Control Structures Chapter 10

A computer program typically contains three structures: instruction sequences, decisions, and loops. A sequence is a set of sequentially executing instructions. A decision is a branch (goto) within a program based upon some condition. A loop is a sequence of instructions that will be repeatedly executed based on some condition. In this chapter we will explore some of the common decision structures in 80x86 assembly language.

10.0 Chapter Overview

This chapter discusses the two primary types of control structures: decision and iteration. It describes how to convert high level language statements like if..then..else, case (switch), while, for etc., into equivalent assembly language sequences. This chapter also discusses techniques you can use to improve the performance of these control structures. The sections below that have a "•" prefix are essential. Those sections with a "❏" discuss advanced topics that you may want to put off for a while.

- • Introduction to Decisions.
- • IF..THEN..ELSE Sequences.
- • CASE Statements.
- ❏ State machines and indirect jumps.
- • Spaghetti code.
- • Loops.
- • WHILE Loops.
- • REPEAT..UNTIL loops.
- • LOOP..ENDLOOP.
- • FOR Loops.
- • Register usage and loops.
- ❏ Performance improvements.
- ❏ Moving the termination condition to the end of a loop.
- ❏ Executing the loop backwards.
- ❏ Loop invariants.
- ❏ Unraveling loops.
- ❏ Induction variables.

10.1 Introduction to Decisions

In its most basic form, a decision is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the if..then..else statement in Pascal:

```
IF (condition is true) THEN stmt1 ELSE stmt2 ;
```

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following Pascal statement:

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

A "brute force" approach to converting this statement into assembly language might produce:

```
                    mov     cl, 1           ;Assume true
                    mov     ax, X
                    cmp     ax, Y
                    jl      IsTrue
                    mov     cl, 0           ;This one’s false
IsTrue:             mov     ax, Z
                    cmp     ax, T
                    jg      AndTrue
                    mov     cl, 0           ;It’s false now
AndTrue:            mov     al, A
                    cmp     al, B
                    je      OrFalse
                    mov     cl, 1           ;Its true if A <> B
OrFalse:            cmp     cl, 1
                    jne     SkipStmt1
       <Code for stmt1 goes here>
SkipStmt1:
```

As you can see, it takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) Pascal statements:

```
                    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;
```

Now compare this with the following “improved” code:

```
                    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:
       <Place code for Stmt1 here>

SkipStmt:
```

Two things should be apparent from the code sequences above: first, a single conditional statement in Pascal may require several conditional jumps in assembly language; second, organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, care should be exercised when dealing with conditional sequences in assembly language.

Conditional statements may be broken down into three basic categories: if..then..else statements, case statements, and indirect jumps. The following sections will describe these program structures, how to use them, and how to write them in assembly language.

10.2 IF..THEN..ELSE Sequences

The most commonly used conditional statement is theif..then or if..then..else statement. These two statements take the following form shown in Figure 10.1.

The if..then statement is just a special case of the if..then..else statement (with an empty ELSE block). Therefore, we’ll only consider the more general if..then..else form. The basic implementation of an if..then..else statement in 80x86 assembly language looks something like this:

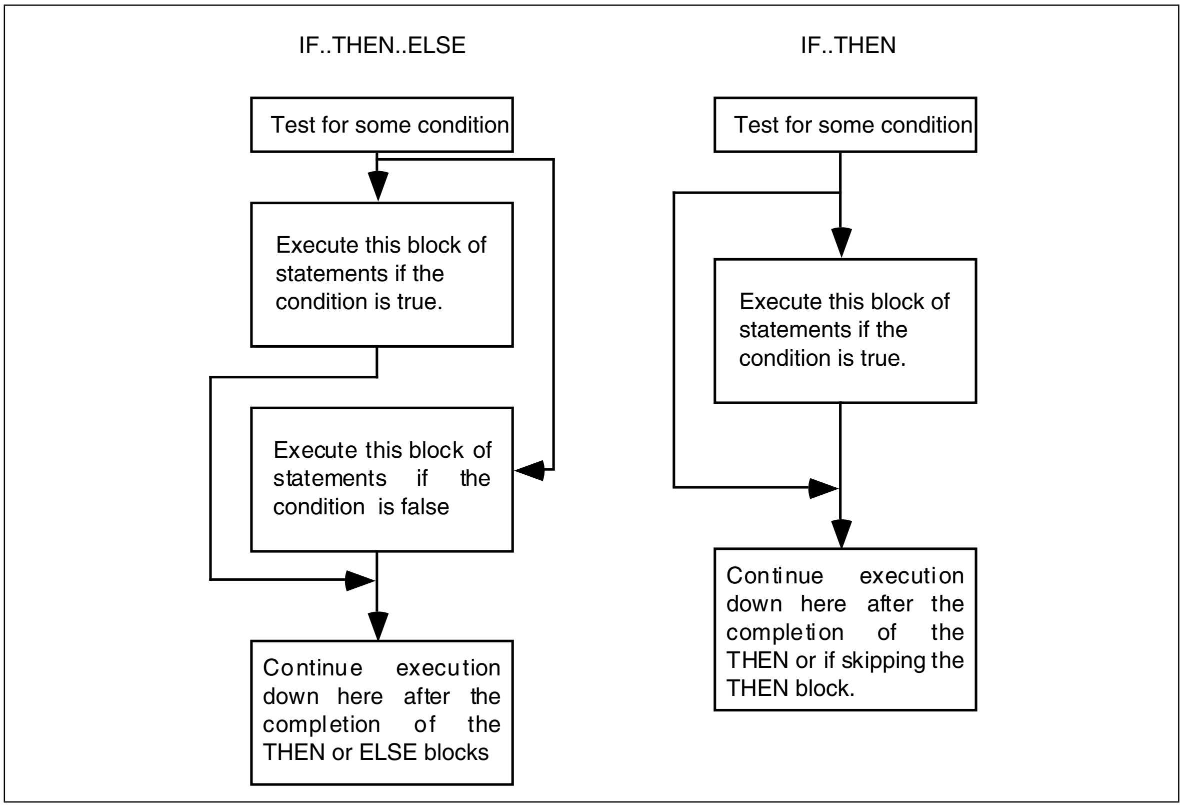
IF..THEN..ELSE
Test for some condition
Execute this block of statements if the condition is true.
Execute this block of statements if the condition is false
Continue execution down here after the completion of the THEN or ELSE blocks
IF..THEN
Test for some condition
Execute this block of statements if the condition is true.
Continue execution down here after the completion of the THEN or if skipping the THEN block.

Figure 10.1 IF..THEN and IF..THEN..ELSE Statement Flow

```
       {Sequence of statements to test some condition}
                 Jcc      ElseCode
       {Sequence of statements corresponding to the THEN block}
                 jmp      EndOfIF

ElseCode:
       {Sequence of statements corresponding to the ELSE block}

EndOfIF:
```

Note: J*cc* represents some conditional jump instruction.

For example, to convert the Pascal statement:

```
       IF (a=b) then c := d else b := b + 1;
```

to assembly language, you could use the following 80x86 code:

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

For simple expressions like (A=B) generating the proper code for an if..then..else statement is almost trivial. Should the expression become more complex, the associated assembly language code complexity increases as well. Consider the following if statement presented earlier:

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

When processing complex if statements such as this one, you'll find the conversion task easier if you break this if statement into a sequence of three different if statements as follows:

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

This conversion comes from the following Pascal equivalences:

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

is equivalent to

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

and

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

is equivalent to

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

In assembly language, the former if statement becomes:

```
                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:
```

As you can probably tell, the code necessary to test a condition can easily become more complex than the statements appearing in the else and then blocks. Although it seems somewhat paradoxical that it may take more effort to test a condition than to act upon the results of that condition, it happens all the time. Therefore, you should be prepared for this situation.

Probably the biggest problem with the implementation of complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. Probably the biggest advantage high level languages offer over assembly language is that expressions are much easier to read and comprehend in a high level language. The HLL version is self-documenting whereas assembly language tends to hide the true nature of the code. Therefore, well-written comments are an essential ingredient to assembly language implementations of if..then..else statements. An elegant implementation of the example above is:

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

```
; THEN Block:

DoIf:           mov      ax, D
                mov      C, ax

; End of IF statement

EndOfIF:
```

Admittedly, this appears to be going overboard for such a simple example. The following would probably suffice:

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

; Test the boolean expression:

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

; THEN Block:

DoIf:           mov      ax, D
                mov      C, ax

; End of IF statement

EndOfIF:
```

However, as your if statements become complex, the density (and quality) of your comments become more and more important.

10.3 CASE Statements

The Pascal case statement takes the following form :

```
CASE variable OF
          const1:stmt1;
          const2:stmt2;
           .
           .
           .
          constn:stmtn
END;
```

When this statement executes, it checks the value of variable against the constants $const_1$ … $const_n$. If a match is found then the corresponding statement executes. Standard Pascal places a few restrictions on the case statement. First, if the value of variable isn't in the list of constants, the result of the case statement is undefined. Second, all the constants appearing as case labels must be unique. The reason for these restrictions will become clear in a moment.

Most introductory programming texts introduce the case statement by explaining it as a sequence of if..then..else statements. They might claim that the following two pieces of Pascal code are equivalent:

```
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');
```

While semantically these two code segments may be the same, their implementation is usually different[1]. Whereas the if..then..else if chain does a comparison for each conditional statement in the sequence, the case statement normally uses an indirect jump to transfer control to any one of several statements with a single computation. Consider the two examples presented above, they could be written in assembly language with the following code:

```
                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:
```

Two things should become readily apparent: the more (consecutive) cases you have, the more efficient the jump table implementation becomes (both in terms of space and speed). Except for trivial cases, the case statement is almost always faster and usually by a large margin. As long as the case labels are consecutive values, the case statement version is usually smaller as well.

What happens if you need to include non-consecutive case labels or you cannot be sure that the case variable doesn't go out of range? Many Pascals have extended the definition of the case statement to include an otherwise clause. Such a case statement takes the following form:

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

If the value of variable matches one of the constants making up the case labels, then the associated statement executes. If the variable's value doesn't match any of the case

1. Versions of Turbo Pascal, sadly, treat the case statement as a form of the if..then..else statement.

labels, then the statement following the otherwise clause executes. The otherwise clause is implemented in two phases. First, you must choose the minimum and maximum values that appear in a case statement. In the following case statement, the smallest case label is five, the largest is 15:

```
CASE I OF
          5:stmt1;
          8:stmt2;
          10:stmt3;
          12:stmt4;
          15:stmt5;
          OTHERWISE stmt6
END;
```

Before executing the jump through the jump table, the 80x86 implementation of this case statement should check the case variable to make sure it's in the range 5..15. If not, control should be immediately transferred to stmt6:

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

The only problem with this form of the case statement as it now stands is that it doesn't properly handle the situation where I is equal to 6, 7, 9, 11, 13, or 14. Rather than sticking extra code in front of the conditional jump, you can stick extra entries in the jump table as follows:

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

Otherwise:       {put stmt6 here}
                 jmp      CaseDone

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

Note that the value 10 is subtracted from the address of the jump table. The first entry in the table is always at offset zero while the smallest value used to index into the table is five (which is multiplied by two to produce 10). The entries for 6, 7, 9, 11, 13, and 14 all point at the code for the Otherwise clause, so if I contains one of these values, the Otherwise clause will be executed.

There is a problem with this implementation of the case statement. If the case labels contain non-consecutive entries that are widely spaced, the following case statement would generate an extremely large code file:

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

In this situation, your program will be much smaller if you implement the case statement with a sequence of if statements rather than using a jump statement. However, keep one thing in mind- the size of the jump table does not normally affect the execution speed of the program. If the jump table contains two entries or two thousand, the case statement will execute the multi-way branch in a constant amount of time. The if statement implementation requires a linearly increasing amount of time for each case label appearing in the case statement.

Probably the biggest advantage to using assembly language over a HLL like Pascal is that you get to choose the actual implementation. In some instances you can implement a case statement as a sequence ofif..then..else statements, or you can implement it as a jump table, or you can use a hybrid of the two:

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

could become:

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

Of course, you could do this in Pascal with the following code:

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

But this tends to destroy the readability of the Pascal program. On the other hand, the extra code to test for 100 in the assembly language code doesn't adversely affect the readability of the program (perhaps because it's so hard to read already). Therefore, most people will add the extra code to make their program more efficient.

The C/C++ switch statement is very similar to the Pascal case statement. There is only one major semantic difference: the programmer must explicitly place a break statement in each case clause to transfer control to the first statement beyond the switch. This break corresponds to the jmp instruction at the end of each case sequence in the assembly code above. If the corresponding break is not present, C/C++ transfers control into the code of the following case. This is equivalent to leaving off the jmp at the end of the case's sequence:

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

This translates into the following 80x86 code:

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

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

```
case0:           <stmt1's code>

case1:           <stmt2's code>

case2:           <stmt3's code>

                 jmp      EndCase      ;Emitted for the break stmt.

case3:           <stmt4's code>
                 jmp      EndCase      ;Emitted for the break stmt.

DefaultCase:     <stmt5's code>
EndCase:
```

10.4 State Machines and Indirect Jumps

Another control structure commonly found in assembly language programs is the *state machine*. A state machine uses a *state variable* to control program flow. The FORTRAN programming language provides this capability with the assigned goto statement. Certain variants of C (e.g., GNU's GCC from the Free Software Foundation) provide similar features. In assembly language, the indirect jump provides a mechanism to easily implement state machines.

So what is a state machine? In very basic terms, it is a piece of code[2] which keeps track of its execution history by entering and leaving certain "states". For the purposes of this chapter, we'll not use a very formal definition of a state machine. We'll just assume that a state machine is a piece of code which (somehow) remembers the history of its execution (its *state*) and executes sections of code based upon that history.

In a very real sense, all programs are state machines. The CPU registers and values in memory constitute the "state" of that machine. However, we'll use a much more constrained view. Indeed, for most purposes only a single variable (or the value in the IP register) will denote the current state.

Now let's consider a concrete example. Suppose you have a procedure which you want to perform one operation the first time you call it, a different operation the second time you call it, yet something else the third time you call it, and then something new again on the fourth call. After the fourth call it repeats these four different operations in order. For example, suppose you want the procedure to add ax and bx the first time, subtract them on the second call, multiply them on the third, and divide them on the fourth. You could implement this procedure as follows:

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

; If this is state 0, add BX to AX and switch to state 1:

                 add      ax, bx
                 inc      State                  ;Set it to state 1
                 ret

; If this is state 1, subtract BX from AX and switch to state 2

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

; If this is state 2, multiply AX and BX and switch to state 3:

TryState2:       cmp      State, 2
```

2. Note that state machines need not be software based. Many state machines' implementation are hardware based.

```
                jne     MustBeState3
                push    dx
                mul     bx
                pop     dx
                inc     State
                ret

; If none of the above, assume we're in State 4. So divide
; AX by BX.

MustBeState3:
                push    dx
                xor     dx, dx      ;Zero extend AX into DX.
                div     bx
                pop     dx
                mov     State, 0    ;Switch back to State 0
                ret
StateMach       endp
```

Technically, this procedure is not the state machine. Instead, it is the variable State and the cmp/jne instructions which constitute the state machine.

There is nothing particularly special about this code. It's little more than a case statement implemented via theif..then..else construct. The only thing special about this procedure is that it remembers how many times it has been called[3] and behaves differently depending upon the number of calls. While this is a *correct* implementation of the desired state machine, it is not particularly efficient. The more common implementation of a state machine in assembly language is to use an *indirect jump*. Rather than having a state variable which contains a value like zero, one, two, or three, we could load the state variable with the *address* of the code to execute upon entry into the procedure. By simply jumping to that address, the state machine could save the tests above needed to execute the proper code fragment. Consider the following implementation using the indirect jump:

```
State           word    State0
StateMach       proc
                jmp     State

; If this is state 0, add BX to AX and switch to state 1:

State0:         add     ax, bx
                mov     State, offset State1          ;Set it to state 1
                ret

; If this is state 1, subtract BX from AX and switch to state 2

State1:         sub     ax, bx
                mov     State, offset State2          ;Switch to State 2
                ret

; If this is state 2, multiply AX and BX and switch to state 3:

State2:         push    dx
                mul     bx
                pop     dx
                mov     State, offset State3          ;Switch to State 3
                ret

; If in State 3, do the division and switch back to State 0:

State3:         push    dx
                xor     dx, dx      ;Zero extend AX into DX.
                div     bx
                pop     dx
                mov     State, offset State0          ;Switch to State 0
                ret
StateMach       endp
```

The jmp instruction at the beginning of the StateMach procedure transfers control to the location pointed at by the State variable. The first time you call StateMach it points at

3. Actually, it remembers how many times, *MOD 4*, that it has been called.

the State0 label. Thereafter, each subsection of code sets the State variable to point at the appropriate successor code.

10.5 Spaghetti Code

One major problem with assembly language is that it takes several statements to realize a simple idea encapsulated by a single HLL statement. All too often an assembly language programmer will notice that s/he can save a few bytes or cycles by jumping into the middle of some programming structure. After a few such observations (and corresponding modifications) the code contains a whole sequence of jumps in and out of portions of the code. If you were to draw a line from each jump to its destination, the resulting listing would end up looking like someone dumped a bowl of spaghetti on your code, hence the term "spaghetti code".

Spaghetti code suffers from one major drawback- it's difficult (at best) to read such a program and figure out what it does. Most programs start out in a "structured" form only to become spaghetti code at the altar of efficiency. Alas, spaghetti code is rarely efficient. Since it's difficult to figure out exactly what's going on, it's very difficult to determine if you can use a better algorithm to improve the system. Hence, spaghetti code may wind up less efficient.

While it's true that producing some spaghetti code in your programs may improve its efficiency, doing so should always be a last resort (when you've tried everything else and you still haven't achieved what you need), never a matter of course. Always start out writing your programs with straight-forward ifs and case statements. Start combining sections of code (via jmp instructions) once everything is working and well understood. Of course, you should never obliterate the structure of your code unless the gains are worth it.

A famous saying in structured programming circles is "After gotos, pointers are the next most dangerous element in a programming language." A similar saying is "Pointers are to data structures what gotos are to control structures." In other words, avoid excessive use of pointers. If pointers and gotos are bad, then the indirect jump must be the worst construct of all since it involves both gotos and pointers! Seriously though, the indirect jump instructions should be avoided for casual use. They tend to make a program harder to read. After all, an indirect jump can (theoretically) transfer control to any label within a program. Imagine how hard it would be to follow the flow through a program if you have no idea what a pointer contains and you come across an indirect jump using that pointer. Therefore, you should always exercise care when using jump indirect instructions.

10.6 Loops

Loops represent the final basic control structure (sequences, decisions, and loops) which make up a typical program. Like so many other structures in assembly language, you'll find yourself using loops in places you've never dreamed of using loops. Most HLLs have implied loop structures hidden away. For example, consider the BASIC statement IF A$ = B$ THEN 100. This if statement compares two strings and jumps to statement 100 if they are equal. In assembly language, you would need to write a loop to compare each character in A$ to the corresponding character in B$ and then jump to statement 100 if and only if all the characters matched. In BASIC, there is no loop to be seen in the program. In assembly language, this very simple if statement requires a loop. This is but a small example which shows how loops seem to pop up everywhere.

Program loops consist of three components: an optional initialization component, a loop termination test, and the body of the loop. The order with which these components are assembled can dramatically change the way the loop operates. Three permutations of these components appear over and over again. Because of their frequency, these loop structures are given special names in HLLs: while loops, repeat..until loops (do..while in C/C++), and loop..endloop loops.

10.6.1 While Loops

The most general loop is the while loop. It takes the following form:

```
WHILE boolean expression DO statement;
```

There are two important points to note about the while loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of the position of the termination test, the body of the loop may never execute. If the termination condition always exists, the loop body will always be skipped over.

Consider the following Pascal while loop:

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

I := 0; is the initialization code for this loop. I is a loop control variable, because it controls the execution of the body of the loop. (I<100) is the loop termination condition. That is, the loop will not terminate as long as I is less than 100. I:=I+1; is the loop body. This is the code that executes on each pass of the loop. You can convert this to 80x86 assembly language as follows:

```
                mov     I, 0
WhileLp:        cmp     I, 100
                jge     WhileDone
                inc     I
                jmp     WhileLp

WhileDone:
```

Note that a Pascal while loop can be easily synthesized using an if and a goto statement. For example, the Pascal while loop presented above can be replaced by:

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

More generally, any while loop can be built up from the following:

```
       optional initialization code
1:     IF not termination condition THEN BEGIN
                 loop body
                 GOTO 1;
       END;
```

Therefore, you can use the techniques from earlier in this chapter to convert if statements to assembly language. All you'll need is an additional jmp (goto) instruction.

10.6.2 Repeat..Until Loops

The repeat..until (do..while) loop tests for the termination condition at the end of the loop rather than at the beginning. In Pascal, the repeat..until loop takes the following form:

```
optional initialization code
REPEAT
          loop body
UNTIL termination condition
```

This sequence executes the initialization code, the loop body, then tests some condition to see if the loop should be repeated. If the boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things to note about the repeat..until loop is that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body executes at least once.

Like the while loop, the repeat..until loop can be synthesized with an if statement and a goto . You would use the following:

```
       initialization code
1:              loop body
       IF NOT termination condition THEN GOTO 1
```

Based on the material presented in the previous sections, you can easily synthesize repeat..until loops in assembly language.

10.6.3 LOOP..ENDLOOP Loops

If while loops test for termination at the beginning of the loop and repeat..until loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. Although Pascal and C/C++[4] don't directly support such a loop, the loop..endloop structure can be found in HLL languages like Ada. The loop..endloop loop takes the following form:

```
       LOOP
                loop body
       ENDLOOP;
```

Note that there is no explicit termination condition. Unless otherwise provided for, the loop..endloop construct simply forms an infinite loop. Loop termination is handled by an if and goto statement[5]. Consider the following (pseudo) Pascal code which employs a loop..endloop construct:

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

In real Pascal, you'd use the following code to accomplish this:

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

In assembly language you'd end up with something like:

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

10.6.4 FOR Loops

The for loop is a special form of the while loop which repeats the loop body a specific number of times. In Pascal, the for loop looks something like the following:

```
       FOR var := initial TO final DO stmt
or
       FOR var := initial DOWNTO final DO stmt
```

Traditionally, the for loop in Pascal has been used to process arrays and other objects accessed in sequential numeric order. These loops can be converted directly into assembly language as follows:

4. Technically, C/C++ *does* support such a loop. "for(;;)" along with break provides this capability.
5. Many high level languages use statements like NEXT, BREAK, CONTINUE, EXIT, and CYCLE rather than GOTO; but they're all forms of the GOTO statement.

In Pascal:

FOR var := start TO stop DO stmt;

In Assembly:

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

; code corresponding to stmt goes here.

                 inc      var
                 jmp      FL
EndFor:
```

Fortunately, most for loops repeat some statement(s) a fixed number of times. For example,

```
FOR I := 0 to 7 do write(ch);
```

In situations like this, it's better to use the 80x86 loop instruction rather than simulate a for loop:

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

Keep in mind that the loop instruction normally appears at the end of a loop whereas the for loop tests for termination at the beginning of the loop. Therefore, you should take precautions to prevent a runaway loop in the event cx is zero (which would cause the loop instruction to repeat the loop 65,536 times) or the stop value is less than the start value. In the case of

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

assuming you don't use the value of var within the loop, you'd probably want to use the assembly code:

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

Remember, the sub and cmp instructions set the flags in an identical fashion. Therefore, this loop will be skipped if stop is less than start. It will be repeated (stop-start)+1 times otherwise. If you need to reference the value of var within the loop, you could use the following code:

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

The downto version appears in the exercises.

10.7 Register Usage and Loops

Given that the 80x86 accesses registers much faster than memory locations, registers are the ideal spot to place loop control variables (especially for small loops). This point is amplified since the loop instruction requires the use of the cx register. However, there are some problems associated with using registers within a loop. The primary problem with using registers as loop control variables is that registers are a limited resource. In particular, there is only one cx register. Therefore, the following will not work properly:

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

The intent here, of course, was to create a set of nested loops, that is, one loop inside another. The inner loop (Loop2) should repeat four times for each of the eight executions of the outer loop (Loop1). Unfortunately, both loops use the loop instruction. Therefore, this will form an infinite loop since cx will be set to zero (which loop treats like 65,536) at the end of the first loop instruction. Since cx is always zero upon encountering the second loop instruction, control will always transfer to the Loop1 label. The solution here is to save and restore the cx register or to use a different register in place of cx for the outer loop:

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

or:

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

Register corruption is one of the primary sources of bugs in loops in assembly language programs, always keep an eye out for this problem.

10.8 Performance Improvements

The 80x86 microprocessors execute sequences of instructions at blinding speeds. You'll rarely encounter a program that is slow which doesn't contain any loops. Since loops are the primary source of performance problems within a program, they are the place to look when attempting to speed up your software. While a treatise on how to write efficient programs is beyond the scope of this chapter, there are some things you should be aware of when designing loops in your programs. They're all aimed at removing unnecessary instructions from your loops in order to reduce the time it takes to execute one iteration of the loop.

10.8.1 Moving the Termination Condition to the End of a Loop

Consider the following flow graphs for the three types of loops presented earlier:

Repeat..until loop:

```
Initialization code
          Loop body
Test for termination
Code following the loop
```

While loop:

```
        Initialization code
        Loop termination test
                Loop body
                Jump back to test
        Code following the loop
```

Loop..endloop loop:

```
        Initialization code
                Loop body, part one
                Loop termination test
                Loop body, part two
                Jump back to loop body part 1
        Code following the loop
```

As you can see, the repeat..until loop is the simplest of the bunch. This is reflected in the assembly language code required to implement these loops. Consider the following repeat..until and while loops that are identical:

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

                stmts                           stmts
                SI := SI + 1;                   SI := SI + 1;

        end;                            until SI > DI;
```

The assembly language code for these two loops is[6]:

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

As you can see, testing for the termination condition at the end of the loop allowed us to remove a jmp instruction from the loop. This can be significant if this loop is nested inside other loops. In the preceding example there wasn't a problem with executing the body at least once. Given the definition of the loop, you can easily see that the loop will be executed exactly 20 times. Assuming cx is available, this loop easily reduces to:

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

Unfortunately, it's not always quite this easy. Consider the following Pascal code:

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

In this particular example, we haven't the slightest idea what si contains upon entry into the loop. Therefore, we cannot assume that the loop body will execute at least once. Therefore, we must do the test before executing the body of the loop. The test can be placed at the end of the loop with the inclusion of a single jmp instruction:

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

6. Of course, a good compiler would recognize that both loops perform the same operation and generate identical code for each. However, most compilers are not this good.

Although the code is as long as the original while loop, the jmp instruction executes only once rather than on each repetition of the loop. Note that this slight gain in efficiency is obtained via a slight loss in readability. The second code sequence above is closer to spaghetti code that the original implementation. Such is often the price of a small performance gain. Therefore, you should carefully analyze your code to ensure that the performance boost is worth the loss of clarity. More often than not, assembly language programmers sacrifice clarity for dubious gains in performance, producing impossible to understand programs.

10.8.2 Executing the Loop Backwards

Because of the nature of the flags on the 80x86, loops which range from some number down to (or up to) zero are more efficient than any other. Compare the following Pascal loops and the code they generate:

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

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

Note that by running the loop from eight down to one (the code on the right) we saved a comparison on each repetition of the loop.

Unfortunately, you cannot force all loops to run backwards. However, with a little effort and some coercion you should be able to work most loops so they operate backwards. Once you get a loop operating backwards, it's a good candidate for the loop instruction (which will improve the performance of the loop on pre-486 CPUs).

The example above worked out well because the loop ran from eight down to one. The loop terminated when the loop control variable became zero. What happens if you need to execute the loop when the loop control variable goes to zero? For example, suppose that the loop above needed to range from seven down to zero. As long as the upper bound is positive, you can substitute the jns instruction in place of the jnz instruction above to repeat the loop some specific number of times:

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

This loop will repeat eight times with I taking on the values seven down to zero on each execution of the loop. When it decrements zero to minus one, it sets the sign flag and the loop terminates.

Keep in mind that some values may look positive but they are negative. If the loop control variable is a byte, then values in the range 128..255 are negative. Likewise, 16-bit values in the range 32768..65535 are negative. Therefore, initializing the loop control variable with any value in the range 129..255 or 32769..65535 (or, of course, zero) will cause the loop to terminate after a single execution. This can get you into a lot of trouble if you're not careful.

10.8.3 Loop Invariant Computations

A loop invariant computation is some calculation that appears within a loop that always yields the same result. You needn't do such computations inside the loop. You can compute them outside the loop and reference the value of the computation inside. The following Pascal code demonstrates a loop which contains an invariant computation:

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

Since J never changes throughout the execution of this loop, the sub-expression "J-2" can be computed outside the loop and its value used in the expression inside the loop:

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

Of course, if you're really interested in improving the efficiency of this particular loop, you'd be much better off (most of the time) computing K using the formula:

$$K = K + ((N+1) \times temp) + \frac{(N+2) \times (N+2)}{2}$$

This computation for K is based on the formula:

$$\sum_{i=0}^{N} i = \frac{(N+1) \times (N)}{2}$$

However, simple computations such as this one aren't always possible. Still, this demonstrates that a better algorithm is almost always better than the trickiest code you can come up with.

In assembly language, invariant computations are even trickier. Consider this conversion of the Pascal code above:

```
                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
```

Of course, the first refinement we can make is to move the loop control variable (I) into a register. This produces the following code:

```
                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
```

This operation speeds up the loop by removing a memory access from each repetition of the loop. To take this one step further, why not use a register to hold the “temp” value rather than a memory location:

```
                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
```

Furthermore, accessing the variable K can be removed from the loop as well:

```
                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
```

One final improvement which is begging to be made is to substitute the loop instruction for the dec cx / cmp cx,-1 / JG FLP instructions. Unfortunately, this loop must be repeated whenever the loop control variable hits zero, the loop instruction cannot do this. However, we can unravel the last execution of the loop (see the next section) and do that computation outside the loop as follows:

```
                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
```

As you can see, these refinements have considerably reduced the number of instructions executed inside the loop and those instructions that do appear inside the loop are very fast since they all reference registers rather than memory locations.

Removing invariant computations and unnecessary memory accesses from a loop (particularly an inner loop in a set of nested loops) can produce dramatic performance improvements in a program.

10.8.4 Unraveling Loops

For small loops, that is, those whose body is only a few statements, the overhead required to process a loop may constitute a significant percentage of the total processing time. For example, look at the following Pascal code and its associated 80x86 assembly language code:

```
        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
```

Each execution of the loop requires five instructions. Only one instruction is performing the desired operation (moving a zero into an element of A). The remaining four instructions convert the loop control variable into an index into A and control the repetition of the loop. Therefore, it takes 20 instructions to do the operation logically required by four.

While there are many improvements we could make to this loop based on the information presented thus far, consider carefully exactly what it is that this loop is doing-- it's simply storing four zeros into A[0] through A[3]. A more efficient approach is to use four mov instructions to accomplish the same task. For example, if A is an array of words, then the following code initializes A much faster than the code above:

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

You may improve the execution speed and the size of this code by using the ax register to hold zero:

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

Although this is a trivial example, it shows the benefit of loop unraveling. If this simple loop appeared buried inside a set of nested loops, the 5:1 instruction reduction could possibly double the performance of that section of your program.

Of course, you cannot unravel all loops. Loops that execute a variable number of times cannot be unraveled because there is rarely a way to determine (at assembly time) the number of times the loop will be executed. Therefore, unraveling a loop is a process best applied to loops that execute a known number of times.

Even if you repeat a loop some fixed number of iterations, it may not be a good candidate for loop unraveling. Loop unraveling produces impressive performance improvements when the number of instructions required to control the loop (and handle other overhead operations) represent a significant percentage of the total number of instructions in the loop. Had the loop above contained 36 instructions in the body of the loop (exclusive of the four overhead instructions), then the performance improvement would be, at best, only 10% (compared with the 300-400% it now enjoys). Therefore, the costs of unraveling a loop, i.e., all the extra code which must be inserted into your program, quickly reaches a point of diminishing returns as the body of the loop grows larger or as the number of iterations increases. Furthermore, entering that code into your program can become quite a chore. Therefore, loop unraveling is a technique best applied to small loops.

Note that the superscalar x86 chips (Pentium and later) have *branch prediction hardware* and use other techniques to improve performance. Loop unrolling on such systems many actually *slow down* the code since these processors are optimized to execute short loops.

10.8.5 Induction Variables

The following is a slight modification of the loop presented in the previous section:

```
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
```

Although unraveling this code will still produce a tremendous performance improvement, it will take 257 instructions to accomplish this task[7], too many for all but the most time-critical applications. However, you can reduce the execution time of the body of the loop tremendously using *induction variables*. An induction variable is one whose value depends entirely on the value of some other variable. In the example above, the index into the array A tracks the loop control variable (it's always equal to the value of the loop control variable times two). Since I doesn't appear anywhere else in the loop, there is no sense in performing all the computations on I. Why not operate directly on the array index value? The following code demonstrates this technique:

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

Here, several instructions accessing memory were replaced with instructions that only access registers. Another improvement to make is to shorten the MOVA[bx],0 instruction using the following code:

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

This code transformation improves the performance of the loop even more. However, we can improve the performance even more by using the loop instruction and the cx register to eliminate the cmp instruction[8]:

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

This final transformation produces the fastest executing version of this code[9].

10.8.6 Other Performance Improvements

There are many other ways to improve the performance of a loop within your assembly language programs. For additional suggestions, a good text on compilers such as "Compilers, Principles, Techniques, and Tools" by Aho, Sethi, and Ullman would be an

7. For this particular loop, the STOSW instruction could produce a big performance improvement on many 80x86 processors. Using the STOSW instruction would require only about six instructions for this code. See the chapter on string instructions for more details.
8. The LOOP instruction is not the best choice on the 486 and Pentium processors since dec cx" followed by "jne lbl" actually executes faster.
9. Fastest is a dangerous statement to use here! But it is the fastest of the examples presented here.

excellent place to look. Additional efficiency considerations will be discussed in the volume on efficiency and optimization.

10.9 Nested Statements

As long as you stick to the templates provides in the examples presented in this chapter, it is very easy to nest statements inside one another. The secret to making sure your assembly language sequences nest well is to ensure that each construct has one entry point and one exit point. If this is the case, then you will find it easy to combine statements. All of the statements discussed in this chapter follow this rule.

Perhaps the most commonly nested statements are the if..then..else statements. To see how easy it is to nest these statements in assembly language, consider the following Pascal code:

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

To convert this nested if..then..else to assembly language, start with the outermost if, convert it to assembly, then work on the innermost if:

```
; if (x = y) then

                 mov      ax, X
                 cmp      ax, Y
                 jne      Else0

; Put innermost IF here

                 jmp      IfDone0

; Else write('Error condition');

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

As you can see, the above code handles the "if (X=Y)..." instruction, leaving a spot for the second if. Now add in the second if as follows:

```
; 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:
```

The nested if appears in italics above just to help it stand out.

There is an obvious optimization which you do not really want to make until speed becomes a real problem. Note in the innermost if statement above that the JMP IFDONE1 instructions simply jumps to a jmp instruction which transfers control to IfDone0. It is very tempting to replace the first jmp by one which jumps directly to IFDone0. Indeed, when you go in and optimize your code, this would be a good optimization to make. However, you shouldn't make such optimizations to your code unless you really need the speed. Doing so makes your code harder to read and understand. Remember, we would like all our control structures to have one entry and one exit. Changing this jump as described would give the innermost if statement two exit points.

The for loop is another commonly nested control structure. Once again, the key to building up nested structures is to construct the outside object first and fill in the inner members afterwards. As an example, consider the following nested for loops which add the elements of a pair of two dimensional arrays together:

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

As before, begin by constructing the outermost loop first. This code assumes that dx will be the loop control variable for the outermost loop (that is, dx is equivalent to "i"):

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

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

; Put innermost FOR loop here

                inc     dx
                jmp     ForLp0
EndFor0:
```

Now add the code for the nested for loop. Note the use of the cx register for the loop control variable on the innermost for loop of this 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

; Put code for A[dx,cx] := b[dx,cx] + C [dx,cx] here

                inc     cx
                jmp     ForLp1
EndFor1:

                inc     dx
                jmp     ForLp0
EndFor0:
```

Once again the innermost for loop is in italics in the above code to make it stand out. The final step is to add the code which performs that actual computation.

10.10 Timing Delay Loops

Most of the time the computer runs too slow for most people's tastes. However, there are occasions when it actually runs *too fast*. One common solution is to create an empty loop to waste a small amount of time. In Pascal you will commonly see loops like:

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

In assembly, you might see a comparable loop:

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

By carefully choosing the number of iterations, you can obtain a relatively accurate delay interval. There is, however, one catch. That relatively accurate delay interval is only going to be accurate on *your* machine. If you move your program to a different machine with a different CPU, clock speed, number of wait states, different sized cache, or half a dozen other features, you will find that your delay loop takes a completely different amount of time. Since there is better than a hundred to one difference in speed between the high end and low end PCs today, it should come as no surprise that the loop above will execute 100 times faster on some machines than on others.

The fact that one CPU runs 100 times faster than another does not reduce the need to have a delay loop which executes some fixed amount of time. Indeed, it makes the problem that much more important. Fortunately, the PC provides a hardware based timer which operates at the same speed regardless of the CPU speed. This timer maintains the time of day for the operating system, so it's very important that it run at the same speed whether you're on an 8088 or a Pentium. In the chapter on interrupts you will learn to actually patch into this device to perform various tasks. For now, we will simply take advantage of the fact that this timer chip forces the CPU to increment a 32-bit memory location (40:6ch) about 18.2 times per second. By looking at this variable we can determine the speed of the CPU and adjust the count value for an empty loop accordingly.

The basic idea of the following code is to watch the BIOS timer variable until it changes. Once it changes, start counting the number of iterations through some sort of loop until the BIOS timer variable changes again. Having noted the number of iterations, if you execute a similar loop the same number of times it should require about 1/18.2 seconds to execute.

The following program demonstrates how to create such a Delay routine:

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

; PPI_B is the I/O address of the keyboard/speaker control
; port. This program accesses it simply to introduce a
; large number of wait states on faster machines. Since the
; PPI (Programmable Peripheral Interface) chip runs at about
; the same speed on all PCs, accessing this chip slows most
; machines down to within a factor of two of the slower
; machines.

PPI_B            equ      61h

; RTC is the address of the BIOS timer variable (40:6ch).
; The BIOS timer interrupt code increments this 32-bit
; location about every 55 ms (1/18.2 seconds). The code
; which initializes everything for the Delay routine
; reads this location to determine when 1/18th seconds
; have passed.

RTC              textequ  <es:[6ch]>

dseg             segment  para public 'data'
```

```
; TimedValue contains the number of iterations the delay
; loop must repeat in order to waste 1/18.2 seconds.

TimedValue          word      0

; RTC2 is a dummy variable used by the Delay routine to
; simulate accessing a BIOS variable.

RTC2                word      0


dseg                ends



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

; Main program which tests out the DELAY subroutine.

Main                proc
                    mov       ax, dseg
                    mov       ds, ax

                    print
                    byte      "Delay test routine",cr,lf,0

; Okay, let's see how long it takes to count down 1/18th
; of a second. First, point ES as segment 40h in memory.
; The BIOS variables are all in segment 40h.
;
; This code begins by reading the memory timer variable
; and waiting until it changes. Once it changes we can
; begin timing until the next change occurs. That will
; give us 1/18.2 seconds. We cannot start timing right
; away because we might be in the middle of a 1/18.2
; second period.

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

; Okay, begin timing the number of iterations it takes
; for an 18th of a second to pass. Note that this
; code must be very similar to the code in the Delay
; routine.

                    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                      ;CX counted down!
                    mov       TimedValue, cx          ;Save away

                    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   ;DOS macro to quit program.
Main            endp

; Delay18-This routine delays for approximately 1/18th sec.
;       Presumably, the variable "TimedValue" in DS has
;       been initialized with an appropriate count down
;       value before calling this 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

; The following code contains two loops. The inside
; nested loop repeats 10 times. The outside loop
; repeats the number of times determined to waste
; 1/18.2 seconds. This loop accesses the hardware
; port "PPI_B" in order to introduce many wait states
; on the faster processors. This helps even out the
; timings on very fast machines by slowing them down.
; Note that accessing PPI_B is only done to introduce
; these wait states, the data read is of no interest
; to this code.
;
; Note the similarity of this code to the code in the
; main program which initializes the TimedValue variable.

                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 Sample Program

This chapter’s sample program is a simple moon lander game. While the simulation isn’t terribly realistic, this program does demonstrate the use and optimization of several different control structures including loops, if..then..else statements, and so on.

```
; Simple "Moon Lander" game.
;
; Randall Hyde
; 2/8/96
;
; This program is an example of a trivial little "moon lander"
; game that simulates a Lunar Module setting down on the Moon's
; surface.  At time T=0 the spacecraft's velocity is 1000 ft/sec
; downward, the craft has 1000 units of fuel, and the craft is
; 10,000 ft above the moon's surface.  The pilot (user) can
; specify how much fuel to burn at each second.
;
; Note that all calculations are approximate since everything is
; done with integer arithmetic.


; Some important constants

InitialVelocity  =        1000
InitialDistance  =        10000
InitialFuel      =        250
MaxFuelBurn      =        25

MoonsGravity     =        5           ;Approx 5 ft/sec/sec
AccPerUnitFuel   =        -5          ;-5 ft/sec/sec for each fuel unit.


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

dseg             segment   para public 'data'

; Current distance from the Moon's Surface:

CurDist          word      InitialDistance

; Current Velocity:

CurVel           word      InitialVelocity

; Total fuel left to burn:

FuelLeft         word      InitialFuel


; Amount of Fuel to use on current burn.

Fuel             word      ?

; Distance travelled in the last second.

Dist             word      ?

dseg             ends


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

```
; GETI-Reads an integer variable from the user and returns its
;       its value in the AX register.  If the user entered garbage,
;       this code will make the user re-enter the value.

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

; Read a string of characters from the user.
;
; Note that there are two (nested) loops here.  The outer loop
; (GetILp) repeats the getsm operation as long as the user
; keeps entering an invalid number.  The innermost loop (ChkDigits)
; checks the individual characters in the input string to make
; sure they are all decimal digits.

GetILp:         getsm

; Check to see if this string contains any non-digit characters:
;
; while (([bx] >= '0') and ([bx] <= '9')  bx := bx + 1;
;
; Note the sneaky way of turning the while loop into a
; repeat..until loop.

                mov      bx, di       ;Pointer to start of string.
                dec      bx
ChkDigits:      inc      bx
                mov      al, es:[bx]  ;Fetch next character.
                IsDigit               ;See if it's a decimal digit.
                je       ChkDigits    ;Repeat if it is.

                cmp      al, 0        ;At end of string?
                je       GotNumber

; Okay, we just ran into a non-digit character.  Complain and make
; the user reenter the value.

                free                  ;Free space malloc'd by getsm.
                print
                byte     cr,lf
                byte     "Illegal unsigned integer value, "
                byte     "please reenter.",cr,lf
                byte     "(no spaces, non-digit chars, etc.):",0
                jmp      GetILp


; Okay, ES:DI is pointing at something resembling a number.  Convert
; it to an integer.

GotNumber:      atoi
                free                  ;Free space malloc'd by getsm.

                pop      bx
                pop      di
                pop      es
                ret
_geti           endp


; InitGame-     Initializes global variables this game uses.

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

```
InitGame          endp


; DispStatus-     Displays important information for each
;                 cycle of the game (a cycle is one second).

DispStatus        proc
                  printf
                  byte     cr,lf
                  byte     "Distance from surface: %5d",cr,lf
                  byte     "Current velocity:      %5d",cr,lf
                  byte     "Fuel left:             %5d",cr,lf
                  byte     lf
                  byte     "Dist travelled in the last second: %d",cr,lf
                  byte     lf,0
                  dword    CurDist, CurVel, FuelLeft, Dist
                  ret
DispStatus        endp


; GetFuel-        Reads an integer value representing the amount of fuel
;                 to burn from the user and checks to see if this value
;                 is reasonable.  A reasonable value must:
;
;                 * Be an actual number (GETI handles this).
;                 * Be greater than or equal to zero (no burning
;                   negative amounts of fuel, GETI handles this).
;                 * Be less than MaxFuelBurn (any more than this and
;                   you have an explosion, not a burn).
;                 * Be less than the fuel left in the Lunar Module.

GetFuel           proc
                  push     ax

; Loop..endloop structure that reads an integer input and terminates
; if the input is reasonable.  It prints a message an repeats if
; the input is not reasonable.
;
; 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     "Enter amount of fuel to burn: ",0
                  geti
                  cmp      ax, MaxFuelBurn
                  jbe      GoodFuel

                  print
                  byte     "The amount you've specified exceeds the "
                  byte     "engine rating,", cr, lf
                  byte     "please enter a smaller value",cr,lf,lf,0
                  jmp      GetFuelLp

GoodFuel:         mov      Fuel, ax
                  cmp      ax, FuelLeft
                  jbe      HasEnough
                  printf
                  byte     "There are only %d units of fuel left.",cr,lf
                  byte     "The Lunar module will burn this rather than %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-
;
;      This routine computes the new velocity and new distance based on the
;      current distance, current velocity, fuel burnt, and the moon's
;      gravity.  This routine is called for every "second" of flight time.
;      This simplifies the following equations since the value of T is
;      always one.
;
; note:
;
;      Distance Travelled = Acc*T*T/2 + Vel*T  (note: T=1, so it goes away).
;      Acc = MoonsGravity + Fuel * AccPerUnitFuel
;
;      New Velocity = Acc*T + Prev Velocity
;
;      This code should really average these values over the one second
;      time period, but the simulation is so crude anyway, there's no
;      need to really bother.

ComputeStatus     proc
                  push     ax
                  push     bx
                  push     dx

; First, compute the acceleration value based on the fuel burnt
; during this second (Acc = Moon's Gravity + Fuel * AccPerUnitFuel).

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

                  add      ax, MoonsGravity        ;Add in Moon's gravity.
                  mov      bx, ax                  ;Save Acc value.

; Now compute the new velocity (V=AT+V)

                  add      ax, CurVel              ;Compute new velocity
                  mov      CurVel, ax

; Next, compute the distance travelled (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 Travelled.
                  neg      ax
                  add      CurDist, ax             ;New distance.

                  pop      dx
                  pop      bx
                  pop      ax
                  ret
ComputeStatus     endp

; GetYorN-        Reads a yes or no answer from the user (Y, y, N, or n).
;                 Returns the character read in the al register (Y or N,
;                 converted to upper case if necessary).

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    "Welcome to the moon lander game.",cr,lf,lf
                byte    "You must manuever your craft so that you touch"
                byte    "down at less than 10 ft/sec",cr,lf
                byte    "for a soft landing.",cr,lf,lf,0

                call    InitGame

; The following loop repeats while the distance to the surface is greater
; than zero.

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    "Your current velocity is %d.",cr,lf
                byte    "That was just a little too fast.  However, as a "
                byte    "consolation prize,",cr,lf
                byte    "we will name the new crater you just created "
                byte    "after you.",cr,lf,0
                dword   CurVel

                jmp     TryAgain

SoftLanding:    printf
                byte    "Congrats!  You landed the Lunar Module safely at "
                byte    "%d ft/sec.",cr,lf
                byte    "You have %d units of fuel left.",cr,lf
                byte    "Good job!",cr,lf,0
                dword   CurVel, FuelLeft

TryAgain:       print
                byte    "Do you want to try again (Y/N)? ",0
                call    GetYorN
                cmp     al, 'Y'
                je      MoonLoop

                print
                byte    cr,lf
                byte    "Thanks for playing!  Come back to the moon “
                byte    “again sometime"
                byte    cr,lf,lf,0

Quit:           ExitPgm                 ;DOS macro to quit program.
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
```

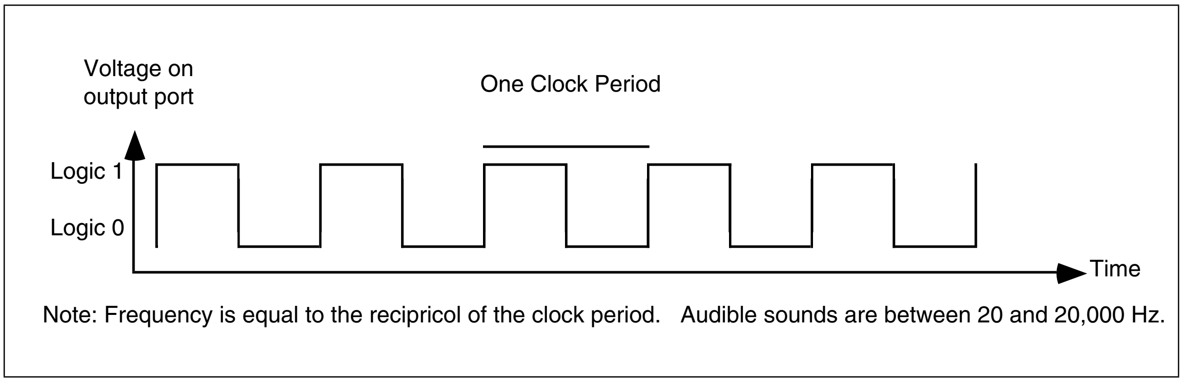
Voltage on
output port
One Clock Period
Logic 1
Logic 0
Time
Note: Frequency is equal to the recipricol of the clock period.   Audible sounds are between 20 and 20,000 Hz.

Figure 10.2 An Audible Sound Wave: The Relationship Between Period and Frequency

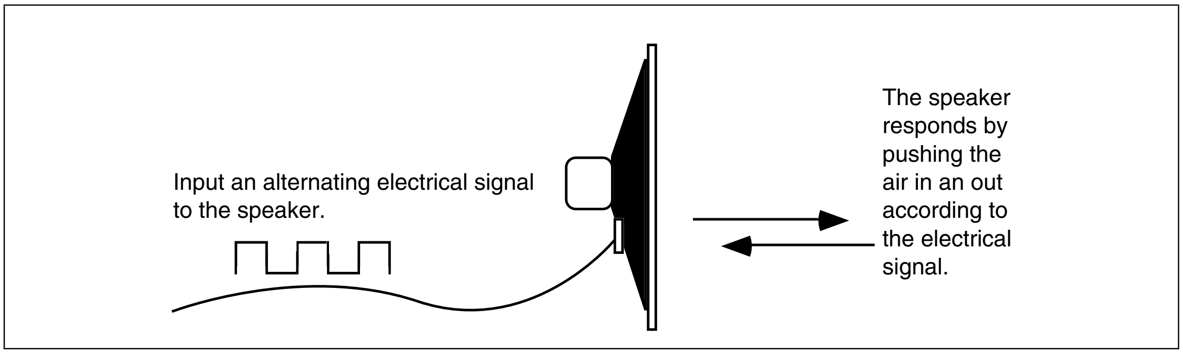

Figure 10.3 A Speaker

10.12 Laboratory Exercises

In this laboratory exercise you will program the timer chip on the PC to produce musical tones. You will learn how the PC generates sound and how you can use this ability to encode and play music.

10.12.1The Physics of Sound

Sounds you hear are the result of vibrating air molecules. When air molecules quickly vibrate back and forth between 20 and 20,000 times per second, we interpret this as some sort of sound. A *speaker* (see Figure 10.3) is a device which vibrates air in response to an electrical signal. That is, it converts an electric signal which alternates between 20 and 20,000 times per second (Hz) to an audible tone. Alternating a signal is very easy on a computer, all you have to do is apply a logic one to an output port for some period of time and then write a logic zero to the output port for a short period. Then repeat this over and over again. A plot of this activity over time appears in Figure 10.2.

Although many humans are capable of hearing tones in the range 20-20Khz, the PC's speaker is not capable of faithfully reproducing the tones in this range. It works pretty good for sounds in the range 100-10Khz, but the volume drops off dramatically outside this range. Fortunately, this lab only requires frequencies in the 110-2,000 hz range; well within the capabilities of the PC speaker.

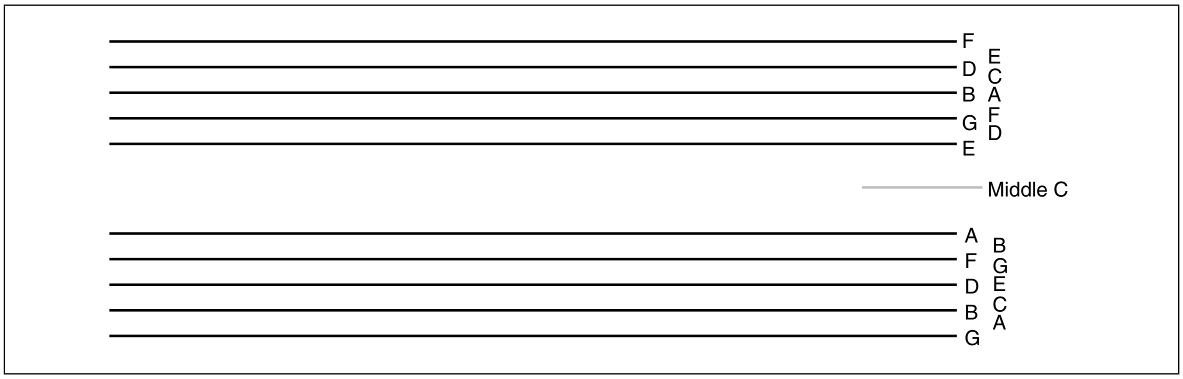

Figure 10.4 A Musical Staff

10.12.2 The Fundamentals of Music

In this laboratory you will use the timer chip and the PC's built-in speaker to produce musical tones. To produce true music, rather than annoying tones, requires a little knowledge of music theory. This section provides a very brief introduction to the notation musicians use. This will help you when you attempt to convert music in standard notation to a form the computer can use.

Western music tends to use notation based on the alphabetic letters A…G. There are a total of 12 notes designated A, A#, B, C, C#, D, D#, E, F, F#, G, and G# [10]. On a typical musical instrument these 12 notes repeat over and over again. For example, a typical piano might have six repetitions of these 12 notes. Each repetition is an *octave.* An octave is just a collection of 12 notes, it need not necessarily start with A, indeed, most pianos start with C. Although there are, technically, about 12 octaves within the normal hearing range of adults, very little music uses more than four or five octaves. In the laboratory, you will implement four octaves.

Written music typically uses two *staffs*. A staff is a set of five parallel lines. The upper staff is often called the *treble* staff and the lower staff is often called the *bass* staff. An examples appears in Figure 10.4.

A musical note, as the notation to the side of the staffs above indicates, appears both on the lines of the staffs and the spaces between the staffs. The position of the notes on the staffs determine which note to play, the *shape* of the note determines its duration. There are *whole* notes, *half* notes, *quarter* notes, *eighth* notes, *sixteenth* notes, and *thirty-second* notes[11]. Note durations are specified relative to one another. So a half note plays for one-half the time of a whole note, a quarter note plays for one-half the time of a half note (one quarter the time of a whole note), etc. In most musical passages, the quarter note is generally the basis for timing. If the *tempo* of a particular piece is 100 *beats per second* this means that you play 100 quarter notes per second.

The duration of a note is determined by its shape as shown in Figure 10.5.

In addition to the notes themselves, there are often brief pauses in a musical passage when there are no notes being played. These pauses are known as rests. Since there is nothing audible about them, only their duration matters. The duration of the various rests is the same as the normal notes; there are whole rests, half rests, quarter rests, etc. The symbols for these rests appear in .

This is but a brief introduction to music notation. Barely sufficient for those without any music training to convert a piece of sheet music into a form suitable for a computer

10. The notes with the "#" (pronounced *sharp*) correspond to the black keys on the piano. The other notes correspond to the white keys on the piano. Note that western music notation also describes *flats* in addition to sharps. A# is equal to B*b* (*b* denotes flat), C# corresponds to D*b*, etc. Technically, B is equivalent to C*b* and C is equivalent to B# but you will rarely see musicians refer to these notes this way.

11. The only reason their aren't shorter notes is because it would be hard to play one note which is 1/64th the length of another.

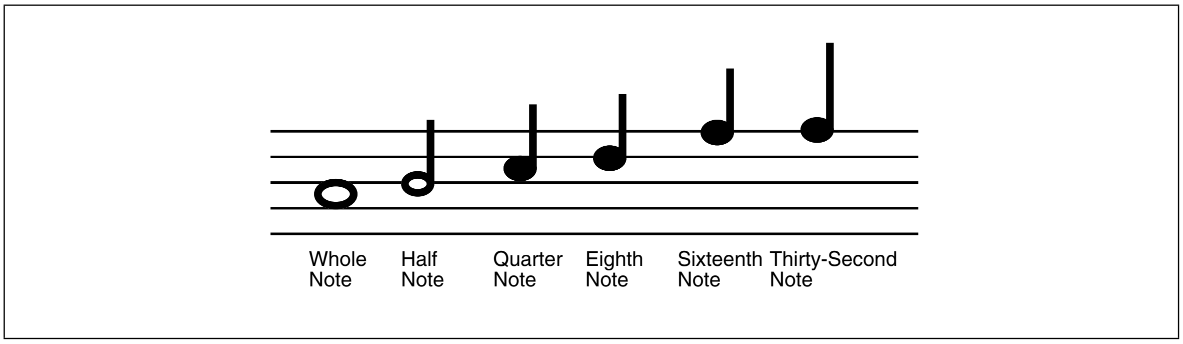

Figure 10.5 Note Durations

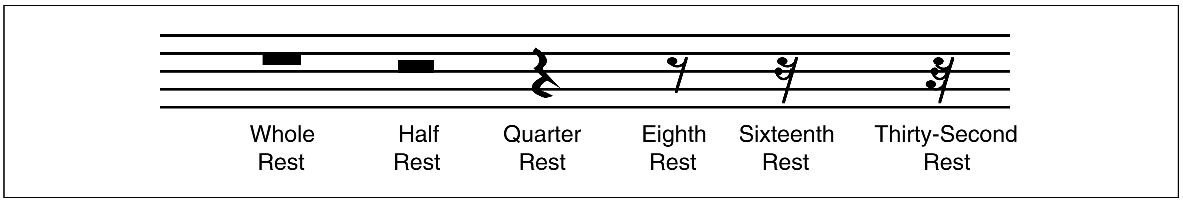

Figure 10.6 Rest Durations

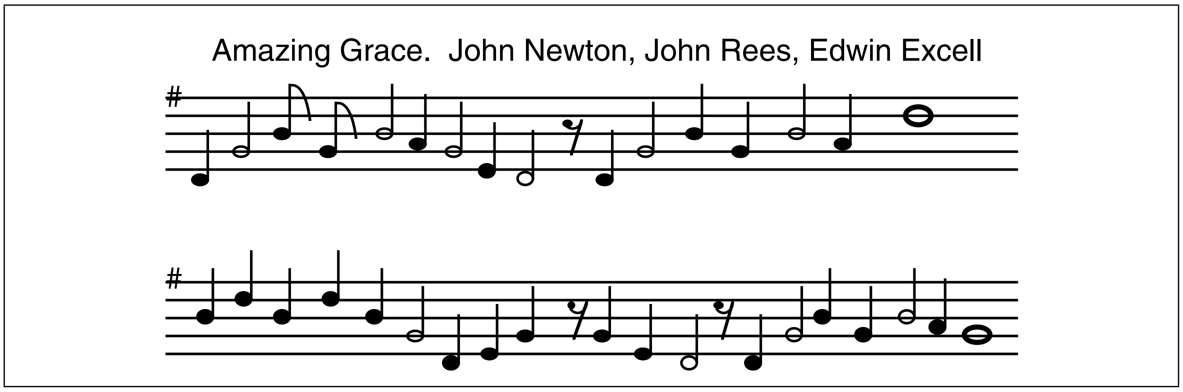

Figure 10.7 Amazing Grace

program. If you are interested in more information on music notation, the library is a good source of information on music theory.

Figure 10.7 provides an adaptation of the hymn "Amazing Grace". There are two things to note here. First, there is no bass staff, just two treble staffs. Second, the sharp symbol on the "F" line indicates that this song is played in "G-Major" and that all F notes should be F#. There are no F notes in this song, so that hardly matters[12].

10.12.3The Physics of Music

Each musical note corresponds to a unique frequency. The A above middle C is generally 440 Hz (this is known as concert pitch since this is the frequency orchestras tune to). The A one octave below this is at 220 Hz, the A above this is 880Hz. In general, to get the next higher A you double the current frequency, to get the previous A you halve the current frequency. To obtain the remaining notes you multiply the frequency of A with a multiple of the twelfth root of two. For example, to get A# you would take the frequency for A

12. In the full version of the song there are F notes on the base clef.

and multiply it by the twelfth root of two. Repeating this operation yields the following (truncated) frequencies for four separate octaves:

Note	Frequency	Note	Frequency	Note	Frequency	Note	Frequency
A 0	110	A 1	220	A 2	440	A 3	880
A # 0	117	A # 1	233	A # 2	466	A # 3	932
B 0	123	B 1	247	B 2	494	B 3	988
C 0	131	C 1	262	C 2	523	C 3	1047
C # 0	139	C # 1	277	C # 2	554	C # 3	1109
D 0	147	D 1	294	D 2	587	D 3	1175
D # 0	156	D # 1	311	D # 2	622	D # 3	1245
E 0	165	E 1	330	E 2	659	E 3	1319
F 0	175	F 1	349	F 2	698	F 3	1397
F # 0	185	F # 1	370	F # 2	740	F # 3	1480
G 0	196	G 1	392	G 2	784	G 3	1568
G # 0	208	G # 1	415	G # 2	831	G # 3	1661

Notes: The number following each note denotes its octave. In the chart above, middle C is C1.

You can generate additional notes by halving or doubling the notes above. For example, if you really need A(-1) (the octave below A0 above), dividing the frequency of A0 by two yields 55Hz. Likewise, if you want E4, you can obtain this by doubling E3 to produce 2638 Hz. Keep in mind that the frequencies above are not exact. They are rounded to the nearest integer because we will need integer frequencies in this lab.

10.12.4 The 8253/8254 Timer Chip

PCs contain a special integrated circuit which produces a period signal. This chip (an Intel compatible 8253 or 8254, depending on your particular computer[13]) contains three different 16-bit counter/timer circuits. The PC uses one of these timers to generate the 1/18.2 second real time clock mentioned earlier. It uses the second of these timers to control the DMA refresh on main memory[14]. The third timer circuit on this chip is connected to the PC's speaker. The PC uses this timer to produces beeps, tones, and other sounds. The RTC timer will be of interest to us in a later chapter. The DMA timer, if present on your PC, isn't something you should mess with. The third timer, connected to the speaker, is the subject of this section.

10.12.5Programming the Timer Chip to Produce Musical Tones

As mentioned earlier, one of the channels on the PC programmable interval timer (PIT) chip is connected to the PC's speaker. To produce a musical tone we need to program this timer chip to produce the frequency of some desired note and then activate the

13. Most modern computers don't actually have an 8253 or 8254 chip. Instead, there is a compatible device built into some other VLSI chip on the motherboard.

14. Many modern computer systems do not use this timer for this purpose and, therefore, do not include the second timer in their chipset.

speaker. Once you initialize the timer and speaker in this fashion, the PC will continuously produce the specified tone until you disable the speaker.

To activate the speaker you must set bits zero and one of the "B Port" on the PC's 8255 Programmable Peripheral Interface (PPI) chip. Port B of the PPI is an eight-bit I/O device located at I/O address 61h. You must use the in instruction to read this port and the out instruction to write data back to it. You must preserve all other bits at this I/O address. If you modify any of the other bits, you will probably cause the PC to malfunction, perhaps even reset. The following code shows how to set bits zero and one without affecting the other bits on the port:

```
in       al, PPI_B    ;PPI_B is equated to 61h
or       al, 3        ;Set bits zero and one.
out      PPI_B, al
```

Since PPI_B's port address is less than 100h we can access this port directly, we do not have to load its port address into dx and access the port indirectly through dx.

To deactivate the speaker you must write zeros to bits zero and one of PPI_B. The code is similar to the above except you force the bits to zero rather than to one.

Manipulating bits zero and one of the PPI_B port let you turn on and off the speaker. It does not let you adjust the frequency of the tone the speaker produces. To do this you must program the PIT at I/O addresses 42h and 43h. To change the frequency applied to the speaker you must first write the value 0B6h to I/O port 43h (the PIT *control word*) and then you must write a 16-bit frequency divisor to port 42h (timer channel two). Since the port is only an eight-bit port, you must write the data using two successive OUT instructions to the same I/O address. The first byte you write is the L.O. byte of the divisor, the second byte you write is the H.O. byte.

To compute the divisor value, you must use the following formula:

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

For example, the divisor for the A above middle C (440 Hz) is 1,193,180/440 or 2,712 (rounded to the nearest integer). To program the PIT to play this note you would execute the following code:

```
mov      al, 0B6h     ;Control word code.
out      PIT_CW, al   ;Write control word (port 43h).
mov      al, 98h      ;2712 is 0A98h.
out      PIT_Ch2, al  ;Write L.O. byte (port 42h).
mov      al, 0ah
out      PIT_Ch2, al  ;Write H.O. byte (port 42h).
```

Assuming that you have activated the speaker, the code above will produce the A note until you deactivate the speaker or reprogram the PIT with a different divisor.

10.12.6Putting it All Together

To create *music* you will need to activate the speaker, program the PIT, and then delay for some period of time while the note plays. At the end of that period, you need to reprogram the PIT and wait while the next note plays. If you encounter a rest, you need to deactivate the speaker for the given time interval. The key point is this *time interval*. If you simply reprogram the PPI and PIT chips at microprocessor speeds, your song will be over and done with in just a few microseconds. Far to fast to hear anything. Therefore, we need to use a delay, such as the software delay code presented earlier, to allow us to hear our notes.

A reasonable tempo is between 80 and 120 quarter notes per second. This means you should be calling the Delay18 routine between 9 and 14 times for each quarter note. A reasonable set of iterations is

- three times for sixteenth notes,
- six times for eighth notes,
- twelve times for quarter notes,
- twenty-four times for half notes, and
- forty-eight times for whole notes.

Of course, you may adjust these timings as you see fit to make your music sound better. The important parameter is the ratio between the different notes and rests, not the actual time.

Since a typical piece of music contains many, many individual notes, it doesn't make sense to reprogram the PIT and PPI chips individually for each note. Instead, you should write a procedure into which you pass a divisor and a count down value. That procedure would then play that note for the specified time and then return. Assuming you call this procedure *PlayNote* and it expects the divisor in ax and the duration (number of times to call Delay18) in cx , you could use the following macro to easily create songs in your programs:

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

The following macro lets you easily insert a rest into your music:

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

Now you can play notes by simply stringing together a list of these macros with the appropriate parameters.

The only problem with this approach is that it is different to create songs if you must constantly supply divisor values. You'll find music creation to be much simpler if you could specify the note, octave, and duration rather than a divisor and duration. This is very easy to do. Simply create a *lookup table* using the following definition:

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

Where Note is 'A';.."G", Sharp is true or false (1 or 0), and Octave is 0..3. Each entry in the table would contain the divisor for that particular note.

10.12.7 Amazing Grace Exercise

Program Ex10_1.asm on the companion CD-ROM is a complete working program that plays the tune "Amazing Grace." Load this program an execute it.

For your lab report: the Ex10_1.asm file uses a "Note" macro that is very similar to the one appearing in the previous section. What is the difference between Ex10_1's Note macro and the one in the previous section? What changes were made to PlayNote in order to accommodate this difference?

The Ex10_1.asm program uses *straight-line code* (no loops or decisions) to play its tune. Rewrite the main body of the loop to use a pair of tables to feed the data to the Note and Rest macros. One table should contain a list of frequency values (use -1 for a rest), the other table should contain duration values. Put the two tables in the data segment and initialize them with the values for the Amazing Grace song. The loop should fetch a pair of values, one from each of the tables and call the Note or Rest macro as appropriate. When the loop encounters a frequency value of zero it should terminate. **Note:** you must call the rest macro at the end of the tune in order to shut the speaker off.

For your lab report: make the changes to the program, document them, and include the print-out of the new program in your lab report.

10.13 Programming Projects

1) Write a program to transpose two 4x4 arrays. The algorithm to transpose the arrays is

```
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;
```

Write a main program that calls a transpose procedure. The main program should read the A array values from the user and print the A and B arrays after computing the transpose of A and placing the result in B.

2) Create a program to play music which is supplied as a string to the program. The notes to play should consist of a string of ASCII characters terminated with a byte containing the value zero. Each note should take the following form:

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

where “Note” is A..G (upper or lower case), “Octave” is 0..3, and “Duration” is 1..8. “1” corresponds to an eighth note, “2” corresponds to a quarter note, “4” corresponds to a half note, and “8” corresponds to a whole note.

Rests consist of an explanation point followed by a “Duration” value.

Your program should ignore any spaces appearing in the string.

The following sample piece is the song “Amazing Grace” presented earlier.

```
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
```

Write a program to play any song appearing in string form like the above string. Using music obtained from another source, submit your program playing that other song.

3) A *C character string* is a sequence of characters that end with a byte containing zero. Some common character string routines include computing the length of a character string (by counting all the characters in a string up to, but not including, the zero byte), comparing two strings for equality (by comparing corresponding characters in two strings, character by character until you encounter a zero byte or two characters that are not the same), and copying one string to another (by copying the characters from one string to the corresponding positions in the other until you encounter the zero byte). Write a program that reads two strings from the user, computes the length of the first of these, compares the two strings, and then copies the first string over the top of the second. Allow for a maximum of 128 characters (including the zero byte) in your strings. Note: do not use the Standard Library string routines for this project.

4) Modify the moon lander game appearing in the Sample Programs section of this chapter (moon.asm on the companion CD-ROM, also see “Sample Program” on page 547) to allow the user to specify the initial velocity, starting distance from the surface, and initial fuel values. Verify that the values are reasonable before allowing the game to proceed.

10.14 Summary

This chapter discussed the implementation of different control structures in an assembly language programs including conditional statements (if..then..else and case statements), state machines, and iterations (loops, including while, repeat..until (do/while), loop..endloop, and for). While assembly language gives you the flexibility to create totally custom control structures, doing so often produces programs that are difficult to read and understand. Unless the situation absolutely requires something different, you should attempt to model your assembly language control structures after those in high level languages as much as possible.

The most common control structure found in high level language programs is the IF..THEN..ELSE statement. You can easily synthesize(if..then and (if..then..else statements in assembly language using the cmp instruction, the conditional jumps, and the jmp instruction. To see how to convert HLL if..then..else statements into assembly language, check out

- "IF..THEN..ELSE Sequences" on page 522

A second popular HLL conditional statement is the case (switch) statement. The case statement provides an efficient way to transfer control to one of many different statements depending on the value of some expression. While there are many ways to implement the case statement in assembly language, the most common way is to use a *jump table*. For case statements with contiguous values, this is probably the best implementation. For case statements that have widely spaced, non-contiguous values, an if..then..else implementation or some other technique is probably best. For details, see

- "CASE Statements" on page 525

State machines provide a useful paradigm for certain programming situations. A section of code which implements a state machine maintains a history of prior execution within a state variable. Subsequent execution of the code picks up in a possibly different "state" depending on prior execution. Indirect jumps provide an efficient mechanism for implementing state machines in assembly language. This chapter provided a brief introduction to state machines. To see how to implement a state machine with an indirect jump, see

- "State Machines and Indirect Jumps" on page 529

Assembly language provides some very powerful primitives for constructing a wide variety of control structures. Although this chapter concentrates on simulating HLL constructs, you can build any convoluted control structure you care to from the 80x86's cmp instruction and conditional branches. Unfortunately, the result may be very difficult to understand, especially by someone other than the original author. Although assembly language gives you the freedom to do anything you want, a mature programmer exercises restraint and chooses only those control flows which are easy to read and understand; never settling for convoluted code unless absolutely necessary. For a further description and additional guidelines, see

- "Spaghetti Code" on page 531

Iteration is one of the three basic components to programming language built around Von Neumann machines[15]. Loop control structures provide the basic iteration mechanism in most HLLs. Assembly language does not provide any looping primitives. Even the 80x86 loop instruction isn't really a loop, it's just a decrement, compare, and branch instruction. Nonetheless, it is very easy to synthesize common loop control structures in assembly language. The following sections describe how to construct HLL loop control structures in assembly language:

- "Loops" on page 531
- "While Loops" on page 532
- "Repeat..Until Loops" on page 532

15. The other two being conditional execution and the sequence.

- “LOOP..ENDLOOP Loops” on page 533
- “FOR Loops” on page 533

Program loops often consume most of the CPU time in a typical program. Therefore, if you want to improve the performance of your programs, the loops are the first place you want to look. This chapter provides several suggestions to help improve the performance of certain types of loops in assembly language programs. While they do not provide a complete guide to optimization, the following sections provide common techniques used by compilers and experienced assembly language programmers:

- “Register Usage and Loops” on page 534
- “Performance Improvements” on page 535
- “Moving the Termination Condition to the End of a Loop” on page 535
- “Executing the Loop Backwards” on page 537
- “Loop Invariant Computations” on page 538
- “Unraveling Loops” on page 539
- “Induction Variables” on page 540
- “Other Performance Improvements” on page 541

10.15 Questions

1) Convert the following Pascal statements to assembly language: (assume all variables are two byte signed integers)

   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) IF (X=0) and ((Y-2) > 1) then Y := Y - 1;

2) Convert the following CASE statement to assembly language:

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

3) Which implementation method for the CASE statement (jump table or IF form) produces the least amount of code (including the jump table, if used) for the following CASE statements?

   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) For question three, which form produces the fastest code?

5) Implement the CASE statements in problem three using 80x86 assembly language.

6) What three components compose a loop?

7) What is the major difference between the WHILE, REPEAT..UNTIL, and LOOP..ENDLOOP loops?

8) What is a loop control variable?

9) Convert the following WHILE loops to assembly language: (Note: don't optimize these loops, stick exactly to the WHILE loop format)

   a) I := 0;

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

   b) CH := ' ';

      WHILE (CH <> '.') DO BEGIN

         CH := GETC;

         PUTC(CH);

      END;

10) Convert the following REPEAT..UNTIL loops into assembly language: (Stick exactly to the REPEAT..UNTIL loop format)

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

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

11) Convert the following LOOP..ENDLOOP loops into assembly language: (Stick exactly to the LOOP..ENDLOOP format)

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) What are the differences, if any, between the loops in problems 4, 5, and 6? Do they perform the same operations? Which versions are most efficient?

13) Rewrite the two loops presented in the previous examples, in assembly language, as efficiently as you can.

14) By simply adding a JMP instruction, convert the two loops in problem four into REPEAT..UNTIL loops.

15) By simply adding a JMP instruction, convert the two loops in problem five to WHILE loops.

16) Convert the following FOR loops into assembly language (Note: feel free to use any of the routines provided in the UCR Standard Library package):

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) The DOWNTO reserved word, when used in conjunction with the Pascal FOR loop, runs a loop counter from a higher number down to a lower number. A FOR loop with the DOWNTO reserved word is equivalent to the following WHILE loop:

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

Implement the following Pascal FOR loops in assembly language:

a) FOR I := start downto stop do WriteLn(I);

b)
```
FOR I := 7 downto 0 do
        FOR J := 0 to 7 do
        K := K*(I-J);
c) FOR I := 255 downto 16 do
        A [I] := A[240-I]-I;
```

18) Rewrite the loop in problem 11b maintaining I in BX, J in CX, and K in AX.

19) How does moving the loop termination test to the end of the loop improve the performance of that loop?

20) What is a loop invariant computation?

21) How does executing a loop backwards improve the performance of the loop?

22) What does unraveling a loop mean?

23) How does unraveling a loop improve the loop's performance?

24) Give an example of a loop that cannot be unraveled.

25) Give an example of a loop that can be but shouldn't be unraveled.