Arithmetic and Logical Operations   Lab Manual, Chapter Seven

The primary contribution of the first high level languages (HLLs) was the automatic translation of algebraic formulae into assembly language. Indeed, the name of the first popular HLL, FORTRAN, was derived from FORmula TRANslation. Creating assembly language sequences that corresponding to some arithmetic computation is one of the first skills new assembly language programmers must master.

Converting an algebraic expression into assembly language isn't difficult. If you can compute the value of an arithmetic expression manually, you will be able to convert to assembly language. After all, you are probably capable of computing only one operation at a time, the 80x86 has the same limitation. By instructing the 80x86 CPU to perform the same steps as you would, you can create an assembly language sequence that will to the job.

7.1 Arithmetic Operations

Converting a complex arithmetic expression to assembly language is a very straightforward process. First, you break the complex operation into a sequence of simple operations, then you convert those simple operations into assembly language. This is completely analogous to computing the value of an expression by hand. Consider the following arithmetic expression:

```
(6 - 2) * (5 + 3) / 2 + 9
```

You do not solve this entire problem at once in your head. The traditional rules instruct you to perform all operations within the deepest parentheses first, then apply operator precedence, from left to right, when you are faced with a string of operations not separated by parentheses. In the example above, most people would probably solve the problem as follows:

```
(4) * (5+3)/2 + 9
  4 * (8)/2 + 9
    32 / 2 + 9
      16 + 9
        25
```

Had this expression contained variables rather than literal constants, you would have solved the problem in an identical fashion, adding only variable/value replacement.

Converting code to assembly language proceeds in a similar fashion Substituting some arbitrary variables for the above, we obtain:

```
(I-J) * (K+L)/2 + M
```

We can convert this to assembly language the same way we solved the former problem. The only difference is that we write the code to compute the values rather than solving for the answer directly. For the expression above, we would start by computing (I-J)

```
mov       bx, I       ;Compute I-J and leave
sub       bx, J       ; the result in BX.
```

Next, we write the code to compute (K+L):

```
mov       ax, K       ;Compute K+L and leave
sub       ax, L       ; the result in AX.
```

For the next step there are two possible options. We could divide the second result in ax by two then multiply ax by bx, or we could perform the multiplication first and then divide by two. Were we using real arithmetic, it wouldn't matter one bit. However, integer division may produce inexact results and integer multiplication can produce overflow. If using the standard 8086 unsigned mul instruction, it is probably best to perform the multiplication first and the division second.

7.300 Suppose you were to multiply 25,001 by three and divide the result by two. Explain why using the 8086 MUL instruction then a division operation is better than dividing 25,001 by two and multiplying the result by three:

____

____

____

____

____

Since we should perform the multiplication first and the division second, the code to accomplish the next two steps is:

```
mul     bx                  ;DX:AX := AX * BX
shr     dx, 1               ;AX := DX:AX / 2
rcr     ax, 1
```

Note the use of a 32 bit `shr` operation. Since the multiply instruction may produce a non-zero result in the `dx` register, it's a good idea to shift the L.O. bit of `dx` into `ax` when performing the division by two. This, by no means, guarantees a correct result. The product of `ax` and `bx` could have been greater than 131,072. However, if the product happens to fall between 65,536 and 131,072 the code above will produce a correct result where the single precision divide by two would not. If you are absolutely sure that the product of `ax` and `bx` will not exceed 65,535 then you can get by with a single precision `shr` operation.

The next step is to add nine to the result, and this completes the computation. This is easily accomplished with the code:

```
add     ax, 9
```

7.301 Suppose you have an 80386 chip and you've just executed the "MUL BX" instruction. You must still perform the 32 bit SHR operation as an extended precision operation. Why?

____

The 32 bit extended precision SHR operation does not prevent overflow from producing an incorrect answer. The sequence of the multiply and divide operations, combined with the 32 bit SHR operation will help produce a correct answer, that would otherwise be incorrect, in a small number of cases. If there is considerable chance of overflow at any point in the computation, and subsequent computations cannot tolerate that overflow, then you must explicitly check for overflow at several points in the computation. The following code sequence implements overflow checking for the above code (again, assuming unsigned variables):

```
mov     bx, I
sub     bx, J
jc      Overflow
mov     ax, K
add     ax, L
jc      Overflow
mul     bx
shr     dx, 1
rcr     ax, 1
test    dx, dx              ;Overflow if DX <> 0.
jnz     Overflow
add     ax, 9
jc      Overflow
```

7.302 Rewrite the above code to work with *signed* 16 bit integers.

When you compute the result of a complex expression like the above, you will probably need compute *temporary* results needed at some other point in the computation. We shall define a temporary result as a value that must be held in a register or other variable location that is not involved in the current computation. In the example above, there was a single temporary result, `(I-J).` A single register (or register pair in the case of the product `dx:ax`) is all that is necessary for the computation beyond this single temporary result (which the computation above holds in the `bx` register).

Dealing with these temporary results is the key to efficient computations. As you've seen already, the order of your computation can directly affect the correctness of the result. Another important thing to consider is that the order of evaluation affects the efficiency of the computation. For example, by reorganizing your computations using algebraic transformations, you may be able to cut down on the number of temporaries you use in a computation. If this frees up registers for other uses or, better yet, avoids the use of memory variables then you may possibly increase the speed of the computation.

Using fewer registers might *not* improve performance. On the 80486 and later processors that provide pipelining or superscalar operation, a computation that uses a single register will suffer from data hazards and pipeline stalls. By using two or more registers it might be possible to parallelize the computation and avoid the stalls. Nevertheless, the key to optimizing such computations is still to organize the computation and schedule the computations of temporary results to avoid pipeline stalls.

Many of the most powerful compiler optimizations in modern compilers today involve restructuring computations to improve system performance. Therefore, it is important that you are capable of performing efficient transformations of expressions as well. If you are writing a compiler, you will need to know how to perform these transformations yourself. After all, you cannot write a program to perform the transformations (i.e., a compiler) if you cannot perform the transformations manually. If you are writing an assembly language program, you had best be able to perform the transformations manually or a compiler will produce better code than your hand written assembly. Even if you never plan to write a compiler or write another assembly language program, understanding this optimization will help you write better HLL code, since you will be able to design arithmetic expressions that the compiler can more easily optimize.

7.303 Why would you actually want to use more registers than absolutely necessary on a processor such as the 80486?

A typical compiler, one that doesn't employ lots of optimization, generally emits code for an expression evaluated on a strictly left to right basis. Anytime precedence or parentheses would force some other evaluation order, the compiler simply creates a temporary to hold the current results until need. Consider the following expression:

```
X+Y*(A-B*D)
```

A poor to mediocre compiler might compile this as follows:

```
Temp := X
Temp2 := Y
Temp3 := A
Temp4 := B*D
Temp3 := Temp3 - Temp4
Temp2 := Temp2 * Temp3
Temp := Temp + Temp2
```

Because of the two multiply operations that occur, mapping this in a top to bottom order directly to assembly language creates a few minor problems since the 80x86 requires certain computations to take place in certain registers (e.g. mul requires dx:ax). Straight 8086 code could be:

```
mov     bx, X
mov     cx, Y
mov     di, A
mov     ax, B
mul     D
sub     di, ax
mov     ax, cx
mul     di
add     bx, di                  ;Result is in bx.
```

Clearly this code could be improved. The point is, however, that this code evaluates the original expression in a left to right fashion much like some low-end compilers.

If you have an 80386 or later processor, you can improve the above code somewhat since you are allowed to multiply two arbitrary registers. However, there is absolutely no reason why you would want to force yourself to compute this expression in a strictly left to right (top to bottom) fashion. By reorganizing the computation you can avoid the use of gratuitous temporary values. Consider the following reorganization:

```
mov     ax, B                   ;Start by computing the
mul     D                       ; innermost multiplication.
mov     bx, A                   ;Now finish off the inside
sub     bx, ax                  ; computation.
mov     ax, Y                   ;Perform the other multiply.
mul     bx
add     ax, X                   ;Result in AX.
```

Clearly this solution is a little better than the previous. Of course, if you are operating on an 80486 or other pipelined processor, almost every instruction above suffers from a hazard (the "MOV bx, A" instruction in the single exception). If you have a pipelined processor, you could reduce some of the hazards using code like the following:

```
mov     ax, B
mov     bx, A
mul     D
mov     cx, Y
sub     bx, ax
mul     cx, bx                  ;Still have a hazard here.
add     cx, X                   ;Still have a hazard here.
```

Note that the first hazard above, on the multiply instruction, may be somewhat mitigated by the fact that the mul instruction takes far longer than one cycle to execute, even on a pipelined processor.

Reorganizing expressions is one of the key talents you must develop to produce efficient assembly language code. The only way to develop this talent is through lots of experience, by converting lots of expressions to assembly language.

To give you some reasonable amount of guidance while developing this experience, each of the following examples is followed by a question that will give you the opportunity to test your understanding of the current example. All of these examples assume the use of 16 bit unsigned integers.

Example: X + Y + Z

Comment: There are two operations of equal precedence and both operators are commutative. In general, a left to right evaluation is a reasonable way to solve this problem:

```
        mov     ax, X
        add     ax, Y
        add     ax, Z
```

7.304 Write code to compute the value of "X or Y or Z" (bitwise OR):

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

Note that the exact order of the evaluation is of no consequence here. There are six permutations of the above code sequence that will compute the same result. The other five are

```
mov     ax, X                       mov     ax, Y
add     ax, Z                       add     ax, X
add     ax, Y                       add     ax, Z

mov     ax, Y                       mov     ax, Z
add     ax, Z                       add     ax, Y
add     ax, X                       add     ax, X

mov     ax, Z
add     ax, X
add     ax, Y
```

Not all operators are commutative. There are only four simple sequences, rather than six, that perform the legal computation (X + Y - Z):

```
mov     ax, X                       mov     ax, Y
add     ax, Y                       add     ax, X
sub     ax, Z                       sub     ax, Z

mov     ax, X                       mov     ax, X
sub     ax, Z                       sub     ax, Z
add     ax, Y                       add     ax, Y
```

7.305 Write code to compute the value of "X - Y + Z"

____

____

____

Expressions involving multiplication and division present some special problems because the 80x86 does not support fully general forms of these instructions. On pre-286 processors, for example, you must use the ax and dx registers for 16x16 bit multiplications, so you must schedule your computations so these registers are available (that is, do not contain any temporary value) when you need to perform the multiply/divide.

7.300 (25001*3)/2=37501.5
(25001 div 3) * 2 =37500
(25001 * 2) div 3 = 37501
Multiplying by two then dividing by three produces an answer that is closer to the true result.

7.301 The MUL instruction leaves the result if a 16x16 multiplication in DX:AX, not EAX.

7.302

```
mov     bx, I
sub     bx, J
jo      Overflow
mov     ax, K
add     ax, L
jo      Overflow
mul     bx
sar     dx, 1
rcr     ax, 1

; DX=0?

test    dx, dx
jz      NoOvr

; DX = FFFF?

dec     dx
jnz     Overflow
NoOvr:
add     ax, 9
jo      Overflow
```

7.303 To avoid hazards that would occur in many cases when two adjacent instructions use the same register.

Since multiplication is commutative, you would typically code a sequence of multiplications just as you would an addition:

```
;             X * Y * Z
              mov       ax, X        ;Note that this
              mul       Y            ; sequence ignores
              mul       Z            ; overflow.
```

Note that multiplication is associative and commutative, so the order of the multiplications is irrelevant. Any permutation of the above instructions would produce identical results.

7.306 Convert the expression I * J * K to assembly language:

______________________________ ________________________________

______________________________ ________________________________

______________________________ ________________________________

Division creates some additional problems. Remember that the 80x86 DIV/IDIV instructions perform a 64/32, 32/16, or 16/8 division operation. In particular, there is no 32/32, 16/16, or 8/8 division operation. You must remember to initialize the H.O. portion of the dividend prior to the division operation. Otherwise, as long as you observe the non-commutative nature of division, you may perform the operation in several different ways, although there is generally a single “best” way to do it.

```
;             X div Y div Z

              mov       ax, X
              cwd
              div       Y
              cwd                    ;Required, MOD is in DX!
              div       Z
```

7.307 Convert the expression I div J div K to assembly language:

______________________________ ________________________________

______________________________ ________________________________

______________________________ ________________________________

______________________________ ________________________________

7.308 Assume bitwise AND and OR have the same precedence and both are commutative. Provide the most obvious translation of the expression “X and Y or Z” to assembly language:

______________________________ ________________________________

______________________________ ________________________________

______________________________ ________________________________

When you begin mixing operators with different properties, such as associativity, commutativity, or precedence, the situation is a little different. There are some opportunities for optimization that arise. Consider the following example:

```
;             X div Y * Z
;
; Straight-forward left to right conversion to assembly language:

              mov       ax, X
              cwd
              div       Y
              mul       X
```

7.304 mov ax, x
or ax, Y
or ax, Z

Reorganizing this expression is possible if it does not rely on the effects of integer arithmetic. If this were real arithmetic, the computation (Z * X / Y) would produce the same result as (X/Y*Z)[1]. If the fact that these two integer expressions may not produce the same result is okay, then there is an opportunity for optimization. In particular, the multiplication operation sets up the dx register for the multiply, saving us from having to explicitly sign or zero extend the dividend prior to the division:

```
            mov       ax, Z
            mul       X
            div       Y
```

7.309 Convert the expression "A * B div D" to assembly language

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

7.310 Assuming you *do* care about the effects of integer arithmetic, convert the expression "I div J * K" to assembly language:

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

7.305
mov ax, X
sub ax, Y
add ax, Z

7.311 Repeat the above assuming you *don't* care about the effects of integer arithmetic on your computations.

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

Once you begin mixing operators with different precedences, the complexity increases. Of course, the more complex some computation is, the more opportunity for optimization you will have. Consider the hairy arithmetic expression:

```
(I + J * (M + N)) * (X * Y div (K + L) - 5) * Z
```

As usual, you should start with the innermost parentheses. However, there are two subexpressions that could be construed to be "innermost" in the above expression. Which should you do first? It is probably a good idea to compute the (K+L) subexpression and the surrounding operations for two reasons. First, the second big parenthetical expression above contains more multiplication and division operations. Second, if you're on an 80386 or later processor, the div instruction still requires the use of ax and dx but the multiply instruction does not. On an 80386 or later processor, you could use the following code for this expression:

```
; ( X * Y div (K + L) - 5) * Z:

            mov       bx, K
            add       bx, L
            mov       ax, X
            mul       Y           ;Inits DX for div!
```

[1] For integer arithmetic it does not always produce the same result. If X=3, Y = 2, and Z=3, (X div Y *Z) equals 3 while (Z * X div Y) produces 4.

```
              div       bx
              sub       ax, 5
              mul       Z

; (I + J * (M + N)):

              mov       bx, M
              add       bx, N
              mov       cx, J
              mul       bx, cx
              add       bx, I

; Final result:

              mul       ax, bx
```

7.312 Rewrite the above code using 8086 instructions only.

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

7.2 Boolean Operations

Boolean operations are those that produce the result true or false. Logical operations, on the other hand, return a bit string as a result. One could think of a logical as a sequence of many boolean operations taking place in parallel, although you rarely use a logical operation to perform several boolean operations at once. We shall consider these two types of operations together since they tend to use the same machine instructions.

If you have an 80386 or later processor, the presence of the `set`*cc* (conditional set) instructions simplifies the conversion of boolean expressions. Since pre-80386 based systems are quickly becoming, if not already, obsolete we'll begin by discussing the generation of boolean results using these instructions.

The `set`*cc* instructions set an eight bit register to zero or one depending on the condition codes. For example, `sete al` sets `al` to zero if the zero flag is clear, it sets `al` to one if the zero flag is set. So if you have a simple Pascal boolean expression such as

```
B := X = Y;
```

you could easily convert it to the assembly language sequence using `sete` thusly:

```
              mov       ax, X
              cmp       ax, Y
              sete      B           ;B is a byte variable.
```

If you do not have the 80386 instructions available, a simple way to handle this is the following:

```
              mov       bl, 0       ;Assume it's false
              mov       ax, X
              cmp       ax, Y
              jne       NotEql
              mov       bl, 1       ;It's true
NotEql:       mov       B, bl
```

If you use zero and one for false and true, many of the 80x86 logical instructions (AND, OR, and XOR) perform the corresponding boolean operations. NOT is the only logical operation that does not directly correspond to a boolean operation (NOT 1 = 0FEh, which is not zero). You can create a boolean NOT operation using the two instruction sequence:

```
                not       value
                and       value, 1
```

With these four logical operations and the set*cc* instruction, it is very easy to convert a complex boolean expression into assembly language. The following example gives an example of a relatively simple translation:

```
;       B := (I = J) and (K <= L) or (M > N);

                mov       ax, I
                cmp       ax, J
                sete      bl
                mov       ax, K
                cmp       ax, L
                setbe     bh
                and       bl, bh       ;Do AND operation.
                mov       ax, M
                cmp       ax, N
                seta      bh
                or        bl, bh
                mov       B, bl
```

7.313 Convert the following statement to assembly using 80386 instructions: B := (I <> J) or B;

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

If you do not have the set*cc* instruction available, the code is a little more complex. For the previous example, some 8086 code to accomplish this is

```
;       B := (I = J) and (K <= L) or (M > N);

                mov       bx, 0                ;Init BL/BH to zero.
                mov       ax, I
                cmp       ax, J
                jne       False1
                mov       bl, 1
False1:         mov       ax, K
                cmp       ax, L
                jnbe      False2
                mov       bh, 1
False2:         and       bl, bh               ;AND the subexps.
                mov       bh, 0
                mov       ax, M
                cmp       ax, N
                jna       False3
                mov       bh, 1
False3:         or        bl, bh               ;Final OR operation.
                mov       B, bl
```

You can optimize the code above by keeping temporary results in the *program counter*. One way to do this is to quit using bl and bh as boolean variables and merge the computations into the code stream as follows:

7.306
Unsigned, no check for overflow (for signed, change mul->imul).
mov ax, I
mul J
mul K

7.307 Signed:
mov ax, I
cwd
div J
cwd
div K

Unsigned:

mov ax, I
xor dx, dx
idiv J
xor dx, dx
idiv K

7.308
mov ax, x
and ax, y
or ax, z

7.309
(use imul/idiv for signed)
mov ax, A
mul B
div D

7.310 Signed version:
mov ax, I
cwd
idiv J
imul K

7.311 Unsigned version:
mov ax, I
mul K
div J

```
                mov     bl, 0
                mov     ax, I
                cmp     ax, J
                jne     False1
                mov     bl, 1
False1:         mov     ax, K
                cmp     ax, L
                jbe     false2
                mov     bl, 0
False2:         mov     ax, M
                cmp     ax, N
                jna     False3
                or      bl, 1
False3:         mov     B,al
```

The slight improvements present in this code come from the fact that it does not create new temporary variables for the various subexpressions. Instead, it updates a single variable as necessary.

7.314 Convert the statement “B := (I <> J) or (K <= L)” to assembly language using the technique outlined above.

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

We can take this idea of embedding the result of the expression and extend it using *short circuit evaluation.* With short circuit evaluation, once we determine that an expression is true or false and cannot be anything else, the code skips any further computations. The former code can be rewritten to take advantage of this as follows:

```
;       B := ((I = J) and (K <= L)) or (M > N);
;       Extra parentheses added to explicitly show precedence of AND/OR.

                mov     bl, 1       ;Assume result is true.
                mov     ax, M
                cmp     ax, N
                ja      Done        ;If this is true, we’re done.
                mov     ax, I
                cmp     ax, J
                jne     False
                mov     ax, K
                cmp     ax, L
                jbe     Done
False:          mov     bl, 0
Done:           mov     B, bl
```

7.315 Convert the statement in question 7.15 to assembly language using short circuit evaluation.

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

The boolean NOT operation is the only one for which there is not a one to one correspondence between a high level language construct and an 80x86 machine instruction. In general, it is often possible to perform a NOT operation without any machine instructions at all! In other cases, it may take two machine instructions to accomplish the task at hand.

The first example to consider is NOTing a boolean variable. Using the standard values of zero for false and one for true, you cannot compute "B:=NOT B;" by using the single instruction:

```
                NOT       B
```

If B is false, this produces 0FFh, which is not one. Likewise, if B is true, the instruction above produces 0FEh which is not zero. Note that the L.O. bit of bit is correct (0 for false, 1 for true) but the remaining bits are all one. We can easily convert the result of the not instruction to a boolean value by setting bits one through seven to zero after executing the not. This is easily accomplished with an and instruction as follows:

```
                NOT       B
                AND       B, 1
```

Unfortunately, this requires two instructions rather than one. What we would really like is to execute only a single instruction to compute the logical NOT of a boolean variable.

To determine a good solution, note what the boolean NOT operation does – it inverts the L.O. bit of a byte without out affecting the remaining bits. There is an 80x86 instruction that lets you invert specific bits within an operand – xor. To invert the L.O. bit of a byte (that is, to take the boolean NOT of a boolean variable), you could use the following (single) instruction:

```
                xor       B, 1
```

7.316 How would you compute the boolean operation "B1 := NOT B2"?

______________________________ ______________________________

______________________________ ______________________________

______________________________ ______________________________

Many times you can easily eliminate a not instruction from your program by using boolean transformations. The following is an especially trivial example, but provides a good basis for additional examples.

```
;       B := NOT (A <= D)

                mov       ax, A         ;A & D are 16 bit
                cmp       ax, D         ; unsigned integers.
                setnbe    B             ;B is 1 byte boolean.
```

This code implemented the NOT operation by using the *opposite* set*cc* instruction, that is, setnbe rather than setbe. Note that seta is also the opposite of setbe, but setnbe more effectively communicates that we want an opposite operation here. Furthermore, a common mistake among assembly language programmers is to temporarily think that setae is the opposite of setbe which, of course, it is not.

Most of the time you will not be able to eliminate the NOT so easily. Consider the following boolean expression:

```
        B := NOT ((A <= F) or (D > E))
```

You cannot simply swap the state of the set*cc* or j*cc* instructions as with the previous example. The expression ((A > F) or (D <= E)) is not equivalent to the above. However, *Demorgan's Theorems* let us easily eliminate the NOT in this example. The applicable Demorgan's theorems are

```
                NOT (A and B) = (NOT A) or (NOT B)
                NOT (A or B) = (NOT A) and (NOT B)
```

Given these theorems, we can rewrite the previous expression as

```
        B := NOT (A <= F) and NOT (D > E)
```

7.312

```
mov     bx, K
add     bx, L
mov     ax, X
mul     Y
div     bx
sub     ax, 5
mul     Z
mov     bx, ax
mov     ax, M
add     ax, N
mul     J
add     ax, I
mul     bx
```

7.313

```
mov     ax, I
cmp     ax, J
setne   al
or      al, B
mov     B, al
```

We've already explored how to eliminate the NOT operation applied to a comparison, the resulting assembly code for the above is

```
              mov       ax, A       ;BL := NOT(A <= F)
              cmp       ax, F
              setnbe    bl
              mov       ax, D       ;BH := NOT (D > E)
              cmp       ax, E
              setna     bh
              and       bl, bh      ;Logical AND of above
              mov       B, bl       ;Save result away.
```

7.317 Provide the code to compute B := NOT ((A = F) and (D <> E))

In general, unless you are applying the boolean NOT operation to a variable and simply storing the result, you can almost always eliminate instructions from your program to compute the result of the NOT. By rearranging your code and using opposite set*cc* or j*cc* instructions, you can remove instructions that compute the NOT of some subexpression.

Short-circuit evaluation applies equally well to expressions you transform with Demorgan's theorems. Consider the previous example:

```
;       B := NOT ((A <= F) or (D > E))
;
; Transformed to:
;
;       B := NOT (A <= F) and NOT (D > E);

              mov       B, 1                  ;Assume true result
              mov       ax, A
              cmp       ax, F
              jnbe      Done
              mov       ax, D
              cmp       ax, E
              jna       Done
              mov       B, 0                  ;It's a false result.
Done:
```

There are a couple of things to note about this code. First, it runs on any 80x86 CPU, not just the 80386 and later CPUs (the examples using set*cc* require an 80386). Second, this code is quite a bit shorter than the previous code and looks like it should run faster. Keep in mind one major disadvantage of short-circuit evaluation, it often executes some branches that flush the pipeline and prefetch queue. Therefore, a few more instructions without branches might actually run *faster* than a shorter sequence with branches on an 80486 or later processor.

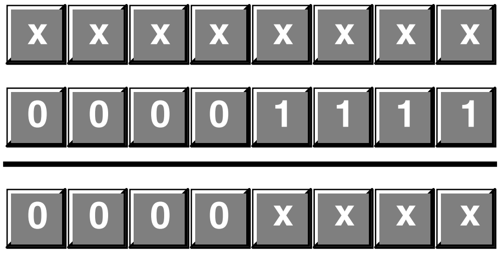
7.318 Repeat Question 7.18 using short-circuit evaluation.

7.3 Logical Operations

The 80x86 logical instructions apply various operations to strings of bits. In addition to the familiar and, or, xor, and not instructions, the logical operations also include several bit shift, rotate, and test instructions. Although the boolean and logical functions share some common operations, the type of results produced by these two sets of functions are quite different. Whereas the boolean operations produce a single true or false result, the logical operations manipulate bits in other values (e.g., integers).

Masking and merging operations are among the more popular logical operations within assembly language programs. *Masking out* is the process of removing a bit field from an object (generally by setting the corresponding bits to zero), *merging in* is the process of inserting some bits into an object, replacing the previous contents.

To mask out a field, that is, set the corresponding bits to zero, you would use the AND operation, placing zeros in the second AND operand where you want zeros in the first operand, placing ones in the second operand where you want to pass the original values:



Note that there is no requirement that you clear a contiguous set of bits. Nor must you use separate instructions to clear several different bit fields. The following example clears bits 0, 1, 3, 4, and 7:

```
and       al, 01100100b
```

You can also mask a field to ones rather than zeros using the or instruction, but this is a rare operation. Most people clear a field to zero prior to merging in some other fields.

7.319 Suppose you want to clear bit positions zero through four, six through nine, and 12 through 15 in the AX register. What instruction would you use to do this?

Merging consists of extracting some bits from one operand and inserting those same bits into and corresponding bit field of a second operand. To merge such values, you must create two masks. The first mask clears all but the desired bits in the first operand and the second mask clears out the bits sitting in the desired location of the second operand. Once you apply these two masks to the corresponding values, you can merge them together with a single or instruction. For example, suppose you want to merge bits four through ten of the ax register into the value in the bx register. You can accomplish this with the following code:

```
and       bx, 1111110000000111b
and       ax, 0000001111111000b
or        ax, bx
```

7.320 Provide the assembly code to merge bits 0…4 and 10…12 in the AX register into the corresponding bits of the BX register.

Merging and masking are two of the three tools you use to *pack* and *unpack* data.. The ability to shift data is the third tool you will need. So the and, or, and shift instructions[2] let you pack and unpack data.

7.314
```
mov     al, 0
mov     bx, I
cmp     bx, J
je      False1
mov     al, 1
False1:
mov     bx, K
cmp     bx, L
jnle    False2
or      al, 1
False2:
mov     B, al
```

7.315
```
mov     al, 1
mov     bx, I
cmp     bx, J
jne     Done
mov     bx, K
cmp     bx, L
jle     Done
mov     al, 0
Done:
mov     B, al
```

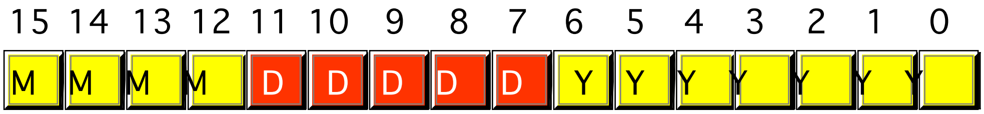
7.316
```
mov     al, B2
xor     al, 1
mov     B1, al
```

Packing data generally consists of taking several different bit zero aligned objects, shifting them to their final position in the resulting object, and merging the values together. For example, suppose you have two nibbles in AL and AH that you wish to convert to a single byte value. Assuming both nibbles are in bits 0…3 and the values of bits 4…7 are unknown, e the following code to pack the two values into AL:

```
shl       ah, 4        ;Also zeros bits 0…3.
and       al, 0Fh      ;Zero bits 4…7.
or        al, ah       ;Merge the values into AL.
```

7.321 Suppose BX contains a five bit value in bits 0…4 and AX contains a ten bit value in bits 1..10. Provide code that merges these two values into AX with the ten bit field winding up in bits 6..15 and the five bit field winding up in bits 1…5.

Unpacking data simply reverses this process. You mask out the bits you don't want and shift the remaining bits down to bit zero (assuming you want a right justified value). For example, consider the DATE data type originally introduced in Chapter Two:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	M	M	M	D	D	D	D	D	Y	Y	Y	Y	Y	Y	Y

To unpack the month field, you could use code like the following:

```
mov       ax, TheDate
and       ah, 11110000b         ;Strip unneeded bits.
shr       ah, 4                 ;Put Day in L.O. bits
mov       Month, ah
```

Unpacking the year value above is very easy. No shifts are necessary. All you must do is zero bit seven of the L.O. byte and the resulting byte is the year value:

```
mov       al, byte ptr TheDate
and       al, 01111111b
mov       Year, al
```

7.322 Write the assembly code to extract the day field from the above code and store this day field in the "Day" variable:

Now consider a somewhat more complex example. Suppose bits zero through two on output port 378h select two of 16 bits from some external device. Also assume that you can read these bits at bit positions three and four on input

[2] The rotate instructions are also useful on occasion when packing and unpacking data.

port 379h. Suppose you want to read all 16 bits from the input port and pack them into the bx register. The following code will do this:

```
            mov     bx, 0       ;Initialize value
            mov     dx, 378h    ;”Address” port adrs.
            mov     ah, 0       ;Bit address
            mov     cx, 8       ;8 pairs of bits.
NextBit:    mov     al, ah      ;Output address zero.
            out     dx, al      ;Output bit adrs.
            inc     dx          ;Point at input port.
            in      al, dx      ;Get input data.
            and     al, 11000b  ;Mask unwanted bits.
            or      bl, al      ;Merge in bits to
            ror     bx, 2       ; the destination.
            inc     ah          ;Try next port.
            dec     dx          ;Point at out port.
            loop    NextBit
```

7.323 The code above leaves input bits 0/1 at bit positions 5 & 6, input bits 2/3 at bit positions 7 & 8, etc. What single ROR instruction do you use to align all the bits at their proper position?

For a second example, suppose that input port 379h contains four input bits at positions three, four, six, and eight. Assume these bits represent the states of some switches connected to the computer and reading a zero means that the switch is in the “on” position, reading a one means that the switch is in the “off” position. Assume that port 378h is read/write. Reading port 378h returns the last value written to that port, writing to port 378h controls eight different LEDs, one on each bit. A one written to a bit on the output port turns on an LED, a zero turns off the corresponding LED. Suppose the four LEDs connected bits 0..4 are supposed to be controlled by the four switches on the input port and the remaining LEDs are not to be changed. The code to accomplish this is

```
mov     dx, 379h    ;Input port adrs.
in      al, dx      ;Read input switches.
not     al          ;Now, 1=on, 0=off.
and     al, 11011000b;Mask unwanted bits.
rcl     al, 1       ;Move bits to pos-
rcl     al, 1       ; itions 0..4
rcl     al, 1
rol     al, 1
rol     al, 1
mov     ah, al      ;Need to save this
and     ah, 0Fh     ; value for merging
dec     dx          ;DX = Output port.
in      al, dx      ;Get LED values.
and     al, 0F0h    ;Merge in new bits.
or      al, ah
out     dx, al
```

Note the sneaky way this code eliminates a single zero in the middle of the four bits by using three RCL and two ROL instructions.

7.317

```
mov     al, 1
mov     bx, a
cmp     bx, f
jne     True1
mov     al, 0
True1:
mov     ah, 1
mov     bx, d
cmp     bx, e
je      Done
mov     ah, 0
Done:
or      al, ah
mov     b, al
```

7.318

```
mov     al, 1
mov     bx, a
cmp     bx, f
jne     Done
mov     al, 0
mov     bx, d
cmp     bx, e
je      Done
mov     al, 0
Done:
mov     b, al
```

7.319 and ax, 0C20h

7.320

```
and     bx, 0E3E0h
and     ax, 1C1Fh
or      bx, ax
```

7.324 Suppose you wanted the switches above to control the LEDs in bits 4...7 rather than 0...3. Write he code to do this (Hint: there is another sneaky sequence similar to the above using two ROR instructions that pack the data properly)

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

____________________ ____________________

Although packing and unpacking data are two very common operations that make use of the 80x86 logical instructions, by no means are they the only ones, nor do they even represent the majority of logical operations performed. You can use these instructions to perform all kinds of arithmetic operations as well.

7.4 Extended Precision Operations

One clear advantage of assembly language over many high level languages is the ability to perform efficient, arbitrary precision arithmetic operations. Knowing how to perform these operations is very important to assembly language programmers.

One could attempt to argue that learning how to perform extended precision arithmetic is a waste of time. After all, integers in existing programming languages are usually sufficient. Indeed, with the trend towards 64 bit processors the need for multiprecision arithmetic seems dubious indeed. After all, who needs to deal with 128 bit integers?

Of course, it wasn't that long ago that people felt 16 bit integers were sufficient. But even if 64 or even 32 bit integers are big enough, there are at least two reasons to learn how to write code to perform extended precision arithmetic. First, the world does not begin and with the 80x86 processor. That is not only to say there are better processors out there (which there are), there are also *worse* processors out there. In pure unit shipments, four bit and eight bit microcontrollers far outnumber all other processors combined. Since it might turn out that *you* will wind up working with one of these processors at one point or another, understanding how to perform arithmetic beyond the bit size of the processor is important.

The second reason for learning multiprecision arithmetic is that you often have to compute some values whose operands not the same size. Often, the same techniques you use to perform multiprecision arithmetic provide the most efficient way to manipulate different sized objects.

The stock logic instructions, `and`, `or`, `xor`, and `not` are very simple to extend to any number of bits. Simply apply the instruction to corresponding groups of bits between the two operands. For example, you could use the following set of instructions to perform a 40 bit AND operation on an 80386 or later CPU:

```
mov     eax, dword ptr Value1
and     eax, dword ptr Value2
mov     dword ptr result, eax
mov     al, byte ptr Value1+4
and     al, byte ptr Value2+4
mov     byte ptr result+4, al
```

7.321

```
shl     ax, 6
shl     bx, 1
and     bx, 3Fh
or      ax, bx
```

7.325 Provide the code to perform a 48 bit logical OR operation on an 80386 or later:

______________________ ______________________

______________________ ______________________

______________________ ______________________

______________________ ______________________

Addition and subtraction are only slightly more complex, using the `adc` and `sbb` instructions to process all but the L.O. byte/word/dword of the operand. The following two examples demonstrate 48 bit addition and subtraction on an 80386 processor:

```
mov    eax, dword ptr I               mov     eax, dword ptr I
add    eax, dword ptr J               sub     eax, dword ptr J
mov    dword ptr K, eax               mov     dword ptr K, eax
mov    ax, word ptr I+4               mov     ax, word ptr I+4
adc    ax, word ptr J+4               sbb     ax, word ptr J+4
mov    word ptr K+4, ax               mov     word ptr K+4, ax
```

7.322

```
mov     ax, TheDate
and     ax, 0F80h
shr     ax, 7
mov     Day, ax
```

7.326 Provide the code to perform a 40 bit subtraction on an 80386 or later:

______________________ ______________________

______________________ ______________________

______________________ ______________________

______________________ ______________________

7.5 Debugging Programs with CodeView

7.323 ror al, 5

In past chapters of this lab manual you've had the opportunity to use CodeView to view the machine state (register and memory values), enter simple assembly language programs, and perform other minor tasks. In this section we will explore one of CodeView's most important capabilities - helping you locate problems within your code. This section discusses three features of CodeView we have ignored up to this point - Breakpoints, Watch operations, and code tracing. These features provide some very important tools for figuring out what is wrong with your assembly language programs.

Code tracing is a feature CodeView provides that lets you execute assembly language statements one at a time and observe the results. Many programmers refer to this operation as *single stepping* because it lets you step through the program one statement per operation. Ultimately, though, the real purpose of single stepping is to let you observe the results of a sequence of instructions, noting all side effects, so you can see why that sequence is not producing desired results.

CodeView provides two easy to use trace/single step commands. Pressing F8 *traces* through one instruction. CodeView will update all affected registers and memory locations and halt on the very next instruction. In the event the current instruction is a `call`, `int`, or other transfer of control instruction, CodeView transfers control to the target location and displays the instruction at that location.

7.327 Suppose the AX register currently contains 1FFh and the current instruction (at location CS:IP) is "MOV AL, 0" What will AX contain after you press the F8 key?

The second CodeView command for single stepping is the *step* command. You can execute the step command by pressing F10. The step command executes the current statement and stops upon executing the statement immediately following it in the program. For most instructions the step and trace commands do the same thing. However, for instructions that transfer control, the trace command follows the flow of control while the step command allows the CPU to run at full speed until returning back to the next instruction. This, for example, lets you quickly execute a subroutine without having to step through all the instructions in that subroutine. You should attempt to using the program trace command (F8) for most debugging purposes and only use the step command (F10) on `call` and int instructions. The step instruction may have some unintended effects on other transfer of control instructions like `loop`, and the conditional branches.

7.328 Which key do you press to *trace* through an instruction? To *step* through an instruction?

The CodeView command window also provides two commands to trace or single step through an instruction. The "T" command traces through an instruction, the "P" command steps over an instruction.

One major problem with tracing through your program is that it is very slow. Even if you hold the F8 key down and let it autorepeat, you'd only be executing 10-20 instructions per second. This is a million (or more) times slower than a typical high-end PC. If the program executes several thousand instructions before even getting to the point where you suspect the bug will be, you would have to execute far too many trace operations to get to that point.

A *breakpoint* is a point in your program where control returns to the debugger. This is the facility that lets you run a program a full speed up to a specific point (the break point) in your program. Breakpoints are, perhaps, the most important tool for locating errors in a machine language program. Since they are so useful, it is not surprising to find that CodeView provides a very rich set of breakpoint manipulation commands.

There are three keystroke commands that let you run your program at full speed and set breakpoints. The F5 command (run) begins full speed execution of your program at CS:IP. If you do not have any breakpoints set, your program will run to completion. If you are interested in stopping your program at some point you should set a breakpoint before executing this command.

Pressing F5 produces the same result as the "Genter" (go) command in the command window. The Go command is a little more powerful, however, because it lets you specify a *non-sticky breakpoint* at the same time. The command window Go commands take the following forms:

```
G enter
G breakpoint_address enter
```

Breakpoints are the subject of the next several paragraphs.

The F7 keystrokes executes at full speed up to the instruction the cursor is on. This sets a *non-sticky* breakpoint. To use this command you must first place the cursor on an instruction in the source window and then press the F7 key. CodeView will set a breakpoint at the specified instruction and start the program running at full speed until it hits a breakpoint.

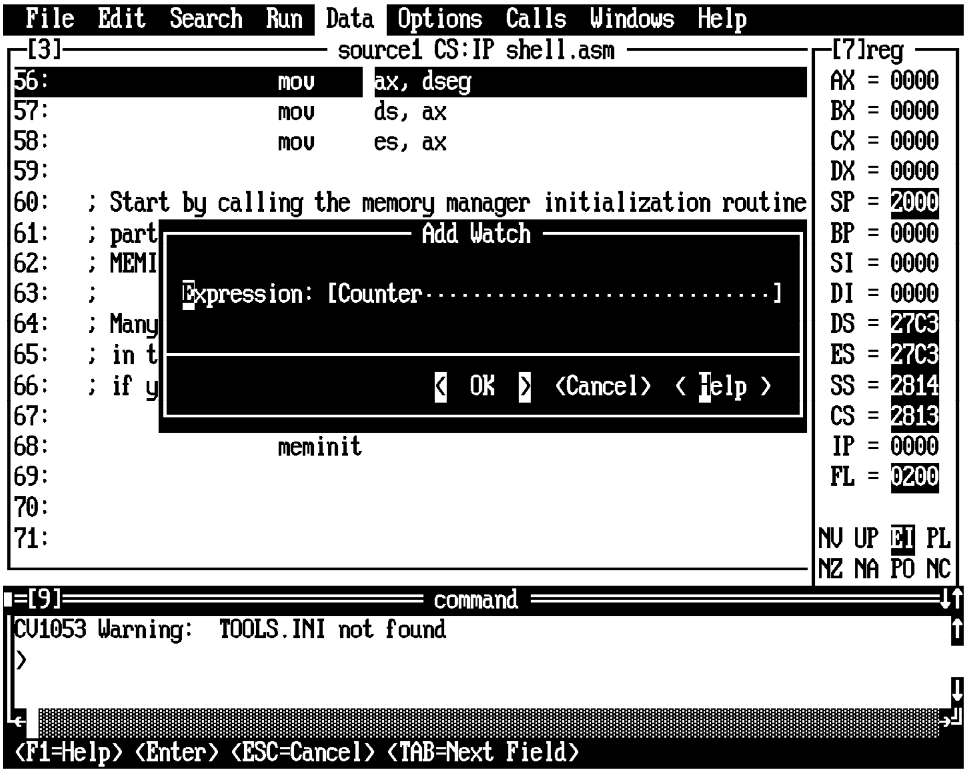
A *non-sticky breakpoint* is one that deactivates whenever control returns back to CodeView. Once CodeView regains control it clears all non-sticky breakpoints. You will have to reset those breakpoints if you still need to stop at that point in your program. Note that CodeView clears the non-sticky breakpoints even if the program stops for some reason other than execution of those non-sticky breakpoints.

One very important thing to keep in mind, especially when using the F7 command to set non-sticky breakpoints, is that you must execute the statement on which the breakpoint was set for the breakpoint to have any effect. If your program skips over the instruction on which you've set the breakpoint, you might not return to CodeView except via program termination. When choosing a point for a breakpoint, you should always pick a *sequence point.* A sequence point is some spot in your program to which all execution paths converge. If you cannot set a breakpoint at a sequence point, you should set several breakpoints in your program if you are not sure the code will execute the statement with the single breakpoint.

The easiest way to set a sticky breakpoint is to move the cursor to the desired statement in the CodeView source window and press F9. This will *brighten* that statement to show that there is a breakpoint set on that instruction. Note that the F9 key only works on 80x86 machine instructions. You cannot use it on blank lines, comments, assembler directives, or pseudo-opcodes.

CodeView's command window also provides several commands to manipulate breakpoints including BC (Breakpoint Clear), BD (Breakpoint Disable), BE (Breakpoint Enable), BL (Breakpoint List), and BP (BreakPoint set). These commands are very powerful and let you set breakpoints on memory modification, expression evaluation, apply counters to breakpoints, and more. See the MASM "Environment and Tools" manual or the CodeView on-line help for more information about these commands.

Another useful debugging tool in CodeView is the *Watch Window.* The watch window displays the values of some specified expressions during program execution. One important use of the watch window is to display the contents of selected variables while your program executes. Upon encountering a breakpoint, CodeView automatically updates all watch expressions. You can add a watch expression to the watch window using the DATA:Add Watch menu item (or control W). This opens up a dialog box that looks like the following:


File Edit Search Run Data Options Calls Windows Help
[3]
source1 CS:IP shell.asm
[7]reg
Add Watch
< OK > <Cancel> < Help >
[9]
command
<F1=Help> <Enter> <ESC=Cancel> <TAB=Next Field>

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

    ; Start by calling the memory manager initialization routine
    ; part
    ; MEMI
    ;
    ; Many
    ; in t
    ; if y

                    meminit
```

Expression: [Counter.............................]

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 2000
BP = 0000
SI = 0000
DI = 0000
DS = 27C3
ES = 27C3
SS = 2814
CS = 2813
IP = 0000
FL = 0200

NV UP EI PL
NZ NA PO NC
```

```
CV1053 Warning:  TOOLS.INI not found
>
```

By typing a variable name (like Counter above) you can add a watch item to the watch window. By opening the watch windows (from the Windows menu item) you can view the values of any watch expressions you've created.

Watch expressions are quite useful because they let you observe how your program affects the values of variables throughout your code. If you place several variable names in the watch list you can execute a section of code up to a break point and observe how that code affected certain variables.

7.324
```
mov  dx, 379h
in   al, dx
not  al
and  al, 11011000b
rcl  al, 1
rcl  al, 1
rcl  al, 1
ror  al, 1
ror  al, 1
and  al, 0F0h
mov  ah, al
dec  dx
in   al, dx
and  al, 0Fh
or   al, ah
out  dx, al
```

Note: you can use
RCL AL, 3
and
ROR AL, 2
for the RCL / ROR instruction sequences above.

7.325 0FFF0h

7.326
```
dp equ <dword ptr>
wp equ <word ptr>
byp equ <byte ptr>

mov  eax, dp Val1
or   eax, dp Val2
mov  dp Result, eax
mov  ax, wp Val1+4
or   ax, wp Val2+4
mov  wp Result+4,ax
```

7.6 Debugging Strategies

Learning how to effectively use a debugger to locate problems in your machine language programs is not something you can learn from a book. Alas, there is a bit of a learning curve to using a debugger like CodeView and learning the necessary techniques to quickly locate the source of an error within a program. For this reason all too many students fall back to debugging techniques they learned in their first or second quarter of programming, namely sticking a bunch of print statements throughout their code. You should not make this mistake. The time you spend learning how to properly use CodeView will pay off very quickly.

7.6.1 Locating Infinite Loops

Infinite loops are a very common problem in many programs. You start a program running and the whole machine locks up on you. How do you deal with this? Well, the first thing to do is to load your program into CodeView. Once you start your program running and it appears to be in an infinite loop, you can manually break the program by pressing the SysReq or Ctrl-Break key. This generally forces control back to CodeView. If you are currently executing in a small loop, you can use the trace command to step through the loop and figure out why it does not terminate.

Another way to catch an infinite loop is to use a *binary search*. To use this technique, place a breakpoint in the middle of your program (or in the middle of the code you wish to test). Start the program running. If it hangs up, the infinite loop is *before* the breakpoint. If you execute the breakpoint, then the infinite loop occurs *after* the breakpoint[3] Once you determine which half of your program contains the infinite loop, the next step is to place another breakpoint half way into that part of the program. If the infinite loop occurred before the breakpoint in the middle of the program, then you should set a new breakpoint one quarter of the way into the program, that is, halfway between the beginning of the program and the original breakpoint. If you got to the original breakpoint without encountering the infinite loop, then set a new breakpoint at the three-quarters point in your program, i.e., halfway between the original breakpoint and the end of your program. Run the program from the beginning again (you can use the CodeView command window command "L" to restart the program from the beginning). If you do not hit any of the three breakpoints you know that the infinite loop is in the first 25% of the program. Otherwise, the current breakpoints at the 25%, 50%, and 75% points in the program will effectively limit the source of the infinite loop to a smaller section of your program. You can repeat this step over and over again until you pinpoint the section of your program containing the infinite loop.

Of course, you should not place a breakpoint within a loop when searching for an infinite loop. Otherwise CodeView will break on each iteration of the loop and it will take you much longer to find the error. Of course, if the infinite loop occurs *inside* some other loop you will eventually need to place breakpoints inside a loop, but hopefully you will find the infinite loop on the first execution of the outside loop. If you do need to place a breakpoint inside a loop that must execute several times before you really want the break to occur, you can attach a *counter* to a breakpoint that counts down from some value before actually breaking. See the MASM Environment and Tools manual, or use CodeView's on-line help facility, to get more details on breakpoint counters.

7.6.2 Incorrect Computations

Another common problem is that you get the wrong result after performing a sequence of arithmetic and logical computations. You can look at a section of code all day long and still not see the problem, but if you trace through the code, the incorrect code because quite obvious.

If you think that a particular computation is not producing a correct result you should set a breakpoint at the first instruction of the computation and run the program at full speed up to that point. *Be sure to check the values of all variables and registers used in the computation.* All too often a bad computation is the result of bad input values, that means the incorrect computation is elsewhere in your program.

[3] Of course, you must make sure that the instruction on which you set the break point is a sequence point. If the code can jump over your breakpoint into the second half of the program, you have proven nothing.

Once you have verified that the input values are correct, you can being tracing the instructions of the computation one at a time. After each instruction executes you should compare the results you actually obtain against those you expected to obtain.

7.327

```
mov  eax, dp Val1
sub  eax, dp Val2
mov  dp Ans, eax
mov  al, byp Val1+4
sbb  al, byp Val2+4
mov  byp Ans, al
```

The main thing to keep in mind when trying to determine why your program is producing incorrect results is that the source of the error could be somewhere else besides the point where you first notice the error. This is why you should always check in input register and variable values before tracing through a section of code. If you find that the input values are *no* correct, then the problem lies elsewhere in your program and you will have to search elsewhere.

7.6.3 Illegal Instructions/Infinite Loops Part II

Sometimes when your program hangs up it is not due to the execution of an infinite loop, but rather you've executed an opcode that is not a valid machine instruction. Other times you will press the SysReq key only to find you are executing code that is nowhere near your program, perhaps out in the middle of RAM and executing some really weird instructions. Most of the time this is due to a stack problem or executing some indirect jump. The best strategy here is to open a memory window and dump some memory around the stack pointer (SS:SP). Try and locate a reasonable return address on the top of stack (or shortly thereafter if there are many values pushed on the stack) and disassemble that code. Somewhere before the return address is probably a call. You should set a breakpoint at that location and begin single stepping into the routine, watching what happens on all indirect jumps and returns. Pay close attention to the stack during all this.

7.328 NOT, AND

7.7 Scanned I/O

Have you ever wondered how a keyboard works? A typical keyboard has in excess of 100 keys. Were the computer to read each key individually it would require one wire for each key on the keyboard. Yet most keyboards today are connected to the computer with fewer than six wires. Of course, most modern keyboards actually have a computer built into the keyboard unit which reads the key switches and then transmits this information serially to the main computer over the connecting cable. Still, there aren't 100 lines inside the keyboard case leading up to the microprocessor which processes keystrokes. That would require too much circuitry and cost far too much.

7.329

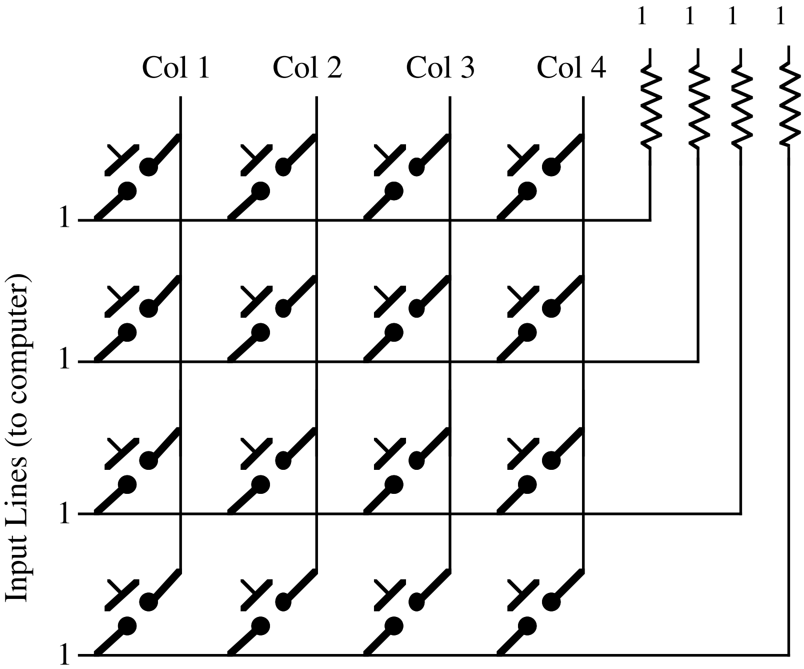
```
AND3macro
and  al, ah
and  al, bl
endm
```

A similar situation exists with output devices. A dot matrix printer, for example, does not provide an impact wire for every dot that appears on the page. Multisegment displays rarely use one output port line for each pixel or segment they display. Once again, the cost of the wiring and the space it would require prohibits this.

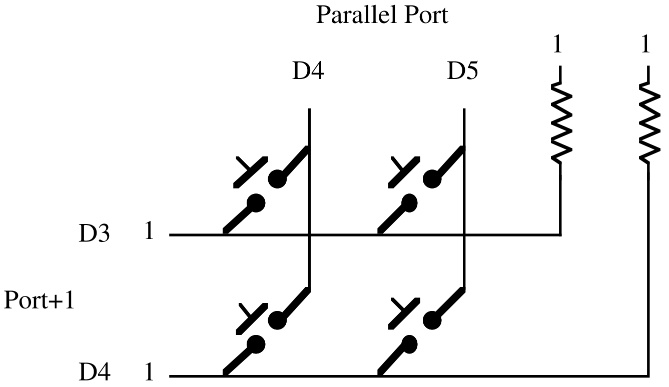
To solve this problem, many I/O devices use *scanning*. Scanning *time multiplexes* the I/O lines between several devices. At one point in time an output line may control one LED or print position on the page, at some other time it controls a different position. In this lab you will experiment with I/O scanning.

7.7.1 Keyboard Scanning

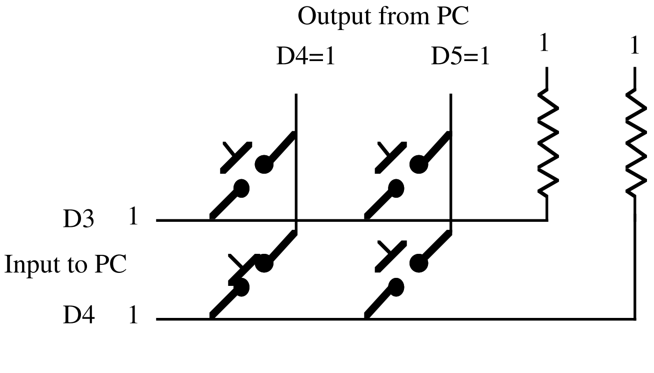
Most modern keyboard units use *keyboard scanning* which allows them to read up to M*N keys using as few as M+N I/O lines. For example, To process 100 keys you could use ten output lines and ten input lines. The trick is to arrange the keys in a *matrix* and use the output lines to control which of the columns of switches in the matrix you wish to read at any one given time. A typical keyboard matrix looks like the following:



This keyboard matrix requires eight I/O lines, four output lines (Col1..Col4 in this schematic) and four input lines. The combination allows for up to 16 keys, the exact number for a given scanned keyboard is NumInputs * NumOutputs. To understand how a scanned keyboard works, consider the schematic for the scanned keyboard you will be building for this chapter's laboratory exercises:

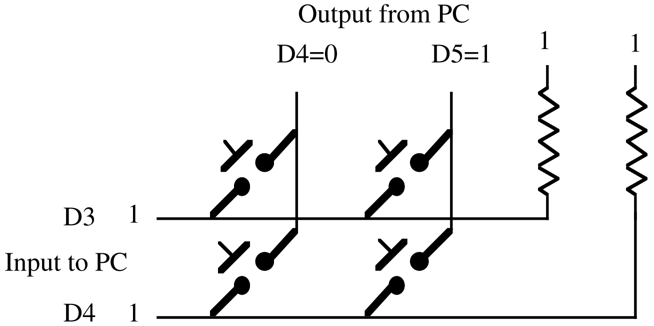


The rows in this matrix connect to bits three and four of the input port on the PC's parallel printer port (at the printer port's base address plus one). Normally (with no switches in the closed position), the PC will read ones on these input bits because there are *pull-up resistors* on these lines that apply a logic one to them. The columns on the matrix connect to bits four and five on the parallel port's output port; the computer can program a zero or one on these lines, although a one is the normal value that appears on the columns.

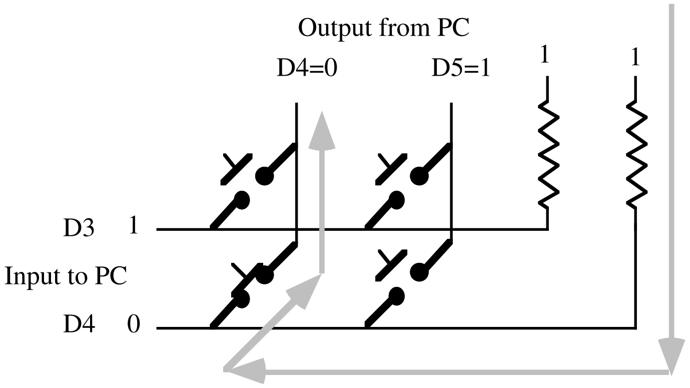


As long as the two columns both contain a one, the inputs on the PC will always be a one even if a switch on the key matrix is in the closed position. This is because the keypress above shorts the D4 column with the D4 row. Since the column and row lines are already one, the value sent to bit four of the input port remains a one.

Now suppose the computer writes a zero to bit four of the output port rather than a one. If all switches are in the open position, the inputs to the PC are still both ones since there is no connection between the zero on the D4 column and the one on the D4 row:

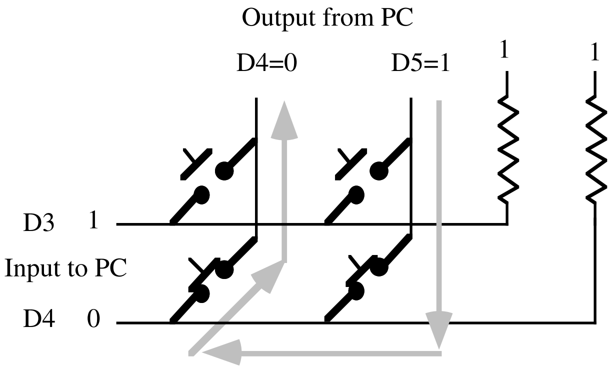


However, suppose bit four on the output port contains a zero and one of the switches on column D4 is in the closed position. This shorts a zero (the column value) with the one placed on the row by the pull up resistor. This creates a signal conflict on the row since the zero and the one are fighting one another. The purpose of the pull up resistor is to weaken the signal so it will "lose" when such a conflict occurs. In essence, pressing the switch shorts the signal from the pull up resistor to ground leaving the input line at a logic zero:

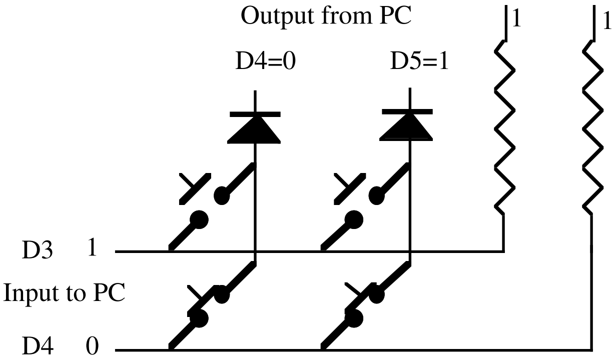


While the D4 column is zero you can read two switches, the two switches connected to the D4 column. Pressing the switches on the other column will not affect the input values. To read the other two switches, you need to program column D4 with a one and column D5 with a zero. Then a zero appearing on the D3 and D4 input lines denote a keypress on the second column of switches.

There is one problem with the circuit above. If two switches on the same row are in the closed position and one column is logic one and the other is logic zero, you will get a dead short across the switches:



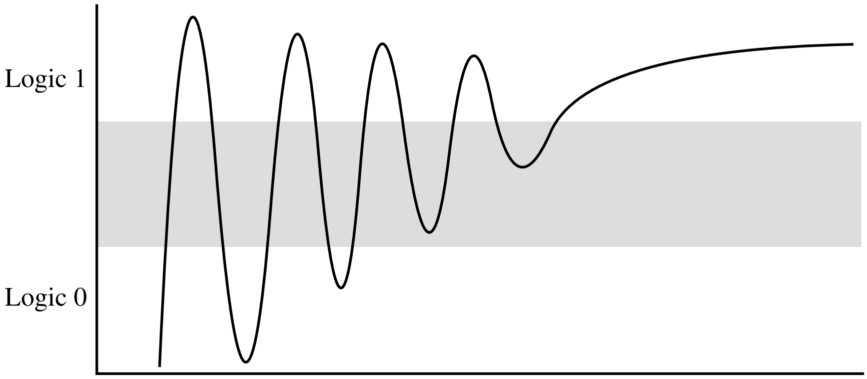
Without a pull up resistor, this dead short could damage the PC's parallel port. By adding two diodes to this circuit, we can protect the PC's parallel port from damage. The revised schematic is



To read these four switches takes two input operations. First, you have to program column D4 with a zero (and D5 with a one) and read the two input bits. This returns the switch settings of the left two switches, above. Then you program column D4 with a one and column D5 with a zero. Reading the input bits in this configuration provides the settings for the right two switches. Regardless of how many columns your keyboard matrix has, only one column at a time should contain a logic zero. Reading the keyboard in this manner is known as *scanning* the keyboard because a zero value scans across the columns in the key matrix.

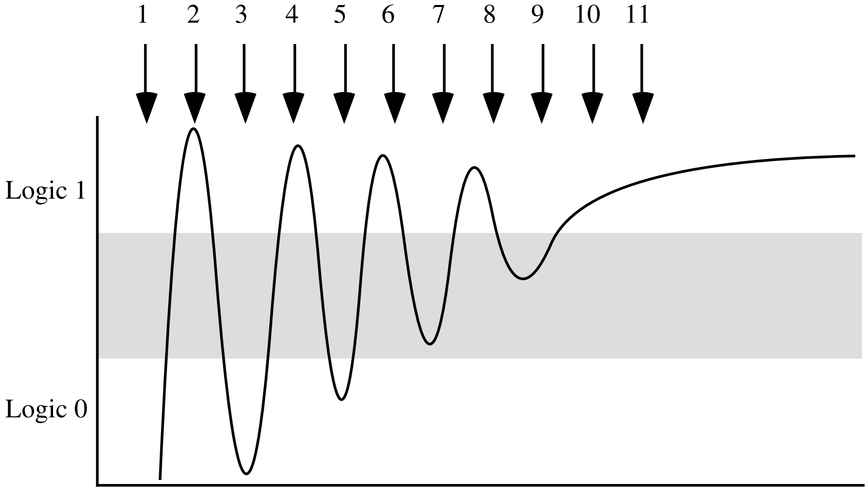
On a typical computer keyboard there are well over 100 keys and a typical key matrix is sixteen output bits and eight input bits (supporting a maximum of 128 keys). The average person can carefully type bursts of keys at about 20-30 characters per second. To avoid missing any keystrokes and avoid transposing the keys (reading one key before another even though the user pressed that other key first), you should *scan* the keyboard (read all the keys) at least once every 10-50 milliseconds.

Scanning the keyboard faster than once every 10 milliseconds or so may introduce the *keybounce* phenomenon. Whenever you press a momentary contact switch, the input signal to the computer does not immediately go from one logic state to another. Instead, there is a brief period during which the contacts on the switch bounce together, making and breaking contact. An analog picture of this looks something like the following:



When you first press the switch there is a big burst of current through it. This raises the voltage to a logic one level. The the switch contacts bounce and break the circuit, this cause the voltage to drop back to a logic zero level. The switch contacts bounce several times, repeating this process.

A problem occurs with keybounce if you scan the key switches too frequently. Suppose you can the keyswitch at the following points during the keybounce:



On the first, third, and fifth scans the input is a logic zero (the seventh and ninth readings might also be a logic zero, though these readings are in an indeterminate area). On the second, fourth, sixth, eighth, ten, and beyond readings, the input is a logic one. Therefore, it will appear to the computer that the user has pressed and released the key three times in quick succession. Had this been a computer keyboard, pressing that key once would have produced three keystrokes.

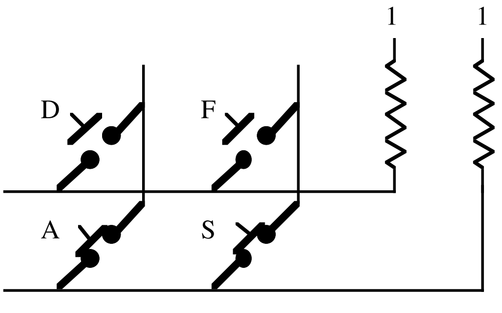
Key contacts on moderate quality keyswitches bounce for between one and five milliseconds. To avoid false readings because of keybounce, you should scan the keyboard less often than once every five milliseconds. Therefore, the 10-50 millisecond figure given earlier will avoid keybounce problems on all but the lowest quality keyboards.

7.7.2 Reading Momentary Key Switches

Reading the dip switches like those appearing in your laboratory components is quite easy, once you've positioned the switches in their ultimate position you can read them at your leisure. Since the switches rarely change during normal operation, you don't need to read the switches very often. Once every second, five seconds, or even once every ten seconds may be sufficient, depending on the application. Consider, however, a momentary key switch like those found on a typical PC keyboard. These switches only register a closure while the operator holds them down. On some keyboards, a typical typist may only hold the key down for a tenth of a second or less. Therefore, the keyboard scanning software needs to scan the keyboard more often that this or it may miss some key presses.

Consider the case where your keyboard software scans a keyboard matrix once every 100 msec (one tenth of a second). If a typist can type slight faster than ten characters per second, it's quite possible for the typist to hit a key just after the software scans the keyboard and release the key just before the software scans the keyboard again. As far as the software is concerned, the key was never down. This, of course, will upset the typist because it will lose certain keystrokes introducing typographical errors into the document.

One solution is to scan the keyboard a little faster than the fastest typist you expect to encounter. This will guarantee that you see every down stroke on the keyboard and the scanning software will not miss any keystrokes. Unfortunately, there is still a problem with this approach. Consider the following section of a keyboard matrix:, note that the user is holding down the "A" and "S" keys at the same time:



On an absolute time scale, one of these keys went down before the other. Therefore, the keyboard should return the code for the first key the user held down. With a slow scanning rate, like one tenth of a second, this may not work properly. For example, suppose the keyboard scans the left column first and the right column second. Also suppose that the user is typing the word "SAND" so the "S" key goes down first and the "A" key goes down second. Finally, assume that the user presses the "S" key immediately after the software scans the keyboard and presses "A" key before the next scan of the keyboard[4]. On the previous scan the software did not see any keys down; on the current scan it sees two keys down. The major question is "which key did the user press first?" The software cannot tell if two (or more) keys go down between two keyboard scan operations. Most likely, it will emit the keycodes in the order it scans the columns. In the example above this will produce the key sequence "AS" rather than "SA" thereby *transposing* the keystrokes.

Since average typists can type two keys in succession *much* faster than they can press and release a single key, it stands to reason that the keyboard scanning routine needs to be faster than they can (reasonably) type two consecutive keys. The only way to prevent possible key transposition is to make sure that you scan the keyboard fast enough to ensure you detect only a single keystroke between keyboard scans (or users, in order to cause transposition, would have to type the keys so fast they couldn't be *sure* they didn't really type them out of order). Generally, scanning the keyboard every 20-40 milliseconds (or faster) should be sufficient. Of course, you don't want to scan the keyboard *too* fast because then your scanning routines will suffer from the effects of keybounce.

There is another major difference between scanning a keyboard matrix and scanning other types of switches. When you non-momentary contact switches, you're generally interested in determining whether the key is in the open or closed position; that is, you only want to know if the switch is on or off. When scanning a keyboard matrix you don't really care if a switch is up or down, what you really want to know is if it just went down or just went up. Most keyboards emit a keycode when you first press a key. Therefore, when scanning a keyboard you need to maintain the status of the keyboard on the *last* scan of the keyboard so you can compare that scan against the current one. If the two readings are different, then the user has just pressed or released a key. On a key down transition the system should emit a keystroke; it should ignore the key up transition. The following code sample demonstrates one way to do this.

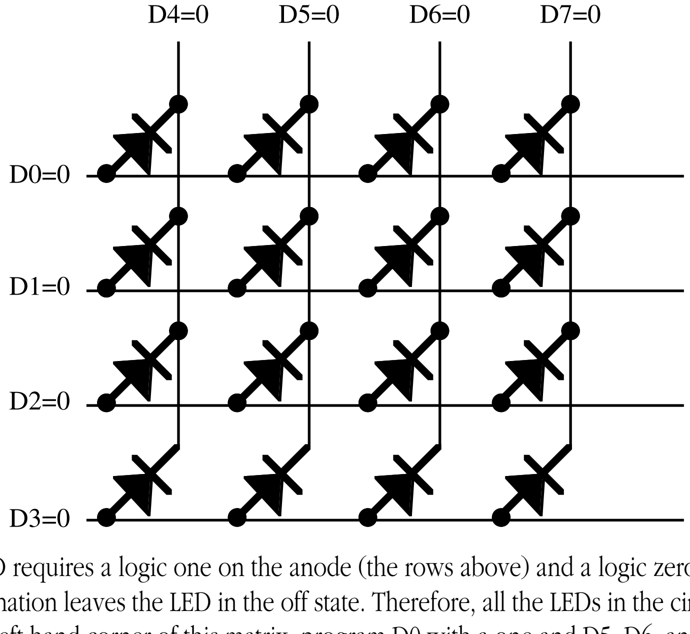
```
; This call to scan reads up to eight keys and returns their current
; settings in the AL register. If a bit is zero, that key is in the
; up position, if a bit is one then the corresponding key is in the
; down position

              mov       al, ThisScan          ;Move previous scan
              mov       LastScan, al          ; to LastScan variable
              call      Scan                  ;Scan the keyboard
              mov       ThisScan, al          ;Save result of scan.

; Okay, let's see if any of the keys have changed. The following
; xor instruction sets AL to zero if the current scan returned the
; same values as the previous scan.

              xor       al, LastScan
              jz        NoDifference

; Okay, AL now contains one bits where the keys have changed, zero
; bits where the keys are the same between the two scans. Lets
; determine which keys just went down. Since "ThisScan" contains
; ones where the keys are currently down, logically ANDing AL with
; "ThisScan" produces a bit map containing ones where the keys
; have just changed state and are in the down posotion. In other words,
; the key just went down.

              and       al, LastScan

; Okay, process each key individually:

              shr       al, 1
              jnc       TryKey1
       < Do whatever you should do for key zero>
```

[4] Most keyboards support *rollover* and do not require that you release a key before pressing the next key.

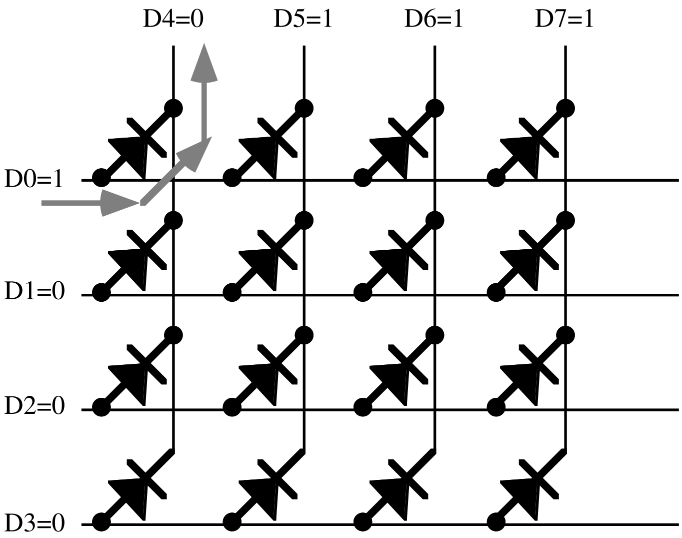
```
TryKey1:      shr       al, 1
              jnc       TryKey2
       < Do whatever you should do for key one>

TryKey2:      shr       al, 1
              jnc       TryKey3
       < Do whatever you should do for key two>

TryKey3:      shr       al, 1
              jnc       TryKey4
       < Do whatever you should do for key three>

TryKey4:      shr       al, 1
              jnc       TryKey5
       < Do whatever you should do for key four>

TryKey5:      shr       al, 1
              jnc       TryKey6
       < Do whatever you should do for key five>

TryKey6:      shr       al, 1
              jnc       TryKey7
       < Do whatever you should do for key six>

TryKey7:      shr       al, 1
              jnc       NoDifference
       < Do whatever you should do for key seven>

NoDifference:
```

7.7.3 Scanning LEDs

You can use this same scanning technique to control output devices as well as read input devices. For example, with eight output lines you can control up to sixteen LEDs rather than the eight LEDs you manipulated in earlier laboratory experiments in this manual. The schematic to this is



To emit light, an LED requires a logic one on the anode (the rows above) and a logic zero on the cathode (the columns above). Any other combination leaves the LED in the off state. Therefore, all the LEDs in the circuit above will be off. To turn on the LED in the upper left hand corner of this matrix, program D0 with a one and D5, D6, and D7 with a logic one:



Programming D5, D6, and D7 with a logic one prevents the other three LEDs on the top row from turning on.

As long as D0 is a logic one, you can turn on or off any LED on the top row by programming bits D4..D7 with a zero (LED on) or a one (LED off). Unfortunately, you cannot control the LEDs on the other rows in an independent fashion at the same time. If D0 and D1 are both a logic one, then setting D4 to zero will turn on both LEDs where D4 intersects with D0 and D1. You cannot program one LED off and the other on while D0 and D1 are both one. The trick is to scan the rows one at a time and turn on the LEDs in each row in succession.

Of course, if you scan through the rows quickly turning specific LEDs on and, conversely, disabling the LEDs in the other rows, the LEDs themselves will only be on for a few brief microseconds, far too short a time to be visible to an observer. Therefore, the program must continually *refresh* the LEDs by constantly scanning them with the appropriate input values to ensure the LEDs are visible. Since the human eye's persistence is about one fifteenth of a second, you should refresh (relight) each LED about once every thirtieth of a second.

7.8 Time Dependent Code

Scanning a keyboard, and LED matrix, and many other operations requires you to introduce a short delay into your software. The most common, although naive, way to create a short loop is to introduce a small delay loop into the program, e.g.,

```
              mov       cx, 10000
DelayLp:      loop      DelayLp
```

The problem with a delay loop like this one is that it will produce inconsistent timings on different CPUs and systems. For example, this loop would take half the time to execute on a 66 MHz 80486 system as it takes on a 33 MHz 80486 system. Any software whose correct operation depends on a software delay loop like the one above will probably fail to run properly on a faster or slower machine than the one you use for software development.

The correct solution is to use a *hardware timer* to measure off a fixed amount of time regardless of the CPU frequency or system speed. Although the PC provides several hardware timer devices, you will not have to learn how to control this hardware in order to do reasonably accurate software delays. This is because the PC's BIOS (Basic Input/Output System) already maintains a real-time clock for you. About every 55 msec one of the system's timer chips generates a *timer interrupt.* The BIOS' timer interrupt handler responds to this interrupt and increments the double word at location 40:6C. The trick to writing a software delay loop that consumes the same amount of time on *any* machine is to count the number of loop iterations you execute between the moment the BIOS changes this memory location to the next time it changes it. After computing the number of iterations necessary for the 55 msec delay, you can delay your software for approximately the same amount of time by executing a similar loop using the computed number of iterations. The following code will compute the number of iterations for a 55 msec delay:

```
; Delay0 provides a short delay so the real delay routine doesn't
; overflow during a 55 msec time period.

Delay0         proc
               push      cx
               mov       cx, 1000
DelayLp0:      loop      DelayLp0
               pop       cx
               ret
Delay0         endp

; InitDelay returns a rough iteration value in the CX register.
; It waits until the BIOS changes the real time clock variable and then
; it counts the number of loop iterations until that value changes
; again.

RTC            textequ   <word ptr es:[6ch]>
DelayConst     word      ?

InitDelay      proc
               push      es            ;Preserve ES, AX, and CX in this procedure
               push      ax
               push      cx

               mov       ax, 40h       ;Point ES at BIOS variables
               mov       es, ax

; Wait until the BIOS real time clock variable changes

               mov       ax, RTC
Wait4RTC:      cmp       ax, RTC       ;The BIOS will change this value
               je        Wait4RTC      ; within 55 msec

; Now that BIOS has just changed the RTC variable, it's time to start counting
; loop iterations until BIOS changes it again, 55 msec later.

               mov       cx, 0         ;Assume up to 65,536 iterations
               mov       ax, RTC       ;Get new RTC value
TimeRTC:       call      Delay0        ;Short delay to eat up some time
               cmp       ax, RTC       ;Was there a change?
               loope     TimeRTC       ; If not, repeat

; The loop above counted CX from 65,536 down to the termination point.
; We need to negate and decrement this value to compute the number of
; loop iterations this system requires for a 55 msec delay. Then save
; this away for use by the real delay routine later.

               neg       cx
               dec       cx
               mov       cs:DelayConst, cx

               pop       cx
               pop       ax
               pop       es
               ret
InitDelay      endp

; After calling the InitDelay routine above to initialize the DelayConst
; variable, call this routine to delay approximately 55 msec (about 1/18th
; of a second).

Delay          proc
               push      es
               push      ax
               push      cx

; The following loop must be nearly identical to the TimeRTC loop in the
; InitDelay procedure. The only difference is that the loop must terminate
; because it counted CX down to zero rather than having the RTC variable
; change. This is easy to arrange by comparing AX against a value that
; never changes.

               mov       ax, cs                 ;Point ES at the segment
               mov       es, ax                 ; containing DelayConst.

               mov       cx, cs:DelayConst      ;DelayConst never changes,
               mov       ax, es:DelayConst      ; so use it as our loop
Delay55ms:    call      Delay0                 ; test variable.
               cmp       ax, es:DelayConst
               loope     Delay55ms              ;Repeats until CX=0.

               pop       cx
               pop       ax
               pop       es
               ret
Delay         endp
```

To use this code, your program needs to call the InitDelay procedure at least once before ever calling the Delay procedure. This will initialize the DelayConst variable with the number of loop iterations Delay must execute for a 55 msec delay. After calling InitDelay, calls to Delay should consume approximately 1/18th seconds.

This technique provides a delay that is approximately 1/18th second on almost any system, assuming the clock speed does not change after you call the InitDelay routine. It is not exactly 1/18th second, but it should be close enough for most purposes. It will certainly be sufficiently close to 55 msec for scanning purposes.

One slight drawback to this scheme is that it only provides a delay of 55 msec. To obtain a longer delay is easy just make successive calls to the Delay routine. For example, to delay for about one second you would use the following procedure:

```
; SecDelay- Delays for about one second. It accomplishes this by calling the
;             55 msec delay routine 18 times (55 msec is approx 1/18 sec).

SecDelay      proc
              push      cx
              mov       cx, 18
SecLp:        call      Delay
              loop      SecLp
              pop       cx
              ret
SecDelay      endp
```

To delay an amount of time less than 55 msec, you will need to divide the DelayConst variable by an appropriate value after calling the InitDelay routine. For example, to delay 10 msec, you need to divide DelayConst by 5.5. You can easily accomplish that by multiplying DelayConst by two and dividing by 11. The following code does this:

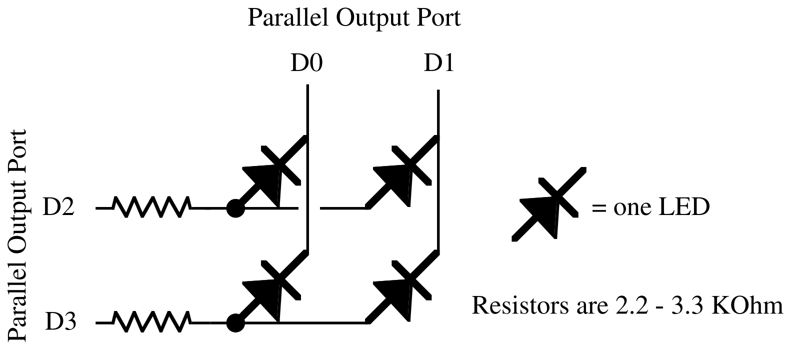
```
              call      InitDelay
              mov       ax, DelayConst
              mov       dx, 0
              shl       ax, 1                  ;32 bit shl operation multiplies
              rcl       dx, 1                  ; dx:ax by two.
              mov       bx, 11
              div       bx                     ;Divide dx:ax by 11.
              mov       DelayConst, ax         ;Save away (DelayConst/5.5)
```

After the above sequence, calls to DelayConst will take approximately 10 msec rather than 55 msec. Please note that the Delay routine will produce increasingly inaccurate delay times as you reduce the value of DelayConst. This is due to the loop inside Delay0. If you have an 80386 or later processor, you can alleviate this problem by eliminating the Delay0 routine and using ecx to control the number of iterations in the InitDelay and Delay subroutines. This, however, makes the InitDelay and Delay routines a bit more complex since the loop instruction does not work properly with the ecx register.

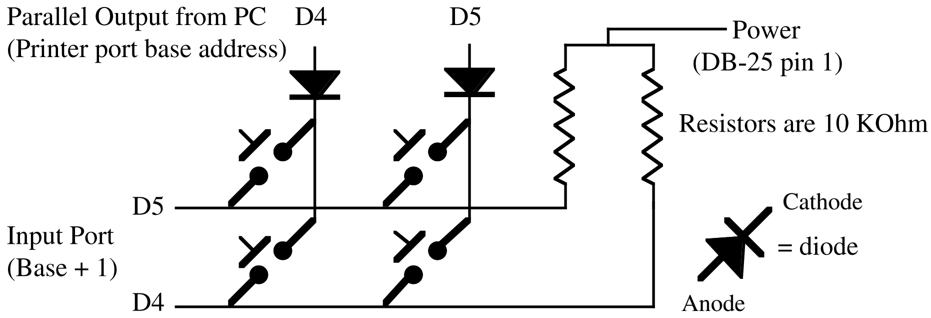
7.9 Constructing the Hardware for the Laboratory

This chapter's laboratory exercises demonstrate keyboard scanning, key bounce, and LED scanning. This will require constructing a new circuit for the exercises. This circuit will allow you to scan four switches and four LEDs using six of your LEDs and the four position DIP switch from your first circuit[5]. You will also use a momentary contact switch to measure keybounce on your system. The schematics for these three experiments are quite simple, they are

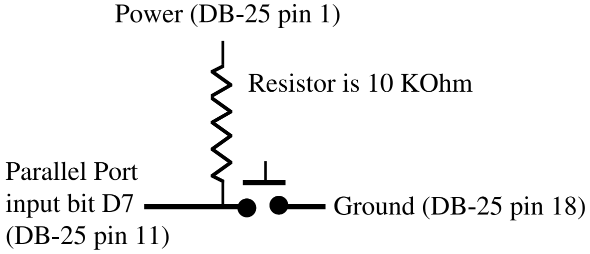
Scanned LED Schematic:



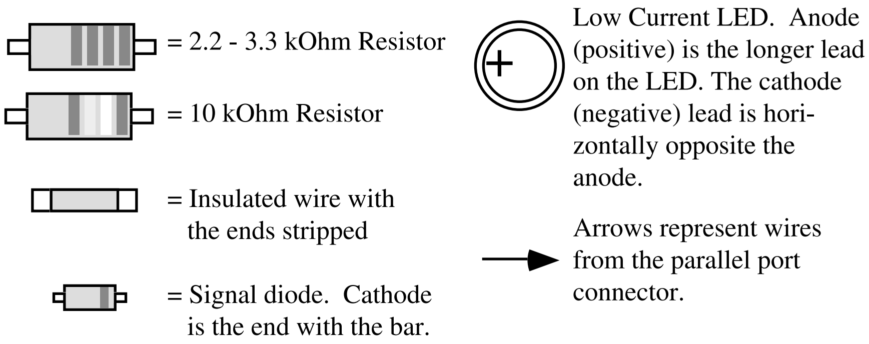
Scanned Keyboard Schematic:



Keybounce Circuit Schematic:



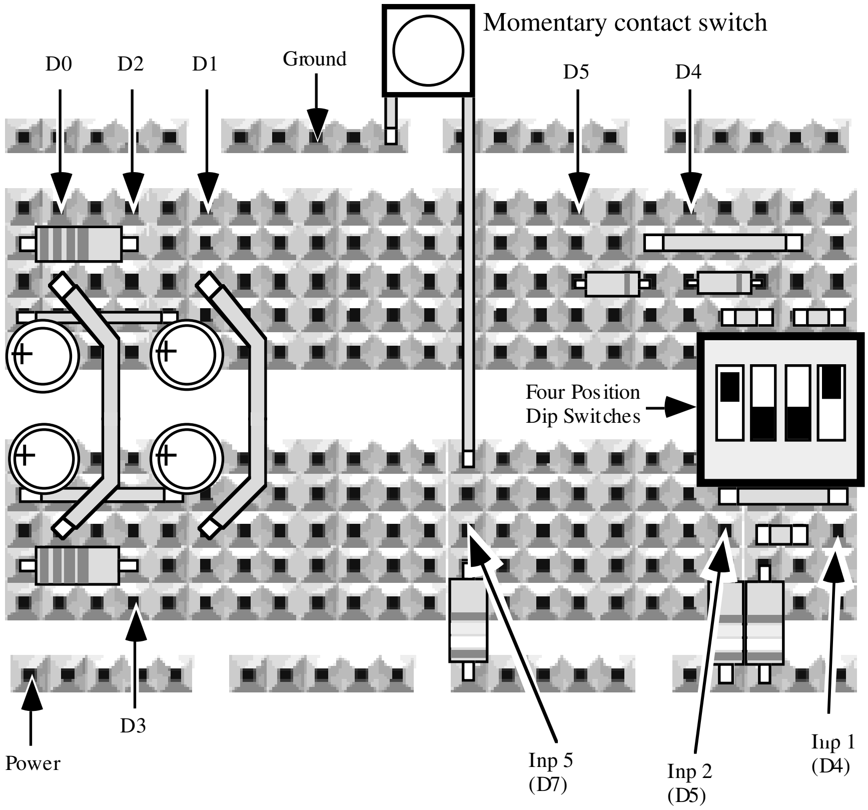
Key to compoents in circuit layout:


= 2.2 - 3.3 kOhm Resistor
= 10 kOhm Resistor
= Insulated wire with the ends stripped
= Signal diode. Cathode is the end with the bar.
+

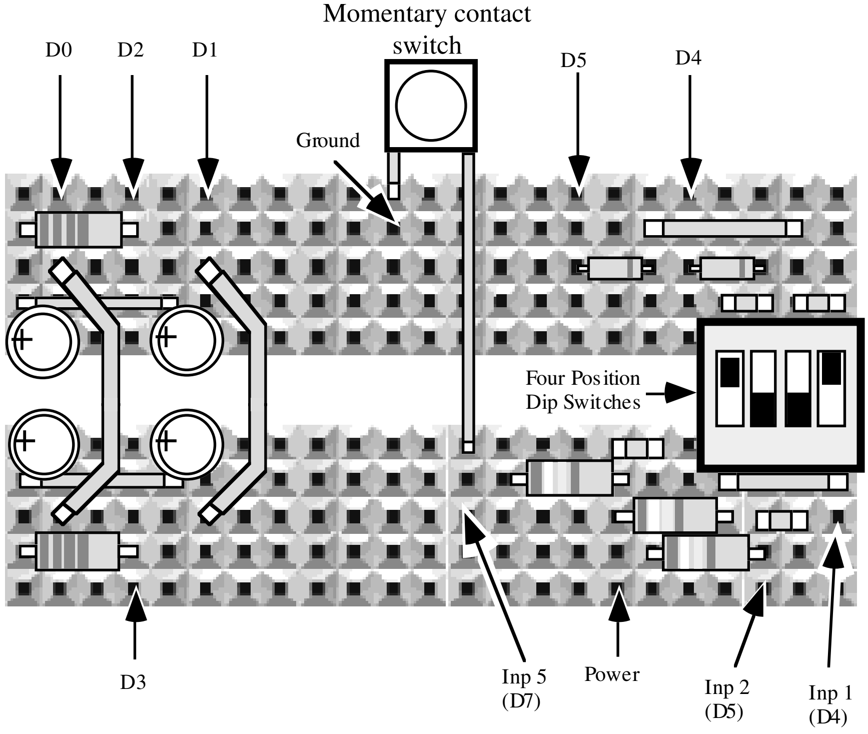
Low Current LED. Anode (positive) is the longer lead on the LED. The cathode (negative) lead is horizontally opposite the anode.

Arrows represent wires from the parallel port connector.

[5] You will use two of the LEDs as diodes to produce the PC hardware from a short circuit when two switches on the same row are in the "on" position.



For those whose prototyping boards do not have the power and ground tie-point busses, use this layout:



7.10 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 on the CodeView debugger explaining in your own words out breakpoints and the watch window work. You should also explain how to run a program from inside CodeView and locate certain types of bugs using CodeView.
- A brief write-up explaining how keyswitch and LED scanning works.
- A brief description of each program you will be using in this laboratory (explain what it does and how to use it).
- You must also assemble the circuitry for this lab.

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.

7.11 Laboratory Exercises

In this laboratory you will perform the following activities:

- Use CodeView to set breakpoints within a program and locate some errors.
- Use CodeView to trace through sections of a program to discover problems with that program.
- Use CodeView to trace through some code you write to verify correctness and observe the calculation one step at a time.
- Write code to control LEDs on an external device and read switches from that device using a *scanning* technique.
- Explore keybounce and visual persistence, two physical real-world constraints on your software.

❏ Exercise 1: Running CodeView. The following program contains several bugs (noted in the comments). Enter this program into the system (note, this code is available as the file "Lab1_7a.asm" on the accompanying diskette):

```
dseg          segment   para public 'data'

I             word      0
J             word      0
K             word      0

dseg          ends

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

; This program is useful for debugging purposes only!
; The intent is to execute this code from inside CodeView.
;
; This program is riddled with bugs. The bugs are very
; obvious in this short code sequence, within a larger
; program these bugs might not be quite so obvious.

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

```
; The following loop increments I until it reaches 10

ForILoop:     inc       I
              cmp       I, 10
              jb        ForILoop

; This loop is supposed to do the same thing as the loop
; above, but we forgot to reinitialize I back to zero.
; What happens?

ForILoop2:    inc       I
              cmp       I, 10
              jb        ForILoop2

; The following loop, once again, attempts to do the same
; thing as the first for loop above. However, this time we
; remembered to reinitialize I. Alas, there is another
; problem with this code, a typo that the assembler cannot
; catch.

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

; The following loop adds I to J until J reaches 100.
; Unfortunately, the author of this code must have been
; confused and thought that AX contained the sum
; accumulating in J. It compares AX against 100 when
; it should really be comparing J against 100.

WhileJLoop:   mov       ax, I
              add       J, ax
              cmp       ax, 100                ;This is a bug!
              jb        WhileJLoop

              mov       ah, 4ch                ;Quit to DOS.
              int       21h
Main          endp
cseg ends

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

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

Assemble this program with the command:

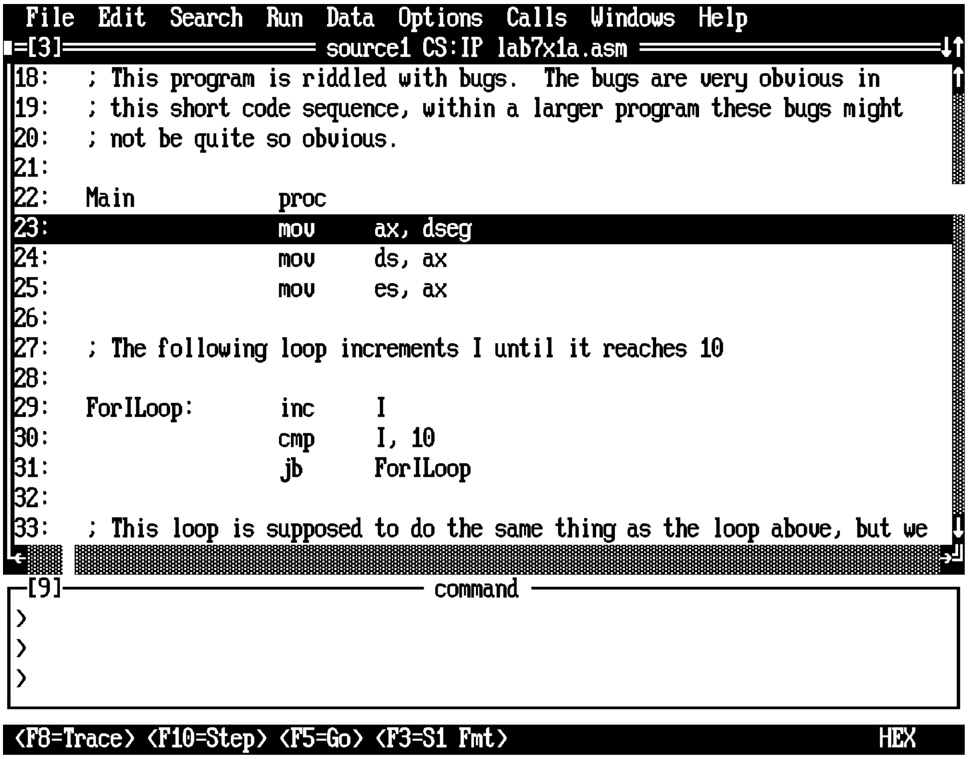
```
ML /Zi lab1_7.asm
```

The “/Zi” option instructions MASM to include debugging information for CodeView in the .EXE file. Note that the “Z” must be uppercase and the “i” must be lower case.

Load this into CodeView using the command:

```
CV lab1_7
```

Your display should now look something like the following:


File Edit Search Run Data Options Calls Windows Help
=[3]= source1 CS:IP lab7x1a.asm
[9] command
>
>
>
<F8=Trace> <F10=Step> <F5=Go> <F3=S1 Fmt>
HEX

```
; This program is riddled with bugs.  The bugs are very obvious in
; this short code sequence, within a larger program these bugs might
; not be quite so obvious.

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

; The following loop increments I until it reaches 10

ForILoop:       inc     I
                cmp     I, 10
                jb      ForILoop

; This loop is supposed to do the same thing as the loop above, but we
```

Note that CodeView highlights the instruction it will execute next (mov ax, dseg in the above code). Try out the trace command by pressing the F10 key three times.This should leave the inc I instruction highlighted. Step through the loop and note all the major changes that take place on each iteration (note: remember jb=jc so be sure to note the value of the carry flag on each iteration as well).

For your lab report: Discuss the results in your lab manual. Also note the final value of I after completing the loop.

❑ Exercise 2: Locating a bug. The second loop in the program contains a major bug. The programmer forgot to reset I back to zero before executing the code starting at label ForILoop2. Trace through this loop until it falls through to the statement at label ForILoop3.

For your lab report: Describe what went wrong and how pressing the F8 key would help you locate this problem.

❑ Exercise 3: Locating another bug. The third loop contains a typo that causes it to restart at label ForILoop. Trace through this code using the F8 key.

For your lab report: Describe the process of tracking this problem down and provide a description of how you could use the trace command to catch this sort of problem.

❑ Exercise 4: Verifying correctness. Program Lab4_7.asm is a corrected version of the above program. Single step through that code and verify that it works correctly.

For your lab report: Describe the differences between the two debugging sessions in your lab manual.

❑ Exercise 5: Using Lab4_7, open a watch window and add the watch expression "I" to that window. Set sticky breakpoints on the three jb instructions in the program. Run the program using the Go command and comment on what happens in the Watch window at each breakpoint.

For your lab report: Describe how you could use the watch window to help you locate a problem in your programs.

❑ Exercise 6: Software Delay Loops. The lab6_7.asm file contains a short software-based delay loop. Run this program and determine the value for the loop control variable that will cause a delay of 11 seconds. Note: the current value was chosen for a 66 MHz 80486 system; if you have a slower system you may want to reduce this value, if you have a faster system, you will want to increase this value. Adjust the value to get the delay as close to 11 seconds as you can on your PC.

For your lab report: Provide the constant for you particular system that produces a delay of 11 seconds. Discuss how to create a delay of 1, 10, 20, 30, or 60 seconds using this code.

For additional credit: After getting the delay loop to run for 11 seconds on your PC, take the executable around to different systems with different CPUs and different clock speeds. Run the program and measure the delay. Describe the differences in your lab report.

❏ Exercise 7: Hardware determined software delay loop. The lab7_7.asm file contains a software delay loop that automatically determines the number of loop iterations by observing the BIOS real time clock variable. Run this software and observe the results.

For your lab report: Determine the loop iteration count and include this value in your lab manual. If your PC has a turbo switch on it, set it to "non-turbo" mode when requested by the program. Measure the actual delay as accurately as you can with the turbo switch in turbo and in non-turbo mode. Include these timings in your lab report.

For additional credit: Take the executable file around to different systems with different CPUs and different clock speeds. Run the program and measure the delays. Describe the differences in your lab report.

❏ Exercise 8: Keyboard and LED scanning. Program lab8_7.asm on the diskette continually scans the switches, stores their values into a byte, then scans the LEDs writing the switch values to the LEDs. Run this program with your circuitry and observe the results. Note that you should never have more than two switches on at the same time. Try turning on three or four switches and observe the results.

For your lab report: When you run the program you will notice that the LEDs are much dimmer than usual. Explain why in your lab report. Include a listing of this program with your lab report. Explain how it scans the keyboard and LEDs. Describe what happens if you turn on three or more switches.

For additional credit: Write two logic functions of two variables. Modify program lab8_7.asm to read the inputs from two of the switches, compute the two results, and write these results back to the LEDs. Demonstrate the program and explain its operation in your lab report.

❏ Exercise 9: Keybounce measurement. Program lab9_7.asm on the diskette is a program that reads the momentary contact switch to see when it changes state. Once it changes state, this program reads the parallel port 50,000 consecutive times and stores away the readings into a memory array. After reading the switch 50,000 times is a row, this program scans through the array to see how many times the values in consecutive array elements are different. This number, divided by two, is the number of times that the key bounced.

This program reads the momentary contact switch on bit seven of the parallel printer input port. The PC's circuitry contains an inverter circuit, so this bit will contain a one when you press the switch, a zero when the switch is in the up position. Keep in mind that this switch reading will be opposite that of the other switches you scan in this laboratory.

There are some very high quality momentary contact switches available. Combined with the fact that fast PCs (e.g., 486s and Pentiums) take a tremendous amount of time to read the parallel port (because of added wait states for I/O devices), it is quite possible that your particular momentary contact switch will not produce any keybounce. If you run the lab9_7 program and it reports zero key bounces, try slamming your finger on the momentary contact switch. Rough treatment generally produces keybounce. If this still doesn't work, use a piece of wire to simulate the momentary contact switch. Connect one end of the wire to ground and poke the other wire in the hole on the other side of the momentary contact switch. This will certainly produce a lot of keybounce, even on fast systems.

For your lab report: Describe how lab9_7.asm works. Explain why the number of value changes in the keyboard values array is twice the number of key bounces. After running lab9_7 several times, compute the minimum, average, and maximum number of key bounces you detect for the series.

For additional credit: Modify the lab9_7.asm program to read one of the dip switches rather than the momentary contact switch. Determine the number of key bounces when you switch the dip switch from one position to another.

- ❏ Exercise 10: Processing keyswitches. Program lab10_7 on the disk reads the momentary contact switch and displays a brief message whenever you press it. Run this program and press the momentary contact switch several times and observe the results.

  For your lab report: Describe what happens when you press the momentary contact switch. Study the source code for this program and explain how it works. Discover how this code debounces the momentary contact switch and describe this process in your lab report.

  For additional credit: Modify this program to scan the dip switches *as well as* the momentary contact switch. Put together a short *scan* routine that returns a bit map containing zeros if a switch is up and ones if a switch is down. Process the bitmap the scan routine returns to determine when a switch transitions from an open position to a closed position. Include the commented printout of this program with your lab report and explain how this program works in your lab report.

7.12 Sample Programs

This chapter's sample programs demonstrate several important concepts including extended precision arithmetic and logical operations, arithmetic expression evaluation, boolean expression evaluation, and packing/unpacking data.

7.12.1 Sample Program #1: 64-Bit Integer I/O

This first sample program reads and writes 64-bit unsigned integers. It demonstrates several extended precision operations. Note that this program requires an 80386 or later CPU.

```
; EX1_7.ASM
;
; This sample program provides two procedures that read and write
; 64-bit unsigned integer values on an 80386 or later processor.

              .xlist
              include   stdlib.a
              includelibstdlib.lib
              .list

              .386
              option    segment:use16

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

dseg          segment   para public 'data'

; Acc64 is a 64 bit value that the ATOU64 routine uses to input
; a 64-bit value.

Acc64         qword     0

; Quotient holds the result of dividing the current PUTU value by
; ten.

Quotient      qword     0

; NumOut holds the string of digits created by the PUTU64 routine.

NumOut        byte      32 dup (0)

; A sample test string for the ATOI64 routine:

LongNumber    byte      "123456789012345678",0
```

```
dseg          ends

cseg          segment   para public ‘code’
              assume    cs:cseg, ds:dseg


; ATOU64-     On entry, ES:DI point at a string containing a
;             sequence of digits.  This routine converts that
;             string to a 64-bit integer and returns that
;             unsigned integer value in EDX:EAX.
;
;             This routine uses the algorithm:
;
;             Acc := 0
;             while digits left
;
;                       Acc := (Acc * 10) + (Current Digit - ‘0’)
;                       Move on to next digit
;
;             endwhile


ATOU64        proc      near
              push      di                     ;Save because we modify it.
              mov       dp Acc64, 0            ;Initialize our accumulator.
              mov       dp Acc64+4, 0

; While we’ve got some decimal digits, process the input string:

              sub       eax, eax               ;Zero out eax’s H.O. 3 bytes.
WhileDigits:  mov       al, es:[di]
              xor       al, ‘0’                ;Translates ‘0’..’9’ -> 0..9
              cmp       al, 10                 ; and everything else is > 9.
              ja        NotADigit

; Multiply Acc64 by ten.  Use shifts and adds to accomplish this:

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

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

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

              pop       edx                    ;Compute Acc64*10 as
              add       dp Acc64, edx          ; Acc64*2 + Acc64*8
              pop       edx
              adc       dp Acc64+4, edx

; Add in the numeric equivalent of the current digit.
; Remember, the H.O. three words of eax contain zero.

              add       dp Acc64, eax          ;Add in this digit

              inc       di                     ;Move on to next char.
              jmp       WhileDigits            ;Repeat for all digits.

; Okay, return the 64-bit integer value in eax.
```

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




; PUTU64-     On entry, EDX:EAX contain a 64-bit unsigned value.
;             Output a string of decimal digits providing the
;             decimal representation of that value.
;
;             This code uses the following algorithm:
;
;                 di := 30;
;                 while edx:eax <> 0 do
;
;                       OutputNumber[di] := digit;
;                       edx:eax := edx:eax div 10
;                       di := di - 1;
;
;                 endwhile
;                 Output digits from OutNumber[di+1]
;                       through OutputNumber[30]

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



              mov       di, dseg               ;This is where the output
              mov       es, di                 ; string will go.
              lea       di, NumOut+30          ;Store characters in string
              std                              ; backwards.
              mov       byp es:[di+1],0        ;Output zero terminating byte.


; Save the value to print so we can divide it by ten using an
; extended precision division operation.

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


; Okay, begin converting the number into a string of digits.

              mov       ecx, 10                ;Value to divide by.
DivideLoop:   mov       eax, dp Quotient+4     ;Do a 64-bit by
              sub       edx, edx               ; 32-bit division
              div       ecx                     ; (see the text
              mov       dp Quotient+4, eax     ;  for details).

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

```
; At this time edx (dl, actually) contains the remainder of the
; above division by ten, so dl is in the range 0..9.  Convert
; this to an ASCII character and save it away.

              mov       al, dl
              or        al, ‘0’
              stosb

; Now check to see if the result is zero.  When it is, we can
; quit.

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

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



; The main program provides a simple test of the two routines
; above.

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

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


Quit:         ExitPgm                        ;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
```

7.12.2 Sample Program #2: Stopwatch Application

This sample program uses the circuity you constructed for this chapter's laboratory exercises. It reads the momentary contact switch to trigger a stopwatch. After you press the momentary contact switch, the computer starts timing until you press the switch a second time. It then displays the number of elapsed seconds.

```
; EX2_7.ASM
;
; This program is a “stopwatch” application.  It uses
; circuitry you build for the lab exercises in lab #7
; to turn the stopwatch on and off.  Whenever you press
; the momentary contact switch, the stopwatch begins
; running.  When you press it a second time, the
; stopwatch terminates execution and display the
; elapsed time.

              .xlist
              include   stdlib.a
              includelibstdlib.lib
              .list

; Location of BIOS variables:

LPT1          textequ   <es:[0ah]>             ;LPT1 port base address.
                                               ; Change to 0Ah for LPT2
                                               ; or 0Ch for LPT3.

RTC           textequ   <es:[6ch]>             ;Real Time Clock variable.

wp            textequ   <word ptr>



dseg          segment   para public ‘data’

; Parallel Port addresses:

InPort        word      ?

; Dummy is a variable the timing loop compares against itself to match
; the timing in the InitDelay routine.

Dummy         word      0

; Timed value is an empirically determined constant which provides a
; suitable delay on whatever machine we are running on.  The program
; computes a reasonable value for this variable.

TimerValue    dword     0

; TimerSave contains a copy of the BIOS timer variable that we read
; when the user first presses the momentary contact switch.
; TimerResult holds the difference in time between the first and
; second press of the momentary contact switch.

TimerSave     dword     0
TimerResult   dword     0

; TimeStr is an array of characters that hold the output string.

TimeStr       byte      16 dup (0)
```

```
dseg            ends



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

; Initialize the TimerValue variable that contains the number of loop
; interations for a 1/18th second delay.

InitDelay       proc
                push      es
                push      ax

; Okay, let's see how long it takes to count down 1/18th of a second.
; RTC is a magic location in the BIOS variables (segment 40h) which
; the Real Time Clock code increments every 55 ms (about 1/18.2 secs).
; This code waits for this location to change, then it counts off how
; long it takes to change again.  By executing that same loop again
; we can get (roughly) equivalent time delays on two separate machines.

                mov       ax, 40h                 ;Segment address of BIOS vars.
                mov       es, ax
                mov       ax, RTC                 ;Wait for timer to change.
RTCMustChange:cmp         ax, RTC
                je        RTCMustChange

; Okay, begin timing the number of iterations it takes for an 18th of a
; second to pass.  The align directive ensures that this loop and Delay's
; corresponding loop both fall on the same cache line boundary.

                mov       wp TimerValue, 0
                mov       wp TimerValue+2, 0
                mov       ax, RTC

                align     16

TimeRTC:        cmp       ax, RTC
                jne       TimerDone

                sub       wp TimerValue, 1
                sbb       wp TimerValue+2, 0
                jne       TimeRTC
                cmp       wp TimerValue, 0
                jne       TimeRTC

; Negate the count down value and decrement it to compute the number
; of times the delay loop must repeat the loop above.

TimerDone:      neg       wp TimerValue+2         ;32-bit negate of
                neg       wp TimerValue           ; TimerValue.
                sbb       wp TimerValue+2, 0

                pop       ax
                pop       es
                ret
InitDelay       endp


; Delay-        This routine delays for roughly a fixed time period on
;               any machine, regardless of CPU or clock rate (May vary by
;               a factor of two or so, but it not as sensitive to CPU
;               speed as a simple LOOP instr).
```

```
Delay         proc
              push      es
              push      ax

              mov       ax, dseg
              mov       es, ax

              push      wp TimerValue+2        ;Save these values
              push      wp TimerValue          ; so we can modify them
              mov       ax, Dummy              ;Compare this with itself.

              align     16

TimeRTC:      cmp       ax, es:Dummy
              jne       DelayDone              ;Never taken.

              sub       wp TimerValue, 1
              sbb       wp TimerValue+2, 0
              jne       TimeRTC
              cmp       wp TimerValue, 0
              jne       TimeRTC

DelayDone:
              pop       wp TimerValue
              pop       wp TimerValue+2
              pop       ax
              pop       es
              ret
Delay         endp



Main          proc
              mov       ax, dseg
              mov       ds, ax

; First, get the base address of the printer port:

              mov       dx, 40h
              mov       es, dx

              mov       dx, LPT1               ;Get printer port base address.
              inc       dx                     ;Add one to get the address
              mov       InPort, dx             ; of the input port.
              inc       dx                     ;Point DX at prtr control port.
              mov       al, 0                  ;Turn on power to circuitry.
              out       dx, al

              print
              byte      cr,lf
              byte      “Stopwatch Application”,cr,lf
              byte      “---------------------”,cr,lf
              byte      cr,lf
              byte      “Press the momentary contact switch to begin “
              byte      “timing an event.”,cr,lf
              byte      “Press it again to stop timing the event.”
              byte      cr,lf
              byte      lf
              byte      0

; Initialize the software delay routine and adjust the TimerDelay
; variable (by dividing it by four) to produce a 14 msec delay rather
; than a 55 ms delay.
```

```
              call      InitDelay

              shr       wp TimerValue+2, 1     ;32-bit shr two bits
              rcr       wp TimerValue, 1       ; operation.
              shr       wp TimerValue+2, 1
              rcr       wp TimerValue, 1




; Okay, continually process the momentary contact switch until the
; user presses it.

StartLoop:
              mov       dx, InPort             ;Read the switch.
              in        al, dx
              test      al, 80h                ;See if it's down.
              jz        StartLoop
              call      Delay                  ;Debounce switch.

; Okay, the user just pressed the momentary contact switch.
; Read the time (in clock ticks) from the DOS BIOS variable
; and save this for later use.  Note: the CLI instruction below
; disables interrupts while reading the timer value from memory.
; This prevents a timer interrupt from changing the value while
; we're in the middle of reading it.  The STI instruction turns
; the interrupt back on, which is necessary if you want BIOS
; to continue updating the timer value.

              mov       dx, 40h                ;Point ES at BIOS variables.
              mov       es, dx
              cli
              mov       ax, RTC                ;Get current timer reading
              mov       wp TimerSave,ax        ; and save away.
              mov       ax, RTC+2
              mov       wp TimerSave+2, ax
              sti

; Wait for the user to release the switch:

Wait4Release: mov       dx, InPort
              in        al, dx
              test      al, 80h
              jnz       Wait4Release
              call      Delay                  ;Debounce switch.

; Okay, wait for the user to press the switch again.

Wait4Press:   mov       dx, InPort
              in        al, dx
              test      al, 80h
              jz        Wait4Press

; The user just pressed the momentary contact switch.  Read the
; BIOS timer variable so we can compute the elapsed time.
; Once again, the interrupts should be off while we read this
; variable so BIOS doesn't update the timer value inbetween our
; reading the two words of this variable.

              cli
              mov       ax, RTC
              sub       ax, wp TimerSave
              mov       wp TimerResult, ax
              mov       ax, RTC+2
              sbb      ax, wp TimerSave+2
              mov      wp TimerResult+2, ax
              sti

; Now it’s time to output the elapsed time (in hh:mm:ss form)
;
; There are approximately 65455 clock ticks in one hour.  The
; following division computes the number of elapsed hours.

              mov      cx, 65455
              mov      dx, wp TimerResult+2
              mov      ax, wp TimerResult
              div      cx
              mov      ch, al                 ;Save # of hours in CH


; There are approximately 1091 clock ticks in a minute, compute
; the minute value here.

              mov      ax, dx                 ;Remainder easily fits ax
              sub      dx, dx
              mov      bx, 1091               ;1091 ticks in a minute
              div      bx
              mov      cl, al                 ;Save # of minutes in CL

; There are approximately 18 clock ticks per second.  Compute
; the seconds value here.

              mov      ax, dx
              sub      dx, dx
              mov      bx, 18
              div      bx
              push     ax                     ;Save seconds.

; Okay, we’re down to fractions of a second.  Multiply the
; remainder by 100 and divide by 18 to get 1/100s of a second.

              mov      ax, dx
              mov      dx, 100
              mul      dx
              mov      bx, 18
              div      bx
              mov      dl, al                 ;Store away 1/100s.
              pop      ax
              mov      dh, al                 ;Store away seconds.


; Okay, all the STDLIB TTOAM routine to convert this time
; to an ASCII string.  Then print the string and free up
; the memory allocated by TTOAM.

              print
              byte     “Time: “,0

              lesi     TimeStr
              ttoa
              puts
              putcr


Quit:         ExitPgm                         ;DOS macro to quit program.
Main          endp
cseg          ends
```

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


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

7.12.3 Sample Program #3: Arithmetic Expressions

This program demonstrates how to convert a few arithmetic expressions into assembly language.

```
; EX3_7.ASM
;
; Several examples demonstrating how to convert various
; arithmetic expressions into assembly language.

              .xlist
              include   stdlib.a
              includelibstdlib.lib
              .list


dseg          segment   para public 'data'

; Arbitrary variables this program uses.

u             word      ?
v             word      ?
w             word      ?
x             word      ?
y             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.

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

              getsm
              atoi
              free

              pop       di
              pop       es
              ret
_geti         endp


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


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

              geti
              mov       u, ax

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

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

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

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


; Okay, compute Z := (X+Y)*(U+V*W)/X and print the result.

              print
              byte      cr,lf
              byte      “(X+Y) * (U+V*W)/X is “,0

              mov       ax, v                  ;Compute V*W
              imul      w                      ; and then add in
              add       ax, u                  ; U.
              mov       bx, ax                 ;Save in a temp location for now.

              mov       ax, x                  ;Compute X+Y, multiply this
              add       ax, y                  ; sum by the result above,
              imul      bx                     ; and then divide the whole
              idiv      x                      ; thing by X.

              puti
              putcr

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

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

              mov       ax, y                  ;Compute y*u first
              imul      u
              mov       dx, X                  ;Now compute X-Y*U
              sub       dx, ax
              mov       cx, dx                 ;Save in temp
              mov       ax, u                   ;Compute U*V
              imul      V
              add       cx, ax                  ;Compute (X-Y*U) + (U*V)

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

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

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

              puti
              putcr


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
```

7.12.4 Sample Program #4:DeMorgan's Theorems

This sample program executes several different boolean expressions. Pairs of these expressions always compute identical results since DeMorgan's theorems claim they are equivalent. Executing this program demonstrates that equivalency.

```
; EX4_7.ASM
;
; This program demonstrates DeMorgan's theorems and
; various other logical computations.


              .xlist
              include   stdlib.a
              includelibstdlib.lib
              .list


dseg          segment   para public 'data'


; Boolean input variables for the various functions
; we are going to test.

a             byte      0
b             byte      0


dseg          ends
```

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


; Get0or1-    Reads a “0” or “1” from the user and returns its
;             its value in the AX register.

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

              getsm
              atoi
              free

              pop       di
              pop       es
              ret
_get0or1      endp




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


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

              get0or1
              mov       a, al

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


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

              mov       ah, 0
              mov       al, a
              and       al, b
              xor       al, 1                 ;Logical NOT operation.

              puti
              putcr
```

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

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

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

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

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

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

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

```
              print
              byte      cr,lf
              byte      “----------------------------------”,cr,lf
              byte      0



Quit:         ExitPgm                        ;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
```

7.12.5 Sample Program #5: Packing and Unpacking Data

This program demonstrates how to work with packed data. It reads a date from the user, checks the consistency of that date, and then packs it into the date format described in Chapter One of the textbook. This program then unpacks that date and uses the STDLIB dtoam routine to convert it to a string for printing.

```
; EX5_7.ASM
;
;       This program demonstrates how to pack and unpack
;       data types.  It reads in a month, day, and year value.
;       It then packs these values into the format the textbook
;       presents in chapter two.  Finally, it unpacks this data
;       and calls the stdlib DTOA routine to print it as text.

              .xlist
              include   stdlib.a
              includelibstdlib.lib
              .list


dseg          segment   para public ‘data’

Month         byte      ?           ;Holds month value (1-12)
Day           byte      ?           ;Holds day value (1-31)
Year          byte      ?           ;Holds year value (80-99)

Date          word      ?           ;Packed data goes in here.

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.
```

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

              getsm
              atoi
              free

              pop       di
              pop       es
              ret
_geti         endp


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


              print
              byte      “Date Conversion Program”,cr,lf
              byte      “-----------------------”,cr,lf
              byte      lf,0


; Get the month value from the user.
; Do a simple check to make sure this value is in the range
; 1-12.  Make the user reenter the month if it is not.

GetMonth:     print
              byte      “Enter the month (1-12): “,0

              geti
              mov       Month, al
              cmp       ax, 0
              je        BadMonth
              cmp       ax, 12
              jbe       GoodMonth
BadMonth:     print
              byte      “Illegal month value, please re-enter”,cr,lf,0
              jmp       GetMonth

GoodMonth:


; Okay, read the day from the user.  Again, do a simple
; check to see if the date is valid.  Note that this code
; only checks to see if the day value is in the range 1-31.
; It does not check those months that have 28, 29, or 30
; day months.

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

GoodDay:


; Okay, get the year from the user.
; This check is slightly more sophisticated.  If the user
; enters a year in the range 1980-1999, it will automatically
; convert it to 80-99.  All other dates outside the range
; 80-99 are illegal.

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

              sub       dx, dx                 ;Zero extend year to 32 bits.
              mov       bx, 100
              div       bx                     ;Compute year mod 100.
              mov       ax, dx
              jmp       GoodYear

TestYear:     cmp       ax, 80
              jb        BadYear
              cmp       ax, 99
              jbe       GoodYear

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

GoodYear:     mov       Year, al


; Okay, take these input values and pack them into the following
; 16-bit format:
;
;      bit 15     8 7      0
;         |      | |      |
;         MMMMDDDD DYYYYYYY


              mov       ah, 0
              mov       bh, ah
              mov       al, Month              ;Put Month into bit positions
              mov       cl, 4                  ; 12..15
              ror       ax, cl

              mov       bl, Day                ;Put Day into bit positions
              mov       cl, 7                  ; 7..11.
              shl       bx, cl

              or        ax, bx                 ;Create MMMMDDDD D0000000
              or        al, Year               ;Create MMMMDDDD DYYYYYYY
              mov       Date, ax               ;Save away packed date.

; Print out the packed date (in hex):

              print
              byte      "Packed date = ",0
              putw
              putcr

; Okay, the following code demonstrates how to unpack this date
; and put it in a form the standard library’s LDTOAM routine can
; use.

              mov       ax, Date              ;First, extract Month
              mov       cl, 4
              shr       ah, cl
              mov       dh, ah                ;LDTOAM needs month in DH.

              mov       ax, Date              ;Next get the day.
              shl       ax, 1
              and       ah, 11111b
              mov       dl, ah                ;Day needs to be in DL.

              mov       cx, Date              ;Now process the year.
              and       cx, 7fh               ;Strip all but year bits.

              print
              byte      “Date: “,0
              ldtoam                          ;Convert to a string
              puts
              free
              putcr

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
```

7.13 Programming Projects

❏ Program #1: Write a program that inputs four integer variables from the user, I, J, K, and L, and prints the results of the following computations. Be sure to check for division by zero and possible arithmetic overflow in each computation.

```
a)     ((I*J)/(I-J) + (I-K) * (L-K)* (K-J)) / 2
b)     (-I)/2 * (J+K-(L*J))/J
c)     (J/(L-2) - 5)*(I-K)
```

❏ Program #2: Write a program that reads two 32 bit hexadecimal values from the user and performs the following extended precision operations these two values:

Addition, Subtraction, AND, OR, XOR

Also perform the following monadic (single operand) operations on both of the operands:

NOT, NEG, SHL, SHR, RCL, RCR, SAR (Shift or rotate values by one bit).

Do *not* use the 80386 32 bit registers. Use only the 16 bit registers available on all processors. Be sure to print the input operands, the operation performed, and the results obtained. Print all values in hex.

❏ Program #3: Write two routines that perform an extended precision (32 bit) ROL and ROR operation on DX:AX. These two routines must leave DX:AX and the carry flag with appropriate values

- Program #4: Write a program that reads a 32 bit hexadecimal value from the user and counts the number of one bits in the input value (e.g., an input of 8888FFFF would produce 20) and prints them to the display. The program should repeat this operation over and over again until the user enters zero.

- Program #5: Modify the stopwatch sample application to print the time in the form HH:MM:SS.DD where DD is time given in hundredths of a second. Of course, the system clock isn't accurate to 1/100 second, but you can still convert the time to this format and display it that way.

- Program #6: Modify sample program #1 in this chapter (the 64-bit integer I/O application) so that it will work on any processor, not just 80386 and later processors.

- Program #7: Modify sample program #5 in this chapter (the date program) so that it :

  a) Properly verifies legal day values (e.g., 28 vs. 29 vs. 30 vs. 31) give the month and year values. Note that a leap year is any year divisible by four unless it is also divisible by 100 and not divisible by 400 (e.g., 1900 is not a leap year but 2000 is a leap year). Hint: You can use the and instruction to easily compute modulo 4

  b) Allows the user to enter any year.

  c) Provides its own conversion from DATE format to a string of the form MMM DD YYYY where MMM is a string of the form JAN, FEB, MAR, etc.

7.14 Answers to Selected Exercises