Opérations arithmétiques et logiques Chapitre 9

Pour programmer en assembleur, juste connaître la fonction de chaque instruction ne suffit pas ; il faut savoir aussi comment les utiliser et les combiner pour produire quelque chose de concret. Beaucoup d'instructions sont utilies pour des opérations qui n'ont rien à voir avec leurs fonctions mathématiques évidentes. Ce chapitre aborde principalement les techniques à utiliser pour convertir les instructions complexes des langages de haut niveau en leurs simples équivalents d'instructions en assembleur. On verra également la façon de produire des opérations mathématiques ou logiques avancées, en incluant les opérations à précision multiple, et les astuces avec diverses instructions.

9.0 Vue d'ensemble du chapitre

Ce chapitre fera l'objet de six sujets principaux : conversion en assembleur d'expressions arithmétiques de haut niveau, les expressions logiques, les expressions arithmétiques et logiques en précision étendue, les opérations logiques, le travail avec des opérandes de différentes tailles, les idiomes machine et arithmétique et, finalement, les opérations de masquage. Tout comme les chapitres précédents, il couvre une matière considérable que vous pourriez ne pas avoir besoin d'assimiler intégralement si vous êtes débutant. Les éléments de la liste suivante marqués par le préfixe "•" sont essentiels. Les sections avec un "❑" discutent de sujets avancés que vous pouvez mettre de côté pour l'instant.

- • Expressions arithmétiques
- • Affectations simples
- • Expressions simples
- • Expressions complexes
- • Opérateurs commutatifs
- • Expressions logiques
- • Opérations en précision multiple
- • Opérations d'addition en précision multiple
- • Opérations de soustraction en précision multiple
- • Comparaisons en précision étendue
- ❑ Multiplicatons en précision étendue
- ❑ Divisions en précision étendue
- ❑ Négations en précision étendue
- • AND, OR, XOR et NOT en précision étendue
- ❑ Opérations de rotation et de décalage en précision étendue
- ❑ Travailler avec des opérands de différentes tailles
- • Multiplication sans MUL ni IMUL
- ❑ Division sans DIV ni IDIV
- ❑ Utiliser AND pour calculer des restes
- ❑ Compteurs modulo-n avec AND
- ❑ Tester pour 0FFFFF…FFFh
- • Opérations de test
- ❑ Tester le signe avec des instructions XOR
- ❑ Opérations de masquage
- ❑ Masquer avec AND
- ❑ Masquer avec OR
- ❑ Compacter et décompacter des types de données
- ❑ Tables de correspondance

Aucun de ces sujets est particulièrement difficile à comprendre. Il y a cependant beaucoup de sujets nouveaux et s'y attarder un peu pourra certainement vous aider à les assimiler mieux. Ce qui est marqué par "•" décrivent les opérations que vous effectuerez fréquemment ; c'est donc une bonne idée de commencer par elles.

9.1 Expressions arithmétiques

Probablement, le premier choc qu'ont les débutants en assembleur est le manque d'intuitivité dans les expressions arithmétiques. Dans la plupart des langages de haut niveau, celles-ci ressemblent beaucoup plus à leur équivalent mathématique :

```
x:=y*z
```

Mais, en assembleur, on aura besoin de plusieurs instructions pour réaliser la même tâche :

```
mov     ax, y
imul    z
mov     x, ax
```

Certainement, la première version est beaucoup plus facile à taper, à lire et à comprendre. C'est l'une des raisons principales - plus que d'autres - pour lesquelles les gens préfèrent les langages de haut niveau.

Malgré le fait que cela comporte plus de frappes, convertir une expression arithmétique en son équivalent en assembleur n'est pas difficile du tout. En considérant le problème par étapes, comme si on devait le résoudre manuellement, on peut parfaitement arriver à la séquence appropriée d'instructions avec toute expression mathématique. En apprenant à convertir ces expressions en assembleur en trois étapes, vous découvrirez que ce n'est pas aussi difficile.

9.1.1 Affectations simples

Les expressions les plus faciles à convertir en assembleur sont les affectations. Elles copient une valeur individuelle dans une variable et prennent l'une des deux formes suivantes :

```
variable := constante
```

ou

```
variable := variable
```

Si la variable se trouve dans le même segment de données (par exemple DSEG), convertir le premier des deux exemples est très facile, vous n'auriez à utiliser que cette instruction :

```
mov     variable, constante
```

qui copie la constante dans la variable.

Le second exemple, est un peu plus compliqué, car l'architecture 80x86 ne fournit pas une instruction mov de type "mémoire à mémoire". Par conséquent, pour produire cet effet, il vous faudra passer la valeur de la variable par un registre. Si vous observez l'encodage de l'instruction mov dans le glossaire, vous remarquerez que mov ax, memory et mov memory, ax sont plus courtes que les instructions impliquant d'autres registres. Par conséquent, si le registre ax est disponible, il vous faudra le préférer aux autres pour toute opération de ce genre. Par exemple :

```
var1 := var2
```

devient :

```
mov     ax, var2
mov     var1, ax
```

Bien sûr, si vous êtes en train d'utiliser le registre ax pour autre chose, un autre registre suffirait. En tout cas, il vous faudra utiliser un registre pour passer la valeur d'une variable.

Cette discussion, bien sûr, suppose que les deux variables sont déjà en mémoire. Vous devriez essayer d'utiliser un registre pour garder la valeur d'une variable toutes les fois qu'il vous sera possible.

9.1.2 Expressions simples

Le niveau suivant de complexité est représenté par les expressions simples. Elles prennent la forme :

```
var := term1 op term2;
```

où var est une variable, $term_1$ et $term_2$ sont des variables ou des constantes et op est un opérateur arithmétique quelconque (addition, soustraction, multiplication et ainsi de suite).

La plupart des expressions simples prennent cette forme. Et le fait que l'architecture 80x86 ait été optimisée spécifiquement pour ce type d'expression ne devrait pas surprendre.

Une conversion typique de ces expressions prend la forme suivante :

```
        mov     ax, term1
        op      ax, term2
        mov     var, ax
```

où op est le mnémonique de n'importe quelle instruction correspondant à une expression arithmétique (par exemple, "+" = add, "-" = sub, etc.).

Néanmoins, vous devriez être conscient de certaines exceptions. Avant tout, l'instruction {i}mul n'admet pas d'opérandes immédiates sur des processeurs antérieurs au 80286. De plus, aucun processeur n'admet d'opérande immédiate avec {i}div. Par conséquent si vous travaillez avec des multiplications ou des divisions et une des deux opérandes est une valeur constante, il vous faudra charger cette constante dans un registre ou dans un emplacement de mémoire avant de multiplier ou de diviser ax par cette valeur. Il va sans dire que si vous utilisez des instructions de multiplication ou de division sur des processeurs comme le 8088 ou le 8086, il vous faudra utiliser les registres ax et dx. Vous ne pouvez pas utiliser des registres arbitraires comme vous feriez avec d'autres instructions. N'oubliez non plus d'utiliser des instructions d'extension signée si vous divisez un nombre de 16/32 bits par un autre de la même taille. Et, finalement, n'oubliez pas que certaines instructions peuvent provoquer un dépassement de capacité. Il vous faudra donc tester également de tels cas après une expression arithmétique.

Voici des exemples d'expressions simples :

```
;X := Y + Z;
        mov     ax, y
        add     ax, z
        mov     x, ax

;X := Y - Z;
        mov     ax, y
        sub     ax, z
        mov     x, ax

;X := Y * Z; {non signée}
        mov     ax, y
        mul     z       ; Pour l'arithmétique signée, on utilise plutôt imul
        mov     x, ax   ; N'oubliez pas que ceci affecte DX

;X := Y div Z; {div non signée}
        mov     ax, y
        mov     dx, 0   ; Extension de zéros AX --> DX
        div     z
        mov     x, ax

;X := Y div Z; {div signée}
        mov     ax, y
        cwd             ; Extension signée AX --> DX
        idiv    z
        mov     x, ax

;X := Y mod Z; {reste non signé}
        mov     ax, y
        mov     dx, 0   ; Extension de zéros AX --> DX
        div     z
        mov     x, dx   ; Le reste était dans DX

;X := Y mod Z; {reste signé}
        mov     ax, y
        cwd             ; Extension signée AX --> DX
        idiv    z
        mov     x, dx   ; Le reste était dans DX
```

Puisqu'il peut se produire une erreur arithmétique, il faudrait généralement tester le résultat de chaque expression avant ou après avoir complété l'opération. Par exemple, l'addition non signée, la soustraction et la multiplication affectent le drapeau carry si un dépassement de capacité se produit. Vous pouvez utiliser les instructions jc et jnc pour tester un

dépassement immédiatement après l'instruction correspondante. De même, vous pouvez utiliser les insstructions jo ou jno après ces séquences pour tester un dépassement arithmétique. Les deux exemples suivants montrent comment le faire pour l'instruction add:

```
X := Y + Z; {non signé}
        mov     ax, y
        add     ax, z
        mov     x, ax
        jc      uOverflow

X := Y + Z; {signé}
        mov     ax, y
        add     ax, z
        mov     x, ax
        jc      sOverflow
```

Certains opérateurs unaires rentrent dans la catégorie des simples expressions. Un bon exemple de ce genre est la négation unaire. Dans un langage de haut niveau, la négation peut prendre l'une des formes suivantes :

```
var := -var             ou              var1 := -var2
```

Notez que var := -constante est en fait une affectation simple et non une expression simple. Vous pouvez spécifier une constante négative comme opérande de l'instruction mov :

```
        mov     var, -14
```

Pour obtenir le même résultat qu'on obtient avec l'instruction ci-dessus, utilisez cette instruction unique :

```
        neg     var
```

Si deux variables différentes sont concernées, utilisez :

```
        mov     ax, var2
        neg     ax
        mov     var1, ax
```

Un dépassement de capacité se produira seulement si vous niez la valeur la plus négative (à savoir, -128 pour une valeur de 8 bits, -32768 pour une valeur de seize bits, etc.). Dans cette éventualité, le 80x86 active le drapeau overflow, de façon à vous permettre de tester ce cas en utilisant jo ou jno. Dans tous les autres cas, le 80x86 met à zéro le drapeau overflow. Le drapeau carry n'a pas de signification après une instruction neg, car neg (logiquement) ne s'applique pas à des opérandes non signées.

9.1.3 Expressions complexes

Une expression complexe est toute expression arithmétique impliquant plus de deux termes et un opérateur. De telles expressions se voient souvent dans des programmes écrits avec des langages de haut niveau. Ces expressions peuvent inclure des parenthèses pour modifier la précédence des opérateurs, des appels de fonctions, des accès à des tableaux, etc. Si certaines d'elles sont tout à fait simples à convertir, d'autres requièrent un certain effort. Cette section décrit dans les grandes lignes, les règles à utiliser.

Des expressions complexes faciles à convertir sont celles qui impliquent trois termes et deux opérandes, par exemple :

```
        W := W - Y - Z;
```

en toute évidence, la conversion dans ce cas requiert deux instructions sub. Cependant, même avec une expression aussi simple, la conversion n'est pas toujours évidente. Il y a en effet deux manières de convertir ceci en assembleur :

```
        mov     ax, w
        sub     ax, y
        sub     ax, z
        mov     w, ax
```

et

```
        mov     ax, y
        sub     ax, z
        sub     w, ax
```

La seconde conversion, étant elle plus courte, paraît plus élégante. Cependant, elle produit un résultat incorrect (en supposant que l'original a une syntaxe de type Pascal). Le problème est l'associativité. La séconde séquence convertie produit en fait : W := W - (Y - Z), ce qui n'est pas la même chose que W := (W - Y) - Z. La façon dont on place les parenthèses autour de la sous-expression influit sur le résultat. Notez que si vous voulez une séquence plus courte, vous pouvez plutôt utiliser ce qui suit :

```
        mov     ax, y
        add     ax, z
        sub     w, ax
```

Ce qui produit W := W - (Y + Z) qui équivaut à W := (W - Y) - Z.

La précédence constitue un autre problème. Considérez l'expression Pascal suivante :

```
        X := W * Y + Z;
```

Encore une fois, il y a deux manières d'évaluer cette expression :

```
        X := (W * Y) + Z;
```

et

```
        X := W * (Y + Z);
```

Maintenant vous commencerez peut-être à penser que ce texte est fou. Chacun sait que la manière correcte d'évaluer l'expression est la seconde dans les deux exemples. Cependant, vous avez tort de penser de cette façon. Le langage APL, par exemple, évalue les expressions uniquement de droite à gauche et ne définit aucune espèce de précedence des opérateurs.

La plupart des langages de haut niveau utilisent un jeu prédéfini de règles de précédence afin de décrire l'ordre d'évaluation dans une expression ayant deux opérateurs différents ou plus. Ils effectuent, par exemple, la multiplication et la division avant l'addition et la soustraction. Ceux qui offrent l'élévation à la puissance (FORTRAN et BASIC), effectuent cette opération en premier avant la multiplication et la division. Ces règles sont intuitives, car presque tout le monde les apprend avant d'entrer à l'école secondaire. Considérez l'expression :

```
        X op1 Y op2 Z
```

Si op_1 prévaut sur op_2, alors l'évaluation sera (X op_1 Y) op_2 Z, sinon, si op_2 est prioritaire par rapport à op_1, l'expression sera évaluée X op_1 (Y op_2 Z). Selon les opérateurs et les opérands impliqués, ces deux calculs pourraient produire des résultats différents.

En convertissant une expression de cette forme en langage assembleur, vous devez vous assurer d'évaluer d'abord la sous-expression avec la plus haute précédence. L'exemple suivant, montre cette technique :

```
; W := X + Y * Z;
        mov     bx, x
        mov     ax, y   ; Doit calculer d'abord Y * Z, car * prévaut sur +
        mul     z
        add     bx, ax  ; C'est maintenant qu'on additionne le produit avec la
        mov     w, bx   ; valeur de X
```

Puisque l'addition est une opération *commutative*, on pourrait optimiser le code ci-dessus pour produire :

```
; W := X + Y * Z;
        mov     ax, y   ; Doit calculer d'abord Y * Z, car * prévaut sur +
        mul     z
        add     ax, x   ; C'est maintenant qu'on additionne le produit avec la
                        ; valeur de X
        mov     w, ax
```

Si deux opérateurs à l'intérieur d'une même expression ont la même priorité, alors vous déterminerez l'ordre d'évaluation en suivant les règles de l'*associativité*. La plupart des opérateurs ont une *associativité par la gauche*, c'est-à-dire qu'il évaluent une expression en procédant de gauche à droite. L'addition, la soustraction, la multiplication et la division sont toutes des opérations ayant une associativité de gauche. Alors qu'un opérateur ayant une *associativité de droite* évalue de droite à gauche. L'exponentation en FORTRAN et BASIC est un bon exemple d'associativité par la droite :

```
2^2^3 est égal à 2^(2^3) et non à (2^2)^3
```

Les règles de la précédence et de l'associativité déterminent l'ordre d'évaluation des expressions. Indirectement, ces règles vous indiquent où placer les parenthèses dans une expression pour déterminer l'ordre d'évaluation. Vous pouvez certainement utiliser les parenthèses pour imposer un certain ordre d'évaluation. Cependant, le concept clé est que l'assembleur doit compléter certaines opérations avant d'autres pour calculer correctement la valeur d'une expression donnée. Les exemples suivants montrent ce principe :

```
; W := X - Y - Z;
        mov     ax, x   ; Tous les termes ont le même opérateur, donc la même
                        ; précédence.
        sub     ax, y   ; Ce qu'il faut donc faire est de les évaluer de gauche
        sub     ax, z   ; à droite.
        mov     w, ax

; W := X + Y * Z;
        mov     ax, y   ; Il faut calculer d'abord Y * Z, car la multiplication
        imul    z       ; prévaut sur l'addition et ensuite, évaluer l'expression
        add     ax, x   ; résultante de gauche à droite.
        mov     w, ax

; W := X / Y - Z;
        mov     ax, x   ; De même, il faut effectuer d'abord la division.
        cwd
        idiv    y
        sub     ax, z
        mov     w, ax

; W := X * Y * Z;
        mov     ax, y   ; L'addition et la multiplication sont commutatives, et
        imul    z       ; l'ordre d'évaluation n'a pas d'importance.
        imul    x
        mov     w, ax
```

Il y a une exception à la règle de l'associativité. Si une expression contient la multiplication et la division, c'est mieux d'effectuer d'abord la multiplication. Par exemple supposez une expression comme :

```
W := X/Z * Z
```

Il vaut mieux calculer X*Z et ensuite diviser le résultat par Y plutôt que diviser X par Y et multiplier le quotient par Z. Il y a deux raisons à cela. Avant tout, rappelez-vous que l'instruction imul produit toujours un résultat de 32 bits (en supposant des opérandes de 16 bits). En effectuant la multiplication en premier, vous *effectuez automatiquement* une extension signée du produit dans le registre dx sans avoir besoin d'utiliser l'instruction cwd. La seconde raison est obtenir une meilleure précision de calcul. Rappelez-vous que les divisions entières produisent souvent des résultats incorrects. Par exemple, si vous effectuez 5/2, vous obtiendrez 2 et non 2.5. Calculer (5/2)*3 produirait 6. Cependant, calculer (5*3)/2 vous donnera la valeur 7, qui est plus près de 7.5, le véritable quotient. Par conséquent, si vous avez affaire à une expression de la forme

```
W := X/Z * Z
```

vous pouvez la convertir en :

```
        mov     ax, x
        imul    z
        idiv    y[1]
        mov     w, ax
```

Certes, si l'algorithme que vous voulez réaliser en assembleur dépend de l'effet de l'arrondissement, vous ne pouvez pas vous servir de cette astuce. Morale de l'histoire : assurez-vous toujours de comprendre pleinement toute expression que vous convertissez. Bien sûr, si votre algorithme exige la division en premier, allez-y.

Considérez l'instruction Pascal suivante :

```
W := X - Y * Z;
```

Ceci est semblable à un exemple précédent, mais on utilise la soustraction au lieu de l'addition. Puisque la soustraction n'est pas commutative, on ne peut pas calculer Y * Z et ensuite soustraire X. Ce qui complique un peu la conversion. Au

[1] Le texte original mentionne `idiv z` ici, mais c'est évidemment incorrect, ndt.

lieu d'une simple séquence de multiplication et d'addition, il faut charger X dans un registre, multiplier Y et Z, en laissant le résultat dans un registre différent et finalement soustraire ce produit de X, par exemple :

```
mov     bx, x
mov     ax, y
imul    z
sub     bx, ax
mov     w, bx
```

Ceci est un exemple banal qui montre la nécessité des *variables temporaires* dans une expression. Ce code utilise le registre bx pour garder temporairement une copie de X pendant qu'on calcule la multiplication entre Y et Z. Au fur et à mesure que votre expression devient plus complexe, le besoin des variables temporaires grandit. Considérez l'exemple suivant :

```
W := (A + B) * (Y + Z);
```

En suivant les règles normales de l'évaluation algébrique, vous effectuerez en premier les expressions qui se trouvent à l'intérieur des parenthèses (autrement dit, les deux expressions avec la plus haute précédence) et vous mettrez leurs résultats à part. Une fois calculés les résultats des deux sous-expressions, on pourra finalement les multiplier. Une manière de travailler avec des expressions complexes est de les réduire en séquences d'expressions simples et stocker les résultats dans des variables temporaires. Par exemple, on peut convertir l'expression ci-dessus en la séquence qui suit :

```
Temp1 := A + B;
Temp2 := Y + Z
W := Temp1 * Temp2
```

Puisque convertir des expressions simples est toujours facile, on peut utiliser cette technique avec n'importe quelle expression, aussi complexe soit-elle. Le code correspondant à ce qu'on vient de voir est :

```
mov     ax, a
add     ax, b
mov     Temp1, ax
mov     ax, y
add     ax, z
mov     Temp2, ax
mov     ax, Temp1
imul    Temp2
mov     w, ax
```

Certes, ce code n'est pas très efficace, car il requiert la déclaration de deux variables temporaires dans votre segment de données, mais, on peut l'optimiser très facilement, en utilisant des registres comme variables temporaires, autant que possible. Avec les registres, le code ci-dessus devient :

```
mov     ax, a
add     ax, b
mov     bx, y
add     bx, z
imul    bx
mov     w, ax
```

Encore un autre exemple :

```
X := (Y + Z) * (A - B) / 10;
```

Ceci peut être converti en un ensemble d'expressions simples :

```
Temp1 := (Y + Z);
Temp2 := (A - B);
Temp1 := Temp1 * Temp2
X := Temp1 / 10
```

Ce qui devient en assembleur :

```
mov     ax, y   ; AX := Y+Z
add     ax, z
mov     bx, a   ; BX := A-B
sub     bx, b
```

```
mul     bx      ; AX := AX * BX, qui produit l'extension signée de AX dans DX
mov     bx, 10
idiv    bx      ; AX := AX / 10
mov     x, ax
```

La chose la plus importante à garder à l'esprit est que les variables temporaires devraient, autant que possible, être des registres, car l'accès aux registres est plus efficace que l'accès aux variables en mémoire. Utilisez les variables seulement si vous êtes à court de registres.

En dernier, convertir une expression complexe en assembleur est un peu différent que résoudre une expression à la main. Au lieu de calculer le résultat de chaque étape, vous écrivez simplement les instructions assembleur qui calculent ces résultats. Puisque vous avez appris à effectuer une opération à la fois, vous savez que le calcul manuel fonctionne avec des "expressions simples" qui composent des expressions complexes. Certainement, convertir ces expressions en assembleur est un jeu d'enfant. Par conséquent, n'importe qui capable de résoudre une expression complexe en à la main peut aussi la convertir en assembleur en suivant les règles des expressions simples.

9.1.4 Opérateurs commutatifs

Si le symbole "@" représente un opérateur quelconque, cet opérateur est commutatif si la relation suivante est toujours vraie :

```
(A @ B) = (B @ A)
```

Comme vous avez vu dans la section précédente, les opérateurs commutatifs sont commodes parce que l'ordre d'évaluation de leurs opérands est immatériel, ce qui vous permet d'arranger un calcul selon vos commodités et utiliser ainsi moins de variables temporaires. Quand vous avez un opérateur commutatif dans une expression donnée, vous devez toujours observer s'il existe une meilleure séquence pour améliorer la taille ou la vitesse de votre code. Les tables suivantes montrent une liste des opérateurs commutatifs et non commutatifs que vous trouvez le plus souvent dans les langages de haut niveau :

Tableau 46: Les opérateurs commutatifs les plus communs

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& ou &	AND logique ou de bit à bit
OR	|| ou |	OR logique ou de bit à bit
XOR	^	OR exclusif logique en Pascal et bit à bit seulement en C/C++
=	==	Égalité
<>	!=	Inégalité

Tableau 47: Les opérateurs non commutatifs les plus communs

Pascal	C/C++	Description
-	-	Soustraction
/ ou DIV	/	Division
MOD	%	Modulo (reste de division)
<	<	Inférieur à
<=	<=	Inférieur ou égal à
>	>	Supérieur à
>=	>=	Supérieur ou égal à

9.2 Expressions logiques (booléennes)

Considérez l'expression suivante dans un programme Pascal :

```
B := ((X=Y) and (A <= C)) or ((Z-A) <> 5);
```

où B est une variable booléenne et toutes les autres variables sont des entiers.

Comment représente-t-on les variables booléennes en assembleur ? Malgré qu'il suffit un bit pour représenter une telle valeur, la plupart des programmeurs allouent un octet ou un mot pour cette tâche. Avec un octet, il y 256 valeurs différentes qu'on peut utiliser pour représenter les deux valeurs booléennes *true* et *false*. Alors, quelle paire de valeurs (ou d'ensemble de valeurs) pourra-t-on utiliser ? À cause de l'architecture elle-même, c'est naturel de penser à des valeurs comme zéro ou non-zéro, positif ou négatif, et non deux valeurs booléennes quelconques. La plupart des programmeurs (et même certains langages de programmation, comme le C) choisissent zéro pour représenter faux et toute autre valeur pour représenter vrai. Certains préfèrent représenter "vrai" et "faux" avec 1 et 0 (respectivement) sans permettre aucune autre valeur. D'autres sélectionnent 0FFFFh pour vrai et 0 pour faux. Vous pourriez aussi utiliser une valeur positive pour représenter le vrai et une valeur négative pour représenter le faux. Toutes ces conventions ont leurs avantages et leurs désavantages.

N'utiliser que 0 et 1 peut offrir un grand avantage : les instruction logiques 80x86 (`and`, `or`, `xor` et moins particulièrement `not`) dépendent exactement de ces valeurs. C'est-à-dire, si vous avez deux variables booléennes A et B, alors les instructions suivantes effectuent sur ces variables les opérations logiques suivantes :

```
mov     ax, A
and     ax, B
mov     C, ax           ; C := A and B

mov     ax, A
or      ax, B
mov     C, ax           ; C := A or B

mov     ax, A
xor     ax, B
mov     C, ax           ; C := A xor B

mov     ax, A           ; Notez que l'instruction NOT ne calcule pas
not     ax              ; correctement B := not A. Autrement dit,
and     ax, 1           ; NOT 0 n'est pas égal à 1.
mov     B, ax           ; B := not A

mov     ax, A           ; Une autre façon d'effectuer B := NOT A
xor     ax, 1
mov     B, ax           ; B := not A
```

Comme noté dans le commentaire ci-dessus, l'instruction `not` n'effectuera pas correctement une négation logique. En effet, NOT 0 c'est 0FFh et NOT 1 c'est 0FEh. Aucun des deux résultats ne donne 0 ou 1. Néanmoins, en faisant un AND logique sur le résultat, on obtient la valeur attendue. Notez que vous pouvez implémenter l'opération *not* de façon plus efficace en utilisant `xor ax, 1`, car elle n'affecte que le bit le moins significatif.

Il résulte évident qu'utiliser 0 pour faux et toute autre valeur pour vrai a beaucoup d'avantage subtils, précisement parce que le test de vérité est souvent implicit dans l'exécution de n'importe quelle instruction logique. Cependant, ce mécanisme a aussi un grand désavantage : vous ne pouvez pas utiliser les instructions 80x86 `and`, `or`, `xor` et `not` pour implémenter les opérations logiques ayant le même nom. Considérez les deux valeurs 55h et 0AAh. Elles sont toutes les deux

différentes de zéro et devraient les deux représenter la valeur *vrai*. Cependant, si vous faites un AND logique entre 55h et 0AAh en utilisant le and du 80x86, le résultat sera zéro, cependant [*vrai* AND *vrai*] devrait produire *vrai* et pas *faux*. Un système qui utilise 0 pour représenter faux et toute autre valeur pour représenter vrai est un *système logique arithmétique*. Un système qui utilise deux valeurs distinctes comme *zéro* et *un* pour représenter faux et vrai est un *système logique*, ou simplement un *système booléen*. Vous pouvez utiliser les deux systèmes, selon votre convénance. Considérez encore l'expression booléenne :

```
B := ((X = Y) and (A <= C)) or ((Z - A) <> 5);
```

Cette expression complexe pourrait être traduite en expressions simples, comme les suivantes :

```
Temp2 := X = Y
Temp1 := A <= D
Temp1 := Temp1 and Temp2
Temp2 := Z - A
Temp2 := Temp2 <> 5
B := Temp1 or Temp2
```

Et le code assembleur résultant serait :

```
                mov     ax, x           ; Tester si X = Y et charger 0 ou 1 dans AX
                cmp     ax, y           ; pour indiquer le résultat de cette
                                        ; comparaison.
                jnz     L1
                mov     al, 1           ; X = Y
                jmp     L2
L1:             mov     al, 0           ; X <> Y
L2:
                mov     bx, A           ; Tester si A <= D et charger 0 ou 1 dans BX
                                        ; pour cmp      bx, D indiquer le résultat de
                                        ; cette comparaison.
                jle     ST1
                mov     bl, 0
                jmp     L3
ST1:            mov     bl, 1
L3:
                and     bl, al          ; Temp1 := Temp1 and Temp2

                mov     ax, Z           ; Tester si (Z-A) <> 5.
                sub     ax, A           ; Temp2 := Z-A;
                cmp     ax, 5           ; Temp2 := Temp2 <> 5;
                jnz     ST2
                mov     al, 0
                jmp     short L4
ST2:            mov     al, 1
L4:
                or      al, bl          ; Temp1 := Temp1 or Temp2;
                mov     B, al           ; B := Temp1;
```

Come vous pouvez voir, cette séquence d'instructions est plutôt complexe. On pourrait l'optiomiser en faisant des hypothèses sur les résultats de ces comparaisons et en définissant des valeurs par défaut, comme suit :

```
                mov     bl, 0           ; En supposant que (X <> Y)
                mov     ax, x
                cmp     ax, y
                jne     L1
                mov     bl, 1           ; X = Y, donc c'est vrai
L1:
                mov     bh, 0           ; En supposant que (A <= D)
                mov     ax, A
                cmp     ax, D
                jnle    L2
                mov     bh, 1           ; A <= D, donc c'est vrai
L2:
                and     bl, bh          ; AND logoque sur les résultats
```

```
                mov     bh, 0           ; En supposant que (Z-A) = 5
                mov     ax, Z
                sub     ax, A
                cmp     ax, 5
                je      L3
                mov     bh, 1           ; (Z-A) <> 5
L3:
                or      bl, bh          ; OR logique sur les résultats
                mov     B, bl           ; Enregistrer le résultat
```

Naturellement, si vous pouvez programmer sur un processeur 80386 ou supérieur, vous pourriez simplifier le code encore plus via les instructions set*cc* :

```
                mov     ax, x
                cmp     ax, y
                sete    al              ; Temp2 := X = Y

                mov     bx, A
                cmp     bx, D
                setle   bl              ; Temp := A <= D
                and     bl, al          ; Temp := TEMP AND TEMP2
                mov     ax, Z
                sub     ax, A           ; Temp2 := Z-A;
                cmp     ax, 5           ; Temp2 := Temp2 <> 5;
                setne   al
                or      bl, al          ; Temp := Temp or Temp2;
                mov     B, bl           ; B := Temp;
```

Cette dernière séquence est évidemment beaucoup mieux que la précédente, mais elle ne s'exécutera que sur les processeurs 80386 ou supérieurs.

Une autre manière de coder des expressions booléennes est de représenter les valeurs booléennes par des états à l'intérieur de votre code. L'idée de base est de ne pas stocker l'état de l'expression dans des variables temporaires, c'est la construction même du code qui nous suggère cet état ou mieux, la position à l'intérieur du code. Considérez l'implémentation suivante de l'expression ci-dessus. D'abord, arrengeons-la pour qu'elle soit :

```
B := ((Z - A) <> 5) or ((X = Y) and (A <= d));
```

Ce qui est parfaitement correct, car l'opération or est commutative. Maintenant, considérez l'implémentation suivante :

```
                mov     B, 1            ; En supposant que le résultat est vrai
                mov     ax, Z           ; (Z-A) <> 5 ?
                sub     ax, A           ; Si cette condition est vraie, le résultat
                cmp     ax, 5           ; est toujours vrai et il n'y a pas besoin
                jne     Done            ; de passer le reste du programme.

                mov     ax, X           ; Si X <> Y, le résultat est faux, peu importe
                cmp     ax, Y           ; le contenu de A et D.
                jne     SetBtoFalse

                mov     ax, A           ; Maintenant, tester si A <= D.
                cmp     ax, D
                jle     Done            ; Si c'est le cas, quitter.
SetBtoFalse:    mov     B, 0            ; Si B est faux, traiter le cas ici
Done:
```

Notez que cette section de code est beaucoup plus courte que la première version (et qui exécute sur tous les processeurs). Les implémentations précédentes sont toutes en mesure de calculer l'expression, mais cette dernière exploite la logique du flux du programme pour améliorer le code. Ce code commence par supposer un résultat vrai et par initialiser la variable B à vrai. Ensuite, il teste si (Z-A) <> 5. Si c'est vrai, le contrôle saute à l'étiquette Done, car B est vrai et la condition OR est accomplie et peu importe le résultat du reste de l'expression. Si contrôle passe à l'instruction mov ax, X, on sait que le résultat de la comparaison précédente donne FALSE. Aucun besoin d'enregistrer ce résultat dans une variable temporaire, puisqu'on connaît implicitement cette valeur par le fait même qu'on est en train d'exécuter la ligne mov ax, X. De même, le deuxième groupe d'instructions teste si X est égal à Y. Si non, on sait déjà que le résultat est FALSE, car le contrôle saute sur l'étiquette SetBtoFalse. Si le programme commence à exécuter le troisième et dernier groupe d'instructions, on sait que le premier résultat a donné FALSE et que le second a donné TRUE. La position même du

code le garantit. Par conséquent, il n'y a pas besoin de déclarer des variables temporaires pour garder l'état de l'expression.

Considérez un autre exemple :

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

L'allure de l'expression suggère une quantité considérable de code. Cependant, en exploitant l'idée du flux de contrôle, ce code se réduit à ceci :

```
                mov     b, 0            ; Supposer que le résultat est faux.
                mov     ax, a           ; Tester si A = E.
                cmp     ax, e
                je      Test2           ; Si c'est le cas, le premier terme est vrai.

                mov     ax, f           ; Sinon, tester la deuxième expression
                cmp     ax, d           ; pour voir si F <> D.
                je      Done            ; Si oui, sauter à Done, sinon continuer.
Test2           mov     ax, a           ; A <> B ?
                cmp     ax, b
                jne     SetBto1         ; Si oui, notre résultat final est TRUE.

                mov     ax, f           ; Si non, tester F = D
                cmp     ax, d
                jne     Done

SetBto1:        mov     b, 1
Done:
```

Voici une autre différence entre l'utilisation du flux de contrôle et l'utilisation de la simple logique de calcul : en utilisant la première méthode, vous pouvez sauter la plupart des instructions implémentées dans la formule booléenne. Ceci est connu sous le terme d'*évaluation de court circuit*. En utilisant le modèle de calcul logique, même avec les instructions **setcc**, il faut passer par toutes les étapes. Gardez à l'esprit que ceci n'est pas nécessairement un désavantage. Sur un processeur à pipelines, le fait d'exécuter plusieurs instructions supplémentaires et ainsi ne pas vider le pipeline et la queue de préchargement peut se révéler beaucoup plus rapide. La meilleure solution dépend toujours des exigences de votre code.

En travaillant avec des expressions booléennes, n'oubliez pas que vous pouvez améliorer votre code en simplifiant ces dernières (voir le chapitre 2). Vous pouvez utiliser des transformations algébriques (spécialement les théorèmes de DeMorgan) et la méthode du mapping pour réduire considérablement la complexité d'une expression.

9.3 Opérations en multiprécision

Un grand avantage de l'assembleur sur les langages de haut niveau est qu'il ne limite pas la taille des entiers. Par exemple, le langage C définit un maximum de trois différentes tailles d'entiers : **short int**, **int** et **long int**. Sur le PC, ces derniers sont souvent des entiers de 16 ou de 32 bits. Bien qu'une machine 80x86 vous limite à des entiers de 8, de 16 ou de 32 bits avec une seule instruction, vous pouvez toujours utiliser plus d'une instruction pour travailler avec des entiers de toute taille désirée. Si vous voulez des entières de taille de 256 bits, pas de problème. La section suivante décrit comment élargir diverses opérations logiques et arithmétiques au-delà des plages de 16, de 32 et de 64 bits.

9.3.1 Opérations d'addition en multiprécision

L'instruction **add** additionne deux nombres de 8, de 16 ou de 32 bits[2]. Après l'exécution de cette instruction, le drapeau de retenue est activé s'il se produit un dépassement de capacité en dehors du bit le plus significatif de la somme.

[2]Comme toujours, l'arithmétique de 32 bits n'est disponible qu'à partir du processeur 80386.

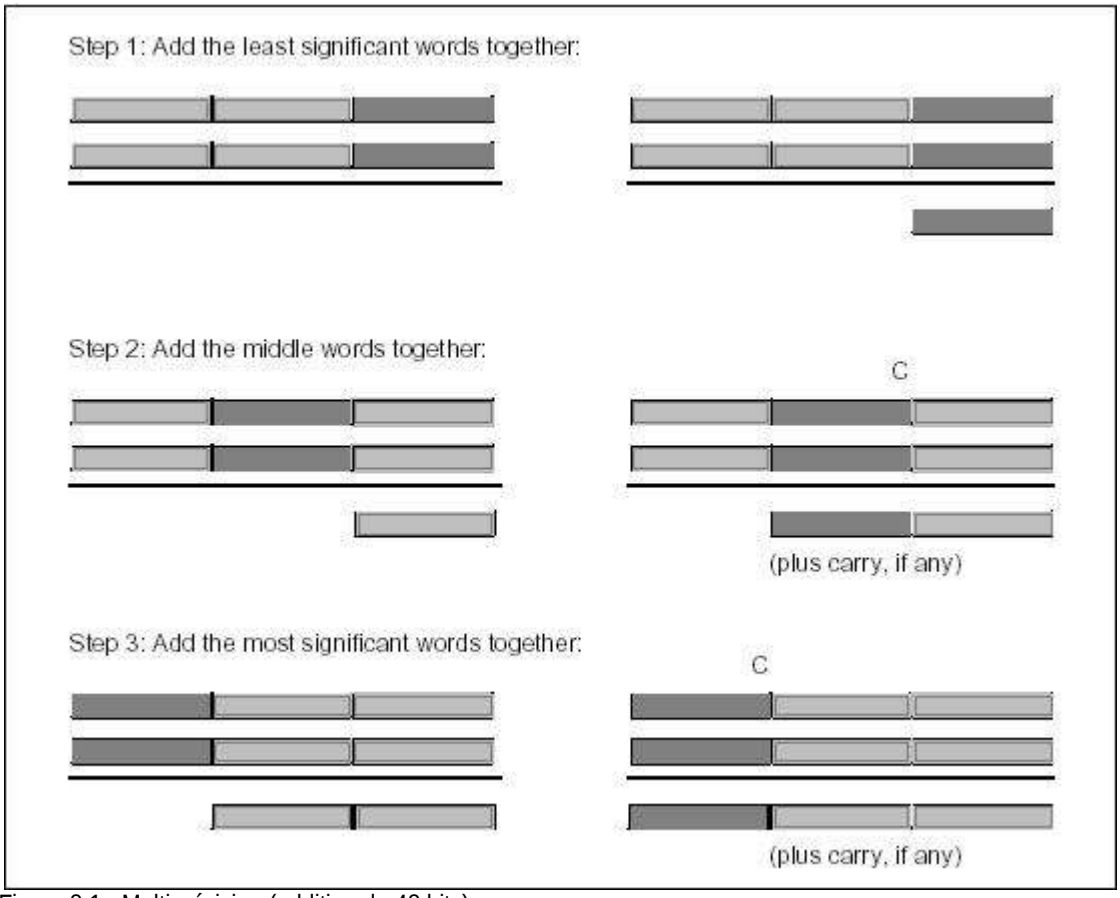


Figure 8.1 - Multiprécision (addition de 48 bits)

Considérez la manière habituelle d'effectuer une telle opération :

Étape 1 : additionner d'abord les chiffres moins significatifs :

```
289                     289
+456    produit        +456
----                   ----
5 avec reste de 1.
```

Étape 2: Additionner les prochains chiffres les plus significatifs, plus la retenue :

```
1 (retenue précédente)
289                     289
+456    produit         +456
----                    ----
   5                      45 avec reste de 1.
```

Étape 3: Additionner les chiffres les plus significatifs, plus la retenue :

```
1 (retenue précédente)
289                     289
+456    produit         +456
----                    ----
45                      745
```

Les machines 80x86 gèrent les opérations arithmétiques en multiprécision de façon identique, mais au lieu d'additionner les nombres un chiffre à la fois, elles additionnent les nombres un octet ou un mot à la fois. Considérez l'addition de 48 bits de la figure 8.1. L'instruction **add** additionne d'abord les mots les moins significatifs. L'instruction **adc** (add with carry) additionne toutes les autres paires de mots, car cette instruction additionne deux opérandes plus le drapeau de retenue en produisant une valeur d'un mot et une (possible) retenue.

Par exemple, supposez avoir deux valeurs de 32 bits à additionner que vous définissez comme suit :

```
X               dword   ?
Y               dword   ?
```

Supposez aussi vouloir stocker le résultat dans une troisième variable, Z, définie avec la directive dword. Le code suivant fait cette besogne :

```
        mov     ax, word ptr X
        add     ax, word ptr Y
        mov     word ptr Z, ax
        mov     ax, word ptr X+2
        adc     ax, word ptr Y+2
        mov     word ptr Z+2, ax
```

Rappellez-vous que ces variables sont déclarées avec la directive dword, par conséquent l'assembleur n'accepterait pas une instruction de la forme mov ax, X parce que ce serait une tentative de charger une valeur de 32 bits dans un registre de 16. Ce code utilise donc la coercition word ptr pour forcer les symboles X, Y et Z à être de 16 bits. Les premières trois instructions additionnent les mots les moins significatifs de X et de Y et gardent les résultats dans le mot le moins significatif de Z. Les dernières trois instructions additionnent les mots les plus significatifs de X et de Y, en incluant la retenue et stockent le résultat dans le mot le plus significatif de Z. Souvenez-vous : les expressions d'adresse de l'allure "X+2" accèdent au mot le plus significatif d'une entité de 32 bits. Ceci est dû au fait que l'espace d'adressage compte par octets et qu'il faut deux octets consécutifs pour former un mot.

Certes, si vous avez un processeur 80386 ou supérieur, vous n'avez pas besoin de passer par tout ceci juste pour additionner deux valeurs de 32 bits. Cependant, si vous additionnez deux valeurs de *64 bits* avec le 80386, vous aurez encore besoin de cette méthode.

Vous pouvez élargir cette technique à tout nombre de bits en utilisant l'instruction adc et en additionnant ainsi les mots les plus significatifs des valeurs. Par exemple, pour additionner deux valeurs de 128 bits, vous pourriez utiliser un code comme le suivant :

```
BigVal1         dword 0,0,0,0           ; Quatre double-mots en 128 bits !
BigVal2         dword 0,0,0,0
BigVal3         dword 0,0,0,0
                .
                .
                .
                mov     eax, BigVal1    ; Pas besoin de dword ptr car il s'agit de
                add     eax, BigVal2    ; variables de type dword.
                mov     BigVal3, eax
                mov     eax, BigVal1+4  ; Additionner sur les valeurs de l'entité
                adc     eax, BigVal2+4  ; à l'entité forte, en utilisant
                mov     eax, BigVal1+8  ; l'instruction ADC.
                adc     eax, BigVal2+8
                mov     BigVal3+8, eax
                mov     eax, BigVal1+12
                adc     eax, BigVal2+12
                mov     1BigVal3+12, eax
```

9.3.2 Opérations de soustraction en multiprécision

Tout comme l'addition, les machines 80x86 effectuent des soustractions multi-octets de la même façon qu'on les effectue manuellement, sauf qu'elles soustraient par octets, par mots ou par doubles-mots à la fois, au lieu que par chiffres décimaux à la fois. Le mécanisme est semblable à celui qu'on a vu avec l'opération add. Vous utilisez simplement l'instruction sub avec les octets / mots / doubles-mots les moins significatifs et l'instruction sbb avec les octets / mots / doubles-mots les plus significatifs. L'exemple suivant montre une soustraction de 32 bits avec des registres de 16 bits du 8086 :

```
var1            dword   ?
var2            dword   ?
diff            dword   ?

                mov     ax, word ptr var1
                sub     ax, word ptr var2
```

```
                mov     word ptr diff, ax
                mov     ax, word ptr var1+2
                sbb     ax, word ptr var2+2
                mov     word ptr diff+2, ax
```

L'exemple qui suit montre une soustraction de 128 bits en utilisant des registres de 32 bits du 80386 :

```
BigVal1         dword   0,0,0,0         ; Pour des doubles-mots en 128 bits !
BigVal2         dword   0,0,0,0
BigVal3         dword   0,0,0,0
                .
                .
                .
                mov     eax, BigVal1    ; Pas besoin de dword ptr car il s'agit de
                sub     eax, BigVal2    ; variables dword.
                mov     BigVal3, eax
                mov     eax, BigVal1+4  ; Soustraire sur les valeurs de l'entité
                sbb     eax, BigVal2+4  ; à l'entité la plus significative,
                mov     BigVal3+4, eax  ; en utilisant les instructions SUB et SBB.
                mov     eax, BigVal1+8
                sbb     eax, BigVal2+8
                mov     BigVal3+8, eax
                mov     eax, BigVal1+12
                sbb     eax, BigVal2+12
                mov     BigVal3+12, eax
```

9.3.3 Comparaisons de valeurs en précision étendue

Malheureusement, il n'y a pas d'instruction "comparer avec reste" dans le cas des valeurs en précision étendue. Pusque les instructions cmp et sub font le même travail, au moins tant que les flags seront concernés, vous pourriez probablement songer à l'instruction sbb pour synthétiser ce type de comparaison ; néanmoins, vous n'auriez raison que partiellement. Il y a une meilleure façon de le faire.

Considérez les deux valeurs non signées 2157h et 1293h. L'octet le moins significatif de ces deux valeurs n'affecte pas le résultat de la comparaison. Comparer simplement 21h et 12h suffit à nous faire réaliser que 21h est plus grand que 12h. En effet, la seule occasion où vous aurez besoin de comparer tous les octets de chaque nombre sera quand les deux octets les plus significatifs seront égaux. Dans tous les autres cas, ne comparer que les bits les plus significatifs nous indique tout ce dont on a besoin pour connaître les relations metriques entre ces valeurs. Sans doute, ceci reste vrai pour tout nombre d'octets et pas juste deux. Le code suivant compare deux nombres signés de 64 bits sur un processeur 80386 ou ultérieur.

```
; Cette séquence transfère le contrôle à l'emplacement "IsGreater" si
; QwordValue > QwordValue2. Elle transfère le contrôle à l'emplacement "IsLess" si
; QwordValue < QwordValue2. Et le contrôle tombe sur la séquence suivante si
; QwordValue = QwordValue2. Le test pour l'inégalité change les opérandes "IsGreater"
; et "IsLess" à "NotEqual".

        mov     eax, dword ptr QWordValue+4     ; Obtenir le mot le plus significatif
        cmp     eax, dword ptr QWordValue2+4
        jg      IsGreater
        jl      IsLess
        mov     eax, dword ptr QWordValue
        cmp     eax, dword ptr QWordValue2
        jg      IsGreater
        jl      IsLess
```

Pour comparer des valeurs non signées, utilisez simplement les instructions ja et jb au lieu de jg et jl.

Vous pouvez facilement synthétiser toute comparaison possible en vous inspirant de la séquence ci-dessus, les exemples suivants montrent comment le faire. Ces exemples font des comparaisons signées, pour effectuer des comparaisons non signées, substituez simplement ja, jae, jb et jbe par jg, jge, jl et jle (respectivement).

```
QW1     qword   ?
QW2     qword   ?
dp      textequ <dword ptr>
```

```
; Test de 64 bits pour voir si QW1 < QW2 (signé).
; Le contrôle tombe sur "IsLess" si QW1 < QW2. Sinon, il tombe sur
; la prochaîne instruction dans le cas contraire.

        mov     eax, dp QW1+4   ; Obtenir le double-mot le plus significatif
        cmp     eax, dp QW2+4
        jg      NotLess
        jl      IsLess
        mov     eax, dp QW1     ; Tomber ici si les doubes-mots les plus significatifs
        cmp     eax, dp QW2     ; sont égaux
        jl      IsLess
NotLess:

; Test de 64 bits pour voir si QW1 <= QW2 (signé).

        mov     eax, dp QW1+4   ; Obtenir le double-mot le plus significatif
        cmp     eax, dp QW2+4
        jg      NotLessEq
        jl      IsLessEq
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        jle     IsLessEq
NotLessEQ:

; Test de 64 bits pour voir si QW1 >QW2 (signé).

        mov     eax, dp QW1+4 ;Get H.O. dword
        cmp     eax, dp QW2+4
        jg      IsGtr
        jl      NotGtr
        mov     eax, dp QW1   ; Le contrôle tombe ici si les doubles-mots sont égaux
        cmp     eax, dp QW2
        jg      IsGtr
NotGtr:

; Test de 64 bits pour voir si QW1 >= QW2 (signé).

        mov     eax, dp QW1+4 ;Get H.O. dword
        cmp     eax, dp QW2+4
        jg      IsGtrEq
        jl      NotGtrEq
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        jge     IsGtrEq
NotGtrEq:

; Test de 64 bits pour voir si QW1 = QW2 (signé ou non signé). Ce code branche
; sur l'étiquette "IsEqual" si QW1 = QW2. Sinon, le contrôle tombe sur la prochaine
; instruction.

        mov     eax, dp QW1+4   ; Obtenir le double-mot le plus significatif
        cmp     eax, dp QW2+4
        jne     NotEqual
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        je      IsEqual
NotEqual:

; Test de 64 bits pour voir si QW1 <> QW2 (signé ou non signé). Ce code branche sur
; l'étiquette "NotEqual" si QW1 <> QW2. Sinon, le contrôle tombe sur la prochaine
; instruction.

        mov     eax, dp QW1+4   ; Obtenir le double-mot le plus significatif
        cmp     eax, dp QW2+4
        jne     NotEqual
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        jne     NotEqual
```

9.3.4 Multiplication en précision étendue

Bien que les multiplications 16x16 ou 32x32 soient normalement suffisantes, il y a des fois qu'on peut avoir besoin de multiplier des valeurs supérieures. Pour ceci, on peut utiliser les instructions **mul** et **imul**.

Il ne doit pas surprendre que les techniques emploiées pour ce type d'opération soient les mêmes qu'on utiliserait en multipliant manuellement deux valeurs.

Considérez ces multiplication multichiffres simples effectuées à la main :

```
1) Multiplier les premiers deux chiffres :
        123
        45
        ---
        15

2) Multiplier 5*2 :
         123
         45
         ---
         15
        10

3) Multiplier 5*1 :
        123
        45
        ---
        15
       10
       5

4) 4*3 :
         123
         45
         ---
         15
        10
        5
        12

5) Multiplier 4*2 :
        123
        45
        ---
        15
       10
       5
       12
       8

6) 4*1 :
         123
         45
         ---
         15
        10
        5
        12
        8
       4

7) Additionner les produits partiels :
        123
        45
        ---
        15
        10
        5
        12
        8
        4
        ------
        5535
```

Les machines 80x86 effectuent les multiplications de la même façon, sauf qu'elles travaillent avec des octets, des mots ou des doubles-mots à la fois. La figure 8.2 montre comment ceci fonctionne.

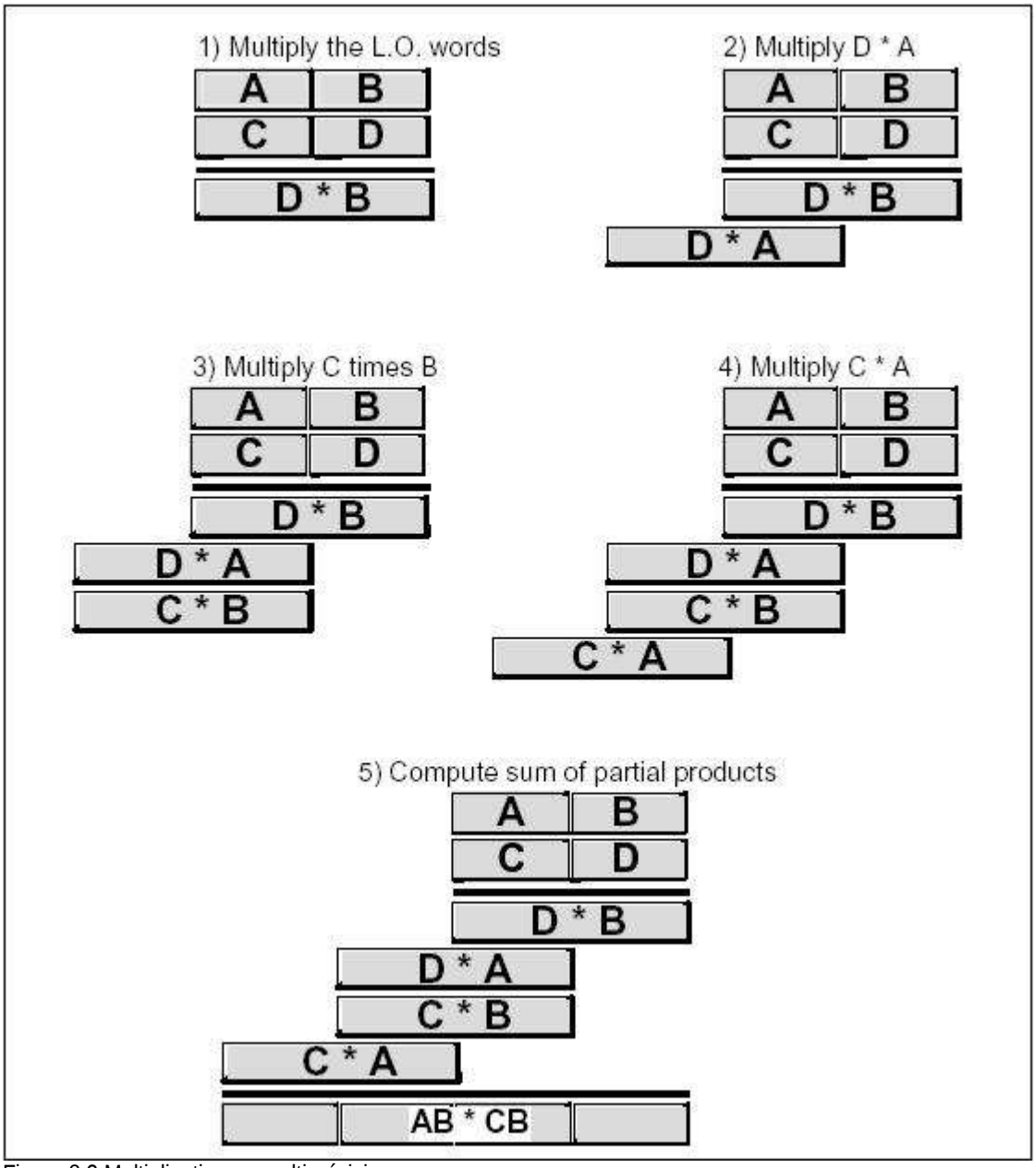


Figure 8.2 Multiplication en multiprécision

La chose la plus importante à se rappeler avec ce genre d'opérations est que vous devez également effectuer une addition en même temps. Additionner les produits partiels demande diverses additions. Le listing suivant montre la façon correcte de multiplier deux valeurs de 32 bits sur un processeur de 16 bits :

Note : *Multiplier* et *Mutiplicand* sont des variables de 32 bits déclarées dans le segment de données avec la directive dword. *Product* est une variable de 64 bits déclarée dans le segment de données avec la directive qword.

```
Multiply        proc    near
                push    ax
                push    dx
                push    cx
                push    bx

; Multiplier le mot le moins significatif (L.O.) de Multiplier Multiplicand fois :

                mov     ax, word ptr Multiplier
                mov     bx, ax                          ; Garder la valeur Multiplicand
```

```
                mul     word ptr Multiplicand           ; Multiplier les mots L.O.
                mov     word ptr Product, ax            ; Garder le produit partiel
                mov     cx, dx                          ; Garder le mot le plus
                                                        ; significatif (H.O.)
                mov     ax, bx                          ; Obtenir Multiplier dans bx
                mul     word ptr Multiplicand+2         ; Multiplier L.O. * H.O.
                add     ax, cx                          ; Additionner produit partiel
                adc     dx, 0                           ; N'oubliez pas la retenue !
                mov     bx, ax                          ; Garder le produit partiel
                mov     cx, dx                          ; pour le moment.

; Multiplier le mot H.O. de Multiplier Multiplicand fois :

                mov     ax, word ptr Multiplier+2 ;Get H.O. Multiplier
                mul     word ptr Multiplicand           ; Fois le mot L.O.
                add     ax, bx                          ; Additionner produit partiel
                mov     word ptr product+2, ax          ; Garder le produit partiel
                adc     cx, dx                          ; Sans oublier la retenue
                mov     ax, word ptr Multiplier+2       ; Multiplier les mot fort
                mul     word ptr Multiplicand+2
                add     ax, cx                          ; Additionner produit partiel
                adc     dx, 0                           ; Sans oublier la retenue
                mov     word ptr Product+4, ax          ; Garder produit partiel
                mov     word ptr Product+6, dx
                pop     bx
                pop     cx
                pop     dx
                pop     ax
                ret
Multiply        endp
```

Ce que vous devez garder à l'esprit à propos de ce code est qu'il fonctionne seulement avec des opérandes non signées. La multiplication d'opérandes signées figurera dans les exercices.

9.3.5 Division en précision étendue

Vous ne pouvez pas synthétiser une division générale n-bit / m-bit avec les instructions **div** et **idiv**. Une telle opération doit être effectuée à l'aide d'une séquence d'instructions de décalage et de soustraction et par conséquent elle est assez embrouillée. Une opération moins générale qui consiste à diviser une quantité de n bits pas une quantité de 32 bits (sur des processeurs 80386 ou ultérieurs) ou par une quantité de 16 bits est plus facilement généralisable et peut être effectuée à l'aide de l'instruction **div**. Le code suivant montre comment diviser une valeur de 64 bits par un diviseur de 16 bits, produisant un quotient de 64 bits et un reste de 16 bits :

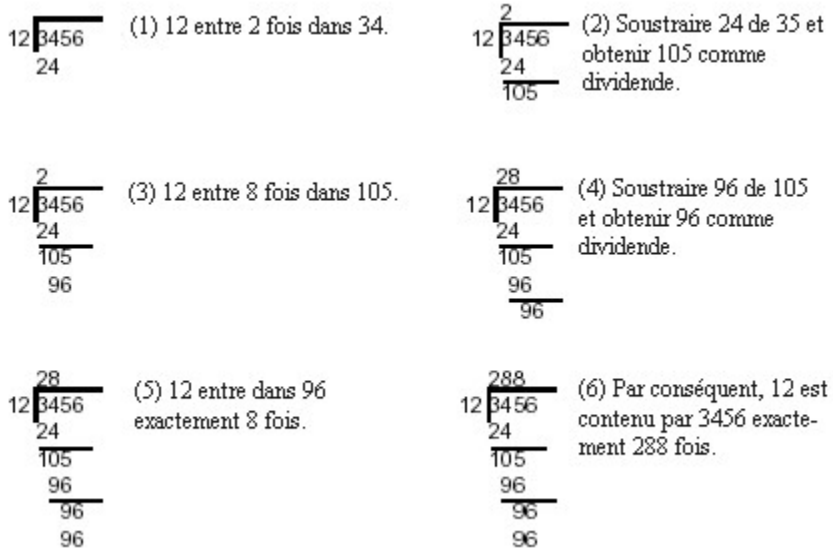
```
dseg            segment para public 'DATA'
dividend        dword   0FFFFFFFFh, 12345678h
divisor         word    16
Quotient        dword   0,0
Modulo          word    0
dseg            ends

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

; Diviser une quantité de 64 bits par une quantité de 16 bits :

Divide64        proc    near
                mov     ax, word ptr dividend+6
                sub     dx, dx
                div     divisor
                mov     word ptr Quotient+6, ax
                mov     ax, word ptr dividend+4
                div     divisor
                mov     word ptr Quotient+4, ax
                mov     ax, word ptr dividend+2
```

```
                div     divisor
                mov     word ptr Quotient+2, ax
                mov     ax, word ptr dividend
                div     divisor
                mov     word ptr Quotient, ax
                mov     Modulo, dx
                ret
Divide64        endp
cseg            ends
```

Ce code peut être étendu à tout nombre de bits en ajoutant simplement des instructions mov / div / mov supplémentaires au début de la séquence. Certes, sur des processeurs 80386 et supérieurs, vous pouvez diviser par une valeur de 32 bits en utilisant les registres edx et eax dans le code qu'on a vu (et quelques autres petits ajustements).

Si vous avez besoin d'utiliser un diviseur plus grand que 16 bits (par exemple de 32 bits sur un processeur adécuat), vous devrez implémenter la division en utilisant la stratégie décalage / soustraction. Malheureusement, un tel algorithme est très lent. Dans cette section, nous allons développer deux algorithmes de division qui fonctionnent sur un nombre arbitraire de bits. Le premier est lent mais facile à comprendre, alors que le second est décidément plus rapide (en général).

Comme pour la multiplication, la meilleure façon de comprendre comment un ordinateur effectue la division est d'étudier le procédé employé en effectuant la division à la main. Considérez la division 3456 / 12 et les étapes nécessaires pour la réaliser :



Cet algorithme est en effet facile en binaire, car à chaque étape vous n'avez pas à déduire combien de fois 12 est contenu dans le reste et vous n'avez non plus à multiplier 12 par ce nombre de fois afin d'obtenir la quantité à soustraire. À chaque étape de l'algorithme binaire, le diviseur finit dans le reste exactement zéro fois ou une fois. En guise d'exemple, considérez la division entre 27 (11011) et 3 (11) :

```
11 | 11011
   11
```

11 est contenu par 11 exactement 11 fois.

```
    1
11 | 11011
   11
   --
   00
```

Soustraire 11 et obtenir 0 comme dividende.

```
    1
11 | 11011
   11
   --
   00
   00
```

11 entre dans 00 0 fois.

```
   10
11 | 11011
   11
   --
   00
   00
   --
   01
```

Soustraire 00 et obtenir 01 comme dividende.

```
   10
11 | 11011
   11
   --
   00
   00
   --
   01
   00
```

11 entre dans 00 0 fois.

```
   100
11 | 11011
   11
   --
   00
   00
   --
   01
   00
   --
   11
```

Soustraire 00 et obtenir 11 comme dividende.

```
   100
11 | 11011
   11
   --
   00
   00
   --
   01
   00
   --
   11
   11
```

11 entre dans 11 une fois.

```
   1001
11 | 11011
   11
   --
   00
   00
   --
   01
   00
   --
   11
   11
   --
   00
```

Ce qui produit le résultat final, 1001.

Il y a pourtant une nouvelle façon d'implémenter cet algorithme binaire qui, en même temps, calcule le quotient et le reste. L'algorithme est le suivant :

```
Quotient := Dividend;
Remainder := 0;
for i:= 1 to NumberBits do

        Remainder:Quotient := Remainder:Quotient SHL 1;
        if Remainder >= Divisor then
                Remainder := Remainder - Divisor;
                Quotient := Quotient + 1;
        endif
endfor
```

NumberBits est le nombre de bits dans les variables Remainder, Quotient, Divisor et Dividend. Notez que l'instruction Quotient := Quotient + 1 met à 1 le bit le moins significatif de Quotient car l'algorithme avait précédemment décalé Quotient d'une position vers la gauche. Le code équivalent 80x86 de cet algorithme est :

```
; Supposer que Dividend (et Quotient) est DX:AX, La variable Divisor est dans CX:BX,
; et Remainder dans SI:DI.
                mov     bp, 32          ; Compte 32 bits dans BP
                sub     si, si          ; Met à zéro le reste
                sub     di, di
BitLoop:        shl     ax, 1           ; Voir la section des décalages décrivant
                rcl     dx, 1           ; le fonctionnement de SHL avec 64 bits
                rcl     di, 1
                rcl     si, 1
                cmp     si, cx          ; Comparer les mots les plus significatifs de
                                        ; Remainder et de Divisor.
                ja      GoesInto
                jb      TryNext
                cmp     di, bx          ; Comparer les mots les moins significatifs.
                jb      TryNext
GoesInto:       sub     di, bx          ; Remainder := Remainder -
                sbb     si, cx          ; Divisor
                inc     ax              ; Mettre à 1 le bit L.O. de AX
TryNext:        dec bp                  ; Répéter 32 fois.
                jne     BitLoop
```

Bien que ce code vous paraîtra court et simple, il a plusieurs inconvénients. D'abord, il ne teste pas la division par zéro (si on essaie de diviser par zéro, le résultat est 0FFFFFFFFh), il fonctionne seulement avec des valeurs non signées et c'est très lent. Certes, tester la division par zéro est très simple, il suffit de comparer le diviseur avec zéro avant d'effectuer la division et de retourner un code approprié si cette erreur se produit. Implémenter l'opération avec des valeurs signées est également simple, on vous expliquera comment le faire sous peu. Malgré tout cela, la performance de cet algorithme laisse encore beaucoup à désirer. Seulement un passage dans la boucle du programme consomme à peu près 30 cycles d'horloge[3] et effectuer tout l'algorithme emploierait au moins 1000 cycles ! C'est un ordre de grandeur encore pire que celui des instructions DIV/IDIV (le plus lentes du jeu d'instructions 80x86).

Il y a une technique que vous pouvez utiliser pour améliorer pour du bon les performances de cette division : vérifier si réellement la variable Divisor a besoin de 32 bits. Souvent, même quand le diviseur est une variable de 32 bits, la valeur elle-même peut être contenue dans 16 bits (c'est-à-dire que le mot le plus significatif du diviseur est zéro). Dans ce cas spécial - qui se produit fréquemment - vous pouvez utiliser l'instruction DIV, qui est beaucoup plus rapide.

9.3.6 Opérations NEG en précision étendue

Bien qu'il y a différentes manières de nier une valeur de précision étendue, la plus courte est d'utiliser une combinaison d'instructions neg et sbb. Cette technique exploite le fait que neg soustrait son opérande de zéro. Particulièrement, elle affecte les flags de la même façon que sub les affecterait si on soustrayait de zéro son opérande de destination. Ce code prend la forme suivante :

```
                neg     dx
```

[3]Ceci varie selon le processeur utilisé.

```
                neg     ax
                sbb     dx,0
```

L'instruction sbb décrémente dx s'il y a une retenue dans le mot faible de l'opération de négation (ce qui aura toujours lieu au moins que ax est égal à zéro).

Effectuer cette opération avec un nombre supplémentaire de bits, de mots ou de doubles-mots est facile. Tout ce que vous avez à faire et de commencer avec l'emplacement fort de l'objet que vous voulez nier et procéder jusqu'à l'emplacement faible de cet objet. Le code suivant calcule une négation de 128 bits sur un processeur 80386 :

```
Value           dword   0,0,0,0         ; Entier de 128 bits.
                .
                .
                .
                neg     Value+12        ; Nier le double-mot le plus significatif.
                neg     Value+8         ; Nier le double-mot précédent.
                sbb     Value+12, 0     ; Ajuster le double-mot le plus significatif.
                neg     Value+4         ; Neg est le second double-mot de l'objet.
                sbb     Value+8, 0      ; Ajuster le 3me double-mot de l'objet.
                sbb     Value+12, 0     ; Ajuster toutes les retenues jusqu'au mot H.O.
                neg     Value           ; Nier le mot le moins significatif.
                sbb     Value+4, 0      ; Ajuster le second double-mot dans l'objet.
                sbb     Value+8, 0      ; Ajuster le 3me double-mot dans l'objet.
                sbb     Value+12, 0     ; Ajuster toutes les retenues jusqu'au mot H.O.
```

Malheureusement, ce code tend à être très grand et lent, à cause de la nécessité de propager la retenue jusqu'aux mots H.O. après chaque opération de négation. Une manière plus simple d'effectuer la même opération avec des valeurs plus larges est de simplement soustraire cette valeur de zéro :

```
Value           dword   0,0,0,0,0       ; Entier de 160 bits.
.
.
.
                mov     eax, 0
                sub     eax, Value
                mov     Value, eax
                mov     eax, 0
                sbb     eax, Value+4
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+8
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+12
                mov     Value+12, ax
                mov     eax, 0
                sbb     eax, Value+16
                mov     Value+16, ax
```

9.3.7 Opérations AND en précision étendue

Effectuer un AND sur un nombre quelconque de mots est très simple : faites simplement un AND entre les mots correspondants entre les deux opérandes et enregistrez le résultat. Par exemple, pour effectuer un AND où les trois opérandes sont de 32 bits, vous pourriez utiliser un code comme celui qui suit :

```
                mov     ax, word ptr source1
                and     ax, word ptr source2
                mov     word ptr dest, ax
                mov     ax, word ptr source1+2
                and     ax, word ptr source2+2
                mov     word ptr dest=2, ax
```

Cette technique s'applique aussi facilement à tout nombre de mots. Tout ce dont vous avez besoin et d'effectuer le AND entre les octets, les mots ou les doubles-mots correspondants.

9.3.8 Opérations OR en précision étendue

Les opérations OR entre multiples mots sont effectués de la même façon que les opérations AND entre multiples mots. Il suffit d'effectuer un OR entre les mots correspondants des deux opérandes. Par exemple, voici un exemple d'OR entre deux valeurs de 48 bits :

```
mov     ax, word ptr operand1
or      ax, word ptr operand2
mov     word ptr operand3, ax
mov     ax, word ptr operand1+2
or      ax, word ptr operand2+2
mov     word ptr operand3+2, ax
mov     ax, word ptr operand1+4
or      ax, word ptr operand2+4
mov     word ptr operand3+4, ax
```

9.3.9 Opérations XOR de précision étendue

Les opérations de type XOR sont effectuées de manière identique que les opérations AND/OR. Is suffit d'effectuer un XOR entre les mots correspondants des deux opérandes pour obtenir un résultat en précision étendue. La séquence de code suivante effectue un XOR entre deux opérandes de 64 bits et garde le résultat dans une variable de 64 bits. Cet exemple fait usage des registres de 32 bits disponibles sur les processeurs 80386 et ultérieurs :

```
mov     eax, dword ptr operand1
xor     eax, dword ptr operand2
mov     dword ptr operand3, eax
mov     eax, dword ptr operand1+4
xor     eax, dword ptr operand2+4
mov     dword ptr operand3+4, eax
```

9.3.10 Opérations NOT en précision étendue

L'instruction NOT inverse tous les bits de l'opérande spécifiée. Elle n'affecte pas les flags (par conséquent, utiliser un saut conditionnel après une instruction not n'a pas de signification). Un not en précision étendue est effectué simplement via un not sur toutes les opérandes affectées. Par exemple, pour effectuer un not sur une valeur de 32 bits dans dx:ax, tout ce dont vous avez besoin est :

```
not     ax      or              not     dx
not     dx                      not     ax
```

Gardez à l'esprit que si vous exécutez l'instruction not deux fois, vous retourneriez à la valeur originale. Notez aussi qu'effectuer un OR exclusif d'une valeur avec une autre avec tous les bits à 1 (par exemple, 0FFh, 0FFFFh, 0FF..FFh) produit le même effet que l'instruction not.

9.3.11 Opérations de décalage en précision étendue

Ces opérations demandent une instruction de décalage et une de rotation. Considérez ce qu'il faut faire pour implémenter un shl de 32 bits en utilisant des opérations de 16 bits :

1) Un zéro doit être décalé dans le bit 0.

2) Les bits de 0 à 14 sont décalés d'une position vers la gauche.

3) Le bit 15 est décalé dans le bit 16.

4) Les bits de 16 à 30 doivent être décalés d'une position vers la gauche.

5) Le bit 31 est décalé dans le carry flag.

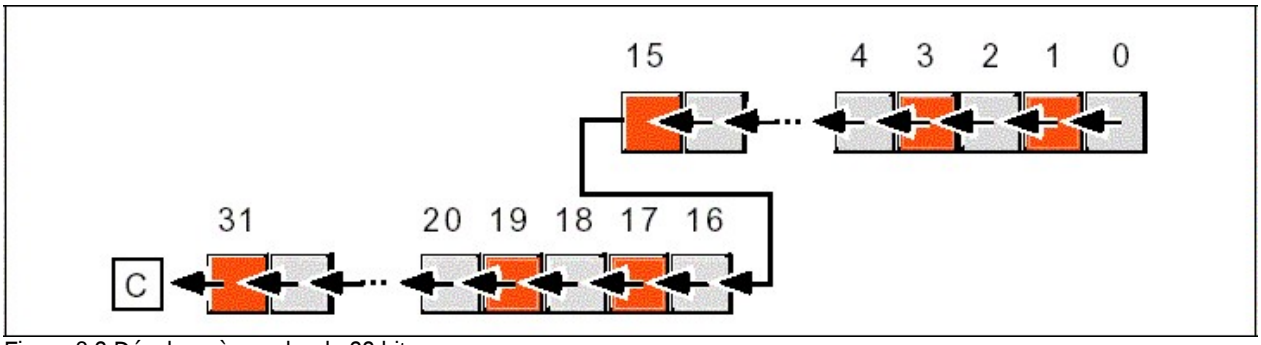


Figure 8.3 Décalage à gauche de 32 bits

Les deux instructions que vous pouvez utiliser pour implémenter un décalage de 32 bits sont **shl** et **rcl**. Par exemple, pour décaler la quantité de 32 bits se trouvant dans dx:ax d'une position vers la gauche, on peut utiliser :

```
shl     ax, 1
rcl     dx, 1
```

Notez que vous ne pouvez décaler une valeur en précision étendue que d'un bit à la fois. Vous ne pouvez pas décaler une telle valeur de diverses positions à la fois en utilisant le registre **cl** ou une valeur immédiate plus grande que 1.

Pour comprendre comment cette séquence d'instructions fonctionne, considérez ces opérations individuellement. L'instruction **shl** décale un zéro dans le bit 0 d'une opérande de 32 bits et décale le bit 15 dans le carry flag. L'instruction **rcl** décale ensuite le carry flag dans le 16_{me} bit, puis décale le bit 31 dans le drapeau de retenue. Le résultat correspond exactement à ce qu'on veut.

Pour effectuer un décalage vers la gauche d'une valeur qui dépasse les 32 bits, vous emploierez simplement des instructions **rcl** additionnelles. Un décalage vers la gauche commence toujours par le mot le moins significatif et toute instruction **rcl** successive opère sur le prochain mot significatif. Par exemple, pour effectuer un décalage à gauche de 48 bis sur un emplacement de mémoire, vous pouvez utiliser les instructions suivantes :

```
shl     word ptr Operand, 1
rcl     word ptr Operand+2, 1
rcl     word ptr Operand+4, 1
```

Si vous voulez décaler vos données de deux ou plusieurs bits à la fois, vous pouvez ou bien répéter la séquence ci-dessus par le nombre désiré de fois (pour un nombre constant de décalages), ou bien placer ces instructions dans une boucle pour les répéter un nombre variable de fois. Par exemple, le code suivant décale la variable **Operand** vers la gauche selon le nombre de fois spécifié dans cx :

```
ShiftLoop:      shl     word ptr Operand, 1
                rcl     word ptr Operand+2, 1
                rcl     word ptr Operand+4, 1
                loop    ShiftLoop
```

Vous implémentez les instructiolns **shr** et **sar** de façon similaire, avec la seule différence que vous devez commencer du bit le plus significatif et procéder vers le moins :

```
DblSAR:         sar     word ptr Operand+4, 1
                rcr     word ptr Operand+2, 1
                rcr     word ptr Operand, 1

DblSHR:         shr     word ptr Operand+4, 1
                rcr     word ptr Operand+2, 1
                rcr     word ptr Operand, 1
```

Il y a une différence fondamentale entre les décalages en précision étendue décrits ici et leurs versions 8/16 bits : les décalages en précision étendue affectent les flags différemment. Par exemple, le **zero flag** est activé si la dernière rotation a produit un résultat nul et pas si toute l'opération a produit un résultat nul. Pour les opérations de décalage à droite, les drapeaux **sign** et **overflow** ne sont pas réglés correctement (alors qu'ils le sont dans le cas du décalage à gauche). Un test supplémentaire est requis si vous avez besoin de tester un de ces flags après une opération de décalage

en précision étendue. Heureusement, le drapeau carry est le plus couramment utilisé après toute opération de décalage et ce flag est réglé correctement.

Les instructions shld et shrd vous permettent d'implémenter efficacement des décalages en multiprécision de divers bits - sur les processeurs 80386 et ultérieurs. Considérez la séquence suivante :

```
ShiftMe         dword   1234h, 5678h, 9012h
                .
                .
                .
                mov     eax, ShiftMe+4
                shld    ShiftMe+8, eax, 6
                mov     eax, ShiftMe
                shld    ShiftMe+4, eax, 6
                shl     ShiftMe, 6
```

Rappellez-vous que l'instruction shld décale les bits de sa seconde opérande dans sa première opérande. Par conséquent, la première instruction shld ci-dessus décale les mots de ShiftMe+4 à ShiftMe+8, *sans affecter la valeur de ShiftMe+4*. La seconde instruction shld ci-dessus décale les mots de ShiftMe à ShiftMe+4. Finalement, l'instruction shl décale le double-mot faible le nombre de fois spécifié. Il y a deux choses importantes à remarquer sur ce code : d'abord, contrairement aux autres opéations de décalage vers la gauche, cette séquence procède du double-mot fort au double-mot faible ; ensuite, le carry flag ne contient pas ce qui est sorti de la dernière opération de décalage. Si vous avez besoin de préserver le carry flag, vous allez avoir besoin d'empiler les drapeaux après le premier shld et les désempiler après shl.

Vous pouvez effectuer un décalage à droite en précision étendue avec l'instruction shrd. Elle fonctionne de la même façon que le code ci-dessus, avec la seule différence que vous procédez du double-mot le moins significatif au double-mot le plus significatif. Dans les exercices on vous demandera d'écrire un exemple de cette opération.

9.3.12 Opérations de rotation en précision étendue

Les opérations rcl et rcr, fonctionnent de manière identique que shl et shr. Par exemple, voici les instructions à utiliser pour effectuer des rotations sur une valeur de 48 bits :

```
                rcl     word ptr operand,1
                rcl     word ptr operand+2, 1
                rcl     word ptr operand+4, 1
                rcr     word ptr operand+4, 1
                rcr     word ptr operand+2, 1
                rcr     word ptr operand, 1
```

La seule différence entre ce code et sa version de décalage est que la première instruction est rcl ou rcr au lieu que shl ou shr. Ces opérations sont donc très simples. Elles apparaîtront dans les exercices. Sur les processeurs 80386 et ultérieurs, vous pouvez utiliser les instructions bt et shld pour implémenter plus facilement des instructions rol et ror en précision étendue. Le code suivant montre précisement cet usage :

```
; Calcule ROL EDX:EAX, 4

                mov     ebx, edx
                shld    edx, eax, 4
                shld    eax, ebx, 4
                bt      eax, 0          ; Met le carry flag à 1 si désiré.
```

La version ror est très similaire ; gardez simplement à l'esprit que vous travaillez d'abord sur la partie la moins significative et ensuite sur la partie la plus significative.

9.4 Opérations sur des opérandes de différentes tailles

Occasionnellement, on peut avoir besoin d'effectuer des opérations entre opérandes de différentes tailles. Par exemple, on pourait additionner un mot à un double-mot ou soustraire un octet d'un mot. La solution est simple : élargir la plus petite valeur de manière à l'adapter à la taille de l'autre et ensuite effectuer l'opération. Pour les opérandes signées, on peut faire une extension signée. Pour des opérandes non signée, l'extension est de de zéro. Ceci fonctionne pour toute opération, bien qu'on montrera des exemples juste pour l'opération d'addition.

Pour ce faire, on applique les instructions d'extension et une fois cela fait, on peut effectuer l'opération qu'on veut. Considérez le code suivant qui additionne un octet à un mot :

```
var1            byte            ?
var2            word            ?

Addition non signée :           Addition signée :

mov     al, var1                mov     al, var1
mov     ah, 0                   cbw
add     ax, var2                add     ax, var2
```

Dans les deux cas, la variable octet a été chargée dans le registre **al**, élargie à 16 bits et finalement additionnée à l'opérande mot. Ce code est parfait si vous pouvez choisir l'ordre d'évaluation des opérations. Parfois, on ne peut pas choisir l'ordre d'évaluation, par exemple, la valeur de 16 bits est déjà dans le registre **ax** et vous voulez lui additionner une valeur de 8 bits. Pour l'addition non signée, vous pouvez utiliser le code suivant :

```
                mov     ax, var2        ; Charge la valeur de 16 bits dans AX
                .                       ; Faire d'autres opération en laissant
                .                       ; la valeur de 16 bits dans AX.
                add     al, var1        ; Additionner la valeur de 8 bits.
                adc     ah, 0           ; Ajouter la retenue dans l'octet H.O.
```

La première instruction **add** de cet exemple additionne l'octet se trouvant dans **var1** à l'octet le moins significatif de la valeur dans l'accumulateur. L'instruction **adc** additionne la retenue de l'octet dans l'octet le plus significatif de l'accumulateur. Il faut faire attention à s'assurer que cette instruction **adc** soit présente. Si vous l'omettez, vous risquez de ne pas obtenir un résultat correct.

Additionner une valeur signée de 8 bits à une valeur signée de 16 bits est un peu plus difficile. Malheureusement, on ne peut pas additionner une valeur immédiate (comme ci-dessus) à l'octet le plus significatif de **ax**. Ceci parce que cet octet peut être soit 00h, soit 0FFh. Si un registre est disponible, la meilleure chose à faire est :

```
                mov     bx, ax          ; BX est le registre disponible.
                mov     al, var1
                cbw
                add     ax, bx
```

Si un registre n'est pas disponible, vous pouvez faire ce qui suit :

```
                add     al, var1
                cmp     var1, 0
                jge     add0
                adc     ah, 0FFh
                jmp     addedFF
add0:           adc     ah, 0
addedFF:
```

Si un autre registre n'est pas disponible, vous pouvez également pousser un registre occupé dans la pile de façon à l'enregistrer pendant l'opération. Par exemple :

```
                push    bx
                mov     bx, ax
                mov     al, var1
                cbw
                add     ax, bx
                pop     bx
```

Une autre alternative consiste à garder la valeur de 16 bits de l'accumulateur dans un emplacement de mémoire et puis procéder comme ci-dessus :

```
                mov     temp, ax
                mov     al, var1
                cbw
                add     ax, temp
```

Tous les exemples vus additionnaient un octet à un mot. En effectuant une extension non signée ou signée sur l'opérande la plus petite, on peut facilement additionner toute paire de valeurs de différentes tailles. Considérez le code suivant qui additionne un octet à une valeur double-mot :

```
var1    byte ?
var2    dword ?
mov     al, var1
cbw
cwd                     ; Élargir à 32 bits dans DX
add     ax, word ptr var2
adc     dx, word ptr var2+2
```

Si vous travaillez avec un 80386 ou supérieur, vous pouvez utiliser :

```
movsx   eax, var1
add     eax, var2
```

Voici un exemple approprié sur un 80386 additionnant une valeur de 8 bits avec un quadword (64 bits) :

```
BVal            byte    -1
QVal            qword   1
                movsx   eax, BVal
                cdq
                add     eax, dword ptr QVal
                adc     edx, dword ptr QVal+4
```

Pour plus d'exemples, regardez les exercices.

9.5 Idiomes matériels et arithmétiques

Un idiome est une idiosyncrasie. Beaucoup d'opérations arithmetiques et d'instructions ont des idiosyncrasies desquelles vous pouvez prendre avantage. Certaines personnes se réfèrent à l'utilisation des idiomes arithmétiques et de machine avec le terme de « programmation astucieuse », de ce genre d'astuces que vous devriez éviter si vous voulez réaliser des programmes bien écrits. Malgré tout cela, certains artifices relatifs à la machine sont bien connus et on les trouve couramment dans les programmes en assembleur. Certains sont très contournés, mais d'autres sont simplements des « trucs et astuces du métier ». Ce livre ne peut même pas envisager de commencer à énumérer tous les artifices qu'on connaît de nos jours, ils sont trop nombreux et la liste est en évolution continuelle. Néanmoins, il y a certains idiomes importants que vous verrez tout le temps, il serait donc bon de les traiter ici.

9.5.1 Multiplier sans MUL et IMUL

Si vous jetez un coup d'œil au temps employé pour l'instruction de multiplication, vous remarquerez qu'il est très long. Sur le 80x86, seules les instructions div et idiv emploient plus de temps. À l'heure de multiplier par une constante, vous pouvez éviter la pénalité des performances, en utilisant des décalages, des additions et des soustractions pour effectuer une multiplication.

Souvenez-vous que l'instruction shl produit le même effet que multiplier l'opérande spécifiée par deux. Décaler de deux positions, a pour effet de multiplier par quatre. Et le faire de trois posions a comme résultat de multiplier par 8. En général, décaler une opérande de *n* bits, équivaut à multiplier cette opérande par 2^n. Toute valeur peut être multipliée par une constante donnée en utilisant une suite de décalages et d'additions ou de décalages et de soustractions. Par exemple, pour multiplier le contenu de ax par 10, il vous suffit de multiplier la valeur par 8 et ensuite lui additionner la même valeur multipliée par deux, c'est-à-dire 10*ax=8*ax+2*ax. Le code pour obtenir ceci est :

```
shl     ax, 1           ; Multiplication de ax par 2
mov     bx, ax          ; Garder la valeur obtenue dans bx
shl     ax, 1           ; Répéter, ce qui correspond à multiplier
                        ; la valeur originale par 4.
shl     ax, 1           ; Multiplier ax par 8, selon le même procédé.
add     ax, bx          ; Additionner avec 2*AX pour avoir 10*AX
```

Le registre ax (ainsi que tout autre registre) peut être multiplié par toute valeur constante beaucoup plus rapidement à l'aide de shl qu'en utilisant mul. Ceci semble difficile à croire, car mul ne requiert que deux instructions :

```
mov     bx, 10
mul     bx
```

Cependant, si vous regardez les temps d'exécution, la séquence avec les décalages est beaucoup plus rapide que l'instruction mul, et ceci dans la plupart des processeurs. Le code est évidemment un peu plus long à écrire, mais il en vaut la peine, considérant ce qu'on gagne en performance. Évidemment, sur les processeurs les plus recents, l'instructions mul est bien plus rapide que sur les anciennes machines, cependant la voie des décalage est encore plus rapide.

Vous pouvez également vous servir de la soustraction pour effectuer la multiplication. Considérez la multiplication suivante :

```
mov     bx, ax          ; Enregistre AX * 1
shl     ax, 1           ; AX := AX * 2
shl     ax, 1           ; AX := AX * 4
shl     ax, 1           ; AX := AX * 8
sub     ax, bx          ; AX := AX * 7
```

Ceci vient du fait que ax*7 = (ax*8)-ax.

Une erreur courante de débutant et de soustraire ou additionner 1 ou 2 au lieu de ax*1 ou ax*2. Ce qui suit *ne calcule pas* ax*7 :

```
shl     ax, 1
shl     ax, 1
shl     ax, 1
sub     ax, 1
```

mais il calcule (8*ax)-1, c'est-à-dire, quelque chose de complétement différent (sauf si ax = 1, évidemment). Gardez cela à l'esprit quand vous utilisez les décalages, les additions et les soustractions pour faire une multiplication.

Pour effectuer certains produits sur les 80386 et ultérieurs, vous pouvez utiliser lea aussi. L'astuce consiste à utiliser le mode d'indexage scalaire. Les exemples suivants montrent certains cas simples :

```
lea     eax, [ecx][ecx]         ;EAX := ECX * 2
lea     eax, [eax]eax*2]        ;EAX := EAX * 3
lea     eax, [eax*4]            ;EAX := EAX * 4
lea     eax, [ebx][ebx*4]       ;EAX := EBX * 5
lea     eax, [eax*8]            ;EAX := EAX * 8
lea     eax, [edx][edx*8]       ;EAX := EDX * 9
```

9.5.2 Division sans DIV et IDIV

Tout comme l'instruction shl peut être utilisée pour simuler une multiplication par certaines puissances de deux, les instructions shr et sar peuvent servir à simuler une division par la même puissance. Malheureusement, vous ne pouvez pas utiliser des décalages, des additions et des soustractions pour effectuer une division par une constante arbitraire aussi facilement qu'il le serait avec la multiplication.

Un autre moyen d'effectuer la division est d'utiliser les instructions de multiplication. Vous pouvez diviser par une certaine valeur en multipliant par son réciproque. L'instruction de multiplication est marginairement plus rapide que celle de division et multiplier par un réciproque est au moins toujours plus rapide que diviser.

Maintenant vous pourriez vous demander : « comment peut-on multiplier par un réciproque si les valeurs qu'on utilise sont toutes des entiers ? » La réponse est qu'on doit tricher pour le faire. Si on veut multiplier par 1/10, il n'y a pas de moyen pour charger cette valeur dans un registre avant d'effectuer la division. Néanmoins, on peut multiplier 1/10 par dix, effectuer notre multiplication et ensuite diviser le résultat par 10 pour obtenir le résultat final. Certes, cela ne vous avance en rien car vous vous trouvez à effectuer une addition et une division supplémentaires. Cependant, supposez de devoir multiplier 1/10 par 65536 (6553), effectuer notre multiplication et ensuite diviser par 65536. Ceci produit encore la bonne opération et, si vous traitez le problème correctement, vous pouvez obtenir l'opération de division gratuitement. Considérez le code suivant qui divide ax par dix :

```
mov     dx, 6554        ; Arrondissement de 65536 / 10
mul     dx
```

Ce code laisse ax/10 dans le registre dx.

Pour comprendre comment cela fonctionne, considérez ce qui arrive en mulipliant ax par 65536 (10000h). Ceci passe simplement ax dans dx et met ax à zéro. Puisque mul est un peu plus rapide que div, utiliser cette technique résulte plus rapide qu'utiliser une simple instruction de division.

Multiplier par un réciproque fonctionne bien quand vous avez besoin d'une division par une constante. Vous pourriez même l'utiliser pour diviser par une variable, mais les avantages obtenus par cette technique ne sont convenables que quand on se trouve à devoir répéter l'opération beaucoup de fois (par la même valeur).

9.5.3 Utiliser AND pour Calculer les Restes

L'instruction and peut être exploitée pour calculer rapidement des restes de la forme :

```
dest    := dest MOD 2^n
```

Pour obtenir un reste avec l'instruction and, il suffit d'exécuter cette instruction avec la valeur 2^n - 1. Par exemple, pour trouver ax = ax mod 8, faites

```
and     ax, 7
```

D'autres exemples :

```
and     ax, 3           ; AX := AX mod 4
and     ax, 0Fh         ; AX := AX mod 16
and     ax, 1Fh         ; AX := AX mod 32
and     ax, 3Fh         ; AX := AX mod 64
and     ax, 7Fh         ; AX := AX mod 128
mov     ah, 0           ; AX := AX mod 256
                        ; (Même que AX and 0FFh)
```

9.5.4 Implémenter des compteurs modulo-n avec AND

Si vous voulez implémenter une variable compteur pouvant s'incrémenter jusqu'à 2^n - 1 et ensuite se réinitialiser à zéro, le code à utiliser est :

```
inc     CounterVar
and     CounterVar, nBits
```

où nBits est une valeur binaire contenant *n* fois des bits justifiés à droite dans le nombre. Par exemple, pour créer un compteur qui procède de façon cyclique de 0 à 15 on peut utiliser ce qui suit :

```
inc     CounterVar
and     CounterVar, 00001111b
```

9.5.5 Tester une valeur en précision étendue pour 0FFFF..FFh

L'instruction and peut être utilisée pour tester rapidement si une valeur multi-mots contient des 1 dans toutes ses positions binaires. Chargez simplement le premier mot dans le registre ax et ensuite effectuez un AND entre le registre ax et tous les mots restant dans la structure de données. Quand l'opération AND est complète, le registre ax contiendra 0FFFFh seulement si tous les mots de la structure contenaient 0FFFFh. Par exemple :

```
mov     ax, word ptr var
and     ax, word ptr var+2
and     ax, word ptr var+4
.
.
.
and     ax, word ptr var+n
cmp     ax, 0FFFFh
je      Is0FFFFh
```

9.5.6 Opérations TEST

Si vous vous rappelez bien, l'instruction *test* n'est autre chose qu'une instruction **and** qui n'enregistre pas le résultat de l'opération (au moins, sauf les réglages des flags). Par conséquent, beaucoup de ce qui a été dit à propos de l'instruction **and** (spécialement pour ce qui concerne la façon d'affecter les flags), s'applique aussi pour l'instruction **test**. Cependant, puisque l'instruction **test** n'affecte pas l'opérande de destination, des test sur plusieurs bits peuvent être effectués sur la même valeur. Considérez le code suivant :

```
test    ax, 1
jnz     Bit0
test    ax, 2
jnz     Bit1
test    ax, 4
jnz     Bit3
etc.
```

Ce code peut être utilisé pour tester efficacement chaque bit du registre **ax** (ou de tout autre type d'opérande). Notez que vous ne pouvez pas utiliser la paire **test** / **cmp** pour faire le test d'une valeur spécifique dans une chaîne de bits (comme vous le feriez avec une paire **and** / **cmp**). Puisque **test** ne balaye pas tout nombre de bits non voulu, l'instruction **cmp** ferait la comparaison sur la valeur originale, au lieu que sur la valeur modifiée. Pour cette raison, on utilise normalement l'instruction **test** rien que pour voir si un bit est activé ou si un groupe de bits le sont. Certes, avec un processeur 80386 ou supérieur on peut également utiliser l'instruction **bt** pour tester des bits individuels dans une opérande.

Une autre application importante de l'instruction **test** est de comparer efficacement un registre avec zéro. L'instruction **test** suivante active le **zero flag** si et seulement si **ax** contient zéro (tout AND sur la même valeur produit la valeur originale ; ceci met le drapeau **zero** à 1 seulement si cette valeur est zéro) :

```
test    ax, ax
```

L'instruction **test** est plus courte que

```
cmp     ax, 0
```

ou

```
cmp     eax, 0
```

bien que ce ne soit pas mieux que

```
cmp     al, 0
```

Notez que vous pouvez utiliser les instructions **and** ou **or** pour tester une valeur de zéro de façon identique qu'avec **test**. Cependant, sur les processeurs à pipeline, comme le 80486 ou le Pentium, il y a moins de risques de provoquer un blocage du pipeline, puisqu'elle ne garde aucun résultat dans le registre de destination.

9.5.7 Tester les signes avec XOR

Vous souvenez-vous des maux associés avec les multiplications signées en multiprécision ? Vous aviez besoin de déterminer le signe du résultat, saisir la valeur absolue des opérandes, les multiplier et enfin ajuster le signe du résultat comme déterminé avant l'opération de multiplication. Le signe du produit de deux nombres est simplement un OR exclusif de leurs signes avant la multiplication. Par conséquent, vous pouvez utiliser l'instruction **xor** pour déterminer le signe du produit de deux nombres à précision étendue. Par exemple :

```
32x32 Multiplication :
                        mov     al, byte ptr Oprnd1+3
                        xor     al, byte ptr Oprnd2+3
                        mov     cl, al                  ; Enregistrement du signe
; Effectuer la multiplication ici (sans oublier de prendre la valeur absolue des
; deux opérandes avant d'effectuer la multiplication).
                        .
                        .
                        .
; Maintenant, ajuster le signe.
                        cmp     cl, 0                   ; Vérification du bit de signe
                        jns     ResultIsPos
; Négation du produit ici.
                        .
```

```
                        .
                        .
ResultIsPos:
```

9.6 Opérations de masquage

Un *masque* est une valeur utilisée pour forcer certains bits à valoir zéro ou un à l'intérieur d'une certaine autre valeur. Un masque, affecte typiquement certains bits dans une opérande (en les forçant à zéro ou à un) et laisse les autres bits inchangés. L'utilisation appropriée des masques vous permet d'*extraire* des bits d'une valeur, d'*insérer* des bits dans une valeur, et de compacter ou décompacter des valeurs. Les sections suivantes décrivent ces opérations en détail.

9.6.1 Opérations de masquage avec l'instruction AND

Si vous jetez un coup d'œil sur la table de vérité de l'opération AND du chapitre 1, vous noterez que si vous mettez une des deux opérandes (ou les deux) à zéro, le résultat est toujours faux. Si vous mettez une opérande à 1, le résultat est toujours la valeur de l'autre opérande. On peut exploiter cette propriété de l'opération AND, pour forcer sélectivement certains bits à valoir zéro dans une valeur sans affecter les autres bits. Ceci est nommé *masquage* de bits.

Comme exemple, considérez le code ASCII des chiffres "0" à "9". Leurs codes correspondent à la plage 30h..39h. Pour convertir un code ASCII en sa valeur numérique correspondante, il suffit de soustraire 30h du code ASCII. Et ceci peut être fait très facilement en effectuant un AND entre le code ASCII et la valeur 0Fh. Ceci balaie les quatre bits forts du code en produisant la valeur numérique correspondante. Vous auriez pu utiliser l'instruction de soustraction, mais il est d'usage conventionnel utiliser l'instruction and pour obtenir ce but.

9.6.2 Opérations de masquage avec l'instruction OR

Tout comme on peut utiliser l'instruction and pour forcer un certain nombre de bits sélectionnés à valoir zéro, on peut utiliser l'instruction OR pour forcer un certain nombre de bits à valoir un. Cette opération est communement appelée *masking on bit*.

Vous souvenez-vous l'opération de masquage effectuée à l'aide de and pour convertir le code ASCII d'un nombre en sa valeur numérique correspondante ? On peut utiliser l'instruction or pour produire l'effet contraire, à savoir, convertir une valeur numérique de 0 à 9 en son code ASCII correspondant. On fait ceci en effectuant un or entre la valeur numérique et la constante hexadécimale 30h.

9.7 Compacter et décompacter des types de données

L'une des utilisations fondamentales des opérations de décalage et de rotation est justement le compactage des données. Les types de données byte et word sont choisis le plus souvent, car l'architecture 80x86 est compatible avec ces tailles du point de vue matériel. Si vous n'avez pas besoin exactement de huit ou de seize bits, utiliser un octet ou un mot pour stocker votre donnée pourraît être un gaspillage d'espace. Le compactage vous donne l'habileté d'insérer deux ou plus valeurs dans un octet ou un mot. Le seul inconvénient en faisant ceci est la perte de performance, car cela prend du temps pour compacter et décompacter une donnée. Néanmoins, pour les applications où la vitesse ne joue pas un rôle critique (ou pour ces portions de l'application où la vitesse n'est pas un facteur clé), l'économie de l'usage de la mémoire peut justifier l'exigence de compacter des données.

Le type de données qui prend le plus d'avantage du compactage est le type booléen. En effet, un seul bit est requis pour représenter les états VRAI ou FAUX. Par conséquent, on peut garder jusqu'à huit valeurs booléennes dans un seul octet. Ceci représente une raison de compression de 8:1, donc, un tableau compacté de données booléennes, ne requiert que 1/8 d'espace d'un tableau correspondant non compacté (où chaque variable booléenne consomme un octet). Par exemple, ce tableau en Pascal :

```
B:packed arrray[0..31] of boolean;
```

ne demande que quatre octets, si compacté à raison d'une valeur par bit. Alors qu'une valeur par octet consommerait 32 octets.

Un tableau booléen compacté requiert deux opérations. Il faut d'abord insérer une valeur dans la variable compactée (souvent appelée *variable de champ compactée*) et ensuite il faut pouvoir extraire cette valeur.

Pour insérer une valeur dans un tableau booléen compacté, il faut aligner le bit source avec sa position dans l'opérande de destination et ensuite stocker ce bit dans cette opérande. Vous pouvez réaliser ceci avec une séquence d'instructions **and**, **or** et **shift**. La première étape consiste à masquer le bit correspondant dans la destination. Pour cela, on utilise une instruction **and**. Ensuite, l'opérande source est décalée de façon à être alignée avec sa position de destination. Et finalement l'opérande source est envoyée dans l'opérande de destination à l'aide d'un **or**. Par exemple, si vous voulez insérer le bit zéro du registre **ax** dans le cinquième bit du registre **cx**, on pourrait se servir du code suivant :

```
and     cl, 0DFh        ; Nettoyer le cinquième bit (de destination)
and     al, 1           ; Nettoyer tout dans al, sauf le bit source.
ror     al, 1           ; Déplacer dans le bit 7
shr     al, 1           ; Déplacer dans le bit 6
shr     al, 1           ; Déplacer dans le bit 5
or      cl, al
```

Ce code est un peu contourné. Il fait tourner la donnée vers la droite au lieu que la décaler dans la gauche, car ceci requiert moins d'instructions de décalage et de rotation.

Pour extraire une valeur booléenne, il vous suffira d'effectuer l'opération inverse. D'abord, vous déplacez le bit désiré dans le bit zéro et ensuite vous masquez tous les autres bits. Par exemple, pour extraire le contenu du cinquième bit du registre **cx** et le placer dans le bit zéro du registre **ax**, faites :

```
mov     al, cl
shl     al, 1   ; Bit 5 à bit 6
shl     al, 1   ; Bit 6 à bit 7
rol     al, 1   ; Bit 7 à bit 0
and     ax, 1   ; Tout nettoyer sauf le bit zéro.
```

Pour tester une variable dans un tableau compacté, vous n'avez pas besoin d'extraire le bit désiré, mais vous pouvez le tester à sa place. Par exemple, pour tester la valeur du bit cinq pour voir si elle est zéro ou un, utilisez :

```
test    cl, 00100000b
jnz     BitIsSet
```

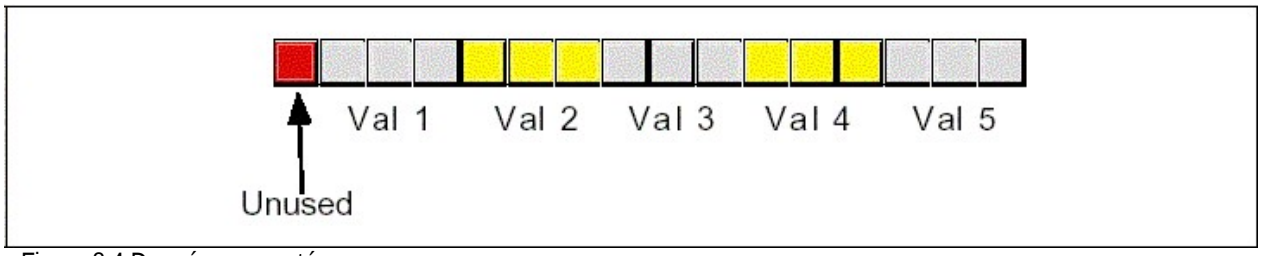


Figure 8.4 Donnée compactée

D'autres types de valeurs compactées peuvent être traités de façon similaire, sauf si vous n'avez besoin de travailler avec deux octets ou plus. Par exemple, supposez avoir compacté cinq champs différents de trois bits dans une valeur de seize bits, comme montré à la figure 8.4.

Si le registre **ax** contient la donnée à compacter dans val3, vous pouvez utiliser le code suivant pour insérer cette donnée dans le troisième champ :

```
mov     ah, al          ; Faire un shl de huit positions
shr     ax, 1           ; Décalage de deux positions vers la droite
shr     ax, 1
and     ax, 11100000b   ; Balayer les bits indésirés
and     DATA, 0FE3Fh    ; Mettre le champ de destination à zéro.
or      DATA, ax        ; Fusionner la nouvelle donnée dans le champ.
```

L'extraction est traitée de façon semblable. D'abord vous vous débarassez des bits indésirés et ensuite vous justifiez le résultat :

```
mov     ax, DATA
```

```
                and     ax, 1Ch
                shr     ax, 1
                shr     ax, 1
                shr     ax, 1
                shr     ax, 1
                shr     ax, 1
                shr     ax, 1
```

Ce code peut être amélioré en utilisant la séquence suivante :

```
                mov     ax, DATA
                shl     ax, 1
                shl     ax, 1
                mov     al, ah
                and     ax, 07h
```

On verra d'autres utilisations des données compactées tout au long de ce livre.

9.8 Tables

Le terme "table" a différentes significations selon les programmeurs. Pour la plupart des experts en assembleur, une table n'est autre chose qu'un tableau initialisé avec une valeur déterminée. Les programmeurs d'assembleur utilisent souvent ces tables pour calculer des fonctions complexes ou autrement lentes. Beaucoup de langages de très haut niveau (par exemple, SNOBOL4 et Icon) sopportent les tables en tant que type de données primitifs. Dans ces langages, les tables sont des tableaux accessibles par des index non entiers (par exemple, nombres décimaux, chaînes de caractères ou tout autre type). Dans cette section, on adoptera les tables du point de vue du programmeur en assembleur.

Une table est un tableau contenant des valeurs préinitialisées qui ne changent pas pendant l'exécution du programme. Une table peut être comparée à un tableau de la même façon qu'on compare une constante entière avec une variable entière. En assembleur, vous pouvez vous servir des tables pour plusieurs choses : calculer des fonctions, contrôler le flux du programme, ou simplement inspectionner quelque chose. En général, les tables fournissent un mécanisme rapide pour effectuer certaines opérations aux dépenses d'un certain espace dans votre programme (car le space superflu, stocke les données tabulaires). Dans les sections suivantes, on verra certaines des utilisations possibles qu'on peut faire des tables dans un programme en assembleur.

9.8.1 Calcul de fonctions par des tables de correspondance

Les tables peuvent effectuer toute sorte de choses en assembleur. Dans les langages de haut niveau, comme Pascal, c'est très facile de créer une formule qui calcule une certaine valeur. Une simple expression similaire à une expression mathématique est équivalente à une quantité considérable de code 80x86. Les programmeurs de l'assembleur tendent à calculer plusieurs valeurs moyennant des tables de correspondance, au lieu que à travers l'exécution d'une certaine fonction. Ce qui a l'avantage d'être plus facile et aussi plus efficace. Considérez la ligne Pascal suivante :

```
if (character >= 'a') and (character <= 'z') then character := chr(ord(character) - 32);
```

Cette ligne de Pascal convertit une variable de caractère en majuscule, si le caractère se trouve dans la plage 'a'..'z'. Le code 80x86 effectuant la même opération est :

```
                mov     al, character
                cmp     al, 'a'
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 05fh        ; Même opération que SUB AL,32
NotLower:       mov     character, al
```

Placez ce code dans une boucle imbriquée et vous auriez bientôt des problèmes de performance si vous n'utilisez pas une table de correspondance. Au contraire, l'utilisation d'une table de correspondance vous permet de réduire cette séquence d'instructions à quatre instructions :

```
                mov     al, character
                lea     bx, CnvrtLower
```

```
                xlat
                mov     character, al
```

CnvrtLower est une table de 256 octets contenant les valeurs 0 à 60h aux indices 0..60h, 41h à 5Ah aux indices 61h..7Ah et 7B à 0FFh aux indices 7Bh..0FFh. L'utilité des tables de correspondance se voit surtout à l'heure Souvent, se servir d'une table de référence aide à augmenter la vitesse de votre code.

À mesure que la complexité de la fonction augmente, les bénéfices des tables de correspondance augmentent de façon exponentielle. Même si vous n'allez probablement jamais utiliser une table de référence pour changer des minuscules en majuscules, considérez ce qui arrive si vous voulez changer de casse :

```
; Par simple calcul :
                mov     al, character
                cmp     al, 'a'
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 05fh
                jmp     ConvertDone
NotLower:       cmp     al, 'A'
                jb      ConvertDone
                cmp     al, 'Z'
                ja      ConvertDone
                or      al, 20h
ConvertDone:
                mov     character, al
```

Le code faisant la même chose via une table de correspondance, se réduit à :

```
                mov     al, character
                lea     bx, SwapUL
                xlat
                mov     character, al
```

Comme vous pouvez noter, quand vous calculez une fonction à l'aide d'une table de correspondance, peu importe la fonction, seule la table change, pas le code faisant la correspondance.

Mais les tables de correspondance ont aussi un problème majeur : les fonctions calculées de cette façon ont un domaine limité. Le domaine d'une fonction est la plage des valeurs d'entrée possibles que la fonction peut accepter (paramètres). Par exemple, la fonction convertissant la casse des caractères a comme domaine le jeu des 256 caractères ASCII.

D'autre part, une fonction comme SIN ou COS accepte l'ensemble des nombres réels comme valeurs d'entrée possibles. Certainement, un tel domaine est de beaucoup supérieur à celui du jeu de caractères. Si vous êtes en train de faire des calculs de fonctions à l'aide des tables de correspondance, il vous faudra limiter le domaine de la fonction à un petit ensemble. Et ceci parce que chaque élément du domaine de la fonction requiert une entrée dans la table. Et vous ne trouverez sûrement pas pratique d'implémenter une fonction se servant d'une table de correspondance et ayant comme domaine l'ensemble des nombres réels !

La plupart des tables de correspondance sont assez petites, normalement elles comptent de 10 à 128 éléments. Ne faites que rarement des tables ayant 1000 ou plus entrées. Les programmeurs n'ont pas la patience de créer (et de vérifier la justesse) des tables de 1000 éléments.

Une autre limitation des fonctions basées sur des tables de correspondance est que les éléments du domaine de la fonction doivent être assez contigus. Les tables de correspondance prennent une valeur d'entrée pour une fonction, l'utilisent comme index de la table et retournent la valeur à cette entrée de la table. Si vous ne passez à une fonction aucune autre valeur que 0, 100, 1000 et 10000, celle-ci semblerait un candidat idéal pour une implémentation via une table de correspondance. Cependant, la table demanderait 10001 éléments différents, à cause de la plage des valeurs d'entrée. Par conséquent, vous ne pouvez pas créer efficacement une telle fonction à l'aide d'une table de référence. Tout au long de cette section, on supposera que le domaine d'une fonction soit un ensemble pratiquement contigu de valeurs.

Les meilleures fonctions pouvant être implémentées par une table de correspondance sont celle dont le domaine se trouve toujours dans la plage 0..255 (ou dans un sous-ensemble de celle-ci). De telles fonctions sont efficacement implémentées par l'instruction xlat. Les routines de conversion de casse qu'on a vues sont un bon exemple. Toute fonction

de cette classe (dont le domaine et la plage se trouve entre 0 et 255) peut être calculée par les mêmes deux instructions (lea bx, table et xlat). La seule chose qui change toujours est la table de correspondance.

L'instruction xlat ne peut être utilisée de façon convéniente pour calculer une fonction si le domaine se trouve en dehors de la plage mentionnée. Il y a trois situations à considérer :

- Le domaine est hors de la plage 0..255, mais les valeurs à représenter sont dans cette plage.
- Le domaine est à l'intérieur de la plage, mais les valeurs à représenter sont hors de cette plage.
- Soit le domain que les valeurs de la table sont hors la plage.

On va considérer chacun des ces cas séparemment.

Si le domaine de la fonction est en dehors de la plage 0..255, mais la nature des valeurs de la fonction tombe dans cette plage, notre table de correspondance demandera plus de 256 éléments, mais on peut représenter chaque élément par un seul octet. Par conséquent, la table de correspondance peut être un tableau d'octets. Pour se servir de l'instruction xlat, les fonctions qui tombent dans cette classe sont les plus efficaces. L'appel de fonction suivant en Pascal :

```
B := Func(X);
```

où Func est :

```
function Func(X:word):byte;
```

peut être traduit avec le code suivant :

```
mov     bx, X
mov     al, FuncTable [bx]
mov     B, al
```

Ce code charge le paramètre de la fonction dans le registre bx, utilise cette valeur (dans la plage 0..??) comme index dans la table FuncTable, charge l'octet à cet emplacement et garde le résultat dans B. Evidemment, cette table doit contenir un entrée valide pour chaque valeur de X possible. Par exemple, supposez que vous vouliez représenter une position de curseur à l'écran dans la plage 0..1999 (dans un affichage vidéo 80x25, il y a 2000 positions de caractère) dans sa coordonnée X ou Y de l'écran. Vous pouvez facilement déterminer la coordonnée X par la fonction X:=Posn mod 80 et la coordonnée Y par la formule Y:=Posn div 80 (où Posn est la position du curseur à l'écran), à l'aide du code :

```
mov     bl, 80
mov     ax, Posn
div     bx
; X est maintenant dans AH et Y dans AL
```

Cependant, l'instruction div dans l'architecture 80x86 est très lente. Si vous aviez besoin d'effectuer ce calcul pour chaque caractère que vous écrivez à l'écran, vous dégraderiez sérieusement la vitesse de votre code d'affichage vidéo. Le code suivant, réalisant ces deux fonctions, améliore considérablement les performances :

```
mov     bx, Posn
mov     al, YCoord[bx]
mov     ah, XCoord[bx]
```

Si le domaine d'une fonction est dans la plage 0..255 mais les valeurs à représenter sont en dehors de cette plage, votre table de correspondance contiendra 256 éléments ou moins, mais la taille de chaque entrée de votre table sera de deux ou plus octets. Si autant que le domaine que les valeurs sont hors de cette plage, chaque entrée de la table réquerra deux ou plus octets pour être représentée et la table contiendra plus de 256 éléments.

En rappel de la formule du chapitre 4 pour indexer des éléments dans un tableau à une dimension (duquel une table représente un cas spécial) :

```
Adresse := Base + index * taille
```

Si les éléments de la fonction demandent deux octets, alors l'index devra être multiplié par deux avant d'être indexé dans la table. Sinon, si chaque entrée requiert plus de deux octets, alors l'index devra être multiplié par la taille de chaque de chaque élément avant de s'en servir comme index dans la table. Par exemple, supposez avoir une fonction F(x) définie par la pseudo déclaration suivante en Pascal :

```
function F(x:0..999):word;
```

Vous pouvez facilement créer cette fonction à l'aide du code suivant (et, bien entendu, la table appropriée) :

```
                mov     bx, X   ; Obtenir la valeur d'entrée de la fonction et
                shl     bx, 1   ; la convertir en un index de type mot dans F.
                mov     ax, F[bx]
```

L'instruction shl multiplie l'index par deux et fournit l'indexage correct pour une table dont les éléments sont de type word. Toute fonction où le domaine est petit et autant contigu que possible est un bon candidat pour être calculée à l'aide d'une table de correspondance. Dans certains cas, des domaines non contigus sont acceptables aussi, au moins dans les limites où le domaine peut être limité à un ensemble approprié de valeurs. De telles opérations sont nommées *conditionnelles* et sont l'objet de la section suivante.

9.8.2 Conditionnement de domaine

Le conditionnement de domaine consiste à prendre un ensemble de valeurs dans le domaine d'une fonction et les adapter de façon à être plus acceptables comme entrées de la fonction. Considérez la fonction suivante :

$$\sin x = \{\sin x \mid x \in [-2\pi, 2\pi]\}$$

Cette représentation de fonction SIN(X) est équivalente à la fonction mathématique *sin x* où

$$-2\pi \leq x \leq 2\pi$$

Comme nous savons tous, la fonction sinus est une fonction circulaire acceptant toute valeur d'entrée réelle. Cependant, cette formule utilisée pour calculer le sinus, n'accepte qu'un petit ensemble de cette plage.

Cette limitation de plage ne présente pas de véritables problèmes, en calculant simplement SIN(X mod (2*pi)), on peut calculer le sinus de toute valeur d'entrée. Modifier une valeur d'entrée de sorte à pouvoir calculer facilement une fonction s'appelle *conditionner l'entrée*. Dans l'exemple ci-dessus, on a calculé X mod 2*pi et utilisé le résultat comme entrée pour la fonction sin. Ceci arrondit X au domaine que sin requiert sans affecter le résultat. On peut appliquer cette technique aussi pour les tables de correspondance. En effet, arrondir l'index pour traiter des entrées de type word est une forme de conditionnement. Considérez la fonction suivante en Pascal :

```
function val(x:word):word; begin
       case x of
0: val := 1;
       1: val := 1;
       2: val := 4;
       3: val := 27;
       4: val := 256;
       otherwise val := 0;
       end;
end;
```

Cette fonction calcule une certaine valeur de x dans la plage 0..4 et retourne 0 si x est en dehors de cette plage. Puisque x peut prendre jusqu'à 65536 valeurs différentes (étant une variable de 16 bits), créer une table acceptant 65536 valeurs quand seuelement les premières cinq valeurs sont différentes de zéro paraît insensé. Cependant, on peut encore utiliser une table de correspondance si on conditionne les entrées. Le code assembleur suivant présente ce principe :

```
                xor     ax, ax  ; AX := 0, en supposant X > 4.
                mov     bx, x
                cmp     bx, 4
                ja      ItsZero
                shl     bx, 1
                mov     ax, val[bx]
ItsZero:
```

Ce code vérifie si x est en dehors de la plage 0..4. Si c'est le cas, il donne automatiquement la valeur 0 à ax, sinon il cherche la valeur correspondante dans la table de correspondance. Avec cette technique, vous pouvez implémenter diverses fonctions qui seraient autrement impraticables.

9.8.3 Génération de tables

Un grand problème quand on utilise des tables de référence est la création de la table elle-même. Ceci est vrai surtout lorsqu'il y a un grand nombre d'entrées dans la table. Imaginez-vous à devoir placer un grand nombre de données dans la table, devoir les insérer une à une et finalement, après un travail laborieux de typing, les vérifier une à la fois pour voir si elles ont été entrées correctement. C'est un travail mortel qui fait consommer beaucoup de temps. Pour certaines tables, ceci est incontournable. Pour d'autres, il existe des raccourcis, comme utiliser l'ordinateur pour générer des tables pour vous. Un exemple, constitue peut-être la meilleure façon pour le décrire. Considérez la modification suivante de la fonction sinus :

$$(\sin x) \times r = \left\langle \frac{(r \times (1000 \times \sin x))}{1000} \right| x \in [0,$$

Dans cette fonction, x est un entier entre 0 et 359 et r est un entier. L'ordinateur peut facilement calculer ceci à l'aide du code suivant :

```
                mov     bx, x
                shl     bx, 1
                mov     ax, Sines [bx]  ; Obtenir SIN(X)*1000
                mov     bx, r           ; Calculer R*(SIN(X)*1000)
                mul     bx
                mov     bx, 1000        ; Calculer R*(SIN(X)*1000))/1000
                div     bx
```

Notez que la multiplication et la division entières ne sont pas associatives. Vous ne pouvez pas supprimer la multiplication par 1000 et la division par 1000 juste comme on fait en mathématiques. De plus, ce code doit calculer la fonction exactement en suivant cet ordre. Tout ce dont on a besoin pour compléter cette fonction est une table de 360 valeures différentes qui correspondent au sinus de l'angle (en degrés) multiplié par 1000. Insérer une telle table dans un programme en assembleur est une tâche extrêmement ennuyeuse et vous avez de fortes chances de commettre plusieurs fautes en insérant et aussi en vérifiant les données. Néanmoins, vous pouvez directement charger un programme de générer cette table. Considérez le code Pascal suivant :

```
program maketable;
var     i:integer;
        r:integer;
        f:text;
begin
        assign(f,'sines.asm');
        rewrite(f);
        for i := 0 to 359 do begin
                r := round(sin(I * 2.0 * pi / 360.0) * 1000.0);
                if (i mod 8) = 0 then begin
                        writeln(f);
                        write(f,' dw ',r);
                end
                else write(f,',',r);
        end;
        close(f);
end.
```

Ce programme produit la sortie suivante :

```
dw 0,17,35,52,70,87,105,122
dw 139,156,174,191,208,225,242,259
dw 276,292,309,326,342,358,375,391
dw 407,423,438,454,469,485,500,515
dw 530,545,559,574,588,602,616,629
dw 643,656,669,682,695,707,719,731
dw 743,755,766,777,788,799,809,819
dw 829,839,848,857,866,875,883,891
dw 899,906,914,921,927,934,940,946
dw 951,956,961,966,970,974,978,982
dw 985,988,990,993,995,996,998,999
dw 999,1000,1000,1000,999,999,998,996
```

```
dw 995,993,990,988,985,982,978,974
dw 970,966,961,956,951,946,940,934
dw 927,921,914,906,899,891,883,875
dw 866,857,848,839,829,819,809,799
dw 788,777,766,755,743,731,719,707
dw 695,682,669,656,643,629,616,602
dw 588,574,559,545,530,515,500,485
dw 469,454,438,423,407,391,375,358
dw 342,326,309,292,276,259,242,225
dw 208,191,174,156,139,122,105,87
dw 70,52,35,17,0,-17,-35,-52
dw -70,-87,-105,-122,-139,-156,-174,-191
dw -208,-225,-242,-259,-276,-292,-309,-326
dw -342,-358,-375,-391,-407,-423,-438,-454
dw -469,-485,-500,-515,-530,-545,-559,-574
dw -588,-602,-616,-629,-643,-656,-669,-682
dw -695,-707,-719,-731,-743,-755,-766,-777
dw -788,-799,-809,-819,-829,-839,-848,-857
dw -866,-875,-883,-891,-899,-906,-914,-921
dw -927,-934,-940,-946,-951,-956,-961,-966
dw -970,-974,-978,-982,-985,-988,-990,-993
dw -995,-996,-998,-999,-999,-1000,-1000,-1000
dw -999,-999,-998,-996,-995,-993,-990,-988
dw -985,-982,-978,-974,-970,-966,-961,-956
dw -951,-946,-940,-934,-927,-921,-914,-906
dw -899,-891,-883,-875,-866,-857,-848,-839
dw -829,-819,-809,-799,-788,-777,-766,-755
dw -743,-731,-719,-707,-695,-682,-669,-656
dw -643,-629,-616,-602,-588,-574,-559,-545
dw -530,-515,-500,-485,-469,-454,-438,-423
dw -407,-391,-375,-358,-342,-326,-309,-292
dw -276,-259,-242,-225,-208,-191,-174,-156
dw -139,-122,-105,-87,-70,-52,-35,-17
```

Il n'y a pas de doute : c'est plus faice écrire le programme en Pascal pour qu'il génère ce code, plutôt qu'écrire manuellement les données elles-mêmes ! Ce petit exemple montre aussi comme le Pascal peut venir en aide au programmeur d'assembleur...

9.9 Exemples de programmes

Les programmes exemples de ce chapitre montrent divers concepts importants, en incluant l'arithmétique en précision étendue et les opérations logiques, les évaluations d'expressions arithmétiques, les évaluations d'expressions booléennes et le compactage / décompactage des données.

9.9.1 Conversion d'expressions arithmétiques en assembleur

Le programme suivant (Pgm9_1.asm) donne certains exemples de conversion d'expressions arithmétiques en code assembleur :

```
; Pgm9_1.ASM
;
; Plusieurs exemples montrant comment convertir diverses expressions arithmétiques en
; assembleur.

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

dseg            segment         para public 'data'

; Les diverses variables utilisées par ce programme.
```

```
u               word    ?
v               word    ?
w               word    ?
x               word    ?
y               word    ?
dseg            ends

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

; GETI lit une variable entière en entrée et elle retourne sa valeur dans le
; registre AX.
geti            textequ <call _geti>
_geti           proc
                push    es
                push    di
                getsm
                atoi
                free
                pop     di
                pop     es
                ret
_geti           endp


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

                print
                byte    "Abitrary expression program",cr,lf
                byte    "---------------------------",cr,lf
                byte    lf
                byte    "Enter a value for u: ",0
                geti

                mov     u, ax

                print
                byte    "Enter a value for v: ",0
                geti
                mov     v, ax

                print
                byte    "Enter a value for w: ",0
                geti
                mov     w, ax


                print
                byte    "Enter a non-zero value for x: ",0
                geti
                mov     x, ax

                print
                byte    "Enter a non-zero value for y: ",0
                geti
                mov     y, ax

; Bon, calculer Z := (X+Y)*(U+V*W)/X et afficher le résultat.

                print
                byte    cr,lf
```

```
                byte    "(X+Y) * (U+V*W)/X is ",0
                mov     ax, v                   ; Calculer V*W
                imul    w                       ; et l'additionner à U
                add     ax, u
                mov     bx, ax          ; Enregistrer dans un emplacement temporaire.
                mov     ax, x           ; Calculer X+Y et multiplier cette somme par
                add     ax, y           ; le résultat ci-dessus
                imul    bx              ; puis diviser le tout
                idiv    x               ; par X.

                puti
                putcr

; Calculer ((X-Y*U) + (U*V) - W)/(X*Y)

                print
                byte    "((X-Y*U) + (U*V) - W)/(X*Y) = ",0

                mov     ax, y           ; Calculer d'abord y*u
                imul    u
                mov     dx, X           ; Maintenant X-Y*U
                sub     dx, ax
                mov     cx, dx          ; Enregistrer dans empl. temp.

                mov     ax, u           ; U*V
                imul    V
                add     cx, ax          ; (X-Y*U) + (U*V)

                sub     cx, w           ; ((X-Y*U) + (U*V) - W)

                mov     ax, x           ; (X*Y)
                imul    y

                xchg    ax, cx
                cwd                     ; NUEMRATEUR/(X*Y)
                idiv    cx

                puti
                putcr

Quit:           ExitPgm                 ; macro du DOS pour sortir.
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
```

9.9.2 Exemple d'opérations booléennes

Le programme d'exemple suivant (Pgm9_2.asm) montre comment manipuler des valeurs booléennes dans un programme assembleur. Il donne également un exemple du théorème de DeMorgan.

```
; Pgm9_2.ASM
;
; Ce programme montre le théorème de DeMorgan et plusieurs autres opérations logiques.

                .xlist
```

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

dseg            segment         para public 'data'

; Voici les variables d'entrée booléennes pour diverses fonctions que nous allons
; tester.

a               byte    0
b               byte    0

dseg            ends

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

; Get0or1 - Lit un "0" ou un "1" en entrée et il retourne sa valeur dans le
; registre AX.

get0or1         textequ         <call _get0or1>
_get0or1        proc
                push    es
                push    di

                getsm
                atoi
                free

                pop     di
                pop     es
                ret
_get0or1        endp

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

                print
                byte    "Demorgan's Theorems",cr,lf
                byte    "-------------------",cr,lf
                byte    lf
                byte    "According to Demorgan's theorems, all results "
                byte    "between the dashed lines",cr,lf
                byte    "should be equal.",cr,lf
                byte    lf
                byte    "Enter a value for a: ",0

                get0or1
                mov     a, al

                print
                byte    "Enter a value for b: ",0
                get0or1
                mov     b, al

                print
                byte    "--------------------------------",cr,lf
                byte    "Computing not (A and B): ",0

                mov     ah, 0
```

```
mov     al, a
and     al, b
xor     al, 1   ; Opération de NOT booléen.

puti
putcr

print
byte    "Computing (not A) OR (not B): ",0
mov     al, a
xor     al, 1
mov     bl, b
xor     bl, 1
or      al, bl
puti

print
byte    cr,lf
byte    "---------------------------------",cr,lf
byte    "Computing (not A) OR B: ",0
mov     al, a
xor     al, 1
or      al, b
puti

print
byte    cr,lf
byte    "Computing not (A AND (not B)): ",0
mov     al, b
xor     al, 1
and     al, a
xor     al, 1
puti

print
byte    cr,lf
byte    "---------------------------------",cr,lf
byte    "Computing (not A) OR B: ",0
mov     al, a
xor     al, 1
or      al, b
puti

print
byte    cr,lf
byte    "Computing not (A AND (not B)): ",0
mov     al, b
xor     al, 1
and     al, a
xor     al, 1
puti

print
byte    cr,lf
byte    "---------------------------------",cr,lf
byte    "Computing not (A OR B): ",0
mov     al, a
or      al, b
xor     al, 1
puti

print
byte    cr,lf
```

```
                byte    "Computing (not A) AND (not B): ",0
                mov     al, a
                xor     al, 1
                and     bl, b
                xor     bl, 1
                and     al, bl
                puti

                print
                byte    cr,lf
                byte    "--------------------------------",cr,lf
                byte    0

Quit:           ExitPgm         ; Macro du DOS pour sortir du programme.

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
```

9.9.3 Entrées/Sorties de 64 bits

Ce programme d'exemple (Pgm9_3) montre comment lire et écrire des entiers de 64 bits. Il fournit les routines ATOU64 et PUTU64 vous permettant de convertir une chaîne de chiffres en un entier non signé de 64 bits et il retourne une valeur sous forme de chaîne décimale de chiffres.

```
; Pgm9_3.ASM
;
; Ce programme d'exemple fournit deux procédure qui lisent et écrivent des entiers
; non signés de 64 bits sur un processeur 80386 ou supérieur.

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

                .386
                option  segment:use16

dp              textequ         <dword ptr>
byp             textequ         <byte ptr>

dseg            segment         para public 'data'

; Acc64 est une valeur de 64 bits que la routine ATOU64 utilise pour permettre l'entrée
; d'une valeur de 64 bits.

Acc64           qword   0

; La variable quotient stocke le résultat de la division entre la valeur courante de
; PUTU et dix.

Quotient        qword   0

; NumOut stocke la chaîne de chiffres créée par la routine PUTU64.
```

```
NumOut          byte    32 dup (0)

; Une chaîne de test pour la routine ATOI64 :

LongNumber      byte    "1234567890123456789",0

dseg ends

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

; ATOU64 -      Au commencement de l'exécution, ES:DI pointe sur une
;               chaîne contenant une séquence de chiffres. Cette routine convertit
;               cette chaîne en un entier de 64 bits et retourne une valeur entière non
;               signée dans EDX:EAX.
;
;               Cette routine utilise l'algorithme :
;
;               Acc := 0
;               while digits left
;
;               Acc := (Acc * 10) + (Current Digit - '0')
;               Move on to next digit
;
;               endwhile

ATOU64          proc    near
                push    di              ; L'enregistrer parce qu'on va le modifier.
                mov     dp Acc64, 0     ; Initialisation de notre accumulateur.
                mov     dp Acc64+4, 0

; Une fois obtenus les chiffres, traiter la chaîne d'entrée :

                sub     eax, eax        ; Zero out eax's H.O. 3 bytes.
WhileDigits:    mov     al, es:[di]
                xor     al, '0'         ; Convertir '0'..'9' -> 0..9
                cmp     al, 10          ; et tout autre chiffre > 9.
                ja      NotADigit

                ; Multiplier Acc64 par dix. Pour cela utiliser des décalages et des
                ; additions :

                shl     dp Acc64, 1     ; Calculer Acc64*2
                rcl     dp Acc64+4, 1

                push    dp Acc64+4      ; Calculer Acc64*2
                push    dp Acc64

                shl     dp Acc64, 1     ; Calculer Acc64*4
                rcl     dp Acc64+4, 1
                shl     dp Acc64, 1     ; Calculer Acc64*8
                rcl     dp Acc64+4, 1

                pop     edx             ; Calculer Acc64*10 en tant que
                add     dp Acc64, edx   ; Acc64*2 + Acc64*8
                pop     edx
                adc     dp Acc64+4, edx

; Ajouter l'équivalent numérique du chiffre courant.
; Souvenez vous que les trois mots forts de eax contiennent zéro.

                add     dp Acc64, eax   ; Additionner ce chiffre
```

```
                inc     di              ; Procéder au caractère suivant.
                jmp     WhileDigits     ; Répéter pour tous les chiffres.

; Retourner la valeur de 64 bits dans eax.

NotADigit:      mov     eax, dp Acc64
                mov     edx, dp Acc64+4
                pop     di
                ret
ATOU64 endp

; PUTU64 -      Au commencement de l'exécution, EDX:EAX contient une valeur non
;               signée de 64 bits. Cette routine produit en sortie une chaîne
;               de chiffres décimaux fournissant la représentation décimale de
;               cette valeur.
;
;               ce code utilise l'algorithme suivant :
;
;               di := 30;
;               while edx:eax <> 0 do
;
;                       OutputNumber[di] := digit;
;                               edx:eax := edx:eax div 10
;                               di := di - 1;
;
;                       endwhile
;                       Output digits from OutNumber[di+1]
;                               through OutputNumber[30]

PUTU64          proc
                push    es
                push    eax
                push    ecx
                push    edx
                push    di
                pushf

                mov     di, dseg        ; C'est ici qu'ira la chaîne de sortie.
                mov     es, di
                lea     di, NumOut+30   ; Enregistrement des caractères dans la chaîne
                std                     ; à l'envers.
                mov     byp es:[di+1],0 ; Produire un 0 de terminaison.

; Enregistrer la valeur à afficher de façon à pouvoir la diviser par dix à l'aide
; d'un opérateur de division de précision étendue.

                mov     dp Quotient, eax
                mov     dp Quotient+4, edx

; Commencement de la conversion du nombre en une chaîne de chiffres.

                mov     ecx, 10                 ; Valeur à diviser pour.
DivideLoop:     mov     eax, dp Quotient+4      ; Effectuer une division 64/32 bits.
                sub     edx, edx
                div     ecx
                mov     dp Quotient+4, eax

                mov     eax, dp Quotient
                div     ecx
                mov     dp Quotient, eax

; Maintenant edx (dl, en vérité) contient le reste de la division par dix,
; de sorte que dl est dans la plage 0..9. Il ne reste qu'à le convertir en un
```

```
; caractère ASCII et à l'enregister.

                mov     al, dl
                or      al, '0'
                stosb

; Vérifier si le résultat est nul. Dans ce cas, on peut quitter.

                mov     eax, dp Quotient
                or      eax, dp Quotient+4
                jnz     DivideLoop

OutputNumber:   inc     di
                puts
                popf
                pop     di
                pop     edx
                pop     ecx
                pop     eax
                pop     es
                ret
PUTU64          endp

; Le programme principal constitue une démonstration simple de l'usage des deux
; routines qu'on vient de créer

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

                lesi    LongNumber
                call    ATOU64
                call    PutU64
                printf
                byte    cr,lf
                byte    "%x %x %x %x",cr,lf,0
                dword   Acc64+6, Acc64+4, Acc64+2, Acc64

Quit:           ExitPgm         ; Macro du DOS pour sortir.
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
```

9.9.4 Types pour compacter et décompacter des données

Ce programme d'exemple démontre comment compacter et décompacter les données à l'aide du type de données Date, présenté au Chapitre 1.

```
; Pgm9_4.ASM
;
;       Ce programme montre comment compacter et décompacter des types de date.
;       Il lit une valeur mois, jour et année.
```

```
;       Il compacte ensuite ces valeurs selon le format présenté dans le chapitre un.
;       Finalement, il décompacte ces données et appelle la routine DTOA de la
;       bibliothèque stdlib pour afficher le résultat sous forme de texte.

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

dseg            segment para public 'data'

Month           byte    ?       ; Valeur de mois (1-12)
Day             byte    ?       ; Valeur de jour (1-31)
Year            byte    ?       ; Valeur d'année (80-99)
Date            word    ?       ; Ici les données compactées.
dseg            ends

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

; GETI- Lit une variable entière de l'utilisateur et retourne sa valeur dans le
; registre AX.

geti            textequ <call _geti>
_geti           proc
                push    es
                push    di

                getsm
                atoi
                free
                pop     di
                pop     es
                ret
_geti           endp

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

                print
                byte    "Date Conversion Program",cr,lf
                byte    "-----------------------",cr,lf
                byte    lf,0

; Obtenir le mois de l'utilisateur.
; Faire une simple vérification pour s'assurer que cette valeur est dans la plage
; 1-12. Si ce n'est pas le cas, répéter l'invite de saisie.

GetMonth:       print
                byte    "Enter the month (1-12): ",0
                geti
                mov     Month, al
                cmp     ax, 0
                je      BadMonth
                cmp     ax, 12
                jbe     GoodMonth
BadMonth:       print
                byte    "Illegal month value, please re-enter",cr,lf,0
                jmp     GetMonth
```

```
GoodMonth:

; Une fois le mois lu, lire le jour et, encore une fois, faire une simple vérification
; sur la justesse de la donnée. Notez que ce code vérifie seulement si le jour est dans
; la plage 1-31 et ne contrôle pas ces mois qui ont 28, 29 ou 30 jours.

GetDay:         print
                byte    "Enter the day (1-31): ",0
                geti
                mov     Day, al
                cmp     ax, 0
                je      BadDay
                cmp     ax, 31
                jbe     GoodDay
BadDay:         print
                byte    "Illegal day value, please re-enter",cr,lf,0
                jmp     GetDay

GoodDay:

; Et, finalement, obtenir l'année. Cette vérification est un peu plus sophistiquée.
; Si l'utilisateur saisit une valeur dans la plage 1980-1999, elle sera automatiquement
; convertie en 80-99. Toutes les autes dates hors de cette plage ne sont pas acceptées.

GetYear:        print
                byte    "Enter the year (80-99): ",0
                geti
                cmp     ax, 1980
                jb      TestYear
                cmp     ax, 1999
                ja      BadYear

                sub     dx, dx          ; Extension de zéro à 32 bits.
                mov     bx, 100
                div     bx              ; Calculer année mod 100.
                mov     ax, dx
                jmp     GoodYear
TestYear:       cmp     ax, 80
                jb      BadYear
                cmp     ax, 99
                jbe     GoodYear

BadYear:        print
                byte    "Illegal year value. Please re-enter",cr,lf,0
                jmp     GetYear
GoodYear:       mov     Year, al

; Maintenant que toutes les valeurs ont été saisies, les compacter dans le format de
; 16 bits suivant :
;
;       bit 15      8 7     0
;           |       | |     |
;           MMMMDDDD DYYYYYYY

                mov     ah, 0
                mov     bh, ah
                mov     al, Month       ; Placer le mois dans les positions 12..15
                mov     cl, 4
                ror     ax, cl

                mov     bl, Day         ; Placer le jour dans les positions 7..11
                mov     cl, 7
                shl     bx, cl
```

```
                or      ax, bx          ; Créer MMMMDDDD D0000000
                or      al, Year        ; Créer MMMMDDDD DYYYYYYY
                mov     Date, ax        ; Enregistrer le résultat compacté.

; Afficher le résultat compacté (en format hexadécimal) :

                print
                byte    "Packed date = ",0
                putw
                putcr


; Le code suivant montre comment décompacter cette date et la transformer en un
; format pouvant être utilisé par la bibliothèque standard LDTOAM.

                mov     ax, Date        ; Extraire le mois
                mov     cl, 4
                shr     ah, cl
                mov     dh, ah          ; LDTOAM nécessite le mois dans DH.

                mov     ax, Date        ; Obtenir ensuite le jour.
                shl     ax, 1
                and     ah, 11111b
                mov     dl, ah          ; Qui doit être dans DL.

                mov     cx, Date        ; Maintenant l'année.
                and     cx, 7fh         ; Balayer tout sauf les bits contenant la
                                        ; valeur.

                print
                byte    "Date: ",0
                LDTOAM                  ; Convertir en chaîne
                puts
                free
                putcr

Quit:           ExitPgm                 ; Macro du DOS pour terminer le programme.
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
```

9.10 Exercices de laboratoire

Dans ce laboratoire, vous allez effectuer les activités suivantes :

- Utiliser CodeView pour insérer des points d'arrêt (breaks) dans un programme et localiser certaines erreurs.
- Utiliser CodeView pour faire la trace à travers certaines sections d'un programme et découvrir des problèmes avec celui-ci.
- Utiliser CodeView pour faire la trace à travers du code écrit par vous pour en vérifier la justesse et observer la séquence et le calcul pas à pas.

9.10.1 Déboguer des programmes avec CodeView

On a déjà eu l'opportunité d'utiliser CodeView dans des chapitres antérieurs pour observer l'état de la machine (valeurs des registres et de la mémoire), entrer des programmes simples et effectuer d'autres tâches mineures. Dans cette section, on explorera l'une des capacités les plus importantes de CodeView : le débogage. Cette section décrit trois des caractéristiques de CodeView qu'on avait ignorées jusqu'à maintenant, à savoir, les points d'arrêt (*breaks points*), les opérations d'inspection (*watches*) et la trace du code. Il s'agit d'outils très importants pour détecter les erreurs logiques dans vos programmes en assembleur.

La *trace* du code vous permet d'exécuter le code d'un programme une ligne à la fois, en permettant ainsi d'observer les résultats produits par chaque instruction. Certains programmeurs se réfèrent à cette opération avec le terme de *single stepping*, car elle vous permet d'exécuter un programme d'un pas à la fois. Mais, la véritable utilité de la trace est d'observer le résultats de chaque portion de code, noter toutes les erreurs de logique et déduire pourquoi une séquence ne produit pas les résultats souhaités.

CodeView fournit deux commandes faciles à utiliser pour tracer un programme. Presser F8 a pour effet d'exécuter l'instruction suivante, de mettre à jour tous les registres et les emplacements de mémoire affectés par l'exécution de cette instruction et d'arrêter l'exécution du programme sur l'instruction suivante (qui sera exécutée par une nouvelle pression de la touche F8, et ainsi de suite). Dans le cas que l'instruction courante soit un `call`, un `int` ou une autre instruction de transfert de contrôle, CodeView effectue ce transfert sur l'emplacement cible et affiche l'instruction à cet emplacement.

La seconde commande de trace est la commande *step*. On peut l'exécuter en pressant F10. Cette commande exécute l'instruction courante et arrête avant l'exécution de l'instruction suivante. Pour la plupart des instructions, les commandes *trace* et *step* font exactement la même chose, cependant, pour les instructions de transfert de contrôle, la commande *trace* fait la trace aussi à l'intérieur des sous-programmes et en suit toutes les instructions, alors que la commande *step* permet au CPU d'exécuter les sous-programmes à toute vitesse et d'arrêter directement sur la ligne qui suit l'appel. Ceci vous permet de passer rapidement dans le corps d'une sous-routine sans avoir à tracer pour chaque instruction contenue dans celle-ci. Normalement, on utilise la commande *trace* la plupart des fois et la commande *step* quand on n'a pas besoin de faire la trace d'une sous-routine, donc à l'occurrence d'une instruction `call` ou `int`. La commande *step* peut également produire des effets imprévus sur d'autres instructions de transfert de contrôle, comme `loop` ou les sauts conditionnels.

La fenêtre de commande de CodeView fournit également deux commandes permettant la même chose, la commande "T" (pour *trace*) et la commande "P" (pour *step*).

L'un des inconvénients majeurs du traçage, c'est sa lenteur. Même en maintenant continuellement pressée la touche F8, vous n'arriverez pas à exécuter plus de 10 ou de 20 instructions par seconde. Ce qui est au moins un million de fois plus lent qu'une exécution normale à toute vitesse d'un PC. Si un programme doit exécuter divers milliers d'instructions avant même d'atteindre le point problématique, il vous faudra exécuter la commande *trace* trop longtemps avant qu'elle vous soit d'aide.

C'est dans ce contexte que les points d'arrêt viennent en aide. Un point d'arrêt est un point où le contrôle du programme revient au débogueur. Ce qui vous permet d'exécuter un programme à toute vitesse jusqu'à un point spécifique de celui-ci (le point d'arrêt) ; l'exécution du programme reste suspendue, ce qui vous permet d'inspectionner l'état du programme à ce moment de son exécution et d'y effectuer éventuellement une trace à partir de ce point. Les points d'arrêt sont probablement l'outil le plus important pour localiser des erreurs. Puisqu'ils sont aussi utiles, ce n'est pas surprenant que CodeView fournisse un jeu très riche de commandes de manipulation des points d'arrêt.

Il y a trois commandes de frappe vous permettant d'exécuter votre programme à toute vitesse, en y plaçant des points d'arrêt. La commande F5 (*run*), commence une exécution à partir de CS:IP. Si vous n'avez placé aucun point d'arrêt, votre programme exécutera jusqu'à sa fin. Si vous êtes intéressés à arrêter votre programme sur un certain point donné, alors il vous faudra placer un point d'arrêt *avant* l'exécution de la commande *run*.

Presser F5 produit le même résultat que la commande "G" (*go*) dans la fenêtre des commandes. La commande *go* est un peu plus puissante, car elle vous permet de lancer un programme en spécifiant un point d'arrêt en même temps. Cette commande prend la forme :

G

G adresse_du_point_d_arrêt

La touche F7 produit une exécution à toute vitesse jusqu'à la position courante du curseur. Ceci permet en fait de placer un point d'arrêt *non formel*. Il va de soi que pour utiliser cette commande il vous faut placer le curseur là où vous intéresse

et ensuite presser F7. CodeView placera un point d'arrêt à cette instruction et exécutera le programme à toute vitesse jusqu'à l'occurrence de celle-ci.

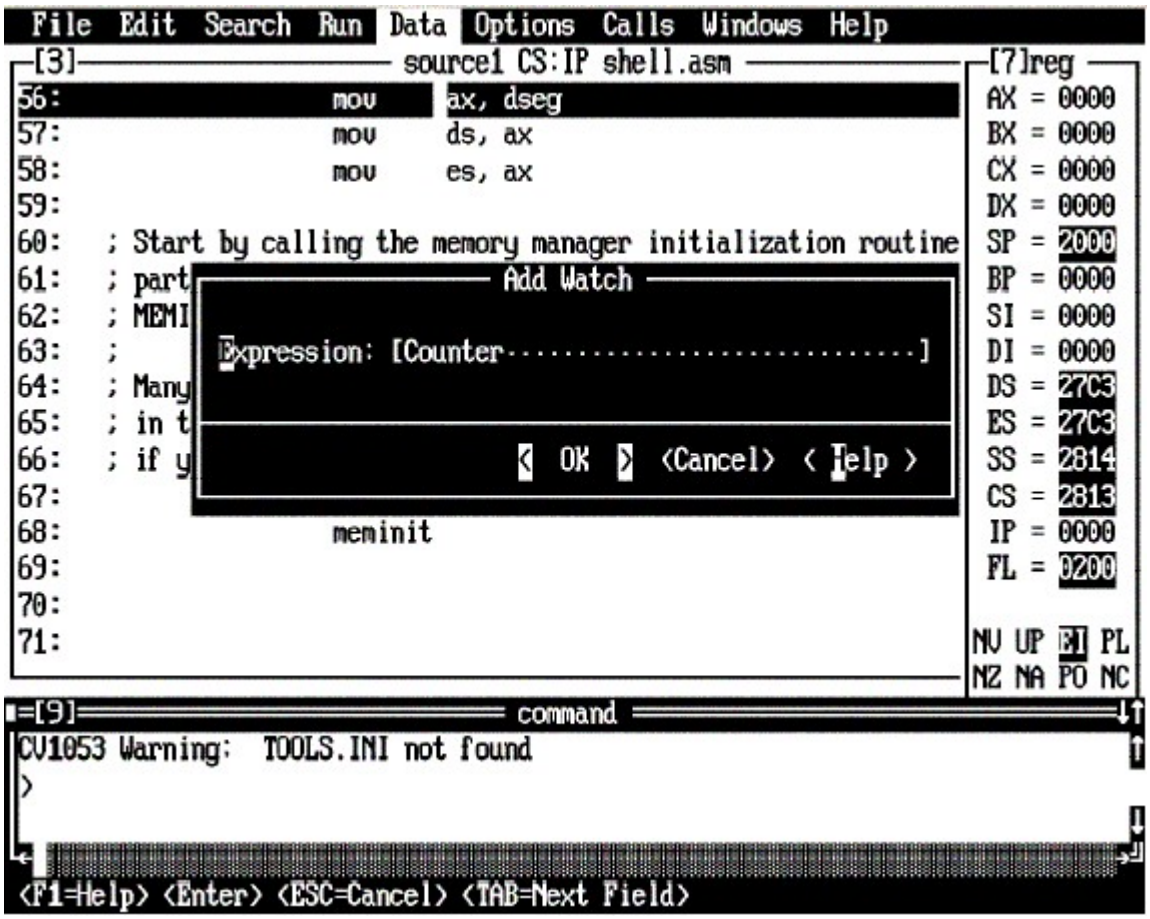
Un point d'arrêt non formel est un un point d'arrêt temporaire qui n'a plus d'effet une fois que CodeView reprend le contrôle. Une fois reprise l'exécution du programme, tous les points d'arrêts non formels sont annulés par CodeView. Il vous faudra redéfinir ces points d'arrêt si vous aurez de nouveau besoin d'arrêter votre programme sur ces mêmes points. Notez que cela arrive même quand le programme cesse son exécution pour d'autres raisons que sa fin.

Une chose très importante à avoir présente, spécialement quand vous utilisez la commande F7, est que vous devez exécuter l'instruction où le point d'arrêt a été placé pour que le point d'arrêt ait un effet quelconque. Si votre programme saute sans exécuter l'instruction où ce point d'arrêt se trouve, le programme ne s'arrêtera plus jusqu'à sa fin normale. Par conséquent, quand vous choisissez un point d'arrêt, ce point devra toujours être une *instruction de séquence*, c'est-à-dire, un endroit du programme où tous les chemins convergent. Si vous ne pouvez pas placer un point d'arrêt sur une instruction séquentielle, il vous faudra placer plusieurs breakpoints, de façon à avoir des points d'arrêts alternatifs.

La meilleure manière de placer un point d'arrêt formel (c'est-à-dire permanent) est de placer le curseur sur l'instruction souhaitée et presser la touche F9. Ceci aura pour effet de changer la couleur de l'instruction en question pour indiquer qu'un point d'arrêt a été placé à cet endroit. Notez que la touche F9 ne fonctionne que sur des instructions machine 80x86. Vous ne pouvez pas l'utiliser sur des lignes vides, sur des commentaires, sur des directives d'assembleur ou sur des pseudo-opcodes.

La fenêtre de commandes de CodeView fournit également plusieurs commandes pour manipuler des points d'arrêt, telles que BC (Breakpoint Clear), BD (Breakpoint Disable), BE (Breakpoint Enable), BL (Breakpoint List) et BP (Breakpoint Set). Ces commandes sont très poussantes et vous permettent d'activer des points d'arrêt lors de la modification de la mémoire ou lors de l'évaluation d'une expression ; on peut aussi appliquer des compteurs sur les points d'arrêt et plus encore. Voyez la partie " Environment and Tools" du manuel de MASM ou bien l'aide en ligne de CodeView pour plus d'informations sur ces commandes.

Un autre outil très utile dans CodeView est la fenêtre inspecteur (*watch window*). Celle-ci affiche les valeurs de certainees expressions spécifiques pendant l'exécution du programme. Une des applications les plus importantes d'une fenêtre inspecteur est d'afficher le contenu de variables sélectionnées pendant l'exécution du programme. A l'occurrence d'un point d'arrêt, CodeView met automatiquement à jour toutes les expressions d'inspection. Vous pouvez ajouter une expression d'inspection dans la fenêtre d'inspection en utilisant la commande *Add Watch* du menu Data. La boîte de dialogue qui en résulte est semblable à :



En tapant un nom de variable, comme "Counter" ci-dessus vous pouvez ajouter un élément d'inspection à la fenêtre. En ouvrant la fenêtre inspecteur (du menu Window), vous pouvez voir les valeurs de toute expression d'inspection créée.

9.10.2 Stratégies de débogage

Apprendre comment utilisser efficacement un débogueur pour localiser des problèmes dans vos programmes en langage machine n'est pas quelque chose que vous pouvez apprendre d'un livre. Et il faut un petit peu d'effort pour utiliser un débogueur comme CodeView et apprendre les techniques nécessaires pour détecter rapidement la source d'une erreur dans un programme. Pour cette raison, trop d'étudiants de niveau avancé sont souvent forcés de reprendre de ce qu'ils avaient appris au début de leurs études. Il faut éviter de commettre cette erreur. Le temps employé pour apprendre à utiliser CodeView comme il faut vous sera rapidement reconnaissant.

9.10.2.1 Détecter des boucles infinies

Les boucles infinies sont des problèmes très communs en beaucoup de programmes. Vous exécutez un programme et toute la machine se bloque devant vous. Que faire ? La première chose à faire est de charger votre programme dans CodeView. Une fois qu'il aura commencé à s'exécuter et il y a une boucle infinie, vous pouvez arrêter manuellement l'exécution du programme en pressant la touche SysRq ou Ctrl-Break. Ceci généralement oblige le contrôle à retourner à CodeView. Si vous êtes actuellement en train d'exécuter une petite boucle, vous pouvez utiliser la commande *trace* pour vous rendre compte enfin qu'est-ce qui n'est pas correct.

Un autre moyen de détecter une boucle infinie est d'utiliser une *recherche binaire*. Pour utiliser cette technique, placez un point d'arrêt au milieu de votre programme (ou au milieu du code que vous voulez tester). Commencez l'exécution du programme. S'il plante, alors la boucle infinie se trouve *avant* le point d'arrêt. Si votre programme parviendra au point d'arrêt, alors la boucle infinie se trouve *après* celui-ci[4]. Une fois que vous déterminez quelle moitié de votre programme contient le problème, alors vous placerez un autre point d'arrêt au milieu de cette dernière partie. Si vous ne trouvez aucun problème alors, vous avez probablement commis une erreur, faites la même chose et placez un point d'arrêt au milieu de la seconde moitié du programme. Exécutez le programme dès le début encore (vous pouvez utiliser la commande de fenêtre de CodeView "L" pour redémarrer le programme dès son début). Localiser ainsi une boucle infinie dans une plage déterminée du programme, par exemple son 25% son 50% etc. permet de la cerner à une petite protion de code. Et vous pouvez continuer à répéter cette étape jusqu'à trouver très exactement l'endroit où se trouve le problème.

Bien évidemment, il ne faut jamais placer un point d'arrêt à l'intérieur d'une boucle si vous êtes en train de chercher une boucle infinie. Sinon, CodeView s'arrêtera à chaque itération de cette boucle et il vous prendra beaucoup plus de temps pour localiser l'erreur. Certes, si la boucle infinie se trouve à l'intérieur d'une autre boucle, alors vous allez avoir besoin à la limite de placer des points d'arrêt à l'intérieur d'une boucle, mais il y a de forte chances de trouver la boucle infinie déjà à la première exécution de la boucle externe. Si vous aurez besoin de placer un point d'arrêt dans une boucle qui doit s'exécuter plusieurs fois avant l'arrêt désiré, vous pouvez attacher un *compteur* au point d'arrêt qui décrémente jusqu'à une certaine valeur avant de produire réellement un effet. Lisez le manuel « Environment and Tools » de MASM ou consultez l'aide de CodeView, pour avoir plus de détails sur ces compteurs.

9.10.2.2 Calculs incorrects

Un autre problème commun est obtenir des résultats incorrects après avoir effectué une séquence d'opérations arithmétiques et logiques. Vous pouvez observer le code source tout le jour et ne rien voir, mais si vous tracez votre code vous avez de bonnes possibilités de détecter l'erreur.

Si vous pensez qu'un calcul particulier n'est pas en train de produire un résultat correct, placez un point d'arrêt dans la première instruction du calcul et exécutez le programme en plaine vitesse jusuq'à ce point. *Assurez-vous de vérifier toutes les variables et les registres impliquées dans la séquence.* Trop souvent, un mauvais calcul est le résultat de mauvaises valeurs d'entrée, ce qui veut dire que la cause du calcul incorrect n'est pas nécessairement à l'intérieur de ce calcul.

[4]Il va de soi qu'il faut s'assurer que l'instruction où vous placez le point d'arrêt soit un point de séquence. Si votre code peut sauter au-dessus de votre point d'arrêt, vous n'avez rien prouvé.

Une fois avoir vérifié que les valeurs d'entrée sont correctes, vous pouvez faire la trace de toutes les instructions d'un calcul à raison d'une à la fois. Après l'exécution de chaque instruction vous pouvez comparer les résultats que vous obtenez avec ceux que vous vous attendiez.

La chose la plus importante à garder à l'esprit dans votre recherche des bogues est que la source de l'erreur peut se trouver quelquepart ailleurs par rapport au point où vous avez noté l'erreur pour la première fois. C'est pourquoi vous devriez toujours vérifier les registres et les variables d'entrée avant de faire la trace dans une section de code. Si vous trouvez que les valeurs d'entrée *ne sont pas* correctes, alors l'erreur ne dépend pas de votre calcul et vous devez chercher ailleurs.

9.10.2.3 Instructions incorrectes / Boucles infinies Partie II

Parfois, quand votre programme plante, ce n'est pas dû à des boucles infinies, mais vous avez plutôt exécuté un opcode qui ne correspond pas à une instruction machine valide. D'autres fois encore, vous pressez la touche SysReq pour ne remarquer que vous êtes en train d'exécuter un code qui n'est nullepart aux voisinages de votre programme, qui est peut-être au milieu de la mémoire RAM et que vous êtes réellement en train d'exécuter des instructions sans signification. La plupart des fois, ceci est dû à des problèmes de pile ou à quelques sauts indirects. La meilleure stratégie dans ces cas est d'ouvrir une fenêtre de mémoire et d'effectuer un *dump* sur quelques emplacements autour du pointeur de pile (SS:SP). Essayez et localisez une adresse de retour raisonnable sur la tête de la pile (où proche, s'il y a beaucoup de valeurs poussées dans la pile) et désassemblez le code. Quelquepart avant l'adresse de retour il y a probablement un appel. Vous devriez placer un point d'arrêt sur cet emplacement et commencer une trace, instruction par instruction, tout au long de la routine, en observant tout ce qui arrive à l'occurrence de tous les sauts indirects et les retours. Prêtez beaucoup d'attention à la pile pendant tout ceci.

9.10.3 Exercice de débogage I : Utiliser CodeView pour trouver des bogues dans un calcul

Exercice 1 : Exécutez CodeView. Le programme suivant contient diverses bogues (notées dans les commentaires). Entrez le programme dans le système (notez que ce code est disponible sous le nom de Ex9_1.asm dans votre répertoire de travail du chapitre 9).

```
dseg            segment para public 'data'
I               word    0
J               word    0
K               word    0
dseg            ends

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

; Ce programme n'est utile qu'à des fins de débogage !
; Le but est de l'exécuter à l'intérieur de CodeView.
;
; Ce programme est bourré de bogues. Dans cette courte séquence
; de code, ces bogues sont assez évidentes, mais ce n'est pas le cas
; dans de plus grands programmes.

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

; La boucle suivante incrémente I jusqu'à 10

ForILoop:       inc     I
                cmp     I, 10
                jb      ForILoop

; Cette autre boucle devrait faire la même chose que celle ci-dessus
; mais on y oublie de réinitialiser I à zéro avant l'étiquette ForILoop2.
```

```
; Qu'est-ce qu'il arrive ?

ForILoop2:      inc     I
                cmp     I, 10
                jb      ForILoop2

; La prochaine boucle tente encore d'effectuer la même opération, mais cette fois,
; on n'oublie pas de réinitialiser I. Pourtant, il y a un autre bogue dans ce
; programme, un branchement dirigé vers la mauvaise étiquette.

                mov     I, 0
ForILoop3:      inc     I
                cmp     I, 10
                jb      ForILoop        ;<<<-- Whoops ! Étiquette incorrecte !

; La boucle suivante additionne I à J tant que J n'atteint 100.
; Malheureusement, l'auteur de ce code doit s'être confondu en pensant que AX
; contient la somme accumulée dans J. Il compare AX avec 100 alors qu'il devrait
; comparer J avec cette valeur.

WhileJLoop:     mov     ax, I
                add     J, ax
                cmp     ax, 100         ; Ceci est un bogue !
                jb      WhileJLoop

                mov     ah, 4ch         ; Retourner au DOS.
                int     21h
Main            endp
cseg            ends

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

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

Assemblez ce programme avec la commande :

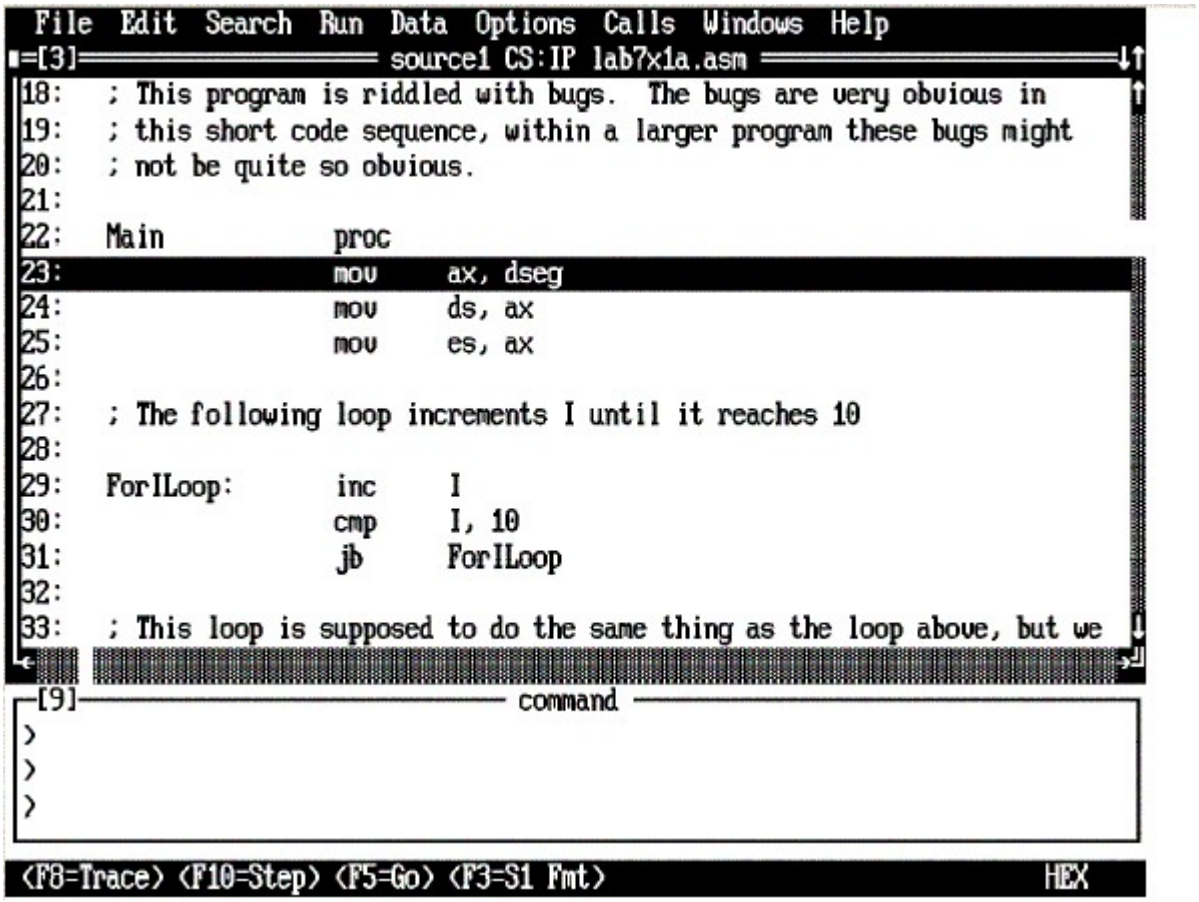
```
ML /Zi  Ex9_1.asm
```

L'option "/Zi" indique à MASM d'inclure les informations de débogage pour CodeView dans le fichier .EXE. Notez que "Z" doit être en majuscules et "i" en minuscules.

Chargez l'exécutable dans CodeView avec la commande :

```
CV Ex9_1
```

Votre écran devrait ressembler maintenant à ce qui suit :



Notez que CodeView souligne l'instruction qui *va être* exécutée et non celle qui *vient d'être* exécutée (dans le code ci-dessus, il s'agit de mov ax, dseg). Essayez la commande trace en pressant la touche F10 trois fois. Ceci devrait laisser l'instruction inc I en évidence. Tracez tout au long de la boucle et notez tous les changements qui se produisent après chaque itération (note : souvenez-vous que jb=jc, donc assurez-vous de noter aussi la valeur du drapeau de retenue après chaque itération).

Pour votre rapport de laboratoire : parlez de vos résultsts dans votre cahier. Notez également la valeur finale de I après la fin de la boucle.

Seconde partie : Localisez les bogues. La seconde boucle du programme est celle qui contient le bogue majeur. Le programmeur oublie de réinitialiser I à 0 avant d'exécuter le code commençant à partir de l'étiquette ForILoop2. Tracez tout au long de la boucle jusqu'à tomber dans la troisième boucle commençant par l'étiquette ForILoop3.

Pour votre rapport de laboratoire : Décrivez ce qui est mauvais et pourquoi le fait de presser la touche F8 vous aidérait à résoudre le problème.

Troisième Partie : Localiser une autre bogue. La troisième boucle du programme contient une erreur de frappe ou de distraction causant un branchement incorrect à l'étiquette ForILoop. Tracez le long de ce code en pressant la touche F8.

Pour votre rapport de laboratoire : Décrivez le processus menant à résoudre ce problème et expliquez comment la trace pourrait vous être d'aide pour détecter l'erreur.

Quatrième Partie : Vérifiez l'exactitude. Le programme Ex9_2.asm est une version corrigée du programme qu'on vient d'analyser. Effectuez une trace à l'intérieur de ce code et constatez que tout y est correct.

Pour votre rapport de laboratoire : Décrivez les différences entre les deux sessions de débogage.

Cinquième Partie : En utilisant Ex9_2.exe[5], ouvrez une fenêtre d'inspection et ajouter l'expression "I" à cette fenêtre. Placez un point d'arrêt formel sur les trois instructions jb. Exécutez le programme en utilisant la commande Go et commentez sur ce qui se passe dans la fenêtre d'inspection pour chaque point d'arrêt.

Pour votre projet de laboratoire : Décrivez comment votre code utilise la fenêtre d'inspection pour vous aider à localiser un problème dans vos programmes.

9.10.4 Exercices sur les logiciels de retardement de boucles

Le programme Ex9_3.asm est un court logiciel permettant de retarder des boucles. Exécutez ce programme et déterminez la valeur de la variable de contrôle de la boucle qui provoquera un retardement de 11 secondes. Remarque : cette valeur a été choisie pour un système 80486 de 66 MHz. Si vous avez un ordinateur plus lent, vous aurez éventuellement besoin de la réduire. Si vous avez une machine plus rapide, augmentez-la. Ajustez cette valeur de sorte à obtenir un délai d'environ 11 secondes sur l'ordinateur que vous utilisez.

Pour votre rapport de laboratoire : Fournissez la constante relative à votre système produisant un délai de 11 secondes. Argumentez aussi sur la possibilité de créer des délais de 1, 10, 20, 30 et 60 secondes à l'aide de ce code.

Pour aller plus loin : Après avoir déterminé la bonne constante pour le délai souhaité, essayez l'exécutable sur différents systèmes avec différentes vitesses d'horloge. Prenez note des différences que vous remarquerez.

Seconde Partie : Délai déterminé par le matériel. Le fichier Ex9_4.asm contient un code de retardement de boucles pouvant déterminer automatiquement le nombre d'itérations par l'observation de l'horloge en temps réel du BIOS. Exécutez ce programme et observez les résultats.

Pour votre rapport de laboratoire : Déterminez le comptage d'itérations et incluez cette valeur dans votre rapport. Si votre PC possède un interrupteur turbo, mettez-le en mode "non-turbo", quand ceci est réquéré par le programme. Mesurez le délai véritable de façon précise, soit en mode turbo qu'en mode non-turbo. Incluez ces temps dans votre rapport.

Pour aller plus loin : Essayez l'exécutable sur différents systèmes avec divers CPU et diverses vitesses d'horloge. Exécutez le programme et mesurez les délais. Décrivez les différences dans votre rapport.

9.11 Projets de programmation

Pas de projets pour ce chapitre.

9.12 Résumé

Ce chapitre a discuté sur les opérations arithmétiques et logiques des processeurs 80x86. Il a présenté les instructions et les techniques nécessaires pour effectuer des opérations sur les entiers de façon semblable à celle des langages de haut niveau. On a également traité les opérations en multiprécision, comment effectuer des opérations arithmétiques à l'aide d'instructions non arithmétiques et comment utiliser des instructions arithmétiques pour effectuer des opérations non arithmétiques.

Les expressions arithmétiques sont beaucoup plus simples dans les langages de haut niveau. En effet, le but original du langage FORTRAN était celui de fournir une version simplifiée (*FORmula TRANslation*) des expressions arithmétiques. Bien qu'en assembleur ce soit plus compliqué que - disons - en Pascal, si vous suivez certaines règles simples, ce n'est pas aussi pénible. Pour une description étape par étape, consultez :

- "Expressions arithmétiques" (9.1)
- "Affectations simples" (9.1.1)
- "Expressions simples", (9.1.2)
- "Expressions complexes" (9.1.3)
- "Opérateurs commutatifs" (9.1.4)

[5]Le texte original plaçait l'exténsion .asm au programme. Cependant, cela semble incorrect, puisque ce fichier va être utilisé par CodeView, c'est plutôt la version .exe qu'il faudrait utiliser, n.d.t.

• "Expressions logiques (booléennes) " (9.2)

L'un des grands avantages de l'assembleur est sa facilité d'effectuer un nombre illimité d'opérations arithmétiques en multiprécision et d'opérations logiques. Ce chapitre a décrit comment effectuer des opérations en précision étendue pour les opérations les plus communes. Consultez :

• "Opérations en multiprécision" (9.3)
• "Opérations d'addition en multiprécision" (9.3.1)
• "Opérations de soustraction en multiprécision" (9.3.2)
• "Comparaisons de valeurs en précision étendue" (9.3.3)
• "Multiplication en précision étendue" (9.3.4)
• "Division en précision étendue" (9.3.5)
• "Opérations NEG en précision étendue" (9.3.6)
• "Opérations AND en précision étendue" (9.3.7)
• "Opérations OR en précision étendue" (9.3.8)
• "Opérations XOR en précision étendue" (9.3.9)
• "Opérations NOT en précision étendue" (9.3.10)
• "Opérations de décalage en précision étendue" (9.3.11)
• "Opérations de rotation en précision étendue" (9.3.4)

Parfois, on peut avoir besoin d'effectuer une opération entre deux opérandes de différents types. Par exemple, vous pouvez avoir besoin d'additionner un octet avec un mot. L'idée générale est d'élargir la plus petite opérande et ensuite effectuer l'opération. Pour tous les détails, consultez :

• "Opérations sur des opérandes de différentes tailles" (9.4)

Bien que le jeu d'instructions 80x86 fournisse différentes voies simples pour effectuer diverses tâches, vous pouvez souvent prendre avantage de différents *idiomes* du jeu à l'aide de certaines opérations arithmétiques pour produire du code plus rapide ou plus court par rapport à la manière "officielle". Ce chapitre a présenté certains de ces idiomes. Pour voir des exemples, consultez :

• "Idiomes matériels et arithmétiques" (9.5)
• "Multiplier sans MUL et IMUL" (9.5.1)
• "Division sans DIV et IDIV" (9.5.2)
• "Utiliser AND pour calculer les restes" (9.5.3)
• "Implémenter des compteurs modulo-n avec AND" (9.5.4)
• "Tester une valeur en précision étendue pour 0FFFF..FFh" (9.5.5)
• "Opérations TEST" (9.5.6)
• "Tester les signes avec XOR" (9.5.7)

Pour manipuler des données compactées, il vous faut l'habileté d'extraire et d'insérer un champ depuis un registrement compacté. Vous pouvez utiliser les instructions logiques `and` et `or` pour masquer les champs que vous voulez manipuler ; vous pouvez utiliser les instructions `shl` et `shr` pour ordonner les données à leurs positions appropriées avant d'insérer ou de retrancher les données. Pour en savoir plus, consultez :

• "Opérations de masquage" (9.6)
• "Opérations de masquage avec l'instruction AND" (9.6.1)
• "Opérations de masquage avec l'instruction OR" (9.6.2)
• "Compacter et décompacter des données" (9.7)

9.13 Questions

1. Décrivez la voie à suivre pour additionner une variable d'un mot non signé avec une d'un octet non signé, afin de produire un résultat d'octet. Expliquez toutes les possibilités d'erreur et qu'est-ce qu'il faudrait faire pour les vérifier.

2. Répondez comme à la question 1, mais pour des valeurs signées.

3. Supposez que `var1` est un mot et que `var2` et `var3` sont des doubles-mots. Quel serait le code 80x86 pour additionner `var1` avec `var2` en laissant le résultat dans `var3`, si

a) var1, var2 et var3 sont non signées ?
b) var1, var2 et var3 sont signées ?

4. "ADD BX, 4" est plus efficace que "LEA BX, 4[BX]". Donnez un exemple où l'instruction LEA serait plus efficace que l'instruction ADD.

5. Donnez une seule instruction LEA sur un 80386, pouvant multiplier EAX par cinq.

6. Sanchant que VAR1 et VAR2 sont des variables de 32 bits déclarées avec la directive DWORD, écrivez les séquences de code pouvant tester ce qui suit :

a) VAR1 = VAR2
b) VAR1 <> VAR2
c) VAR1 < VAR2 (valeurs signées et non signées
d) VAR1 <= VARW pour les deux)
e) VAR1 > VAR2
f) VAR1 >= VAR2

7. Convertissez en assembleur les expressions suivantes, en emploiant des décalages, des additions et des soustractions à la place que des multiplications :

a) AX * 5
b) AX * 129
c) AX * 1024
d) AX * 20000

8. Quelle est la meilleure manière de diviser le registre AX par les constantes suivantes ?

a) 8 b) 255 c) 1024 d) 45

9. Décrivez comment pourriez-vous multiplier une valeur de 8 bits dans AL par 256 (en laissant le résultat dans AX). N'utilisez rien de plus que deux instructions mov.

10. Comment pourriez-vous effectuer un AND entre la valeur dans AX et 0ffh en n'utilisant autre chose qu'une instruction mov ?

11. Supposez que le registre AX contient deux valeurs binaires compactées. Supposez que les premiers quatre bits moins significatifs contiennent une valeur dans la plage 0..15 et que les 12 bits plus significatifs restants contiennent une valeur dans la plage 0..4095. Maintenant supposez vouloir vérifier si la portion de 12 bits contient ou pas la valeur 295. Expliquez comment vous pouvez réaliser ceci en n'emploiant que deux instructions.

12. Comment pourriez-vous utiliser l'instruction TEST (ou une séquence d'instructions TEST) pour vérifier si le bit 0 et 4 du registre AL sont les deux mis à 1 ? Comment l'instruction TEST pourrait-elle être utilisée si les deux bits sont activés ? Et comment pourrait-elle être utilisée dans le cas contraire (les deux valent 0) ?

13. Pourquoi le registre CL ne peut être utilisé comme une opérande compteur pendant un décalage d'opérandes en multiprécision ? Par exemple, pourquoi les instructions suivantes ne décalent pas la valeur dans (DX,AX) de trois bits vers la gauche ?

```
mov     cl, 3
shl     ax, cl
rcl     dx, cl
```

14. Fournissez une séquence d'instructions pouvant effectuer des opérations ROL et ROR en précision étendue (32 bits) en n'utilisant que des instructions 8086.

15. Donnez une séquence d'instructions implémentant une opération ROR de 64 bits à l'aide des instructions SHRD et BT du 80386.

16 Donnez le code 80386 pour effectuer les comparaisons de 64 bits suivantes, sachant que vous êtes en train de calculer X := Y *opération* Z avec X, Y et Z définis comme suit :

```
X               dword   0, 0
Y               dword   1, 2
Z               dword   3, 4
```

Effectuez les opérations suivantes entre Y et Z en chargeant le résultat dans X :

a) addition	b) soustraction	c) multiplication
d) AND	e) OR	f) XOR
g) Négation	h) NOT logique	