Arithmetic and Logical Operations Chapter Nine

There is a lot more to assembly language than knowing the operations of a handful of machine instructions. You've got to know how to use them and what they can do. Many instructions are useful for operations that have little to do with their mathematical or obvious functions. This chapter discusses how to convert expressions from a high level language into assembly language. It also discusses advanced arithmetic and logical operations including multiprecision operations and tricks you can play with various instructions.

9.0 Chapter Overview

This chapter discusses six main subjects: converting HLL arithmetic expressions into assembly language, logical expressions, extended precision arithmetic and logical operations, operating on different sized operands, machine and arithmetic idioms, and masking operations. Like the preceding chapters, this chapter contains considerable material that you may need to learn immediately if you're a beginning assembly language programmer. The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- Arithmetic expressions
- Simple assignments
- Simple expressions
- Complex expressions
- Commutative operators
- Logical expressions
- Multiprecision operations
- Multiprecision addition operations
- Multiprecision subtraction operations
- Extended precision comparisons
- ❑ Extended precision multiplication
- ❑ Extended precision division
- ❑ Extended precision negation
- Extended precision AND, OR, XOR, and NOT
- ❑ Extended precision shift and rotate operations
- ❑ Operating on different sized operands
- Multiplying without MUL and IMUL
- ❑ Division without DIV and IDIV
- ❑ Using AND to compute remainders
- ❑ Modulo-n Counters with AND
- ❑ Testing for 0FFFFF...FFFh
- Test operations
- ❑ Testing signs with the XOR instructions
- ❑ Masking operations
- ❑ Masking with the AND instructions
- ❑ Masking with the OR instruction
- ❑ Packing and unpacking data types
- ❑ Table lookups

None of this material is particularly difficult to understand. However, there are a lot of new topics here and taking them a few at a time will certain help you absorb the material better. Those topics with the "•" prefix are ones you will frequently use; hence it is a good idea to study these first.

9.1 Arithmetic Expressions

Probably the biggest shock to beginners facing assembly language for the very first time is the lack of familiar arithmetic expressions. Arithmetic expressions, in most high level languages, look similar to their algebraic equivalents:

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

In assembly language, you'll need several statements to accomplish this same task, e.g.,

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

Obviously the HLL version is much easier to type, read, and understand. This point, more than any other, is responsible for scaring people away from assembly language.

Although there is a lot of typing involved, converting an arithmetic expression into assembly language isn't difficult at all. By attacking the problem in steps, the same way you would solve the problem by hand, you can easily break down any arithmetic expression into an equivalent sequence of assembly language statements. By learning how to convert such expressions to assembly language in three steps, you'll discover there is little difficulty to this task.

9.1.1 Simple Assignments

The easiest expressions to convert to assembly language are the simple assignments. Simple assignments copy a single value into a variable and take one of two forms:

```
variable := constant
```

or

```
variable := variable
```

If variable appears in the current data segment (e.g., DSEG), converting the first form to assembly language is easy, just use the assembly language statement:

```
mov     variable, constant
```

This move immediate instruction copies the constant into the variable.

The second assignment above is somewhat complicated since the 80x86 doesn't provide a memory–to-memory mov instruction. Therefore, to copy one memory variable into another, you must move the data through a register. If you'll look at the encoding for the mov instruction in the appendix, you'll notice that the mov ax, memory and mov memory, ax instructions are shorter than moves involving other registers. Therefore, if the ax register is available, you should use it for this operation. For example,

```
var1 := var2;
```

becomes

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

Of course, if you're using the ax register for something else, one of the other registers will suffice. Regardless, you must use a register to transfer one memory location to another.

This discussion, of course, assumes that both variables are in memory. If possible, you should try to use a register to hold the value of a variable.

9.1.2 Simple Expressions

The next level of complexity up from a simple assignment is a simple expression. A simple expression takes the form:

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

Var is a variable, $term_1$ and $term_2$ are variables or constants, and op is some arithmetic operator (addition, subtraction, multiplication, etc.).

As simple as this expression appears, most expressions take this form. It should come as no surprise then, that the 80x86 architecture was optimized for just this type of expression.

A typical conversion for this type of expression takes the following form:

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

Op is the mnemonic that corresponds to the specified operation (e.g., "+" = add, "-" = sub, etc.).

There are a few inconsistencies you need to be aware of. First, the 80x86's {i}mul instructions do not allow immediate operands on processors earlier than the 80286. Further, none of the processors allow immediate operands with {i}div. Therefore, if the operation is multiplication or division and one of the terms is a constant value, you may need to load this constant into a register or memory location and then multiply or divide ax by that value. Of course, when dealing with the multiply and divide instructions on the 8086/8088, you must use the ax and dx registers. You cannot use arbitrary registers as you can with other operations. Also, don't forget the sign extension instructions if you're performing a division operation and you're dividing one 16/32 bit number by another. Finally, don't forget that some instructions may cause overflow. You may want to check for an overflow (or underflow) condition after an arithmetic operation.

Examples of common simple expressions:

```
X := Y + Z;

                mov     ax, y
                add     ax, z
                mov     x, ax

X := Y - Z;

                mov     ax, y
                sub     ax, z
                mov     x, ax

X := Y * Z; {unsigned}

                mov     ax, y
                mul     z           ;Use IMUL for signed arithmetic.
                mov     x, ax       ;Don't forget this wipes out DX.

X := Y div Z; {unsigned div}

                mov     ax, y
                mov     dx, 0       ;Zero extend AX into DX
                div     z
                mov     x, ax

X := Y div Z; {signed div}

                mov     ax, y
                cwd                 ;Sign extend AX into DX
                idiv    z
                mov     x, ax

X := Y mod Z; {unsigned remainder}

                mov     ax, y
                mov     dx, 0       ;Zero extend AX into DX
                div     z
                mov     x, dx       ;Remainder is in DX
```

```
X := Y mod Z; {signed remainder}

                mov     ax, y
                cwd                     ;Sign extend AX into DX
                idiv    z
                mov     x, dx           ;Remainder is in DX
```

Since it is possible for an arithmetic error to occur, you should generally test the result of each expression for an error before or after completing the operation. For example, unsigned addition, subtraction, and multiplication set the carry flag if an overflow occurs. You can use the jc or jnc instructions immediately after the corresponding instruction sequence to test for overflow. Likewise, you can use the jo or jno instructions after these sequences to test for signed arithmetic overflow. The next two examples demonstrate how to do this for the add instruction:

```
X := Y + Z; {unsigned}

                mov     ax, y
                add     ax, z
                mov     x, ax
                jc      uOverflow

X := Y + Z; {signed}

                mov     ax, y
                add     ax, z
                mov     x, ax
                jo      sOverflow
```

Certain unary operations also qualify as simple expressions. A good example of a unary operation is negation. In a high level language negation takes one of two possible forms:

```
var := -var             or          var1 := -var2
```

Note that var := -constant is really a simple assignment, not a simple expression. You can specify a negative constant as an operand to the mov instruction:

```
                mov     var, -14
```

To handle the first form of the negation operation above use the single assembly language statement:

```
                neg     var
```

If two different variables are involved, then use the following:

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

Overflow only occurs if you attempt to negate the most negative value (-128 for eight bit values, -32768 for sixteen bit values, etc.). In this instance the 80x86 sets the overflow flag, so you can test for arithmetic overflow using the jo or jno instructions. In all other cases the80x86 clears the overflow flag. The carry flag has no meaning after executing the neg instruction since neg (obviously) does not apply to unsigned operands.

9.1.3 Complex Expressions

A complex expression is any arithmetic expression involving more than two terms and one operator. Such expressions are commonly found in programs written in a high level language. Complex expressions may include parentheses to override operator precedence, function calls, array accesses, etc. While the conversion of some complex expressions to assembly language is fairly straight-forward, others require some effort. This section outlines the rules you use to convert such expressions.

A complex function that is easy to convert to assembly language is one that involves three terms and two operators, for example:

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

Clearly the straight-forward assembly language conversion of this statement will require two sub instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

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

The second conversion, since it is shorter, looks better. However, it produces an incorrect result (assuming Pascal-like semantics for the original statement). Associativity is the problem. The second sequence above computes W := W - (Y - Z) which is not the same as W := (W - Y) - Z. How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

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

This computes W:=W-(Y+Z). This is equivalent to W := (W - Y) - Z.

Precedence is another issue. Consider the Pascal expression:

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

Once again there are two ways we can evaluate this expression:

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

By now, you're probably thinking that this text is crazy. Everyone knows the correct way to evaluate these expressions is the second form provided in these two examples. However, you're wrong to think that way. The APL programming language, for example, evaluates expressions solely from right to left and does not give one operator precedence over another.

Most high level languages use a fixed set of precedence rules to describe the order of evaluation in an expression involving two or more different operators. Most programming languages, for example, compute multiplication and division before addition and subtraction. Those that support exponentiation (e.g., FORTRAN and BASIC) usually compute that before multiplication and division. These rules are intuitive since almost everyone learns them before high school. Consider the expression:

```
                X op1 Y op2 Z
```

If op_1 takes precedence over op_2 then this evaluates to (X op_1 Y) op_2 Z otherwise if op_2 takes precedence over op_1 then this evaluates to X op_1 (Y op_2 Z). Depending upon the operators and operands involved, these two computations could produce different results.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

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

                mov     bx, x
                mov     ax, y           ;Must compute Y * Z first since
                mul     z               ; "*" has the highest precedence.
                add     bx, ax          ;Now add product with X's value.
                mov     w, bx           ;Save away result.
```

Since addition is a *commutative* operation, we could optimize the above code to produce:

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

                mov     ax, y       ;Must compute Y * Z first since
                mul     z           ; "*" has the highest precedence.
                add     ax, x       ;Now add product with X's value.
                mov     w, ax       ;Save away result.
```

If two operators appearing within an expression have the same precedence, then you determine the order of evaluation using *associativity* rules. Most operators are *left associative* meaning that they evaluate from left to right. Addition, subtraction, multiplication, and division are all left associative. A *right associative* operator evaluates from right to left. The exponentiation operator in FORTRAN and BASIC is a good example of a right associative operator:

```
2^2^3 is equal to 2^(2^3) not (2^2)^3
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

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

                mov     ax, x       ;All the same operator, so we need
                sub     ax, y       ; to evaluate from left to right
                sub     ax, z       ; because they all have the same
                mov     w, ax       ; precedence.

; W := X + Y * Z

                mov     ax, y       ;Must compute Y * Z first since
                imul    z           ; multiplication has a higher
                add     ax, x       ; precedence than addition.
                mov     w, ax

; W := X / Y - Z

                mov     ax, x       ;Here we need to compute division
                cwd                 ; first since it has the highest
                idiv    y           ; precedence.
                sub     ax, z
                mov     w, ax

; W := X * Y * Z

                mov     ax, y       ;Addition and multiplication are
                imul    z           ; commutative, therefore the order
                imul    x           ; of evaluation does not matter
                mov     w, ax
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is always better to perform the multiplication first. For example, given an expression of the form:

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

It is better to compute X*Z and then divide the result by Y rather than divide X by Y and multiply the quotient by Z. There are two reasons this approach is better. First, remember that the imul instruction always produces a 32 bit result (assuming 16 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the dx register so you do not have to sign extend ax prior to the division. This saves the execution of the cwd instruction. A second reason for doing the multiplication first is to increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5/2 you will get the value two, not 2.5. Computing (5/2)*3 produces six. However, if you compute (5*3)/2 you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

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

You can usually convert this to assembly code:

```
mov     ax, x
imul    z
idiv    z
mov     w, ax
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following Pascal statement:

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

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute Y * Z and then subtract X from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load X into a register, multiply Y and Z leaving their product in a different register, and then subtract this product from X, e.g.,

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

This is a trivial example that demonstrates the need for *temporary variables* in an expression. The code uses the `bx` register to temporarily hold a copy of `X` until it computes the product of `Y` and `Z`. As your expression increase in complexity, the need for temporaries grows. Consider the following Pascal statement:

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

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

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

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov     ax, a
add     ax, b
mov     Temp1, ax
mov     ax, y
add     ax, z
mov     temp2, ax
mov     ax, temp1,
imul    temp2
mov     w, ax
```

Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

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

Yet another example:

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

This can be converted to a set of four simple expressions:

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

You can convert these four simple expressions into the assembly language statements:

```
mov     ax, y       ;Compute AX := Y+Z
add     ax, z
mov     bx, a       ;Compute BX := A-B
sub     bx, b
mul     bx          ;Compute AX := AX * BX, this also sign
mov     bx, 10      ; extends AX into DX for idiv.
idiv    bx          ;Compute AX := AX / 10
mov     x, ax       ;Store result into X
```

The most important thing to keep in mind is that temporary values, if possible, should be kept in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the results. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

9.1.4 Commutative Operators

If "@" represents some operator, that operator is *commutative* if the following relationship is always true:

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

As you saw in the previous section, commutative operators are nice because the order of their operands is immaterial and this lets you rearrange a computation, often making that computation easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, you should always check to see if there is a better sequence you can use to improve the size or speed of your code. The following tables list the commutative and non-commutative operators you typically find in high level languages:

Table 46: Some Common Commutative Binary Operators

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& or &	Logical or bitwise AND
OR	|| or |	Logical or bitwise OR
XOR	^	(Logical or) Bitwise exclusive-OR
=	==	Equality
<>	!=	Inequality

Table 47: Some Common Noncommutative Binary Operators

Pascal	C/C++	Description
-	-	Subtraction
/ or DIV	/	Division
MOD	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

9.2 Logical (Boolean) Expressions

Consider the following expression from a Pascal program:

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

B is a boolean variable and the remaining variables are all integers.

How do we represent boolean variables in assembly language? Although it takes only a single bit to represent a boolean value, most assembly language programmers allocate a whole byte or word for this purpose. With a byte, there are 256 possible values we can use to represent the two values *true* and *false*. So which two values (or which two sets of values) do we use to represent these boolean values? Because of the machine's architecture, it's much easier to test for conditions like zero or not zero and positive or negative rather than to test for one of two particular boolean values. Most programmers (and, indeed, some programming languages like "C") choose zero to represent false and anything else to represent true. Some people prefer to represent true and false with one and zero (respectively) and not allow any other values. Others select 0FFFFh for true and 0 for false. You could also use a positive value for true and a negative value for false. All these mechanisms have their own advantages and drawbacks.

Using only zero and one to represent false and true offers one very big advantage: the 80x86 logical instructions (and, or, xor and, to a lesser extent, not) operate on these values exactly as you would expect. That is, if you have two boolean variables A and B, then the following instructions perform the basic logical operations on these two variables:

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

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

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

mov     ax, A       ;Note that the NOT instruction does not
not     ax          ; properly compute B := not A by itself.
and     ax, 1       ; I.e., (NOT 0)does not equal one.
mov     B, ax       ;B := not A;

mov     ax, A       ;Another way to do B := NOT A;
xor     ax, 1
mov     B, ax       ;B := not A;
```

Note, as pointed out above, that the not instruction will not properly compute logical negation. The bitwise not of zero is 0FFh and the bitwise not of one is 0FEh. Neither result is zero or one. However, by anding the result with one you get the proper result. Note that

you can implement the not operation more efficiently using the xor ax, 1 instruction since it only affects the L.O. bit.

As it turns out, using zero for false and anything else for true has a lot of subtle advantages. Specifically, the test for true or false is often implicit in the execution of any logical instruction. However, this mechanism suffers from a very big disadvantage: you cannot use the 80x86 and, or, xor, and not instructions to implement the boolean operations of the same name. Consider the two values 55h and 0AAh. They're both non-zero so they both represent the value true. However, if you logically and 55h and 0AAh together using the 80x86 and instruction, the result is zero. (True and true) should produce true, not false. A system that uses non-zero values to represent true and zero to represent false is an *arithmetic logical system*. A system that uses the two distinct values like zero and one to represent false and true is called a *boolean logical system*, or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression:

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

The simple expressions resulting from this expression might be:

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

The assembly language code for these expressions could be:

```
                mov     ax, x       ;See if X = Y and load zero or
                cmp     ax, y       ; one into AX to denote the result
                jnz     L1          ; of this comparison.
                mov     al, 1       ;X = Y
                jmp     L2
L1:             mov     al, 0       ;X <> Y
L2:
                mov     bx, A       ;See if A <= D and load zero or one
                cmp     bx, D       ; into BX to denote the result of
                jle     ST1         ; this comparison.
                mov     bl, 0
                jmp     L3
ST1:            mov     bl, 1
L3:
                and     bl, al      ;Temp := Temp and Temp2

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

As you can see, this is a rather unwieldy sequence of statements. One slight optimization you can use is to assume a result is going to be true or false and initialize the corresponding boolean result ahead of time:

```
                mov     bl, 0       ;Assume X <> Y
                mov     ax, x
                cmp     ax, Y
                jne     L1
                mov     bl, 1       ;X is equal to Y, so make this true.
L1:
                mov     bh, 0       ;Assume not (A <= D)
                mov     ax, A
                cmp     ax, D
                jnle    L2
                mov     bh, 1       ;A <= D so make this true
```

```
L2:
                and     bl, bh      ;Compute logical AND of results.

                mov     bh, 0       ;Assume (Z-A) = 5
                mov     ax, Z
                sub     ax, A
                cmp     ax, 5
                je      L3:
                mov     bh, 1       ;(Z-A) <> 5
L3:
                or      bl, bh      ;Logical OR of results.
                mov     B, bl       ;Save boolean result.
```

Of course, if you have an 80386 or later processor, you can use the set*cc* instructions to simplify this a bit:

```
                mov     ax, x
                cmp     ax, y
                sete    al          ;TEMP2 := X = Y

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

This code sequence is obviously much better than the previous one, but it will only execute on 80386 and later processors.

Another way to handle boolean expressions is to represent boolean values by states within your code. The basic idea is to forget maintaining a boolean variable throughout the execution of a code sequence and use the position within the code to determine the boolean result. Consider the following implementation of the above expression. First, let's rearrange the expression to be

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

This is perfectly legal since the or operation is commutative. Now consider the following implementation:

```
                mov     B, 1        ;Assume the result is true.
                mov     ax, Z       ;See if (Z-A) <> 5
                sub     ax, A       ;If this condition is true, the
                cmp     ax, 5       ; result is always true so there
                jne     Done        ; is no need to check the rest.

                mov     ax, X       ;If X <> Y, the result is false,
                cmp     ax, Y       ; no matter what A and D contain
                jne     SetBtoFalse

                mov     ax, A       ;Now see if A <= D.
                cmp     ax, D
                jle     Done        ;If so, quit.
SetBtoFalse:    mov     B, 0        ;If B is false, handle that here.
Done:
```

Notice that this section of code is a lot shorter than the first version above (and it runs on all processors). The previous translations did everything computationally. This version uses program flow logic to improve the code. It begins by assuming a true result and sets the B variable to true. It then checks to see if (Z-A) <> 5. If this is true the code branches to the done table because B is true no matter what else happens. If the program falls through to the mov ax, X instruction, we know that the result of the previous comparison is false. There is no need to save this result in a temporary since we implicitly know its value by the fact that we're executing the mov ax, X instruction. Likewise, the second group of statements above checks to see if X is equal to Y. If it is not, we already know the result is false

so this code jumps to the SetBtoFalse label. If the program begins executing the third set of statements above, we know that the first result was false and the second result was true; the position of the code guarantees this. Therefore, there is no need to maintain temporary boolean variables that keep track of the state of this computation.

Consider another example:

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

Computationally, this expression would yield a considerable amount of code. However, by using flow control you can reduce it to the following:

```
                mov     b, 0        ;Assume result is false.
                mov     ax, a       ;See if A = E.
                cmp     ax, e
                je      test2       ;If so, 1st subexpression is true.

                mov     ax, f       ;If not, check 2nd subexpression
                cmp     ax, d       ; to see if F <> D.
                je      Done        ;If so, we're done, else fall
                                    ; through to next tests.
Test2:          mov     ax, a       ;Does A <> B?
                cmp     ax, b
                jne     SetBto1     ;If so, we're done.

                mov     ax, f       ;If not, see if F = D.
                cmp     ax, d
                jne     Done

SetBto1:        mov     b, 1
Done:
```

There is one other difference between using control flow vs. computation logic: when using control flow methods, you may skip the majority of the instructions that implement the boolean formula. This is known as *short-circuit evaluation*. When using the computation model, even with the set*cc* instruction, you wind up executing most of the statements. Keep in mind that this is not necessarily a disadvantage. On pipelined processors it may be much faster to execute several additional instructions rather than flush the pipeline and prefetch queue. You may need to experiment with your code to determine the best solution.

When working with boolean expressions don't forget the that you might be able to optimize your code by simplifying those boolean expressions (see "Simplification of Boolean Functions" on page 52). You can use algebraic transformations (especially DeMorgan's theorems) and the mapping method to help reduce the complexity of an expression.

9.3 Multiprecision Operations

One big advantage of assembly language over HLLs is that assembly language does not limit the size of integers. For example, the C programming language defines a maximum of three different integer sizes: short int, int, and long int. On the PC, these are often 16 or 32 bit integers. Although the 80x86 machine instructions limit you to processing eight, sixteen, or thirty-two bit integers with a single instruction, you can always use more than one instruction to process integers of any size you desire. If you want 256 bit integer values, no problem. The following sections describe how extended various arithmetic and logical operations from 16 or 32 bits to as many bits as you please.

9.3.1 Multiprecision Addition Operations

The 80x86 add instruction adds two 8, 16, or 32 bit numbers[1]. After the execution of the add instruction, the 80x86 carry flag is set if there is an overflow out of the H.O. bit of

1. As usual, 32 bit arithmetic is available only on the 80386 and later processors.

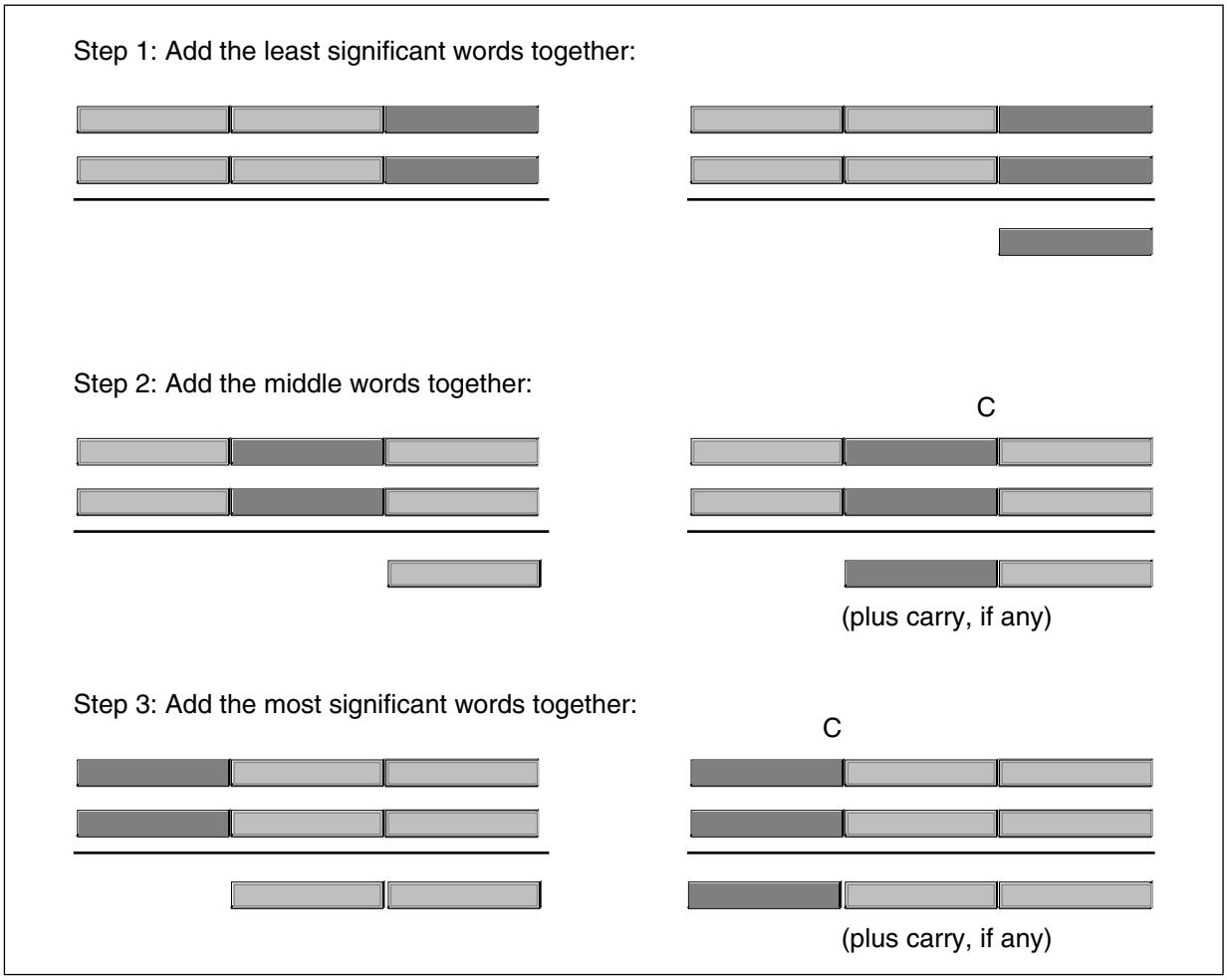


Figure 8.1 Multiprecision (48-bit) Addition

the sum. You can use this information to do multiprecision addition operations. Consider the way you manually perform a multidigit (multiprecision) addition operation:

Step 1: Add the least significant digits together:

```
 289                              289
+456          produces           +456
----                             ----
                                    5 with carry 1.
```

Step 2: Add the next significant digits plus the carry:

```
  1 (previous carry)
 289                                      289
+456                 produces             +456
----                                      ----
   5                                        45 with carry 1.
```

Step 3: Add the most significant digits plus the carry:

```
                                          1 (previous carry)
 289                                      289
+456                 produces             +456
----                                      ----
  45                                      745
```

The 80x86 handles extended precision arithmetic in an identical fashion, except instead of adding the numbers a digit at a time, it adds them a byte or a word at a time. Consider the three-word (48 bit) addition operation in Figure 8.1.

The add instruction adds the L.O. words together. The adc (add with carry) instruction adds all other word pairs together. The adc instruction adds two operands plus the carry flag together producing a word value and (possibly) a carry.

For example, suppose that you have two thirty-two bit values you wish to add together, defined as follows:

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

Suppose, also, that you want to store the sum in a third variable, Z, that is likewise defined with the dword directive. The following 80x86 code will accomplish this task:

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

Remember, these variables are declared with the dword directive. Therefore the assembler will not accept an instruction of the form mov ax, X because this instruction would attempt to load a 32 bit value into a 16 bit register. Therefore this code uses the *word ptr* coercion operator to coerce symbols X, Y, and Z to 16 bits. The first three instructions add the L.O. words of X and Y together and store the result at the L.O. word of Z. The last three instructions add the H.O. words of X and Y together, along with the carry out of the L.O. word, and store the result in the H.O. word of Z. Remember, address expressions of the form "X+2" access the H.O. word of a 32 bit entity. This is due to the fact that the 80x86 address space addresses bytes and it takes two consecutive bytes to form a word.

Of course, if you have an 80386 or later processor you needn't go through all this just to add two 32 bit values together, since the 80386 directly supports 32 bit operations. However, if you wanted to add two *64 bit* integers together on the 80386, you would still need to use this technique.

You can extend this to any number of bits by using the adc instruction to add in the higher order words in the values. For example, to add together two 128 bit values, you could use code that looks something like the following:

```
BigVal1         dword   0,0,0,0         ;Four double words in 128 bits!
BigVal2         dword   0,0,0,0
BigVal3         dword   0,0,0,0
                  .
                  .
                  .
                mov     eax, BigVal1    ;No need for dword ptr operator since
                add     eax, BigVal2    ; these are dword variables.
                mov     BigVal3, eax

                mov     eax, BigVal1+4  ;Add in the values from the L.O.
                adc     eax, BigVal2+4  ; entity to the H.O. entity using
                mov     BigVal3+4, eax  ; the ADC instruction.

                mov     eax, BigVal1+8
                adc     eax, BigVal2+8
                mov     BigVal3+8, eax

                mov     eax, BigVal1+12
                adc     eax, BigVal2+12
                mov     BigVal3+12, eax
```

9.3.2 Multiprecision Subtraction Operations

Like addition, the 80x86 performs multi-byte subtraction the same way you would manually, except it subtracts whole bytes , words, or double words at a time rather than decimal digits. The mechanism is similar to that for the add operation, You use the sub instruction on the L.O. byte/word/double word and the sbb instruction on the high order

values. The following example demonstrates a 32 bit subtraction using the 16 bit registers on the 8086:

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

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

The following example demonstrates a 128-bit subtraction using the 80386 32 bit register set:

```
BigVal1         dword       0,0,0,0         ;Four double words in 128 bits!
BigVal2         dword       0,0,0,0
BigVal3         dword       0,0,0,0
                  .
                  .
                  .
                mov         eax, BigVal1    ;No need for dword ptr operator since
                sub         eax, BigVal2    ; these are dword variables.
                mov         BigVal3, eax

                mov         eax, BigVal1+4  ;Subtract the values from the L.O.
                sbb         eax, BigVal2+4  ; entity to the H.O. entity using
                mov         BigVal3+4, eax  ; the SUB and SBB instructions.

                mov         eax, BigVal1+8
                sbb         eax, BigVal2+8
                mov         BigVal3+8, eax

                mov         eax, BigVal1+12
                sbb         eax, BigVal2+12
                mov         BigVal3+12, eax
```

9.3.3 Extended Precision Comparisons

Unfortunately, there isn't a "compare with borrow" instruction that can be used to perform extended precision comparisons. Since the cmp and sub instructions perform the same operation, at least as far as the flags are concerned, you'd probably guess that you could use the sbb instruction to synthesize an extended precision comparison; however, you'd only be partly right. There is, however, a better way.

Consider the two unsigned values 2157h and 1293h. The L.O. bytes of these two values do not affect the outcome of the comparison. Simply comparing 21h with 12h tells us that the first value is greater than the second. In fact, the only time you ever need to look at both bytes of these values is if the H.O. bytes are equal. In all other cases comparing the H.O. bytes tells you everything you need to know about the values. Of course, this is true for any number of bytes, not just two. The following code compares two signed 64 bit integers on an 80386 or later processor:

```
; This sequence transfers control to location "IsGreater" if
; QwordValue > QwordValue2. It transfers control to "IsLess" if
; QwordValue < QwordValue2. It falls though to the instruction
; following this sequence if QwordValue = QwordValue2. To test for
; inequality, change the "IsGreater" and "IsLess" operands to "NotEqual"
; in this code.

                mov       eax, dword ptr QWordValue+4    ;Get H.O. dword
                cmp       eax, dword ptr QWordValue2+4
                jg        IsGreater
                jl        IsLess
                mov       eax, dword ptr QWordValue
                cmp       eax, dword ptr QWordValue2
                jg        IsGreater
                jl        IsLess
```

To compare unsigned values, simply use the ja and jb instructions in place of jg and jl.

You can easily synthesize any possible comparison from the sequence above, the following examples show how to do this. These examples do signed comparisons, substitute ja, jae, jb, and jbe for jg, jge, jl, and jle (respectively) to do unsigned comparisons.

```
QW1             qword    ?
QW2             qword    ?

dp              textequ  <dword ptr>

; 64 bit test to see if QW1 < QW2 (signed).
; Control transfers to "IsLess" label if QW1 < QW2. Control falls
; through to the next statement if this is not true.

                mov      eax, dp QW1+4          ;Get H.O. dword
                cmp      eax, dp QW2+4
                jg       NotLess
                jl       IsLess
                mov      eax, dp QW1            ;Fall through to here if H.O.
                cmp      eax, dp QW2            ; dwords are equal.
                jl       IsLess
NotLess:

; 64 bit test to see if QW1 <= QW2 (signed).

                mov      eax, dp QW1+4          ;Get H.O. dword
                cmp      eax, dp QW2+4
                jg       NotLessEq
                jl       IsLessEq
                mov      eax, dp QW1
                cmp      eax, dword ptr QW2
                jle      IsLessEq
NotLessEQ:

; 64 bit test to see if QW1 >QW2 (signed).

                mov      eax, dp QW1+4          ;Get H.O. dword
                cmp      eax, dp QW2+4
                jg       IsGtr
                jl       NotGtr
                mov      eax, dp QW1            ;Fall through to here if H.O.
                cmp      eax, dp QW2            ; dwords are equal.
                jg       IsGtr
NotGtr:

; 64 bit test to see if QW1 >= QW2 (signed).

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

; 64 bit test to see if QW1 = QW2 (signed or unsigned). This code branches
; to the label "IsEqual" if QW1 = QW2. It falls through to the next instruction
; if they are not equal.

                mov      eax, dp QW1+4          ;Get H.O. dword
                cmp      eax, dp QW2+4
                jne      NotEqual
                mov      eax, dp QW1
                cmp      eax, dword ptr QW2
                je       IsEqual
NotEqual:
```

```
; 64 bit test to see if QW1 <> QW2 (signed or unsigned). This code branches
; to the label "NotEqual" if QW1 <> QW2. It falls through to the next
; instruction if they are equal.

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jne     NotEqual
                mov     eax, dp QW1
                cmp     eax, dword ptr QW2
                jne     NotEqual
```

9.3.4 Extended Precision Multiplication

Although a 16x16 or 32x32 multiply is usually sufficient, there are times when you may want to multiply larger values together. You will use the 80x86 single operand mul and imul instructions for extended precision multiplication.

Not surprisingly (in view of how adc and sbb work), you use the same techniques to perform extended precision multiplication on the 80x86 that you employ when manually multiplying two values.

Consider a simplified form of the way you perform multi-digit multiplication by hand:

```
1) Multiply the first two
   digits together (5*3):

       123
        45
       ---
        15

2) Multiply 5*2:

       123
        45
        ---
        15
       10

 3) Multiply 5*1:

       123
        45
       ---
        15
       10
       5

 4) 4*3:

       123
        45
        ---
        15
       10
       5
       12

 5) Multiply 4*2:

        123
         45
        ---
         15
        10
        5
        12
        8

 6) 4*1:

        123
         45
        ---
         15
        10
        5
        12
        8
       4

 7) Add all the partial products together:

        123
         45
        ---
         15
        10
        5
        12
        8
       4
       ------
       5535
```

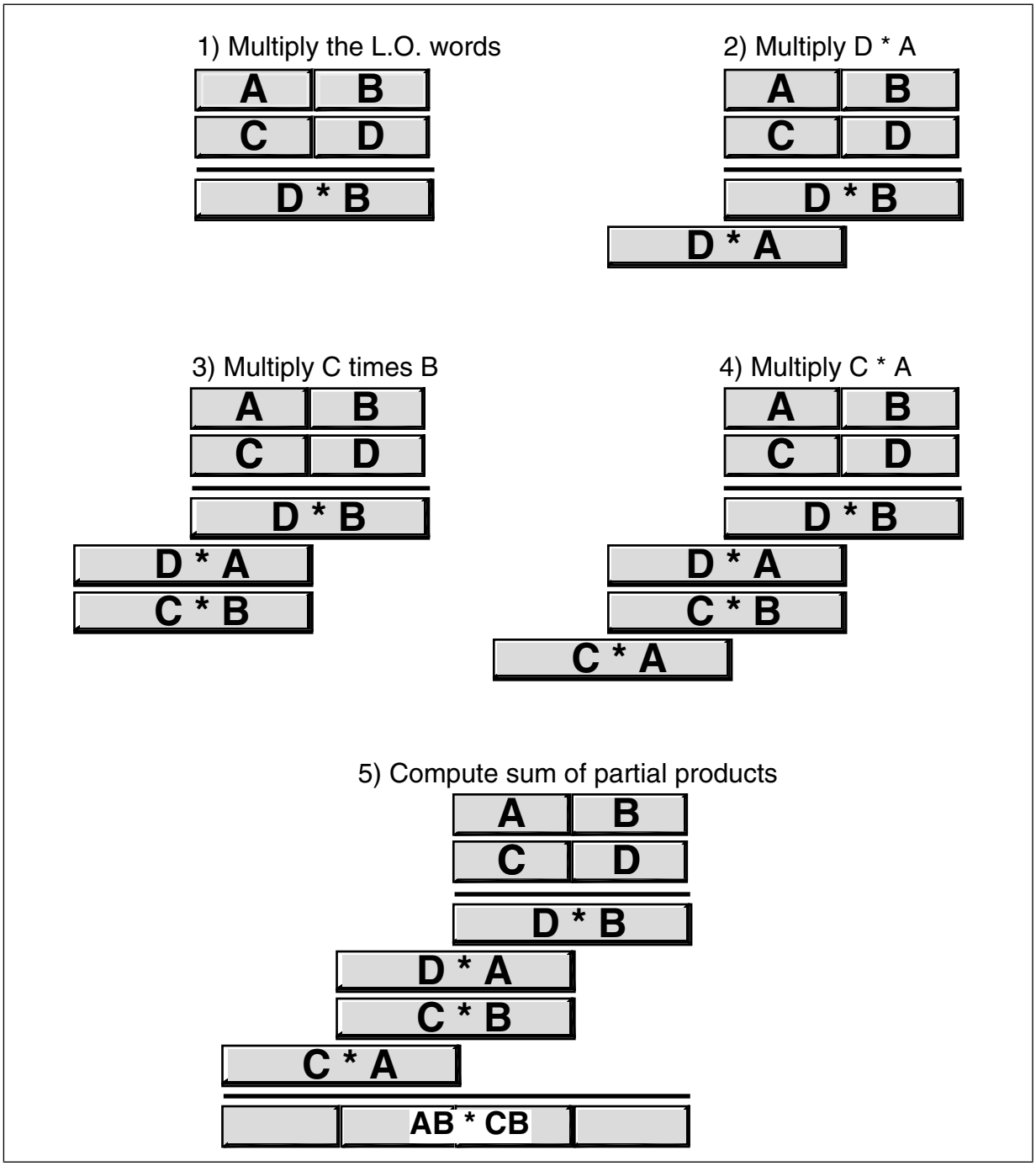


Figure 8.2 Multiprecision Multiplication

The 80x86 does extended precision multiplication in the same manner except that it works with bytes, words, and double words rather than digits. Figure 8.2 shows how this works.

Probably the most important thing to remember when performing an extended precision multiplication is that you must also perform a multiple precision addition at the same time. Adding up all the partial products requires several additions that will produce the result. The following listing demonstrates the proper way to multiply two 32 bit values on a 16 bit processor:

Note: Multiplier and Multiplicand are 32 bit variables declared in the data segment via the dword directive. Product is a 64 bit variable declared in the data segment via the qword directive.

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

; Multiply the L.O. word of Multiplier times Multiplicand:

                mov     ax, word ptr Multiplier
                mov     bx, ax                      ;Save Multiplier val
                mul     word ptr Multiplicand       ;Multiply L.O. words
                mov     word ptr Product, ax        ;Save partial product
                mov     cx, dx                      ;Save H.O. word

                mov     ax, bx                      ;Get Multiplier in BX
                mul     word ptr Multiplicand+2     ;Multiply L.O. * H.O.
                add     ax, cx                      ;Add partial product
                adc     dx, 0                       ;Don't forget carry!
                mov     bx, ax                      ;Save partial product
                mov     cx, dx                      ; for now.

; Multiply the H.O. word of Multiplier times Multiplicand:

                mov     ax, word ptr Multiplier+2   ;Get H.O. Multiplier
                mul     word ptr Multiplicand       ;Times L.O. word
                add     ax, bx                      ;Add partial product
                mov     word ptr product+2, ax      ;Save partial product
                adc     cx, dx                      ;Add in carry/H.O.!

                mov     ax, word ptr Multiplier+2   ;Multiply the H.O.
                mul     word ptr Multiplicand+2     ; words together.
                add     ax, cx                      ;Add partial product
                adc     dx, 0                       ;Don't forget carry!
                mov     word ptr Product+4, ax      ;Save partial product
                mov     word ptr Product+6, dx

                pop     bx
                pop     cx
                pop     dx
                pop     ax
                ret
Multiply        endp
```

One thing you must keep in mind concerning this code, it only works for unsigned operands. Multiplication of signed operands appears in the exercises.

9.3.5 Extended Precision Division

You cannot synthesize a general n-bit/m-bit division operation using the div and idiv instructions. Such an operation must be performed using a sequence of shift and subtract instructions. Such an operation is extremely messy. A less general operation, dividing an n bit quantity by a 32 bit (on the 80386 or later) or 16 bit quantity is easily synthesized using the div instruction. The following code demonstrates how to divide a 64 bit quantity by a 16 bit divisor, producing a 64 bit quotient and a 16 bit remainder:

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

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

; Divide a 64 bit quantity by a 16 bit quantity:

Divide64        proc     near

                mov      ax, word ptr dividend+6
                sub      dx, dx
```

```
                        div     divisor
                        mov     word ptr Quotient+6, ax
                        mov     ax, word ptr dividend+4
                        div     divisor
                        mov     word ptr Quotient+4, ax
                        mov     ax, word ptr dividend+2
                        div     divisor
                        mov     word ptr Quotient+2, ax
                        mov     ax, word ptr dividend
                        div     divisor
                        mov     word ptr Quotient, ax
                        mov     Modulo, dx
                        ret
Divide64                endp
cseg                    ends
```

This code can be extended to any number of bits by simply adding additional mov / div / mov instructions at the beginning of the sequence. Of course, on the 80386 and later processors you can divide by a 32 bit value by using edx and eax in the above sequence (with a few other appropriate adjustments).

If you need to use a divisor larger than 16 bits (32 bits on an 80386 or later), you're going to have to implement the division using a shift and subtract strategy. Unfortunately, such algorithms are very slow. In this section we'll develop two division algorithms that operate on an arbitrary number of bits. The first is slow but easier to understand, the second is quite a bit faster (in general).

As for multiplication, the best way to understand how the computer performs division is to study how you were taught to perform long division by hand. Consider the operation 3456/12 and the steps you would take to manually perform this operation:

```
12 ⌈3456        (1) 12 goes into 34 two times.
   24
```

```
     2
12 ⌈3456        (2) Subtract 24 from 35
   24             and drop down the
   ---            105.
   105
```

```
     2
12 ⌈3456        (3) 12 goes into 105
   24             eight times.
   ---
   105
    96
```

```
     28
12 ⌈3456        (4) Subtract 96 from 105
   24             and drop down the 96.
   ---
   105
    96
    --
    96
```

```
     28
12 ⌈3456        (5) 12 goes into 96
   24             exactly eight times.
   ---
   105
    96
    --
    96
    96
```

```
     288
12 ⌈3456        (6) Therefore, 12
   24             goes into 3456
   ---            exactly 288 times.
   105
    96
    --
    96
    96
```

This algorithm is actually easier in binary since at each step you do not have to guess how many times 12 goes into the remainder nor do you have to multiply 12 by your guess to obtain the amount to subtract. At each step in the binary algorithm the divisor goes into the remainder exactly zero or one times. As an example, consider the division of 27 (11011) by three (11):

```
11 ⌈11011       11 goes into 11 one time.
   11
```

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

Subtract out the 11 and bring down the zero.

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

11 goes into 00 zero times.

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

Subtract out the zero and bring down the one.

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

11 goes into 01 zero times.

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

Subtract out the zero and bring down the one.

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

11 goes into 11 one time.

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

This produces the final result of 1001.

There is a novel way to implement this binary division algorithm that computes the quotient and the remainder at the same time. The algorithm is the following:

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

       Remainder:Quotient := Remainder:Quotient SHL 1;
       if Remainder >= Divisor then

                 Remainder := Remainder - Divisor;
                 Quotient := Quotient + 1;

       endif
endfor
```

NumberBits is the number of bits in the Remainder, Quotient, Divisor, and Dividend variables. Note that the Quotient := Quotient + 1 statement sets the L.O. bit of Quotient to one since this algorithm previously shifts Quotient one bit to the left. The 80x86 code to implement this algorithm is

```
; Assume Dividend (and Quotient) is DX:AX, Divisor is in CX:BX,
; and Remainder is in SI:DI.

                mov     bp, 32      ;Count off 32 bits in BP
                sub     si, si      ;Set remainder to zero
                sub     di, di
BitLoop:        shl     ax, 1       ;See the section on shifts
                rcl     dx, 1       ; that describes how this
                rcl     di, 1       ; 64 bit SHL operation works
                rcl     si, 1
                cmp     si, cx      ;Compare H.O. words of Rem,
                ja      GoesInto    ; Divisor.
                jb      TryNext
                cmp     di, bx      ;Compare L.O. words.
                jb      TryNext

GoesInto:       sub     di, bx      ;Remainder := Remainder -
                sbb     si, cx      ;                Divisor
                inc     ax          ;Set L.O. bit of AX
TryNext:        dec     bp          ;Repeat 32 times.
                jne     BitLoop
```

This code looks short and simple, but there are a few problems with it. First, it does not check for division by zero (it will produce the value 0FFFFFFFFh if you attempt to divide by zero), it only handles unsigned values, and it is very slow. Handling division by zero is very simple, just check the divisor against zero prior to running this code and return an appropriate error code if the divisor is zero. Dealing with signed values is equally simple, you'll see how to do that in a little bit. The performance of this algorithm, however, leaves a lot to be desired. Assuming one pass through the loop takes about 30 clock cycles[2], this algorithm would require almost 1,000 clock cycles to complete! That's an order of magnitude worse than the DIV/IDIV instructions on the 80x86 that are among the slowest instructions on the 80x86.

There is a technique you can use to boost the performance of this division by a fair amount: check to see if the divisor variable really uses 32 bits. Often, even though the divisor is a 32 bit variable, the value itself fits just fine into 16 bits (i.e., the H.O. word of Divisor is zero). In this special case, that occurs frequently, you can use the DIV instruction which is much faster.

9.3.6 Extended Precision NEG Operations

Although there are several ways to negate an extended precision value, the shortest way is to use a combination of neg and sbb instructions. This technique uses the fact that neg subtracts its operand from zero. In particular, it sets the flags the same way the sub

2. This will vary depending upon your choice of processor.

instruction would if you subtracted the destination value from zero. This code takes the following form:

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

The sbb instruction decrements dx if there is a borrow out of the L.O. word of the negation operation (which always occurs unless ax is zero).

To extend this operation to additional bytes, words, or double words is easy; all you have to do is start with the H.O. memory location of the object you want to negate and work towards the L.O. byte. The following code computes a 128 bit negation on the 80386 processor:

```
Value           dword   0,0,0,0     ;128 bit integer.
                 .
                 .
                 .
                neg     Value+12    ;Neg H.O. dword
                neg     Value+8     ;Neg previous dword in memory.
                sbb     Value+12, 0 ;Adjust H.O. dword
                neg     Value+4     ;Neg the second dword in object.
                sbb     Value+8, 0  ;Adjust 3rd dword in object.
                sbb     Value+12, 0 ;Carry any borrow through H.O. word.
                neg     Value       ;Negate L.O. word.
                sbb     Value+4, 0  ;Adjust 2nd dword in object.
                sbb     Value+8, 0  ;Adjust 3rd dword in object.
                sbb     Value+12, 0 ;Carry any borrow through H.O. word.
```

Unfortunately, this code tends to get really large and slow since you need to propogate the carry through all the H.O. words after each negate operation. A simpler way to negate larger values is to simply subract that value from zero:

```
Value           dword   0,0,0,0,0   ;160 bit integer.
                 .
                 .
                 .
                mov     eax, 0
                sub     eax, Value
                mov     Value, eax
                mov     eax, 0
                sbb     eax, Value+4
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+8
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+12
                mov     Value+12, ax
                mov     eax, 0
                sbb     eax, Value+16
                mov     Value+16, ax
```

9.3.7 Extended Precision AND Operations

Performing an n-word and operation is very easy – simply and the corresponding words between the two operands, saving the result. For example, to perform the and operation where all three operands are 32 bits long, you could use the following code:

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

This technique easily extends to any number of words, all you need to is logically and the corresponding bytes, words, or double words in the corresponding operands.

9.3.8 Extended Precision OR Operations

Multi-word logical or operations are performed in the same way as multi-word and operations. You simply or the corresponding words in the two operand together. For example, to logically or two 48 bit values, use the following code:

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

9.3.9 Extended Precision XOR Operations

Extended precision xor operations are performed in a manner identical to and/or – simply xor the corresponding words in the two operands to obtain the extended precision result. The following code sequence operates on two 64 bit operands, computes their exclusive-or, and stores the result into a 64 bit variable. This example uses the 32 bit registers available on 80386 and later processors.

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

9.3.10 Extended Precision NOT Operations

The not instruction inverts all the bits in the specified operand. It does not affect any flags (therefore, using a conditional jump after a not instruction has no meaning). An extended precision not is performed by simply executing the not instruction on all the affected operands. For example, to perform a 32 bit not operation on the value in (dx:ax), all you need to do is execute the instructions:

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

Keep in mind that if you execute the not instruction twice, you wind up with the original value. Also note that exclusive-oring a value with all ones (0FFh, 0FFFFh, or 0FF..FFh) performs the same operation as the not instruction.

9.3.11 Extended Precision Shift Operations

Extended precision shift operations require a shift and a rotate instruction. Consider what must happen to implement a 32 bit shl using 16 bit operations:

1) A zero must be shifted into bit zero.

2) Bits zero through 14 are shifted into the next higher bit.

3) Bit 15 is shifted into bit 16.

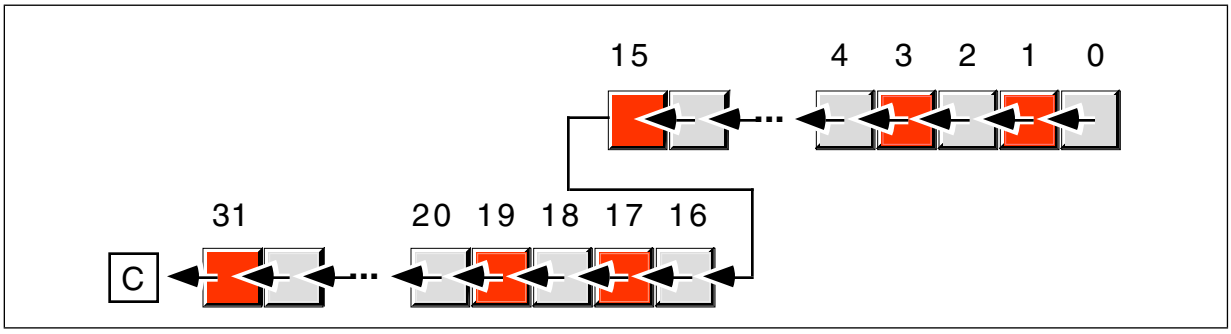


Figure 8.3 32-bit Shift Left Operation

4) Bits 16 through 30 must be shifted into the next higher bit.

5) Bit 31 is shifted into the carry flag.

The two instructions you can use to implement this 32 bit shift are shl and rcl. For example, to shift the 32 bit quantity in (dx:ax) one position to the left, you'd use the instructions:

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

Note that you can only shift an extended precision value one bit at a time. You cannot shift an extended precision operand several bits using the cl register or an immediate value greater than one as the count using this technique

To understand how this instruction sequence works, consider the operation of these instructions on an individual basis. The shl instruction shifts a zero into bit zero of the 32 bit operand and shifts bit 15 into the carry flag. The rcl instruction then shifts the carry flag into bit 16 and then shifts bit 31 into the carry flag. The result is exactly what we want.

To perform a shift left on an operand larger than 32 bits you simply add additional rcl instructions. An extended precision shift left operation always starts with the least significant word and each succeeding rcl instruction operates on the next most significant word. For example, to perform a 48 bit shift left operation on a memory location you could use the following instructions:

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

If you need to shift your data by two or more bits, you can either repeat the above sequence the desired number of times (for a constant number of shifts) or you can place the instructions in a loop to repeat them some number of times. For example, the following code shifts the 48 bit value Operand to the left the number of bits specified in cx:

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

You implement shr and sar in a similar way, except you must start at the H.O. word of the operand and work your way down to the L.O. word:

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

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

There is one major difference between the extended precision shifts described here and their 8/16 bit counterparts – the extended precision shifts set the flags differently than

the single precision operations. For example, the zero flag is set if the last rotate instruction produced a zero result, not if the entire shift operation produced a zero result. For the shift right operations, the overflow, and sign flags aren't set properly (they are set properly for the left shift operation). Additional testing will be required if you need to test one of these flags after an extended precision shift operation. Fortunately, the carry flag is the flag most often tested after a shift operation and the extended precision shift instructions properly set this flag.

The shld and shrd instructions let you efficiently implement multiprecision shifts of several bits on 80386 and later processors. Consider the following code sequence:

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

Recall that the shld instruction shifts bits from its second operand into its first operand. Therefore, the first shld instruction above shifts the bits from ShiftMe+4 into ShiftMe+8 *without affecting the value in ShiftMe+4.* The second shld instruction shifts the bits from ShiftMe into ShiftMe+4. Finally, the shl instruction shifts the L.O. double word the appropriate amount. There are two important things to note about this code. First, unlike the other extended precision shift left operations, this sequence works from the H.O. double word down to the L.O. double word. Second, the carry flag does not contain the carry out of the H.O. shift operation. If you need to preserve the carry flag at that point, you will need to push the flags after the first shld instruction and pop the flags after the shl instruction.

You can do an extended precision shift right operation using the shrd instruction. It works almost the same way as the code sequence above except you work from the L.O. double word to the H.O. double word. The solution is left as an exercise at the end of this chapter.

9.3.12 Extended Precision Rotate Operations

The rcl and rcr operations extend in a manner almost identical to that for shl and shr . For example, to perform 48 bit rcl and rcr operations, use the following instructions:

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

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

The only difference between this code and the code for the extended precision shift operations is that the first instruction is a rcl or rcr rather than a shl or shr instruction.

Performing an extended precision rol or ror instruction isn't quite as simple an operation. The 8086 extended precision versions of these instructions appear in the exercises. On the 80386 and later processors, you can use the bt, shld, and shrd instructions to easily implement an extended precision rol or ror instruction. The following code shows how to use the shld instruction to do an extended precision rol:

```
; Compute ROL EDX:EAX, 4

                mov      ebx, edx
                shld     edx, eax, 4
                shld     eax, ebx, 4
                bt       eax, 0       ;Set carry flag, if desired.
```

An extended precision ror instruction is similar; just keep in mind that you work on the L.O. end of the object first and the H.O. end last.

9.4 Operating on Different Sized Operands

Occasionally you may need to compute some value on a pair of operands that are not the same size. For example, you may need to add a word and a double word together or subtract a byte value from a word value. The solution is simple: just extend the smaller operand to the size of the larger operand and then do the operation on two similarly sized operands. For signed operands, you would sign extend the smaller operand to the same size as the larger operand; for unsigned values, you zero extend the smaller operand. This works for any operation, although the following examples demonstrate this for the addition operation.

To extend the smaller operand to the size of the larger operand, use a sign extension or zero extension operation (depending upon whether you're adding signed or unsigned values). Once you've extended the smaller value to the size of the larger, the addition can proceed. Consider the following code that adds a byte value to a word value:

```
var1            byte    ?
var2            word    ?

Unsigned addition:          Signed addition:

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

In both cases, the byte variable was loaded into the al register, extended to 16 bits, and then added to the word operand. This code works out really well if you can choose the order of the operations (e.g., adding the eight bit value to the sixteen bit value). Sometimes, you cannot specify the order of the operations. Perhaps the sixteen bit value is already in the ax register and you want to add an eight bit value to it. For unsigned addition, you could use the following code:

```
                mov     ax, var2    ;Load 16 bit value into AX
                 .                  ;Do some other operations leaving
                 .                  ; a 16 bit quantity in AX.
                add     al, var1    ;Add in the 8 bit value.
                adc     ah, 0       ;Add carry into the H.O. word.
```

The first add instruction in this example adds the byte at var1 to the L.O. byte of the value in the accumulator. The adc instruction above adds the carry out of the L.O. byte into the H.O. byte of the accumulator. Care must be taken to ensure that this adc instruction is present. If you leave it out, you may not get the correct result.

Adding an eight bit signed operand to a sixteen bit signed value is a little more difficult. Unfortunately, you cannot add an immediate value (as above) to the H.O. word of ax. This is because the H.O. extension byte can be either 00h or 0FFh. If a register is available, the best thing to do is the following:

```
                mov     bx, ax      ;BX is the available register.
                mov     al, var1
                cbw
                add     ax, bx
```

If an extra register is not available, you might try the following code:

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

Of course, if another register isn't available, you could always push one onto the stack and save it while you're performing the operation, e.g.,

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

Another alternative is to store the 16 bit value in the accumulator into a memory location and then proceed as before:

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

All the examples above added a byte value to a word value. By zero or sign extending the smaller operand to the size of the larger operand, you can easily add any two different sized variables together. Consider the following code that adds a signed byte operand to a signed double word:

```
var1            byte    ?
var2            dword   ?

                mov     al, var1
                cbw
                cwd                             ;Extend to 32 bits in DX
                add     ax, word ptr var2
                adc     dx, word ptr var2+2
```

Of course, if you have an 80386 or later processor, you could use the following code:

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

An example more applicable to the 80386 is adding an eight bit value to a quadword (64 bit) value, consider the following code:

```
BVal            byte    -1
QVal            qword     1

                movsx   eax, BVal
                cdq
                add     eax, dword ptr QVal
                adc     edx, dword ptr QVal+4
```

For additional examples, see the exercises at the end of this chapter.

9.5 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncracies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as "tricky programming" that you should always avoid in well written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well-known and commonly found in assembly language programs. Some of them can be really tricky, but a good number of them are simply "tricks of the trade." This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

9.5.1 Multiplying Without MUL and IMUL

If you take a quick look at the timing for the multiply instruction, you'll notice that the execution time for this instruction is rather long. Only the div and idiv instructions take longer on the 8086. When multiplying by a constant, you can avoid the performance penalty of the mul and imul instructions by using shifts, additions, and subtractions to perform the multiplication.

Remember, a shl operation performs the same operation as multiplying the specified operand by two. Shifting to the left two bit positions multiplies the operand by four. Shifting to the left three bit positions multiplies the operand by eight. In general, shifting an operand to the left n bits multiplies it by 2^n. Any value can be multiplied by some constant using a series of shifts and adds or shifts and subtractions. For example, to multiply the ax register by ten, you need only multiply it by eight and then add in two times the original value. That is, 10*ax = 8*ax + 2*ax. The code to accomplish this is

```
shl     ax, 1       ;Multiply AX by two
mov     bx, ax      ;Save 2*AX for later
shl     ax, 1       ;Multiply AX by four
shl     ax, 1       ;Multiply AX by eight
add     ax, bx      ;Add in 2*AX to get 10*AX
```

The ax register (or just about any register, for that matter) can be multiplied by most constant values much faster using shl than by using the mul instruction. This may seem hard to believe since it only takes two instructions to compute this product:

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

However, if you look at the timings, the shift and add example above requires fewer clock cycles on most processors in the 80x86 family than the mul instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it. Of course, on the later 80x86 processors, the mul instruction is quite a bit faster than the earlier processors, but the shift and add scheme is generally faster on these processors as well.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by seven:

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

This follows directly from the fact that ax*7 = (ax*8)-ax.

A common error made by beginning assembly language students is subtracting or adding one or two rather than ax*1 or ax*2. The following does *not* compute ax*7:

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

It computes (8*ax)-1, something entirely different (unless, of course, ax = 1). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

You can also use the lea instruction to compute certain products on 80386 and later processors. The trick is to use the 80386 scaled index mode. The following examples demonstrate some simple cases:

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

```
lea     eax, [edx][edx*8]       ;EAX := EDX * 9
```

9.5.2 Division Without DIV and IDIV

Much as the shl instruction can be used for simulating a multiplication by some power of two, the shr and sar instructions can be used to simulate a division by a power of two. Unfortunately, you cannot use shifts, additions, and subtractions to perform a division by an arbitrary constant as easily as you can use these instructions to perform a multiplication operation.

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. The multiply instruction is marginally faster than the divide instruction; multiplying by a reciprocal is almost always faster than division.

Now you're probably wondering "how does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by one tenth, there is no way you can load the value $1/10^{th}$ into an 80x86 register prior to performing the division. However, we could multiply $1/10^{th}$ by 10, perform the multiplication, and then divide the result by ten to get the final result. Of course, this wouldn't buy you anything at all, in fact it would make things worse since you're now doing a multiplication by ten as well as a division by ten. However, suppose you multiply 1/10th by 65,536 (6553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides ax by ten:

```
mov     dx, 6554    ;Round (65,536/10)
mul     dx
```

This code leaves ax/10 in the dx register.

To understand how this works, consider what happens when you multiply ax by 65,536 (10000h). This simply moves ax into dx and sets ax to zero. Multiplying by 6,554 (65,536 divided by ten) puts ax divided by ten into the dx register. Since mul is marginally faster than div , this technique runs a little faster than using a straight division.

Multiplying by the reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal only pays off if you perform the division many, many times (by the same value).

9.5.3 Using AND to Compute Remainders

The and instruction can be used to quickly compute remainders of the form:

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

To compute a remainder using the and instruction, simply and the operand with the value 2^n-1. For example, to compute ax = ax mod 8 simply use the instruction:

```
and     ax, 7
```

Additional examples:

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

9.5.4 Implementing Modulo-n Counters with AND

If you want to implement a counter variable that counts up to 2^n-1 and then resets to zero, simply using the following code:

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

where nBits is a binary value containing n one bits right justified in the number. For example, to create a counter that cycles between zero and fifteen, you could use the following:

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

9.5.5 Testing an Extended Precision Value for 0FFFF..FFh

The and instruction can be used to quickly check a multi-word value to see if it contains ones in all its bit positions. Simply load the first word into the ax register and then logically and the ax register with all the remaining words in the data structure. When the and operation is complete, the ax register will contain 0FFFFh if and only if all the words in that structure contained 0FFFFh. E.g.,

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

9.5.6 TEST Operations

Remember, the test instruction is an and instruction that doesn't retain the results of the and operation (other than the flag settings). Therefore, many of the comments concerning the and operation (particularly with respect to the way it affects the flags) also hold for the test instruction. However, since the test instruction doesn't affect the destination operand, multiple bit tests may be performed on the same value. Consider the following code:

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

This code can be used to successively test each bit in the ax register (or any other operand for that matter). Note that you cannot use the test/cmp instruction pair to test for a specific value within a string of bits (as you can with the and/cmp instructions). Since test doesn't strip out any unwanted bits, the cmp instruction would actually be comparing the original value rather than the stripped value. For this reason, you'll normally use the test instruction to see if a single bit is set or if one or more bits out of a group of bits are set. Of course, if you have an 80386 or later processor, you can also use the bt instruction to test individual bits in an operand.

Another important use of the test instruction is to efficiently compare a register against zero. The following test instruction sets the zero flag if and only if ax contains zero (anything anded with itself produces its original value; this sets the zero flag only if that value is zero):

```
test ax, ax
```

The test instruction is shorter than

```
cmp     ax, 0
```

or

```
cmp     eax, 0
```

though it is no better than

```
cmp     al, 0
```

Note that you can use the and and or instructions to test for zero in a fashion identical to test. However, on pipelined processors like the 80486 and Pentium chips, the test instruction is less likely to create a hazard since it does not store a result back into its destination register.

9.5.7 Testing Signs with the XOR Instruction

Remember the pain associated with a multi-precision signed multiplication operation? You need to determine the sign of the result, take the absolute value of the operands, multiply them, and then adjust the sign of the result as determined before the multiplication operation. The sign of the product of two numbers is simply the exclusive-or of their signs before performing the multiplication. Therefore, you can use the xor instruction to determine the sign of the product of two extended precision numbers. E.g.,

```
32x32 Multiply:
                mov     al, byte ptr Oprnd1+3
                xor     al, byte ptr Oprnd2+3
                mov     cl, al                  ;Save sign

; Do the multiplication here (don't forget to take the absolute
; value of the two operands before performing the multiply).

                .
                .
                .

; Now fix the sign.

                cmp     cl, 0                   ;Check sign bit
                jns     ResultIsPos

; Negate the product here.

                .
                .
                .

ResultIsPos:
```

9.6 Masking Operations

A *mask* is a value used to force certain bits to zero or one within some other value. A mask typically affects certain bits in an operand (forcing them to zero or one) and leaves other bits unaffected. The appropriate use of masks allows you to *extract* bits from a value, *insert* bits into a value, and pack or unpacked a packed data type. The following sections describe these operations in detail.

9.6.1 Masking Operations with the AND Instruction

If you'll take a look at the truth table for the and operation back in Chapter One, you'll note that if you fix either operand at zero the result is always zero. If you set that operand to one, the result is always the value of the other operand. We can use this property of the and instruction to selectively force certain bits to zero in a value without affecting other bits. This is called *masking out* bits.

As an example, consider the ASCII codes for the digits "0"..."9". Their codes fall in the range 30h..39h respectively. To convert an ASCII digit to its corresponding numeric value, you must subtract 30h from the ASCII code. This is easily accomplished by logically anding the value with 0Fh. This strips (sets to zero) all but the L.O. four bits producing the numeric value. You could have used the subtract instruction, but most people use the and instruction for this purpose.

9.6.2 Masking Operations with the OR Instruction

Much as you can use the and instruction to force selected bits to zero, you can use the or instruction to force selected bits to one. This operation is called *masking in bits*.

Remember the masking out operation described earlier with the and instruction? In that example we wanted to convert an ASCII code for a digit to its numeric equivalent. You can use the or instruction to reverse this process. That is, convert a numeric value in the range 0..9 to the ASCII code for the corresponding digit, i.e., '0'..'9'. To do this, logically or the specified numeric value with 30h.

9.7 Packing and Unpacking Data Types

One of the primary uses of the shift and rotate instructions is packing and unpacking data. Byte and word data types are chosen more often than any other since the 80x86 supports these two data sizes with hardware. If you don't need exactly eight or 16 bits, using a byte or word to hold your data might be wasteful. By packing data, you may be able to reduce memory requirements for your data by inserting two or more values into a single byte or word. The cost for this reduction in memory use is lower performance. It takes time to pack and unpack the data. Nevertheless, for applications that aren't speed critical (or for those portions of the application that aren't speed critical), the memory savings might justify the use of packed data.

The data type that offers the most savings when using packing techniques is the boolean data type. To represent true or false requires a single bit. Therefore, up to eight different boolean values can be packed into a single byte. This represents an 8:1 compression ratio, therefore, a packed array of boolean values requires only one-eighth the space of an equivalent unpacked array (where each boolean variable consumes one byte). For example, the Pascal array

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

requires only four bytes when packed one value per bit. When packed one value per byte, this array requires 32 bytes.

Dealing with a packed boolean array requires two operations. You'll need to insert a value into a packed variable (often called a packed field) and you'll need to extract a value from a packed field.

To insert a value into a packed boolean array, you must align the source bit with its position in the destination operand and then store that bit into the destination operand. You can do this with a sequence of and, or, and shift instructions. The first step is to mask out the corresponding bit in the destination operand. Use an and instruction for this. Then the source operand is shifted so that it is aligned with the destination position, finally the source operand is or'd into the destination operand. For example, if you want to insert bit zero of the ax register into bit five of the cx register, the following code could be used:

```
and     cl, 0DFh    ;Clear bit five (the destination bit)
and     al, 1       ;Clear all AL bits except the src bit.
ror     al, 1       ;Move to bit 7
shr     al, 1       ;Move to bit 6
shr     al, 1       ;move to bit 5
or      cl, al
```

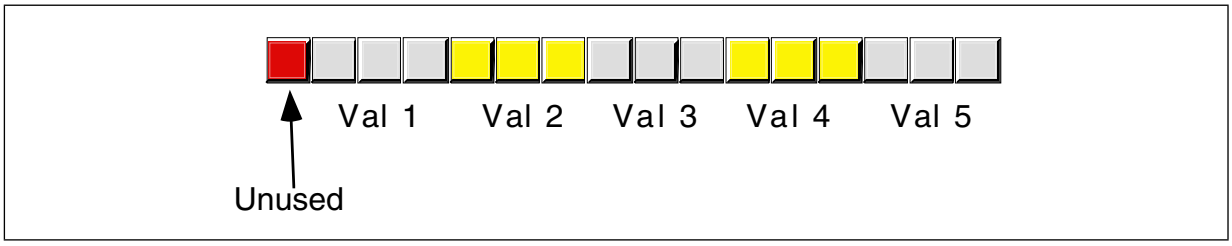


Figure 8.4 Packed Data

This code is somewhat tricky. It rotates the data to the right rather than shifting it to the left since this requires fewer shifts and rotate instructions.

To extract a boolean value, you simply reverse this process. First, you move the desired bit into bit zero and then mask out all the other bits. For example, to extract the data in bit five of the cx register leaving the single boolean value in bit zero of the ax register, you'd use the following code:

```
mov     al, cl
shl     al, 1       ;Bit 5 to bit 6
shl     al, 1       ;Bit 6 to bit 7
rol     al, 1       ;Bit 7 to bit 0
and     ax, 1       ;Clear all bits except 0
```

To test a boolean variable in a packed array you needn't extract the bit and then test it, you can test it in place. For example, to test the value in bit five to see if it is zero or one, the following code could be used:

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

Other types of packed data can be handled in a similar fashion except you need to work with two or more bits. For example, suppose you've packed five different three bit fields into a sixteen bit value as shown in Figure 8.4.

If the ax register contains the data to pack into value3, you could use the following code to insert this data into field three:

```
mov     ah, al        ;Do a shl by 8
shr     ax, 1         ;Reposition down to bits 6..8
shr     ax, 1
and     ax, 11100000b ;Strip undesired bits
and     DATA, 0FE3Fh  ;Set destination field to zero.
or      DATA, ax      ;Merge new data into field.
```

Extraction is handled in a similar fashion. First you strip the unneeded bits and then you justify the result:

```
mov     ax, DATA
and     ax, 1Ch
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
```

This code can be improved by using the following code sequence:

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

Additional uses for packed data will be explored throughout this book.

9.8 Tables

The term "table" has different meanings to different programmers. To most assembly language programmers, a table is nothing more than an array that is initialized with some data. The assembly language programmer often uses tables to compute complex or otherwise slow functions. Many very high level languages (e.g., SNOBOL4 and Icon) directly support a table data type. Tables in these languages are essentially arrays whose elements you can access with an non-integer value (e.g., floating point, string, or any other data type). In this section, we will adopt the assembly language programmer's view of tables.

A Table is an array containing preinitialized values that do not change during the execution of the program. A table can be compared to an array in the same way an integer constant can be compared to an integer variable. In assembly language, you can use tables for a variety of purposes: computing functions, controlling program flow, or simply "looking things up". In general, tables provide a fast mechanism for performing some operation at the expense of some space in your program (the extra space holds the tabular data). In the following sections we'll explore some of the many possible uses of tables in an assembly language program.

9.8.1 Function Computation via Table Look Up

Tables can do all kinds of things in assembly language. In HLLs, like Pascal, it's real easy to create a formula which computes some value. A simple looking arithmetic expression is equivalent to a considerable amount of 80x86 assembly language code. Assembly language programmers tend to compute many values via table look up rather than through the execution of some function. This has the advantage of being easier, and often more efficient as well. Consider the following Pascal statement:

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

This Pascal if statement converts the character variable character from lower case to upper case if character is in the range 'a'..'z'. The 80x86 assembly language code that does the same thing is

```
                mov     al, character
                cmp     al, 'a'
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 05fh     ;Same operation as SUB AL,32
NotLower:       mov     character, al
```

Had you buried this code in a nested loop, you'd be hard pressed to improve the speed of this code without using a table look up. Using a table look up, however, allows you to reduce this sequence of instructions to just four instructions:

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

CnvrtLower is a 256-byte table which contains the values 0..60h at indices 0..60h, 41h..5Ah at indices 61h..7Ah, and 7Bh..0FFh at indices 7Bh..0FFh. Often, using this table look up facility will increase the speed of your code.

As the complexity of the function increases, the performance benefits of the table look up method increase dramatically. While you would almost never use a look up table to convert lower case to upper case, consider what happens if you want to swap cases:

```
Via computation:

                mov     al, character
                cmp     al, 'a'
```

```
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 05fh
                jmp     ConvertDone

NotLower:       cmp     al, 'A'
                jb      ConvertDone
                cmp     al, 'Z'
                ja      ConvertDone
                or      al, 20h
ConvertDone:
                mov     character, al
```

The table look up code to compute this same function is:

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

As you can see, when computing a function via table look up, no matter what the function is, only the table changes, not the code doing the look up.

Table look ups suffer from one major problem – functions computed via table look ups have a limited domain. The domain of a function is the set of possible input values (parameters) it will accept. For example, the upper/lower case conversion functions above have the 256-character ASCII character set as their domain.

A function such as SIN or COS accepts the set of real numbers as possible input values. Clearly the domain for SIN and COS is much larger than for the upper/lower case conversion function. If you are going to do computations via table look up, you must limit the domain of a function to a small set. This is because each element in the domain of a function requires an entry in the look up table. You won't find it very practical to implement a function via table look up whose domain the set of real numbers.

Most look up tables are quite small, usually 10 to 128 entries. Rarely do look up tables grow beyond 1,000 entries. Most programmers don't have the patience to create (and verify the correctness) of a 1,000 entry table.

Another limitation of functions based on look up tables is that the elements in the domain of the function must be fairly contiguous. Table look ups take the input value for a function, use this input value as an index into the table, and return the value at that entry in the table. If you do not pass a function any values other than 0, 100, 1,000, and 10,000 it would seem an ideal candidate for implementation via table look up, its domain consists of only four items. However, the table would actually require 10,001 different elements due to the range of the input values. Therefore, you cannot efficiently create such a function via a table look up. Throughout this section on tables, we'll assume that the domain of the function is a fairly contiguous set of values.

The best functions that can be implemented via table look ups are those whose domain and range is always 0..255 (or some subset of this range). Such functions are efficiently implemented on the 80x86 via the XLAT instruction. The upper/lower case conversion routines presented earlier are good examples of such a function. Any function in this class (those whose domain and range take on the values 0..255) can be computed using the same two instructions (`lea bx,table` / `xlat`) above. The only thing that ever changes is the look up table.

The `xlat` instruction cannot be (conveniently) used to compute a function value once the range or domain of the function takes on values outside 0..255. There are three situations to consider:

- The domain is outside 0..255 but the range is within 0..255,
- The domain is inside 0..255 but the range is outside 0..255, and
- Both the domain and range of the function take on values outside 0..255.

We will consider each of these cases separately.

If the domain of a function is outside 0..255 but the range of the function falls within this set of values, our look up table will require more than 256 entries but we can represent each entry with a single byte. Therefore, the look up table can be an array of bytes. Next to look ups involving the xlat instruction, functions falling into this class are the most efficient. The following Pascal function invocation,

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

where Func is

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

consists of the following 80x86 code:

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

This code loads the function parameter into bx, uses this value (in the range 0..??) as an index into the FuncTable table, fetches the byte at that location, and stores the result into B. Obviously, the table must contain a valid entry for each possible value of X. For example, suppose you wanted to map a cursor position on the video screen in the range 0..1999 (there are 2,000 character positions on an 80x25 video display) to its X or Y coordinate on the screen. You could easily compute the X coordinate via the function X:=Posn mod 80 and the Y coordinate with the formula Y:=Posn div 80 (where Posn is the cursor position on the screen). This can be easily computed using the 80x86 code:

```
mov     bl, 80
mov     ax, Posn
div     bx

; X is now in AH, Y is now in AL
```

However, the div instruction on the 80x86 is very slow. If you need to do this computation for every character you write to the screen, you will seriously degrade the speed of your video display code. The following code, which realizes these two functions via table look up, would improve the performance of your code considerably:

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

If the domain of a function is within 0..255 but the range is outside this set, the look up table will contain 256 or fewer entries but each entry will require two or more bytes. If both the range and domains of the function are outside 0..255, each entry will require two or more bytes and the table will contain more than 256 entries.

Recall from Chapter Four the formula for indexing into a single dimensional array (of which a table is a special case):

```
Address := Base + index * size
```

If elements in the range of the function require two bytes, then the index must be multiplied by two before indexing into the table. Likewise, if each entry requires three, four, or more bytes, the index must be multiplied by the size of each table entry before being used as an index into the table. For example, suppose you have a function, F(x), defined by the following (pseudo) Pascal declaration:

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

You can easily create this function using the following 80x86 code (and, of course, the appropriate table):

```
mov     bx, X       ;Get function input value and
shl     bx, 1       ; convert to a word index into F.
mov     ax, F[bx]
```

The shl instruction multiplies the index by two, providing the proper index into a table whose elements are words.

Any function whose domain is small and mostly contiguous is a good candidate for computation via table look up. In some cases, non-contiguous domains are acceptable as well, as long as the domain can be coerced into an appropriate set of values. Such operations are called conditioning and are the subject of the next section.

9.8.2 Domain Conditioning

Domain conditioning is taking a set of values in the domain of a function and massaging them so that they are more acceptable as inputs to that function. Consider the following function:

$$\sin x = \langle \sin x | x \in [-2\pi, 2\pi] \rangle$$

This says that the (computer) function SIN(x) is equivalent to the (mathematical) function *sin x* where

$$-2\pi \le x \le 2\pi$$

As we all know, sine is a circular function which will accept any real valued input. The formula used to compute sine, however, only accept a small set of these values.

This range limitation doesn't present any real problems, by simply computing SIN(X mod (2*pi)) we can compute the sine of any input value. Modifying an input value so that we can easily compute a function is called conditioning the input. In the example above we computed X mod 2*pi and used the result as the input to the sin function. This truncates X to the domain sin needs without affecting the result. We can apply input conditioning can be applied to table look ups as well. In fact, scaling the index to handle word entries is a form of input conditioning. Consider the following Pascal function:

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

This function computes some value for x in the range 0..4 and it returns zero if x is outside this range. Since x can take on 65,536 different values (being a 16 bit word), creating a table containing 65,536 words where only the first five entries are non-zero seems to be quite wasteful. However, we can still compute this function using a table look up if we use input conditioning. The following assembly language code presents this principle:

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

This code checks to see if x is outside the range 0..4. If so, it manually sets ax to zero, otherwise it looks up the function value through the val table. With input conditioning, you can implement several functions that would otherwise be impractical to do via table look up.

9.8.3 Generating Tables

One big problem with using table look ups is creating the table in the first place. This is particularly true if there are a large number of entries in the table. Figuring out the data to place in the table, then laboriously entering the data, and, finally, checking that data to make sure it is valid, is a very time-staking and boring process. For many tables, there is no way around this process. For other tables there is a better way – use the computer to generate the table for you. An example is probably the best way to describe this. Consider the following modification to the sine function:

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

This states that x is an integer in the range 0..359 and r is an integer. The computer can easily compute this with the following code:

```
mov     bx, X
shl     bx, 1
mov     ax, Sines [bx] ;Get SIN(X)*1000
mov     bx, R          ;Compute R*(SIN(X)*1000)
mul     bx
mov     bx, 1000       ;Compute (R*(SIN(X)*1000))/1000
div     bx
```

Note that integer multiplication and division are not associative. You cannot remove the multiplication by 1000 and the division by 1000 because they seem to cancel one another out. Furthermore, this code must compute this function in exactly this order. All that we need to complete this function is a table containing 360 different values corresponding to the sine of the angle (in degrees) times 1,000. Entering a table into an assembly language program containing such values is extremely boring and you'd probably make several mistakes entering and verifying this data. However, you can have the program generate this table for you. Consider the following Turbo Pascal program:

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

This program produces the following output:

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

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

Obviously it's much easier to write the Turbo Pascal program that generated this data than to enter (and verify) this data by hand. This little example shows how useful Pascal can be to the assembly language programmer!

9.9 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.

9.9.1 Converting Arithmetic Expressions to Assembly Language

The following sample program (Pgm9_1.asm on the companion CD-ROM) provides some examples of converting arithmetic expressions into assembly language:

```
; Pgm9_1.ASM
;
; Several examples demonstrating how to convert various
; arithmetic expressions into assembly language.

                .xlist
                include     stdlib.a
                includelib  stdlib.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
```

9.9.2 Boolean Operations Example

The following sample program (Pgm9_2.asm on the companion CD-ROM) demonstrates how to manipulate boolean values in assembly language. It also provides an example of Demorgan's Theorems in operation.

```
; Pgm9_2.ASM
;
; This program demonstrates DeMorgan's theorems and
; various other logical computations.

                .xlist
                include     stdlib.a
                includelib  stdlib.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
```

9.9.3 64-bit Integer I/O

This sample program (Pgm9_3.asm on the companion CD-ROM) shows how to read and write 64-bit integers. It provides the ATOU64 and PUTU64 routines that let you convert a string of digits to a 64-bit unsigned integer and output a 64-bit unsigned integer as a decimal string to the display.

```
; Pgm9_3.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
                includelib  stdlib.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 terminator.

; 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
```

9.9.4 Packing and Unpacking Date Data Types

This sample program demonstrates how to pack and unpack data using the Date data type introduced in Chapter One.

```
; Pgm9_4.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
                includelib  stdlib.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
```

9.10 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.

9.10.1 Debugging Programs with CodeView

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.

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.

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 "G" (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

G *breakpoint_address*

The F7 keystroke 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.

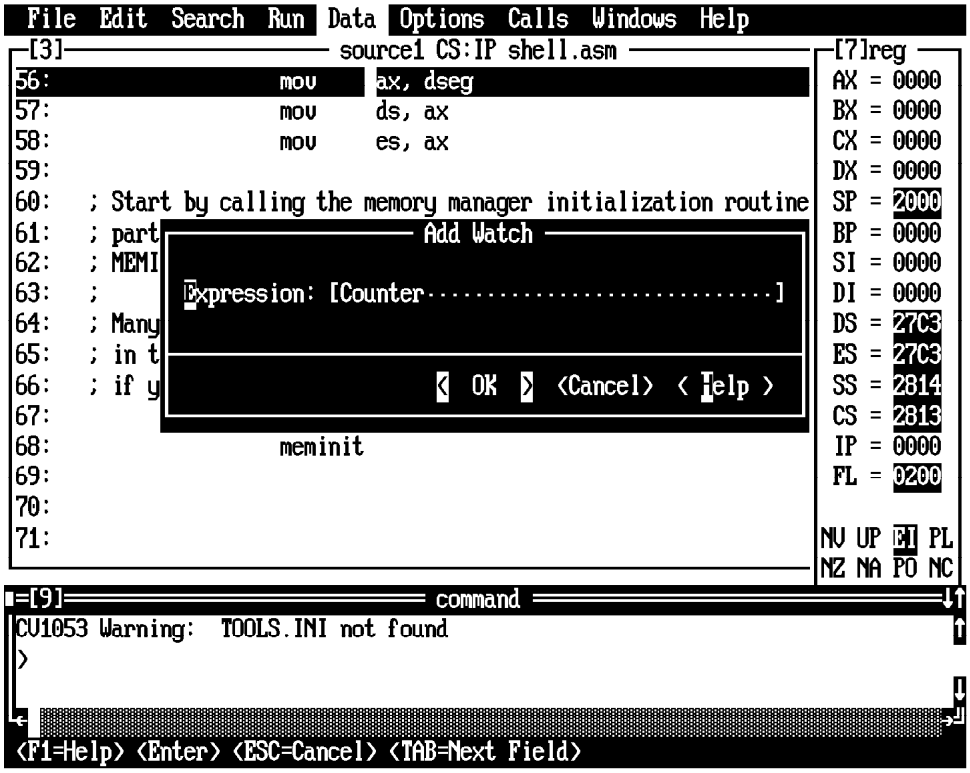
One very important thing to keep in mind, especially when using the F7 command to set non-sticky breakpoints, is that you must execute the statement on which the breakpoint was set for the breakpoint to have any effect. If your program skips over the instruction on which you've set the breakpoint, you 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. This opens up a dialog box that looks like the following:



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.

9.10.2 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.

9.10.2.1 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.

9.10.2.2 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.

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.

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.

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.

9.10.2.3 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.

9.10.3 Debug Exercise I: Using CodeView to Find Bugs in a Calculation

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 Ex9_1.asm on the companion CD-ROM):

```
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:

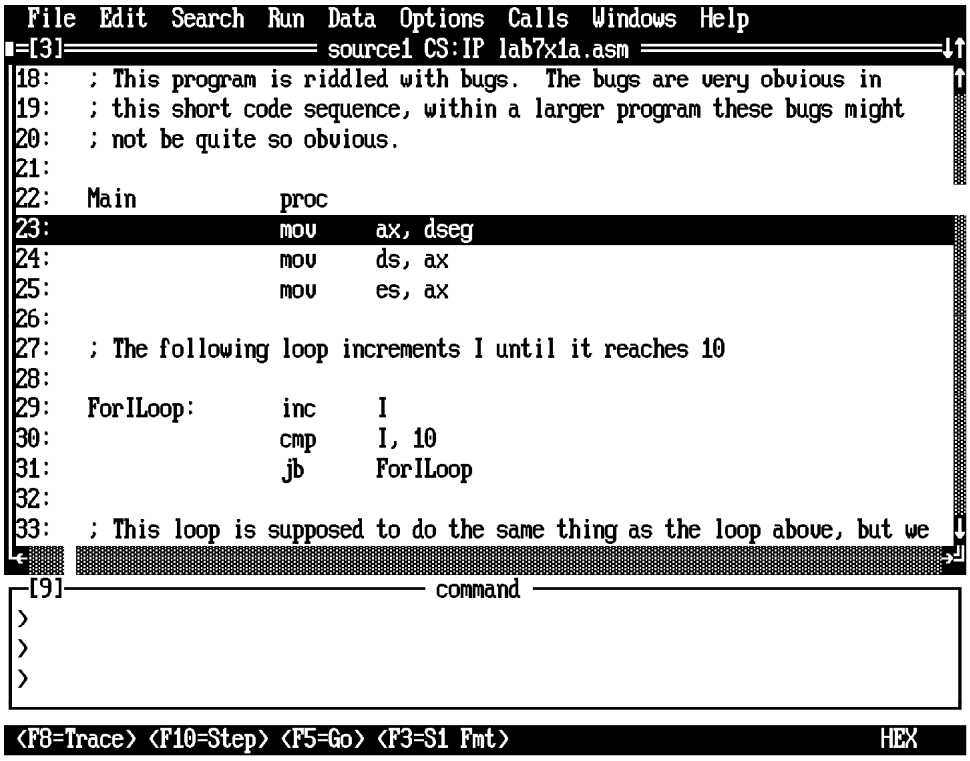
```
ML /Zi Ex9_1.asm
```

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 Ex9_1
```

Your display should now look something like the following:



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.

Part Two: 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 ForlLoop2. Trace through this loop until it falls through to the statement at label ForlLoop3.

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

Part 3: Locating another bug. The third loop contains a typo that causes it to restart at label ForlLoop. 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.

Part 4: Verifying correctness. Program Ex9_2.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.

Part 5: Using Ex9_2.asm, 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.

9.10.4 Software Delay Loop Exercises

Software Delay Loops. The Ex9_3.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.

Part 2: Hardware determined software delay loop. The Ex9_4.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.

9.11 Programming Projects

9.12 Summary

This chapter discussed arithmetic and logical operations on 80x86 CPUs. It presented the instructions and techniques necessary to perform integer arithmetic in a fashion similar to high level languages. This chapter also discussed multiprecision operations, how to perform arithmetic operations using non-arithmetic instructions, and how to use arithmetic instructions to perform non-arithmetic operations.

Arithmetic expressions are much simpler in a high-level language than in assembly language. Indeed, the original purpose of the FORTRAN programming language was to provide a FORMula TRANslator for arithmetic expressions. Although it takes a little more effort to convert an arithmetic formula to assembly language than it does to, say, Pascal, as long as you follow some very simple rules the conversion is not hard. For a step-by-step description, see

- "Arithmetic Expressions" on page 460
- "Simple Assignments" on page 460
- "Simple Expressions" on page 460
- "Complex Expressions" on page 462
- "Commutative Operators" on page 466
- "Logical (Boolean) Expressions" on page 467

One big advantage to assembly language is that it is easy to perform nearly unlimited precision arithmetic and logical operations. This chapter describes how to do extended precision operations for most of the common operations. For complete instructions, see

- "Multiprecision Operations" on page 470
- "Multiprecision Addition Operations" on page 470
- "Multiprecision Subtraction Operations" on page 472
- "Extended Precision Comparisons" on page 473
- "Extended Precision Multiplication" on page 475
- "Extended Precision Division" on page 477
- "Extended Precision NEG Operations" on page 480
- "Extended Precision AND Operations" on page 481
- "Extended Precision OR Operations" on page 482
- "Extended Precision NOT Operations" on page 482
- "Extended Precision Shift Operations" on page 482
- "Extended Precision Rotate Operations" on page 484

At certain times you may need to operate on two operands that are different types. For example, you may need to add a byte value to a word value. The general idea is to extend the smaller operand so that it is the same size as the larger operand and then compute the result on these like-sized operands. For all the details, see

- "Operating on Different Sized Operands" on page 485

Although the 80x86 instruction set provides straight-forward ways to accomplish many tasks, you can often take advantage of various idioms in the instruction set or with respect to certain arithmetic operations to produce code that is faster or shorter than the obvious way. This chapter introduces a few of these idioms. To see some examples, check out

- "Machine and Arithmetic Idioms" on page 486
- "Multiplying Without MUL and IMUL" on page 487
- "Division Without DIV and IDIV" on page 488
- "Using AND to Compute Remainders" on page 488
- "Implementing Modulo-n Counters with AND" on page 489
- "Testing an Extended Precision Value for 0FFFF..FFh" on page 489

To manipulate packed data you need the ability to extract a field from a packed record and insert a field into a packed record. You can use the logical and and or instructions to mask the fields you want to manipulate; you can use the shl and shr instructions to position the data to their appropriate positions before inserting or after extracting data. To learn how to pack and unpack data, see

9.13 Questions

1) Describe how you might go about adding an unsigned word to an unsigned byte variable producing a byte result. Explain any error conditions and how to check for them.

2) Answer question one for signed values.

3) Assume that var1 is a word and var2 and var3 are double words. What is the 80x86 assembly language code that will add var1 to var2 leaving the sum in var3 if:

   a) var1, var2, and var3 are unsigned values.

   b) var1, var2, and var3 are signed values.

4) "ADD BX, 4" is more efficient than "LEA BX, 4[BX]". Give an example of an LEA instruction which is more efficient than the corresponding ADD instruction.

5) Provide the single 80386 LEA instruction that will multiply EAX by five.

6) Assume that VAR1 and VAR2 are 32 bit variables declared with the DWORD pseudo-opcode. Write code sequences that will test the following:

   a) VAR1 = VAR2

   b) VAR1 <> VAR2

   c) VAR1 < VAR2 (Unsigned and signed versions

   d) VAR1 <= VAR2 for each of these)

   e) VAR1 > VAR2

   f) VAR1 >= VAR2

7) Convert the following expressions into assembly language code employing shifts, additions, and subtractions in place of the multiplication:

   a) AX*15

   b) AX*129

   c) AX*1024

   d) AX*20000

8) What's the best way to divide the AX register by the following constants?

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

9) Describe how you could multiply an eight bit value in AL by 256 (leaving the result in AX) using nothing more than two MOV instructions.

10) How could you logically AND the value in AX by 0FFh using nothing more than a MOV instruction?

11) Suppose that the AX register contains a pair of packed binary values with the L.O. four bits containing a value in the range 0..15 and the H.O. 12 bits containing a value in the range 0..4095. Now suppose you want to see if the 12 bit portion contains the value 295. Explain how you could accomplish this with two instructions.

12) How could you use the TEST instruction (or a sequence of TEST instructions) to see if bits zero and four in the AL register are both set to one? How would the TEST instruction be used to see if either bit is set? How could the TEST instruction be used to see if neither bit is set?

13) Why can't the CL register be used as a count operand when shifting multi-precision operands. I.e., why won't the following instructions shift the value in (DX,AX) three bits to the left?

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

14) Provide instruction sequences that perform an extended precision (32 bit) ROL and ROR operation using only 8086 instructions.

15) Provide an instruction sequence that implements a 64 bit ROR operation using the 80386 SHRD and BT instructions.

16) Provide the 80386 code to perform the following 64 bit computations. Assume you are computing X := Y op Z with X, Y, and Z defined as follows:

```
X               dword    0, 0
y               dword    1, 2
z               dword    3, 4
```

a) addition b) subtraction c) multiplication

c) Logical AND d) Logical OR e) Logical XOR

f) negate g) Logical NOT