être à zéro.

Les instructions aaa et daa modifient le résultat d'une addition binaire pour le corriger selon l'arithmétique décimale ou ASCII. Par exemple, pour additionner deux valeurs BCD, on doit les additionner comme si elles étaient des nombres binaires et ensuite on doit exécuter daa afin de corriger le résultat. De la même façon, vous pouvez utiliser aaa pour ajuster le résultat selon le mode ASCII après avoir effectué une instruction add. Veuillez noter que ces deux instructions prennent pour acquis que les opérandes de add sont des valeurs décimales ou ASCII correctes. Si vous additionnez des valeurs binaires (non décimales ou non ASCII) et vous essayez de les ajuster à l'aide de ces instructions, vos résultats seront erronés.

Le choix du nom « arithmétique ASCII » est malheureux, car ces valeurs ne sont pas de vrais caractères ASCII. Un nom comme « BCD décompacté » serait plus approprié. Néanmoins Intel utilise le nom ASCII, donc ce livre le fait aussi pour ne pas créer des confusions inutiles. Néanmoins, vous entendrez souvent le terme « BCD décompacté » pour décrire ce type de donnée.

L'instruction aaa, que vous exécutez généralement après une instruction add, adc ou xadd, vérifie la valeur de al pour voir si un overflow de type BCD s'est produit. Elle fonctionne selon l'algorithme de base suivant :

```
 if((al and 0Fh) > 9 or (AuxC[10] = 1) then
                if(8088 or 8086)[11] then
                        al := al + 6
                else
                        ax := ax + 6
                endif

                ah := ah + 1
                AuxC := 1       ;Active le drapeau de retenue auxiliaire
                Carry := 1      ;et le drapeau de retenue simple
else
                AuxC := 0       ;Désactive le drapeau de retenue auxiliaire
                Carry := 0      ;et le drapeau de retenue simple
endif
al := al and 0Fh
```

L'instruction aaa est principalement utile pour additionner des chaînes de chiffres quand il y a exactement un chiffre décimal par octet dans une chaîne de nombres. Ce livre ne s'occupera pas des chaînes numériques BCD

[10]AuxC indique le drapeau *auxiliary carry* dans le registre flags.
[11]Les processeurs 8086/8088 fonctionnent différemment par rapport aux microprocesseurs plus modernes, mais pour toutes les opérandes valides, tous les membres de la famille 80x86 produisent des résultats corrects.

ou ASCII, donc vous pouvez tranquillement les ignorer pour l'instant. Vous pourrez sans doute utiliser l'instruction aaa toutes les fois que vous avez besoin de l'algorithme ci-dessus, mais ce sera probablement une situation rare.

L'instruction daa fonctionne comme aaa, mais elle traite des valeurs BCD compactées et non les valeurs décompactées correspondant à un chiffre décimal par octet que aaa traite. Tout comme pour aaa, la fonction principale de daa est d'additionner des chaînes de chiffres BCD (avec deux chiffres par octet). L'algorithme de daa est :

```
if ((al and 0Fh) > 9 or (AuxC = 1)) then
               al := al + 6
               AuxC := 1              ;Active le drapeau auxiliary carry
endif
if ((al > 9Fh) or (Carry = 1)) then
               al := al + 60h
               Carry := 1             ;Active le drapeau carry
endif
```

6.5.2 Les instructions de soustraction : SUB, SBB, DEC, AAS et DAS

Les instructions sub (substract), sbb (substract with borrow (soustraire avec retenue)), dec (décrément), aas (ASCII adjust for substraction), and das (decimal adjust for substraction), fonctionnent exactement comme vous vous attendez. Leur syntaxe est très semblable à celle de l'instruction add :

```
sub     reg, reg
sub     reg, mem
sub     mem, reg
sub     reg, val_immediate
sub     mem, val_immediate
sub     eax/ax/al, val_immediate

;Les formes de sbb sont identiques à la forme de sub.

dec     reg
dec     mem
dec     reg16
aas
das
```

L'instruction sub calcule la valeur dest := dest - src. sbb effectue dest := dest - src - C. Comme vous savez, la soustraction n'est pas commutative. Si vous voulez avoir le résultat de dest := src - dest, vous aurez besoin d'utiliser plusieurs instructions, en supposant que vous voulez préserver l'opérande de source.

Un dernier sujet qui vaut la peine de mentionner est comment l'instruction sub affecte les flags[12]. Les instructions sub, sbb et dec affectent les drapeaux de la manière suivante :

- Elles activent le flag zéro si le résultat est nul, ce qui arrive seulement si les opérandes sont égales pour sub et sbb, alors que dec active ce drapeau seulement lorsqu'elle décrémente la valeur 1.
- Toutes ces instructions activent le drapeau *sign* seulement si elles produisent un résultat négatif.
- Elles activent le flag *overflow* s'il se produit un dépassement signé de capacité.
- Le flag *auxiliary carry* est activé en accord avec les systèmes BCD/ASCII.
- Le drapeau de parité est affecté selon que le nombre de bits valant 1 dans le résultat est pair ou impair.
- sub et sbb activent le drapeau de retenue (*carry*) si un dépassement non signé de capacité a lieu. Notez que dec ne l'affecte jamais.

aas, tout comme sa correspondante aaa, permet d'effectuer des opérations sur des chaînes de nombres ASCII avec un chiffre décimal de 0 à 9 pour chaque octet. Vous pouvez utiliser cette instruction après un sub ou un sbb sur une valeur ASCII. Elle utilise l'algorithme suivant :

```
if ((al and 0Fh) > 9 or AuxC = 1) then
               al := al - 6
               ah := ah - 1
```

[12]SBB les affecte de manière similaire, sans pourtant oublier que SBB effectue dest - src - C.

```
                AuxC := 1       ;Active le drapeau de retenue auxiliaire
                Carry := 1      ;et le drapeau de retenue simple
else
                AuxC := 0       ;Désactive le flag auxiliary carry
                Carry := 0      ;et carry
endif
al := al and 0Fh
```

L'instruction das fait la même chose, mais pour des valeurs BCD, et elle utilise l'algorithme suivant :

```
if ((al and 0Fh) > 9 or AuxC = 1) then
                al := al - 6
                AuxC := 1
endif
if (al > 9Fh or Carry = 1) then
                al := al - 60h
                Carry := 1      ;active le drapeau de retenue
endif
```

Puisque la soustraction n'est pas commutative, vous ne pouvez pas utiliser sub aussi librement que add. Les exemples qui suivent montrent certains des problèmes que vous pourriez rencontrer :

```
;J := K - J
                mov     ax, K   ;Bon essai, mais il calcule J:=J-K
                sub     J, ax   ;et la soustraction n'est pas commutative !

                mov     ax, K   ;Solution correcte.
                sub     ax, J
                mov     J, ax

;J := J - (K + M) -- n'oubliez pas que c'est équivalent à J := J - K - M
                mov     ax, K   ;Effectue AX := K + M
                add     ax, M
                sub     J, ax   ;Effectue J := J - (K + M)

                mov     ax, J   ;Une autre solution, mais moins efficace
                sub     ax, K
                sub     ax, M
                mov     J, ax
```

Notez que les instructions sub et sbb, comme add et adc, fournissent des formes courtes pour soustraire une constante du registre accumulateur (al, ax ou eax). Pour cette raison, essayez de garder les soustractions arithmétiques dans l'accumulateur autant que possible. Ces instructions fournissent également une forme plus courte quand vous soustrayez des constantes dans la plage -128...127 à un registre ou à un emplacement de mémoire. L'instruction fait automatiquement une extension signée pour ces constantes selon la taille de destination avant d'effectuer la soustraction. Lisez l'annexe D pour les détails.

En pratique, on n'a pas vraiment besoin d'une instruction qui soustrait une constante d'un registre ou d'une variable : additionner une valeur négative produit le même résultat. Néanmoins, Intel fournit des instructions de soustraction directes.

Après une soustraction, les drapeaux de contrôle de code (*carry*, *sign*, *overflow* et *zero*) contiennent des valeurs que vous pouvez tester pour voir si une opérande est égale, différente, inférieure, inférieure ou égale, supérieure, ou supérieure/égale à l'autre opérande. Voyez le paragraphe suivant pour plus de détails.

6.5.3 L'instruction CMP

L'instruction cmp (compare) est identique à l'instruction sub, avec une différence cruciale : elle ne garde pas le résultat dans l'opérande de destination. La syntaxe est très similaire à celle de sub, sa forme générale est :

```
                cmp     dest, src
```

Les formes spécifiques sont :

```
                cmp     reg, reg
```

```
cmp     reg, mem
cmp     mem, reg
cmp     reg, donnee_immediate
cmp     mem, donnee_immediate
cmp     eax/ax/al, donnee_immediate
```

Cette instruction met à jour les drapeaux du registre flags, selon le résultat de l'opération de soustraction qu'elle effectue (destination - source). Vous pouvez tester les résultats en contrôlant les drapeaux appropriés du registre flags. Pour des détails sur cette opération voir les paragraphes 6.6.5 (instructions d'« Activation sous Condition ») et 6.9.4 (instructions de « Saut conditionnel »).

Normalement, on effectue un saut conditionnel après une instruction cmp. Ce procédé (comparer deux valeurs et ajuster les flags, puis les tester avec les instructions conditionnelles de saut), constitue un mécanisme très efficace pour faire prendre des décisions à un programme.

Probablement, le meilleur point de départ, à l'heure d'explorer les instructions de comparaison est voir exactement comment cmp affecte les flags. Considérez l'instruction suivante :

```
cmp     ax, bx
```

Cette instruction calcule ax-bx et modifie les drapeaux selon le résultat de l'opération. Les flags sont modifiés comme suit :

Z : Le flag *zero* est activé si et seulement si ax = bx. C'est le seul cas où ax - bx produit un résultat nul. Donc, vous pouvez utiliser le flag zéro pour vérifier l'égalité ou l'inégalité.

S : Le flag *sign* est activé si le résultat est négatif. De premier abord, on pourrait croire que ce drapeau est activé si ax est inférieur à bx, mais ce n'est pas toujours le cas. Si ax=7FFFh et bx=-1 (0FFFFh), le fait de soustraire ax de bx produit 8000h, lequel est négatif (et donc le drapeau de signe est mis). Par conséquent, pour des comparaisons signées, le drapeau de signe ne contient pas l'état correct. Pour des opérandes non signées, considérez ax = 0FFFFh et bx=1. ax est plus grand que bx, mais leur différence est 0FFFEh, laquelle est encore négative. En vérité, les drapeaux de signe et de dépassement de capacité, pris ensemble, peuvent être utilisés pour comparer deux valeurs signées.

O : Le drapeau de dépassement de capacité (*overflow*) est activé après une opération de comparaison si, la différence entre ax et bx produit un dépassement de capacité (négatif ou positif). Comme mentionné ci-dessus, le drapeau *sign* et le drapeau *overflow* sont utilisés ensemble pour effectuer des comparaisons signées.

C : Le drapeau de retenue est mis après une opération de comparaison si le fait de soustraire bx de ax requiert une retenue. Ce qui arrive seulement quand ax est plus petit que bx, étant ax et bx non signés.

L'instruction cmp affecte aussi le drapeau de parité et le drapeau de retenue auxiliaire, mais on teste rarement ces drapeaux après une opération de comparaison. Etant donné que cmp active les flags de cette manière, vous pouvez tester la comparaison des deux opérandes avec les drapeaux suivants :

```
cmp Oper1, Oper2
```

Tableau 27 : codes de condition des flags après CMP

Opérandes non signées :	Opérandes signées :
Z : égalité / inégalité	Z : égalité / inégalité
C : $oper_1 < oper_2$ (C=1) $oper_1 >= oper_2$ (C=0)	C : pas de signification
S : pas de signification	S : voir ci-dessous
O : pas de signification	O : voir ci-dessous

Pour des comparaisons signées, S (sign) et O (overflow), pris ensemble, ont la signification suivante : Si ((S=0) et (O=1)) ou ((S=1) et (O=0)) alors oper1<oper2 quand il s'agit d'une comparaison signée. Si ((S=0) et (O=0)) ou ((S=1) et (O=1)) alors oper1>=oper2 quand il s'agit d'une comparaison signée.

Pour comprendre pourquoi ces drapeaux sont activés de cette façon, considérez les exemples suivants :

```
oper1           moins           oper2           S       O
-----           -----           -----           -       -

0FFFF (-1)        -             0FFFE (-2)      0       0
08000 (-32768)    -             00001           0       1
0FFFE (-2)        -             0FFFF (-1)      1       0
07FFF (32767)     -             0FFFF (-1)      1       1
```

Rappellez-vous que l'opération cmp est réellement une soustraction, par conséquent, le premier exemple ci-dessus effectue (-1)-(-2) et on a (+1). Le résultat est positif et un dépassement ne se produit pas, donc les deux drapeaux S et O sont à zéro. Puisque (S xor O)[13] vaut zéro, alors oper1 est supérieure ou égale à oper2.

Dans le second exemple, cmp effectue (-32768)-(+1) = (-32769). Puisqu'un entier de 16 bits ne peut représenter ce nombre, le résultat fait le tour pour devenir 7FFFh (+32767) et il active le drapeau de dépassement de capacité. Puisque le résultat est positif (au moins dans les limites de 16 bits), le drapeau de signe est nettoyé (mis à zéro). Puisque (S xor O) vaut 1 ici, alors oper1 est inférieure à oper2.

Dans le troisième exemple, cmp effectue (-2)-(-1), ce qui donne -1. Il n'y a pas de dépassement de capacité (le flag O vaut 0), mais le résultat est négatif, donc le flag sign est mis à 1. Et puisque (S xor O) vaut 1, alors oper1 est encore inférieure à oper2.

Finalement, dans le dernier exemple, cmp calcule (32767) - (-1), ce qui donne (+32768). Ceci active overflow. De plus, la valeur fait le tour et devient 8000h (-32768) et le drapeau de signe est également activé. Et, puisque (S xor O) vaut zéro ici, alors oper1 est supérieure ou égale à oper2.

6.5.4 Les instructions CMPXCHG et CMPXCHG8B

L'instruction cmpxchg (compare and exchange) est disponible seulement sur les processeurs 80486 et supérieurs et elle supporte la syntaxe suivante :

```
cmpxchg         reg, reg
cmpxchg         mem, reg
```

Les opérandes doivent avoir la même taille (huit, seize ou trente-deux bits). Cette instruction utilise aussi le registre accumulateur en choisissant automatiquement al, ax ou eax pour satisfaire la taille des opérandes.

Cette instruction compare al, ax ou eax avec la première opérande et active le flag zéro si elles sont égales. Si c'est le cas, cmpxchg copie la seconde opérande dans la première, sinon, la première opérande est copiée dans l'accumulateur. L'algorithme suivant décrit l'opération :

```
cmpxchg         oper1, oper2
if({al/ax/eax} = oper1) then[14]
        zero := 1               ;Active le flag zéro
        oper1 := oper2
else
        zero := 0               ;Nettoie le flag
        {al/ax/eax} := oper1
endif
```

cmpxchg supporte des structures de données de certains systèmes d'exploitation requérant des *opérations atomiques*[15] et les *sémaphores*. Sans doute, si vous pouvez insérer cet algorithme dans votre programme, alors vous pouvez utiliser cmpxchg de façon appropriée.

Note : contrairement à l'instruction cmp, cmpxchg affecte seulement le drapeau zéro. Vous ne pouvez pas tester les autres flags comme vous pouvez le faire avec cmp.

Le processeur Pentium supporte une instruction de comparaison et permutation de 64 bits : cmpxchg8b. La syntaxe est :

```
cmpxchg8b       ax, mem64
```

[13] Se souvenir que xor a, b = (a and b') or (a' and b), n.d.t.
[14] Le choix de al, ax ou eax sera fait en vertu de la taille des opérandes et les deux doivent avoir la même taille.
[15] Une opération atomique est une opération que le système d'exploitation ne peut pas interrompre.

L'instruction compare la valeur de 64 bits dans edx:eax avec la valeur en mémoire. Si elles sont égales, alors le Pentium stocke la valeur de ecx:ebx dans la mémoire, sinon, il charge la paire edx:eax avec le contenu de cet emplacement. Cette instruction affecte le drapeau zéro selon le résultat et n'affecte aucun autre flag.

6.5.5 L'instruction NEG

neg (negate) prend le complément à deux d'un octet ou d'un mot. Elle comporte une seule opérande (destination) et la nie. La syntaxe est :

```
neg     dest
```

Elle calcule ce qui suit :

```
dest := 0 - dest
```

Ce qui inverse effectivement le signe de l'opérande.

Si l'opérande est zéro, le signe ne change pas, même si ceci met à zéro le drapeau de retenue. Nier toute autre valeur affecte ce drapeau. Alors que soumettre à *not* un octet contenant -128, un mot contenant -32768 ou un double-mot contenant -2 147 483 648 ne change pas l'opérande, mais active le drapeau overflow. *neg* met toujours à jour les drapeaux A, S, P et Z comme si vous utilisiez l'instruction sub.

Les formes possibles sont :

```
neg     reg
neg     mem
```

Les opérandes peuvent être de huit, seize ou trente-deux bits.

Quelques exemples :

```
;J := -J
                neg     J
;J := -K
                mov     ax, K
                neg     ax
                mov     J, ax
```

6.5.6 Les instructions de multiplication : MUL, IMUL et AAM

Les instructions de multiplication vous offrent un premier exemple d'irrégularité du jeu 80x86. Des instructions comme add, adc, sub, sbc et beaucoup d'autres utilisent un octet mod-reg-r/m pour admettre deux opérandes. Malheureusement, il n'y a pas assez de bits dans l'octet d'opcode du 8086 pour supporter toutes les instructions ; le 8086 utilise donc les bits de reg dans l'octet mod-reg-r/m comme extension de l'opcode. Par exemple, inc, dec et neg ne requièrent pas deux opérandes et les CPU 80x86 utilisent les *bits reg* comme extension pour les huit bits de l'opcode traditionnel. Ceci fonctionne merveilleusement avec des instructions ayant une seule opérande, permettant aux concepteurs Intel d'encoder diverses instructions (huit, en fait) avec un seul opcode.

Mais l'instruction de multiplication requiert un traitement spécial et les concepteurs Intel, étant encore à court d'opcodes, ont conçu une instruction de multiplication comprenant une seule opérande. Le champ reg contient une extension de l'opcode au lieu d'une valeur de registre et la multiplication est, sans aucun doute, une opération à deux opérandes. Le 8086 sous-entend toujours l'accumulateur (al, ax, eax) comme registre de destination. Cette irrégularité rend l'instruction de multiplication du 8086 une opération un peu plus difficile que les autres, parce que une des deux opérandes doit se trouver dans l'accumulateur. Si Intel a adopté cette approche non orthodoxe c'est que ses concepteurs estimaient que l'instruction de multiplication allait être utilisée beaucoup moins souvent que *add* et *sub*.

Un problème qui surgit en rendant disponible seulement une forme mod-reg-r/m est qu'on ne peut pas multiplier l'accumulateur par une constante, car l'octet mod-reg-r/m ne supporte pas le mode d'adressage immédiat. Intel a bientôt découvert le besoin de multiplier l'accumulateur par une constante et ils ont donné un

support pour ce problème à partir du processeur 80286[16]. C'était spécialement important pour l'accès aux tableaux à plusieurs dimensions. A partir du 80386, Intel a généralisé une forme de multiplication permettant des opérandes mod-reg-r/m standard.

Il y a maintenant deux formes d'instruction de multiplication : une forme non signée (mul) et une signée (imul). Contrairement aux additions et aux soustractions, on a besoin d'instructions séparées pour ces deux opérations.

La multiplication admet les syntaxes suivantes :

```
;Multiplications non signées :
        mul     reg
        mul     mem

;Multiplications signées :
        imul    reg
        imul    mem
        imul    reg, reg, immediate     (2)
        imul    reg, mem, immediate     (2)
        imul    reg, immediate          (2)
        imul    reg, reg                (3)
        imul    reg, mem                (3)

;Multiplications BCD :
        aam

;2- Disponible seulement sur processeurs 80286 et supérieurs
;3- Disponible seulement sur processeurs 80386 et supérieurs
```

Comme vous pouvez le voir, les instructions de multiplication constituent un vrai désordre. Et, pire, ce n'est qu'en utilisant au moins un 80386 que vous pouvez en exploiter pleinement les fonctionnalités. Et, finalement, il y a même certaines restrictions à ces instructions qui ne sont pas évidentes ci-dessus. Hélas, le seul moyen de s'y retrouver est de mémoriser leur fonctionnement.

mul, disponible dans tous les processeurs, multiplie des opérandes de huit, seize ou trente-deux bits. Notez qu'en multipliant deux valeurs de *n* bits, le résultat peut exiger jusqu'à 2*n bits. Par conséquent, si l'opérande est une quantité de huit bits, le résultat peut demander seize bits. De la même façon, une opérande de 16 bits produit un résultat de 32 bits et une opérande de 32 bits peut engendrer un résultat de 64 bits.

L'instruction mul, avec une opérande de huit bits, multiplie le registre al par l'opérande et garde le résultat de 16 bits dans ax. Donc :

```
        mul     operande8
;ou

        imul    operande8

;produit :

        ax := al * operande8
```

où "*" représente une multiplication non signée pour mul et une multiplication signée pour imul.

Si vous spécifiez une opérande de 16 bits, mul et imul calculent :

```
        dx:ax := ax * operande16
```

où "*" a la même signification que ci-dessus et dx:ax veut dire que dx contient le mot le plus significatif du résultat de 32 bits et ax contient le mot le moins significatif du même résultat.

Si vous spécifiez une opérande de 32 bits, alors mul et imul effectuent :

```
        edx:eax := eax * operande32
```

[16]Dans la puce originale du 8086, la multiplication par une constante était toujours plus rapide à l'aide des décalages, des additions et des soustractions. Ce fut peut-être pour cette raison que Intel ne donna pas trop d'importance à ce type de multiplication. Cependant, la version de mul pour le 80286 était plus rapide que celle développée pour le 8086, donc elle pouvait désormais remplacer les instructions de décalage, addition et soustraction correspondantes.

où edx correspond au double-mot le plus significatif du résultat et eax au double-mot le moins significatif de ce dernier.

Si un produit 8x8, 16x16 ou 32x32 requiert plus de huit, seize ou trente-deux bits (respectivement), les instructions mul et imul activent les drapeaux *carry* et *overflow*.

mul et *imul* affectent les flags A, P, S et Z. Notez particulièrement que les drapeaux *sign* et *zero* ne contiennent pas des valeurs significatives après avoir effectué ces deux instructions.

imul (multiplication entière) fonctionne sur des opérandes signées. Il y a diverses formes de cette instruction au fur et à mesure qu'Intel à essayé de généraliser cette instruction avec des processeurs successifs. Les paragraphes précédents décrivent la première forme de cette instruction, la version avec une seule opérande. Trois autres formes sont disponibles seulement à partir du 80286 et elles fournissent la capacité de multiplier un registre par une valeur immédiate. Les deux dernières formes, disponibles seulement à partir du 80386 permettent de multiplier un registre arbitraire avec un autre registre arbitraire ou un emplacement de mémoire. Ces versions (en accord aussi avec les tailles respectives) sont :

```
imul    oper1, oper2, immed     ;Forme générale

imul    reg16, reg16, immed8
imul    reg16, reg16, immed16
imul    reg16, mem16, immed8
imul    reg16, mem16, immed16
imul    reg16, immed8
imul    reg16, idded16
imul    reg32, reg32, immed8        (3)
imul    reg32, reg32, immed32       (3)
imul    reg32, mem32, immed8        (3)
imul    reg32, mem32, idded32       (3)
imul    reg32, immed8               (3)
imul    reg32, immed32              (3)

;3- Disponible seulement sur des processeurs 80386 et supérieurs.
```

Les instructions imul reg, immed, constituent des syntaxes spéciales que l'assembleur fournit ; les encodages de ces instructions sont les mêmes que imul reg, reg, immed. L'assembleur fournit simplement la même valeur de registre pour les deux opérandes.

Ces instructions effectuent :

```
operande1 := operande2 * valeur_immediate
operande1 := operande1 * valeur_immediate
```

À part le nombre d'opérandes, il y a plusieurs différences entre ces formes et les versions de mul/imul à une seule opérande :

- Il n'y a pas de multiplication 8x8 bits disponible (l'opérande $immediate_8$ constitue simplement une forme plus courte de l'instruction. À l'intérieur, le CPU effectue une extension signée de l'opérande à 16 ou à 32 bits, selon la nécessité).
- Ces instructions ne produisent pas un résultat de 2*n bits c-à-d, une multiplication 16x16 produit un résultat de 16 bits ; de même, une multiplication 32x32 produit un résultat de 32 bits. Ces instructions modifient seulement les drapeaux de retenue et de dépassement de capacité si le résultat est trop grand pour le registre de destination.
- La version 80286 d'imul permet une constante comme opérande, alors que les versions standard de mul/imul non.

Les deux dernières versions de imul sont disponibles seulement à partir du 80386. Avec ces formats additionnels, l'instruction imul est *presque* aussi générale que l'instruction add[17]:

```
imul reg, reg
imul reg, mem
```

Ces instructions effectuent :

[17]Mais, il y a encore certaines restrictions pour ce qui concerne la taille des opérandes qu'il faut tenir en considération, par exemple, le non support pour les opérandes de huit bits.

```
reg := reg * reg
reg := reg * mem
```

Les deux opérandes doivent avoir la même taille. Par conséquent, comme pour la version 80286 de l'instruction imul, vous devez tester les flags *carry* et *overflow* pour détecter tout dépassement éventuel. Si cela a lieu, le CPU perd les bits forts du résultat.

Note importante : gardez à l'esprit que le drapeau zéro contient une valeur indéterminée après une instruction de multiplication et que vous ne pouvez pas le tester pour voir si le résultat est nul. L'effet est le même avec le drapeau de signe. Si vous avez besoin de tester ces flags, vous pouvez comparer le résultat avec zéro après avoir testé les drapeaux de retenue et de dépassement de capacité.

aam (ASCII Adjust after Multiplication), comme aaa et aas, permet d'ajuster une valeur décimale décompactée après une multiplication. Cette instruction opère directement sur le registre ax et suppose que vous avez multiplié deux valeurs de huit bits de la plage 0..9 et que le résultat a été placé dans ax (en réalité, le résultat sera gardé dans al, puisque 9*9 = 81, la plus grande valeur possible ; ah doit contenir zéro). Cette instruction divise ax par 10, laisse le quotient dans ah et le reste dans al :

```
ah := ax div 10
al := ax mod 10
```

Contrairement aux autres instructions de mise en format décimal/ASCII, les programmes utilisent aam régulièrement, puisque les conversions entre bases utilisent cet algorithme.

Note : aam contient un opcode de deux octets, le second étant la constante immédiate 10. Les programmeurs ont découvert que si l'on substitue une autre valeur immédiate à cette constante, vous pouvez changer le diviseur de l'algorithme ci-dessus. Cette caractéristique, cependant, n'est pas documentée. Elle fonctionne sur tous les types de processeurs qu'Intel a produits jusqu'à présent ; cependant, il n'y a pas de garantie que les processeurs à venir continuent à lui donner du support. Bien sûr, avec des processeurs 80286 et supérieurs vous pouvez effectuer la multiplication avec une constante, mais cette astuce n'est guère nécessaire sur des systèmes modernes.

Sur les processeurs 80x86, il n'y a pas d'instruction *dam* (Decimal Adjust for Multiplication).

L'usage le plus fréquent de l'instruction imul est de calculer les offsets des tableaux multidimensionnels et celle-ci a été sûrement la raison principale pour laquelle Intel a intégré dans les 80286 la capacité de multiplier un registre par une constante. Dans le chapitre quatre, on a utilisé l'instruction mul standard pour les calculs des index des tableaux ; cependant, cette syntaxe étendue d'imul constitue un meilleur choix, comme on montre dans cet exemple :

```
MonTableau      word    8 dup ( 7 dup ( 6 dup (?)))     ;tableau 8x7x6
J               word    ?
K               word    ?
M               word    ?
                  .
                  .
;MonTableau[J, K, M] := J + K - M

                mov     ax, J
                add     ax, K
                sub     ax, M

                mov     bx, J   ;index := ((J*7 + K) * 6 + M) * 2
                imul    bx, 7
                add     bx, K
                imul    bx, 6
                add     bx, M
                add     bx, bx  ;BX := BX * 2

                mov     MonTableau[bx], ax
```

N'oubliez pas que les instructions de multiplication sont très lentes, souvent dix fois plus lentes qu'une instruction d'addition. En définitive, il y a des moyens plus rapides de multiplier une valeur par une constante. Regardez le paragraphe 9.5.1 (chapitre neuf) pour tous les détails.

6.5.7 Les instructions de division : DIV, IDIV et AAD

Ces instructions effectuent une division 64/32 bits (pas disponible sur des processeurs antérieurs au 80386), de 32/16 bits ou de 16/8 bits. Elles admettent les formes :

```
;Pour divisions non signées :
div     reg
div     mem

;Pour divisions signées :
idiv    reg
idiv    mem

;Ajustement pour division ASCII :
aad
```

L'instruction div effectue une division non signée. Si l'opérande est de huit bits, div divise le registre ax par l'opérande, en laissant le quotient dans al et reste (modulo) dans ah. Si l'opérande est une quantité de 16 bits, alors l'instruction de 32 bits divise le contenu de dx:ax par l'opérande en laissant le quotient dans ax et le reste dans dx. Avec des opérandes de 32 bits, div divise la valeur de 64 bits dans edx:eax par l'opérande, en laissant le quotient dans eax et le reste dans edx.

Vous ne pouvez pas juste diviser une valeur de huit bits par une autre. Si le dénominateur est une valeur de huit bits, le numérateur doit être de seize bits. Si vous avez besoin de diviser une valeur non signée de huit bits par une autre, vous devez d'abord effectuer une extension non signée sur le numérateur, en l'élargissant à 16 bits ; vous pouvez le faire en chargeant le numérateur dans al et zéro dans ah ; ensuite, vous pouvez diviser ax par le dénominateur pour produire le résultat correct. *Ne pas effectuer cette extension à 16 bits avant d'effectuer la division peut amener le 80x86 à produire des résultats erronés !*[18] Si vous devez diviser deux valeurs de 16 bits, vous emploierez le même système, vous élargissez le registre AX (qui contient le numérateur) en plaçant zéro dans le registre dx[19]. Si vous avez besoin de diviser deux valeurs de 32 bits, il en sera de même, en "élargissant" le registre eax dans edx (en chargeant zéro dans edx) avant la division.

En travaillant avec des entiers signés, vous aurez besoin d'effectuer une extension signée de al, dans ax, de ax dans eax ou de eax dans edx, avant d'utiliser idiv, selon la taille de l'opérande. Pour ce faire, utilisez cbw, cwd, cdq ou movsx. Si vous n'effectuez pas d'extension signée avant div, vos résultats seront probablement incorrects.

Il y a un autre fait à noter à propos des instructions de division : elles peuvent également causer des erreurs fatales. D'abord il y a la fameuse division par zéro ; ensuite le quotient peut être trop grand pour les registres eax, ax ou al. Par exemple, la division 16/8 bits de "8000h / 2" produit le quotient 4000h avec un reste de zéro, mais 4000h est trop grand pour huit bits ! Si ceci arrive, ou si vous essayez de diviser une valeur par zéro, le processeur générera l'exception *int 0*, ce qui veut dire que le BIOS affichera *"division by zero"* ou *"divide error"* et le programme sera interrompu. Si cela arrive, il se peut que vous n'avez pas étendu votre numérateur avant l'exécution de la division. Puisque cette erreur fait planter les programmes, il faut faire beaucoup d'attention aux valeurs qu'on emploie quand on effectue des divisions.

Les flags *auxiliary carry, carry, overflow, parity, sign* et *zero* sont indéfinis après une opération de division. Si un dépassement de capacité a lieu (ou si vous encourez à la division par zéro), le 80x86 exécuté une INT 0 (*interrupt zero*).

Notez que les processeurs 80286 et ultérieurs ne fournissent pas de formes spéciales de idiv, comme ils le font pour imul. La plupart des programmeurs utilisent la division encore moins que la multiplication, par conséquent, les concepteurs d'Intel ne se sont pas préoccupés de créer des instructions spéciales pour cette opération. Notez qu'il n'y a pas de moyen de diviser un registre par une valeur immédiate, vous devrez la charger dans un registre ou dans un emplacement de mémoire avant d'effectuer la division.

L'instruction aad (*ASCII adjust before division*) est une autre opération décimale non compactée. Elle décompose les valeurs décimales codées binaires décompactées avant une opération de division ASCII. Bien que ce livre ne couvre pas d'arithmétique BCD, aad reste utile pour d'autres opérations. L'algorithme qui la décrit est :

[18] Les erreurs de logique ne sont pas des erreurs de syntaxe, ils sont acceptés par l'assembleur, mais ils produisent des résultats incorrects. Ce sont, ce qu'on appelle populairement "bogues" (anglicisation de "bug"), n.d.t.

[19] Sur des processeurs 80386 ou ultérieurs, vous pouvez utiliser l'instruction movzx.

```
        al := ah*10 + al
        ah := 0
```

Cette instruction est utile pour convertir des chaînes de chiffres en valeurs entières (voir les questions à la fin de ce chapitre).

Les exemples suivants montrent comment diviser une valeur de seize bits par une autre :

```
;J := K / M (non signé)
                mov     ax, K
                mov     dx, 0   ;Extension non signée
        ;En pratique, on devrait vérifier que M est différent de 0
        ;ici qu'il est >
                div     M
                mov     J, ax

;J := K / M (signé)
                mov     ax, K   ; charger la dividende
                cwd             ;Extension signée de AX dans DX
        ;En pratique, on devrait vérifier que M soit différent de 0
        ;ici qu'il est >
                idiv    M
                mov     J, ax

;J := (K * M) / P
                mov     ax, K   ;Notez que l'instruction imul produit
                imul    M       ;un résultat de 32 bits dans DX:AX, donc
                idiv    P       ;on n'a pas besoin d'élargir ax ici.
                mov     J, ax   ;Espérez et priez que le résultat
                                ;tienne dans 16 bits !
```

6.6 Instructions logiques et de manipulation de bits

La famille 80x86 fournit cinq instructions logiques, quatre de rotation et trois de décalage. Les instructions logiques sont and, or, xor, test et not ; les instructions de rotation sont ror, rol, rcr et rcl ; de décalage sont shl/sal, shr et sar. Les processeurs 80386 et ultérieurs fournissent un jeu d'opérations encore plus riche : il y a bt, bts, btr, btc, bsf, bsr, shld, shrd et l'ensemble d'instructions set (set*cc*).

Ces instructions peuvent manipuler des bits, convertir des valeurs, effectuer des opérations logiques, compacter/décompacter des données et effectuer des opérations arithmétiques. Les paragraphes suivants décrivent chacune de ces instructions en détail.

6.6.1 Les instructions logiques : AND, OR, XOR et NOT

Les instructions logiques 80x86 fonctionnent sur une base bit-à-bit (*bitwise*) ; elles s'appliquent autant sur des données de 8 bits que sur des données de 16 ou de 32 bits. *and*, *not*, *or* et *xor* font ce qui suit :

```
        and     dest, src       ;dest := dest and src
        or      dest, src       ;dest := dest or src
        xor     dest, src       ;dest := dest xor src
        not     dest            ;dest := not dest

 ;Les variantes spécifiques sont :
        and     reg, reg
        and     mem, reg
        and     reg, mem
        and     reg, donnee_immediate
        and     mem, donnee_immediate
        and     eax/ax/al, donnee_immediate

        ;or et xor utilisent les mêmes formats que and

        not     reg
        not     mem
```

Sauf pour not, elles affectent les flags comme suit :

- Elles nettoient carry.
- Elles nettoient overflow.
- Elles activent le drapeau zéro si le résultat est zéro, et le nettoient sinon.
- Elles copient l'octet le plus significatif du résultat dans le drapeau de signe.
- Elles ajustent le drapeau *parity* selon la parité (du nombre de bits) du résultat.
- Elles affectent aussi le flag de retenue auxiliaire.

L'instruction not n'affecte aucun flag.

Tester le flag *zero* après ces instructions est particulièrement utile. *and* active ce flag si *les deux opérandes n'ont aucun 1 dans les positions de bits correspondantes* (puisque ceci produirait un résultat de 0). Par exemple, si l'opérande de source contient un seul bit à 1, alors le flag *zero* sera activé si le bit correspondant dans la destination est zéro ; sinon le résultat sera 1. L'instruction *or* vaudra zéro seulement si les deux opérandes contiennent zéro. Alors que *xor* produit un résultat de zéro si et seulement si les deux opérandes sont égales. Les programmeurs exploitent souvent ce fait pour nettoyer un registre de 16 ou 32 bits, puisqu'une instruction de la forme

```
xor     reg16, reg16
```

est plus rapide que l'instruction mov reg, 0 comparable.

Tout comme les instructions d'addition et de soustraction, and, or et xor admettent aussi des formes spéciales avec le registre accumulateur et les données immédiates (constantes). Ces formes sont plus courtes et parfois plus rapides que la forme générale "mov reg, imm". Bien que généralement avec ces instructions, on ne pense pas en termes de valeurs signées, toutefois la technologie 80x86 fournit des versions de "reg/mem, imm" qui font des extensions de signe de valeurs de la plage -128..127 à des valeurs de 16 ou 32 bits si nécessaire.

Comme toujours, les opérandes doivent avoir la même taille. Sur des processeurs antérieurs au 80386, cette taille pouvait être de 8 ou de 16 bits ; sur des processeurs à partir du 80386, la taille peut atteindre 32 bits. Ces instructions effectuent l'opération logique bit-à-bit indiquée sur leurs opérandes. Pour des détails sur ces opérations, revoir le chapitre 1.

On peut utiliser l'opération *and* pour mettre à zéro des bits sélectionnés dans l'opérande de destination. Ceci est connu comme *masquage des données* (*masking out data*). De même, vous pouvez utiliser l'instruction *or* pour forcer certains bits à 1 dans l'opérande de destination (voir la section 9.6 pour plus de détails). Vous pouvez utiliser ces instructions avec les instructions de décalage et de rotation pour compacter et décompacter des données (voir encore la section 9.6).

6.6.2 Les instructions de décalage : SHL/SAL, SHR, SAR, SHLD et SHRD

Les processeurs 80x86 supportent trois différents types de décalages (shl et sal sont la même instruction) : shl (shift left), sal (shift arithmetic left), shr (shift right) et sar (shift arithmetic right). Les processeurs 80386 et ultérieurs fournissent deux décalages additionnels : shld et shrd.

Les instructions de décalage déplacent les bits à l'intérieur d'un registre ou d'un emplacement de mémoire. Le format général est :

```
shl     dest, nb_fois
sal     dest, nb_fois
shr     dest, nb_fois
sar     dest, nb_fois
```

où *dest* est la valeur à décaler et *nb_fois* spécifie le nombre de positions binaires à décaler. Par exemple, shl décale les bits dans la destination vers la gauche selon un nombre de positions spécifié par l'opérande *nb_fois*. Les instructions shld et shrd ont le format :

```
shld    dest, source, nb_fois
shrd    dest, source, nb_fois
```

Les formes spécifiques de ces instructions sont :

```
shl     reg, 1
```

```
shl     mem, 1
shl     reg, imm        ;(2)
shl     mem, imm        ;(2)
shl     reg, cl
shl     mem, cl

;sal est un synonyme de shl et utilise le même format.
;shr a le même format que shl.
;sar a le même format que shl.

shld    reg, reg, imm   ;(3)
shld    mem, reg, imm   ;(3)
shld    reg, reg, cl    ;(3)
shld    mem, reg, cl    ;(3)

;shrd a le même format que shld.

;(2)- Disponible seulement sur des processeurs 80286 ou ultérieurs
;(3)- Disponible seulement sur des processeurs 80386 ou ultérieurs
```

Pour les processeurs 8088 et 8086, le nombre de bits à décaler est soit 1, soit la valeur de cl. Sur des processeurs 80286 et supérieurs vous pouvez utiliser toute constante immédiate de huit bits. Il va de soi que cette valeur doit être inférieure ou égale au nombre de bits de l'opérande de destination. Ce serait une perte de temps décaler vers la gauche al de neuf bits (car huit produiraient le même résultat, comme vous verrez bientôt). Algorithmiquement, vous pouvez imaginer l'opération de décalage de plus d'une position comme suit :

```
for temp := 1 to nb_fois do
        shift dest, 1
```

Il y a des différences mineures dans la façon dont les instructions de décalage traitent le drapeau overflow quand *nb_fois* est différent de 1, mais vous pouvez ignorer ce fait la plupart du temps.

Les instructions shl, sal, shr et sar fonctionnent avec des opérandes de huit, seize ou trente-deux bits, alors que shld et shrd fonctionnent seulement avec des opérandes de destination de 16 ou de 32 bits.

6.6.2.1 SHL/SAL

Les termes shl et sal sont synonymes, représentent la même instruction et utilisent le même encodage binaire. Ces instructions déplacent chaque bit de l'opérande de destination d'une position binaire vers la gauche selon le nombre de fois spécifié par la seconde opérande. Après chaque décalage, le bit le moins significatif reçoit la valeur zéro et le bit le plus significatif finit dans le drapeau de retenue (voir Figure 6.2).

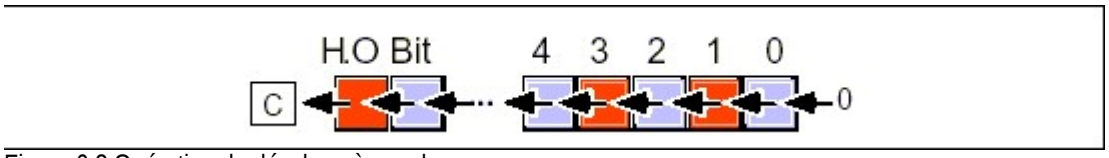


Figure 6.2 Opération de décalage à gauche

L'instruction shl/sal affecte les drapeaux comme suit :

- Si le nombre de fois est zéro, l'instruction shl n'affecte aucun flag.
- Le drapeau carry contient le dernier bit décalé en dehors du bit le plus haut de l'opérande.
- Overflow contient 1 si le bit fort avant le décalage était différent du bit le plus haut actuel. Si le nombre de décalages est supérieur à 1, alors overflow est indéfini.
- Le drapeau de zéro vaut 1 si le décalage produit zéro.
- Le drapeau de signe contiendra le bit le plus haut du résultat.
- Le flag parity contiendra 1 si l'octet le moins significatif du résultat a un nombre pair de bits valant 1.
- Le flag de retenue auxiliaire est toujours indéfini après une instruction de décalage.

L'instruction de décalage à gauche est particulièrement utile pour compacter des données. Par exemple, supposez que vous avez deux quartets dans al et ah que vous voulez combiner en un octet ; vous pouvez utiliser le code suivant :

```
shl     ah, 4   ;Forme qui demande un 80286 ou supérieur
or      al, ah  ;Fusionne ah dans les quatre bits forts de al
```

Il va sans dire que, pour que ce code fonctionne correctement, al doit contenir une valeur de la plage 0..F, (l'instruction de décalage à gauche nettoie automatiquement le quartet fort de ah avant l'exécution de or). Si, avant cette opération, les quatre bits les plus hauts de al sont différents de zéro, vous pouvez les nettoyer facilement avec un and :

```
shl     ah, 4   ;Déplace les bits les plus bas en position haute
and     al, 0Fh ;Nettoie les 4 bits les plus hauts (avec 0)
or      al, ah  ;Fusionne les bits
```

Puisque le fait de décaler une valeur entière d'une position vers la gauche équivaut à la multiplier par deux, il va de soi que vous pouvez utiliser cette instruction pour multiplier une valeur par des puissances de deux :

```
shl     ax, 1   ;AX*2
shl     ax, 2   ;AX*4
shl     ax, 3   ;AX*8
shl     ax, 4   ;AX*16
shl     ax, 5   ;AX*32
shl     ax, 6   ;AX*64
shl     ax, 7   ;AX*128
shl     ax, 8   ;AX*256
etc.
```

Notez que shl ax, 8 équivaut aux deux instructions suivantes :

```
mov     ah, al
mov     al, 0
```

L'instruction shl/sal peut multiplier par deux des valeurs aussi non signées que signées, pour chaque décalage. Cette instruction affecte le drapeau de retenue si le résultat est trop grand pour l'opérande de destination (c'est-à-dire qu'un dépassement de capacité non signé se produit). De même, cette instruction affecte overflow si la taille du résultat signé dépasse la capacité de la destination. Ceci se produit si, après un décalage, vous placez un 0 dans le bit le plus haut d'un nombre négatif ou bien un 1 dans le bit le plus haut d'un nombre positif.

6.6.2.2 SAR

L'instruction SAR décale les bits de la destination vers la droite, en préservant la valeur originale du bit le plus haut[20] (voir la figure 6.3).

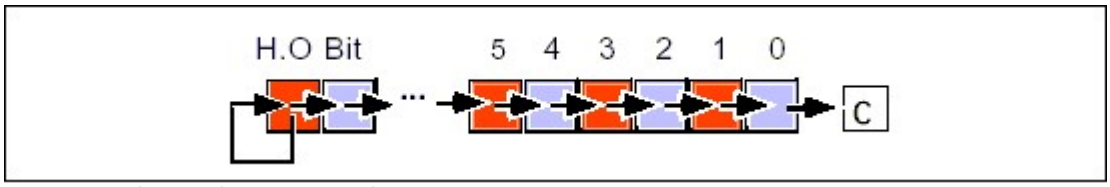


Figure 6.3 Décalage à droite arithmétique

Cette instruction modifie les drapeaux comme suit :

- S'il n'y a pas de décalage, l'instruction n'affecte aucun flag.
- Le drapeau carry contient le dernier bit décalé en dehors du bit les plus bas de l'opérande.

[20]Autrement dit, il s'agit d'un décalage signé, n.d.t.

- Overflow contient 0 si le nombre de décalages est égal à 1. Un dépassement de capacité ne peut jamais se produire avec sar. Cependant, si le nombre des décalages est supérieur à 1, la valeur de overflow est indéfinie.
- Le drapeau de zéro vaut 1 si le dernier décalage a produit zéro.
- Le drapeau de signe contiendra le bit haut du résultat.
- Le drapeau de parité contiendra 1 si l'octet le moins significatif du résultat a un nombre pair de bits valant 1.
- Le flag de retenue auxiliaire est toujours indéfini après une instruction de décalage.

La fonction principale de sar est effectuer une division signée de l'opérande par une puissance de deux. Chaque décalage vers la droite équivaut à diviser la valeur par deux. Des décalages à droite multiples ont pour effet de diviser le résultat précédent par deux, donc plusieurs décalages produisent les résultats suivants :

```
sar     ax, 1   ;AX/2
sar     ax, 2   ;AX/4
sar     ax, 3   ;AX/8
sar     ax, 4   ;AX/16
sar     ax, 5   ;AX/32
sar     ax, 6   ;AX/64
sar     ax, 7   ;AX/128
sar     ax, 8   ;AX/256
etc.
```

Il y a une très importante différence entre sar et idiv. idiv arrondit toujours à zéro, alors que sar arrondit toujours à l'entier le plus proche inférieur ou égal au résultat. Pour des résultats positifs, un décalage arithmétique d'une position produit le même résultat d'une division entière par deux. Cependant, si le quotient est négatif, idiv arrondit vers zéro et sar vers moins l'infini. Les exemples suivants démontrent la différence :

```
mov     ax, -15
cwd
mov     bx, 2
idiv            ;Produit -7

mov     ax, -15
sar     ax, 1   ;Produit -8
```

Gardez ceci à l'esprit si vous utilisez sar pour des divisions entières.

L'instruction sar ax, 8 copie en fait ah dans al et effectue une extension signée de al dans ax. Ceci parce que sar ax, 8 décale entièrement ah dans al et copie le bit le plus haut de ah dans toutes les positions de cet octet[21]. Il en résulte que, sur des processeurs 80286 ou supérieurs, vous pouvez utiliser sar pour étendre avec signe un registre dans un autre. Les séquences d'instructions suivantes fournissent des exemples de cet usage :

```
;Equivalent à CBW :
                mov     ah, al
                sar     ah, 7

;Equivalent à CWD :
                mov     dx, ax
                sar     dx, 15

;Equivalent à CDQ :
                mov     edx, eax
                sar     edx, 31
```

Il peut paraître bête d'utiliser deux instructions quand une seule suffirait, mais cbw, cwd et cdq étendent uniquement al vers ax, ax vers dx:ax et eax vers edx:eax. De même, movsx copie son opérande signée dans une destination dont la taille est deux fois plus grande que la source. L'instruction sar, au contraire, permet d'effectuer une extension signée d'un registre dans un autre registre ayant la même taille et contenant les bits du signe.

```
;Extension signée de cx dans bx:cx
                mov     cx, bx
                sar     cx, 15
```

[21]En effet, huit décalages d'une position viennent de se produire, n.d.t.

6.6.2.3 SHR

L'instruction shr décale les bits de la destination vers la droite ; à chaque décalage, le bit le plus haut est affecté par un zéro (voir figure 6.4).

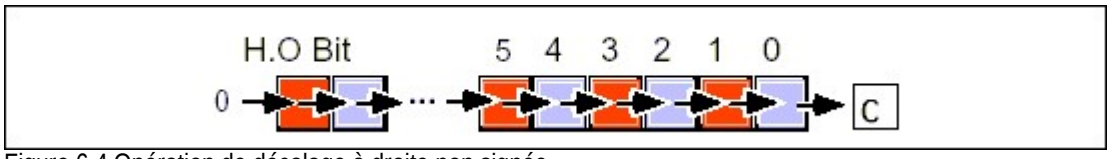


Figure 6.4 Opération de décalage à droite non signée

L'instruction SHR modifie les drapeaux comme suit :

- Si le nombre de positions du décalage est zéro, l'instruction n'affecte aucun flag.
- Le drapeau carry contient le dernier bit décalé en dehors du bit de poids faible de l'opérande.
- Si le nombre de décalages est égal à 1 le drapeau overflow contiendra la valeur que le bit haut avait avant le décalage (c'est-à-dire, l'instruction active le flag overflow si le signe change). Cependant, si le nombre des décalages est supérieur à 1, la valeur d'*overflow* est indéfinie.
- Le drapeau de zéro vaut 1 si le décalage produit un résultat de zéro.
- Le drapeau de signe contiendra le bit haut du résultat, qui est toujours zéro.
- Le drapeau de parité contiendra 1 s'il y aura un nombre pair de "1" dans l'octet le moins significatif du résultat.
- Le flag de retenue auxiliaire est toujours indéfini après une instruction de décalage.

L'instruction de décalage à droite est particulièrement utile pour décompacter des données. Par exemple, supposez que vous voulez extraire les deux quartets du registre al, en plaçant le quartet le plus haut dans ah et le quartet le plus bas dans al. Vous pouvez le faire comme suit :

```
mov     ah, al  ;Obtenir une copie du quartet haut
shr     ah, 4   ;Déplacer le quartet haut dans le quartet bas
                ;nettoyer le quartet haut.
and     al, 0Fh ;Supprimer le quartet haut de al
```

Puisque décaler une valeur entière non signée vers la droite est équivalent à la diviser par deux, vous pouvez également utiliser cette instruction pour réaliser des divisions par des puissances de deux :

```
shr     ax, 1   ;AX/2
shr     ax, 2   ;AX/4
shr     ax, 3   ;AX/8
shr     ax, 4   ;AX/16
shr     ax, 5   ;AX/32
shr     ax, 6   ;AX/64
shr     ax, 7   ;AX/128
shr     ax, 8   ;AX/256
etc.
```

Notez que shr ax, 8 est équivalent aux deux instructions suivantes :

```
mov     al, ah
mov     ah, 0
```

Souvenez-vous que les divisions par deux via shr fonctionnent seulement avec des opérandes non signées. Si ax contient -1 et vous exécutez shr ax, 1, le résultat dans ax sera 32767 (7FFFh) et non le -1 ou le 0 que vous vous attendiez. Utilisez l'instruction sar pour diviser un entier signé par une puissance de deux quelconque.

6.6.2.4 Les instructions SHLD et SHRD

Ces deux instructions produisent des décalages à gauche et à droite de double précision et sont disponibles seulement sur des processeurs 80386 ou ultérieurs. Leur forme générique est :

```
shld    operande1, operande2, donnee_immediate
shld    operande1, operande2, cl
shrd    operande1, operande2, donnee_immediate
shrd    operande1, operande2, cl
```

operande$_2$ doit être un registre de 16 ou de 32 bits, *operande1* peut être un registre ou un emplacement de mémoire et les deux opérandes doivent avoir la même taille. L'opérande *donnee_immeditate* peut être une valeur allant de 0 à n-1, où *n* est le nombre de bits dans les deux opérandes ; ce nombre spécifie le nombre de bits à décaler.

L'instruction shld décale les bits de l'opérande$_1$ vers la gauche. Le bit le plus haut affectera le drapeau de retenue (carry) et le bit le plus haut d'opérande$_2$ est décalé dans le bit le plus base d'opérande$_1$. Notez que cette instruction ne modifie pas la valeur d'opérande$_2$, mais en utilise simplement une copie temporaire pendant le décalage. Si le nombre de décalages est *n*, alors shld décale le bit *n-1* dans le drapeau carry. shld décale aussi les *n* bits hauts d'opérande$_2$ dans les *n* bits bas d'opérande$_1$. Graphiquement l'instruction shld est illustrée à la figure 6.5.

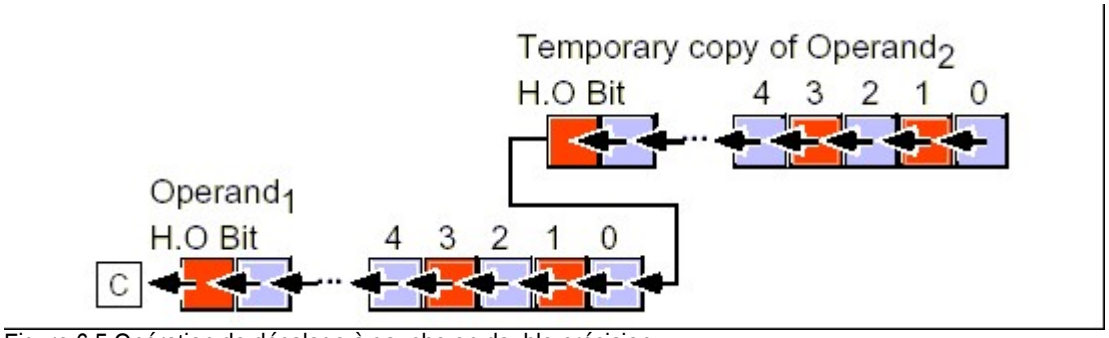


Figure 6.5 Opération de décalage à gauche en double précision

L'instruction shld, affecte les flags comme suit :

- Si le nombre de positions du décalage est zéro, aucun flag ne sera affecté.
- Le drapeau carry contient le dernier bit décalé en dehors de l'opérande$_1$.
- Si le nombre de décalages est 1, le drapeau *overflow* contiendra 1 si le bit du signe de l'opérande$_1$ change pendant l'opération. Cependant, si le nombre des décalages est supérieur à 1, la valeur de overflow sera indéfinie.
- Le drapeau zéro vaut 1 si le décalage produit un résultat de zéro.
- Le drapeau de signe contiendra le bit haut du résultat.

Cette instruction est utile pour compacter des données de plusieurs sources différentes. Par exemple, supposez que vous voulez créer un mot en fusionnant les quartets hauts de quatre autres mots. Vous pouvez le faire comme suit :

```
mov     ax, valeur4     ;Obtenir le quartet haut
shld    bx, ax, 4       ;Copier les bits hauts de AX dans BX
mov     ax, valeur3     ;Obtenir le quartet #2
shld    bx, ax, 4       ;Fusionner dans bx
mov     ax, valeur2     ;Obtenir le quartet #1
shld    bx, ax, 4       ;Fusionner dans bx
mov     ax, valeur1     ;Obtenir le quartet bas.
shld    bx, ax, 4       ;BX contient maintenant les 4 quartets.
```

L'instruction shrd est semblable à shld, mais, naturellement, le décalage se fait vers la droite. Pour avoir une vision exacte du fonctionnement de shrd, considérez la figure 6.6.

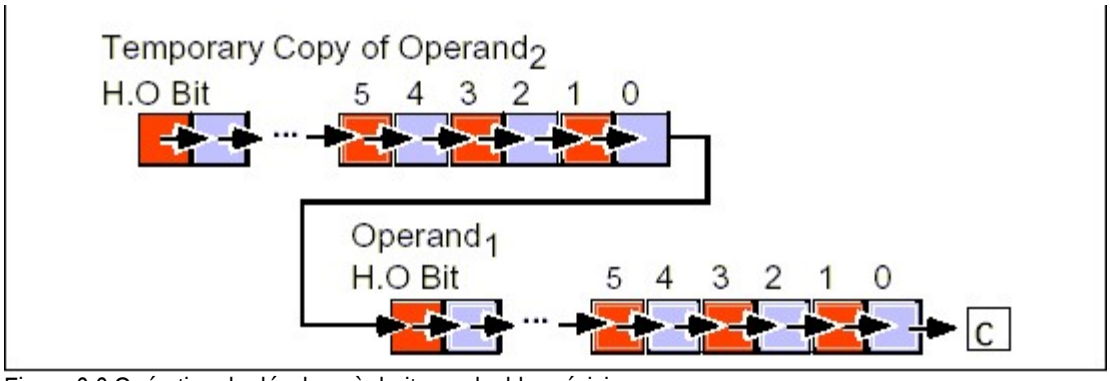


Figure 6.6 Opération de décalage à droite en double précision

Cette instruction modifie les drapeaux comme suit :

- Aucun flag ne sera affecté s'il ne se produit pas de décalage.
- Le drapeau carry contient le dernier bit décalé en dehors du bit le plus bas de l'opérande$_1$.
- Si le nombre de décalages est égal à 1 le drapeau overflow contiendra 1 si le bit du signe de l'opérande$_1$ change pendant l'opération. Cependant, si le nombre des décalages est supérieur à 1, la valeur de overflow sera indéfinie.
- Le drapeau de zéro vaut 1 si le décalage produit un résultat de zéro.
- Le drapeau de signe contiendra le bit le plus haut du résultat.

Franchement, ces deux instructions pourraient être un peu plus utiles si l'opérande$_2$ pouvait être un emplacement de mémoire. Intel les a conçues pour permettre des décalages rapides (64 bits ou plus) en précision multiple. Pour plus d'informations sur ces utilisations, voir le paragraphe 9.3.11.

L'instruction shrd est légèrement plus utile que shld pour compacter des données. Par exemple, supposons que ax contient une valeur de la plage 0..99 représentant une année (1900..1999), que bx contient une valeur de la plage 1..31 représentant un jour et cx une valeur de 1 à 12 représentant un mois (voir le paragraphe 1.10). Vous pouvez facilement utiliser shrd pour compacter toutes ces informations dans dx :

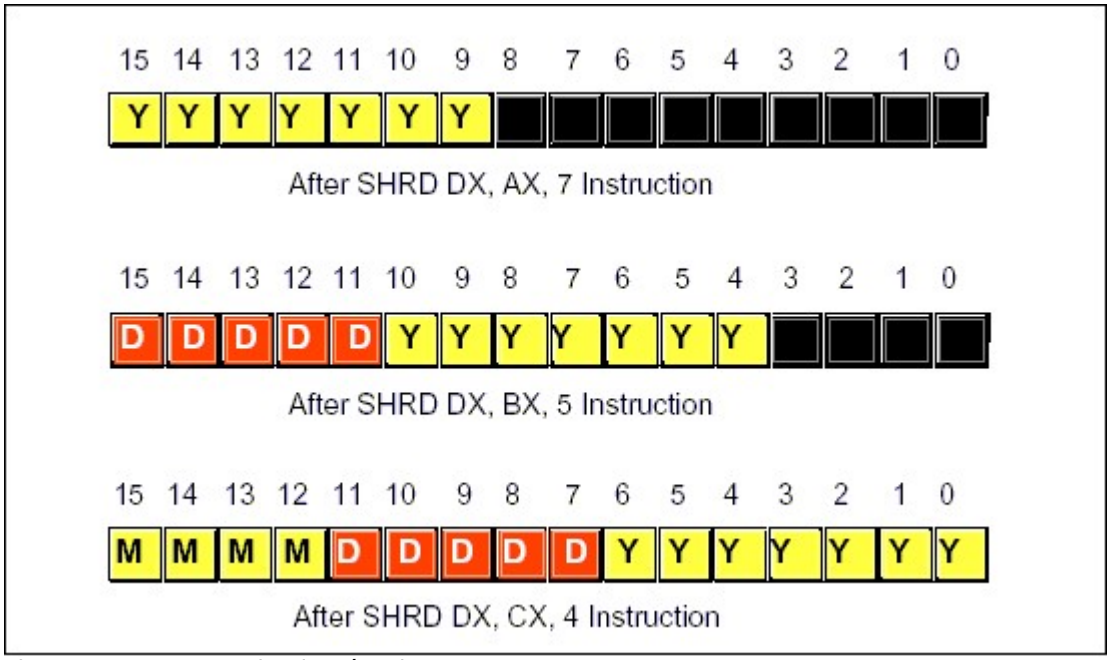


Figure 6.7 Compactage des données via SHRD

```
shrd    dx, ax, 7
shrd    dx, bx, 5
shld    dx, cx, 4
```

Regardez la figure 6.7 pour visualiser l'exemple pas à pas.

6.6.3 Les instructions de rotation : RCL, RCR, ROL et ROR

Les instructions de rotation décalent les bits tout comme les instructions de décalage, mais le bit sortant de l'opérande est réinséré de l'autre côté de celle-ci. Ces instructions comprennent rcl (*rotate through carry left*), rcr (*rotate through carry right*), rol (*rotate left*) et ror (*rotate right*). Elles prennent la forme :

```
rcl     dest, nb_fois
rol     dest, nb_fois
rcr     dest, nb_fois
ror     dest, nb_fois
```

Les versions spécifiques sont :

```
rcl     reg, 1
rcl     mem, 1
rcl     reg, imm        ;Seulement sur des 80286 ou supérieurs
rcl     mem, imm        ;idem
rcl     reg, cl
rcl     mem, cl

;rol utilise le même format que rcl.
;rcr utilise le même format que rcl.
;ror utilise le même format que rcl.
```

6.6.3.1 RCL

Le rcl (rotate through carry left), comme son nom l'indique, fait tourner les bits vers la gauche, remplit le carry flag avec la valeur du bit haut et revient au bit zéro à droite (voir Figure 6.8).

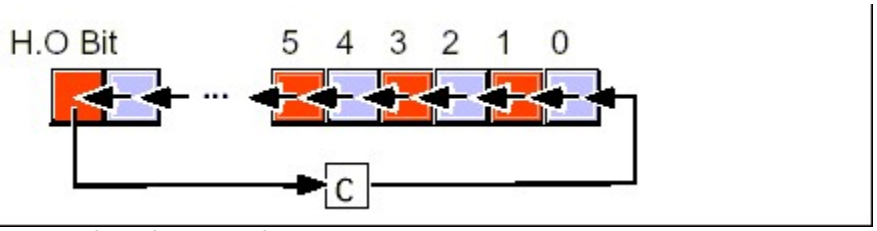


Figure 6.8 L'opération *Rotate Through Carry Left*

Notez que si vous faites un rcl sur un objet n+1 fois, où *n* est le nombre de bits dans l'objet, vous revenez à votre valeur originale. Gardez à l'esprit cependant que certains drapeaux peuvent contenir des valeurs différentes après n+1 rotations.

rcl affecte les flags comme suit :

- Le drapeau carry contient le bit haut qui est sorti de l'opérande.
- Si le nombre de décalages est 1, rcl affecte le drapeau overflow selon le changement du signe. En cas de plusieurs décalages, la valeur de overflow reste indéfinie.
- L'instruction rcl ne modifie pas les drapeaux zero, sign, parity ou auxiliary carry.

Avertissement important : contrairement aux instructions de décalage, les instructions de rotation n'affectent pas les drapeaux de zéro, de signe, de parité ou de retenue auxiliaire. Ce manque de rigueur peut vous causer beaucoup d'ennuis si vous l'oubliez et tentez de tester ces drapeaux après une opération rcl. Si vous avez besoin

de tester un de ces flags, testez d'abord carry et overflow (si nécessaire), puis comparez le résultat avec zéro pour activer les autres drapeaux.

6.6.3.2 RCR

L'instruction rcr est le complément de rcl. Elle décale les bits vers la droite, les bits les plus bas sortants vont dans le drapeau de retenue et retournent ensuite dans les bits hauts (voir figure 6.9).

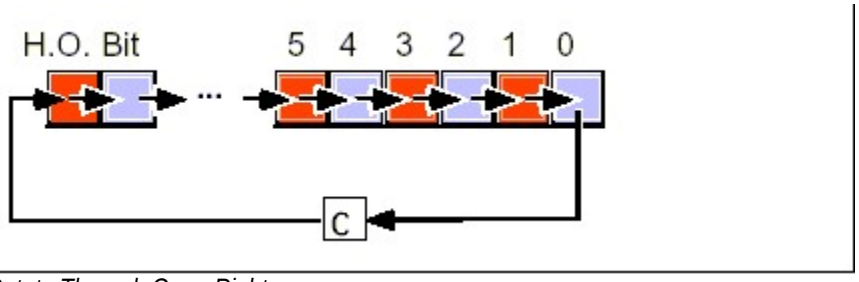


Figure 6.9 L'opération Rotate Through Carry Right

Cette instruction affecte les flags comme rcl :

- Le drapeau carry contient le dernier bit bas qui est sorti de l'opérande.
- Si le nombre de décalages est 1, rcr affecte le drapeau overflow selon le changement du signe (voulant dire que les valeurs du bit haut et du carry n'étaient pas les mêmes avant l'exécution de cette instruction). En cas de plusieurs décalages, la valeur d'overflow est indéfinie.
- L'instruction rcr ne modifie pas les drapeaux zero, sign, parity ou auxiliary carry.

Gardez à l'esprit l'avertissement donné pour rcl ci-dessus.

6.6.3.3 ROL

L'instruction rol est semblable à rcl, dans la mesure où elle effectue une rotation à gauche selon un nombre spécifié de bits. La différence majeure est que le bit haut sortant n'affecte pas le carry flag, mais il retourne directement au bit zéro. rol copie aussi le résultat du dernier bit sortant dans le drapeau de retenue (voir figure 6.10).

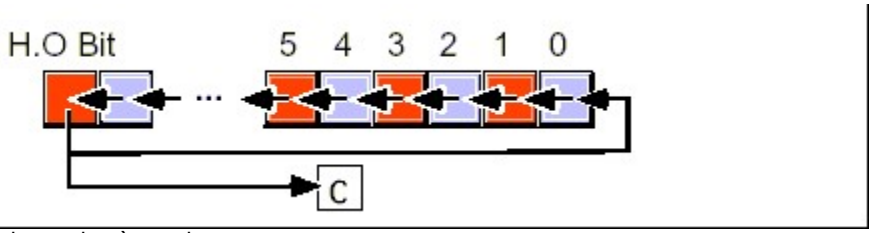


Figure 6.10 Opération de rotation à gauche

L'instruction rol affecte les flags de manière identique à rcl. En fait, la seule différence avec rcl est celle qu'on vient de mentionner. *Et n'oubliez pas l'avertissement à propos des flags !*

Tout comme shl, l'instruction rol est souvent utile pour compacter ou décompacter des données. Par exemple supposez de vouloir extraire les bits 10..14 de ax et laisser ces quatre bits dans la plage 0..4. Les deux séquences d'instructions qui suivent font précisément ceci :

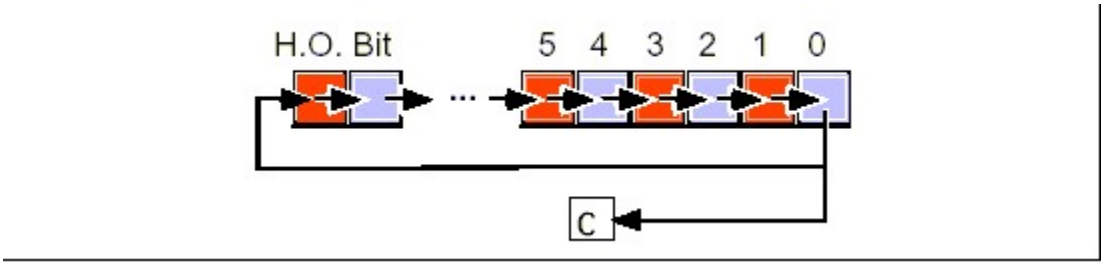
```
shr     ax, 10
and     ax, 1Fh

rol     ax, 6
and     ax, 1Fh
```

6.6.3.4 ROR

L'instruction ror se rapporte à l'instruction rcr de la même manière que l'instruction rol par rapport à rcl. C'est-à-dire qu'il s'agit presque de la même opération, à l'exception de la source de le bit d'entrée de l'opérande. Plutôt que de décaler le drapeau de retenue précédent dans le bit le plus haut de l'opération de destination, il décale le bit zéro dans le bit le plus haut (voir Figure 6.11).

Figure 6.11 Opération de rotation à droite



ror affecte les flags de façon identique à rcr. La seule différence avec rcr est qu'elle fait rentrer le bit sortant à gauche directement dans le bit zéro de l'opérande, au lieu de le faire passer dans carry. *N'oubliez pas, encore une fois, l'avertissement about les drapeaux affectés !*

6.6.4 Les opérations sur les bits

Manipuler les bits, est une des opérations qui sont beaucoup plus faciles en assembleur qu'en tout autre langage. Et il ne faut pas s'étonner. La plupart des langages de haut niveau tendent à faire écran entre vous et la représentation machine sous-jacente des types de données[22]. Des instructions comme and, or, xor, not et des instructions comme les décalages et les rotations permettent de tester, activer, nettoyer, inverser ou aligner des champs de bits à l'intérieur de chaînes de bits. Même le langage C++, fameux pourtant pour ses opérateurs de manipulation de bits, ne fournit pas autant de possibilités de manipulation que l'assembleur.

Les processeurs 80386 et supérieurs de la famille 80x86, vont même beaucoup plus loin. En fait, à côté des instructions logiques, de décalage et de rotation, il y a des instructions permettant de tester, activer, nettoyer ou invertir des bits *spécifiques* à l'intérieur d'une opérande ou de rechercher des bits activés. Ces instructions sont :

```
test    dest, source
bt      source, index
btc     source, index
btr     source, index
bts     source, index
bsf     dest, source
bsr     dest, source
```

Les formes spécifiques sont :

```
test    reg, reg
test    reg, mem
test    mem, reg ;C'est la même instruction que test reg, mem
test    reg, imm
test    mem, imm
test    eax/ax/al, imm

bt      reg, reg                        ;(3)
bt      mem, reg                        ;(3)
bt      reg, imm                        ;(3)
bt      mem, imm                        ;(3)

;btc utilise le même format que bt      (3)
;btr utilise le même format que bt      (3)
;bts utilise le même format que bt      (3)
```

[22]D'ailleurs le fait de masquer les détails de bas niveau est l'une des fonctions principales de ces langages.

```
bsf     reg, reg                        ;(3)
bsf     reg, mem                        ;(3)

;bsr utilise les mêmes formats que bsf   (3)

;3- Cette instruction est seulement disponible sur 80386 ou
supérieur
```

Notez que bt, btc, btr, bts, bsf et bsr requièrent des opérandes de 16 ou de 32 bits.

Les opérations sur les bits sont utiles pour implémenter des fonctions graphiques primitives (monochromes) de mappage (tables) de bits ou pour implémenter un jeu de données via des tables de bits.

6.6.4.1 Test

Cette instruction effectue un AND sur ses deux opérandes et modifie les drapeaux, mais n'enregistre pas le résultat. *test* et *and* ont la même relation que cmp et sub. Normalement, on l'utilise pour savoir si un bit contient la valeur 1. Considérez l'exemple suivant :

```
test    al, 1
```

Cette instruction effectue un and entre al et 1 ; si le bit zéro de al contient 1, alors le résultat est non-zéro et le processeur nettoie le flag zéro. Si le bit zéro de al n'est pas activé, alors le résultat est zéro (FAUX) et l'opération *test* active le drapeau zéro. Vous pouvez tester ce drapeau après cette instruction pour savoir si le bit le plus bas du registre al contenait 1 ou 0.

test peut aussi vérifier si un ou plusieurs bits dans un registre ou un emplacement de mémoire sont à 1. Considérez ce qui suit :

```
test    dx, 105h
```

Cette instruction effectue un ET logique entre dx et la valeur 105h. Ceci produira un résultat différent de zéro (en mettant aussi à zéro le *zero flag*), si au moins un des bits zéro, deux ou huit contiennent 1. Ils doivent être tous à zéro pour que le drapeau zéro soit activé.

L'instruction test affecte les flags de manière identique à l'instruction and :

- Le drapeau de retenue est nettoyé.
- Il en va de même pour le drapeau overflow.
- Active le drapeau zéro si le résultat est zéro et le nettoie sinon.
- Copie le bit le plus haut du résultat et le place dans le drapeau de signe.
- Active le drapeau de parité selon la parité (nombre de bits activés) de l'octet bas du résultat.
- Rend indéterminé le flag de retenue auxiliaire.

6.6.4.2 Les instructions pour tester les bits : BT, BTS, BTR et BTC

Sur des processeurs 80386 et ultérieurs, on peut utiliser l'instruction bt (*bit test*) pour tester un bit isolé. Sa seconde opérande spécifie l'*index du bit* de la première opérande. bt copie le bit adressé dans le drapeau de retenue. Par exemple,

```
bt      ax, 12
```

copie le bit 12 du registre ax dans le drapeau de retenue.

Les instructions bt/bts/btr/btc admettent seulement des opérandes de 16 ou de 32 bits. Il ne s'agit pas d'une limitation de l'instruction. Après tout, si vous voulez tester le bit 3 du registre al, vous pouvez tout simplement tester le bit 3 du registre ax tout entier. D'autre part, si l'index dépasse la taille du registre, le résultat est indéfini.

Si la première opérande est un emplacement de mémoire, bt teste le bit à l'offset donné, quelle que soit la valeur de l'index. Par exemple, si bx contient 65, alors

```
bt      TestezMoi, bx
```

copiera le bit 1 de l'emplacement TestezMoi+8 dans le drapeau de retenue. Encore une fois, la taille de l'opérande ne compte pas. Pour tous les usages et les objectifs, l'opérande de mémoire est un octet et vous pouvez tester tout bit après cet octet avec un index approprié. Le bit que bt est en train de tester est à la position *index mod 8* et à l'offset de mémoire *adresse réelle + index / 8*.

bts, btr et btc copient aussi le bit adressé dans le drapeau carry ; cependant, ces instructions activent/désactivent (nettoient) ou complémentent (inversent) aussi l'octet ciblé de la première opérande après l'avoir copié dans le flag. Ceci permet des opérations *test and set*, *test and clear* et *test and invert*, nécessaires pour certains algorithmes.

Ces instructions n'affectent pas d'autre flags que carry.

6.6.4.3 Recherche de bits (bit scanning) : BSF et BSR

Les instructions bsf (*Bit Scan Forward*) et bsr (*Bit Scan Reverse*) recherchent le premier ou le dernier bit activé dans une quantité de 16 ou de 32 bits. La forme générale de ces instructions est :

```
bsf     dest, source
bsr     dest, source
```

bsf localise le premier bit activé dans l'opérande source, en procédant du bit zéro (bit bas) jusqu'au bit le plus haut. bsr fait la même chose, mais en sens contraire, du dernier bit au bit zéro. Si ces instructions trouvent un 1, elles nettoient le flag zéro et mettent l'index (0..31) dans l'opérande de destination. Si l'opérande source est zéro, ces instructions activent le drapeau zéro et laissent dans la destination une valeur indéfinie[23].

Pour capturer le premier bit contenant zéro (et non 1), faites une copie de l'opérande source et inversez-la (via not), puis exécutez bsf ou bsr sur la valeur inversée. Si la valeur originale de la source ne contenait pas de zéros, alors le flag zéro sera activé après cette opération, sinon l'opérande de destination contiendra la position du premier bit nul rencontré.

6.6.5 Les instructions « Set on Condition » (activer à condition que)

Les instructions *set on condition* (ou set*cc*) mettent leur opérande (registre ou variable de 8 bits) à zéro ou à un, selon les valeurs du registre flags. Les formats généraux sont :

```
setcc   reg8
setcc   mem8
```

Set*cc* représente l'un des mots-clés apparaissant dans les tables qui suivent. Ces instructions placent 0 dans leur opérande correspondante si la condition est fausse ou placent 1 dans le cas contraire.

Table 28 : instructions set*cc* qui testent Flags

instruction	Signification	Condition	Commentaires
SETC	Set if carry	Carry = 1	Même chose que SETB, SETNAE
SETNC	Set if not carry	Carry = 0	Même chose que SETNB, SETAE
SETZ	Set if zero	Zero = 1	Même chose que SETE
SETNZ	Set if not zero	Zero = 0	Même chose que SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if not sign	Sign = 0	
SETO	Set if overflow	Overflow = 1	
SETNO	Set if not overflow	Overflow = 0	
SETP	Set if parity	Parity = 1	Même chose que SETPE
SETPE	Set if parity even (parité paire)	Parity = 1	Même chose que SETP
SETNP	Set if not parity	Parity = 0	Même chose que SETPO
SETPO	Set if parity odd (parité impaire)	Parity = 0	Même chose que SETNP

[23]Dans beaucoup de processeurs, si l'opérande source est nulle, le CPU laisse l'opérande de destination inaltérée. Cependant, certaines versions du 80486 modifient l'opérande de destination, donc il ne faudrait pas compter sur la stabilité de la destination, quel que ce soit le résultat de l'instruction.

Les instructions ci-dessus testent simplement les flags sans autre signification sous-jacente. Vous pouvez, par exemple, utiliser setc pour vérifier le drapeau de retenue après un décalage, une rotation, un test de bits ou une opération arithmétique. De même, vous pouvez utiliser setnz après une instruction de test pour vérifier le résultat.

D'ailleurs, l'instruction cmp fonctionne de pair avec les instructions set*cc*. Tout de suite après une instruction de comparaison, les flags donnent des informations relatives aux valeurs de ces opérandes et vous permettent de vérifier si une opérande est plus petite, égale, supérieure, ou n'importe quelle combinaison de ces dernières.

Il y a deux groupes d'instructions set*cc* qui sont très utiles après une opération de comparaison. Le premier groupe concerne les comparaisons *non signées* alors que l'autre concerne les comparaisons *signées*.

Table 29 : instructions setcc pour comparaisons non signées

instruction	Signification	Condition	Commentaires
SETA	Set if above (>)	Carry = 0, Zero = 0	Même chose que SETNBE
SETNBE	Set if not below or equal (not <=)	Carry = 0, Zero = 0	Même chose que SETA
SETAE	Set if above or equal (>=)	Carry = 0	Même chose que SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Même chose que SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Même chose que SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Même chose que SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 ou Zero = 1	Même chose que SETNA
SETNA	Set if not above (not >)	Carry = 1 ou Zero = 1	Même chose que SETBE
SETE	Set if equal (=)	Zero = 1	Même chose que SETZ
SETNE	Set if not equal (≠)	Zero = 0	Même chose que SETNZ

Le tableau correspondant pour des comparaisons signées est :

Table 30 : instructions setcc pour comparaisons signées

instruction	Signification	Condition	Commentaires
SETG	Set if greater (>)	Sign = Overflow ou Zero = 0	Même chose que SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Overflow ou Zero = 0	Même chose que SETG
SETGE	Set if greather than or equal	Sign = Overflow	Même chose que SETNL
SETNL	Set if not less than (not <)	Sign = Overflow	Même chose que SETGE
SETL	Set if less than (<)	Sign ≠ Overflow	Même chose que SETNGE
SETNGE	Set if not greather or equal (not >=)	Sign ≠ Overflow	Même chose que SETL
SETLE	Set if less than or equal (<=)	Sign ≠ Overlow ou Zero = 1	Même chose que SETNG
SETNG	Set if not greather than (not >)	Sign ≠ Overlow ou Zero = 1	Même chose que SETLE
SETE	Set if equal (=)	Zero = 1	Même chose que SETZ
SETNE	Set if not equal (≠)	Zero = 0	Même chose que SETNZ

Les instructions set*cc* sont particulièrement intéressantes parce qu'elles peuvent convertir le résultat d'une comparaison en une valeur booléenne (vrai/faux ou 1/0). Ceci est spécialement important quand on traduit des instructions de langages de haut niveau en instructions d'assembleur. L'exemple suivant montre comment on le fait :

```
; Bool := A <= B
              mov     ax, A   ;en supposant que A et B sont des entiers signés
              cmp     ax, B
              setle   Bool    ;où Bool doit être une variable d'un octet
```

Puisque les instructions set*cc* produisent toujours 0 ou 1, on peut combiner leur résultat avec celui des instructions logiques and et or pour calculer des valeurs booléennes complexes :

```
; Bool := ((A <= B) and (D = E)) or (F <> G)
```

```
mov     ax, A
cmp     ax, B
setle   bl
mov     ax, D
cmp     ax, E
setle   bh
and     bl, bh
mov     ax, F
cmp     ax, G
setne   bh
or      bl, bh
mov     Bool, bh
```

Pour plus d'exemples voir le paragraphe 9.2 (chapitre 9).

Les instructions setcc produisent toujours un résultat de huit bits, étant donné que l'octet est la plus petite opérande sur un processeur 80x86. Cependant, vous pouvez facilement utiliser les instructions de décalage et de rotation pour réunir huit valeurs booléennes dans un seul octet (compactage). Les instructions suivantes comparent huit valeurs avec zéro et copient le *zero flag* de chaque comparaison dans les bits correspondants de al :

```
cmp     Val7, 0
setne   al      ;Place la première valeur dans le bit 0
cmp     Val6, 0 ;Teste la valeur pour le bit 6
setne   ah      ;Copie le drapeau zéro dans le registre ah
shr     ah, 1   ;Copie le drapeau zéro dans carry
rcl     al, 1   ;Décale le carry dans l'octet du résultat
cmp     Val5, 0 ;Teste la valeur pour le bit 5
setne   ah
shr     ah, 1
rcl     al, 1
cmp     Val4, 0 ;Teste la valeur pour le bit 4
setne   ah
shr     ah, 1
rcl     al, 1
cmp     Val3, 0 ;Teste la valeur pour le bit 3
setne   ah
shr     ah, 1
rcl     al, 1
cmp     Val2, 0 ;Teste la valeur pour le bit 2
setne   ah
shr     ah, 1
rcl     al, 1
cmp     Val1, 0 ;Teste la valeur pour le bit 1
setne   ah
shr     ah, 1
rcl     al, 1
cmp     Val0, 0 ;Teste la valeur pour le bit 0
setne   ah
shr     ah, 1
rcl     al, 1
```

Maintenant AL contient les drapeaux zéro des huit comparaisons.

6.7 Instructions d'E/S

La famille des processeurs 80x86 supporte deux instructions d'entrées/sorties : in et out[24]. Leur syntaxe est :

```
in      eax/ax/al, port
in      eax/ax/al, dx
```

[24]En réalité, à partir des processeurs 80286 on peut disposer de quatre instructions. Les autres deux seront présentées dans la prochaine section.

```
out     port, eax/ax/al
out     dx, eax/ax/al
```

où *port* est une valeur entre 0 et 255.

La technologie 80x86 peut avoir jusqu'à 65536 ports différents d'E/S (nécessitant une adresse E/S de 16 bits). La valeur du *port* ci-dessus, est cependant d'un octet. Par conséquent, vous pouvez adresser directement seulement les premiers 256 ports d'E/S dans l'espace d'adressage 80x86. Pour adresser la totalité des 65536 ports, il faut charger l'adresse du port désiré dans le registre dx (si cette adresse est supérieure à 255 évidemment), puis y accéder indirectement. L'instruction *in* lit une valeur dans le port spécifié et la copie dans l'accumulateur. L'instruction *out* écrit la valeur de l'accumulateur dans le port d'E/S spécifiée.

Veuillez noter que les instructions in et out n'ont rien de spécial en soi ; elles constituent simplement une autre version de l'instruction mov, accédant à un espace de mémoire différent (l'espace d'adressage des entrées/sorties).

Ces instructions n'affectent aucun flag. Voici des exemples :

```
in      al, 60h         ;Lit un port du clavier

mov     dx, 378h        ;Pointe sur LPT1 : port des données
in      al, dx          ;Lit une valeur du port d'imprimante
inc     ax              ;Incrémente le code ASCII de 1
out     dx, al          ;Ecrit la donnée de al dans le port
                        ;d'imprimante
```

6.8 Instructions de chaînes de caractères

Les processeurs 80x86 supportent douze instructions de chaînes (strings) :

- movs (*move string*) — *déplace une chaîne*
- lods (*load string element into the accumulator*) — *charge chaîne dans al/ax/eax*
- stos (*store accumulator into string element*) — *charge al/ax/eax dans chaîne*
- scas (*scan string and check for match against the value in the accumulateur*) — *cherche la valeur de al/ax/eax dans chaîne*
- cmps (*compare two strings*) — *compare deux chaînes*
- ins (*input a string from I/O port*) — *charge chaîne d'un port d'E/S*
- outs (*output a string to an I/O port*) — *envoie une chaine sur un port d'E/S*
- rep (*repeat a string operation*) — *répète une instruction de chaîne*
- repz (*repeat while zero*) — *répète tant que zéro*
- repe (*repeat while equal*) — *répète tant qu'égal*
- repnz (*repeat while not zero*) — *répète tant que non zéro*
- repne (*repeat while not equal*) — *répète tant que non égal*

On peut utiliser movs, stos, scas, cmps, ins et outs pour manipuler un élément isolé (octet, mot ou double-mot) d'une chaîne ou pour manipuler une chaîne en entier. Généralement, on utilise seulement l'instruction lods pour ne manipuler qu'un élément à la fois.

Ces instructions peuvent fonctionner sur des chaînes d'octets, de mots ou de doubles-mots. Pour spécifier la taille de l'objet, ajouter simplement b, w ou d à la fin du mot-clé de l'instruction, par exemple, lodsb, movsw, cmpsd, etc. Il va de soi que les formes supportant le double-mot sont disponibles seulement sur des processeurs 80386 ou supérieurs.

movs et *cmps* supposent que ds:si contient l'adresse segmentée de la chaîne source et que es:di contient l'adresse segmentée de la chaîne de destination. *lods* attend que ds:si pointe sur la chaîne source et que l'accumulateur sert de destination. *scas* et *stos* s'attendent à que es:di pointe sur la chaîne de destination et que l'accumulateur contient la valeur source.

movs déplace un élément de chaîne (octet, mot ou double-mot) de la valeur de ds:si à la valeur d'es:di. Après avoir effectué le déplacement, l'instruction incrémente ou décrémente les registres *si* et *di* par un, deux ou quatre

selon qu'il s'agisse d'octets, de mots ou de doubles-mots. Le CPU incrémente ces registres si le drapeau de direction est à zéro et il les décrémente s'il vaut 1.

L'instruction movs est utile pour déplacer des blocs de données entre zones de mémoire. Vous pouvez l'utiliser pour déplacer des chaînes, des tableaux et d'autres structures de données multi-octets :

```
movs{b,w,d}:    es:[di] := ds:[si]
                if direction_flag = 0 then
                        si := si + taille;
                        di := di + taille;
                else
                        si := si - taille;
                        di := di - taille;
                endif;
```

Notez : *taille* vaut un pour des octets, deux pour des mots et quatre pour des doubles-mots.

L'instruction cmps compare l'octet, le mot ou le double-mot qui se trouve dans ds:si avec le contenu de es:di et modifie les drapeaux en conséquence. Après la comparaison, cmps incrémente ou décrémente si et di par un, deux ou quatre selon la taille de l'instruction et l'état du flag de direction :

```
cmps{b,w,d}:    cmp ds:[si], es:[di]
                if direction_flag = 0 then
                        si := si + taille;
                        di := di + taille;
                else
                        si := si - taille;
                        di := di - taille;
                endif;
```

L'instruction lods déplace l'octet, le mot ou le double-mot de ds:si dans al, ax ou eax. Puis, elle incrémente ou décrémente le registre si par un, deux ou quatre, selon la taille de l'instruction et l'état du drapeau de direction. Cette instruction est utile pour charger une séquence d'octets, de mots ou de doubles-mots d'un tableau pour effectuer certaines opérations sur ces valeurs et ensuite passer à la valeur suivante de la chaîne[25].

```
lods{b,w,d}:    eax/ax/al := ds:[si]
                if direction_flag = 0 then
                        si := si + taille;
                else
                        si := si - taille;
                endif;
```

L'instruction stos place al, ax ou eax dans l'adresse spécifiée par es:di. Encore, di est incrémenté ou décrémenté selon la taille de l'instruction et la valeur du flag de direction. Cette instruction a plusieurs utilisations. En combinaison avec lods, on peut charger (via lods), manipuler et stocker des éléments de chaîne. Utilisée seule, stos peut rapidement stocker une valeur dans une structure de données à plusieurs octets.

```
stos{b,w,d}:    es:[di] := eax/ax/al
                if direction_flag = 0 then
                        di := di + taille;
                else
                        di := di - taille;
                endif;
```

L'instruction scas compare al, ax ou eax avec la valeur gardée dans es:di, puis elle ajuste di conformément au résultat. Cette instruction modifie les flags de la même façon que cmp et cmps ; elle est parfaite pour rechercher une valeur particulière dans une structure de données à plusieurs octets.

```
scas{b,w,d}:    cmp eax/ax/al, es:[di]
                if direction_flag = 0 then
                        di := di + taille;
                else
```

[25]Notez que *chaîne* et *tableau* sont presque des synonymes, n.d.t.

```
                di := di - taille;
        endif;
```

L'instruction ins accepte un octet, un mot ou un double-mot à partir d'un port d'E/S spécifié par le registre dx. Elle stocke ensuite la valeur d'entrée dans l'emplacement de mémoire es:di et incrémente ou décrémente di de façon appropriée. Cette instruction est disponible seulement sur des processeurs à partir du 80286.

```
ins{b,w,d}:     es:[di] := port(dx)
                if direction_flag = 0 then
                        di := di + taille;
                else
                        di := di - taille;
                endif;
```

L'instruction outs charge l'octet, le mot ou le double-mot à ds:si et le place dans le port d'E/S spécifié par le registre dx. Le registre *si* est incrémenté ou décrémenté selon les mêmes règles que ci-dessus.

```
outs{b,w,d}:    port(dx) := ds:[si]
                if direction_flag = 0 then
                        si := si + taille;
                else
                        si := si - taille;
                endif;
```

Comme on vient de l'expliquer, les instructions de chaînes sont utiles, mais on peut faire encore mieux ! Quand elles sont combinées avec les préfixes rep, repz, repe, repnz et repne, une simple instruction de chaîne peut travailler sur une chaîne tout entière. Pour avoir plus d'informations sur ces préfixes, voir le chapitre dédié aux chaînes.

6.9 Instructions de contrôle de flux

Les instructions vues jusqu'à présent s'exécutent de manière séquentielle ; cela dit, le CPU exécute chacune de ces instructions selon leur ordre d'apparition dans le programme. Mais, pour écrire de vrais programmes, il faut plusieurs structures de contrôle et non seulement une séquence. Comme exemples, vous avez les instructions if, les boucles et les appels de sous-routine (calls). Puisque les compilateurs réduisent tous les codes sources en code assembleur, il n'est pas surprenant que ce dernier comprenne les instructions nécessaires pour pouvoir réaliser toutes ces structures. Les instructions de contrôle de flux 80x86 tombent dans trois catégories : les transferts inconditionnels, les transferts conditionnels et les instructions d'appel et de renvoi des sous-routines. La section suivante décrit ces structures.

6.9.1 Sauts inconditionnels

L'instruction jmp (abbréviation de *jump*) transfère le contrôle inconditionnellement à un autre point du programme. Il y a six formes de cette instruction : un saut intersegment/direct, deux sauts intrasegment/directs, un saut intersegment/indirect et deux sauts intrasegment/indirects. Les sauts *intrasegment* se font toujours à l'intérieur du même segment de code, alors que les autres transfèrent le contrôle dans un autre segment.

Ces instructions utilisent généralement la même syntaxe :

```
jmp     cible
```

L'assembleur les différencie par leurs opérandes :

```
jmp     disp8    ;intrasegment direct (déplacement de 8 bits)
jmp     disp16   ;intrasegment direct (déplacement de 16 bits)
jmp     adrs32   ;intersegment direct (adresse segmentée de 32 bits)
jmp     mem16    ;intrasegment indirect (opérande de mémoire de 16 bits)
jmp     reg16    ;intrasegment indirect (registre)
jmp     mem32    ;intersegment indirect (opérande de mémoire de 32 bits)
```

Intersegment est synonyme de *far*, alors qu'intrasegment est synonyme de *near*.

Les deux sauts intrasegment/directs diffèrent seulement dans leur longueur. La première forme consiste en un opcode et un déplacement de huit bits. Le CPU effectue une extension signée à seize bits et l'additionne au registre ip. Cette instruction peut brancher dans la plage -128..127 (en octets) depuis le début de l'instruction qui suit le saut (c.-à.-d. dans la plage -126..129 autour de l'instruction courante).

La seconde forme de ce saut, a une longueur de trois octets avec un déplacement de deux. Cette instruction permet des sauts dans la plage -32768..32767 octets et peut transférer le contrôle n'importe où dans le segment de code courant. Le CPU additionne simplement le déplacement de deux octets au registre IP.

Ces deux premiers sauts se servent un schéma d'adressage *relatif.* L'offset encodé comme partie de l'opcode *n'est pas* l'adresse cible, mais la distance entre le saut et la cible. Heureusement, MASM calculera cette distance automatiquement pour vous, ce qui vous évite de la calculer manuellement. Dans beaucoup d'aspects, ces instructions ne sont rien de plus qu'un *add ip, disp*.

Le saut intersegment/direct a cinq octets de long ; les quatre derniers octets contiennent une adresse segmentée (l'offset dans le second et le troisième octet, le segment dans le quatrième et cinquième octet). Cette instruction copie l'offset dans le registre ip et le segment dans le registre cs. L'exécution de l'instruction suivante continue à partir de la nouvelle adresse dans cs:ip. Contrairement aux deux instructions de saut précédentes, l'adresse suivant l'opcode est l'adresse absolue de l'instruction cible ; cette version n'utilise pas d'adressage relatif, mais elle charge cs:ip avec une valeur immédiate de 32 bits.

Pour les trois sauts directs décrits ci-dessus, vous spécifiez normalement l'adresse cible à l'aide d'une *étiquette d'instruction* (*statement label*). Une étiquette est généralement un identificateur suivi par un ":", habituellement sur la même ligne qu'une instruction machine exécutable. L'assembleur détermine l'offset de l'instruction après l'étiquette et calcule automatiquement la distance entre l'instruction de saut et l'étiquette d'instruction. Par conséquent, vous n'avez pas à vous préoccuper de déterminer ces distances manuellement. Par exemple, la courte boucle suivante lit continuellement le port parallèle d'imprimante et en inverse le bit le plus bas. Ceci produit un signal électrique (*square wave* ou *onde carrée*) dans l'une des lignes de sortie du port.

```
                mov     dx, 378h        ;Adresse du port d'imprimante
BouclerSansFin: in      al, dx          ;Lire un caractère sur le port d'entrée
                xor     al, 1           ;Inverser le bit le plus bas
                out     dx, al          ;Envoyer la donnée sur le port
                jmp     BoucleSansFin   ;Répéter toujours...
```

La quatrième forme de saut inconditionnel est l'instruction de saut indirect d'intrasegment. Il requiert une opérande de mémoire de 16 bits. Cette forme de saut transfère le contrôle à l'adresse qui se trouve à l'intérieur de l'offset donné par les deux octets de l'opérande de mémoire. Par exemple :

```
WordVar         word    AdresseCible
                  .
                  .
                  .
                jmp     WordVar
```

transfère le contrôle à l'adresse spécifiée par la valeur de l'emplacement de mémoire de 16 bits WordVar. Ceci *ne provoque pas* un saut à l'instruction se situant à l'adresse WordVar, mais saute à l'instruction *dont l'adresse* est gardée dans la variable WordVar. Notez que cette forme de *jump* est équivalente à :

```
                mov     ip, WordVar
```

Quoique l'exemple ci-dessus utilise une seule variable de 16 bits contenant une adresse indirecte, vous pouvez utiliser *tout* mode d'adressage indirect, et pas seulement le *déplacement seul*. Vous pouvez utiliser aussi des modes d'adressage de mémoire indirects, comme dans ce qui suit :

```
                jmp     DispSeul
                jmp     Disp[bx]        ;Disp est un tableau de mots
                jmp     Disp[bx][si]
                jmp     [bx][26]
                etc.
```

[26]Techniquement, ceci est syntaxiquement incorrect parce que MASM ne peut pas déterminer la taille de l'opérande de mémoire. Continuez à lire pour les détails.

Considérez le mode d'adressage indexé ci-dessus (disp[bx]). Ce mode charge le mot de l'emplacement disp+bx et copie la valeur dans le registre ip ; ce qui permet de créer un tableau de pointeurs et de sauter au pointeur spécifié via un index de tableau. Cosidérez l'exemple suivant :

```
TableauAdr      word    instr1, instr2, instr3, instr4
                 .
                 .
                 .
                mov     bx, I           ;Où I est dans la plage 0..3
                add     bx, bx          ;Index pour un tableau de mots
                jmp     TableauAdr[bx]  ;Saute à instr1, instr2, etc, selon I
```

La chose importante à noter ici est que le saut indirect proche charge un mot de la mémoire et le copie dans le registre ip ; il ne saute pas à l'adresse spécifiée, mais le fait indirectement à travers le pointeur de 16 bits à l'emplacement de mémoire spécifique.

Le cinquième type de saut transfère le contrôle à l'offset donné dans un registre général de 16 bits. Notez que vous pouvez utiliser *tout* registre général, et pas seulement bx, si, di, ou bp. Une instruction de la forme

```
                jmp     ax
```

est approximativement équivalente à

```
                mov     ip, ax
```

Notez que les deux formes précédentes (registre ou mémoire indirecte) sont en fait la même instruction. Les champs mod et r/m de l'octet mod-reg-r/m spécifient un registre ou une adresse de mémoire indirecte. Voir l'annexe D pour les détails.

La sixième forme de cette instruction, le saut intersegment/indirect, comprend une opérande de mémoire contenant un pointeur de 32 bits. Le CPU copie le double-mot à cette adresse dans la paire de registres cs:ip. Par exemple,

```
PtrFar          dword   AdresseCible
                 .
                 .
                 .
                jmp     PtrFar
```

transfère le contrôle à l'adresse segmentée spécifiée par les quatre octets à l'adresse PtrFar. Cette instruction est sémantiquement identique à l'instruction (chimérique) :

```
                lcs     ip, PtrFar      ;charge cs:ip depuis PtrFar
```

Comme pour les sauts indirects proches décrits ci-dessus, ce saut vous permet de spécifier tout mode d'adressage mémoire arbitraire (et valide). Vous n'êtes pas limité au déplacement seul qu'on vient d'utiliser.

MASM utilise un mode d'adressage indirect *near* ou *far* selon le type de l'emplacement de mémoire que vous spécifiez. Si la variable est de type word, MASM générera automatiquement un saut *near* et indirect. Si, au contraire, la variable est un dword, alors MASM émettra l'opcode pour un saut indirect et éloigné. Certaines formes de mode d'adressage, malheureusement, ne spécifient pas intrinsèquement une taille. Par exemple, [bx] est définitivement une opérande de mémoire, mais est-ce qu'elle pointe sur un mot ou sur un double-mot ? Elle *peut* pointer sur n'importe quoi. Par conséquent, MASM refusera une instruction comme :

```
                jmp     [bx]
```

MASM ne pourra d'aucune façon déterminer si celui-ci doit être un saut indirect proche ou éloigné. Pour lever l'ambiguïté, on aura besoin d'utiliser un *opérateur de coercition de type*. Le chapitre huit expliquera à fond les opérateurs de coercition ; pour l'instant utilisez simplement l'une des deux instructions suivantes :

```
                jmp     word ptr [bx]
                jmp     dword ptr [bx]
```

Les modes d'adressage indirects des registres ne sont pas les seuls capables de produire de telles ambiguïtés. Vous pourriez tomber dans ce problème même en utilisant les modes d'adressage indexé et basé/indexé :

```
                jmp     word ptr 5[bx]
```

```
                jmp     dword ptr 9[bx][si]
```

En théorie, on pourrait utiliser les instructions de sauts indirects et les instructions set*cc* pour transférer le contrôle de manière conditionnelle à quelque emplacement donné. Par exemple, le code suivant transfère le contrôle à *iftrue* si la variable X de type word est égale à la variable Y de type word. Sinon, il transfère le contrôle à *iffalse.*

```
JmpTbl          word    iffalse, iftrue
                 .
                 .
                 .
                mov     ax, X
                mov     ax, Y
                sete    bl
                movzx   ebx, bl
                jmp     JmpTbl[ebx*2]
```

Comme vous verrez bientôt, il y a un meilleur moyen de le faire: en utilisant les instructions de saut conditionnel.

6.9.2 Les instructions CALL et RET

Les instructions call et ret gèrent l'appel et le retour des sous-routines. Il y a cinq formes d'appels et six de retour.

```
        call    disp16  ;intrasegment direct, 16 bits relatif.
        call    adrs32  ;intrasegment direct, adresse segmentée de 32 bits
        call    mem16   ;intrasegment indirect, pointeur mémoire de 16 bits
        call    reg16   ;intrasegment indirect, pointeur registre de 16 bits
        call    mem32   ;intersegment indirect, pointeur mémoire de 32 bits

        ret             ;retour near ou far
        retn            ;retour near
        retf            ;retour far
        ret     disp    ;retour near ou far et pop
        retn    disp    ;retour near et pop
        retf    disp    ;retour far et pop
```

Les instructions call ont la même forme que les instructions jmp, sauf qu'il n'y a pas d'appel d'intrasegment court (deux octets).

La version far de l'instruction call fait ce qui suit :

- Elle pousse le registre cs dans la pile.
- Elle pousse dans la pile l'offset de 16 bits de l'instruction qui suit l'appel.
- Elle copie l'adresse réelle de 32 bits dans la paire de registres cs:ip. Puisque call permet le même mode d'adressage que jmp, elle peut obtenir l'adresse cible en utilisant un mode d'adressage relatif en mémoire ou dans un registre.
- L'exécution continue à la première instruction de la sous-routine. Cette instruction est l'opcode à l'adresse cible calculé à l'étape précédente.

La version near de l'instruction call fait ce qui suit :

- Elle pousse dans la pile l'offset de l'instruction suivant l'appel.
- Elle copie l'adresse réelle de 16 bits dans le registre ip. Puisque l'instruction call comprend le même mode d'adressage que l'instruction jmp, elle peut obtenir l'adresse cible en utilisant un mode d'adressage relatif en mémoire ou dans un registre.
- L'exécution continue à la première instruction de la sous-routine. Cette instruction est l'opcode de l'adresse cible calculé à l'étape précédente.

La version call $disp_{16}$ utilise un adressage relatif. Vous pouvez obtenir l'adresse effective de la cible en additionnant ce déplacement de 16 bits avec l'adresse de retour (comme l'instruction jmp relative, le déplacement est la distance entre l'instruction qui *suit* l'appel et l'adresse cible).

La version call $adrs_{32}$ utilise le mode d'adressage direct. Une adresse segmentée de 32 bits suit immédiatement l'opcode de call. Cette version copie cette valeur directement dans la paire de registres cs:ip. Sur beaucoup d'aspects, ceci est équivalent au mode d'adressage immédiat, puisque la valeur que cette instruction copie dans cs:ip suit immédiatement l'instruction.

La version call mem_{16} utilise le mode d'adressage indirect. Comme l'instruction jmp, cette forme de call charge le mot à l'emplacement de mémoire spécifié et utilise la valeur de ce mot en tant qu'adresse cible. Souvenez-vous, vous pouvez utiliser *tout* mode d'adressage de mémoire avec cette instruction. Le mode d'adressage de déplacement seul en constitue la forme la plus commune, mais les autres sont tout aussi valides :

```
call    CallTbl[bx]     ;Index d'un tableau de pointeurs
call    word ptr[bx]    ;BX pointe sur le mot à utiliser
call    WordTbl[bx][si] ;etc.
```

Notez que le choix du mode d'adressage affecte uniquement le calcul de l'adresse réelle pour la sous-routine cible. Ces instructions d'appel poussent encore l'offset de l'instruction suivant l'appel. Puisqu'il s'agit d'appels proches (car ils obtiennent leur adresse cible d'un emplacement de mémoire de 16 bits), ils poussent tous l'adresse de retour de 16 bits dans la pile.

Call reg_{16} fonctionne juste comme l'appel indirect ci-dessus, sauf qu'il utilise la valeur de 16 bits dans un registre pour l'adresse cible. Cette version est véritablement équivalente à call mem_{16}. Les deux formes spécifient leur adresse effective en utilisant un octet mod-reg-r/m. Pour la version call reg_{16} les bits de mod contiennent 11b, donc le champ r/m spécifie un registre au lieu d'un mode d'adressage en mémoire. Naturellement, cette instruction pousse aussi l'offset de 16 bits de l'instruction suivante dans la pile comme adresse de retour.

Call mem_{32} est un appel indirect de 32 bits. L'adresse de mémoire spécifiée par cette instruction doit être une valeur de type double word. Cette forme de l'instruction call charge l'adresse segmentée de 32 bits à son adresse physique correspondante et copie la valeur dword dans la paire cs:ip. Elle copie également dans la pile l'adresse segmentée de 32 bits de l'instruction suivante (en poussant d'abord la valeur de segment et ensuite celle de l'offset). Comme l'instruction call mem_{16}, vous pouvez utiliser tout mode d'adressage valide :

```
call    DWordVar
call    DWordTbl[bx]
call    dword ptr[bx]
etc.
```

Synthétiser l'instruction call à l'aide de deux ou trois autres instructions est relativement facile. Vous pouvez réaliser l'équivalent d'un call proche en utilisant une instruction push et une instruction jmp.

```
push    <offset de l'instruction après jmp>
jmp     sous-routine
```

Et un appel far serait similaire, vous auriez juste besoin d'ajouter un push cs avant des deux instructions ci-dessus, de façon à pousser une adresse de retour far dans la pile.

L'instruction ret retourne le contrôle au code appelant de la sous-routine. Elle fait ceci en dépilant l'adresse de retour de la pile et en transférant le contrôle du programme à l'instruction correspondant à cette adresse de retour. Les retours intrasegment (near) enlèvent de la pile une adresse de retour de 16 bits et la placent dans le registre ip, alors que les retours intersegment (far) enlèvent de la pile un offset de 16 bits et le placent dans le registre ip et il placent ensuite une valeur de segment vers le registre cs. Ces instructions équivalent effectivement à :

```
retn:           pop     ip
retf:           popd    cs:ip
```

Evidemment, les appels à des sous-routines *near* doivent correspondre à des retours *near* et les appels à des sous-routines *far* doivent correspondre à des retours *far*. Si vous mélangez les appels *near* avec des retours *far* ou vice-versa, vous laisserez la pile dans un état incohérent et vous *ne retournerez probablement pas* à l'adresse où il se trouvait l'instruction suivante avant l'appel. Bien sûr, une autre question importante sur l'usage des instructions ret et call est qu'il faut s'assurer de ne pas pousser quelque chose dans la pile et ensuite oublier de le désempiler avant de retourner au code appelant. En assembleur, les problèmes de pile sont la cause principale des erreurs dans les sous-routines. Considérez le code suivant :

```
Sous-routine    push    ax
```

```
push    bx
.
.
.
pop     bx
ret
.
.
.
call    Sous-routine
```

L'instruction call pousse l'adresse de retour dans la pile et, ensuite, elle transfère le contrôle à la première instruction de la sous-routine. Les deux premiers push empilent ax et bx afin de préserver leur valeur, parce que la sous-routine les modifiera. Malheureusement, il y a une erreur de programmation, car la sous-routine désempile seulement bx et oublie de le faire avec ax. Cela veut dire que, quand la sous-routine tentera de retourner au code appelant, ce sera la valeur de ax et non l'adresse de retour qui se trouvera au sommet de la pile. Par conséquent, la sous-routine retourne le contrôle à l'adresse spécifiée par la valeur initiale de ax, avec des résultats imprévisibles. Puisque ax peut avoir 65536 valeurs différentes, il y a une probabilité de 1/65536 que la valeur de ax corresponde à la bonne adresse de retour et la chance ici ne joue pas en votre faveur ! Il est plus que probable qu'un tel code plante la machine. Morale de l'histoire : vous devez toujours vous assurer que l'adresse de retour soit dans la tête de la pile avant d'exécuter l'instruction de retour.

Tout comme l'instruction call, c'est très facile de simuler ret à l'aide de deux instructions 80x86. Tout ce dont vous avez besoin est de désempiler l'adresse de retour et de la copier dans le registre ip. Pour des opérations de retour de type *near*, cette opération est très simple, il vous suffira de retirer cette adresse de retour de la pile et de faire un saut indirect à travers ce registre :

```
pop     ax
jmp     ax
```

Simuler un retour far est un peu plus difficile, car vous devez charger cs:ip en une seule opération. La seule instruction capable de le faire (sans compter un ret *far* naturellement) est jmp mem_{32}. Voyez les exercices à la fin de ce chapitre pour plus de détails.

Il y a deux autres versions de l'instruction ret. Elles sont identiques à celles décrites ci-dessus, sauf qu'un déplacement de 16 bits suit leur opcode. Le CPU additionne cette valeur au pointeur de pile immédiatement après avoir désempilé l'adresse de retour. Ce mécanisme supprime les paramètres poussés dans la pile avant de retourner au code appelant. Voir le paragraphe 11.5.9 pour plus de détails.

L'assembleur vous permet de taper ret sans les suffixes "f" ou "n". Si vous le faites, l'assembleur saura s'il s'agit d'un retour far ou near. Regardez le chapitre sur les procédures et les fonctions pour plus de détails.

6.9.3 Les instructions INT, INTO, BOUND et IRET

L'instruction int (*software interrrupt*) est une forme très spéciale d'instruction call. Alors que call appelle des sous-routines à l'intérieur de votre programme, int, pour sa part, appelle les routines du système et d'autres sous-routines spéciales. La différence majeure entre les *routines d'interruption de service* et les procédures standard est que vous pouvez avoir n'importe quel nombre de procédures différentes dans un programme, alors que le système supporte au plus 256 routines de services d'interruption différentes. Un programme appelle une sous-routine en spécifiant l'*adresse* de celle-ci ; alors qu'il appelle une routine d'interruption de service en spécifiant le *numéro d'interruption* particulier de cette routine. Ce chapitre décrira seulement comment appeler une routine d'interruption de service en utilisant les instructions int, into et bound et comment retourner d'une de ces routines en utilisant l'instruction iret.

Il y a quatre versions différentes de l'instruction int. La première est :

```
int     nn
```

(où *nn* est une valeur entre 0 et 255). Elle permet d'appeler une routine d'interruption parmi les 256 disponibles. Cette forme de l'instruction int a une longueur de 2 octets : le premier est l'opcode et le second est la donnée immédiate contenant le numéro d'interruption.

Bien que vous pouvez utiliser l'instruction int pour appeler des procédures (routines d'interruption de service) écrites par vous, le but primaire de cette instruction est de réaliser un *appel système*. Un appel système n'est autre chose qu'un appel de sous-routine fournie par un système comme DOS, PC-BIOS[27], souris ou quelque autre logiciel résidant dans la machine avant le commencement de l'exécution de votre programme. Puisqu'on se réfère toujours à un appel de système via son numéro d'interruption, au lieu de son adresse en mémoire, votre programme n'a donc pas besoin de connaître l'adresse en mémoire de la sous-routine. L'instruction int fournit une *liaison dynamique* à votre programme. Le CPU détermine l'adresse réelle du service d'interruption pendant l'exécution en consultant une *table de vecteurs d'interruption*, ce qui permet aux programmeurs des sous-routines système de changer leur code (y compris le point d'entrée) sans crainte d'affecter aucun programme écrit avant le changement. Tant que le système continuera à utiliser le même numéro d'interruption, le CPU appellera automatiquement le service d'interruption à sa nouvelle adresse.

Le seul problème avec l'instruction int est qu'elle supporte seulement 256 routines d'interruption différentes. Rien que le MS-DOS supporte plus de 100 appels. Le BIOS et les autres systèmes en fournissent des milliers. Ce qui dépasse largement le nombre de toutes les interruptions réservées par Intel. Une solution commune utilisée par la plupart des appels système est d'utiliser un numéro unique d'interruption pour une classe donnée d'appels et ensuite passer un numéro de fonction dans l'un des registres 80x86 (en général, le registre ah). Par exemple, MS-DOS utilise un seul numéro d'interruption , 21h. Pour choisir une fonction particulière du DOS, vous chargez un *code de fonction DOS* dans le registre ah avant d'exécuter l'instruction int 21h. Par exemple, pour mettre fin à un programme et retourner le contrôle à MS-DOS, on charge ah avec 4Ch et on appele le DOS avec l'instruction int 21h :

```
mov     ah, 4ch         ;Opcode de fin du DOS
int     21h             ;Appel au DOS
```

L'interruption de clavier du BIOS est un autre bon exemple. *int 16h* est responsable de tester le clavier et de lire la donnée depuis celui-ci. La routine du BIOS fournit plusieurs appels pour lire un caractère et capturer du code depuis le clavier, pour voir si des caractères sont disponibles dans le tampon système de frappe (type ahead buffer), vérifier l'état des flags du clavier et ainsi de suite. Pour choisir une opération spécifique, vous chargez le numéro de fonction dans le registre ah avant d'exécuter int 16h. Le tableau qui suit, énumère les fonctions possibles :

Table 31 : fonctions du clavier supportées par le BIOS

Numéro de fonction (AH)	Paramètres d'entrée	Paramètres de sortie	Description
0		al - caractère ASCII ah - code capture de frappe (scan code)	Lit un caractère. Lit le caractère suivant disponible du tampon système de frappe. Attend une frappe (keystroke[28]) si le tampon est vide.
1		ZF - activé si pas de frappe disponible. ZF - désactivé si frappe disponible. AL : code ASCII AH : code capture de frappe	Vérifie si un caractère est disponible dans le tampon de caractères. Active le flag zéro s'il n'y a pas de touche disponible. S'il y en a une, alors la fonction retourne le code ASCII et le code de capture dans ax. Cette valeur de ax reste indéfinie s'il n'y a pas de touche disponible.
2		AL - shift flags	Retourne dans al, l'état courant des shift flags. Ces flags sont définis comme suit : bit 7 : état de Insert bit 6 : état de Capslock bit 5 : état de Numlock bit 4 : état de Scroll lock bit 3 : alt pressé

[27]BIOS est un acronyme pour "Basic Input/Output System".

[28]Notez qu'il y a une différence entre un caractère et un keystroke (*frappe*, littéralement) ; un caractère est simplement un code ASCII, alors qu'un keystroke n'est autre chose que le code numérique de chaque touche du clavier de l'ordinateur, n.d.t.

			bit 2 : ctrl pressé bit 1 : shift gauche pressé bit 0 : shift droit pressé
3	al = 5 bh = 0, 1, 2, 3 pour un délai de ¼, ½, ¾ ou 1 seconde. bl=0..1Fh pour 30/sec à 2/sec.		Définit un débit de répétition automatique[29]. Le registre bh contient le temps écoulé avant la frappe d'une touche et l'opération de répétition automatique, alors que le registre bl contient le débit de répétition automatique.
5	ch = code de capture frappe cl = code ASCII		Garde le code d'une touche dans le tampon. Cette fonction garde cette valeur dans le registre cx à la fin du tampon. Notez que le code de capture de frappe dans ch n'a pas à correspondre au code ASCII qui apparaît dans cl. Cette routine insérera simplement la donnée que vous fournissez dans le tampon système de frappe.
10h		al - caractère ASCII ah - code de capture frappe	Lit un caractère étendu. Comme l'appel ah=0, sauf que celui-ci passe tous les codes de touches, alors que l'appel ah=0[30] rejette les codes qui ne sont pas compatibles avec le PC/XT.
11h		ZF - activé si pas de touche ZF - désactivé si touche disponible al - code ASCII ah - code de capture frappe	Semblable à l'appel ah = 01h, mais ah=11h ne rejette pas les codes qui ne sont pas compatibles avec PC/XT (c'est-à-dire les touches additionnelles trouvées dans un clavier à 101 touches).
12h		al : shift flags ah : shift flags étendus	Retourne l'état courant des shift flags dans ax. Ils sont définis comme suit : bit 15 : frappé SysReq bit 14 : Capslock actuellement pressé bit 13 : Numlock actuellement pressé bit 12 : Scroll lock actuellement pressé bit 11 : Alt droit pressé bit 10 : Ctrl droit pressé bit 9 : Alt gauche pressé bit 8 : Ctrl gauche pressé bit 7 : état de Insert bit 6 : état de Capslock bit 5 : état de Numlock bit 4 : état de Scroll lock bit 3 : un des deux Alt pressés bit 2 : un des deux Ctrl pressés bit 1 : Shift gauche pressé bit 0 : Shift droit pressé[31]

Par exemple, pour lire un caractère du tampon système de frappe et laisser le code ASCII dans al, vous pouvez utiliser le code suivant :

```
mov     ah, 0           ;attendre pour une touche disponible,
int     16h             ;puis lire cette touche
```

[29]C.-à.-d. le débit de répétition d'affichage d'un caractère quand on maintient une touche pressée, n.d.t.
[30]Il y avait ah=0 dans l'original, mais il s'agit d'une erreur, n.d.t.
[31]Noter que ces codes sont légèrement différents par rapport à ceux de l'appel ah=2, car il s'agit d'une fonction différente, n.d.t.

```
mov     caractere, al   ;Enregistrer le caractère lu
```

De même, si vous désirez tester le tampon système de frappe pour voir si une touche est disponible, *mais sans lire le keystroke (code de frappe)*, vous pouvez utiliser le code suivant :

```
mov     ah, 1           ;Tester si une touche est disponible.
int     16h             ;Activer le zéro flag si cette touche
                        ;n'est pas disponible.
```

Pour plus d'informations à propos du BIOS et du MS-DOS voir le début du chapitre 13.

La seconde forme de l'instruction int constitue un cas spécial :

```
int     3
```

int 3 est une forme spéciale d'instruction d'interruption qui a seulement un octet de taille. CodeView et d'autres débogueurs l'utilisent comme instruction de point d'arrêt. Chaque fois que vous voyez un point d'arrêt sur une instruction de votre programme, le débogueur est en train de remplacer le premier octet de l'opcode de l'instruction par l'instruction int 3. Quand votre programme exécute int 3, il effectue un "appel système" au débogueur, de façon que celui-ci peut reprendre le contrôle du CPU. Quand cela arrive, le débogueur est en train de remplacer l'instruction int 3 par le code original.

En travaillant avec un débogueur, on peut explicitement utiliser l'instruction int 3 pour arrêter l'exécution du programme et retourner le contrôle au débogueur. *Il ne s'agit pas pourtant de la façon habituelle de terminer un programme*. Si vous tentez d'exécuter int 3 dans le DOS, au lieu de le faire sous le contrôle d'un débogueur, vous avez de grandes chances de planter le système.

La troisième forme de l'instruction int est into. Cette instruction provoque un point d'arrêt dans le programme seulement si le drapeau overflow est activé. Vous pouvez vous en servir pour tester rapidement un dépassement de capacité arithmétique après avoir effectué une opération arithmétique. Sémantiquement, cette instruction est équivalente à :

```
if overflow = 1 then int 4
```

Il ne faudrait pas utiliser cette instruction sans lui avoir fourni un traitement d'interruption, car on risque encore une fois de planter le système.

La quatrième interruption logicielle, fournie par des processeurs 80286 et supérieurs est l'instruction bound. Cette instruction a la syntaxe :

```
bound   reg, mem
```

et exécute l'algorithme suivant :

```
if (reg < [mem]) or (reg > [mem+sizeof(reg)]) then int 5
```

mem signifie que le contenu d'un emplacement de mémoire et sizeof(reg) vaut deux ou quatre selon que le registre est de 16 ou 32 bits. L'opérande de mémoire doit avoir le double de la taille de l'opérande de registre. Cette instruction en compare les valeurs via une comparaison entière *signée*.

Les concepteurs d'Intel l'ont ajoutée pour permettre une vérification rapide de la plage d'une valeur dans un registre. Ceci est utile pour Pascal, par exemple, qui vérifie la validité des limites des tableaux et également si un entier donné (aka un index) fait partie de cette plage.

Cependant, il y a deux problèmes avec l'instruction bound : dans des processeurs 80486 et Pentium/586, elle est généralement plus lente que la séquence d'instructions qu'elle devrait remplacer[32] :

```
cmp     reg, limInferieure
jl      horsDesLimites
cmp     reg, limSuperieure
jg      horsDesLimites
```

Sur les pouces 80486 et Pentium/586, la séquence qu'on vient de voir requiert seulement qutre cycles d'horloge, en supposant que vous pouvez utiliser le mode d'adressage direct et qu'aucun branchement n'ait lieu[33]

[32]La prochaine section décrit les instructions **jg** et **jl**.
[33]Généralement on s'attend que les cas de violations de limites des tableaux sont très rares.

;l'instruction bound requiert 7-8 cycles d'orloge dans des circonstances semblables et en supposant aussi que les opérandes de mémoire sont dans le cache.

Le second problème de cette instruction est qu'elle exécute une interruption int 5 si une violation de limites se produit. IBM, dans sa sagesse infinie, a décidé d'utiliser le traitement de routine d'interruption int 5 pour faire une capture d'écran. Donc, si vous exécutez une instruction bound et que la valeur est hors des limites du tableau, le système imprimera, par défaut, une copie de l'écran via l'imprimante. Si vous remplacez le traitement par défaut int 5 par un de vos propres traitements, le fait de presser la touche PrtSc transférera le contrôle à votre traitement bound. Bien qu'il y a des moyens de contourner ce problème, beaucoup de gens ne se préoccupent pas de bound, compte tenu aussi de sa lenteur.

Toute exécution d'une instruction int quelconque déclenche les événements suivants :

- Le système empile le registre flag.
- Puis, il empile aussi *cs* et *ip*.
- Puis, il utilise le numéro d'interruption (l'instruction into est l'interruption n.4 et bound est l'interruption n.5), multiplié par quatre en tant qu'index dans la table des vecteurs d'interruption, ensuite, il copie le double-mot correspondant à cet index de la table dans cs:ip.

L'instruction int diffère d'une instruction call en deux points majeurs. Avant tout, les instructions d'appel ont des longueurs allant de deux à six octets, alors que les instructions int ont généralement une longueur de deux octets (les exceptions à cette règle sont int 3, into et bound). Ensuite, et surtout, elles placent les flags et les adresses de retour dans la pile alors qu'une instruction call pousse uniquement l'adresse de retour. Notez aussi que les instructions int empilent toujours une adresse de retour de type *far* (c.-à.-d. une valeur cs et un offset dans le segment de code), alors que seul un *call* de type *far* empile une adresse de retour de 32 bits.

Puisque *int* place le registre *flags* dans la pile, il faut utiliser une instruction de retour spéciale : iret (*interrupt return*), pour revenir (return) d'une routine appelée via des instructions int. Si vous revenez d'une procédure d'interruption via l'instruction ret, *flags* sera laissé dans la pile lors du retour à l'appelant. L'instruction iret est équivalente aux deux séquences ret et popf (en présumant, bien sûr, que vous exécutez popf avant de retourner le contrôle à l'adresse pointée par le double-mot au sommet de la pile).

Les instructions int nettoient le drapeau de trace (T) du registre *flags*. Elles n'affectent aucun autre drapeau. L'instruction iret, par sa propre nature, peut affecter tous les drapeaux, puisqu'elle les dépile.

6.9.4 Sauts conditionnels

Bien que les instructions jmp, call et ret permettent le transfert de contrôle, elles ne sont pas suffisantes pour couvrir tous les cas de décisions possibles. Les instructions de saut conditionnel complètent le cadre. Les sauts conditionnels sont l'outil de base pour créer des boucles et d'autres instructions d'exécution conditionnelle comme if..then.

Ces sauts testent un ou plusieurs drapeaux du registre *flags* pour voir s'ils répondent à un état particulier (tout comme les instructions set*cc*). Si celui-ci correspond, le contrôle est transféré à l'emplacement ciblé. S'il échoue, le CPU ignore le saut conditionnel et l'exécution continuera avec la prochaine instruction après le saut. Certaines instructions peuvent tester des conditions pour les drapeaux *sign*, *carry*, *overflow* et *zero*. Par exemple, après l'exécution d'une instruction de décalage à gauche, vous pouvez interroger le drapeau *carry* pour déterminer si le bit débordant était 1. De même, vous pouvez tester la condition du drapeau zéro après une instruction de test pour vérifier si des bits spécifiques étaient à 1. La plupart du temps, cependant, vous exécuterez un saut conditionnel après une instruction de comparaison. *cmp* modifie les flags de manière à pouvoir tester les conditions "plus petit que", "plus grand que", "égal", etc.

Note : la documentation Intel définit divers synonymes ou alias pour diverses instructions de saut conditionnel. Les tables suivantes montrent tous les alias pour une instruction particulière. Ces tables montrent aussi le cas de branchement opposé. Vous en verrez bientôt l'utilité.

Table 32 : instructions J*cc* qui testent les flags

instruction	Description	Condition	Alias	Opposé
JC	Jump if carry	Carry=1	JB, JNAE	JNC

JNC	Jump if not carry	Carry=0	JNB, JAE	JC
JZ	Jump if zero	Zero=1	JE	JNZ
JNZ	Jump if not zero	Zero=0	JNE	JZ
JS	Jump if sign	Sign=1		JNS
JNS	Jump if no sign	Sign=0		JS
JO	Jump if overflow	Overflow=1		JNO
JNO	Jump if no overflow	Overflow=0		JO
JP	Jump if parity	Parity=1	JPE	JNP
JPE	Jump if parity even	Parity=1	JP	JPO
JNP	Jump if no parity	Parity=0	JPO	JP
JPO	Jump if parity odd	Parity=0	JNP	JPE

Table 33 : instructions J*cc* pour comparaisons non signées

instruction	Description	Condition	Alias	Opposé
JA	Jump if above (>)	Carry=0 Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0 Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry=0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry=0	JNC, JAE	JB
JB	Jump if below (<)	Carry=1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry=1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry=1 ou Zero=1	JNA	JNBE
JNA	Jump if not above (not >)	Carry=1 ou Zero=1	JBE	JA
JE	Jump if equal (=)	Zero=1	JZ	JNE
JNE	Jump if not equal (≠)	Zero=0	JNZ	JE

Table 34: instructions J*cc* pour comparaisons signées

instruction	Description	Condition	Alias	Opposé
JG	Jump if greater (>)	Sign=Overflow ou Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign=Overflow ou Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign=Overflow	JNL	JGE
JNL	Jump if not less than (not <)	Sign=Overflow	JGE	JL
JL	Jump if less than (<)	Sign≠Overflow	JNGE	JNL
JNGE	Jump if not greater than or equal (not >=)	Sign≠Overflow	JL	JGE
JLE	Jump if less than or equal (<=)	Sign≠Overflow ou Zero=1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign≠Overflow ou Zero=1	JLE	JG
JE	Jump if equal (=)	Zero=1	JZ	JNE
JNE	Jump if not equal (≠)	Zero=0	JNZ	JE

Sur des processeurs 80286 et antérieurs toutes ces instructions ont deux octets de long. Le premier des deux octets est un opcode et le second est un déplacement d'un octet. Bien que cela donne du code compact, toutefois un déplacement d'un octet représente seulement une plage de ±128 bits. Il y a un truc simple pour détourner cette limitation sur des processeurs anciens :

- Indépendamment du jump utilisé, passez à la version opposée (voir les tables ci-dessus).

- Une fois que vous avez choisi le branchement opposé, si la condition opposée est fausse alors, faites passer le contrôle sur un jmp que, lui, branche sur la cible éloignée.

Par exemple, pour convertir :

```
                jc      Cible
```

en forme longue, utilisez la séquence suivante d'instructions :

```
                jnc     sauteSaut       ;Si la condition opposée est fausse
                jmp     Cible           ;alors sauter inconditionnellement
nePasSauter:    <autres instructions>   ;sinon, tout suit normalement
```

Si le drapeau carry est à zéro (NC=no carry), le contrôle est transféré à l'étiquette nePasSauter au même point où vous seriez si vous utilisiez jc Cible. Si le flag carry vaut 1 lors de cette séquence, alors le contrôle est passé à l'instruction jmp, qui exécute un saut sur Cible. Puisque l'instruction jmp permet aussi bien un déplacement de 16 bits que des opérandes far, à travers de cette astuce vous pouvez sauter n'importe où dans la mémoire.

Un bref commentaire sur la colonne "Opposé" des tables ci-dessus est peut-être nécessaire. Comme mentionné plus tôt, quand vous avez besoin d'effectuer des branchements dépassant la plage -128..+128 bits, vous pouvez savoir tout de suite quel est le branchement opposé en consultant les tables. Vous pouvez également constater que diverses instructions de branchement ont des synonymes, ce qui veut dire qu'il y en aura également pour des instructions de sauts opposés. Mais, *n'utilisez pas n'importe quel alias en étendant un saut pour une branche hors de la plage ±128*. A deux exceptions près, une règle très simple peut décrire complètement comment générer un branchement opposé est celle-ci :

- Si la seconde lettre d'une instruction j*cc* n'est pas un "n", alors insérez un "n" après le "j". Par exemple, *je* devient *jne* et *jl* devient *jnl*.
- Si la seconde lettre d'une instruction j*cc* est un "n", alors elevez-le et vous obtiendrez le branchement contraire. Par exemple, jng devinent jg et jne devient je.

Les deux exceptions à cette règle sont jpe (jump pariry even) et jpo (jump parity odd). Mais ces exceptions ne causent pas beaucoup d'ennui, car : a) vous ne testez le drapeau de parité que rarement et b) vous pouvez parfaitement utiliser les équivalents jp et jnp au lieu de jpe et jpo. Et la règle "N ou Non N" s'applique parfaitement à jp et jnp.

Bien que vous savez que jge est l'opposé de jl, prenez l'habitude d'utiliser jnl au lieu de jge. Il est trop facile, dans un contexte important, de se dire que "greater est l'opposé de less" et de substituer jg au lieu de jge. Vous pouvez éviter cette confusion en utilisant toujours la règle "N/Pas de N".

MASM 6.x et d'autres assembleurs modernes de type 80x86 convertissent automatiquement des branchements hors plage en la séquence du branchement opposé. Il y a une option qui permet de désactiver cette caractéristique. En effet, lors de situations critiques de performance sur des processeurs 80286 ou inférieurs, vous voudrez désactiver cette option pour ajuster vous même les branchements hors plage manuellement. La raison en est très simple : cette automatisation vide *toujours* la queue de préchargement quelle que soit la condition qui est remplie, puisque le CPU saute de toutes façons. Ce qui rend inutile le pipeline. Une bonne utilisation des branchements permet de ne pas vider le pipeline ou la queue de préchargement si on ne prend pas le saut. Si une condition se produit beaucoup plus souvent que l'inverse, on préfèrera utiliser le saut conditionnel vers un jmp proche, pour continuer à boucler comme avant. Par exemple, si vous avez une instruction "je cible" et que *cible* est hors de la plage -128..+128 par rapport au saut, alors, au lieur d'utiliser "je cible", vous feriez mieux de convertir ce code en:

```
                je      vaALaCible
                .
                .
                .
vaALaCible:     jmp     cible
```

Bien que maintenant un branchement sur cible requiert deux sauts, c'est beaucoup plus efficace que la conversion standard si le drapeau zéro vaut régulièrement zéro lors de l'exécution de l'instruction je.

Les processeurs 80386 et ultérieurs fournissent une version étendue des sauts conditionnels qui fait quatre octets de long (les deux derniers représentant un déplacement de 16 bits). Ces sauts conditionnels permettent de transférer le contrôle n'importe où dans le segment de code courant. Par conséquent il n'y a pas à se

préoccuper d'appliquer la conversion manuelle pour étendre la plage du saut. Si vous avez indiqué à MASM que vous êtes en train d'utiliser un processeur 80386 ou supérieur, alors la version de quatre octets des instructions de saut sera automatiquement choisie. Regardez le chapitre huit pour apprendre comment le faire.

Les sauts conditionnels 80x86 vous permettent de diviser le flux des programmes en un ou deux chemins selon des conditions logiques. Supposez que vous vouliez incrémenter le registre *ax,* si *bx* et *cx* sont égaux. Vous pourriez faire ceci via le code suivant :

```
                cmp     bx, cx
                jne     enjamberInc
                inc     ax
enjamberInc:    <autres instructions>
```

Ici, il s'agit d'utiliser le branchement *opposé* pour sauter par-dessus les instructions que vous utiliseriez si la condition était vraie. Utilisez toujours la règle du"N / Non N" pour choisir un branchement opposé. Vous pouvez commettre ici la même erreur en choisissant une branche opposée qu'en étendant un saut hors plage.

Vous pouvez également vous servir des instructions de sauts conditionnels pour simuler des boucles plus concises. Par exemple, la séquence suivante lit une suite de caractères de l'utilisateur et stocke chaque caractère dans des éléments successifs d'un tableau, jusqu'à ce que l'utilisateur appuie sur Entrée (*carriage return*) :

```
                mov     di, 0
LireCarBcl:     mov     ah, 0           ;INT 16 lit l'opcode des touches.
                int     16h
                mov     Input[di], al
                inc     di
                cmp     al, 0dh         ;Le code ASCII du retour à la ligne
                jne     LireCarBcl
                mov     Input[di-1], 0  ;Remplacez le carriage return avec 0
```

Pour plus d'informations à propos de l'usage des sauts conditionnels pour synthétiser des instructions IF, des boucles et d'autres structures de contrôle, consultez le chapitre 10.

Tout comme les instructions set*cc*, les sauts conditionnels se présentent sous deux catégories de base : les instructions qui testent certaines valeurs de flags (comme jz, jc, jno) et celles qui vérifient des conditions (plus petit que, plus grand que, etc.). Un saut conditionnel lorsqu'il teste une condition, fait suite presque toujours à une instruction cmp ; celle-ci modifie les flags, donc on peut utiliser des instructions comme ja, jae, jb, jbe, je ou jne pour des tests non signés tels que $<$, $\leq$, $=$, $\neq$, $>$, $\geq$. En même temps, cmp modifie les flags de façon à vous permettre d'effectuer également des comparaisons signées via les instructions jl, jle, je, jne, jg et jge.

Les instructions de saut conditionnel testent uniquement les drapeaux, elles n'affectent aucun de ces derniers.

6.9.5 Les instructions JCXZ/JECXZ

L'instruction jcxz (*jump if cx is zero*) saute à l'adresse cible si cx contient zéro. Bien qu'on peut l'utiliser toutes les fois qu'on a besoin de tester si cx vaut 0, cette instruction est généralement utilisée avant des boucles construites avec des instructions de saut. L'instruction loop peut répéter une séquence d'opérations cx fois. Si cx vaut zéro, loop répétera son opération 65536 fois. Vous pouvez utiliser jcxz pour sortir de cette boucle si cx vaut zéro.

L'instruction jecxz, disponible seulement sur des processeurs 80386 ou ultérieurs fait exactement la même chose que jcxz, mais cette elle teste le registre ecx en entier. Notez que jcxz, testera seulement et toujours cx, même si votre processeur est un 386 en mode 32 bits.

Il n'y a pas de versions opposées de jcxz et jecxz. Par conséquent, vous ne pourrez pas utiliser la règle "n/ non n" pour élargir la portée de jcxz et jecxz. La manière la plus facile pour résoudre ce problème est de diviser l'instruction en deux instructions effectuant la même tâche :

```
                jcxz    Cible
```

devient :

```
                test    cx, cx          ;Active le flag zero si cx = 0
```

```
                je      Cible
```

Maintenant, vous pouvez facilement élargir l'instruction je à l'aide des techniques décrites dans la section précédente.

L'instruction test ci-dessus activera le drapeau zéro si, et seulement si cx contient zéro. Après tout, s'il y avait n'importe quel bit différent de zéro dans cx, effectuer un AND sur ces bits produirait un résultat différent de zéro. Il s'agit d'une façon efficiente de voir si un registre de 16 ou de 32 bits est nul. En fait, ces deux instructions sont plus rapides que l'instruction jcxz sur un processeur 80486 ou ultérieur. Intel recommande définitivement d'utiliser cette séquence au lieu de jcxz, si vous avez besoin de plus de vitesse. Sans doute jcxz est plus courte que ces deux instructions, mais elle n'est pas plus rapide. Et c'est justement un bon exemple d'exception à la règle « plus court c'est souvent plus rapide ».

L'instruction jcxz n'affecte aucun flag.

6.9.6 L'instruction LOOP

Cette instruction décrémente le registre cx, puis saute à l'emplacement indiqué si le registre cx n'est pas nul. Étant donné que cette instruction décrémente un registre et vérifie sa valeur, le fait de donner à cx une valeur initiale de zéro correspond à répéter la boucle 65536 fois. Si vous ne désirez pas ce résultat, utilisez jcxz pour sortir de la boucle.

L'instruction loop n'a pas de forme "opposée" et comme les instructions jcxz/jecxz, elle est limitée sur tous les processeurs à une plage de ±128 bits. Si vous voulez élargir cette plage, vous devrez utiliser deux instructions à la place:

```
;"loop étiquette" devient
                dec     cx
                jne     etiquette
```

Vous pouvez adapter facilement le jne pour n'importe quelle distance.

Il n'y a pas d'instruction eloop qui boucle tant que ecx est différent de zéro (il y a l'instruction loope, mais elle fait quelque chose de totalement différent), et la raison est simple : à partir du 80386 les concepteurs d'Intel ont décidé d'arrêter d'offrir le support pour l'instruction loop. Certes, elle est toujours là pour assurer la compatibilité avec d'anciens programmes, mais il s'avère que les instructions dec/jne sont effectivement *plus rapides* sur des processeurs de 32 bits. Ce résultat étrange est dû à des problèmes dans le décodage de loop et le fonctionnement du pipeline.

Bien que le nom de l'instruction loop suggère la création de boucles, ce que cette instruction fait réellement est décrémenter cx et sauter à l'adresse cible tant que cx est différent de zéro après chaque décrément. Vous pouvez utiliser cette instruction partout où vous voulez produire cet effet et pas seulement en créant des boucles. Cependant, elle est très utile pour faire répéter une séquence d'instructions un certain nombre de fois. Par exemple, la boucle suivante initialise les 256 éléments d'un tableau d'octets avec des valeurs croissantes :

```
                mov     ecx, 255
TabBoucle:      mov     Tab[ecx], cl
                loop    TabBoucle
                mov     Tab[0], 0
```

La dernière instruction est nécessaire parce que la boucle ne continue pas son cycle quand cx est égal à zéro. Par conséquent, le dernier élément du tableau que cette boucle traite est Tab[1], d'où la dernière instruction.

L'instruction loop n'affecte aucun drapeau.

6.9.7 L'instruction LOOPE/LOOPZ

Loope/loopz (*loop while equal / loop while zero*, qui sont synonymes l'un de l'autre) saute à l'adresse ciblée si le registre cx est différent de zéro et le *zero flag* est activé. Ce qui est très utile après une instruction cmp ou cmps , en plus, c'est un peu plus rapide que les instructions 80386/486 correspondantes, *si vous utilisez toutes les fonctionalités de l'instruction*. Cependant, elle fait des ravages avec le pipeline et le fonctionnement

superscalaire du Pentium, donc vous devriez probablement lui préférer des instructions décomposées. Voici ce qu'elle fait :

```
cx := cx - 1
if ZeroFlag = 1 and cx ≠ 0, goto target
```

Elle échoue si l'une des deux situations surgit : soit le drapeau zéro est désactivé, soit cx a atteint zéro. En testant le drapeau zéro après l'instruction (avec je ou jne, par exemple), vous pouvez déterminer la cause de la terminaison.

L'instruction est utile aussi quand vous voulez répéter une boucle tant qu'une certaine valeur est égale à une autre, mais qu'il y a un nombre maximal d'itérations que vous voulez permettre. Par exemple, la boucle suivante lit les valeurs d'un tableau pour chercher le premier octet différent de zéro, mais elle ne continue pas à chercher après la fin du tableau :

```
                mov     cx, 16  ;Au plus 16 éléments
                mov     bx, -1  ;Index du tableau (notez le inc qui suit)
Chercher:       inc     bx      ;Passer à l'élément suivant
                cmp     Tab[bx], 0      ;Voir si l'élément est égal à zéro
                loope   Chercher        ;Répéter si c'est le cas
                je      ToutZeros       ;Sauter si tous les éléments valent 0
```

Notez que cette instruction *n'est pas* l'opposé de loopnz/loopne, donc, vous ne pouvez pas l'utiliser pour étendre un saut hors plage. Si vous avez besoin de sauter au-delà des ±128 bits, il vous faudra des instructions alternatives. Par exemple :

```
                jne     sortie
                dec     cx
                je      sortie2
                jmp     cible
sortie:         dec     cx      ;loope décrémente cx, même si ZF=0
sortie2:
```

L'instruction loope/loopz n'affecte aucun flag.

6.9.8 L'instruction LOOPNE/LOOPNZ

Cette instruction fonctionne de manière analogue à loope/loopz, mais loopne/loopnz (loop while not equal/not zero) répète tant que cx est différent de zéro et (ici vient la différence) et le drapeau zéro reste désactivé. L'algorithme est :

```
cx := cx - 1
if ZeroFlag = 0 and cx ≠ 0, goto target
```

Vous pouvez déterminer si loopne a fini son exécution en testant la valeur du drapeau zéro. Si le *zero flag* est désactivé, la boucle loopne finit son exécution parce que la décrémentation du registre cx a atteint zéro, sinon, elle termine parce que le zero flag a été activé.

Cette instruction *n'est pas* l'opposé de loope/loopz. Si l'adresse cible est hors de portée, vous aurez besoin d'utiliser des instructions alternatives, comme celles qui suivent :

```
                je      sortie
                dec     cx
                je      sortie2
                jmp     cible
sortie:         dec     cx      ;loope décrémente cx, même si ZF=1
sortie2:
```

Vous pouvez vous servir de loopne pour boucler un nombre maximal de fois pendant qu'on attend qu'une condition devienne vraie. Par exemple, vous pouvez parcourir un tableau tant qu'il contient des éléments ou qu'un certain octet n'est pas trouvé à l'intérieur de ce tableau. Par exemple :

```
                mov     cx, 16          ;Le nombre maximal est 16
                mov     bx, -1          ;Index du tableau
```

```
RepTantQueNonZ: inc     bx              ;Passer à l'élément suivant
                cmp     Tab[bx], 0      ;Cet élément contient zéro ?
                loopne  RepTantQueNonZ  ;Sortir si c'est le cas ou bien bx > 16
```

Bien que loope/loopz et loopne/loopnz sont plus lentes que les instructions individuelles qui pourraient les remplacer, il y a un usage spécifique de ces instructions dans les contexte où la vitesse n'est pas un facteur particulièrement important, à savoir les boucles d'attente pendant les opérations d'entrées/sorties. Supposez que le bit 7 du port 379h contient 1 si le périphérique est occupé et 0 s'il est libre. S'il fallait écrire une donnée sur ce port, on pourrait utiliser :

```
                mov     dx, 379h
AttendreLibre:  in      al, dx          ;Lire le port
                test    al, 80h         ;Tester si le bit 7 est à 1
                jne     AttendreLibre   ;Si occupé, retenter
```

Le seul problème avec ce code est le risque d'une boucle infinie. Dans un système réel, un câble peut être débranché, quelqu'un peut éteindre le périphérique et tout autre incident peut survenir, ce qui provoquerait une boucle sans fin et bloquerait l'ordinateur. Des pilotes ou logiciels robustes ajoutent généralemenent une minuterie à des boucles comme celles-ci, de sorte que si un périphérique ne répond plus après un certain temps la boucle s'arrête et signale une erreur. Le code suivant réalisera ça :

```
                mov     dx, 379h        ;Entrer l'adresse du port
                mov     cx, 0           ;Quitter après 65536 itérations
AttendreLibre:  in      al, dx          ;Lire le port
                test    al, 80h         ;Tester si le bit 7 est à 1
                loopne  AttendreLibre   ;Répéter si occupé et temps non échoué
                jne     TempsExpire     ;Sortir si cx=0, parce qu'expiré
```

Vous auriez pu utiliser loope/loopz si le test du bit concernait 0 au lieu de 1.

Les instructions loopne/loopnz n'affectent aucun flag.

6.10 Instructions diverses

Dans la famille 80x86, il y a plusieurs instructions qui ne tombent dans aucune des catégories décrites dans ce chapitre. Généralement, il s'agit d'instructions qui manipulent des drapeaux individuels, qui fournissent des services spéciaux du processeur ou traitent des opérations en mode privilégié.

Les instructions qui manipulent directement les drapeaux du registre flags sont :

- clc Désactive (Clear) Carry Flag
- stc Active (Set) Carry Flag
- cmc Complémente Carry Flag
- cld Désactive Direction Flag
- std Active Direction Flag
- cli Désactive I*nterrupt Enable/Disable Flag*
- sti Active *Interrupt Enable/Disable Flag*

Notez : soyez prudent en utilisant cli dans vos programmes, car un usage impropre pourrait planter votre machine et vous obliger á redémarrer.

L'instruction **nop** ne fait rien d'autre que gaspiller quelques cycles d'horloge et occuper un octet de mémoire. Les programmeurs l'utilisent surtout à des fins de débogage ou pour réserver un espace. Cette instruction n'est pas unique, c'est simplement un synonyme de xchg ax, ax.

L'instruction **hlt** arrête le processeur jusqu'au prochain reset ou jusqu'à la production d'une interruption non-masquable ou une interruption spécifique (en présumant que les interruptions sont activées). Généralement, il est déconseillé d'utiliser cette instruction sur un IBM PC, à moins que vous sachiez parfaitement ce que vous faites. *Cette instruction n'est pas l'équivalent de l'instruction* halt *qu'on a vu au chapitre des instructions factices x86. Il ne faut pas l'utiliser pour arrêter des programmes.*

Le 80x86 fournit une autre instruction préfixe, lock, qui, comme l'instruction rep, affecte l'instruction suivante. Cependant, cette instruction a peu de sens sur la plupart des systèmes PC. Sa fonction est de coordonner des

ordinateurs ayant des CPU multiples. Au fur et à mesure que de tels systèmes deviennent disponibles, cette instruction finira par être utile[34]. Vous n'avez pas besoin de vous en préoccuper ici.

Le Pentium fournit deux autres instructions qui peuvent intéresser ceux qui programment en mode réel (DOS). Ces instructions sont **cpuid** et **rdtsc**. Si vous chargez eax avec zéro et vous exécutez cpuid, le Pentium (ainsi que les processeurs ultérieurs), retournera la valeur maximale que cpuid peut permettre comme paramètre d'eax. Pour le Pentium, cette valeur est un. Si vous chargez eax avec 1 et exécutez cpuid, le Pentium retournera, dans le registre eax, les informations d'identification du CPU. Tant qu'Intel ne produira de nouvelles puces dans la famille des Pentiums, cette instruction ne sera pas d'une grande utilité.

L'autre nouvelle instruction est **rdtsc** (*read time stamp counter*). Le Pentium gère un compteur de 64 bits qui compte les cycles d'horloge à partir d'un démarrage. L'instruction rdtsc copie la valeur courante du compteur dans la paire edx:eax. Vous pouvez l'utiliser pour mesurer de façon très précise le temps employé par certaines séquences de code.

À côté des instructions présentées ici, les processeurs 80286 et ultérieurs fournissent un ensemble d'instructions *pour le mode protégé*. Ce livre ne les considérera pas, car elles ont un intérêt seulement pour ceux qui écrivent des systèmes d'exploitation. Les programmes qui tournent sur ces systèmes (Windows, Unix, OS/2, etc.) ne s'en servent pas. Donc, elles sont réservées exclusivement au développement de ces systèmes ou à l'écriture de pilotes pour ces systèmes.

6.11 Exemples de programmes

Les programmes suivants montrent l'usage de diverses instructions vues dans ce chapitre[35].

6.11.1 Arithmétique simple (première partie)

```
; Simple arithmétique
; Ce programme montre quelques instructions d'arithmétique simple.

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386                    ;Pour utiliser les registres et les
                option segment:use16    ;modes d'adressage étendus.

dseg            segment para public 'data'

;Voici quelques définitions des variables qu'on va déclarer :

uint            typedef         word    ;Entiers non signés
integer         typedef         sword   ;Entiers signés

;Variables
j               integer ?
k               integer ?
l               integer ?

u1              uint    ?
u2              uint    ?
u3              uint    ?

dseg            ends
```

[34]Il existe des ordinateurs à processeurs multiples qui ont plusieurs Pentium installés. Cependant, ces systèmes utilisent les deux CPU seulement pendant l'exécution de Windows NT, OS/2 ou quelque autre système d'exploitation qui supporte le multiprocessage symétrique.

[35]A partir de ce moment, les codes sources (sauf les commentaires et les titres) ne seront plus traduits en français. Ceci pour vous familiariser avec le style d'écriture des codes sources en assembleur, qui sont toujours en anglais, même dans des pays non anglophones, n.d.t.

```
cseg            segment para public 'code'
assume          cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Initialisation des variables :

                mov     j, 3
                mov     k, -2
                mov     u1, 254
                mov     u2, 22

; Calculer L := j+k et u3 := u1+u2

                mov     ax, J
                add     ax, K
                mov     L, ax
                mov     ax, u1  ;On utilise ADD aussi bien pour les
                add     ax, u2  ;instructions signées et non signées
                mov     u3, ax

; Calculer L := j-k et u3 := u1-u2

                mov     ax, J
                sub     ax, K
                mov     L, ax
                mov     ax, u1  ;Même remarque avec SUB que pour
                                ;la séquence d'instructions précédente
                sub     ax, u2
                mov     u3, ax

; Calculer L := -L

                neg L

; Calculer L := -J
                mov     ax, J   ;L'instruction NEG n'a d'intérêt que
                neg     ax      ;sur des valeurs signées
                mov     L, ax

; Calculer K := K + 1 à l'aide de l'instruction INC

                inc K

; Calculer u2 := u2 + 1 via l'instruction INC
; Notez qu'INC peut être utilisée autant sur des valeurs signées que non signées

                inc u2

; Calculer J := J - 1 en utilisant DEC.

                dec J

; Calculer u2 := u2 - 1 à l'aide de DEC.
; Notez que DEC peut s'utiliser aussi bien sur des valeurs signées que non signées.

                dec u2

Quit:           mov     ah, 4ch ;opcode du DOS pour sortir du programme.
                int 21h         ;Appel du DOS.
Main            endp

cseg            ends
```

```
sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
                end     Main
```

6.11.1 Arithmétique simple (deuxième partie)

```
; Arithmetique simple
; Ce programme montre d'autres instructions d'arithmétique simple.

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386                    ; Pour pouvoir utiliser
                                        ; les registres et les
                option segment:use16    ; modes d'adressage étendus.

dseg            segment para public 'data'

; Quelques définitions de types qu'on utilisera :

uint            typedef word            ;Entiers non signés.
integer         typedef sword           ;Entiers signés.

; Certaines variables que l'on peut utiliser :

j               integer ?
k               integer ?
l               integer ?
u1              uint    ?
u2              uint    ?
u3              uint    ?

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Initialisation des variables :

                mov     j, 3
                mov     k, -2
                mov     u1, 254
                mov     u2, 22

; Multiplication étendue en utilisant les instructions 8086.
;
; Notez que les opérandes signées et non signées requièrent des instructions
; de multiplication séparées
;
; L := J * K (en ignorant tout dépassement de capacité)

                mov     ax, J
                imul    K       ;Calcule DX:AX := AX * K
```

```
                mov     L, ax   ;Ignorer le dépassement dans DX.

; u3 := u1 * u2

                mov     ax, u1
                mul     u2      ;Calcule DX:AX := AX * U2
                mov     u3, ax  ;Ingorer le dépassement dans DX.

; Division étendue en utilisant les instructions 8086.
;
; Comme dans le cas de la multiplication, il y a des instructions séparées selon
; que les opérandes sont signées ou non signées.
;
; Il est absolument impératif que ces séquences d'instructions fassent aussi
; des extensions de signe ou à zéro sur leurs opérandes de 32 bits *avant*
; l'opération de division. Ne pas le faire, peut provoquer une erreur de division et
; planter le programme.
;
; L := J div K

                mov     ax, J
                cwd                     ;*DOIT* élargir AX à DX:AX !
                idiv    K               ;AX := DX:AX/K, DX := DX:AX mod K
                mov     L, ax

; u3 := u1/u2

                mov     ax, u1
                mov     dx, 0           ;Doit élargir AX à DX:AX !
                div     u2              ;AX := DX:AX/u2, DX := DX:AX mod u2
                mov     u3, ax

; Voici les versions spécialisées de IMUL disponibles sur 80286, 80386
; et supérieurs.
; Techniquement, ces instructions fonctionnent seulement avec des opérandes signées.
; Cependant, elles fonctionnent bien aussi avec des opérandes non signées.
; Notez aussi qu'elles produisent un résultat de 16 bits et modifient
; le flag overflow si un dépassement se produit.
;
; L := J * 10 (seulement sur 80286 et supérieurs)

                imul    ax, J, 10               ;AX := J*10
                mov     L, ax

; L := J * K (seulement sur 80386 et ultérieurs)

                mov     ax, J
                imul    ax, K
                mov     L, ax

Quit:           mov ah, 4ch             ;Opcode du DOS pour sortir.
                int 21h                 ;Appelle DOS.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends

                end     Main
```

6.11.3 Opérations logiques

```
; Opérations logiques
; Ce programme montre l'usage des instructions AND, OR, XOR, et NOT

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386                    ;Pour pouvoir utiliser
                                        ; les registres et les
                option segment:use16    ;modes d'adressage étendus.

dseg            segment para public 'data'

; Quelques définitions de variables :

j               word    0FF00h
k               word    0FFF0h
l               word    ?

c1              byte    'A'
c2              byte    'a'

LowerMask       byte    20h

dseg ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Calculer L := J       AND K (opération AND logique sur les bits):

                mov     ax, J
                and     ax, K
                mov     L, ax

; Calculer L := J OR K (opération OR logique sur les bits):

                mov     ax, J
                or      ax, K
                mov     L, ax

; Calculer L := J XOR K (opération XOR logique sur les bits):

                mov     ax, J
                xor     ax, K
                mov     L, ax

; Calculer L := NOT L (opération NOT logique sur les bits):

                not     L

; Calculer L := NOT J (opération NOT logique sur les bits):

                mov     ax, J
                not     ax
                mov     L, ax
```

```
; Désactiver les bits 0..3 dans J :

                and     J, 0FFF0h

; Activer ler bits 0..3 dans K :

                or      K, 0Fh

; Inverser les bits 4..11 dans L :

                xor     L, 0FF0h

; Convertir le caractère dans c1 en minuscules :

                mov     al, c1
                or      al, LowerMask
                mov     c1, al

; Convertir le caractère dans c2 en majuscules :

                mov     al, c2
                and     al, 5Fh         ;Met le bit 5 à zéro.
                mov     c2, al

Quit:           mov ah, 4ch             ;Opcode du DOS pour sortir.
                int 21h                 ;Appelle DOS.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
                end     Main
```

6.11.4 Opérations de décalage et de rotation

```
; instructions de décalage et de rotation

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386                    ;Pour pouvoir utiliser
                                        ; les registres et les
                option segment:use16    ;modes d'adressage étendus.

dseg            segment para public 'data'

; La structure suivante contient les valeurs des bits d'un octet mod-reg-r/m.

mode            struct
modbits         byte    ?
reg             byte    ?
rm              byte    ?
mode            ends

Adrs1           mode    {11b, 100b, 111b}
modregrm                byte    ?

var1            word    1
```

```
var2            word    8000h
var3            word    0FFFFh
var4            word    ?

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Rotations et décalages directs dans la mémoire :
;
; var1 := var1 shl 1

                shl     var1, 1

; var1 := var1 shr 1

                shr     var1, 1

; Sur des processeurs 80286 et ultérieurs, vous pouvez décaler plus d'un bit à la
fois :

                shl     var1, 4
                shr     var1, 4

; Le décalage arithmétique à droite conserve le bit fort après chaque décalage.
; L'instruction SAR suivante produit la valeur 0FFFFh dans var2 :

                sar     var2, 15

; Sur tous les processeurs, vous pouvez spécifier un compte de décalage dans
; le registre CL. L'instruction suivante redonne la valeur 8000h à  var2 :

                mov     cl, 15
                shl     var2, cl

; Vous pouvez utiliser les instructions logiques, de décalage et de rotation
; pour compacter et décompacter des données. Pour exemple, l'instruction suivante
; extrait les bits 10..13 de var3 et garde cette valeur dans var4 :

                mov     ax, var3
                shr     ax, 10  ;Déplace les bits 10..13 dans les bits 0..3.
                and     ax, 0Fh ;Garde seulement les bits 0..3.
                mov     var4, ax

; Vous pouvez utiliser les instructions de rotation pour accélérer le calcul
; de cette valeur sur des processeurs plus lents comme le 80286.

                mov     ax, var3
                rol     ax, 6   ;Six rotations au lieu de 10 décalages.
                and     ax, 0Fh
                mov     var4, ax

; Vous pouvez utiliser les instructions SHL et OR pour fusionner facilement dans une
; seule valeur des champs séparés. Par exemple, le code suivant fusionne les
; champs mod, reg, et r/m (se trouvant sur des octets séparés) dans un seul octet
; mod-reg-r/m :

                mov     al, Adrs1.modbits
                shl     al, 3
                or      al, Adrs1.reg
                shl     al, 3
```

```
                or      al, Adrs1.rm
                mov     modregrm, al

; Si vous utilisiez un processeur 8086 ou 8088 vous auriez à utiliser le code
;suivant :

                mov     al, Adrs1.modbits       ;Obtient le champ mod.
                shl     al, 1
                shl     al, 1
                shl     al, 1[36]
                or      al, Adrs1.reg   ;Obtient le champ reg.
                mov     cl, 3
                shl     al, cl          ;Fait de l'espace pour le champ r/m.
                or      al, Adrs1.rm    ;Fusionne dans le champ r/m.
                mov     modregrm, al    ;Enregistre le résultat.

Quit:           mov     ah, 4ch         ;Opcode du DOS pour sortir
                int 21h                 ;Appel du DOS.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
                end     Main
```

6.11.5 Opérations sur les bits et instructions SET*cc*

```
; Opérations sur les bits et instructions SETcc

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386                    ;Pour pouvoir utiliser
                                        ; les registres et les
                option segment:use16    ;modes d'adressage étendus.

dseg            segment para public 'data'

; Quelques définitions de types pour les variables qu'on va déclarer :

uint            typedef word            ;Entiers non signés.
integer         typedef sword           ;Entiers signés.

; Déclaration de quelques variables

j               integer ?
k               integer ?
u1              uint    2
u2              uint    2
Result          byte    ?

dseg            ends

cseg            segment para public 'code'
```

[36] Attention ! Cette ligne manquait dans l'original, mais il s'agit sûrement d'une erreur, car pour décaler al de trois positions, il faut absoulment trois instructions shl, n.d.t.

```
                assume cs:cseg, ds:dseg

Main proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Initialisation des variables

                mov     j, -2
                mov     k, 2

; Les instructions SETcc gardent un 1 ou un 0 dans leurs opérandes si la condition
; spécifiée est vraie ou fausse respectivement. L'instruction TEST effectue un AND
; logique sur ses opérandes en modifiant les flags en conséquence (en particulier,
; TEST active/désactive le drapeau zéro s'il y a/il n'y a pas un résultat de zéro).
; On peut utiliser ces deux faits pour copier un seul bit (étendu par des zéros)
dans une opérande octet.

                test    j, 11000b       ;Teste les bits #4 et #5.
                setne   Result          ;Result=1 si les bits #4 et #5
                                        ; de J valent 1.
                test    k, 10b          ;Teste le bit #1.
                sete    Result          ;Result=1 si le bit #1 = 0.

; Les instructions SETcc sont particulièrement utiles après une instruction CMP.
; Vous pouvez activer une valeur booléenne selon le résultat de la comparaison.
;
; Result := j <= k

                mov     ax, j
                cmp     ax, k
                setle   Result  ;Notez que "...le" s'applique à des
                                ; valeurs signées.

; Result := u1 <= u2

                mov     ax, u1
                cmp     ax, u2
                setbe   Result  ;Notez que "...be" s'applique à des valeurs
                                ; non signées.

; Ce qui est bien avec les résultats que les instructions SETcc produisent est qu'on
; peut leur appliquer des AND, OR ou XOR et obtenir les mêmes résultats que l'on
; obtiendrait avec des langages de haut niveau comme C, Pascal ou BASIC.
;
; Result := (j < k) AND (u1 > u2)

                mov     ax, j
                cmp     ax, k
                setl    bl              ;Utilisez "...l" pour des
                                        ; comparaisons signées.
                mov     ax, u1
                cmp     ax, u2
                seta    al              ;Utilisez "...a" pour des
                                        ; comparaisons non signées.
                and     al, bl          ;Applique un AND aux deux
                                        ;résultats booléens
                mov     Result, al      ;et garde le résultat.

; Parfois, vous pouvez utiliser les instructions de décalage et de rotation pour
; tester si un bit spécifique est à 1. Par exemple, SHR coupe le bit 0 dans le
; drapeau de retenue et SHR copie le bit fort dans ce même drapeau. On peut tester
; facilement ces bits comme suit :
;
; Result := bit #15 de J
```

```
                mov     ax, j
                shl     ax, 1
                setc    Result

; Result := bit #0 de u1

                mov     ax, u1
                shr     ax, 1
                setc    Result

; Si vous n'utilisez pas un processeur 80386 ou supérieur et que vous ne pouvez pas
; utiliser les instructions SETcc, vous pouvez souvent les simuler. Considérez les
; deux séquences ci-dessus récrites pour un 8086 :
;
; Result := bit #15 de J

                mov     ax, j
                rol     ax, 1           ;Copie le bit 15 dans le bit 0.
                and     al, 1           ;Élimine les autres bits.
                mov     Result, al

; Result := bit #0 de u1 :

                mov     ax, u1
                and     al, 1           ;Élimine les bits superflus.
                mov     Result, al

Quit:           mov     ah, 4ch         ;Opcode du DOS pour sortir.
                int 21h                 ;Appel du DOS.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
                end     Main
```

6.11.6 Opérations sur les chaînes de caractères

```
; Opérations de chaînes

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386                    ;Pour pouvoir utiliser les registres et
                option segment:use16    ; les modes d'adressage étendus.

dseg            segment para public 'data'

String1         byte    "String",0
String2         byte    7 dup (?)
Array1          word    1, 2, 3, 4, 5, 6, 7, 8
Array2          word    8 dup (?)

dseg ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg
```

```
Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Les instructions de chaînes vous permettent de copier facilement des données
; d'un tableau à un autre. Si le drapeau de direction vaut 0, l'instruction
; movsb fait l'équivalent de ce qui suit :
;
;       mov es:[di], ds:[si]
;       inc     si
;       inc     di
;
; Le code suivant copie les sept octets de String1 dans String2 :

                cld             ;Nécessaire si vous voulez incrémenter SI/DI
                lea     si, String1
                lea     di, String2
                movsb                   ;String2[0] := String1[0]
                movsb                   ;String2[1] := String1[1]
                movsb                   ;String2[2] := String1[2]
                movsb                   ;String2[3] := String1[3]
                movsb                   ;String2[4] := String1[4]
                movsb                   ;String2[5] := String1[5]
                movsb                   ;String2[6] := String1[6]

; La séquence suivante montre comment vous pouvez utiliser LODSW et STOWS
; pour manipuler les éléments des tableaux pendant le transfert.
; Le code suivant effectue :
;
; Array2[0] := Array1[0]
; Array2[1] := Array1[0] * Array1[1]
; Array2[2] := Array1[0] * Array1[1] * Array1[2]
; etc.
;
; Sans doute, il serait plus efficace de mettre le code suivant dans
; une boucle, mais ça va venir.

                lea     si, Array1
                lea     di, Array2
                lodsw
                mov     dx, ax
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
```

```
                stosw
                lodsw
                imul    ax, dx
                mov     dx, ax
                stosw

Quit:           mov ah, 4ch             ;Opcode du DOS pour sortir.
                int 21h                 ;Appel du DOS.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
end             Main
```

6.11.7 Sauts inconditionnels[37]

```
; Sauts inconditionnels

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386
                option segment:use16

dseg            segment para public 'data'

; Pointeurs sur des instructions dans le segment de code

IndPtr1         word    IndTarget2
IndPtr2         dword   IndTarget3

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Les instructions JMP transfèrent le contrôle à des emplacements spécifiés par
; le champ opérande. Il s'agit d'habitude d'étiquettes qui apparaissent dans le
; programme.
;
; Il y a beaucoup de variantes de l'instruction JMP. La première est un opcode de
; deux ctets qui transfère le contrôle dans les +/-128 octets autour
; de l'instruction courante :

                jmp     CloseLoc
                nop
CloseLoc:

; La forme suivante est une instruction de trois octets qui permet de
```

[37]Notez que ce chapitre, dans l'original se titulait *Conditional Jump*. Il s'agit d'une erreur que la traduction a corrigée, n.d.t.

```
; sauter n'importe où dans le segment de code courant. Normalement,
; l'assembleur choisit la version la plus courte d'une instruction
; JMP donnée. L'opérande "near ptr" impose un JMP proche (de trois
; octets) :

                jmp     near ptr NearLoc
                nop
NearLoc:

; La troisième variante à considérer est une instruction de cinq octets capable de
; supporter une adresse segmentée complète. Avec cette version de JMP, on peut
; sauter n'importe où dans le programme, même dans un autre segment. L'opérande "far
; ptr" a comme effet d'imposer ce comportement :

                jmp     far ptr FarLoc
                nop
FarLoc:

; Vous pouvez également charger l'adresse cible d'un JMP proche dans un
; registre et sauter indirectement à l'endroit voulu. Notez que vous pouvez
; utiliser pour ceci tout registre général, vous n'êtes pas limités aux
; registres BX, SI, DI ou BP.

                lea     dx, IndTarget
                jmp     dx
                nop
IndTarget:

; Vous pouvez même sauter indirectement via une variable en mémoire.
; C'est-à-dire, vous pouvez sauter directement à travers une variable
; pointeur sans avoir à charger d'abord le pointeur dans
; un registre (le chapitre 8 décrit pourquoi les étiquettes
; suivantes requièrent deux fois deux points).

                jmp     IndPtr1
                nop
IndTarget2:

; Vous pouvez enfin effectuer un saut éloigné indirect par le biais d'une
; variable. Il suffira de spécifier une variable de type dword dans
; l'opérande d'une instruction JMP :

                jmp     IndPtr2
                nop
IndTarget3:

Quit:           mov     ah, 4ch
                int     21h

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends

                end     Main
```

6.11.8 Instructions CALL et INT

```
; instructions CALL et INT
```

```
                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386
                option segment:use16

dseg            segment para public 'data'

; Quelques pointeurs sur nos sous-routines:

SPtr1           word    Subroutine1
SPtr2           dword   Subroutine2

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Subroutine1     proc    near
                ret
Subroutine1     endp

Subroutine2     proc    far
                ret
Subroutine2     endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Appel proche:

                call    Subroutine1

; Appel éloigné :
                call    Subroutine2

; Appel indirect proche via les registres :
                lea     cx, Subroutine1
                call    cx

; Appel indirect proche via la mémoire :
                call    SPtr1

; Appel indirect éloigné via la mémoire :

                call    SPtr2

; INT transfère le contrôle à une routine dont l'adresse apparaît dans la table
; des vecteurs d'interruption (voir le chapitre sur les interruptions pour avoir
; des détails sur cette table). L'appel suivant indique au BIOS du PC
; d'afficher à l'écran le caractère ASCII dans AL.

                mov     ah, 0eh
                mov     al, 'A'
                int     10h

; INTO génère une INT 4 si le drapeau de dépassement de capacité est à 1
; et devient NOP si ce drapeau vaut 0. Vous pouvez utiliser cette instruction
; après une opération arithmétique pour tester rapidement l'éventualité
; d'un dépassement fatal. Note : la séquence qui suit, *ne génère pas*
; un dépassement. Ne la modifiez pas à moins d'ajouter une routine INT 4 dans
; la table des vecteurs d'interruption.
```

```
                mov     ax, 2
                add     ax, 4
                into

Quit:           mov ah, 4ch
                int 21h

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends
                end     Main
```

6.11.9 Sauts conditionnels (première partie)

```
Sauts conditionnels I

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386
                option segment:use16

dseg            segment para public 'data'

J               sword   ?
K               sword   ?
L               sword   ?

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Les sauts conditionnels du 8086 sont limités à une plage de +/- 128 octets
; parce que leur opcode est seulement de deux octets (un octet pour l'opcode
; et l'autre pour le déplacement).

                .8086
                ja      lbl
                nop
lbl:

; MASM 6.x corrigera automatiquement les sauts hors de plage. Les deux séquences qui
; suivent sont équivalentes :

                ja      lbl2
                byte    150 dup (0)
lbl2:
                jna     Temp
                jmp     lbl3
```

```
Temp:
                byte    150 dup (0)
lbl3:

; Les processeurs 80386 et supérieurs supportent des versions spéciales
; des sauts conditionnels permettant un déplacement de deux octets.
; Si vous spécifiez un processeur 80386, MASM 6.x assemblera le code
; en en tenant compte.

                .386
                ja      lbl4
                byte    150 dup (0)
lbl4:

; Les sauts conditionnels fonctionnent bien en combinaison avec l'instruction
; CMP, vous permettant d'exécuter certaines séquences d'instructions seulement
; si une condition est vraie (ou fausse).
;
; if (J <= K) then
; L := L + 1
; else L := L - 1

                mov     ax, J
                cmp     ax, K
                jnle    DoElse
                inc     L
                jmp     ifDone
DoElse:         dec L
ifDone:

; Vous pouvez aussi utiliser un saut conditionnel pour créer une boucle :
;
; while (j >= k) do begin
;
; j := j - 1;
; k := k + 1;
; L := j * k;
; end;

WhlLoop:                mov     ax, j
                cmp     ax, k
                jnge    QuitLoop
                dec     j
                inc     k
                mov     ax, j
                imul    ax, k
                mov     L, ax
                jmp     WhlLoop

QuitLoop:

Quit:           mov     ah, 4ch         ;Opcode du DOS pour sortir du programme.
                int     21h             ;Appel du DOS.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends

                end     Main
```

6.11.10 Sauts conditionnels (deuxième partie)

```
; instructions de sauts conditionnels (partie II)

                .xlist
                 include        stdlib.a
                 includelib     stdlib.lib
                .list

                .386
                option segment:use16

dseg            segment para public 'data'

Array1          word    1, 2, 3, 4, 5, 6, 7, 8
Array2          word    8 dup (?)
String1         byte    "This string contains lower case characters",0
String2         byte    128 dup (0)
j               sword   5
k               sword   6
Result          byte    ?

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Vous pouvez utiliser LOOP pour répéter une séquence d'instructions un nombre
; déterminé de fois. Considérez le code extrait du programme EX6_5.ASM qui
; a utilisé des instructions de manipulation de chaînes.
;
; Le code suivant utilise une instruction LOOP pour calculer :
;
; Array2[0] := Array1[0]
; Array2[1] := Array1[0] * Array1[1]
; Array2[2] := Array1[0] * Array1[1] * Array1[2]
; etc.

                cld
                lea     si, Array1
                lea     di, Array2
                mov     dx, 1           ;Initialise pour la première fois.
                mov     cx, 8           ;Huit éléments dans les tableaux.
LoopHere:               lodsw
                imul    ax, dx
                mov     dx, ax
                stosw
                loop    LoopHere

; L'instruction LOOPNE est très utile pour contrôler les boucles qui
; terminent sous une certaine condition ou quand la boucle dépasse un
; certain nombre d'itérations (répétitions). Par exemple, supposez
; que string1 contient une séquence de caractères terminée par un
; octet qui contient zéro. Si vous voulez convertir ces caractères en
; majuscules et les copier dans string2, vous pouvez utiliser le code suivant.
; Notez que celui-ci ne copie pas plus de 127 caractères, car string2 permet
; seulement un nombre maximal de 127 octets (plus un zéro de terminaison).

                lea     si, String1
                lea     di, String2
```

```
                mov     cx, 127         ;Au plus 127 caractères dans string2.
CopyStrLoop:    lodsb                   ;Obtention des caractères depuis string1.
                cmp     al, 'a'         ;Tester si les caractères en
                                        ; minuscules sont non signés.
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 5Fh         ;Convertir en minuscules.
NotLower:
                stosb
                cmp     al, 0           ;Tester le caractère de terminaison.
                loopne  CopyStrLoop     ;Sortir si al ou cx = 0.

; Si vous n'utilisez pas un CPU 80386 ou supérieur et qu'il vous faut
; la fonctionnalité des instructions SETcc, vous pouvez facilement
; obtenir le même résultat en utilisant ce qui suit :
;
; Result := J <= K;

                mov     Result, 0       ;Suppose que c'est faux.
                mov     ax, J
                cmp     ax, K
                jnle    Skip1
                mov     Result, 1       ;Mettre à 1 si J <= K.
Skip1:

; Result := J = K;
                mov     Result, 0       ;Suppose que c'est faux.
                mov     ax, J
                cmp     ax, K
                jne     Skip2
                mov     Result, 1
Skip2:

Quit:           mov ah, 4ch             ;Opcode du DOS pour sortir du programme.
                int 21h                 ;Appel du DOS.

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte 1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte 16 dup (?)
zzzzzzseg       ends

                end     Main
```

6.12 Exercices de laboratoire

Dans cet ensemble d'exercices, vous écrirez des programmes en IBM/L *instruction BenchMarking Language*[38]. Cet outil vous permet de tester la durée de certaines séquences d'instructions.

6.12.1 Le système IBM/L

IBM/L vous permet de chronométrer des séquences d'instructions pour voir combien de temps elles prennent *réellement* pour s'exécuter. Les temps de cycle dans la plupart des livres de langage assembleur 80x86 sont horriblement inexacts

[38]Vous trouverez ce programme dans le sous-répertoire du chapitre 6.

car ils supposent le meilleur cas absolu. Ce n'est pas le cas avec l'IBM/L. Cet outil a une valeur inestimable à l'heure d'optimiser un programme. On peut essayer diverses séquences qui produisent le même résultat et voir lesquelles sont plus rapides.

L'IBM/L se sert de l'horloge système à 1/18 de seconde et il évalue les exécutions en termes de tics d'horloge. Par conséquent, son usage serait complètement inutile si on voulait mesurer la vitesse d'une seule instruction (car l'exécution d'une instruction isolée emploie beaucoup moins de temps). L' IBM/L fonctionne en exécutant en répétition une séquence de code pendant de milliers (parfois de millions) de fois et en mesurant le temps écoulé. En faisant cela, il soustrait automatiquement le temps d'exécution de la boucle.

L'IBM/L est un compilateur qui traduit un code source en assembleur. En assemblant et en exécutant le programme résultant, il donne une liste de durées relatives pour diverses séquences d'instructions. Un programme IBM/L consiste en courtes séquences de code assembleur et quelques instructions de contrôle de flux décrivant comment mesurer les performances de ces séquences. Un programme IBM/L prend la forme suivante :

```
#data
        <déclarations de variables>[39]
#enddata

#unravel <constante nombre entier>
#repetitions <constante nombre entier>
#code ("title")
%init
        <instructions initiales dont la durée ne compte pas>
%eachloop
        <instructions répétées une fois dans chaque boucle; durée ignorée>
%discount
        <instructions effectuées pour chaque séquence; durée ignorée>
%do
        <instructions à mesurer>
#endcode

<sections #code..#endcode additionnelles>

#end
```

Notez : les sections %init, %eachloop et %discount sont optionnelles.

Un programme IBM/L commence avec une section "#DATA" optionnelle qui se termine avec une directive "#ENDDATA". Toutes les lignes entre ces deux directives sont copiée dans un segment de données d'un programme assembleur en sortie. Normalement, on place dans cette partie des variables globales.

```
Exemple de section de données
#DATA
I               word    ?
J               word    ?
K               dword   ?
ch              byte    ?
ch2             byte    ?
#ENDDATA
```

Ces lignes seront copiées dans le segment de données du programme que l'IBM/L créera. Ces variables seront disponibles dans toutes les séquences #code..#endcode que vous placez dans votre programme.

A la suite de la section des données, on trouve une ou plusieurs sections de code. Une section de code consiste en directives #repetition et #unravel optionnelles, suivies par des directives #code..#endcode requises.

#repetition a la syntaxe suivante :

```
#repetition     <constante_entier>
```

(Le # doit être au début de la ligne). La constante nombre entier est une valeur de 32 bits, vous permettant de spécifier des valeurs allant de zéro à environ deux milliards. Mais généralement, ces valeurs sont inférieures à

[39]Tout ce qui est contenu entre <> est une partie de la syntaxe à remplacer par du code valide. Par exemple, dans if <condition> <instructions>, il faut remplacer <condition> par une condition quelconque et <instructions> par des instructions quelconques, n.d.t.

quelque centaines et encore moins sur des machines plus lentes. Plus la valeur est grande plus précise sera l'analyse de la durée. Néanmoins, utiliser un grand nombre de répétitions équivaut à ralentir l'exécution du programme écrit par IBM/L.

La directive #repetition indique à IBM/L de générer une boucle qui répétera le segment de code suivant *constante_entiere* fois. Si vous ne spécifiez pas de valeur (souvenez-vous que la directive est optionnelle), la valeur par défaut est 327680. Une fois que vous avez spécifié une valeur, elle sera utilisée pour toutes les séquences qui suivront, au moins jusqu'à la prochaine spécification. L'instruction #repetition doit impérativement apparaître en dehors de toute séquence #code..#endcode et affecte la ou les sections #code qui suivront.

Si vous êtes intéressé par le temps d'exécution d'une seule ligne exécutée plusieurs fois dans une boucle, sachez que placer ces instructions dans une boucle serrée peut affecter grandement l'exactitude des résultats. N'oubliez pas que le fait d'exécuter une instruction de transfert de contrôle (nécessaire pour une boucle) vide la queue de préchargement, ce qui a un grand impact sur les temps d'exécution. C'est ici que la directive #unravel vient en aide. Elle permet de copier un bloc de code plusieurs fois dans le corps d'une boucle, en réduisant par conséquent la surcharge que les sauts conditionnels et d'autres instructions de contrôle de boucle représentent. La directive #unravel a la syntaxe suivante :

```
#unravel                <comptage>
```

Le caractère # doit être au début de la ligne. *comptage* est une constante entière de 16 bits qui spécifie le nombre de fois que l' IBM/L doit copier le code dans le corps de la boucle.

Notez que la séquence du code dans la section #CODE exécutera *comptage*constante_entier* fois, car la directive unravel répète la séquence de code *comptage* fois dans la boucle qui se répète *constante_entier* fois.

Dans sa forme la plus simple, la directive #CODE ressemble à ce qui suit :

```
#CODE ("Title")
%DO
        <instructions en assembleur>
#ENDCODE
```

Le titre peut être toute chaîne de caractères que vous choisissez. IBM/L affichera ce titre en affichant les résultats de sa section code. IBM/L prendra les instructions spécifiées et les placera dans une boucle (un nombre multiple de fois si la directive #unravel est présente). Pendant l'exécution, le fichier source en assembleur chronmétrera ce code et affichera un temps en nombre de tics d'horloge. Par exemple :

```
#unravel 16                     ;16 copies du code à l'intérieur de la boucle
#repetitions 960000             ;960000 fois...
#code ("instruction MOV AX, 0")  ;Titre de l'instruction
%do
        mov ax, 0
#endcode
```

Le code ci-dessus génère un programme en assembleur qui exécutera *mov ax, 0* 16*960000 fois et annoncera le temps employé par cette exécution.

Beaucoup de programmes IBM/L ont des sections de code multiples. De nouvelles sections peuvent suivre immédiatement les précédentes, par exemple :

```
#unravel 16                     ;16 copies du code à l'intérieur de la boucle
#repetitions 960000             ;960000 fois...
#code ("instruction MOV AX, 0")  ;Titre de l'instruction
%do
        mov     ax, 0
#endcode

#code ("instruction XOR AX, AX")
%do
        xor     ax, ax
#endcode
#end
```

La séquence qu'on vient de voir exécutera *mov ax, 0* et *xor ax, ax* pour un nombre total de 16*960000 fois et annoncera le temps que ces exécutions ont pris. En comparant les résultats, vous pourrez déterminer laquelle des deux séquences est la plus rapide.

Tout comme en assembleur, le point-virgule indique un commentaire que le compilateur IBM/L ignorera. Les commentaires écrits en IBM/L ne seront pas insérés dans le fichier de sortie que l' IBM/L génère.

Tous les programmes IBM/L doivent se terminer avec une directive #END. Voici un exemple de programme complet :

```
#data
                even
i               word    ?
                byte    ?
j               word    ?
#enddata

#unravel 16
#repetitions 32, 30000
#code ("MOV de mots alignés");
%do
                mov     ax, i
#endcode
#code ("MOV de mots non alignés")
%do
                mov     ax, j
#endcode
#end
```

Notez qu'on aurait pu insérer deux sections optionnelles entre #code et %do. La première aurait pu être %init, qui commence une section d'initialisation ; IBM/L émet ces sections d'initialisation avant la boucle, exécute ce code seulement une fois. Il ne compte pas leur temps d'exécution en chronométrant la boucle. Ceci permet d'affecter des valeurs importantes avant un test, affectations qui ne seront pas prises en comptes dans la mesure du temps d'exécution.

```
#data
i               dword   ?
#enddata
#repetitions    100000
#unravel        1
#code
%init
                mov     word ptr i, 0
                mov     ptr i+2, 0
%do
                mov     cx, 200
lbl:            inc     word ptr i
                jnz     NotZero
                inc     word ptr i+2
NotZero:        loop    lbl             ;lbl = label = étiquette
#endcode
#end
```

Parfois, vous voulez que la directive #repetitions répète une section de code plusieurs fois. Cependant, vous pouvez vouloir parfois n'exécuter certaines instructions qu'une seule fois dans chaque itération (c'est-à-dire à ne pas copier le code plusieurs fois dans la boucle). La directive %eachloop permet justement ceci. Notez que l'IBM/L ne prend pas en compte le temps d'exécution consommé par un code exécuté dans une section %eachloop.

Exemple :

```
#data
i               word    ?
j               word    ?
#enddata

#repetitions 40000
#unravel 128
```

```
#code
%init           ;ce qui suit n'est exécuté qu'une fois
                mov     i, 0
                mov     j, 0

%eachloop       ;ce qui suit est exécuté 40 000 fois et non 128 * 40 000 fois
                inc     j

%do             ;et, finalement, le contenu de la boucle, exécuté 128 * 40 000 fois
                inc     i
#endcode
#end
```

Dans le code qu'on vient de voir, l'IBM/L compte exclusivement le temps requis pour incrémenter i et il ne mesure pas les instructions faisant partie des sections %init et %eachloop. Le code de ce %eachloop est exécuté seulement une fois pour chaque itération et n'est pas conditionné par la directive %unravel (l'instruction inc i ci-dessus, par exemple, s'exécute 128 fois par itération de boucle à cause de #unravel. Il peut vous arriver aussi de devoir insérer certaines instructions dans la section %do mais de ne pas vouloir les mesurer. Pour réaliser ceci, il vous faudra utiliser la directive %discount. Voici un exemple complet :

```
#data
        <déclarations de données>
#enddata
#repetitions <valeur1>, <valeur2>
#unravel <comptage>
#code
%init
        <code d'initialisation exécuté seulement une fois>
%eachloop
        <code d'initialisation de boucle, exécuté une fois pour chaque itération>
%discount
        <instructions non mesurées, exécutées une fois pour chaque itération>
%do
        <les instructions que vous voulez mesurer>
#endcode
<sections de code additionnelles>
#end
```

Pour utiliser l'application l'IBM/L vous aurez besoin de plusieurs fichiers. IBML.EXE est le programme exécutable et on l'exécute comme suit :

```
C:> IBML nomfichier.IBM
```

Ceci lit un fichier source IBML (le nomfichier.IBM) et écrit un programme en assembleur sur la sortie standard (l'écran). Normalement, on peut avoir besoin de rediriger ce programme dans un fichier facilement lisible :

```
C:> IBML nomfichier.IBM >nomfichier.asm
```

Une fois que le fichier source .asm a été créé, vous pouvez l'assembler et l'exécuter. Le fichier .exe qui en résulte, affichera les statistiques des temps d'exécution.

Pour exécuter correctement un programme IBML, il faut inclure le fichier IBMLINC.A dans le répertoire de travail courant. Il s'agit d'un squelette source de programme assembleur dans lequel IBM/L insérera le code source en assembleur. Ce fichier est librement modifiable[40]. Gardez cependant à l'esprit que l'IBM/L, en lisant ce fichier, s'attend à des marqueurs (";##") pour savoir où insérer le code. Si vous modifiez IBMLINC.A, soyez donc très prudent avec ces marqueurs.

Le fichier source en sortie s'attend à la présence de la bibliothèque UCR standard et par conséquent aux fichiers d'inclusion STDLIB (stdlib.a) et au fichier de la bibliothèque (stdlib.lib).

Dans le Chapitre 1, vous avez appris à installer les fichiers de la bibliothèque standard sur votre disque dur. Ces fichiers doivent être présents dans votre répertoire courant de travail (ou bien dans vos variables d'environnement), sinon, l'assembleur n'assemblera pas correctement le fichier source assembleur en sortie. Voir le Chapitre 1 pour revoir ce sujet et le Chapitre 7 pour en savoir plus sur la bibliothèque en tant que telle.

[40]Mais avant d'y jouer dessus, n'oubliez jamais d'en faire une copie de sauvegarde, n.d.t.

Maintenant, voici certains exemples de programmes, vous aidant à mieux vous familiariser avec le style de l'IBM/L :

```
; Programme IBML d'exemple : TESTMUL.IBM.
; Ce code compare le temps d'exécution de l'instruction MUL avec
; diverses instructions shift et add équivalentes.

#repetitions 480000
#unravel 1
; Le contrôle suivant vérifie combien il faut de temps pour multiplier deux
; valeurs à l'aide de IMUL.

#code ("Multiply by 15 using IMUL")
%do
                .286
                mov     cx, 128
                mov     bx, 15
MulLoop1:       mov     ax, cx
                imul    bx
                loop    MulLoop1
#endcode

; Faire le même test avec les instructions IMUL étendues (pour processeurs 80286
; ou ultérieurs)

#code ("Multiplying by 15 using IMUL")
%do
                mov     cx, 128
MulLoop2:       mov     ax, cx
                imul    ax, 15
                loop    MulLoop2
#endcode

; Maintenant, on multiplie par 15 en utilisant un décalage de quatre bits
; et une soustraction.

#code ("Multiplying by 15 using shifts and sub")
%init
%do
                mov     cx, 128
MulLoop3:       mov     ax, cx
                mov     bx, ax
                shl     ax, 4
                sub     ax, bx
                loop    MulLoop3
#endcode
#end
```

Sortie générée par TESTMUL.IBM :

```
IBM/L 2.0
Public Domain instruction Benchmarking Language
by Randall Hyde, inspired by Roedy Green
All times are measured in ticks, accurate only to ±2.

CPU: 80486

Computing Overhead: Multiply by 15 using IMUL
Testing: Multiply by 15 using IMUL
Multiply by 15 using IMUL :370
Computing Overhead: Multiplying by 15 using IMUL
Testing: Multiplying by 15 using IMUL
Multiplying by 15 using IMUL :370
Computing Overhead: Multiplying by 15 using shifts and sub
Testing: Multiplying by 15 using shifts and sub
Multiplying by 15 using shifts and sub :201
```

```
; Programme IBML d'exemple MOVs
; Ce code compare l'usage de mov de registre à registre et l'usage de mov
; de registre à mémoire

#data
i               word    ?
j               word    ?
k               word    ?
l               word    ?
#enddata
#repetitions 30720000
#unravel 1

; Le contrôle suivant vérifie combien il faut de temps pour passer des données
; de registre à registre et de registre à mémoire

#code ("Register-Register moves, no Hazards")
%do
                mov     bx, ax
                mov     cx, ax
                mov     dx, ax
                mov     si, ax
                mov     di, ax
                mov     bp, ax
#endcode

#code ("Register-Register moves, with Hazards")
%do
                mov     bx, ax
                mov     cx, bx
                mov     dx, cx
                mov     si, dx
                mov     di, si
                mov     bp, di
#endcode

#code ("Memory-Register moves, no Hazards")
%do
                mov     ax, i
                mov     bx, j
                mov     cx, k
                mov     dx, l
                mov     ax, i
                mov     bx, j
#endcode

#code ("Register-Memory moves, no Hazards")
%do
                mov     i, ax
                mov     j, bx
                mov     k, cx
                mov     l, dx
                mov     i, ax
                mov     j, bx
#endcode
#end
```

Sortie générée par ce programme :

```
IBM/L 2.0
Public Domain instruction Benchmarking Language
by Randall Hyde, inspired by Roedy Green
All times are measured in ticks, accurate only to Ò 2.
CPU: 80486
```

```
Computing Overhead: Register-Register moves, no Hazards
Testing: Register-Register moves, no Hazards
Register-Register moves, no Hazards :25
Computing Overhead: Register-Register moves, with Hazards
Testing: Register-Register moves, with Hazards
Register-Register moves, with Hazards :51
Computing Overhead: Memory-Register moves, no Hazards
Testing: Memory-Register moves, no Hazards
Memory-Register moves, no Hazards :67
Computing Overhead: Register-Memory moves, no Hazards
Testing: Register-Memory moves, no Hazards
Register-Memory moves, no Hazards :387
```

6.12.2 Exercices IBM/L

Le répertoire du chapitre 6 contient différents exemples de programmes IBM/L (les fichiers avec l'extension .ibm). Ex6_1.ibm teste trois séquences qui calculent la valeur absolue d'un entier. Ex6_2.ibm teste trois façons différentes de faire un décalage de huit bits à gauche. Ex6_3.ibm teste l'accès à des données de type word à des adresses paires et impaires de la mémoire. Ex6_4.ibm compare le temps requis pour charger es:bx d'un emplacement de mémoire avec le temps requis pour faire la même chose avec une constante. Ex6_5.ibm compare le temps requis pour permuter deux registres avec et sans l'instruction XCHG. Ex6_6.ibm compare l'instruction de multiplication avec les instructions décalage/addition équivalentes. Ex6_7.ibm compare la vitesse d'un mov reg, reg avec celle d'un mov reg, mem.

Compilez chacun de ces programmes à l'aide de la commande DOS suivante :

```
ibml ex6_1.ibm >ex6_1.asm
```

Pour votre rapport : IBM/L écrit sa sortie dans la sortie standard, donc utilisez une commande de redirection vers un fichier comme celle ci-dessus. Une fois créé le fichier, assemblez-le avec MASM et exécutez le résultat. Incluez le listing et le résultat dans votre rapport. **Pour aller plus loin :** écrivez vos propres programmes IBM/L afin de tester certaines séquences d'instructions. Incluez également vos codes sources IBM/L.

Attention : Afin d'obtenir les résultats les plus précis évitez d'exécuter les programmes créés par vos fichiers IBM/L sous Windows, mais lancez-les plutôt à partir du mode DOS pur.

6.13 Projets de programmation

1) Écrivez une courte routine "GetLine" qui lit jusqu'à 80 caractères de l'utilisateur et place ces caractères à des emplacements successifs d'un tampon dans votre segment de données. Pour saisir et afficher les caractères utilisez les appels du BIOS INT 16h et INT 10h décrits dans ce chapitre. La ligne doit terminer à la première occurrence du caractère *retour à la ligne* (code ASCII 0Dh) ou après que l'utilisateur a entré le 80me caractère. Assurez-vous de compter le nombre de caractères qui sont effectivement saisis par l'utilisateur, pour utiliser cette donnée plus tard. Dans le répertoire du chapitre 6, il y a un programme "shell" conçu pour ce projet (proj6_1.asm).

2) Modifiez la routine que vous venez d'écrire pour qu'elle traite correctement le caractère de retour arrière (*backspace*) (code ASCII 08h). Chaque fois que l'utilisateur presse cette touche, il faudra supprimer la frappe précédente du tampon (et s'il n'y a pas un caractère précédent dans le tampon, alors le backspace sera ignoré).

3) Vous pouvez utiliser l'opération XOR pour crypter et décrypter des données. Si vous faites un XOR sur tous les caractères d'un message contre une certaine valeur, vous finirez par brouiller le message. Vous pouvez récupérer le message original en appliquant encore un XOR sur les caractères de ce message avec la même valeur. Modifiez le code du projet n°2 de sorte qu'il encode chaque octet du message avec la valeur 0Fh et affiche le message ainsi encrypté à l'écran. Après l'avoir affiché, décryptez-le en lui appliquant de nouveau un XOR 0Fh. Affichez alors le message original. Notez que vous pouvez utiliser la valeur du compteur obtenue par le code "GetLine" pour déterminer combien de caractères il faut traiter.

4) Écrivez une routine "PutString" capable d'afficher les caractères pointés par la paire de registres es:di. Cette routine doit afficher tous les caractères, jusqu'au zéro de terminaisons de chaîne (mais sans inclure ce dernier). Elle doit aussi préserver tous les registres qu'elle modifie. Il y a un programme "shell" (PROJ6_4.ASM)[41] fourni dans le répertoire du chapitre 6 qui a été conçu pour ce projet.

5) Pour produire une valeur entière de 16 bits dans son équivalent de chaîne nuémérique dans le code ASCII (par exemple le chiffre 1 a le numéro 48), vous pouvez utiliser l'algorithme suivant :

```
if value = 0 then write('0')
else begin
        DivVal := 10000;
        while (Value mod DivVal) = 0 do begin
                Value := Value mod DivVal;
                DivVal := DivVal div 10;
        end;

        while (DivVal > 1) do begin
                write ( chr( Value div DivVal + 48)); (* 48 = '0' *)
                Value := Value mod DivVal;
                DivVal := DivVal div 10;
        end;
end;
```

Écrivez une courte routine qui prend une valeur quelconque dans ax et l'affiche comme une chaîne décimale correspondante. Utilisez l'INT 10h pour afficher les caractères à l'écran. Vous pouvez utiliser le programme "shell" fourni dans le répertoire du Chapitre Six (proj6_5.asm) pour commencer ce projet.

6) Pour saisir un entier de 16 bits du clavier, vous avez besoin d'utiliser un code qui utilise l'algorithme suivant :

```
Value := 0
repeat
        getchar(ch);
        if (ch >= '0') and (ch <= '9') then begin
                Value := Value * 10 + ord(ch) - ord('0');
        end;
until (ch < '0') or (ch > '9');
```

Utilisez l'instruction INT 16h (décrite dans ce chapitre) pour saisir des caractères tapés au clavier. Utilisez la routine de sortie du projet 4 pour afficher le résultat entré. Pour commencer le projet, vous pouvez vous servir du fichier modèle PROJ6_6.ASM du répertoire du chapitre 6.

6.14 Résumé

La famille des processeurs 80x86 présente un jeu CISC d'instructions très riche. Tous les processeurs de cette catégorie maintiennent leur compatibilité avec les processeurs antérieurs et en admettant toutes les instructions. Les programmes écrits pour cette famille fonctionnent généralement pour tous ses membres, alors que ceux écrits avec les nouvelles instructions des processeurs à partir du 80286 fonctionneront seulement sur des CPU modernes. La même chose vaut pour les processeurs 80386 et tous les suivants.

Les CPU décrits dans ce chapitre sont les 8086/8088, le 80286, le 80386, le 80486 et le Pentium (80586). Intel a produit également un processeur 80186, mais celui-ci n'a pas été très utilisé dans les ordinateurs personnels[42].

Le jeu d'instructions 80x86 peut être logiquement divisé en huit catégories :

- "Instructions de transfert de données" (6.3)
- "Instructions de conversion" (6.4)
- "Instructions arithmétiques", (6.5)
- "Instructions logiques et de manipulation de bits" (6.6)

[41]Ce programme n'est pas présent dans le dossier du chapitre 6, malgré cette mention, n.d.t.
[42]Il y a, par conséquent, peu de PC qui se servent de ce CPU. Les applications de contrôle ont été les principaux utilisateurs de ce système. Le 80186 comprend la plupart des instructions spécifiques du 80286, mais il ne dispose pas des instructions du mode protégé qui ont été intégrés à partir du 80286.

• "Instructions d'E/S" (6.7)
• "Instructions de traitement des chaînes" (6.8)
• "Instructions de contrôle de flux" (6.9)
• "Instructions diverses" (6.10)

Beaucoup d'instructions affectent différents bits du registre drapeaux (flags) des 80x86. Certaines peuvent tester ces drapeaux, comme si elles étaient des valeurs booléennes. Les *flags* indiquent aussi le résultat d'une comparaison (égalité, infériorité ou supériorité). Ce sujet a été traité dans les sections suivantes :

• "Le registre d'état du processeur" (6.1)
• "Les instructions S*et on Condition*" (6.6.5)
• "Les instructions de sauts conditionnels" (6.9.4)

Il y a aussi diverses instructions permettant de transférer des données entre les registres et la mémoire. Il s'agit d'instructions qu'on utilise le plus souvent. La famille 80x86 fournit un nombre diverse de ces instructions, afin d'aider à faciliter la tâche d'écrire des programmes rapides et efficaces. Pour les détails voir :

• "Instructions de transfert de données" (6.3)
• "L'instruction MOV" (6.3.1)
• "L'instruction XCHG" (6.3.2)
• "Les instructions LDS, LES, LFS et LSS" (6.3.3)
• "L'instruction LEA" (6.3.4)
• "Les instructions PUSH et POP" (6.3.5)
• "Les instructions LAHF et SAHF" (6.3.6)

Il existe également plusieurs instructions pour convertir des données d'une forme à l'autre, pour les extensions signées et les extensions de zéros pour les instructions de traduction de valeurs d'une table et pour celles de conversion grand/petit endien. Voir :

• "Les instructions MOVZX, MOVSX, CBW, CWD, CWDE et CDQ" (6.4.1)
• "L'instruction BSWAP" (6.4.2)
• "L'instruction XLAT" (6.4.3)

Les instructions arithmétiques permettent toutes les opérations communes : addition, multiplication, soustraction, division, négation, comparaisons, incréments, décréments, ainsi que d'autres instructions de support à l'arithmétique BCD (binaire codé décimal) : AAA, AAD, AAM, AAS, DAA et DAS. Pour des informations sur ces instructions, voir :

• "Instructions arithmétiques" (6.5)
• "Les instructions d'addition : ADD, ADC, INC, XADD, AAA et DAA" (6.5.1)
• "Les instructions ADD et ADC" (6.5.1.1)
• "L'instruction INC" (6.5.1.2)
• "L'instruction XADD" (6.5.1.3)
• "Les instructions de soustraction : SUB, SBB, DEC, AAS et DAS" (6.5.2)
• "L'instruction CMP" (6.5.3)
• "Les instructions CMPXCHG et CMPXCHG8B" (6.5.4)
• "L'instruction NEG" (6.5.5)
• "Les instructions de multiplication : MUL, IMUL et AAM" (6.5.6)
• "Les instructions de division : DIV, IDIV et AAD" (6.5.7)

Vous avez également à disposition un riche ensemble d'opérations booléennes, de décalage, de rotation et de manipulation de bits. Ces instructions manipulent individuellement les bits de leurs opérandes vous permettant d'effectuer des AND, OR, XOR et NOT, de faire des décalages et des rotations, de tester et activer/nettoyer/invertir les bits et finalement mettre une opérande à 0 ou à 1 selon l'état du registre flags. Pour plus d'informations, revoir :

• "Instructions logiques et de manipulation de bits (6.6)
• "Les instructions logiques : AND, OR, XOR et NOT" (6.6.1)
• "Les instructions de rotation : RCL, RCR, ROL et ROR" (6.6.3)
• "Les opérations sur les bits" (6.6.4)
• "Les instructions « Set on Condition »" (6.6.5)

Il y a aussi deux instructions d'E/S dans le jeu 80x86 : IN et OUT. Il s'agit vraiment de formes spéciales d'instruction MOV, qui agissent dans l'espace d'adressage des entrées/sorties au lieu que dans l'espace d'adressage de la mémoire. Normalement, on utilise ces instructions pour accéder aux registres matériels des composantes périphériques. On a traité ces instructions au paragraphe :

- "Instructions d'E/S" (6.7)

La famille 80x86 fournit un répertoire étendu d'instructions manipulant des chaînes de données. Ces instructions comprennent movs, lods, stos, scas, cmps, ins, outs, rep, repz, repe, repnz et repne. Pour plus d'informations :

- "Instructions de chaînes" (6.8)

Les instructions de transfert de contrôle permettent de créer des boucles, des sous-routines, des séquences conditionnelles et divers autres tests. On a vu ces instructions aux paragraphes suivants :

- "Instructions de contrôle de flux" (6.9)
- "Sauts inconditionnels" (6.9.1)
- "Les instructions CALL et RET" (6.9.2)
- "Les instructions INT, INTO, BOUND et IRET" (6.9.13)
- "Sauts conditionnels" (6.9.4)
- "Les instructions JCXZ/JECXZ" (6.9.5)
- "L'instruction LOOP" (6.9.6)
- "L'instruction LOOPE/LOOPZ" (6.9.7)
- "L'instruction LOOPNE/LOOPNZ" (6.9.8)

Ce chapitre décrit finalement d'autres instructions diverses. Ces instructions manipulent directement les drapeaux dans le registre flags, donnent accès à certains services du processeur ou effectuent des opérations en mode protégé. Le chapitre n'a fait qu'une brève mention des instructions en mode protégé, car elles ne sont pas d'usage dans des applications autres que des programmes système ou des systèmes d'exploitation. Référez-vous à un ouvrage plus spécifique pour en avoir une description détaillée. Ce sujet a été traité au paragraphe :

- "Instructions diverses" (6.10)

5.11 Questions

1. Donnez un exemple pour montrer qu'il faut n+1 bits pour contenir la somme de deux valeurs binaires de n bits.

2. En insérant des instructions avant ADC et SBB, on peut forcer ces dernières à se comporter exactement comme ADD et SUB. Quelle instruction faudrait-il insérer pour que ADC ait le même effet que ADD ? Et quelle instruction faudrait-il insérer pour que SBB se comporte comme SUB ?

3. Etant donné que vous pouvez utiliser les instructions PUSH et POP pour manipuler des données qui se trouvent au sommet de la pile, expliquez comment vous pourriez modifier une adresse de retour au sommet de la pile, de sorte à permettre qu'une instruction RET retourne deux octets après l'adresse de retour originale.

4. Donnez au moins quatre manières différentes d'ajouter 2 à la valeur du registre BX. Aucune d'elles ne doit employer plus de deux instructions (sachez qu'il existe jusqu'à six moyens de le faire).

5. Supposez que l'adresse cible des quatre sauts conditionnels suivants soit au-delà de la plage possible pour des branchements courts. Modifiez chacune de ces instructions de sorte qu'elles exécutent des opérations correctes (par exemple vous pouvez sauter inconditionnellement sur toute la distance) :

   a) JS Etiquette1 b) JE Etiquette2 c) JZ Etiquette3

   d) JC Etiquette4 e) JBE LaBas f) JG Etiquette5

6. Expliquez la différence entre le drapeau de retenue (carry flag) et le drapeau de dépassement de capacité (overflow flag).

7. Dans quel contexte utiliseriez-vous les instructions CBW et CWD pour effectuer l'extension signée d'une valeur ?

8. Quelle est la différence entre les instructions "MOV reg, immédiat" et "LEA reg, adresse" ?

9. Qu'est-ce que l'instruction INT nn empile que l'instruction CALL FAR n'empile pas ?

10. Dans quel but utilise-t-on normalement l'instruction JCXZ ?

11. Expliquez le fonctionnement des instructions LOOP, LOOPE/LOOPZ et LOOPNE/LOOPNZ.

12. Quels registres autres que flags sont-ils affectés par les instructions MUL, IMUL, DIV et IDIV ?

13. Énumérez trois différences entre les instructions "DEC AX" et "SUB AX, 1".

14. Parmi les instructions logiques, de décalage et de rotation, lesquelles n'affectent pas le zero flag ?

15. Pourquoi l'instruction SAR nettoie-t-elle toujours le drapeau overflow ?

16. Sur le 80386, l'instruction IMUL est presque complètement orthogonale (généralisée). Presque. Donnez certains exemples de formes permises pour ADD qui n'ont pas d'équivalent en IMUL.

17. Pourquoi Intel n'a pas généralisé IDIV comme il avait fait pour IMUL ?

18. De quelle(s) instruction(s) auriez-vous besoin pour lire une valeur de huit bits à l'adresse d'Entrées/Sorties 378h ? Donnez les instructions spécifiques pour faire cela.

19. Quels sont les flags utilisés par le 80x86 pour marquer un dépassement de capacité arithmétique non signé ?

20. Et quels sont les drapeaux correspondants pour un dépassement signé ?

21. Quels flags les systèmes 80x86 utilisent pour marquer les conditions *non signées* suivantes ? Et comment ces drapeaux devraient-ils être pour que la condition soit VRAIE ?

    a) Égal b) Non égal c) Inférieur

    d) Inférieur ou égal e) Supérieur à f) Supérieur ou égal

22. Répétez la dernière question pour des comparaisons *signées*.

23. Expliquez le fonctionnement des instructions CALL et RET. Décrivez pas à pas ce que chaque variante de ces instructions fait.

24. La séquence suivante permute les valeurs des variables I et J :

```
xchg    ax, i
xchg    ax, j
xchg    ax, i
```

    Sur le 80486 les instructions "MOV reg, mem" et "MOV mem, reg" consomment un seul cycle d'horloge (sous les conditions appropriées), alors que l'instruction "XCHG reg, mem" demande trois cycles. Donnez une séquence plus rapide que ci-dessus, appropriée au 80486.

25. Sur le 80386, l'instruction "MOV reg, mem" requiert quatre cycles, "MOV mem, reg" consomme trois cycles et "XCHG reg, mem" en demande cinq. Donnez une séquence plus rapide du problème de permutation présenté à la question 24, appropriée au 80386.

26. Sur le 80486 les instructions "MOV acc, mem"[43] et "MOV reg, mem" consomment seulement un cycle pour s'exécuter (sous les conditions appropriées). En présumant que le reste ne change pas, pourquoi serait-il mieux d'utiliser "MOV acc, mem" au lieu de "MOV reg, mem" pour charger une valeur dans AL/AX/EAX ?

27. Quelles instructions chergent des valeurs de 32 bits sur les processeurs antérieurs au 80386 ?

28. Comment pourriez-vous utiliser PUSH et POP de sorte à préserver la valeur de AX entre deux sections de votre code ?

[43] Où *acc* représente soit AL, soit AX, soit EAX, n.d.t.

29. Si, immédiatement après être entré dans une sous-routine, vous utilisez "pop ax", quelle valeur aurez-vous dans AX ?

30. Quel est l'un des usages principaux de l'instruction SAHF ?

31. Quelle est la différence entre CWD et CWDE ?

32. L'instruction BSWAP permet de convertir entre valeur *big endian* et *little endian* de 32 bits. Quelle instruction pourriez-vous utiliser pour faire la même chose entre valeurs de 16 bits ?

33. Quelle instruction utiliseriez-vous pour convertir entre une valeur *little endian* de 32 bits et une valeur *big endian* de 32 bits ?

34. Expliquez comment utiliser l'instruction XLAT pour convertir un caractère alphabétique majuscule se trouvant dans AL (en supposant qu'il était en minuscules) sans modifier le autres valeurs dans AL.

35. A quelle instruction CMP ressemble-t-elle le plus ?

36. A quelle instruction TEST ressemble-t-elle le plus ?

37. Quelle est la fonction de l'instruction NEG ?

38. Dans quelles deux circonstances les instructions DIV et IDIV échouent ?

39. Quelle est la différence entre RCL et ROL ?

40. Écrivez un court segment de code à l'aide de l'instruction LOOP pouvant appeler la sous-routine "Appellez_Moi" 25 fois.

41. Sur les CPU 80486 et Pentium, l'instruction LOOP n'est pas aussi rapide que ses instructions décomposées qui font la même chose. Récrivez le code de la question 40 pour produire un exécutable plus rapide.

42. Comment détermineriez-vous le "saut opposé" étant donné un saut conditionnel ? Pourquoi est-il le meilleur algorithme ?

43. Donnez un exemple d'instruction BOUND et expliquez ce que l'exemple fait.

44. Quelle est la différence entre les instructions IRET et RET (far) ?

45. L'instruction BT (bit test) copie un bit spécifique dans le drapeau carry. Si ce bit vaut 1 alors le drapeau sera activé et si le bit vaut 0, le drapeau sera nettoyé. Supposez vouloir mettre à zéro le drapeau carry si le bit était à 0 et l'activer dans le cas contraire. Quelle serait l'instruction à exécuter après BT ?

46. Vous pouvez simuler une instruction de retour *far* à l'aide d'une variable de type double word et deux instructions. Quelles seraient ces instructions ?