Structures de contrôle Chapitre 10

Un programme informatique contient typiquement trois structures : les séquences d'instruction, les décisions et les boucles. Une séquence est un ensemble d'instructions à exécuter une après l'autre. Une décision est un branchement (**goto**) dans un programme basé sur une condition. Une boucle est une séquence d'instructions qui seront exécutées de façon répétitive tant qu'une condition tient. Dans ce chapitre, nous allons explorer quelques structures de décision courantes en assembleur 80x86.

10.0 Vue d'ensemble du chapitre

Ce chapitre traitera les deux types primaires de structures de contrôle : décision et itération. Il décrit comment convertir des instructions de langage de haut niveau telles que **if..then..else**, **case** (**switch**), **while** , **for** etc., en des séquences équivalentes en assembleur. Ce chapitre couvrira également des techniques que vous pouvez employer pour améliorer l'exécution de ces structures de contrôle. Les sections au-dessous de celle qui ont le préfixe "●" sont essentielles. Les sections avec un "•" sont à propos de sujets avancés que vous pouvez vouloir remettre à plus tard.

- Introduction aux décisions.
- Séquences IF..THEN..ELSE.
- Instructions CASE.
- ❋ Machines d'état et sauts indirects.
- Code spaghetti.
- Boucles.
- Boucles WHILE .
- Boucles REPEAT..UNTIL.
- LOOP..ENDLOOP.
- Boucles FOR.
- Utilisation des registres et boucles.
- ❋ Améliorations des performances.
- ❋ Déplacer la condition de terminaison à la fin d'une boucle.
- ❋ Exécuter la boucle à l'envers.
- ❋ Invariants de boucle.
- ❋ Démêlage des boucles.
- ❋ Variables d'induction

10.1 Introduction aux décisions

Dans sa forme la plus simple, une décision est une sorte de branchement dans le code qui commute entre deux chemins d'exécution possibles en se basant sur une condition. Normalement (cependant, pas toujours), des séquences d'instruction conditionnelles sont implémentées avec les instructions de branchement conditionnel. Les instructions conditionnelles correspondent à l'instruction **if..then..else** en Pascal :

```
IF (condition est vraie) THEN stmt1 ELSE stmt2;
```

L'assembleur, comme d'habitude, offre beaucoup plus de flexibilité pour traiter les instructions conditionnelles. Considérez l'instruction Pascal suivante :

```
IF ((X<Y) and (Z>T)) or (A <> B) THEN stmt1;
```

Une approche "force brute" pour convertir cette instruction en assembleur pourrait produire :

```
                mov     cl, 1           ;Suppose condition vraie
                mov     ax, X
                cmp     ax, Y
                jl      IsTrue
                mov     cl, 0           ;Celle-ci est fausse
IsTrue:         mov     ax, Z
                cmp     ax, T
                jg      AndTrue
                mov     cl, 0           ;C'est faux maintenant
```

```
AndTrue:            mov    al, A
                    cmp    al, B
                    je     OrFalse
                    mov    cl, 1        ;C'est vrai si A <> B
OrFalse:            cmp    cl, 1
                    jne    SkipStmt1
       <Le code pour stmt1 vient ici>
```

`SkipStmt1:` Comme vous pouvez le voir, il faut un nombre considérable d'instructions conditionnelles uniquement pour traiter l'expression de l'exemple ci-dessus. Ceci correspond vaguement aux instructions Pascal (équivalentes) :

```
                    cl := true;
                    IF (X >= Y) then cl := false;
                    IF (Z <= T) then cl := false;
                    IF (A <> B) THEN cl := true;
                    IF (CL = true) then stmt1;
```

Comparez maintenant ceci avec le code "amélioré" suivant :

```
                     mov    ax, A
                    cmp    ax, B
                    jne    DoStmt
                    mov    ax, X
                    cmp    ax, Y
                    jnl    SkipStmt
                    mov    ax, Z
                    cmp    ax, T
                    jng    SkipStmt
   DoStmt:
          <Placez le code pour Stmt1 ici>
   SkipStmt:
```

Deux choses découlent des séquences de code ci-dessus : d'abord, une instruction conditionnelle simple en Pascal peut exiger plusieurs sauts conditionnels en assembleur ; en second lieu, l'organisation des expressions complexes dans une séquence conditionnelle peut affecter l'efficacité du code. Par conséquent, traiter des séquences conditionnelles en assembleur demande un soin particulier.

Des instructions conditionnelles peuvent être décomposés en trois catégories de base : les instructions **if..then..else,** les instructions **case** et les sauts indirects. Les sections suivantes décriront ces structures de programme, comment les employer, et comment les écrire en assembleur.

10.2 Séquences IF..THEN..ELSE

L'instruction conditionnelle la plus couramment utilisée est l'instruction **if..then** ou **if..then..else**. Ces deux instructions prennent la forme suivante représentée sur la figure 10.1.

L'instruction **if..then** est seulement un cas spécial de l'instruction **if..then..else** (avec un bloc ELSE vide). Par conséquent, nous considérerons seulement la forme plus générale **if..then..else**. L'implémentation de base d'une instruction **if..then..else** en assembleur 80x86 ressemble quelque peu à cela :

```
              {Séquence d'instructions pour tester une condition}
                    Jcc    ElseCode
              { Séquence d'instructions correspondant au bloc THEN}
                    jmp    EndOfIF
ElseCode:
              { Séquence d'instructions correspondant au bloc ELSE}
EndOfIF:
```

Note : J cc représente une instruction de branchement conditionnel quelconque.

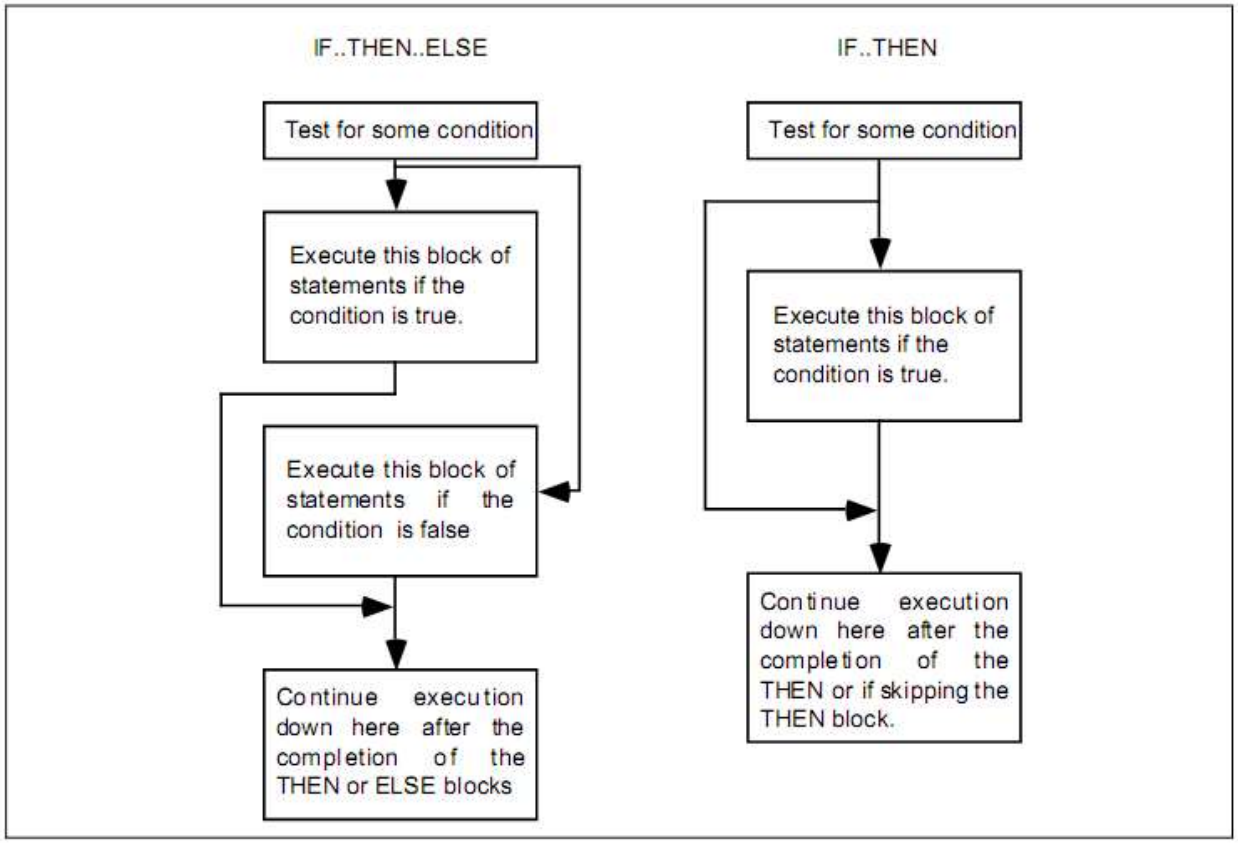

Figure 10.1 Flux d'Instructions IF..THEN et IF..THEN..ELSE

Par exemple, pour convertir l'instruction Pascal :

```
IF (a=b) THEN c := d ELSE b:= b + 1;
```

vous pouvez employer le code 80x86 suivant :

```
                mov     ax, a
                cmp     ax, b
                jne     ElseBlk
                mov     ax, d
                mov     c, ax
                jmp     EndOfIf
ElseBlk:
                inc     b
EndOfIf:
```

Pour des expressions simples comme (**A=B**) produire du code approprié pour une instruction **if..then..else** est presque enfantin. Si l'expression devient plus complexe, la complexité associée au code assembleur augmente aussi. Considérez l'instruction **if** suivante présentée plus tôt :

```
IF ((X > Y) and (Z < T)) or (A <> B) THEN C := D;
```

Lorsque vous traitez des instructions **if** complexes telles que celle-ci, vous trouverez la tâche de conversion plus facile si vous brisez cette instruction **if** en une séquence de trois instructions **if** différentes comme suit :

```
IF (A<>B) THEN C := D
IF (X > Y) THEN IF (Z < T) THEN C := D;
```

Cette conversion vient de l'équivalent Pascal suivantes :

```
IF (expr1 AND expr2) THEN stmt;
```

qui équivaut à :

```
IF (expr1) THEN IF (expr2) THEN stmt;
```

et

```
IF (expr1 OR expr2) THEN stmt;
```

qui équivaut à

```
IF (expr1) THEN stmt;
IF (expr2) THEN stmt;
```

En assembleur, l'instruction **if** précédente devient :

```
                    mov    ax, A
                    cmp    ax, B
                    jne    DoIF
                    mov    ax, X
                    cmp    ax, Y
                    jng    EndOfIf
                    mov    ax, Z
                    cmp    ax, T
                    jnl    EndOfIf
DoIf:
                    mov    ax, D
                    mov    C, ax
EndOfIF:
```

Comme vous imaginez, le code nécessaire pour tester ue conditrion peut devenir beaucoup plus complexe que les instructions apparaissant dans les blocs **else** et **then**. Bien qu'il semble quelque peu paradoxal qu'il faut plus d'effort pour tester une condition que pour agir selon ses résultats, cela arrive tout le temps. Donc, vous devez être préparé à cette situation.

Le plus gros problème, probablement, avec l'implémentation d'instructions conditionnelles complexes en assembleur est essayer de comprendre ce que vous avez fait après avoir écrit le code. Le plus grand avantage, sans doute, des langages de haut niveau sur l'assembleur est que les expressions sont plus faciles à lire et à comprendre. La version HLL est auto-documentée, tandis que l'assembleur tend à cacher la vraie nature du code. Par conséquent, des nombreux commentaires bien écrits sont un ingrédient essentiel de l'implémentation des instructions **if..then..else** en assembleur. Une implémentation élgante de l'exemple ci-dessus est :

```
; IF ((X > Y) AND (Z < T)) OR (A <> B) THEN C := D;
; Implémenté comme :
; IF (A <> B) THEN GOTO DoIF;

                    mov    ax, A
                    cmp    ax, B
                    jne    DoIF

; IF NOT (X > Y) THEN GOTO EndOfIF;

                    mov    ax, X
                    cmp    ax, Y
                    jng    EndOfIf

; IF NOT (Z < T) THEN GOTO EndOfIF ;
                    mov    ax, Z
                    cmp    ax, T
                    jnl    EndOfIf

; Bloc THEN:

DoIf:       mov    ax, D
                    mov    C, ax

; Fin de l'instruction IF

EndOfIF:
```

Certes, cela semble aller trop loin pour un exemple aussi simple. Ce qui suit pourrait probablement suffire :

```
; IF ((X > Y) AND (Z < T)) OR (A <> B) THEN C := D;
```

```
; Teste l'expression booléenne :

                    mov    ax, A
                    cmp    ax, B
                    jne    DoIF
                    mov    ax, X
                    cmp    ax, Y
                    jng    EndOfIf
                    mov    ax, Z
                    cmp    ax, T
                    jnl    EndOfIf

; Bloc THEN:

DoIf:       mov    ax, D
                    mov    C, ax

; Fin de l'instruction IF

EndOfIF:
```

Cependant, au fur et à mesuret que vos instructions **if** deviennent complexes, la densité (et la qualité) de vos commentaires devient de plus en plus plus importante.

10.3 Instructions CASE

L'instruction Pascal CASE prend la forme suivante :

```
CASE variable OF
       const 1 :stmt 1 ;
       const 2 :stmt 2 ;
       .
       .
       .
       const n :stmt n
END;
```

Quand cette instruction s'exécute, elle vérifie la valeur de la variable avec les constantes $const_1$ ... $const_n$ Si une correspondance est trouvée alors l'instruction qui suit s'exécute. Le Pascal Standard place quelques restrictions à l'instruction **case**. D'abord, si la valeur de la variable n'est pas dans la liste des constantes, le résultat de l'instruction **case** n'est pas défini. En second lieu, toutes les constantes apparaissant comme étiquettes de **case** doivent être uniques. La raison de ces restrictions deviendra claire d'ici peu.

La plupart des textes d'introduction à la programmation introduisent l'instruction **case** en la présentant comme une séquence d'instructions **if..then..else**. Ils déclarent parfois que les deux extraits de code Pascal suivants sont équivalents :

```
CASE I OF
       0: WriteLn('I=0');
       1: WriteLn('I=1');
       2: WriteLn('I=2');
END;
```

```
IF I = 0 THEN WriteLn('I=0')
ELSE IF I = 1 THEN WriteLn('I=1')
ELSE IF I = 2 THEN WriteLn('I=2');
```

Alors que sémantiquement ces deux segments de code peuvent être identiques, leur exécution est habituellement[1] différente. Là où l'enchaînement **if..then..else** entraîne une comparaison pour chaque instruction conditionnelle dans la séquence, l'instruction **case** utilise normalement un saut indirect pour transférer le contrôle à

[1]Des versions du Turbo Pascal, malheureusement, traitent l'instruction **case** comme une forme de l'instruction **if..then..else**.

l'une des diverses instructions après une comparaison unique. Les deux exemples qu'on a présentés ci-dessus pourrait être écrit en assembleur avec le code suivant :

```
                    mov    bx, I
                    shl    bx, 1 ;Multiply BX by two
                    jmp    cs:JmpTbl[bx]
JmpTbl              word   stmt0, stmt1, stmt2
Stmt0:              print
                    byte   "I=0",cr,lf,0
                    jmp    EndCase

Stmt1:              print
                    byte   "I=1",cr,lf,0
                    jmp    EndCase

Stmt2:              print
                    byte "I=2",cr,lf,0
EndCase:

; IF..THEN..ELSE form:

                    mov    ax, I
                    cmp    ax, 0
                    jne    Not0
                    print
                    byte   "I=0",cr,lf,0
                    jmp    EndOfIF

Not0:         cmp   ax, 1
                    jne    Not1
                    print
                    byte   "I=1",cr,lf,0
                    jmp    EndOfIF

Not1:         cmp   ax, 2
                    jne    EndOfIF
                    Print
                    byte   "I=2",cr,lf,0
EndOfIF:
```

À noter deux choses : plus vous avez de cas (consécutifs), plus l'exécution avec une table de saut devient efficace (à la fois en termes d'espace et de vitesse). Sauf les cas les plus simples, l'instruction **case** est presque toujours plus rapide et habituellement de beaucoup.Tant que les étiquettes **case** sont des valeurs consécutives, la version de l'instruction **case** par rapport à **IF-ELSE** occupe également moins d'espace.

Que se produit-il si vous devez inclure des étiquettes **case** non-consécutives ou si vous n'avez pas de garantie que la variable **case** reste dans sa **plage** ? Beaucoup de versions de Pascal ont étendu la définition de l'instruction **case** pour inclure une clause **otherwise**. Cette instruction **case** prend la forme suivante :

```
CASE variable OF
       const:stmt;
       const:stmt;
       . .
       . .
       . .
       const:stmt;
       OTHERWISE stmt
END;
```

Si la valeur de la variable correspond à une des constantes constituant les étiquettes **case**, alors l'instruction associée s'exécute. Si la valeur de la variable ne correspond à aucune des constantes constituant les étiquettes **case**, alors l'instruction suivant la clause **otherwise** s'exécute. La clause **otherwise** est mise en application en deux phases. D'abord, vous devez choisir les valeurs minimum et maximum qui apparaissent dans une instruction **case**. Dans les instructions **case** suivantes, l'étiquette **case** la plus petite est cinq, la plus grande est 15 :

```
CASE I OF
       5:stmt1;
```

```
            8:stmt2;
            10:stmt3;
            12:stmt4;
            15:stmt5;
            OTHERWISE stmt6
      END;
```

Avant d'exécuter le saut à travers la table de saut, l'implémentation 80x86 de cette instruction devrait vérifier que la variable **case** est dans la plage 5..15. Si ce n'est pas le cas, le contrôle devrait être immédiatement transféré à stmt6 :

```
                    mov    bx, I
                    cmp    bx, 5
                    jl     Otherwise
                    cmp    bx, 15
                    jg     Otherwise
                    shl    bx, 1
                    jmp    cs:JmpTbl-10[bx]
```

Le seul problème avec cette forme de l'instruction **case** est qu'elle ne gère pas correctement la situation où I est égale à 6, 7, 9, 11, 13 ou 14. Plutôt que de coller du code supplémentaire devant le saut conditionnel, vous pouvez coller des entrées supplémentaires dans la table de saut comme suit :

```
                    mov    bx, I
                    cmp    bx, 5
                    jl     Otherwise
                    cmp    bx, 15
                    jg     Otherwise
                    shl    bx, 1
                    jmp    cs:JmpTbl-10[bx]

Otherwise:    ;{mettez stmt6 ici}
                    jmp    CaseDone

JmpTbl              word   stmt1, Otherwise, Otherwise, stmt2, Otherwise
                    word   stmt3, Otherwise, stmt4, Otherwise, Otherwise
                    word   stmt5
                    ;etc.
```

Notez que la valeur 10 est soustraite de l'adresse de la table de saut. La première entrée dans la table est toujours à l'offset zéro tandis que la plus petite valeur utilisée comme index dans la table est cinq (et qui est multipliée par deux pour produire 10). Les entrées pour 6, 7, 9, 11, 13, et 14 pointent toutes sur le code pour la clause **otherwise**, aussi si I contient une de ces valeurs, la clause **otherwise** sera exécutée.

Il y a un problème avec cette implémentation de l'instruction **case**. Si les étiquettes **case** contiennent des entrées non-consécutives largement espacées, l'instruction suivante produira un fichier de code extrêmement grand :

```
      CASE I OF
            0: stmt1;
            100: stmt2;
            1000: stmt3;
            10000: stmt4;
            Otherwise stmt5
      END;
```

Dans cette situation, votre programme sera beaucoup plus petit si vous implémentez l'instruction **case** avec une séquence d'instructions **if** au lieu d'employer un instruction de saut. Cependant, gardez une chose à l'esprit - la taille de la table de saut normalement n'affecte pas la vitesse d'exécution du programme. Que la table de saut contienne deux entrées ou deux mille, l'instruction **case** exécutera la branchement multiple dans un laps de temps constant. L'implémentation de l'instruction **if** exige un laps de temps linéairement croissant pour chaque étiquette **case** apparaissant dans l'instruction **case**.

L'avantage sans doute le plus grand à employer l'assembleur au lieu d'un langage de haut niveau comme le Pascal est que vous ayez le choix de l'implémentation effective. Tantôt vous pourrez implémenter une instruction **case** comme séquence d'instructions **if..then..else**, tantôt vous pourrez l'implémenter avec une table de saut, tantôt vous porrez employer un hybride des deux :

```
CASE I OF
       0:stmt1;
       1:stmt2;
       2:stmt3;
       100:stmt4;
       Otherwise stmt5
END;
```

deviendrait :

```
                mov    bx, I
                cmp    bx, 100
                je     Is100
                cmp    bx, 2
                ja     Otherwise
                shl    bx, 1
                jmp    cs:JmpTbl[bx]
                ;etc.
```

Naturellement, vous pouvez faire la même chose en Pascal à l'aide du code suivant :

```
IF I = 100 then stmt4
ELSE CASE I OF
       0:stmt1;
       1:stmt2;
       2:stmt3;
       Otherwise stmt5
END;
```

Mais ceci tend à détruire la lisibilité du programme Pascal. D'autre part, le code supplémentaire pour tester 100 dans le code assembleur ne compromet pas la lisibilité du programme (peut-être parce qu'il est déjà très difficile à lire). Par conséquent, la plupart ajoutera le code supplémentaire pour rendre leur programme plus efficace.

L'instruction C/C++ **switch** est très semblable à l'instruction **case** du Pascal. Il n'y a qu'une différence sémantique majeure : le programmeur doit explicitement placer une instruction **break** dans chaque clause **case** pour transférer le contrôle à la première instruction au delà de **switch**. Ce **break** correspond à l'instruction **jmp** à la fin de chaque séquence **case** dans le code assembleur ci-dessus. Si le **break** correspondant n'est pas présent, C/C++ transfère le contrôle au le code du cas suivant. Cela équivaut à omettre le **jmp** à la fin de la séquence **case :**

```
switch (i)
{
       case 0:      stmt1;
       case 1:      stmt2;
       case 2:      stmt3;
              break;
       case 3:      stmt4;
              break;
       default:     stmt5;
}
```

Ceci se traduit en code 80x86 suivant :

```
                mov    bx, i
                cmp    bx, 3
                ja     DefaultCase

                shl    bx, 1
                jmp    cs:JmpTbl[bx]
JmpTbl          word   case0, case1, case2, case3

case0:          < code de stmt1>

case1:          < code de stmt2>

case2:          < code de stmt3>
                jmp    EndCase          ;Émis pour l'instruction break.

case3:          < code de stmt4>
```

```
                    jmp    EndCase             ;Émis pour l'instruction break.

DefaultCase: < code de stmt5>
EndCase:
```

10.4 Machines d'état et sauts indirects

Une autre structure de contrôle qu'on trouve souvent dans des programmes en assembleur est la *machine d'état*. Une machine d'état utilise une variable d'état pour contrôler le flux du programme. Le langage de programmation Fortran fournit cette fonctionnalité avec l'instruction goto assigné. Certaines variantes du C (par exemple, le GCC GNU de Free Software Foundation), fournissent de telles fonctionnalités. En assembleur, le saut indirect fournit un mécanisme pour implémenter facilement des machines d'état.

D'abord, qu'est-ce qu'une machine d'état ? En termes très simples, une machine d'état est un fragment de code[2] qui garde trace de l'historique de son exécution à l'entrée et à la sortie de certains « états ». Dans le cadre de ce chapitre, nous n'utiliserons pas une définition très formelle d'une machine d'état. Nous supposerons seulement que c'est du code capable de se souvenir (d'une manière ou d'une autre) de l'historique de son exécution (son *état*) et exécute des sections de code basées sur cet historique.

Dans un sens très aporximatif, tous les programmes sont des machines d'état. Les registres de la CPU et les valeurs dans la mémoire constituent « l'état » de cette machine. Cependant, nous les utiliserons dans un point de vue beaucoup plus de plus restreint. En effet, pour la plupart des utilisations, seulement une variable unique (ou la valeur dans le registre IP) dénotera l'état en cours.

Considèrons maintenant un exemple concret. Supposez que vous avez une procédure dont vous voulez qu'elle effectue une opération la première fois que vous l'appellez, une opération différente la deuxieme fois que vous l'appelez, encore autre chose la troisième fois que vous l'appelez et enfin quelque chose de nouveau encore au quatrième appel. Après celui-ci, elle répète ces quatre opérations différentes dans l'ordre. Par exemple, supposez que vous voulez que la procédure ajoute **ax** à **bx** la première fois, les soustraie au deuxième appel, les multiplie au troisième et les divise au quatrième. Vous pouvez implémenter cette procédure comme suit :

```
State        byte   0
StateMach           proc
                    cmp    state,0
                    jne    TryState1

; Si on est dans l'état 0, ajoute BX à AX et passe à l'état 1:

                    add    ax, bx
                    inc    State        ;Faire passer à état 1
                    ret

; Si on est dans l'état 1, soustrait BX de AX et passe à l'état 2:

TryState1:   cmp    State, 1
                    jne    TryState2
                    sub    ax, bx
                    inc    State
                    ret

; Si on est dans l'état 2, muliplie BX par AX et passe à l'état 3:

TryState2:   cmp    State, 2
                    jne    MustBeState3
                    push   dx
                    mul    bx
                    pop    dx
                    inc    State
                    ret
```

[2] Notez que les machines d'état n'ont pas besoin d'être basées sur le logiciel. L'implémentation de beaucoup d'elles est basée sur le matériel.

```
; Si rien, dans le code ci-dessus ne s'applique, supposons que nous sommes dans
; l'état 4. Alors divisons AX par BX.

MustBeState3:
                    push   dx
                    xor    dx, dx              ;Etend à zéro AX dans DX.
                    div    bx
                    pop    dx
                    mov    State, 0            ;Fait repasser à l'état 0
                    ret
StateMach           endp
```

Techniquement, cette procédure n'est pas la machine d'état. En fait, c'est la variable **State** et les instructions **cmp**/**jne** qui constituent la machine d'état.

Il n'y a rien particulièrement spécial dans ce code. C'est peu de chose de plus qu'une instruction **case** implémentée via une construction **if..then..else**. La seule chose de spécial dans cette procédure est qu'elle se rappelle combien de fois elle a été appelée[3] et se comporte différemment selon le nombre d'appels. Bien que ce soit une implémentation correcte de la machine d'état désirée, elle n'est pas particulièrement efficace. Les implémentations les plus courantes d'une machine d'état en assembleur employent un *saut indirect*. Plutôt que d'avoir une variable d'état qui contient une valeur comme zéro, un, deux, ou trois, nous pourrions charger la variable d'état avec *l'adresse* du code à exécuter à l'entrée dans le procédure. En sautant simplement à cette adresse, la machine d'état pourrait s'épargner les tests ci-dessus nécessaires pour exécuter le fragment de code approprié. Considérez l'implémentation suivante qui utilise le saut indirect :

```
State        word   State0
StateMach           proc
                    jmp    State

; Si on est dans l'état 0, ajoute BX à AX et passe à l'état 1:

State0:             add    ax, bx
                    mov    State, offset State1            ;Le met à état 1
                    ret

; Si on est dans l'état 1, soustrait BX de AX et passe à l'état 2:

State1:             sub    ax, bx
                    mov    State, offset State2            ;Passe à l'état 2
                    ret

; Si on est dans l'état 2, muliplie BX par AX et passe à l'état 3:

State2:             push   dx
                    mul    bx
                    pop    dx
                    mov    State, offset State3            ;Passe à l'état 3
                    ret

; Si on est dans l'état 3, faire la division et repasser à l'état 0:

State3:             push   dx
                    xor    dx, dx ;Zero extend AX into DX.
                    div    bx
                    pop    dx
                    mov    State, offset State0            ;Passe à l'état 0
                    ret
StateMach           endp
```

[3]En fait, elle se rappelle combien de fois, MOD 4, elle a été appelée.

L'instruction **jmp** au début de la procédure **StateMach** transfère le contrôle à l'emplacement pointé par la variable **State**. La première fois que vous appelez **StateMach,** elle pointe sur l'étiquette **State0**. Ensuite, chaque sous-section du code positionne la variable **State** pour qu'elle pointe sur le code suivant approprié.

10.5 Code spaghetti

Un problème majeur de l'assembleur est qu'il a besoin de plusieurs instructions pour réaliser une idée simple encapsulée que dans un langage de haut niveau se réaliserait en une seule instruction. Bien trop souvent, en programmant en assembleur, on peut remarquer qu'on peut gagner quelques bytes ou cycles en sautant au milieu d'une structure de programmation. Après quelques observations de cette sorte (et leurs modifications correspondantes), le code contient une suite de sauts de part et d'autre des sections du code. Si on traçait une ligne décrivant la direction du flux de contrôle après chaque saut, on noterait une figure qui ressemble à un plat de spaghetti tombé sur votre code source, d'où le terme "code spaghetti".

Ce type de code comporte un inconvénient majeur - il est difficile (au mieux) à suivre et ce n'est pas facile de comprendre ce qu'il fait. La plupart des programmes commencent de façon « structurée » seulement pour devenir du code spaghetti au nom de l'efficacité. Hélas, le code spaghetti est rarement efficace. Puisqu'il est difficile de comprendre exactement ce qui se passe, il est très difficile de déterminer si vous pouvez employer un meilleur algorithme pour améliorer le système. Par conséquent, le code spaghetti n'est pas aussi bon qu'il paraît.

Même si la production du code spaghetti peut parfois améliorer vos programmes, il faudrait toujours utiliser ce genre de programmation comme dernier recours (quand vous avez tout essayé sans arriver à ce dont vous avez besoin), il ne devrait jamais être utilisé comme une technique de programmation systematique. Commencez toujours vos programmes avec les instructions classiques **if** et **case** et à combiner des sections de code à l'aide de l'instruction **jmp** seulement une fois que tout marche correctement et il est bien compris. Naturellement, vous devriez ne jamais altérer la structure de votre code à moins que le gain n'en vaille la peine.

Un dicton célèbre dans les cercles de programmation structurée est « après les **gotos**, les pointeurs sont les éléments les plus dangereux dans un langage de programmation ». Un adage semblable est « Les pointeurs sont aux structures de données ce que les **gotos** sont aux structures de contrôle ». En d'autres termes, évitez l'utilisation excessive des pointeurs. Si les pointeurs et les **gotos** sont mauvais, alors le saut indirect doit être la plus mauvaise construction de toutes puisqu'il implique des **gotos** et des pointeurs ! Sérieusement cependant, les instructions de saut indirect devraient être évitées. Elles tendent à rendre un programme plus difficile à lire. Après tout, un saut indirect peut (théoriquement) transférer le contrôle à n'importe quelle étiquette dans un programme. Imaginez combien il serait difficile de suivre le flux à travers un programme si vous n'avez aucune idée de ce qu'un pointeur contient et que vous rencontrez un saut indirect utilisant ce pointeur. Par conséquent, vous devriez toujours être prudent à l'heure d'utiliser des sauts indirects.

10.6 Boucles

Les boucles sont ce qu'il manquait dans la description des structures de contrôle de base (séquences, décisions et boucles) qui composent un programme typique. Comme tant d'autres structures en assembleur, vous utiliserez les boucles là où vous n'auriez jamais pensé d'utiliser des boucles. La plupart des langages de haut niveau ont des structures de boucle implicites sous-jacentes. Par exemple, considérez l'instruction BASIC **IF A$ = B$ THEN 100**. Cette instruction **if** compare deux chaînes et saute à l'instruction 100 si elles sont égales. En assembleur, vous devrez écrire une boucle pour comparer chaque caractère dans A$ avec le caractère correspondant dans B$ et ensuite sauter à l'instruction 100 si et seulement si, tous les caractères correspondent. En BASIC, on ne voit aucune boucle dans le programme. En assembleur, cette instruction **if** très simple nécessite une boucle. C'est un tout petit exemple qui montre comment les boucles semblent surgir de partout.

Les boucles de programme se composent de trois composantes : une composante facultative d'initialisation, un test de terminaison de boucle et le corps de la boucle. L'ordre dans lequel ces composantes sont assemblées peut radicalement changer la manière dont la boucle fonctionne. Trois permutations de ces composantes apparaissent à tout instant. En raison de leur fréquence, ces structures de boucle sont dotées de noms spéciaux dans les HLL : boucles **while** , boucles **repeat..until** (**do..while** en C/C++) et boucles **loop..endloop**.

10.6.1 Boucles While

La boucle la plus générale est la boucle **while**. Elle prend la forme suivante :

```
WHILE  expression booléenne DO instruction;
```

Il y a deux points importants à noter au sujet de la boucle **while**. D'abord, le test pour la terminaison apparaît au début de la boucle. En second lieu, comme conséquence directe de la position du test de terminaison, le corps de la boucle peut ne jamais s'exécuter. Si la condition de terminaison existe toujours, le corps de boucle sera toujours évité.

Considérez la boucle **while** Pascal suivante :

```
I := 0;
WHILE  (I<100) do I := I + 1;
```

I : = 0; est le code d'initialisation de cette boucle. I est une variable de contrôle de boucle, parce qu'elle contrôle l'exécution du corps de la boucle. **(I<100)** est la condition de terminaison de boucle. C'est à dire, la boucle ne se terminera pas tant que **I** vaut moins de 100. **I:=I+1;** est le corps de boucle. C'est le code qui s'exécute à chaque passage de la boucle. Vous pouvez convertir ceci en assembleur 80x86 comme suit :

```
                    mov    I, 0
While Lp:           cmp    I, 100
                    jge    While Done
                    inc    I
                    jmp    While Lp
While Done:
```

Notez qu'une boucle **while** Pascal peut être facilement synthétisée en utilisant un **if** et une instruction **goto**. Par exemple, le boucle **while** Pascal présentée ci-dessus peut être remplacée par :

```
I := 0;
1:     IF (I<100) THEN BEGIN
       I := I + 1;
       GOTO 1;
END;
```

Plus généralement, toute boucle **while** peut être construite sur la base suivante :

```
       code d'initialisation optionnel
1:     IF not condition de terminaison THEN BEGIN
              corps de boucle
              GOTO 1;
       END;
```

Par conséquent, vous pouvez employer les techniques vues plus tôt dans de ce chapitre pour convertir des instructions **if** en assembleur. Tout ce dont vous aurez besoin est une instruction **jmp** (**goto**) additionnelle.

10.6.2 Boucles de type Repeat..Until

Les boucles **repeat..until** (**do..while**) testent l'arrêt conditionnent à la fin de la boucle et non au début. En Pascal, la boucle r**epeat..until** prend la forme suivante :

```
code d'initialisation optionnel
REPEAT
       corps de boucle
UNTIL condition de terminaison
```

Cette séquence exécute le code d'initialisation, le corps de boucle, puis examine une condition pour voir si la boucle est répétée. Si l'expression booléenne s'évalue à faux, la boucle se répète; sinon la boucle se termine. Les deux choses à noter au sujet de la boucle **repeat..until** est que le test de terminaison apparaît à la fin de la boucle et, comme conséquence directe de ceci, le corps de boucle s'exécute au moins une fois. Comme la boucle **while**, la boucle **repeat..until** peut être synthétisée avec une instruction **if** et un **goto**. Vous emploieriez ce qui suit :

```
code d'initialisation
```

```
1:      corps de boucle
        IF NOT condition de terminaison THEN GOTO 1
```

A partir de ce qu'on a présenté dans les sections précédentes, vous pouvez facilement synthétiser des boucles **repeat..until** en asssembleur.

10.6.3 Boucles LOOP..ENDLOOP

Si les boucles **while** testent pour la terminaison au début de la boucle et les boucles **repeat..until** vérifient la terminaison à la fin de la boucle, le seul endroit qui reste à examiner pour déterminer la terminaison est au milieu de la boucle. Les langages Pascal et C/C++[4] ne supportent pas directement une boucle de ce type, mais vous pouvez trouver des implémentations de ce type dans des langages de haut niveau comme ADA. La boucle **loop..endloop** prend la forme suivante :

```
LOOP
        corps de boucle
ENDLOOP;
```

Notez qu'il n'y a pas de condition de terminaison explicite. À moins qu'on y pourvoie, la construction **loop..endloop** forme simplement une boucle infinie. La terminaison de boucle est gérée par une instruction **if** et **goto**[5]. Considérez le (pseudo) code Pascal suivant qui utilise une construction **loop..endloop** :

```
LOOP
        READ(ch)
        IF ch = '.' THEN BREAK;
        WRITE(ch);
ENDLOOP;
```

En vrai Pascal, vous utiliseriez le code suivant pour accomplir ceci :

```
1:
        READ(ch);
        IF ch = '.' THEN GOTO 2; (* Turbo Pascal supporte BREAK! *)
        WRITE(ch);
        GOTO 1
2:
```

En assembleur, vous vous trouveriez avec quelque chose comme :

```
LOOP1:      getc
            cmp    al, '.'
            je     EndLoop
            putc
            jmp    LOOP1
EndLoop:
```

10.6.4 Boucles FOR

La boucle **for** est une forme spéciale de la boucle **while** qui répète le corps de boucle un nombre de fois déterminé. En Pascal, ce serait :

```
FOR var := initiale TO finale DO stmt
```

ou

```
FOR var := initiale DOWNTO finale DO stmt
```

Traditionnellement, la boucle **for** en Pascal a été utilisée pour traiter des tableaux et d'autres objets consultés dans l'ordre numérique séquentiel. Ces boucles peuvent être converties directement en assembleur comme suit :

En Pascal :

[4]Techniquement, le C/C++ supporte une boucle de ce type. "for(;;)" avec break fournit cette fonctionnalité.

[5]Beaucoup de langages de haut niveau utilisient des instructions comme NEXT, BREAK, CONTINUE, EXIT et CYCLE plutôt que GOTO ; mais elles sont toutes des formes différentes de l'instruction GOTO.

```
        FOR var := start TO stop DO stmt;
```

En Assembleur :

```
                    mov    var, start
FL:                 mov    ax, var
                    cmp    ax, stop
                    jg     EndFor

; Le code correspondant à stmt vient ici.

                    inc    var
                    jmp    FL
EndFor:
```

Heureusement, la plupart des boucles **for** répètent un/des instruction(s) un nombre de fois fixe. Par exemple,

```
    FOR I := 0 TO 7 DO write(ch);
```

Dans les situations comme celle-ci, il vaut mieux employer l'instruction de boucle du 80x86 plutôt que simuler une boucle **for** :

```
                    mov    cx, 7
LP:                 mov    al, ch
                    call   putc
                    loop   LP
```

Gardez à l'esprit que l'instruction **loop** apparaît normalement à la fin d'une boucle tandis que la boucle **for** teste la terminaison au début de la boucle. Par conséquent, vous devriez prendre des précautions pour empêcher une boucle de s'emballer au cas où cx est zéro (ce qui amènerait l'instruction **loop** à répéter la boucle 65.536 fois) ou que la valeur de terminaison est moindre que la valeur de début. Dans le cas de

```
    FOR var := start TO stop DO stmt;
```

en supposant que vous n'utilisez pas la valeur var dans la boucle, vous utiliseriez probablement le code assembleur :

```
                    mov    cx, stop
                    sub    cx, start
                    jl     SkipFor
                    inc    cx
LP:                 stmt
                    loop   LP
SkipFor:
```

Rappelez-vous, les instructions **sub** et **cmp** modifient les drapeaux de façon identique. Par conséquent, cette boucle sera sautée si **stop** est plus petit que **start**. Elle sera répétée (**stop**-**start**)**+1** fois autrement. Si vous devez référencer la valeur de **var** dans la boucle, vous pourriez employer le code suivant :

```
                    mov    ax, start
                    mov    var, ax
                    mov    cx, stop
                    sub    cx, ax
                    jl     SkipFor
                    inc    cx
LP:                 stmt
                    inc    var
                    loop   LP
SkipFor:
```

La version **downto** apparaît dans les exercices.

10.7 Les boucles et l'utilisation des registres

Etant donné que le 80x86 accède à ses registres beaucoup plus rapidement qu'aux emplacements en mémoire, les registres sont l'endroit idéal pour placer des variables de contrôle de boucle (particulièrement pour de petites boucles). Ce point est amplifié du fait que l'instruction de boucle nécessite l'utilisation du registre **cx**. Cependant, il y a quelques problèmes associés à l'utilisation des registres dans une boucle. Le problème primordial à l'heure d'utiliser les registres à ces fins est que les registres sont une ressource limitée. En particulier, il n'y a qu'un registre **cx**. Par conséquent, ce qui suit ne fonctionnera pas correctement :

```
                    mov    cx, 8
Loop1:              mov    cx, 4
Loop2:              stmts
                    loop   Loop2
                    stmts
                    loop   Loop1
```

L'intention ici, naturellement, était de créer un ensemble de boucles imbriquées, c.-à-d., une boucle à l'intérieur d'une autre. La boucle interne (**Loop2**) devrait se répéter quatre fois pour chacune des huit exécutions de la boucle externe (**Loop1**). Malheureusement, les deux boucles emploient l'instruction **loop**. Par conséquent, ceci donnera une boucle infinie puisque **cx** sera mis à zéro (que **loop** traite comme 65.536) à l'fin de la première instruction **loop**. Puisque **cx** est toujours à zéro quand on rencontre la deuxième instruction **loop**, le contrôle sera toujours transféré à l'étiquette **Loop1**. La solution ici est sauver et restaurer le registre **cx** ou d'utiliser un registre différent de **cx** pour la boucle externe :

```
                    mov    cx, 8
Loop1:              push   cx
                    mov    cx, 4
Loop2:              stmts
                    loop   Loop2
                    pop cx
                    stmts
                    loop   Loop1
```

ou :

```
                    mov    bx, 8
Loop1:              mov    cx, 4
Loop2:              stmts
                    loop   Loop2
                    stmts
                    dec    bx
                    jnz    Loop1
```

La corruption de registre est l'une des sources primordiales des bogues dans les boucles des programmes en assembleur, soyez toujours vigilant face à ce problème.

10.8 Amélioration des performances

Les microprocesseurs 80x86 exécutent des séquences d'instructions à des vitesses stupéfiantes. Vous rencontrerez rarement un programme lent qui ne contient aucune boucle. Puisque les boucles sont la source primaire des problèmes de performances dans un programme, elles sont l'endroit à examiner pour essayer d'accélérer votre logiciel. Étant donné qu'un traité sur la façon d'écrire des programmes efficaces est au delà des ambitions de ce chapitre, il y a quelques points quand-même à prendre en compte à l'heure de concevoir des boucles. Et ils consistent à enlever des instructions inutiles de vos boucles afin de réduire le temps qu'elles prennent pour exécuter une itération.

10.8.1 Déplacer la condition de terminaison à la fin d'une boucle

Considérez le graphe de flux suivant pour les trois types de boucles présentées plus tôt :

Boucle **Repeat..until** :

```
Code d'initialisation
        Corps de boucle
Test pour la terminaison
Code suivant la boucle
```

Boucle **While** :

```
Code d'initialisation
Test pour la terminaison
        Corps de boucle
        Saute en arrière au test
Code suivant la boucle
```

Boucle **Loop..endloop** :

```
Code d'initialisation
        Corps de boucle, première partie
        Test pour la terminaison
        Corps de boucle, deuxième partie
        Saute en arrière au corps de boucle 1
Code suivant la boucle
```

Comme vous pouvez le voir, la boucle **repeat..until** est la plus simple du groupe. Ceci est reflété dans le code d'assembleur exigé pour implémenter ces boucles. Considérez les boucles **repeat..until** et **while** suivantes qui sont identiques :

```
SI := DI - 20;
while (SI <= DI) do
begin

      stmts
      SI := SI + 1;


end;
```

```
SI := DI - 20;
repeat


stmts
      SI := SI + 1;


until SI > DI;
```

Le code assembleur pour ces deux boucles est [6] :

```
      mov   si, di
      sub   si, 20
WL1:  cmp   si, di
      jnle  QWL
      stmts
      inc   si
      jmp   WL1
QWL:
```

```
      mov   si, di
      sub   si, 20
U:    stmts
      inc   si
cmp   si, di
      jng   U
```

Comme vous pouvez le voir, déterminer la condition de terminaison à la fin de la boucle nous a permis d'enlever une instruction **jmp**. Ceci peut être significatif si cette boucle est imbriquée à l'intérieur d'autres boucles. Dans l'exemple précédent il n'y avait pas un problème pour exécuter le corps au moins une fois. Étant donné la définition de la boucle, vous pouvez facilement voir qu'elle sera exécutée exactement 20 fois. En supposant que **cx** est disponible, cette boucle se réduit à :

```
                lea   si, -20[di]
                mov   cx, 20
WL1:            stmts
                inc   si
                loop  WL1
```

Malheureusement, ce n'est pas toujours aussi facile que ça. Considérez le code Pascal suivant :

```
WHILE (SI <= DI) DO BEGIN
      stmts
      SI := SI + 1;
END;
```

[6] Bien sûr, un bon compilateur reconnaîtrait que ces deux boucles exécutent la même opération et génèrent un code identique pour les deux. Cependant, tous les compilateurs ne sont pas aussi bons que ça.

Dans cet exemple particulier, nous n'avons pas d'idée de ce que **si** contient à l'entrée de la boucle. Par conséquent, nous ne pouvons pas présupposer que le corps de boucle s'exécutera au moins une fois. Donc, nous devons faire le test avant d'exécuter le corps. Il peut être placé à la fin de la boucle par l'inclusion d'une seule instruction **jmp** :

```
                    jmp    short Test
RU:                 stmts
                    inc    si
Test:       cmp     si, di
                    jle    RU
```

Bien que le code soit aussi long que dans la boucle **while** originale, l'instruction **jmp** ne s'exécute qu'une fois plutôt qu'à chaque répétition de la boucle. Notez que ce léger gain en efficacité est obtenu à l'aide d'une perte légère de lisibilité. La deuxième séquence de code ci-dessus est plus près du code spaghetti que l'implémentation originale. C'est souvent le prix d'un petit gain de performances. Par conséquent, vous devriez soigneusement analyser votre code pour vous assurer que le gain de performance équivaut la perte de clarté. Le plus souvent, les programmeurs de langage d'assemblage sacrifient la clarté pour des gains douteux en performances, produisant des programmes impossibles à comprendre.

10.8.2 Exécuter la boucle en ordre décroissant

En raison de la nature des drapeaux sur le 80x86, les boucles qui descendent (ou montent) à partir d'un nombre jusqu'à zéro sont plus efficaces que les autres. Comparez les boucles Pascal suivantes et le code qu'elles produisent :

```
for I := 1 to 8 do
            K := K + I - J;

            mov    I, 1
FLP:        mov    ax, K
            add    ax, I
            sub    ax, J
            mov    K, ax
            inc    I
            cmp    I, 8
            jle    FLP
```

```
for I := 8 downto 1 do
            K := K + I - j;

            mov    I, 8
FLP:  mov   ax, K
      add   ax, I
      sub   ax, J
      mov   K, ax
            dec    I
            jnz    FLP
```

Notez qu'en exécutant la boucle de huit à un (le code du côté droit) nous avons économisé une comparaison à chaque répétition de la boucle.

Malheureusement, vous ne pouvez pas forcer toutes les boucles à fonctionner en arrière. Cependant, avec un peu d'effort et de rigueur vous devriez pouvoir façonner la plupart des boucles pour qu'elles fonctionnent de cette façon. Une fois que vous obtenez une boucle opérant ainsi, elle est une bonne candidate pour l'instruction **loop** (qui aura une exécution meilleure sur les CPU pre-486).

L'exemple ci-dessus marchait bien parce que la boucle descendait de huit à un. La boucle s'est terminée quand la variable de contrôle de boucle est arrivée à zéro. Que se passe-t-il si vous devez exécuter la boucle avec une variable de contrôle de boucle qui finit à zéro ? Par exemple, supposez que la boucle ci-dessus doive s'étendre de sept à zéro. Tant que la limite supérieure est positive, vous pouvez substituer l'instruction **jns** à l'instruction **jnz** pour répéter la boucle ci-dessus un nombre de fois spécifique :

```
            mov    I, 7
FLP:        mov    ax, K
            add    ax, I
            sub    ax, J
            mov    K, ax
            dec    I
            jns    FLP
```

Cette boucle se répétera huit fois avec I prenant la valeur de sept à zéro à chaque exécution de la boucle. Mais quand la décrémentation atteint -1, le drapeau de signe se verra activé et la boucle terminera.

Gardez à l'esprit que quelques valeurs peuvent sembler positives mais sont en fait négatives. Si la variable de contrôle de boucle est un byte, alors les valeurs dans la plage 128..255 sont négatives. De même, les valeurs de 16 bits dans la plage 32768..65535 sont négatives. Par conséquent, l'initialisation de la variable de contrôle de boucle avec n'importe quelle valeur dans la plage 129..255 ou 32769..65535 (ou, naturellement, zéro) provoquera la terminaison de la boucle après une seule exécution. Ceci peut vous mettre dans de beaux draps si vous n'êtes pas prudent.

10.8.3 Calculs invariables dans une boucle

Un calcul invariable dans une boucle est un calcul donnant toujours le même résultat. Vous n'avez pas besoin de faire de tels calculs à l'intérieur de la boucle. Vous pouvez les calculer à l'extérieur et faire référence à la valeur du calcul à l'intérieur. Le code Pascal suivant montre une boucle qui contient un calcul invariable :

```
FOR I : = 0 TO N DO
    K : = K+(I+J-2);
```

Puisque J ne change jamais, l'expression secondaire "J-2" peut être calculée en dehors de la boucle et on peut n'utiliser que son résultat à l'intérieur :

```
temp := J-2;
FOR I : = 0 TO N DO
    K : = K+(I+temp);
```

Naturellement, si vous désirez vraiment améliorer l'efficacité de cette boucle en particulier, vous feriez mieux, la plupart du temps à utiliser cette formule :

Le calcul de K est basé sur cette formule :

Cependant, de simples calculs comme celui-ci ne sont pas toujours possibles. Cela étant, ceci démontre qu'un meilleur algorithme est presque toujours meilleur que le code le plus astucieux que vous puissiez trouver.

En assembleur, les calculs invariables sont encore plus astucieux. Considérez cette conversion du code Pascal ci-dessus :

```
            mov    ax, J
            add    ax, 2
            mov    temp, ax
            mov    ax, n
            mov    I, ax
FLP:        mov    ax, K
            add    ax, I
            sub    ax, temp
            mov    K, ax
            dec    I
            cmp    I, -1
            jg     FLP
```

Naturellement, la première amélioration que nous pouvons apporter est de placer la variable de contrôle de boucle (**I**) dans un registre. Ceci produit le code suivant :

```
            mov    ax, J
            inc    ax
            inc    ax
            mov    temp, ax
            mov    cx, n
```

```
FLP:        mov    ax, K
            add    ax, cx
            sub    ax, temp
            mov    K, ax
            dec    cx
            cmp    cx, -1
            jg     FLP
```

Cette opération accélère la boucle en enlevant un accès mémoire de chaque répétition de la boucle. Pour continuer dans ce sens, pourquoi ne pas utiliser un registre pour contenir la valeur "temp" plutôt qu'un emplacement de mémoire :

```
            mov    bx, J
            inc    bx
            inc    bx
            mov    cx, n
FLP:        mov    ax, K
            add    ax, cx
            sub    ax, bx
            mov    K, ax
            dec    cx
            cmp    cx, -1
            jg     FLP
```

En outre, l'accès à la variable K peut être aussi enlevé de la boucle :

```
            mov    bx, J
            inc    bx
            inc    bx
            mov    cx, n
            mov    ax, K
FLP:        add    ax, cx
            sub    ax, bx
            dec    cx
            cmp    cx, -1
            jg     FLP
            mov K, ax
```

Une dernière amélioratrion qui demande à être apportée est de substituer l'instruction **loop** à **dec cx** / **cmp cx,** -1 / **JG FLP**. Malheureusement, cette boucle doit être répétée quand la variable de contrôle de boucle atteint zéro et l'instruction **loop** ne peut pas faire ça. Cependant, nous pouvons occulter la dernière exécution de la boucle (voir la section suivante) et faire ce calcul en dehors de la boucle comme suit :

```
            mov    bx, J
            inc    bx
            inc    bx
            mov    cx, n
            mov    ax, K
FLP:        add    ax, cx
            sub    ax, bx
            loop   FLP
            sub    ax, bx
            mov    K, ax
```

Comme vous pouvez le voir, ces améliorations ont considérablement réduit le nombre d'instructions exécutées à l'intérieur de la boucle et les instructions qui apparaissent à l'intérieur de la boucle sont très rapides puisqu'elles référencent toutes des registres et non des emplacements mémoire.

Enlever des calculs invariables et des accès mémoire inutiles d'une boucle (en particulier une boucle intérieure dans un ensemble de boucles imbriquées), peut produire des améliorations de performances spectaculaires dans un programme.

10.8.4 Démonter les boucles

Pour de petites boucles, c.-à-d., celles dont le corps ne contient que quelques instructions, la charge exigée pour gérer la boucle peut constituer un pourcentage significatif de la durée totale de traitement. Par exemple, regardez le code Pascal suivant et son code assembleur 80x86 associé :

```
      FOR I := 3 DOWNTO 0 DO A [I] := 0;

            mov    I, 3
FLP:        mov    bx, I
            shl    bx, 1
            mov    A [bx], 0
            dec    I
            jns    FLP
```

Chaque exécution de la boucle nécessite cinq instructions. Une seule instruction effectue l'opération désirée (entrant zéro dans un élément de A). Les quatre instructions restantes convertissent la variable de contrôle de boucle en index dans A et contrôlent la répétition de la boucle. Par conséquent, cela prend 20 instructions pour faire l'opération logiquement nécessitée par quatre.

Bien qu'on pourrait améliorer cette boucle beaucoup, si nous nous basions sur les informations présentées jusqu'ici, considérez soigneusement ce que c'est que fait exactement cette boucle -- elle ne fait que garder quatre zéros dans A[0] à A[3]. Une approche plus efficace serait d'utiliser quatre instructions **mov** pour accomplir la même tâche. Par exemple, si A est un tableau de mots, alors le code suivant initialise A beaucoup plus rapidement que le code ci-dessus :

```
            mov    A, 0
            mov    A+2, 0
            mov    A+4, 0
            mov    A+6, 0
```

Vous pouvez améliorer la vitesse d'exécution et la taille de ce code en utilisant le registre **ax** pour contenir zéro :

```
            xor    ax, ax
            mov    A, ax
            mov    A+2, ax
            mov    A+4, ax
            mov    A+6, ax
```

Bien que ce soit un exemple insignifiant, il montre l'avantage du démontage de boucle. Si cette simple boucle se trouvait enfouie à l'intérieur d'un ensemble de boucles imbriquées, la réduction par un rapport de 5 à 1 des instructions pourrait probablement doubler les performances de cette section de votre programme.

Naturellement, vous ne pouvez pas démonter toutes les boucles. Des boucles qui s'exécutent un nombre variable de fois ne peuvent pas être démontées parce qu'il y a rarement moyen de déterminer (à l'assemblage) le nombre de fois que la boucle a besoin d'être exécutée. Par conséquent, le démontage de boucle est un procédé mieux adapté aux boucles qui s'exécutent un nombre déterminé de fois.

Mais, même dans ce cas, la boucle peut ne pas être un bon candidat pour le démontage. Ce dernier produit des améliorations de performances impressionnantes quand le nombre d'instructions nécessitées pour contrôler la boucle (et effectuer d'autres opérations de gestion), représente un pourcentage significatif du nombre total d'instructions dans la boucle. Si la boucle ci-dessus avait contenu 36 instructions dans son corps (en dehors des quatre instructions de gestion de boucle), alors l'amélioration de performances serait, au mieux, seulement 10% (comparé aux 300-400% dont elle bénéficie maintenant). Par conséquent, les coûts du démontage d'une boucle, à savoir, tout le code supplémentaire qui doit être inséré dans votre programme, atteint rapidement un point de rendement moindre au fur et à mesure que le corps de la boucle se développe ou que le nombre d'itérations augmente. En outre, entrer ce code dans votre programme peut devenir une vraie corvée. C'est pourquoi le démontage des boucles trouve sa meilleure application dans les boucles avec peu d'instructions.

Notez que les puces x86 superscalaires (Pentium et au-delà) ont un *matériel de prévision de branchement* et utilisent d'autres techniques pour améliorer les performances. Le déroulement des boucles sur des systèmes de ce type peut, en fait, *ralentir* le code, si votre boucle n'est pas courte.

10.8.5 Variables d'induction

Ce qui suit est une légère modification de la boucle présentée à la section précédente :

```
      FOR I := 0 TO 255 DO A [I] := 0;
```

```
            mov    I, 0
FLP:        mov    bx, I
            shl    bx, 1
            mov    A [bx], 0
            inc    I
            cmp    I, 255
            jbe    FLP
```

Bien que démonter ce code continuera à produire une amélioration énorme de performances, cela prendra tout quand-même 257 instructions pour accomplir cette tâche[7], beaucoup trop pour n'importe quelle application, sauf les applications en temps-réel. Cependant, vous pouvez réduire grandement le temps d'exécution du corps de la boucle en utilisant des des *variables d'induction*. Une variable d'induction est une variable dont la valeur dépend entièrement de la valeur d'une autre variable. Dans l'exemple ci-dessus, l'index dans le tableau A suit la variable de contrôle de boucle (il est toujours égal à la valeur de la variable de contrôle de boucle fois deux). Puisque **I** n'apparaît nulle part ailleurs dans la boucle, cela n'a pas de sens de faire tous les calculs sur **I**. Pourquoi ne pas opérer directement sur la valeur de l'index de tableau ? Le code suivant démontre cette technique :

```
            mov    bx, 0
FLP:        mov    A [bx], 0
            inc    bx
            inc    bx
            cmp    bx, 510
            jbe    FLP
```

Ici, plusieurs instructions accédant à la mémoire ont été remplacées par des instructions qui accèdent seulement aux registres. Une autre amélioration à apporter est de raccourcir l'instruction **MOV A[bx], 0** en utilisant le code suivant :

```
            lea    bx, A
            xor    ax, ax
FLP:        mov    [bx], ax
            inc    bx
            inc    bx
            cmp    bx, offset A+510
            jbe    FLP
```

Cette transformation de code améliore encore les performances de la boucle. Cependant, nous pouvons en améliorer l'exécution encore plus en utilisant l'instruction **loop** et le registre **cx** pour éliminer l'instruction cmp[8] :

```
            lea    bx, A
            xor    ax, ax
            mov    cx, 256
FLP:        mov    [bx], ax
            inc    bx
            inc    bx
            loop   FLP
```

Cette dernière transformation de la boucle produit la version la plus rapide en exécution de ce code[9].

10.8.6 Autres Améliorations de Performances

Il y a beaucoup d'autres manières d'améliorer les performances d'une boucle dans vos programmes en assembleur. Pour des suggestions additionnelles, un bon livre sur les compilateurs tels que "Compilers, Principles, Techniques, and Tools" par Aho, Sethi, et Ullman serait une bonne référence. Des considérations additionnelles d'efficacité seront discutées dans le volume sur l'efficacité et l'optimisation.

[7]Pour cette boucle particulière, l'instruction STOSW pourrait produire une grande amélioration de performances sur beaucoup de processeurs 80x86. Utiliser l'instruction STOSW nécessiterait environ six instructions pour ce code. Voyez le chapitre sur les instructions de chaîne pour plus de détails.

[8]L'instruction LOOP n'est pas le meilleur choix sur les 486 et les processeurs Pentium puisque "dec cx "suivi de "jne lbl" s'exécute en fait plus rapidement.

[9]Plus rapide est une assertion dangereuse à utiliser ici ! Mais c'est le plus rapide des exemples présentés ici.

10.9 Instructions imbriquées

Tant que vous vous en tenez aux modèles fournis dans les exemples présentés dans ce chapitre, il est très facile d'imbriquer des blocs d'instructions les uns à l'intérieur des autres. Le secret pour s'assurer que vos séquences en assembleur s'imbriquent bien est de s'assurer que chaque construction a un point d'entrée et un point de sortie. Si c'est le cas, alors vous trouverez facile de combiner des blocs d'instructions. Toutes les instructions traitées dans ce chapitre suivent cette règle.

Les instructions sans doute les plus couramment imbriquées sont les instructions **if..then..else**. Pour voir à quel point il facile est d'imbriquer ces instructions en assembleur, considérez le code Pascal suivant :

```
      if (x = y) then
            if (I >= J) then writeln('At point 1')
            else writeln('At point 2)
      else write('Error condition');
```

Pour convertir cet **if..then..else** imbriqué en assembleur, commencez par le bloc **if** externe, convertissez-le en assembleur, puis passez au bloc **if** le plus interne :

```
; if (x = y) then

            mov   ax, X
            cmp   ax, Y
            jne   Else0

; Mettre le bloc IF le plus interne ici

            jmp   IfDone0

; Else write('Error condition');

Else0:      print
            byte  "Error condition",0
IfDone0:
```

Comme vous pouvez le voir, le code ci-dessus gère l'instruction "if (X=Y)...", en laissant un endroit pour le second bloc **if**. Ajoutez maintenant le second bloc **if** comme suit :

```
; if (x = y) then

            mov   ax, X
            cmp   ax, Y
            jne   Else0

;     IF ( I >= J) then writeln('At point 1')

            mov   ax, I
            cmp   ax, J
            jnge  Else1
            print
            byte  "At point 1",cr,lf,0
            jmp   IfDone1

;     Else writeln ('At point 2');

Else1:      print
            byte "At point 2",cr,lf,0
IfDone1:

            jmp   IfDone0

; Else write('Error condition');

Else0:      print
            byte  "Error condition",0
```

```
IfDone0:
```

Le bloc **if** Imbriqué apparaît en italiques ci-dessus pour le faire ressortir.

Il existe une optimisation évidente que vous pouvez ne pas vouloir considérer à moins que la vitesse ne soit un problème réel pour vous. Notez, dans le bloc d'instructions **if** le plus interne, ci-dessus, que l'instruction **JMP IFDONE1** saute simplement à une instruction **jmp** qui transfère le contrôle à **IfDone0**. Il est très tentant de remplacer le premier **jmp** par un autre qui saute directement à **IFDone0**. En effet, quand vous commencez à optimiser votre code, ce serait une bonne optimisation à faire. Cependant, ces optimisations sont recommandées seulement quand la vitesse est une priorité, car elles peuvent rendre votre code plus dur à lire et à comprendre. Le comportement désiré d'une structure de contrôle est qu'elle n'ait qu'une entrée et une sortie. Modifier ce saut comme décrit donnerait au bloc **if** le plus interne deux points de sortie.

La boucle **for** est une autre structure de contrôle couramment imbriquée. Encore une fois, la clef pour construire des structures imbriquées est commencer par l'objet extérieur et compléter les membres intérieurs après. Comme exemple, considérez les boucles **for** imbriquées suivantes qui aditionnent les éléments d'une paire de tableaux bidimensionnels :

```
for I : = 0 to 7 do
    for k : = 0 to 7 do
           A [i, j] : = B [i, j] + C [i, j];
```

Comme précédemment, commencez par construire la boucle extérieure en premier. Ce code suppose que **dx** sera la variable de contrôle de boucle pour la boucle externe (c'est-à-dire, **dx** est équivalent à "i") :

```
; for dx := 0 to 7 do

            mov    dx, 0
ForLp0:     cmp    dx, 7
            jnle   EndFor0

; Mettre la boucle FOR la plus interne ici

            inc    dx
            jmp    ForLp0
EndFor0:
```

Ajoutez maintenant le code pour la boucle **for** imbriquée. Notez l'utilisation du registre **cx** comme variable de contrôle de boucle dans la boucle **for** la plus interne de ce code.

```
; for dx := 0 to 7 do

            mov    dx, 0
ForLp0:     cmp    dx, 7
            jnle   EndFor0

;     for cx := 0 to 7 do

            mov    cx, 0
ForLp1:     cmp    cx, 7
            jnle   EndFor1

; Mettre le code pour A[dx,cx] := b[dx,cx] + C [dx,cx] ici

            inc    cx
            jmp    ForLp1
EndFor1:

            inc    dx
            jmp    ForLp0
EndFor0:
```

De nouveau, la boucle **for** la plus interne est en italiques dans le code ci-dessus pour la faire ressortir. L'étape finale est d'ajouter le code qui exécute le calcul effectif.

10.10 Boucles de retard

La plupart du temps, l'ordinateur est trop lent pour le goût de la plupart des gens. Cependant, il y a des occasions où il est vraiment trop rapide. Une solution courante est de créer une boucle vide pour perdre un peu de temps. En Pascal vous verrez couramment des boucles comme :

```
for i : = 1 to 10000  do;
```

En assembleur, vous pourriez voir une boucle comparable :

```
                mov    cx, 8000h
DelayLp:   loop   DelayLp
```

En choisissant soigneusement le nombre d'itérations, vous pouvez obtenir un intervalle de retard relativement précis. Il y a, cependant, un hic. Cet intervalle de retard relativement précis va être précis seulement sur *votre* ordinateur. Si vous déplacez votre programme sur une machine différente avec un CPU différente, avec une fréquence d'horloge, un nombre de *wait states* différents, un cache de taille différente, ou encore une demi-douzaine d'autres particularités, vous constaterez que votre boucle retard prend un laps de temps complètement différent. Puisqu'il y a une différence de un à cent dans la vitesse les PC aujourd'hui, entre les plus lents et les plus rapides, cela ne devrait pas vous surprendre que la boucle ci-dessus s'exécutera 100 fois plus rapidement sur certaines machines que sur d'autres.

Le fait qu'un CPU tourne 100 fois plus rapidement que d'autres ne réduit pas la nécessité d'avoir une boucle de retard qui s'exécute pendant un certain laps de temps fixe. En effet, cela rend le problème encore plus important. Heureusement, le PC fournit un timer basé sur le matériel qui fonctionne à la même vitesse indépendamment de la vitesse du processeur. Ce timer maintient l'heure pour le logiciel d'exploitation, il est donc important qu'il fonctionne à la même vitesse que vous soyez sur un 8088 ou un Pentium. Dans le chapitre sur les interruptions, vous apprendrez à patcher effectivement dans ce dispositif pour exécuter diverses tâches. Pour l'instant, nous tirerons simplement profit du fait que cette puce chrono force le CPU à incrémenter un emplacement de mémoire de 32 bits (40:6ch) environ 18.2 fois par seconde. En regardant cette variable, nous pouvons déterminer la vitesse du CPU et ajuster la valeur de compte pour une boucle vide en conséquence.

L'idée fondamentale du code suivant est d'observer la variable timer du BIOS jusqu'à ce qu'elle change. Une fois qu'elle a changé, commencez à compter le nombre d'itérations avec une boucle quelconque jusqu'à ce que la variable du timer du BIOS change encore. Avoir noté le nombre d'itérations, si vous exécutez une boucle semblable le même nombre de fois, elle devrait avoir besoin d'environ 1/18.2 secondes pour s'exécuter.

Le programme suivant démontre comment créer une telle routine **Delay** :

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

; PPI_B est l'adresse d'E/S du port du contrôle de clavier/haut-
; parleur. Ce programme lui accède simplement en introduisant un
; grand nombre d'états d'attente (wait states) sur les machines plus
; rapides. Puisque le chipset PPI (Interface de Périphérique
; Programmable) fonctionne à environ la même vitesse sur tous les PCS,
; accéder à ce chipset ralentit la plupart des machines par un facteur
; de deux sur les machines les plus lentes.

PPI_B        equ    61h

; RTC est l'adresse de la variable timer du BIOS (40:6ch).
; Le code d'interruption du timer du BIOS incrémente cet emplacement
; de 32 bits environ toutes les 55 ms (1/18.2 secondes). Le code qui
; initialise tout pour la routine Delay lit cet emplacement pour
; déterminer quand les 1/18èmes de secondes sont passés.

RTC                textequ      <es:[6ch]>

dseg               segment      para public 'data'

; TimedValue contient le nombre d'itérations que la boucle retard
```

```
; doit répéter de façon à perdre 1/18.2 secondes.

TimedValue   word  0

; RTC2 est une variable bidon utilisée par la routine Delay pour
; simuler l'accès à une variable du BIOS.

RTC2               word  0

dseg               ends

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

; Programme principal qui teste la sous-routine DELAY.

Main               proc
                   mov   ax, dseg
                   mov   ds, ax
                   print
                   byte  "Delay test routine",cr,lf,0

; OK, voyons combien de temps cela prend pour compter 1/18ème d'une
; seconde. D'abord, faisons pointer ES sur le segment 40h en mémoire
; Toutes les variables du BIOS sont dans le segment 40h.
;
; Ce code commence par lire la variable de mémoire timer et attendre
; jusqu'à ce qu'elle change. Une fois qu'elle a changé nous pouvons
; commencer à chronométrer jusqu'à ce que le prochain changement se
; produise. Cela nous donnera 1/18.2 secondes. Nous ne pouvons pas
; commencer à chronométrer de but en blanc car nous pourrions être au
; milieu d'une période de 1/18.2 de secondes.

                   mov   ax, 40h
                   mov   es, ax
                   mov   ax, RTC
RTCMustChange:     cmp   ax, RTC
                   je    RTCMustChange

; OK, commençons à chronométrer le nombre d'itérations qu'un 18ème
; d'une seconde prend. Notez que ce code doit être très semblable au
; code dans la routine Delay.

                   mov   cx, 0
                   mov   si, RTC
                   mov   dx, PPI_B
TimeRTC:           mov   bx, 10
DelayLp:           in    al, dx
                   dec   bx
                   jne   DelayLp
                   cmp   si, RTC
                   loope TimeRTC

                   neg   cx                      ;Compte à rebours de CX!
                   mov   TimedValue, cx          ;Le sauver

                   mov   ax, ds
                   mov   es, ax

                   printf
                   byte  "TimedValue = %d",cr,lf
                   byte  "Press any key to continue",cr,lf
                   byte  "This will begin a delay of five "
                   byte  "seconds",cr,lf,0
                   dword TimedValue
```

```
                    getc

                    mov    cx, 90
DelayIt:            call   Delay18
                    loop   DelayIt

Quit:         ExitPgm            ;Macro DOS pour quitter programme.
Main                endp

; Delay18-          Cette routine retarde pour approximativement 1/18ème
;                   de sec. Vraisemblablement, la variable "TimedValue"
;                   dans DS a été initialisée avec une valeur de compte
;                   à rebours appropriée avant d'appeler ce code.

Delay18             proc   near
                    push   ds
                    push   es
                    push   ax
                    push   bx
                    push   cx
                    push   dx
                    push   si

                    mov    ax, dseg
                    mov    es, ax
                    mov    ds, ax

; Le code suivant contient deux boucles. La boucle imbriquée interne
; se répète 10 fois. La boucle extérieure se répète le nombre de fois
; déterminé pour perdre 1/18.2 secondes. Cette boucle accède au port
; matériel "PPI_B" afin d'introduire beaucoup d'états d'attente sur
; les processeurs les plus rapides. Ceci aide à égaliser les délais
; sur les machines très rapides en les ralentissant.
; Notez que l'accès à PPI_B est seulement fait pour introduire ces
; wait states, les données lues sont sans intérêt pour ce code.
;
; Notez la similitude de ce code avec le code dans le programme
; principal qui initialise la variable TimedValue.

                    mov    cx, TimedValue
                    mov    si, es:RTC2
                    mov    dx, PPI_B

TimeRTC:            mov    bx, 10
DelayLp:            in     al, dx
                    dec    bx
                    jne    DelayLp
                    cmp    si, es:RTC2
                    loope TimeRTC

                    pop    si
                    pop    dx
                    pop    cx
                    pop    bx
                    pop    ax
                    pop    es
                    pop    ds
                    ret
Delay18             endp

cseg                ends

sseg                segment      para stack 'stack'
stk                 word         1024 dup (0)
sseg                ends
                      end        Main
```

10.11 Exemple de programme

Le programme d'exemple de ce chapitre est un simple jeu d'alunissage. Bien que la simulation n'en soit pas terriblement réaliste, ce programme démontre l'utilisation et l'optimisation de plusieurs structures de contrôle différentes comprenant des boucles, des instructions if..then..else, et ainsi de suite.

```
; Petit jeu "Moon Lander"
;
; Randall Hyde
; 2/8/96
;
; Ce programme est un exemple d'un petit jeu "moon lander" sans
; prétemptions qui simule un module lunaire alunissant sur la surface.
; de la lune. Au temps T=0 la vitesse de descente du vaisseau spatial
; est de 1000 pieds/sec, le vaisseau a 1000 unités de carburant, et le
; vaisseau est à 10.000 pieds au-dessus de la surface de la lune. Le
; pilote (l'utilisateur) peut spécifier combien de carburant brûler à
; chaque seconde.
;
; Notez que tous les calculs sont approximatifs puisque tout est
; fait avec l'arithmétique en nombres entiers.

; Quelques constantes importantes

InitialVelocity     =      1000
InitialDistance     =      10000
InitialFuel  =      250
MaxFuelBurn  =      25
MoonsGravity =      5                   ;Approx 5 pieds/sec/sec
AccPerUnitFuel      =      -5                  ;-5 pieds/sec/sec par unité carbu

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

dseg                segment      para public 'data'

; Distance actuelle de la Surface de la Lune:

CurDist             word   InitialDistance

; Vitesse actuelle:

CurVel              word   InitialVelocity

; Total carburant restant à brûler:

FuelLeft            word   InitialFuel

; Quantité de carburant à utiliser pour le jet courant.

Fuel                word   ?

; Distance parcourue dans la dernière seconde.

Dist                word   ?

dseg                ends

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

; GETI-      Lit une variable entier donnée par l'utilisateur et renvoie
;            sa valeur dans le registre AX. Si l'utilisateur entre des
;            inepties, ce code incitera l'utilisateur à ressaisir la
```

```
;            valeur.

geti                texteq      <call _geti>
_geti       proc
                    push   es
                    push   di
                    push   bx

; Lit une chaîne de caractères fournis par l'utilisateur.
;
; Notez qu'il y a deux boucles (imbriquées) ici. La boucle externe
; (GetILp) répète l'opération getsm tant que l'utilisateur rentre un
; nombre invalide. La boucle interne (ChkDigits) vérifie les
; caractères individuels dans la chaîne en entrée pour s'assurer
; qu'ils sont tous des chiffres décimaux.

GetILp:             getsm

; Vérifie que cette chaîne ne contient pas de caractères non-numéraux:
;
; while (([bx] >= '0') and ([bx] <= '9') bx := bx + 1;
;
; Notez la manière astucieuse de transformer la boucle while en boucle
; repeat..until,

                    mov    bx, di              ;Pointeur sur début de la chaîne.
                    dec    bx
ChkDigits:   inc    bx
                    mov    al, es:[bx]  ;Trouve caractère suivant.
                    IsDigit                    ;Caractère décimal ?
                    je     ChkDigits           ;Répète si oui.
                    cmp    al, 0        ;Fin de chaîne ?
                    je     GotNumber

; OK, nous venon de trouver un caractère non-décimal. Se plaindre et
; demander à l'utilisateur de réentrer la valeur.

                    free                       ;Libère espace alloué par getsm.
                    print
                    byte   cr,lf
                    byte   "Valeur entière non signée illégale, "
                    byte   "S.V.P., ressaisis.",cr,lf
                    byte   "(pas d'espaces, caracts non-chiffres, etc.):",0
                    jmp    GetILp

; OK, ES:DI pointe sur quelquechose ressemblant à un nombre. Le
; convertir en entier.

GotNumber:   atoi
                    free                       ;Libère espace alloué par getsm.

                    pop    bx
                    pop    di
                    pop    es
                    ret
_geti       endp

; InitGame-  Initialise les variables global que ce jeu utilise.

InitGame            proc
                    mov    CurVel, InitialVelocity
                    mov    CurDist, InitialDistance
                    mov    FuelLeft, InitialFuel
                    mov    Dist, 0
                    ret
InitGame            endp
```

```
; DispStatus-       Affiche les informations important pour chaque cycle
; du jeu (un cycle est une seconde).

DispStatus   proc
                    printf
                    byte   cr,lf
                    byte   "Distance de la surface: %5d",cr,lf
                    byte   "Vitesse actuelle: %5d",cr,lf
                    byte   "Carburant restant: %5d",cr,lf
                    byte   lf
                    byte   "Distance parcourue dans la seconde: %d",cr,lf
                    byte   lf,0
                    dword CurDist, CurVel, FuelLeft, Dist
                    ret
DispStatus   endp

; GetFuel-   Lit une valeur en nombre entier représentant la
;                   quantité de combustible à brûler fournie par
;                   l'utilisateur et la contrôle pour voir si cette valeur
;                   est raisonnable. Une valeur raisonnable doit :
;
;                   * Être un nombre valide (GETI gère ceci).
;                   * Être supérieur ou égal à zéro (pas de quantités de
;                    carburant négatives, GETI gère ceci).
;                   * Être inférieure à MaxFuelBurn (sinon vous avez une
;                    explosion, pas un jet).
;                   * Être inférieure au carburant restant dans le Module ;
       Lunaire.

GetFuel             proc
                    push   ax

; Structure Loop..endloop qui lit un entier en entrée et se termine
; si l'entrée est raisonnable. Elle affiche un message et se répète si
; l'entrée n'est pas raisonnable.
;
; loop
;      get fuel;
;      if (fuel < MaxFuelBurn) then break;
;      print error message.
; endloop
;
; if (fuel > FuelLeft) then
;
;      fuel = fuelleft;
;      print appropriate message.
;
; endif

GetFuelLp:   print
                    byte   "Entre la quantité de fuel à brûler: ",0
                    geti
                    cmp    ax, MaxFuelBurn
                    jbe    GoodFuel

                    print
                    byte   "La quantité specifiée excède le "
                    byte   "débit du moteur,", cr, lf
                    byte   "S.V.P., entre une valeur moindre",cr,lf,lf,0
                    jmp    GetFuelLp

GoodFuel:           mov    Fuel, ax
                    cmp    ax, FuelLeft
                    jbe    HasEnough
                    printf
```

```
                    byte   "Il ne te reste que %d unités de fuel.",cr,lf
                    byte   "Le Module Lunaire les brûlera au lieu de %d"
                    byte   cr,lf,0
                    dword FuelLeft, Fuel

                    mov    ax, FuelLeft
                    mov    Fuel, ax

HasEnough:   mov    ax, FuelLeft
                    sub    ax, Fuel
                    mov    FuelLeft, ax
                    pop    ax
                    ret
GetFuel             endp

; ComputeStatus-          Cette routine calcule la nouvelle vitesse et la
;                         nouvelle distance basées sur la distance
;                         actuelle, la vitesse actuelle, le carburant
;                         brûlé et la pesanteur de la lune. Cette routine
;                         est appelée à chaque "seconde" de temps de vol
;                         Ceci simplifie les équations suivantes puisque la
;                         valeur de T est toujours un
;
; note :
;
; Distance Parcourue = Acc*T*T/2 + Vel*T (note: T=1, on l'ignore).
; Acc = MoonsGravity + Fuel * AccPerUnitFuel
;
; Nouvelle Vitesse = Acc*T + Vitesse Précédente
;
; Ce code devrait en fait faire la moyenne de ces valeurs au cours
; de la période de temps d'une seconde, mais la simulation est de
; toutes façons grossière, il y a pas besoin s'en occuper vraiment.

ComputeStatus       proc
                    push   ax
                    push   bx
                    push   dx

; D'abord, calculer la valeur d'accélération basée sur le carburant
; brûlé pendant cette seconde
; (Acc = Gravité de la Lune + Fuel * AccPerUnitFuel).

                    mov    ax, Fuel ;Compute
                    mov    dx, AccPerUnitFuel  ; Fuel*AccPerUnitFuel
                    imul   dx

                    add    ax, MoonsGravity    ;Ajoute gravité de la lune.
                    mov    bx, ax              ;Sauve valeur Acc.

; Maintenant, calcule la nouvelle vitesse (V=AT+V)

                    add    ax, CurVel          ;Calcule nouvelle vitesse
                    mov    CurVel, ax

; Ensuite, calcule la distance parcourue (D = 1/2 * A * T^2 + VT +D)

                    sar    bx, 1        ;Acc/2
                    add    ax, bx              ;Acc/2 + V (T=1!)
                    mov    Dist, ax            ;Distance Parcourue.
                    neg    ax
                    add    CurDist, ax  ;Nouvelle distance.

                    pop    dx
                    pop    bx
                    pop    ax
```

```
                    ret
ComputeStatus       endp

; GetYorN-   Lit une réponse yes ou no de l'utilisateur (Y, y, N,
;                   ou n). Renvoie le caractère lu dans le registre AL
;                   (Y ou N, converti en majuscule si besoin).

GetYorN             proc
                    getc
                    ToUpper
                    cmp    al, 'Y'
                    je     GotIt
                    cmp    al, 'N'
                    jne    GetYorN
GotIt:              ret
GetYorN             endp

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

MoonLoop:           print
                    byte   cr,lf,lf
                    byte   "Bienvenue au jeu Moon Lander.",cr,lf,lf
                    byte   "Tu dois manoeuvrer le vaisseau pour te poser "
                    byte   "à moins de 10 pieds/sec",cr,lf
                    byte   "pour un alunissage en douceur.",cr,lf,lf,0

                    call   InitGame

; La boucle suivante se répète tant que la distance à la surface est
; plus grande que zéro.

WhileStillUp:       mov    ax, CurDist
                    cmp    ax, 0
                    jle    Landed

                    call   DispStatus
                    call   GetFuel
                    call   ComputeStatus
                    jmp    WhileStillUp

Landed:             cmp    CurVel, 10
                    jle    SoftLanding

                    printf
                    byte   "Ta vitesse actuelle est %d.",cr,lf
                    byte   "C'était un petit peu trop rapide. Mais,"
                    byte   "comme lot de consolation,",cr,lf
                    byte   "nous donnerons ton nom au nouveau cratère "
                    byte   "que tu viens de créer.",cr,lf,0
                    dword CurVel

                    jmp    TryAgain

SoftLanding: printf
                    byte   "Bravo! Tu as posé le Module Lunaire intact à "
                    byte   "%d pieds/sec.",cr,lf
                    byte   "Il te reste %d unités de carburant.",cr,lf
                    byte   "Bon travail !",cr,lf,0
                    dword   CurVel, FuelLeft

TryAgain:           print
                    byte   "Veux-tu réessayer (Y/N)? ",0
```

```
                    call   GetYorN
                    cmp    al, 'Y'
                    je     MoonLoop

                    print
                    byte   cr,lf
                    byte   "Merci d'avoir joué! Reviens bientôt "
                    byte   "sur la lune !"
                    byte   cr,lf,lf,0

Quit:         ExitPgm                    ;Macro DOS pour quitter 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
```

10.12 Exercices de laboratoire

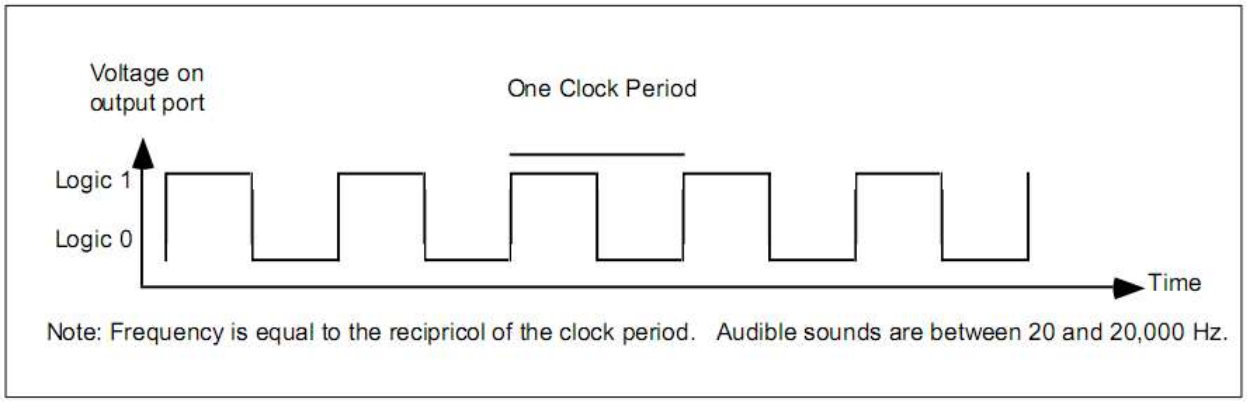

Figure 10.2 Une Onde Sonore Audible : Relation entre Période et Fréquence

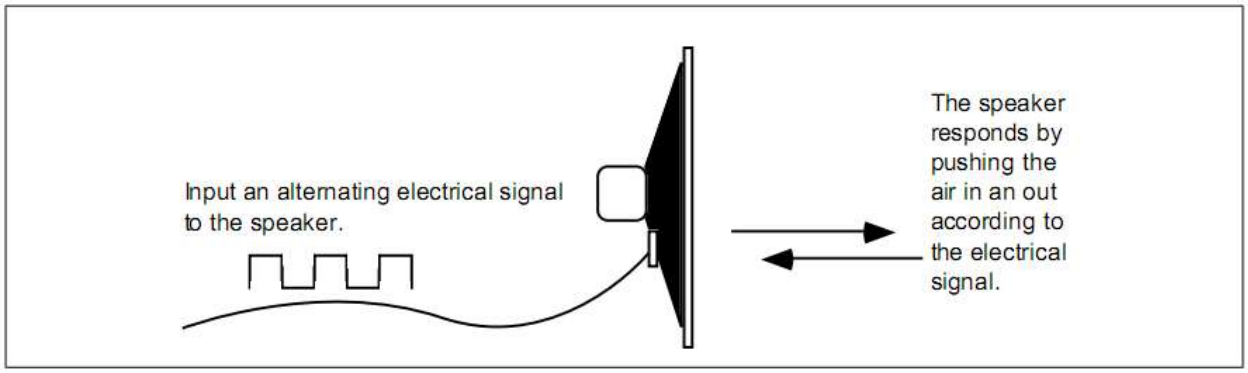

Figure 10.3 Un Haut-Parleur

Dans ces exercices de laboratoire, vous programmerez le chipset du timer sur le PC pour produire des tonalités musicales. Vous apprendrez comment le PC produit des sons et comment vous pouvez utiliser cette capacité pour coder et jouer de la musique.

10.12.1 Physique du son

Les sons que vous entendez sont le résultat de la vibration des molécules d'air. Quand les molécules d'air vibrent rapidement alternativement entre 20 et 20.000 fois par seconde, nous interprétons ceci comme une sorte de son. Un *haut-parleur* (voir le schéma 10.3) est un appareil qui fait vibrer l'air en réponse à un signal électrique. C'est-à-dire, il convertit un signal électrique qui alterne entre 20 et 20.000 périodes par seconde (Hz) en une tonalité audible. Faire alterner un signal est très facile sur un ordinateur, tout que vous avez à faire est d'appliquer un un logique à un port de sortie pendant un certain temps et ensuite écrire un zéro logique au port de sortie pendant une courte période. Répétez ensuite ceci sans arrêt. Un graphe de cette activité dans le temps apparaît sur la Figure 10.2.

Bien que beaucoup d'humains soient capables d'entendre des tonalités dans la plage 20-20Khz, le haut-parleur du PC n'est pas capable de reproduire fidèlement les tonalités dans cette plage. Il fonctionne très bien pour des sons dans la plage 100-10Khz, mais le volume baisse nettement en dehors de cette plage. Heureusement, ce laboratoire nécessite seulement des fréquences dans la plage de 110-2.000 hertz ; tout à fait en conformité avec les possibilités du haut-parleur du PC.

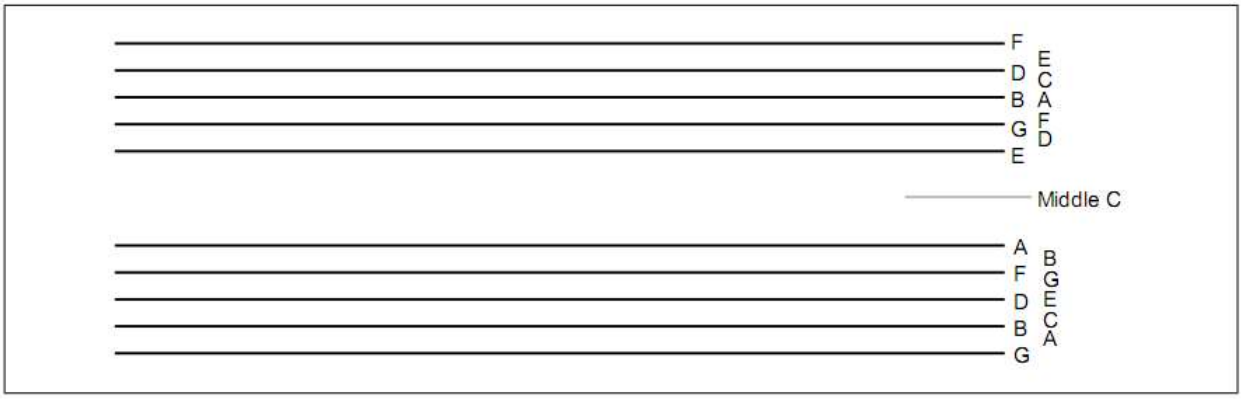

Figure 10.4 Une Portée de musique

10.12.2 Les Bases de la Musique

Dans ce laboratoire, vous utiliserez le chipset du timer et le haut-parleur intégré du PC pour produire des tonalités musicales. Pour produire la vraie musique, au lieu de tonalités désagréables, il faut une connaissance minimale de la théorie de la musique. Cette section apporte une introduction très brève à la notation musicale. Ceci vous aidera quand vous essayerez de convertir de la musique en notation standard sous une forme que l'ordinateur peut utiliser.

La musique occidentale tend à utiliser la notation basée sur les lettres alphabétiques A... G (La à Sol en France). Il y a un total de 12 notes indiquées La, La#, Si, Do, Do#, Ré, Ré#, Mi, Fa, Fa#, Sol, et Sol#[10]. Sur un instrument de musique classique, ces 12 notes se répètent à plusieurs reprises. Par exemple, un piano ordinaire peut avoir six répétitions de ces 12 notes. Chaque répétition est une *octave*. Une octave est juste une collection de 12 notes, elle n'a pas besoin nécessairement de commencer par La (A), en effet, la plupart des pianos commencent par Do (C). Bien qu'il y ait, techniquement, environ 12 octaves dans la zone d'audition normale des adultes, peu de musiques utilisent plus de quatre ou cinq octaves. Dans le laboratoire, vous travaillerez sur quatre octaves.

La musique écrite utilise classiquement deux *portées*. Une portée est un ensemble de cinq lignes parallèles. La portée supérieure s'appelle souvent la portée *haute* et la portée inférieure s'appelle souvent la portée *basse*. Un exemple apparaît sur la Figure 10.4.

Une note musicale, comme la notation à côté des portées ci-dessus l'indique, apparaît soit sur les lignes des portées soit sur les espaces entre les lignes. La position des notes sur la portée détermine quelle note jouer, la *forme* de la note détermine sa durée. Il y a les *rondes*, les *blanches*, les *noires*, les *croches*, les *demi-croches*, et les *quart-de-croches*[11]. La durée d'une note est indiquée relativement à une autre. Ainsi la blanche joue la moitié de la période d'une ronde, une noire joue la moitié de la période d'un blanche (un quart de la période d'une ronde), etc... Dans la

[10] Les notes avec "#" (prononcé dièse) correspondent aux touches noires du piano. Les autres notes correspondent aux touches blanches du piano. Notez que la notation musicale occidentale décrit également des bémols en plus des dièses. La# est égal à Si*b* (*b* signifie bémol), Do# correspond à Ré*b*, etc... Techniquement, Si est équivalent à Do*b* et Do est équivalent à Si# mais vous verrez rarement des musiciens se référer à ces notes de cette façon.

[11] La seule raison pour laquelle il n'y a pas de notes plus courtes est parce qu'il serait difficile de jouer une note qui est 1/64th de la longueur d'une autre.

plupart des passages musicaux, la noire sert généralement de base à la mesure du temps. Si le *tempo* d'un morceau particulier est de 100 battements par seconde ceci signifie que vous jouez 100 noires par seconde.

La durée d'une note est déterminée par sa forme comme représenté sur la Figure 10.5.

En plus des notes elles-mêmes, il y a souvent de brèves pauses dans un passage musical quand aucune note n'est jouée. Ces pauses sont connues sous le nom de silences. Puisqu'ils nont rien d'audible, seule leur durée importe. La durée des divers silences est identique aux notes normales ; il y a des pauses, des demi -pauses, des silences, etc... Les symboles pour ces silences apparaissent ici.

Ce n'est qu'une brève introduction à la notation musicale. Tout juste suffisante pour quelqu'un sans formation musicale pour qu'il convertisse une partition musicale sous la forme appropriée à un programme informatique. Si vous êtes intéressé par plus d'information sur la notation musicale, les bibliothèques sont de bonnes sources d'information sur la théorie de musique.

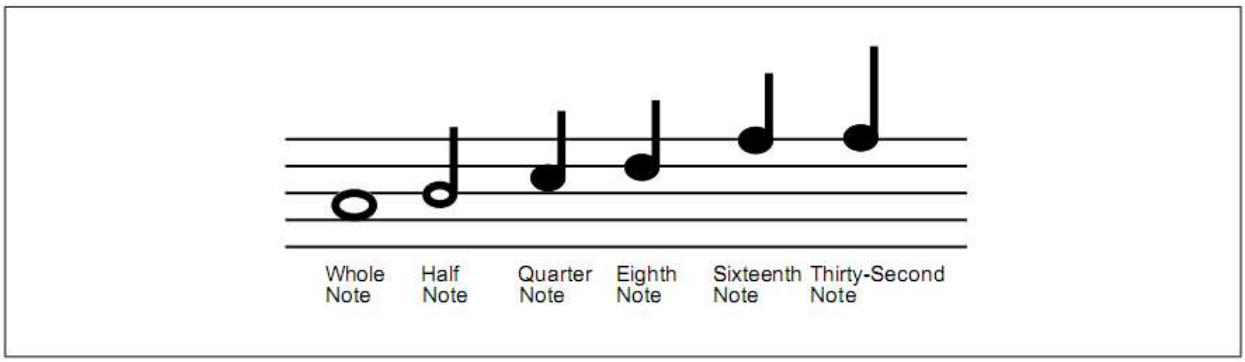

Figure 10.5 Durée des notes

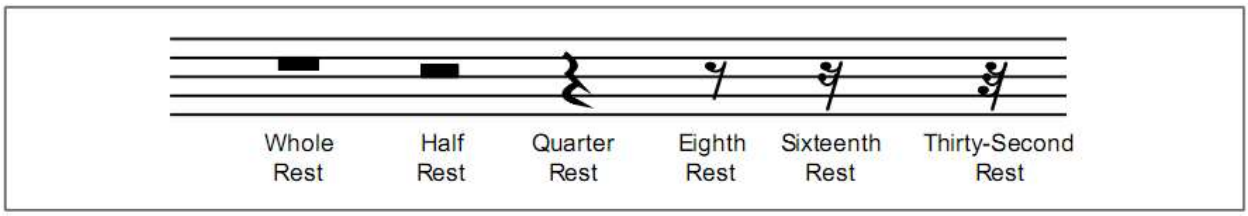

Figure 10.6 Durée des silences

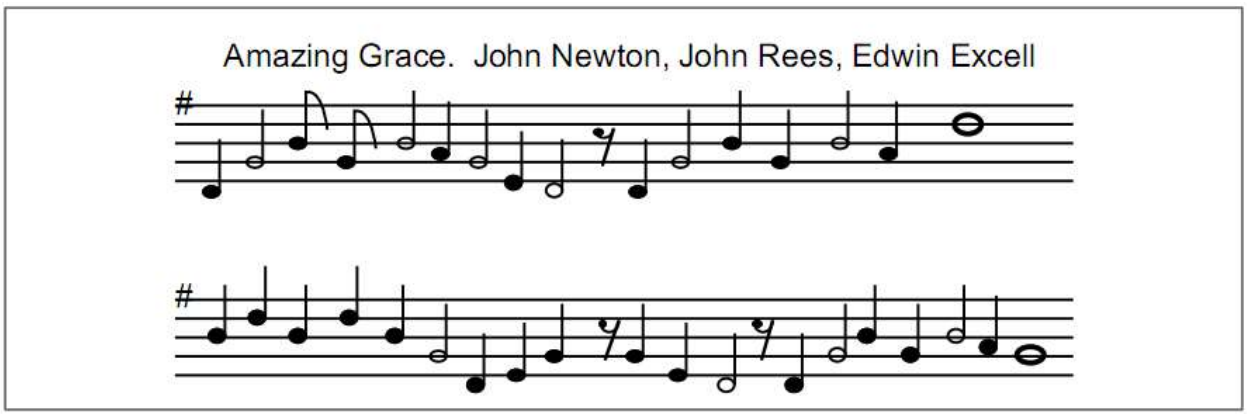

Figure 10.7 Amazing Grace

La Figure 10.7 fournit une adaptation de l'hymne "Amazing Grace". Il y a deux choses à noter ici. D'abord, il n'y a pas de portée basse, seulement deux portées hautes. En second lieu, le symbole dièse sur la ligne de Fa indique que cette chanson est jouée en "Sol-Majeur" et que toutes les notes Fa devraient être Fa#. Il n'y a aucune note Fa dans ce morceau, mais cela ne fait rien[12].

10.12.3 La Physique de la Musique

[112] Dans la version complète du morceau, il y a des notes fa sur la portée principale.

Chaque note musicale correspond à une fréquence unique. Le La au-dessus du Do medium est généralement de 440 hertz (on appelle cela la tonallité de concert puisque c'est la fréquence sur laquelle les orchestres s'accordent). Le La une octave au-dessous de celui-ci est à 220 hertz, le La au-dessus à 880Hz. En général, pour obtenir le La suivant en hauteur, vous doublez la fréquence courante, pour obtenir le La précédent, vous divisez par deux la fréquence courante. Pour obtenir les notes restantes, vous multipliez la fréquence de La avec un multiple de la douzième racine de deux. Par exemple, pour obtenir La#, vous prendriez la fréquence de La et la multiplieriez par la douzième racine de deux. La répétition de cette opération donne les fréquences (arropndies) suivantes pour quatre octaves séparées :

Note	Fréquence	Note	Fréquence	Note	Fréquence	Note	Fréquence
La 0	110	La 1	220	La 2	440	La 3	880
La# 0	117	La# 1	233	La# 2	466	La# 3	932
Si 0	123	Si 1	247	Si 2	494	Si 3	988
Do 0	131	Do 1	262	Do 2	523	Do 3	1047
Do# 0	139	Do# 1	277	Do# 2	554	Do# 3	1109
Ré 0	147	Ré 1	294	Ré 2	587	Ré 3	1175
Ré# 0	156	Ré# 1	311	Ré# 2	622	Ré# 3	1245
Mi 0	165	Mi 1	330	Mi 2	659	Mi 3	1319
Fa 0	175	Fa 1	349	Fa 2	698	Fa 3	1397
Fa# 0	185	Fa# 1	370	Fa# 2	740	Fa# 3	1480
Sol 0	196	Sol 1	392	Sol 2	784	Sol 3	1568
Sol# 0	208	Sol# 1	415	Sol# 2	831	Sol# 3	1661

Notes : Le nombre après chaque note détermine son octave. Dans le diagramme ci-dessus, le Do medium est Do 1.

Vous pouvez produire des notes additionnelles en divisant par deux ou en doublant les notes ci-dessus. Par l'exemple, si vous avez vraiment besoin de La (-1) (l'octave au-dessous de La 0 ci-dessus), diviser la fréquence de La 0 par deux donne 55Hz. De même, si vous voulez Mi 4, vous pouvez obtenir ceci en doublant Mi 3 qui donne 2638 hertz. Gardez à l'esprit que les fréquences ci-dessus ne sont pas exactes. Elles sont arrondies au nombre entier le plus proche parce que nous aurons besoin des fréquences en nombre entier dans ce laboratoire.

10.12.4 Le chipset de timer 8253/8254

Les PCS contiennent un circuit intégré spécial qui produit un signal périodique. Ce chipset (compatible Intel 8253 ou 8254, selon votre ordinateur spécifique[13]) contient trois circuits 16 bits compteur/timer différents. Le PC utilise un de ces timers pour produire l'horloge temps réel 1/18.2 secondes citée précédemment. Il utilise le second de ces timers pour contrôler le rafraîchissement de la DMA en mémoire centrale[14]. Le troisième circuit de timer sur ce chipset est relié au haut-parleur du PC. Le PC utilise ce timer pour produire des signaux sonores, des tons et d'autres sons. Le timer RTC nous intérêssera dans un chapitre ultérieur. Le timer de DMA, s'il est présent sur votre PC, n'est pas quelque chose que vous devriez bidouiller. Le troisième timer, relié au haut-parleur, est le sujet de cette section.

10.12.5 Programmation du chipset du timer pour produire des notes de musique

[113] La plupart des ordinateurs modernes n'ont pas un vrai chipset 8253 ou 8254. A la place, il y a un dispositif compatible inclu dans un autre chipset VLSI de la carte mère.

[114] Beaucoup de systèmes informatiques modernes n'utilisent pas ce timer à cette fin et, en conséquence, n'incluent pas le deuxième timer dans leur jeu de puces.

Comme on l'a mentionné précédemment, un des canaux sur le chipset du timer d'intervalles programmable du PC (PIT : Programmable Interval Timer) est relié au haut-parleur du PC. Pour produire une note de musique, nous devons programmer ce chipset de timer pour produire la fréquence de la note désirée et ensuite activer le haut-parleur. Une fois que vous avez initialisé le timer et le haut-parleur de cette façon, le PC produira sans interruption la note indiquée jusqu'à ce que vous désactiviez le haut-parleur.

Pour activer le haut-parleur vous devez placer les bits zéro et un du "port B" sur le chipset d'Interface Programmable de Périphérique 8255 du PC (PPI : Programmable Peripheral Interface). Le port B de la PPI est une unité d'E/S de huit bits située à l'adresse d'E/S 61h. Vous devez utiliser l'instruction **in** pour lire ce port et l'instruction **out** pour écrire des données dans ce port. Vous devez préserver tous les autres bits à cette adresse d'E/S. Si vous modifiez tout autre bit, vous ferez probablement dysfonctionner le PC, ou le faire rebooter. Le code suivant montre comment mettre à un les bits zéro et un sans affecter d'autre bits sur le port :

```
in      al, PPI_B           ;PPI_B equate 61h
or      al, 3         ;Met à un les bits zéro et un
out     PPI_B, al
```

Puisque l'adresse du port PPI_B est inférieure à 100h nous pouvons accéder à ce port directement, nous n'avons pas besoin de charger son adresse de port dans **dx** et accéder au port indirectement par **dx**.

Pour désactiver le haut-parleur, vous devez écrire des zéros aux bits zéro et un de **PPI_B**. Le code est semblable à ce qui précède sauf que vous forcez le bit à zéro au lieu de un.

Manipuler les bits zéro et un du port **PPI_B** vous a permis de démarrer et arrêter le haut-parleur. Il ne vous permet pas d'ajuster la fréquence de la tonalité que le haut-parleur produit. Pour ce faire, vous devez programmer le PIT aux adresses d'E/S 42h et 43h. Pour changer la fréquence appliquée au haut-parleur vous devez d'abord écrire la valeur 0B6h au port d'"E/S 43h (le *mot de contrôle* du PIT) et ensuite vous devez écrire un diviseur de fréquence sur 16 bits sur le port 42h (canal deux du timer). Puisque le port est seulement un port de huit bits, vous devez écrire les données en utilisant deux instructions OUT successives à la même adresse d'E/S. Le premier byte que vous écrivez est le byte de L.O. du diviseur, le deuxième byte vous écrivez est le byte de H.O.

Pour calculer la valeur de diviseur, vous devez utiliser la formule suivante :

$$\frac{1193180}{Frequency} = Divisor$$

Par exemple, le diviseur pour La audessus du Do medium (440 hertz) est 1.193.180/440, soit 2.712 (arrondi au nombre entier le plus proche). Pour programmer le PIT pour jouer cette note vous exécuteriez le code suivant :

```
mov    al, 0B6h            ;Code mot de contrôle.
out    PIT_CW, al    ;Ecrit mot de contrôle (port 43h)
mov    al, 98h             ;2712 est 0A98h.
out    PIT_Ch2, al   ;Ecrit byte L.O. (port 42h).
mov    al, 0ah
out    PIT_Ch2, al   ;Ecrit byte H.O. (port 42h).
```

Supposant que vous avez activé le haut-parleur, le code ci-dessus produira la note La jusqu'à ce que vous désactiviez le haut-parleur ou reprogrammiez le PIT avec un diviseur différent.

10.12.6 Assemblons le Tout

Pour créer de la *musique*, vous devrez activer le haut-parleur, programmer le PIT, et ensuite attendre une certaine période tandis que la note joue. À la fin de cette période, vous devez reprogrammer le PIT et attendre tandis que la note suivante joue. Si vous rencontrez un silence, vous devez désactiver le haut-parleur pour l'intervalle de temps donné. Le point clé est cet *intervalle de temps*. Si vous reprogrammez seulement les chipsetx PPI et de PIT aux vitesses du microprocesseur, votre chanson sera passée en quelques micro-secondes. Beaucoup trop vite pour qu'on entende quoi que ce soit. Par conséquent, nous devons utiliser un retard, tel que le code de retard logiciel présenté plus tôt, pour nous permettre d'entendre nos notes.

Le tempo raisonnable est entre 80 et 120 noires par minute. Ceci veut dire que vous devrez appeler la routine Delay18 entre 9 et 14 fois pour chaque noire. Un ensemble raisonnable d'itérations est

- trois fois pour les doubles-croches,
- six fois pour les croches,
- douze fois pour les noires,
- vingt-quatre fois pour les blanches
- et quarante-huit fois pour les rondes.

Naturellement, vous pouvez ajuster ces cadences selon vos goûts pour que votre musique sonne mieux. Le paramètre important est le rapport entre les différents notes et les silences, pas le temps réel.

Puisqu'un morceau de musique contient beaucoup, beaucoup de notes individuelles, cela n'a pas de sens de reprogrammer les chipsets PIT et PPI individuellement pour chaque note. Au lieu de cela, vous devriez écrire une procédure à laquelle vous passez un diviseur et une valeur compte-à-rebours. Cette procédure jouera alors cette note pendant le temps indiqué et retournera ensuite. En supposant que vous appelez cette procédure *PlayNote* et qu'elle attend le diviseur dans **ax** et la durée (nombre de fois où on appelle Delay18) dans **cx**, vous pourriez utiliser la macro suivante pour créer facilement des chansons dans vos programmes :

```
Note                macro  divisor, duration
                    mov    ax, divisor
                    mov    cx, duration
                    call PlayNote
                    endm
```

La macro suivant vous laisse facilement insérer un silence dans votre musique :

```
Rest                macro  Duration
                    local  LoopLbl
                    mov    cx, Duration
LoopLbl:            call   Delay18
                    loop   LoopLbl
                    endm
```

Maintenant, vous pouvez jouer des notes en chaînant simplement ensemble une liste de ces macros avec les paramètres appropriés.

Le seul problème avec cette approche est qu'il est difficile de créer des chansons si vous devez constamment fournir des valeurs de diviseur. Vous trouverez la création de musique beaucoup plus simple si vous pouvez indiquer la note, octave, et durée plutôt qu'un diviseur et une durée. C'est très facile à réaliser. Créez simplement une *table de référence* utilisant la définition suivante :

```
Divisors : array [Note, Sharp, Octave] of word;
```

Où Note est "A";.."G", Sharp (dièse) est vrai ou faux (1 ou 0), et Octave est 0..3. Chaque entrée dans la table contiendrait le diviseur pour cette note particulière.

10.12.7 L'Exercice Amazing Grace

Le programme Ex10_1.asm sur le CD-ROM d'accompagnement est un programme complètement fonctionnel qui joue l'air "Amazing Grace". Chargez ce programme et exécutez-le.

Pour votre rapport de laboratoire : le fichier Ex10_1.asm utilise une macro "**Note**" qui est très semblable à celle qui apparaît dans la section précédente. Quelle est la différence entre la macro **Note** d'Ex10_1's et celle de la section précédente ? Quels changements ont été faits dans **PlayNote** afin d'adapter cette différence ?

Le programme Ex10_1.asm utilise le *code en ligne directe* (aucune boucle ou décision) pour jouer son air. Récrivez le corps principal de la boucle pour utiliser une paire de tables pour alimenter en données les macros **Note** et **Rest**. Une table devrait contenir une liste des valeurs de fréquence (utilisez -1 pour un silence), l'autre table devrait contenir des valeurs de durée. Mettez les deux tables dans le segment de données et initialisez-les avec les valeurs pour la chanson Amazing grace. La boucle devrait chercher une paire de valeurs, une dans chacune des tables et appeler les macros **Note** ou **Rest** comme approprié. Quand la boucle rencontre une valeur de fréquence de zéro elle devrait se terminer. **Note :** vous devez appeler la macro **Rest** à la fin de l'air afin de fermer le haut-parleur.

Pour votre rapport de laboratoire : faites les changements au programme, documentez-les, et incluez le listing imprimé du nouveau programme dans votre rapport de laboratoire.

10.13 Projets de Programmation

1) Ecrivez un programme pour transposer deux tableaux 4x4. L'algorithme pour transposer les tableaux est

```
for i := 0 to 3 do
       for j := 0 to 3 do begin

              temp := A [i,j];
              A [i,j] := B [j,i];
              B [j,i] := temp;
       end;
```

Ecrivez un programme principal qui appelle une procédure de transposition. Le programme principal devrait lire les valeurs du tableau A données par l'utilisateur et imprimer les tableaux A et B après calcul de la transposition de A et placement du résultat dans B.

2) Créez un programme pour jouer de la musique qui est fournie comme chaîne au programme. Les notes à jouer devraient se composer d'une chaîne de caractères ASCII terminés par un byte contenant la valeur zéro. Chaque note devrait prendre la forme suivante :

```
(Note)(Octave)(Duration)
```

où "Note" est A..G (majuscules ou minuscules), "Octave" est 0..3 et "Durée" est 1..8. "1" correspond à une croche, "2" correspond à une noire, "4" correspond à une blanche et "8" correspond à une ronde.

Les repos se composent d'un point d'exclamation suivi d'une valeur "Durée".

Votre programme devrait ignorer toutes les espaces apparaissant dans la chaîne.

Le morceau suivant en exemple est la chanson "Amazing Grace" présentée plus tôt.

```
Music  byte   "d12 g14 b11 g11 b14 a12 g14 e12 d13 !1 d12 "
              byte   "g14 b11 g11 b14 a12 d28"
              byte   "b12 d23 b11 d21 b11 g14 d12 e13 g12 e11 "
              byte   "d13 !1 d12 g14 b11 g11 b14 a12 g18"
              byte   0
```

Ecrivez un programme pour jouer n'importe quelle chanson apparaissant sous forme de chaîne comme la chaîne ci-dessus. En utilisant la musique obtenue à partir d'une autre source, soumettez votre programme qui joue l'autre chanson.

3) Une *chaîne de caractères C* est une séquence des caractères qui se termine par un byte contenant zéro. Quelques routines de chaîne de caractères courantes incluent le calcul de la longueur d'une chaîne de caractères (en comptant tous les caractères de la chaîne jusqu'au byte zéro, celui-ci non compris), la comparaison de deux chaînes (en comparant les caractères correspondants dans deux chaînes, caractère par caractère jusqu'à ce que vous rencontriez un byte zéro ou deux caractères qui ne sont pas identiques) et copie d'une chaîne dans une autre (en copiant les caractères d'une chaîne dans les positions correspondantes de l'autre jusqu'à ce que vous rencontriez le byte zéro). Écrivez un programme qui lit deux chaînes depuis l'utilisateur, calcule la longueur de la première de celles-ci, compare les deux chaînes et ensuite copie la première chaîne sur la seconde. Tenez compte d'un maximum de 128 caractères (byte zéro y compris) dans vos chaînes. Note : n'utilisez pas les routines de chaîne de la Bibliothèque Standard pour ce projet.

4) Modifiez le jeu moon lander présenté dans la section Echantillon de Programme de ce chapitre (moon.asm sur le CD-ROM d'accompagnement, voir également "Echantillon de Programme" à la section 10.11) pour permettre à l'utilisateur d'indiquer les valeurs initiales de la vitesse initiale, de distance de départ de la surface et de carburant. Vérifiez que les valeurs sont raisonnables avant de permettre au jeu de continuer.

10.14 Résumé

Ce chapitre a traité l'implémentation de différentes structures de contrôle dans des programmes en assembleur comprenant des instructions conditionnelles (instructions **if..then..else** et case), des machines d'état, et des itérations (boucles, incluant **while**, **repeat..until** (**do/while**), **loop..endloop** et **for**). Bien que l'assembleur vous donne de la flexibilité pour créer des structures de contrôle selon vos desiderata, en abuser produit très souvent des

programmes difficiles à lire et à comprendre. A moins que la situation nécessite absolument quelque chose de différent, vous devriez essayer de modeler, autant que possible, vos structures de contrôle en assembleur sur celles des langages de haut niveau.

La structure de contrôle la plus courante qu'on trouve dans les programmes des langages de haut niveau est l'instruction IF.THEN..ELSE. Vous pouvez facilement synthétiser les instructions **if..then** et **if..then..else** en assembleur en utilisant l'instruction **cmp**, les sauts conditionnels et l'instruction **jmp**. Pour voir comment convertir des instructions HLL **if..then..else** en assembleur, se reporter à

- "Séquences IF..THEN..ELSE" à la section 10.2

Une autre instruction conditionnelle populaire est l'instruction **case** (**switch**). L'instruction **case** fournit une manière efficace de transférer le contrôle à une parmi nombreuses instructions différentes selon la valeur d'une expression donnée. Alors qu'il y a beaucoup de manières d'implémenter l'instruction **case** en assembleur, la manière la plus courante est d'utiliser une *table de saut.* Pour des instructions **case** avec des valeurs contiguës, c'est probablement la meilleure implémentation. Pour les instructions **case** qui ont des valeurs très espacées, non-contiguës, une imlplémentation avec **if..then..else** ou quelque autre technique est probablement meilleure. Pour des détails, voir

- "Instructions CASE" à la section 10.3

Les machines d'état fournissent une solution utile pour certaines situations de programmation. Une section du code qui implémente une machine d'état maintient une histoire de l'exécution antérieure dans une variable d'état. L'exécution ultérieure du code reprend dans un "état" éventuellement différent selon l'exécution antérieure. Les sauts indirects fournissent un mécanisme efficace pour implémenter des machines d'état en assembleur. Ce chapitre fournit une brève introduction aux machines d'état. Pour voir comment implémenter une machine d'état avec un saut indirect, voir

- "Machines d'état et sauts indirects" à la section 10.4

L'assembleur fournit des structures primitives très puissantes pour construire une grande variété de structures de contrôle. Bien que ce chapitre se concentre sur la simulation des constructions des langages de haut niveau, vous pouvez construire n'importe quelle structure de contrôle élaborée selon vos désirs à partir de l'instruction 80x86 **cmp** et des branchements conditionnels. Malheureusement, le résultat peut être très difficile à comprendre, particulièrement par quelqu'un autre que l'auteur original. Bien que l'assembleur vous donne la liberté de faire ce que vous voulez, un programmeur aguerri fera preuve de rigueur et choisira seulement des flux de contrôle qui sont faciles à lire et à comprendre ; il n'optera jamais pour du code compexe sauf si absolument nécessaire. Pour une description et des conseils additionnels, voir

- "Code spaghetti" à la section 10.5

L'itération est l'une des trois composantes basiques des langages de programmation établis autour des machines de Von Neumann[15]. Les structures de contrôle boucle fournissent le mécanisme basique d'itération dans la plupart des langages de haut niveau. L'assembleur ne fournit aucune instruction de base pour les boucles. Même l'instruction 80x86 **loop** n'est pas vraiment une boucle, elle ne fait que décrémenter, comparer, et faire un branchement. Néanmoins, il est très facile de synthétiser les structures de contrôle de boucle courantes en assembleur. Les sections suivantes décrivent comment construire les structures de contrôle de boucle HLL en assembleur :

- "Boucles" à la section 10.6
- "Boucles while" à la section 10.6.1
- "Boucles repeat..until " à la section 10.6.2
- "Boucles LOOP..ENDLOOP " à la section 10.6.3
- "Boucles FOR" à la section 10.6.4

Les boucles de programme consomment souvent la majeure partie du temps de CPU dans un programme classique. Par conséquent, si vous voulez améliorer les performances de vos programmes, les boucles sont le premier emplacement que vous devez regarder. Ce chapitre fournit plusieurs suggestions pour aider à améliorer les performances de certains types de boucles dans des programmes en assembleur. Bien qu'elles ne fournissent pas

[115] Les autres deux étant l'exécution conditionnelle et la séquence

un guide complet d'optimisation, les sections suivantes fournissent des techniques courantes employées par des compilateurs et des programmeurs expérimentés en assembleur :

- "Les boucles et l'utilisation des registres" à la section 10.7
- "Améliorations des performances" à la section 10.8
- "Reporter la condition de terminaison à la fin d'une boucle" à la section 10.8.1
- "Exécuter la boucle à l'envers" à la section 10.8.2
- "Calculs invariables dans une boucle" à la section 10.8.3
- "Démonter une boucle" à la section 10.8.4
- "Variables d'induction" à la section 10.8.5
- "Autres améliorations de performances" à la section 10.8.6

10.15 Questions

1) Convertissez les instructions Pascal suivantes en assembleur : (supposez que toutes les variables sont des nombres entiers signés de deux bytes)

   a) IF (X = Y) THEN A := B;

   b) IF (X <= Y) THEN X := X + 1 ELSE Y := Y - 1;

   c) IF NOT ((X <= Y) AND (Z <> T)) THEN Z := T ELSE X := T;

   d) d) IF (X=0) AND ((Y-2) > 1) THEN Y := Y - 1;

2) Convertissez l'instruction CASE suivante en assembleur :

```
CASE I OF
       0 : I := 5;
       1 : J := J+1;
       2 : K := I+J;
       3 : K := I-J;
       Otherwise I := 0;
END;
```

3) Quelle méthode d'implémentation de l'instruction CASE (table de saut ou forme IF) produit le moins de code (y compris la table de saut, si on l'utilise) pour les instructions CASE suivantes ?

a)
```
CASE I OF
              0:stmt;
              100:stmt;
              1000:stmt;
       END;
```

b)
```
CASE I OF
              0:stmt;
              1:stmt;
              2:stmt;
              3:stmt;
              4:stmt;
       END;
```

4) Pour la question trois, quelle forme produit le code le plus rapide ?

5) Implémentez les instructions CASE du problème trois en langage en utilisant l'assembleur 80x86.

6) De quels trois composantes consiste une boucle ?

7) Quelle est la différence principale entre les boucles WHILE, REPEAT..UNTIL et LOOP..END-LOOP ?

8) Qu'est-ce qu'une variable de contrôle de boucle ?

9) Convertissez les boucles WHILE suivantes en assembleur : (note : n'optimisez pas ces boucles, collez exactement au format de la boucle WHILE)

a)
```
I := 0 ;
WHILE (I < 100) DO I := I + 1;
```

b)
```
CH := '';
    WHILE (CH <> '.') DO BEGIN
            CH := GETC;
            PUTC(CH);
    END;
```

10) Convertissez les boucles REPEAT..UNTIL suivantes en assembleur : (collez exactement au format de boucle REPEAT..UNTIL).

a)
```
I := 0;
    REPEAT
            I := I + 1;
    UNTIL I >= 100;
```

b)
```
REPEAT
            CH := GETC;
            PUTC(CH);
    UNTIL CH = '.';
```

11) Convertissez les boucles LOOP..ENDLOOP suivantes en assembleur : (collez exactement au format de LOOP..ENDLOOP)

a)
```
I := 0;     LOOP
            I := I + 1;   IF I = 100 THEN   BREAK;
    ENDLOOP;
```

b)
```
LOOP
            CH := GETC;         IF CH = '.' THEN   BREAK;   PUTC(CH);
 ENDLOOP ;
```

12) Quelles sont les différences, le cas échéant, entre les boucles dans les problèmes 4, 5, et 6 ? Effectuent-elles les mêmes opérations ? Quelles sont les versions les plus efficaces ?

13) Réécrivez les deux boucles présentées dans les exemples précédents, en assembleur, aussi efficacement que vous pouvez.

14) En ajoutant uniquement une instruction JMP, convertissez les deux boucles dans le problème quatre en boucles REPEAT..UNTIL.

15) En ajoutant uniquement une instruction JMP, convertissez les deux boucles dans le problème cinq en boucles. WHILE

16) Convertissez les boucles FOR suivantes en assembleur (Note : vous êtes libres d'utiliser n'importe quelle routines fournie dans le pack de la Bibliothèque Standard de l'UCR) :

a)
```
FOR I := 0 TO 100 DO WriteLn(I);
```

b)
```
FOR I := 0 TO 7 DO
        FOR J := 0 TO 7 DO
                K := K*(I-J);
```

c)
```
FOR I := 255 TO 16 DO
        A [I] := A[240-I]-I;
```

17) Le mot réservé DOWNTO, utilisé en conjonction avec la boucle Pascal FOR, fait aller un compteur de boucle d'un nombre élevé vers un nombre inférieur. A La boucle FOR avec le mot réservé DOWNTO est équivalente à la boucle WHILE suivante :

```
loopvar := initial;
while (loopvar >= final) do begin
        stmt;
        loopvar := loopvar-1;
end;
```

Implémentez les boucles FOR Pascal suivantes en assembleur :

a)
```
FOR I := start DOWNTO stop DO WriteLn(I);
```

b)
```
FOR I := 7 DOWNTO 0 DO
        FOR J : = 0 à 7 DO
                K : = K*(I-J);
```

c) FOR I : = 255 DOWNTO 16 DO
A [I] := A[240-I]-I;

18) Réécrivez la boucle dans le problème 11b en gardant I dans BX, J dans CX et K dans AX.

19) Comment le fait de déplacer le test de terminaison de boucle à la fin de la boucle améliore l'implémentation de cette boucle ?

20) Qu'est-ce qu'un calcul invariable dans une boucle ?

21) Comment exécuter une boucle à l'envers améliore-t-il l'implémentation de la boucle ?

22) Que signifie démonter une boucle ?

23) Comment démonter une boucle améliore-t-il l'implémentation de la boucle ?

24) Donnez un exemple d'une boucle qui ne peut pas être démontée.

25) Donnez un exemple d'une boucle qui peut être démontée mais ne devrait pas l'être.