With the 80x86's wide variety of control transfer instructions you can easily generate any program flow you need. Too easily, in fact. While assembly language provides the tools to let you create a brand new control structure "on the spot," an undisciplined approach can lead to programs which are nearly impossible to read and understand.

High level language designers realized this long ago and invented new languages with restricted control abstractions to force this discipline. This, combined with the field of Software Engineering and the rise of *structured programming* provided dramatic increases in the quality of software over the past two decades. Today, structured programming techniques are taken for granted by most professional programmers.

Unfortunately, assembly language is still stuck in the dark ages with respect to control structures. While many high level languages today have omitted the GOTO statement in favor of structured IF..THEN..ELSE, CASE, REPEAT..UNTIL, and WHILE, assembly still provides little more than conditional and unconditional jumps. Since you can synthesize almost *any* control structure from these conditional and unconditional jumps, you might believe that you should use these statements to their full potential. However, as mentioned above, doing so can destroy the readability of your code. Therefore, you should stick to synthesizing those same HLL control flow constructions so that your programs will be easier to read.

In this chapter you will explore the synthesis of high level language constructs in assembly language and how they improve the readability of your programs. You will also study the costs associated with structured programming. In this laboratory you will also begin studying some "real world" applications and put what you've learned to work. In particular, you will write some code to generate music using the built-in speaker in your PC.

8.1 Decisions with the IF..THEN Statement

The most fundamental unit of execution on a typical Von Neumann machine is the *sequence*- a list of instructions which the machine executes one after another. The second most common execution unit is the *conditional*- execution (or not) of a statement based upon some pre-existing condition. In modern HLLs, the most commonly used conditional statement is the IF..THEN..ELSE statement. Since most programmers learning assembly language for the first time are quite familiar with this statement, it makes a good starting point for the discussion of conditional execution.

You can actually use any of the 80x86's flow of control instructions (jumps, calls, returns, interrupts, etc.) as conditional instructions. Most of the time, however, you will probably use the 80x86 conditional jump instructions to conditionally execute some sequence of instructions. Unfortunately, the loose syntax of the conditional jump instructions provide you with an overwhelming number of ways to accomplish the task at hand. While they are certainly more flexible than HLL alternatives, this flexibility has a cost: the number of alternative solutions is greater (and therefore more confusing to a beginning) and, therefore, there are many different solutions to the same problem which different programmers use. One programmer's solution might be completely different from another's. As a result, they will have difficulty understanding each other's code. We're not talking about implementing some obscure or complex algorithm here; we're talking about implementing an IF..THEN..ELSE statement in assembly language!

The best solution for beginning assembly language programmers is to personally enforce some structure where MASM does not. If you define your conditional operations in terms of high level IF..THEN..ELSE statements and then convert them to assembly language using a *template driven* approach, you should have no problems writing conditional statements in assembly language nor will you or your colleagues have trouble understanding the result. Now it is quite true that using this template driven approach may producing code which is larger and slower than it needs to be, however now is not the time to worry about efficiency. Once you are very comfortable with assembly language you can worry a little more about the speed of your programs.

8.1 What are the two problems beginners face with the flexibility of assembly language conditional statements? _____

_____

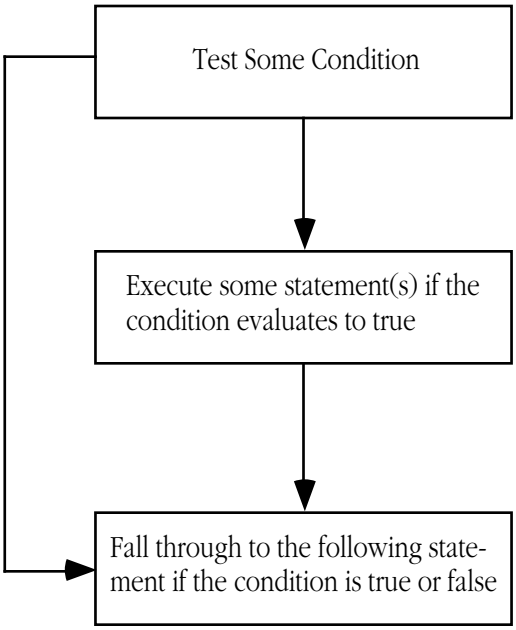
_____

The IF..THEN statement is an important special case of the IF..THEN..ELSE. Since the IF..THEN is somewhat easier to implement and understand, its implementation is a good starting point. Consider the IF..THEN statement in Pascal:

```
IF boolean_expression THEN statement
```

We can diagram this as follows:

IF..THEN Structure:



The implementation of this flow diagram in assembly language is surprisingly similar to the computation of boolean expressions in the last chapter. Indeed, about the only difference is that you execute the specified statements rather than set some variable to zero or one.

One major advantage to HLLs is that it is often easier to read and understand a conditional expression than it is to read and understand the same condition in assembly language. Therefore, it makes sense to organize your assembly expressions to make them easier to read and understand.

8.2 What condition does the Pascal statement "IF (X = Y) and (A < B) then Writeln('True'):" test?

_____

_____

8.3 What condition does the following assembly code test?

```
        mov     ax, X
        cmp     ax, Y
        jne     DontDoIt
        mov     ax, A
        cmp     ax, B
        jng     DontDoIt
        print
        byte    "TRUE",cr,lf,0
DontDoIt:
```

_____

_____

8.1 The number of alternative solutions is overwhelming and different programmers choose different solutions to a problem.

8.4 Which of the above two code sequences do you believe is easier to read and understand? _____

Since conditional statements are a little more difficult to read and understand in assembly language, you have to work a little harder to clarify your code. Without question, the most important thing you can do to improve the readability of your conditional sequences is to place a comment before each code section which describes the test you are performing. Consider the code from question 8.3, rewritten with a *single comment*:

```
; IF (X = Y) and (A > B) then print "TRUE"

                mov     ax, X
                cmp     ax, Y
                jne     DontDoIt
                mov     ax, A
                cmp     ax, B
                jng     DontDoIt
                print
                byte    "TRUE",cr,lf,0
DontDoIt:
```

Under normal circumstances, this single comment would be insufficient. It would be a good idea to explain *why* you are checking to see if X equals Y and A is greater than B. However, this single comment goes a long way towards demystifying the code which immediately follows it. As a general rule, you should always place such a comment in your programs before any implementation of an IF..THEN statement.

8.5 What comment should precede the following assembly code?

```
                mov     ax, I
                cmp     ax, J
                jne     Done
                print
                byte    "Yes",cr,lf,0
Done:
```

_____

_____

As the code in the previous diagram, example, and questions indicates, a typical implementation of an IF..THEN statement consists of some code to test the desired condition. If the condition is *false*, the code *jumps over* the statements to execute when the condition is true. The true sequence (those statements following THEN in the IF statement) merges together with the false flow immediately afterwards.

Do not get the idea that this is the *only* way to deal with a conditional statement. There are many other possible ways to implement the IF..THEN statement. Consider the following code:

```
; IF I = J THEN print "Yes"

                mov     ax, I
                cmp     ax, J
                jne     SomeWhere
                print
                byte    "Yes",cr,lf,0
                jmp     SomeWhere
Here:
```

The false and true code streams will merge back together at the "SomeWhere" label. Unfortunately, "SomeWhere" does not immediately follow the true sequence which tends to confuse those who would normally expect this. Even though the following code is slightly less efficient (because the false control flow executes two jumps instead of one), it is a better implementation

of the above code because most programmers reading your code will find it easier to read and understand:

```
; IF I = J THEN print "Yes"

                mov     ax, I
                cmp     ax, J
                jne     ItsFalse
                print
                byte    "Yes",cr,lf,0
ItsFalse:
                jmp     SomeWhere
```

There is one additional benefit to organizing your code in this fashion: it will be much easier to maintain. Suppose, for example, that you wanted to print the string "Here I Am" immediately after the IF in both the true and false paths. In a high level language like Pascal you would simple stick the output statement immediately after the IF..THEN statement. In the example immediately above, you could stick a call to PRINT immediately before the JMP instruction. In the previous example, however, you would have to either rewrite the code or go place the print statement at the SomeWhere label. Neither of these last two alternatives is good if you want the new PRINT statement near the conditional sequence.

You must also pay careful attention to the statements in the true sequence if you want your code to be readable. In general, all statements which belong to the true sequence should immediately follow the condition test. This might seem obvious, but assembly language code does not require this to be true. Consider the following code sequence:

```
; IF (X = Y) THEN BEGIN
;         A := B;
;         X := X + 1;
;         Y := Y - 1;
; END;

                mov     ax, X
                cmp     ax, Y
                jne     ItsFalse
                mov     ax, B
                mov     A, ax
                jmp     IncX
FinishTrue:     dec     Y
ItsFalse:
                 .
                 .
                 .
IncX:           inc     X
                jmp     FinishTrue
```

Of course, this example is especially trivial and it would be hard to imagine someone actually writing code in this fashion but it is not that hard to create a more complex example where you would be tempted to interrupt the flow of control in exactly this manner. Note, by the way, that the code above is not equivalent to a procedure call. Had the JMP to IncX been replaced with a procedure call, it would have been perfectly obvious to the casual reader that control would resume at the DEC instruction. This is not at all clear in the above code. Should there be a reason why you *have* to write code in this manner, you should place appropriate comments in the code to let the reader know exactly what is going to happen:

```
                mov     ax, X
                cmp     ax, Y
                jne     ItsFalse
                mov     ax, B
                mov     A, ax
                jmp     IncX            ;IncX jumps back to FinishTrue.
FinishTrue:     dec     Y
ItsFalse:
```

Although the comment above helps the reader understand what is going on, any code written in this fashion will come under considerable criticism by most programmers and your reputation will be hurt.

This is not to imply that you can have no transfer of control instructions inside the true sequence. There may be other IFs or loops within the true sequence. What you need to avoid is jumping in and out of the true sequence (procedure calls are an obvious exception to this rule).

As mentioned earlier, the assembly language code you use to implement the condition test is very similar to the code you use to compute a boolean expression. You can use short-circuit evaluation, optimization via DeMorgan's theorems, and all the other techniques presented in the last chapter to test the boolean expression. The only component of these test worth further discussion at this point is the conditional jump which separates the true and false flow of control paths. Since there are some many synonyms for the various conditional jump instructions, you should adopt a strict convention concerning their use. By doing so, you can make your programs easier to read by other programmers who adopt the same convention.

8.2 (X=Y) and (A<B)

Consider the following Pascal statement:

```
IF ( X < Y ) THEN writeln('Hello');
```

8.3 (X=Y) and (A>B)

You can implement this in assembly language in at least two different ways:

```
; Method 1:
                mov     ax, X
                cmp     ax, Y
                jge     Done
                print
                byte    "Hello",cr,lf,0
Done:


; Method 2:
                mov     ax, X
                cmp     ax, Y
                jnl     Done
                print
                byte    "Hello",cr,lf,0
Done:
```

8.4 Obviously the Pascal code is easier to understand.

Since JGE and JNL are synonyms for the same instruction, the two code sequences above produce exactly the same object code.

Since these two sequences produce the same object code we cannot claim one version is better because it is faster or shorter. Since they are equally efficient, the other important question to ask is "Which is easier to read and understand?" There is a very good argument that the second version is the easier of the two to understand. Assuming you've followed the advice presented earlier of placing the comment "; IF (X<Y) THEN writeln('Hello');" before the above two sequences, most people will have an easier time correlating "X < Y" with the JNL instruction than they will with the "JGE" instruction. While it's true that JNL is opposite of "X<Y", it is much easier to forget the "not" in the instruction than to perform the mental gymnastics necessary to get from "jump if greater or equal" to "X < Y." Therefore, when faced with choosing an appropriate instruction for a conditional test, you should always choose the conditional jump which is the negation of the condition you want to test. You should read such a sequence as "jump around the true sequence if the condition is false."

8.5 "IF I=J then print YES"

8.6 Convert "IF X >=Y THEN writeln('X>=Y');" to assembly language using the above convention:

_____ _____

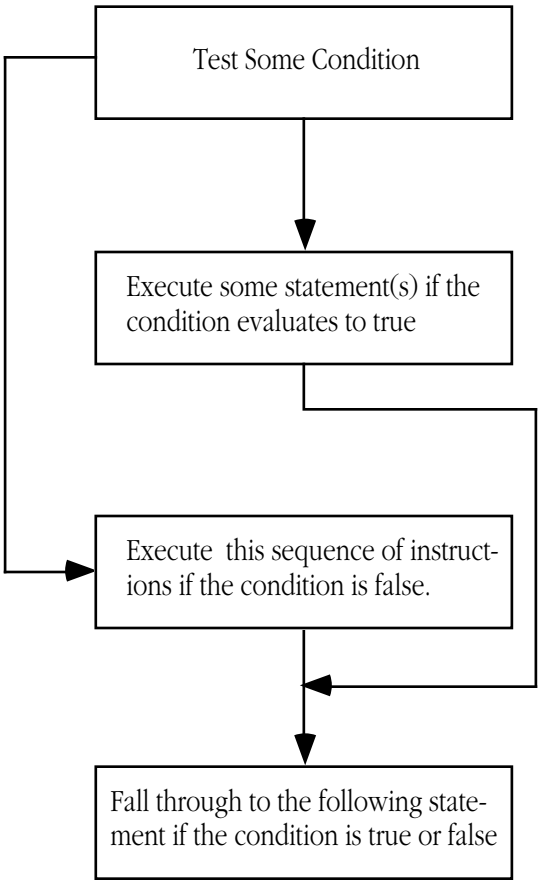
_____ _____

8.2 Decisions with the IF..THEN..ELSE Statement

The IF..THEN..ELSE statement requires only a minor modification to the IF..THEN statement described in the previous section. Rather than having the false flow of control merge back with the true flow of control at the end of the true sequence, the false flow transfers to its own sequence of instructions.

By convention, most assembly language programmers organize their IF..THEN..ELSE statements with the true sequence immediately following the conditional test and the false sequence (the ELSE section) following the true sequence. Since the true sequence cannot execute the statements in the false sequence, there is usually a jump at the end of the true sequence which skips to the end of the false code. Pictorially, this appears as follows:

IF..THEN..ELSE Structure:



Example:

```
; if (X <> Y) then write('Yes') else write('no');

                mov     ax, X
                cmp     ax, Y
                je      ElsePart
                print
                byte    "Yes",0
                jmp     IfDone

ElsePart:       print
                byte    "no",0
IfDone:
```

As with the simple IF..THEN translations, you really want to keep the true and false sequences together and in the order shown above. Except for procedure calls and the terminating JMP on the true sequence, you should avoid any instructions which transfer control out of the true or false sequences.

8.3 CASE Statements

Most HLLs provide some form of multiway branch capability. The CASE statement in Pascal, the SWITCH statement in C, ON..GOTO in BASIC, the computed GOTO in FORTRAN, are some examples of such statements. Although a compiler writer is free to implement these control structures in whatever manner they like, most compilers use a *jump table* to implement a multiway conditional test.

In assembly language, implementing a multiway test is very simple. You create an array of addresses, select an element of that array at run time, and jump to the selected address. Each address in the array must point at a sequence of instructions associated with the index into the array of addresses. To execute the selection operation you use the value of the CASE expression as an index into the array, fetch the specified address, and jump to it:

```
                mov     bx, CaseValue
                shl     bx, 1
                mov     bx, cs:Array[bx]
                jmp     bx
Array           word    Stmt0, Stmt1, Stmt2, ...
```

CaseValue represents the ultimate result of a CASE expression. The SHL instruction multiplies BX by two since Array is an array of words. The "MOV BX, cs:Array[BX]" instruction loads BX with the desired element of the array. The JMP instruction transfer control to this target location. Note that the last two instructions are written this way to emphasize the array access. In fact, you could combine the last to instructions into a single instruction: "JMP cs:Array[BX]".

8.7 If you wanted to use double word addresses rather than word addresses, how would you rewrite the code above?

_____

_____

_____

_____

As you might expect from the examples presented in the sections on IF..THEN and IF..THEN..ELSE, you can organize the code for a CASE statement in many different ways. However, some forms are better than others if you care about readable code. In general, suppose you have a CASE statement in Pascal which takes the form:

```
case i of
      0: stmt0;
      1: stmt1;
      2: stmt2;
          .
          .
          .
      n: stmtn;
      otherwise stmtn+1;
end;
```

Given this arrangement you should organize the assembly code in a similar fashion. That is, your assembly code should begin with the indirect jump code and the table of addresses. Following this should be the code for $STMT_0$ followed by a jump to the first statement beyond the OTHERWISE sequence. Converted to assembly code, the above sequence, sans the otherwise clause, might look like the following:

```
                mov     bx, i
                shl     bx, 1
                jmp     cs:CaseTbl[bx]
CaseTbl         word    Stmt0, Stmt1, Stmt2, ..., Stmtn
```

8.6

```
mov   ax, X
cmp   ax, Y
jnge  Done
print
byte  "X>-Y"
byte  cr,lf,0
Done:
```

```
Stmt0:          <Code for Stmt0 goes here>
                jmp      CaseDone

Stmt1:          <Code for Stmt1 goes here>
                jmp      CaseDone

Stmt2:          <Code for Stmt2 goes here>
                jmp      CaseDone
                 .
                 .
                 .
Stmtn:          <Code for Stmtn goes here>
CaseDone:
```

Before considering the OTHERWISE section of the case statement, there are two important details to discuss in the above code. First, note the "CS:" prefix on the CaseTbl label in the JMP instruction. In general, it's best to place the jump table immediately after the JMP instruction so other can easily determine the destination targets of the jump. Since this is in the code segment and DS generally won't be pointing at your code segment, you may need to use the CS: override. Of course, you could place the CaseTbl array in the data segment and avoid the cost of the segment override prefix, but that introduces problems of its own. For readability and other reasons you should generally try to place the array of target addresses after the JMP instruction. Hence you will typically have a "CS:" prefix on the array name[1]

A second problem with the CASE statement occurs if the case value is out of range. *Standard* Pascal claims the result is undefined if "i" in the previous CASE statement does not have a corresponding case value. The reason is obvious when you look at the jump table implementation of a CASE statement. If there are only cases 0, 1, 2, and 3 in the case statement and the case value is four, the jump table code will jump to the address specified by the two bytes just beyond the end of the table. This is exactly what the Pascal standard means when it says the result is undefined. Who knows what the value of those two bytes are going to be?

The OTHERWISE clause is an extension to Pascal to handle this problem. If the value of the CASE expression does not appear as a case label, most modern Pascals will do one of two things, they will either continue execution with the first statement beyond the CASE's END or they will transfer control to the statement after the OTHERWISE clause, if present. There are three possibilities the CASE statement must handle- the case value is less than the smallest case label, the case value is larger than the largest case label, or the case value is a "hole" in the list of case values. A simple IF..THEN test before the CASE can handle the first two cases, filling in the holes in the jump table with the address of the otherwise clause handles the last case. As an example, consider the following Pascal code:

```
case i of
      3: write(n);
      5: write(i);
      7: write(j);
      9: write(k);
      otherwise writeln('error');
end;
```

The assembly code you could use for this might be:

```
                mov      bx, i
                cmp      bx, 3
                jl       Otherwise
                cmp      bx, 9
                jg       Otherwise
                shl      bx, 1
                jmp      cs:jmptbl-6[bx]
jmptbl          word     Stmt3, Otherwise, Stmt5, Otherwise, Stmt7
                word     Otherwise, Stmt9

Stmt3:          mov      ax, n
                puti
                jmp      CaseDone
```

1. Actually, if you have an appropriate ASSUME directive, the "CS:" prefix is not necessary, the assembler will automatically add one for you. However, it's generally a good idea to explicitly place this segment prefix in your code.

```
Stmt5:          mov       ax, i
                puti
                jmp       CaseDone

Stmt7:          mov       ax, j
                puti
                jmp       CaseDone

Stmt9:          mov       ax, k
                puti
                jmp       CaseDone

Otherwise:      print
                byte      "Error",cr,lf,0
CaseDone:
```

8.7

```
mov   bx, CaseVal
shl   bx, 2
jmp   cs:Array
Array dword Stmt0
      dword Stmt1
       .
       .
       .
```

8.8 What would be wrong with implementing a CASE statement like the following using the above technique?

case i of

1: write(i);

100: writeln('2');

1000: writeln('3');

otherwise writeln('Not 1, 100, or 1000.');

end;

_____

_____

_____

The big advantage of the jump table implementation for a CASE statement is that it executes in a constant amount of time, regardless of the number of case labels or the value of the case expression. As the question above demonstrates, however, there may be a problem with the amount of memory that this implementation uses. Fortunately, you can substitute an IF..THEN..ELSE..IF... construct if the jump table grows too large.

8.4 Loops

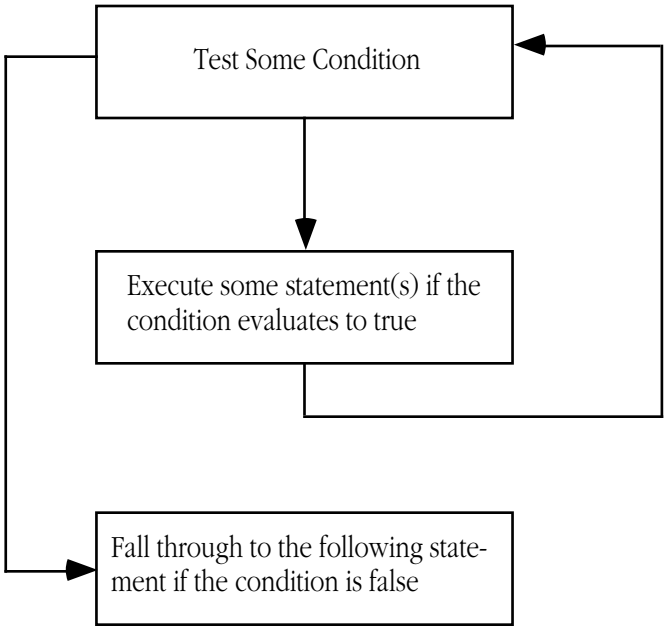
After the conditional sequences, loop sequences are the most common in a program. In various high level languages there are three generic types of loops- those with a test at the beginning, in the middle, and at the end of the loop. Following the Pascal/Ada/Modula 2 lead we will refer to these as WHILE loops, LOOP..ENDLOOPs, and REPEAT..UNTIL loops. A fourth common loop construct appearing in HLLs, the FOR loop, is really a special case of the WHILE loop[2].

To create a loop in an assembly language program is very easy. Any sort of jump instruction which transfers control to a part of your program which you've already executed can form a loop. As with the IF..THEN..ELSE and CASE statements, however, it makes a lot of sense to simulate as closely as possible the loop constructs found in HLLs.

The WHILE loop consists of a termination test, a loop body, and a JMP back to the termination test. Graphically, the WHILE loop looks like the following:

2. There are other types of loop constructs as well. For example, in Chapter Nine you will learn about iterators and the FOREACH loop.

While Structure:



As you can see from the diagram, this is simply an IF..THEN structure with a JMP following the true sequence which takes you back to the conditional test. The following is an example of a Pascal WHILE loop and the corresponding assembly code:

```
          while (X < Y) and (A = B) do begin
                 .
                 .
                 .
                X := X + 1;
          end;


WhileLp:        mov       ax, X
                cmp       ax, Y
                jnl       WhlDone
                mov       ax, A
                cmp       ax, B
                jne       WhlDone
                 .
                 .
                 .
                inc       X
                jmp       WhlLoop
WhlDone:
```

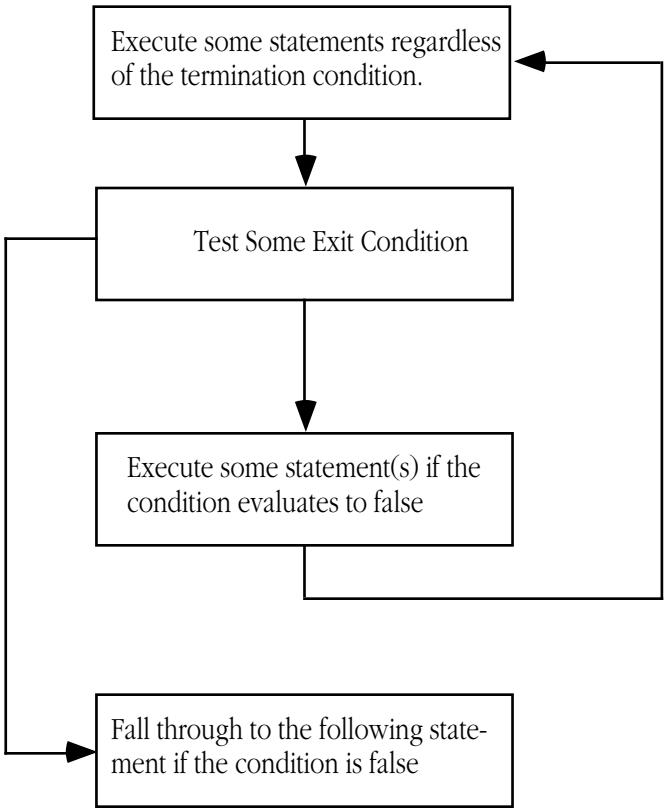
8.9 There is only one instruction in the above sequence which makes this sequence different from the IF..THEN sequence. What is that statement?

_____

The LOOP..ENDLOOP construct is very similar to the WHILE loop except that there are usually statements in the loop prior to the termination test. In the code above, this would correspond to some statements between the WhileLp label and the "MOV AX, X" instruction. Graphically, it looks like the following:

8.8 The resulting table would be too large.

LOOP..ENDLOOP Structure:



8.10 In terms of the assembly code you would write for a LOOP..ENDLOOP structure, what is the difference between a WHILE loop and a LOOP..ENDLOOP? _____

_____

_____

_____

_____

The following short pseudo-Pascal segment and accompanying assembly code demonstrate the translation of a LOOP..ENDLOOP statement:

```
LOOP
      write('Enter an integer (0 quits):');
      readln(i);
      if i=0 then break; {break means exit loop}
      writeln('You entered ',i);
ENDLOOP;
```

```
LoopEntry:      print
                byte    "Enter an integer (0 quits):",0
                getsm                       ;Read a line of text
                atoi                        ;Convert to an integer
                free                        ;Free memory malloc'd by GETSM

                cmp     ax, 0               ;Termination test.
                je      LoopDone

                print
                byte    "You entered ",0
                puti
                putcr
                jmp     LoopEntry
LoopDone:
```

The LOOP..ENDLOOP construct is the most general. You can construct any of the other three loops (plus the Pascal FOR loop) using nothing more than a LOOP..ENDLOOP. To create a WHILE loop, for example, you would simply put the termination test at the beginning of the loop.

8.11 How would you create a REPEAT..UNTIL loop using the LOOP..ENDLOOP construct?

_____

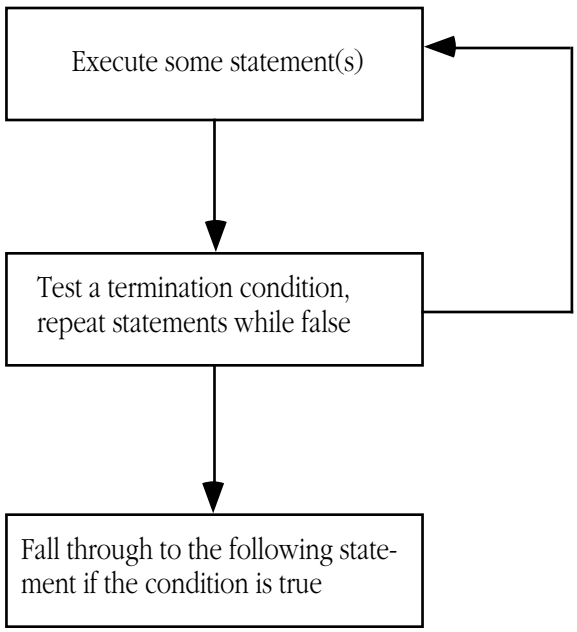
_____

_____

_____

The REPEAT..UNTIL loop turns out to be the easiest loop construct to generate in assembly language. Graphically, it takes the following form:

Repeat..Until Structure:



The simplification which exists here is the fact that the loop termination test, which occurs at the bottom of the REPEAT..UNTIL loop replaces the extra JMP instruction which is necessary for the WHILE and LOOP..ENDLOOP statements. Since there is one less JMP instruction in the loop, the code runs marginally faster. For this reason, many programmers attempt to convert all loops to REPEAT..UNTIL loops. The basic idea is outlined in the textbook and will not be repeated here since we're more interested in function rather than efficiency at this point.

As with the IF..THEN {..ELSE} and CASE statements, there are a number of ways you can implement these three types of loops in an assembly language program. However, if you stick to the templates given for the above three types of loops, you will find your programs are much easier to read, maintain, and debug. You should strive to maintain a single entry point and a single exit point in all your control structures. This does not necessarily produce the fastest or the shortest code. However, you should learn to write *correct* code before you worry about writing *fast* or *short* code. Once you've gotten your program(s) to work, you can worry about performance or size problems.

8.9 The "JMP WhlLoop" instruction at the end of the loop code.

8.10 Only the position of the loop termination code.

8.5 FOR Loops

Although the FOR loop is really nothing more than a special case of the WHILE loop, it is one of the more commonly used loops in high level languages. Indeed, the FOR loop (or its counterpart) is the only loop available in many languages include (standard) BASIC and older versions FORTRAN. Since it is so commonly used, it makes sense to spend a little time discussing various implementations of FOR in assembly language.

There are three common uses of the FOR loop which deserve special attention:

- Repeating a sequence of statements some fixed number of times and the body of the loop always executes at least once. The statements in the body do not refer to the FOR loop control variable, the only reason for using FOR is to easily control the number of executions.
- Repeating a loop a fixed number of times (at least once) and the statements in the body of the loop *do* refer to the value of the loop control variable.
- Repeating a loop some arbitrary number of times (including not at all).

In the first case above, the program wants to execute a sequence of statements a fixed number of times (say, 10) and uses the FOR loop to accomplish this. No other features of the FOR loop are necessary for proper execution. The easiest way to accomplish this, if the loop body is short (less than 128 bytes) is to use the 80x86 LOOP instruction. Simply load CX with the number of iterations and let it run:

```
                mov     cx, 10
TenTimes:       print
                byte    "Print this 10 times",cr,lf,0
                loop    TenTimes
```

If the loop body needs to refer to the loop control variable, yet executes a fixed number of times, you can still use the LOOP instruction. For example, the following loop zeros out an array of words using the value in BX as the loop control variable and the value in CX to control the number of iterations:

```
                mov     cx, 10          ;Ten iterations
                mov     bx, 0           ;Loop control var
                mov     ax, 0           ;Value to init array
TenTimes:       mov     Array[bx], ax
                add     bx, 2           ;On to next element
                loop    TenTimes
```

Note in both of the above cases the loop works like a REPEAT..UNTIL loop since the test is at the bottom of the loop. Since the loop executes a fixed number of times and always executes at least once, this is still correct.

The general case does not allow you to use a REPEAT..UNTIL structure. It is quite possible for the FOR loop to skip over the loop body without executing it at all. Therefore, you must use a WHILE loop if you have no idea what the starting and ending values for the loop will be. Consider the following Pascal FOR loop:

```
FOR i := j to k do statement
```

8.12 Under what condition(s) will the above FOR loop *not* execute the associated statement?

_____

_____

This FOR loop is roughly equivalent to the following WHILE statement[3]

```
i := j;
while (i <= k) do begin

        statement
        i := i + 1;

end;
```

Since you've already seen how to convert a WHILE loop into assembly language, you automatically know how to convert the above FOR loop into assembly given this translation to a WHILE loop. The assembly code is

```
; FOR i := j to k do statement

        mov     ax, j
        mov     i, ax
ForLoop:mov     ax, i
        cmp     ax, k
        jnle    EndFor
        <Code for Statement goes here>
        inc     I
        jmp     ForLoop
```

Most programmers will try to use a register for the loop control variable. Assuming you could use DX in the above loop, you'd wind up with a slightly more efficient piece of code:

```
; for dx := j to k do statement

        mov     dx, j
ForLoop:cmp     dx, k
        jnle    EndFor
        <Code for Statement goes here>
        inc     dx
        jmp     ForLoop
```

8.13 Rewrite the code above for the statement "for i := j downto k do *statement*;" (hint: you need only change two lines).

_____ _____

_____ _____

_____ _____

8.6 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:

3. *Roughly* equivalent because many Pascal compilers can more easily optimize the FOR loop and often generate different code for it.

8.11 Place the loop termination test just before the ENDLOOP.

```
          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:

8.14 Supply the code to compute "Aa[dx,cx] := Ba[dx,cx] + Ca[dx, cx];" Assume the arrays are all declared as "Aa, Ba, Ca: array [0..7, 0..7] of integer;"

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

8.7 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
                includelibstdlib.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]>
```

8.12 If J is greater than K upon encountering the FOR loop.

8.13

```
mov   dx, j
ForLoop:
cmp   dx, k
jnge  EndFor
<Code goes here>
dec   dx
jmp   ForLoop
EndFor
```

```
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
```

8.14

```
mov   bx, dx
shl   bx, 3
add   bx, cx
shl   bx, 1
mov   ax, Ba[bx]
add   ax, Ca[bx]
mov   Aa[bx]. ax
```

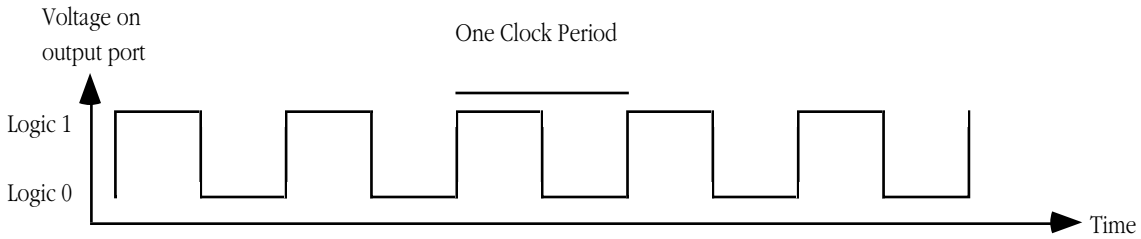
```
cseg            ends

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

8.8 The 8253/8254 Timer Chip

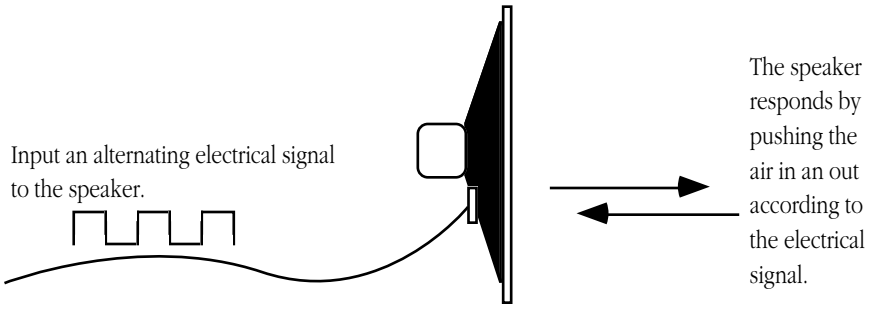
PCs contain a special integrated circuit which produces a period signal. This chip (an Intel compatible 8253 or 8254, depending on your particular computer[4]) contains three different 16-bit counter/timer circuits. The PC uses one of these timers to generate the 1/18.2 second real time clock mentioned in the previous section. It uses the second of these timers to control the DMA refresh on main memory[5]. 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.

8.8.1 The 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* 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 might look like the following:



Note: Frequency is equal to the recipricol of the clock period. Audible sounds are between 20 and 20,000 Hz.



4. 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.

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

8.15 Suppose you supplied a 440 Hz electrical signal to the speaker. What would the frequency of the audible tone be? _____

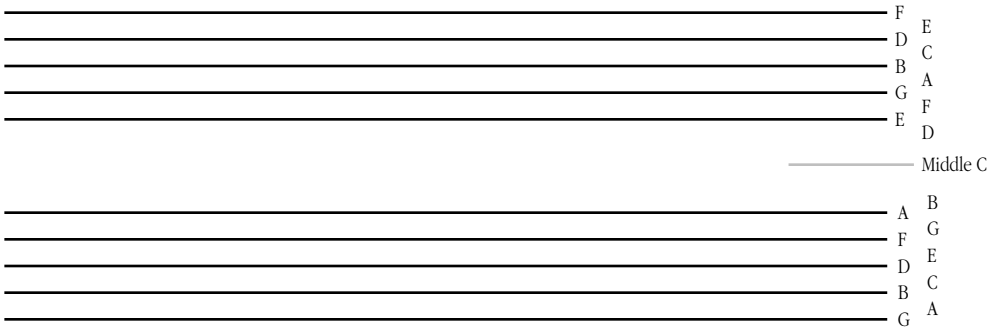
Although many humans are capable of hearing tones in the range 20-20Khz, the PC's speaker is not capable of faithfully reproducing the tones in this range. It works pretty good for sounds in the range 100-10Khz, but the volume drops off dramatically outside this range. Fortunately, this lab only requires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

8.8.2 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#[6]. 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. They generally look like the following:

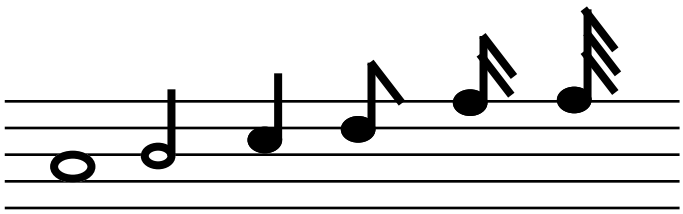


A musical note, as the notation to the side of the staffs above indicates, appears both on the lines of the staffs and the spaces between the staffs. The position of the notes on the staffs determine which note to play, the *shape* of the note determines its duration. There are *whole* notes, *half* notes, *quarter* notes, *eighth* notes, *sixteenth* notes, and *thirty-second* notes[7]. 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 follows:

6. 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.

7. 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.



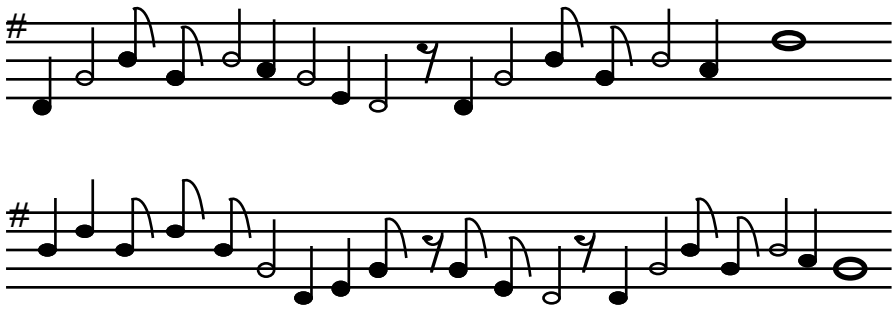
Whole Note | Half Note | Quarter Note | Eighth Note | Sixteenth Note | Thirty-Second Note

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 are the following:

Whole Rest | Half Rest | Quarter Rest | Eighth Rest | Sixteenth Rest | Thirty-Second Rest

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 program. If you are interested in more information on music notation, the library is a good source of information on music theory. The following example is 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[8].

Amazing Grace. John Newton, John Rees, Edwin Excell



8.16 What is the sequence of notes (read from left to right, top to bottom) in the above song?

_____

_____

_____

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

8.17 What are the relative timings (durations) of each of the above notes and rests? _____

_____

_____

_____

8.15 440 hz.

8.8.3 The 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 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.

8.18 What would the frequency be for the E below E0? _____

8.19 What would the frequency be for E4? _____

8.8.4 Programming 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 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.

8.20 What instruction do you use to force bits zero and one to zero? _____

8.21 What is the code which will deactivate the speaker?

_____

_____

_____

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 addresses42h 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.
mov     al, 98h             ;2712 is 0A98h.
out     PIT_Ch2, al         ;Write L.O. byte
mov     al, 0ah
out     PIT_Ch2, al         ;Write H.O. byte
```

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.

8.22 What is the divisor value for Middle C? _____

8.8.5 Putting 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.

8.16 D G B G B A G E D D G B G B A D

B D B D B G D E G G E D D G B G B A G

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.

8.17 Let 8=1/8th note or rest, 4= 1/4 note, 2=1/2 note, and 1=whole note.

4 2 8 8 2 4 2 4 2 8 4 2 8 8 2 4 1

4 4 8 8 8 2 4 4 8 8 8 8 2 8 4 2 8 8 2 4 1

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.

8.18 82.5 hz (83)

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:

8.19 About 2638 Hz.

```
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.

8.23 This array is a 7 x 2 x 4 array of words. What is the formula you would use to access Divisors [I,J,K] assuming row major ordering?

_____

8.24 Assuming AL contains I ('A'..'G'), AH contains J (0..1), and BL contains K (0..3) provide the code which would load the value of DIVISORS [I,J,K] into AX:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

8.9 Before Coming to the Laboratory

Your pre-lab report should contain the following:

- A copy of this lab guide chapter with all the questions answered and corrected.
- A write-up describing the theory of sound and music and how we use the PIT and PPI chips to generate music.
- A table containing divisor values for all the notes in the four octaves presented in this chapter.
- Some sheet music or music in some other form which you can provide as input to your program.
- Source code, debugged and tested, which contains the Delay18 routine presented in this chapter (the Delay18 routine is supplied on the diskette accompanying this lab manual), the PlayNote routine, and the PlayRest routine. The PlayNote routine should be the version which accepts the Note (A..G), Sharp, and Octave parameters.
- A sequence of calls to PlayNote/PlayRest (perhaps via macros) which play your musical selection.

See Chapter Two of this laboratory manual for an example pre-lab report.

Note: your Teaching Assistant or Lab Instructor may elect to give a quiz before the lab begins on the material covered in the laboratory. You will do quite well on that quiz if you've properly prepared for the lab and studied up on the stuff prior to attending the lab. If you simply copy the material from someone else you will do poorly on the quiz and you will probably not finish the lab. Do not take this pre-lab exercise lightly.

8.10 Laboratory Exercises

In this laboratory you will experience the following:

- You will perform experiments with software based delay loops.
- You will use your music program to produce music on the PC's speaker.
- You will adjust the timing of your music by varying a software based time delay

❑ Exercise 1: Software Delay Loops. Run the "Delay" program provided in this chapter. Verify that it takes about five seconds to execute. Modify the program to run for 20 seconds rather than five. Run the program several times, measuring the delay on each run. In your lab report, comment on the accuracy of the delay loop.

❑ Exercise 2: Alignment of the Delay Loops. The alignment of the delay loop can have an impact on the execution time of that loop. At the very least, the loops should begin on a boundary which is a multiple of the machine word size (word on 8086/80286/80386sx, dword on 80386/80486, qword on Pentium). On the 80486 and later processors which provide on-chip caches, it's possible to get slightly better performance by aligning the code on a cache line (16 bytes for 80486, 32 bytes for Pentium). While there is no need to make the code run any faster, there is a need to make it run consistently. The big problem with consistently is that the initialization loop which determines *TimedValue* might be on a different byte boundary than the actual timing loop

in Delay18. As a result, the initialization loop may run at a different speed than the actual delay code. To ensure consistency between the two we have to ensure that both sets of loops begin on the same byte boundary.

8.25 What assembler directive can you use to align a code sequence to a given boundary?

8.26 Assuming you are using an 80486 processor with 32-bit memory, what instruction should you place before the loops to ensure they both lie on reasonable boundaries?

Align the initialization code on a four-byte boundary using the above directive and then follow the directive with three NOPs. This forces the beginning of the loop to (Address MOD 4)+3 which always forces an extra read for the opcode. Align the loops in the Delay18 routine to an even four byte boundary. Adjust the main program to delay for 20 seconds (as in exercise 1). Run the program several times and measure the result. Then align both sets of loops to an even four byte boundary. Retime the program and compare your results.

- Exercise 3: Effect of Wait States. Remove the IN instruction from the two loops in the Delay18. Note the value of the TimedValue variable with and without the IN instruction. Look up the timing for the IN instruction and comment on the speed of the program with and without the IN. How do you explain the discrepancy? Note: on very fast machines you may need to change the value of the innermost loop counter from 10 to a larger value, such as 50.
- Exercise 4: Alignment, again. With the IN instruction removed, repeat exercise two. Comment on the results. As you can see, the large number of wait states introduced by the IN instruction allows the CPU to overlap many operations it could not otherwise do. Note that this effect is exaggerated on the more powerful processors. If you are using a low-end CPU (or a DOS emulator) you may not see significant differences.
- Exercise 5: Playing Music. Using the Note and Rest macros, write a short program to play the "Amazing Grace" tune.
- Exercise 6: Play it again, Sam. Using sheet music you obtain, write a short program to play a tune of your choice.

8.11 Programming Projects

- Program #2: 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;
```

Use the same data values for A and B appearing in program #1 above.

- Program #3: 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)
```

8.20 AND al, 0FCh

8.21

```
in    al, PPI_B
and   al, 0FCh
out   PPI_B, al
```

8.22 4554

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.

8.23 ((I*2+J)*4+K)* 2

8.24

```
shl   al, 1
add   al, ah
shl   al, 2
add   bl, al
mov   bh, 0
shl   bx, 1
mov ax,
      divisors[bx]
```

8.25 .ALIGN *value*

8.26 .ALIGN 4