Advanced Arithmetic Chapter Four

4.1 Chapter Overview

This chapter deals with those arithmetic operations for which assembly language is especially well suited and high level languages are, in general, poorly suited. It covers three main topics: extended precision arithmetic, arithmetic on operands who sizes are different, and decimal arithmetic.

By far, the most extensive subject this chapter covers is multi-precision arithmetic. By the conclusion of this chapter you will know how to apply arithmetic and logical operations to integer operands of any size. If you need to work with integer values outside the range ±2 billion (or with unsigned values beyond four billion), no sweat; this chapter will show you how to get the job done.

Operands whose sizes are not the same also present some special problems in arithmetic operations. For example, you may want to add a 128-bit unsigned integer to a 256-bit signed integer value. This chapter discusses how to convert these two operands to a compatible format so the operation may proceed.

Finally, this chapter discusses decimal arithmetic using the BCD (binary coded decimal) features of the 80x86 instruction set and the FPU. This lets you use decimal arithmetic in those few applications that absolutely require base 10 operations (rather than binary).

4.2 Multiprecision Operations

One big advantage of assembly language over high level languages is that assembly language does not limit the size of integer operations. For example, the C programming language defines a maximum of three different integer sizes: short int, int, and long int[1]. On the PC, these are often 16 and 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 to add 256 bit integer values together, no problem, it's relatively easy to accomplish this in assembly language. The following sections describe how extended various arithmetic and logical operations from 16 or 32 bits to as many bits as you please.

4.2.1 Multiprecision Addition Operations

The 80x86 ADD instruction adds two eight, sixteen, or thirty-two bit numbers[2]. 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 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.
```

1. Newer C standards also provide for a "long long int" which is usually a 64-bit integer.
2. As usual, 32 bit arithmetic is available only on the 80386 and later processors.

```
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 together a byte, word, or dword at a time. Consider the three double word (96 bit) addition operation in Figure 4.1.

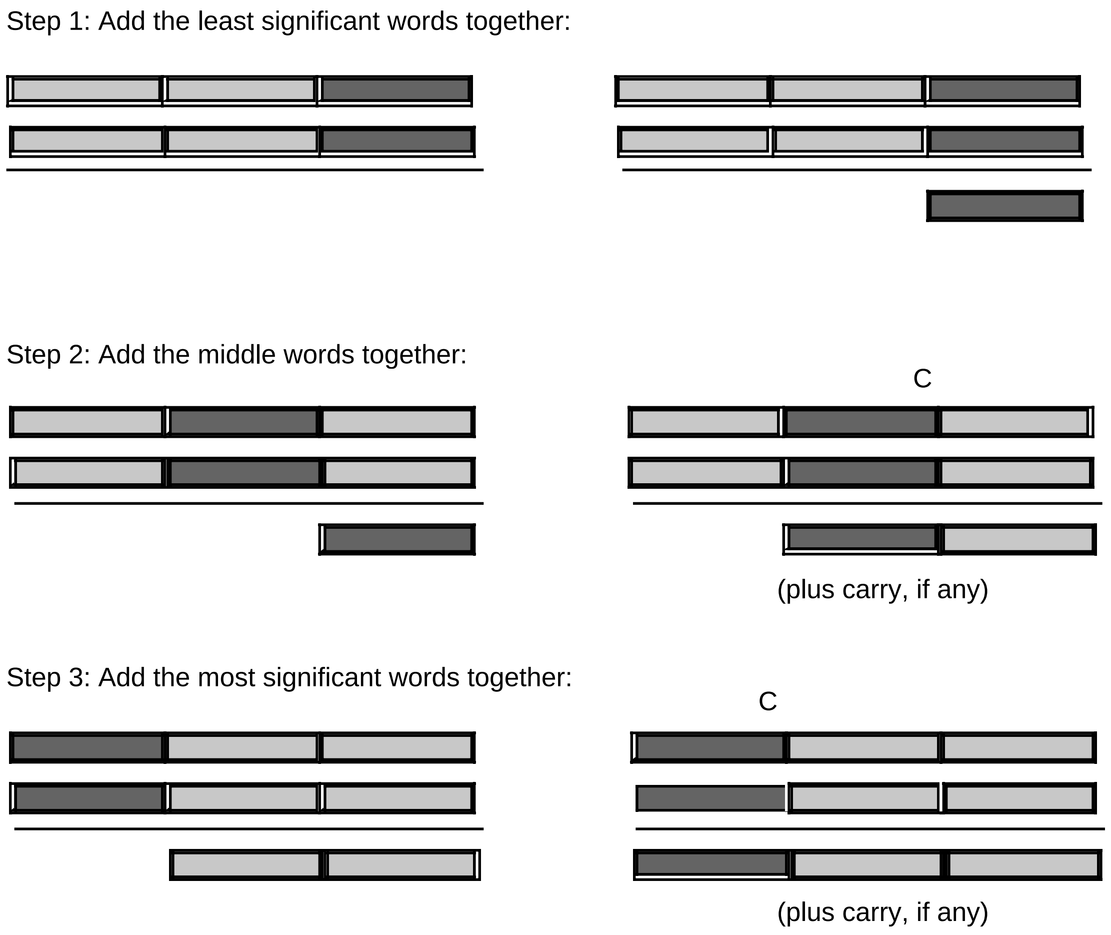

Step 1: Add the least significant words together:
Step 2: Add the middle words together:
C
(plus carry, if any)
Step 3: Add the most significant words together:
C
(plus carry, if any)

Figure 4.1 Adding Two 96-bit Objects Together

As you can see from this figure, the idea is to break up a larger operation into a sequence of smaller operations. Since the x86 processor family is capable of adding together, at most, 32 bits at a time, the operation must proceed in blocks of 32-bits or less. So the first step is to add the two L.O. double words together much as we would add the two L.O. digits of a decimal number together in the manual algorithm. There is nothing special about this operation, you can use the ADD instruction to achieve this.

The second step involves adding together the second pair of double words in the two 96-bit values. Note that in step two, the calculation must also add in the carry out of the previous addition (if any). If there was a carry out of the L.O. addition, the ADD instruction sets the carry flag to one; conversely, if there was no carry out of the L.O. addition, the earlier ADD instruction clears the carry flag. Therefore, in this second addition, we really need to compute the sum of the two double words plus the carry out of the first instruction. Fortunately, the x86 CPUs provide an instruction that does exactly this: the ADC (add with carry) instruction. The ADC instruction uses the same syntax as the ADD instruction and performs almost the same operation:

```
adc( source, dest );  // dest := dest + source + C
```

As you can see, the only difference between the ADD and ADC instruction is that the ADC instruction adds in the value of the carry flag along with the source and destination operands. It also sets the flags the same way the ADD instruction does (including setting the carry flag if there is an unsigned overflow). This is exactly what we need to add together the middle two double words of our 96-bit sum.

In step three of Figure 4.1, the algorithm adds together the H.O. double words of the 96-bit value. Once again, this addition operation also requires the addition of the carry out of the sum of the middle two double words; hence the ADC instruction is needed here, as well. To sum it up, the ADD instruction adds the L.O. double words together. The ADC (add with carry) instruction adds all other double word pairs together. At the end of the extended precision addition sequence, the carry flag indicates unsigned overflow (if set), a set overflow flag indicates signed overflow, and the sign flag indicates the sign of the result. The zero flag doesn’t have any real meaning at the end of the extended precision addition (it simply means that the sum of the H.O. two double words is zero, this does not indicate that the whole result is zero).

For example, suppose that you have two 64-bit values you wish to add together, defined as follows:

```
static
    X: qword;
    Y: qword;
```

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

```
    mov( (type dword X), eax );         // Add together the L.O. 32 bits
    add( (type dword Y), eax );         // of the numbers and store the
    mov( eax, (type dword Z) );         // result into the L.O. dword of Z.

    mov( (type dword X[4]), eax );      // Add together (with carry) the
    adc( (type dword Y[4]), eax );      // H.O. 32 bits and store the result
    mov( eax, (type dword Z[4]) );      // into the H.O. dword of Z.
```

Remember, these variables are qword objects. Therefore the compiler will not accept an instruction of the form "mov(X, eax);" because this instruction would attempt to load a 64 bit value into a 32 bit register. This code uses the coercion operator to coerce symbols *X*, *Y*, and *Z* to 32 bits. The first three instructions add the L.O. double words of *X* and *Y* together and store the result at the L.O. double word of *Z*. The last three instructions add the H.O. double words of *X* and *Y* together, along with the carry out of the L.O. word, and store the result in the H.O. double word of *Z*. Remember, address expressions of the form “X[4]” access the H.O. double word of a 64 bit entity. This is due to the fact that the x86 address space addresses bytes and it takes four consecutive bytes to form a double word.

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:

```
type
    tBig: dword[4];  // Storage for four dwords is 128 bits.

static
```

```
    BigVal1: tBig;
    BigVal2: tBig;
    BigVal3: tBig;
     .
     .
     .
    mov( BigVal1[0], eax );   // Note there is no need for (type dword BigValx)
    add( BigVal2[0], eax );   // because the base type of BitValx is dword.
    mov( eax, BigVal3[0] );

    mov( BigVal1[4], eax );
    adc( BigVal2[4], eax );
    mov( eax, BigVal3[4] );

    mov( BigVal1[8], eax );
    adc( BigVal2[8], eax );
    mov( eax, BigVal3[8] );

    mov( BigVal1[12], eax );
    adc( BigVal2[12], eax );
    mov( eax, BigVal3[12] );
```

4.2.2 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 (subtract with borrow) instruction on the high order values. The following example demonstrates a 64 bit subtraction using the 32 bit registers on the x86:

```
static
    Left:  qword;
    Right: qword;
    Diff:  qword;
         .
         .
         .
    mov( (type dword Left), eax );
    sub( (type dword Right), eax );
    mov( eax, (type dword Diff) );

    mov( (type dword Left[4]), eax );
    sbb( (type dword Right[4]), eax );
    mov( (type dword Diff[4]), eax );
```

The following example demonstrates a 128-bit subtraction:

```
type
    tBig: dword[4];  // Storage for four dwords is 128 bits.

static
    BigVal1: tBig;
    BigVal2: tBig;
    BigVal3: tBig;
     .
     .
     .

    // Compute BigVal3 := BigVal1 - BigVal2
    mov( BigVal1[0], eax );   // Note there is no need for (type dword BigValx)
    sub( BigVal2[0], eax );   // because the base type of BitValx is dword.
    mov( eax, BigVal3[0] );

    mov( BigVal1[4], eax );
    sbb( BigVal2[4], eax );
    mov( eax, BigVal3[4] );

    mov( BigVal1[8], eax );
    sbb( BigVal2[8], eax );
    mov( eax, BigVal3[8] );

    mov( BigVal1[12], eax );
    sbb( BigVal2[12], eax );
    mov( eax, BigVal3[12] );
```

4.2.3 Extended Precision Comparisons

Unfortunately, there isn't a "compare with borrow" instruction that you can use 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 $2157 and $1293. The L.O. bytes of these two values do not affect the outcome of the comparison. Simply comparing $21 with $12 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 unsigned 64 bit integers:

```
// 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( (type dword QWordValue[4]), eax );  // Get H.O. dword
        cmp( eax, (type dword QWordValue2[4]));
        jg IsGreater;
        jl IsLess;

        mov( (type dword QWordValue[0]), eax );  // If H.O. dwords were equal,
        cmp( eax, (type dword QWordValue2[0]));  // then we must compare the
        ja IsGreater;                            // L.O. dwords.
        jb IsLess;

// Fall through to this point if the two values were equal.
```

To compare signed values, simply use the JG and JL instructions in place of JA and JB for the H.O. words (only). You must continue to use unsigned comparisons for all but the H.O. double words you're comparing.

You can easily synthesize any possible comparison from the sequence above, the following examples show how to do this. These examples demonstrate signed comparisons, substitute JA, JAE, JB, and JBE for JG, JGE, JL, and JLE (respectively) for the H.O. comparisons if you want unsigned comparisons.

```
static
```

```
    QW1: qword;
    QW2: qword;

const
    QW1d: text := "(type dword QW1)";
    QW2d: text := "(type dword QW2)";

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

    mov( QW1d[4], eax );   // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg NotLess;            // Substitute ja here for unsigned comparison.
    jl IsLess;             // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax );   // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jb IsLess;
NotLess:

// 64 bit test to see if QW1 <= QW2 (signed).  Jumps to "IsLessEq" if the
// condition is true.

    mov( QW1d[4], eax );   // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg NotLessEQ;          // Substitute ja here for unsigned comparison.
    jl IsLessEQ;           // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax );   // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jbe IsLessEQ;
NotLessEQ:


// 64 bit test to see if QW1 > QW2 (signed).  Jumps to "IsGtr" if this condition
// is true.

    mov( QW1d[4], eax );   // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg IsGtr;              // Substitute ja here for unsigned comparison.
    jl NotGtr;             // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax );   // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    ja IsGtr;
NotGtr:


// 64 bit test to see if QW1 >= QW2 (signed).  Jumps to "IsGtrEQ" if this
// is the case.

    mov( QW1d[4], eax );   // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg IsGtrEQ;            // Substitute ja here for unsigned comparison.
    jl NotGtrEQ;           // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax );   // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jae 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( QW1d[4], eax );   // Get H.O. dword
    cmp( eax, QW2d[4] );
    jne NotEqual;

    mov( QW1d[0], eax );   // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    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( QW1d[4], eax );   // Get H.O. dword
    cmp( eax, QW2d[4] );
    jne NotEqual;

    mov( QW1d[0], eax );   // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jne NotEqual;

// Fall through to this point if they are equal.
```

You cannot directly use the HLA high level control structures if you need to perform an extended precision comparison. However, you may use the HLA hybrid control structures and bury the appropriate comparison into this statements. Doing so will probably make your code easier to read. For example, the following *if..then..else..endif* statement checks to see if *QW1 > QW2* using a 64-bit extended precision signed comparison:

```
if
( #{
    mov( QW1d[4], eax );
    cmp( eax, QW2d[4] );
    jg true;

    mov( QW1d[0], eax );
    cmp( eax, QW2d[0] );
    jna false;
}# ) then

    << code to execute if QW1 > QW2 >>

else

    << code to execute if QW1 <= QW2 >>

endif;
```

If you need to compare objects that are larger than 64 bits, it is very easy to generalize the code above. Always start the comparison with the H.O. double words of the objects and work you way down towards the L.O. double words of the objects as long as the corresponding double words are equal The following example compares two 128-bit values to see if the first is less than or equal (unsigned) to the second:

```
type
```

```
        t128: dword[4];

    static
        Big1: t128;
        Big2: t128;
         .
         .
         .
        if
        ( #{
            mov( Big1[12], eax );
            cmp( eax, Big2[12] );
            jb true;
            mov( Big1[8], eax );
            cmp( eax, Big2[8] );
            jb true;
            mov( Big1[4], eax );
            cmp( eax, Big2[4] );
            jb true;
            mov( Big1[0], eax );
            cmp( eax, Big2[0] );
            jnbe false;
        }# ) then

            << Code to execute if Big1 <= Big2 >>

        else

            << Code to execute if Big1 > Big2 >>

        endif;
```

4.2.4 Extended Precision Multiplication

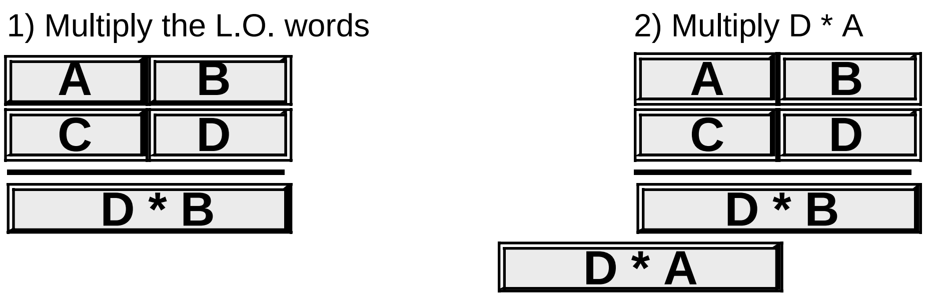
Although an 8x8, 16x16, or 32x32 multiply is usually sufficient, there are times when you may want to multiply larger values together. You will use the x86 single operand MUL and IMUL instructions for extended precision multiplication.

Not surprisingly (in view of how we achieved extended precision addition using ADC and SBB), you use the same techniques to perform extended precision multiplication on the x86 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           2) Multiply 5*2:
   digits together (5*3):

    123                                 123
     45                                  45
    ---                                  ---
     15                                  15
                                        10
```

```
3) Multiply 5*1:                       4) Multiply 4*3:

   123                                     123
    45                                      45
   ---                                      ---
    15                                      15
   10                                      10
   5                                       5
                                           12


5) Multiply 4*2:                       6) Multiply 4*1:

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

7) Add all the partial products together:

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

The 80x86 does extended precision multiplication in the same manner except that it works with bytes, words, and double words rather than digits. Figure 4.2 shows how this works



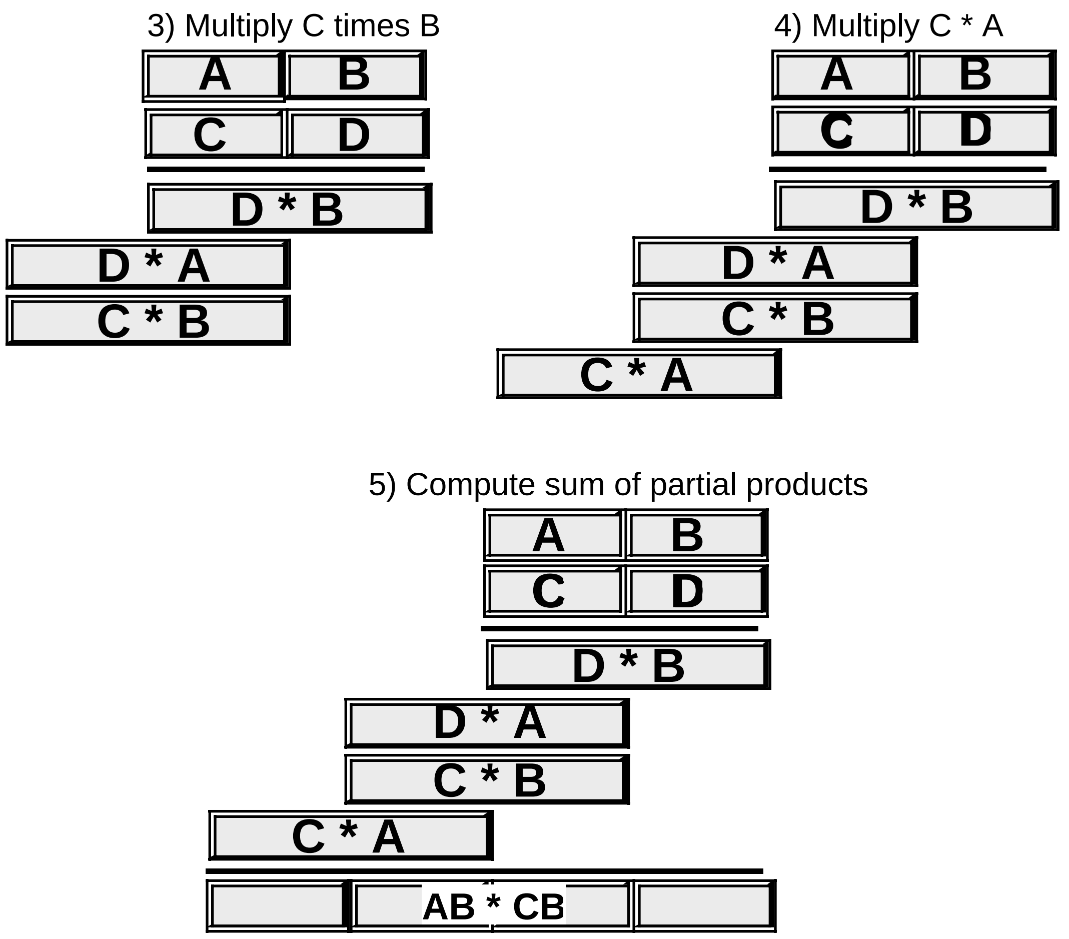


Figure 4.2 Extended Precision Multiplication

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 64 bit values on a 32 bit processor:

Note: *Multiplier* and *Multiplicand* are 64 bit variables declared in the data segment via the qword type. *Product* is a 128 bit variable declared in the data segment via the qword[2] type.

```
program testMUL64;
#include( "stdlib.hhf" )

type
    t128:dword[4];

procedure MUL64( Multiplier:qword; Multiplicand:qword; var Product:t128 );
const
    mp: text := "(type dword Multiplier)";
    mc: text := "(type dword Multiplicand)";
    prd:text := "(type dword [edi])";
```

```
begin MUL64;

    mov( Product, edi );

    // Multiply the L.O. dword of Multiplier times Multiplicand.

    mov( mp, eax );
    mul( mc, eax );     // Multiply L.O. dwords.
    mov( eax, prd );    // Save L.O. dword of product.
    mov( edx, ecx );    // Save H.O. dword of partial product result.

    mov( mp, eax );
    mul( mc[4], eax );  // Multiply mp(L.O.) * mc(H.O.)
    add( ecx, eax );    // Add to the partial product.
    adc( 0, edx );      // Don't forget the carry!
    mov( eax, ebx );    // Save partial product for now.
    mov( edx, ecx );

    // Multiply the H.O. word of Multiplier with Multiplicand.

    mov( mp[4], eax );  // Get H.O. dword of Multiplier.
    mul( mc, eax );     // Multiply by L.O. word of Multiplicand.
    add( ebx, eax );    // Add to the partial product.
    mov( eax, prd[4] ); // Save the partial product.
    adc( edx, ecx );    // Add in the carry!
    pushfd();           // Save carry out here.

    mov( mp[4], eax );  // Multiply the two H.O. dwords together.
    mul( mc[4], eax );
    popfd();            // Retrieve carry from above
    adc( ecx, eax );    // Add in partial product from above.
    adc( 0, edx );      // Don't forget the carry!
    mov( eax, prd[8] ); // Save the partial product.
    mov( edx, prd[12] );

end MUL64;

static
    op1: qword;
    op2: qword;
    rslt: t128;


begin testMUL64;

    // Initialize the qword values (note that static objects
    // are initialized with zero bits).

    mov( 1234, (type dword op1 ));
    mov( 5678, (type dword op2 ));
    MUL64( op1, op2, rslt );

    // The following only prints the L.O. qword, but
    // we know the H.O. qword is zero so this is okay.

    stdout.put( "rslt=" );
    stdout.putu64( (type qword rslt));

end testMUL64;
```

Program 4.1 Extended Precision Multiplication

One thing you must keep in mind concerning this code, it only works for unsigned operands. To multiply two signed values you must note the signs of the operands before the multiplication, take the absolute value of the two operands, do an unsigned multiplication, and then adjust the sign of the resulting product based on the signs of the original operands. Multiplication of signed operands appears in the exercises.

This example was fairly straight-forward since it was possible to keep the partial products in various registers. If you need to multiply larger values together, you will need to maintain the partial products in temporary (memory) variables. Other than that, the algorithm that Program 4.1 uses generalizes to any number of double words.

4.2.5 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 and is extremely messy. However, a less general operation, dividing an n-bit quantity by a 32 bit quantity is easily synthesized using the DIV instruction. This section presents both methods for extended precision division.

Before describing how to perform a multi-precision division operation, you should note that some operations require an extended precision division even though they may look calculable with a single DIV or IDIV instruction. Dividing a 64-bit quantity by a 32-bit quantity is easy, as long as the resulting quotient fits into 32 bits. The DIV and IDIV instructions will handle this directly. However, if the quotient does not fit into 32 bits then you have to handle this problem as an extended precision division. The trick here is to divide the (zero or sign extended) H.O dword of the dividend by the divisor, and then repeat the process with the remainder and the L.O. dword of the dividend. The following sequence demonstrates this:

```
static
    dividend: dword[2] := [$1234, 4];  // = $4_0000_1234.
    divisor:  dword := 2;              // dividend/divisor = $2_0000_091A
    quotient: dword[2];
    remainder:dword;
      .
      .
      .
    mov( divisor, ebx );
    mov( dividend[4], eax );
    xor( edx, edx );            // Zero extend for unsigned division.
    div( ebx, edx:eax );
    mov( eax, quotient[4] );    // Save H.O. dword of the quotient (2).
    mov( dividend[0], eax );    // Note that this code does *NOT* zero extend
    div( ebx, edx:eax );        //  EAX into EDX before this DIV instr.
    mov( eax, quotient[0] );    // Save L.O. dword of the quotient ($91a).
    mov( edx, remainder );      // Save away the remainder.
```

Since it is perfectly legal to divide a value by one, it is certainly possible that the resulting quotient after a division could require as many bits as the dividend. That is why the *quotient* variable in this example is the same size (64 bits) as the *dividend* variable. Regardless of the size of the dividend and divisor operands, the remainder is always no larger than the size of the division operation (32 bits in this case). Hence the *remainder* variable in this example is just a double word.

Before analyzing this code to see how it works, let's take a brief look at why a single 64/32 division will *not* work for this particular example even though the DIV instruction does indeed calculate the result for a 64/32 division. The naive approach, assuming that the x86 were capable of this operation, would look something like the following:

```
    // This code does *NOT* work!
```

```
mov( dividend[0], eax );    // Get dividend into edx:eax
mov( divident[4], edx );
div( divisor, edx:eax );    // Divide edx:eax by divisor.
```

Although this code is syntactically correct and will compile, if you attempt to run this code it will raise an *ex.DivideError* exception. The reason, if you'll remember how the DIV instruction works, is that the quotient must fit into 32 bits; since the quotient turns out to be $2_0000_091A, it will not fit into the EAX register, hence the resulting exception.

Now let's take another look at the former code that correctly computes the 64/32 quotient. This code begins by computing the 32/32 quotient of *dividend[4]/divisor.* The quotient from this division (2) becomes the H.O. double word of the final quotient. The remainder from this division (0) becomes the extension in EDX for the second half of the division operation. The second half divides *edx:dividend[0]* by *divisor* to produce the L.O. double word of the quotient and the remainder from the division. Note that the code does not zero extend EAX into EDX prior to the second DIV instruction. EDX already contains valid bits and this code must not disturb them.

The 64/32 division operation above is actually just a special case of the more general division operation that lets you divide an arbitrary sized value by a 32-bit divisor. To achieve this, you begin by moving the H.O. double word of the dividend into EAX and zero extending this into EDX. Next, you divide this value by the divisor. Then, without modifying EDX along the way, you store away the partial quotients, load EAX with the next lower double word in the dividend, and divide it by the divisor. You repeat this operation until you've processed all the double words in the dividend. At that time the EDX register will contain the remainder. The following program demonstrates how to divide a 128 bit quantity by a 32 bit divisor, producing a 128 bit quotient and a 32 bit remainder:

```
program testDiv128;
#include( "stdlib.hhf" )

type
    t128:dword[4];

procedure div128
(
        Dividend:   t128;
        Divisor:    dword;
    var QuotAdrs:   t128;
    var Remainder:  dword
);  @nodisplay;

const
    Quotient: text := "(type dword [edi])";

begin div128;

    push( eax );
    push( edx );
    push( edi );

    mov( QuotAdrs, edi );       // Pointer to quotient storage.

    mov( Dividend[12], eax );   // Begin division with the H.O. dword.
    xor( edx, edx );            // Zero extend into EDX.
    div( Divisor, edx:eax );    // Divide H.O. dword.
    mov( eax, Quotient[12] );   // Store away H.O. dword of quotient.

    mov( Dividend[8], eax );    // Get dword #2 from the dividend
    div( Divisor, edx:eax );    // Continue the division.
```

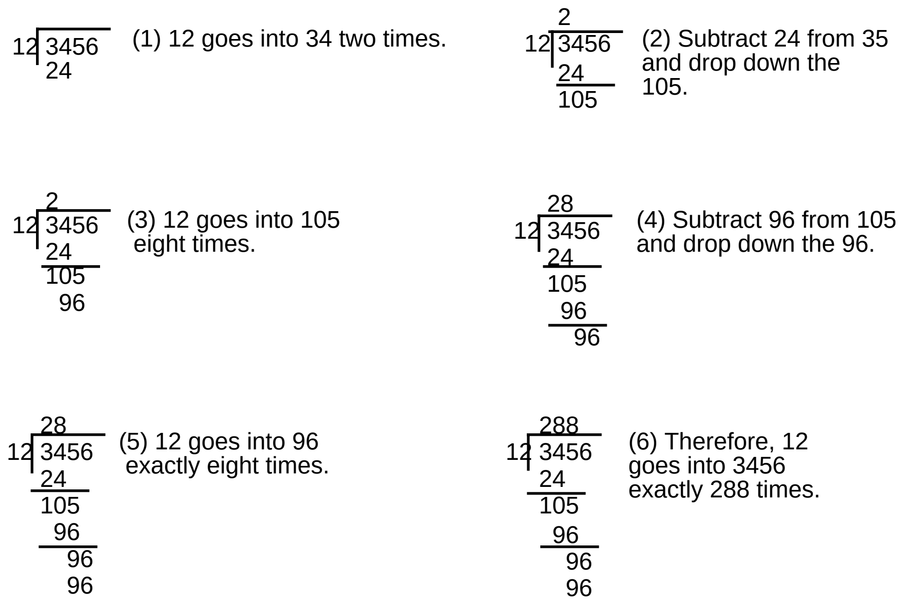
```
        mov( eax, Quotient[8] );    // Store away dword #2 of the quotient.

        mov( Dividend[4], eax );    // Get dword #1 from the dividend.
        div( Divisor, edx:eax );    // Continue the division.
        mov( eax, Quotient[4] );    // Store away dword #1 of the quotient.

        mov( Dividend[0], eax );    // Get the L.O. dword of the dividend.
        div( Divisor, edx:eax );    // Finish the division.
        mov( eax, Quotient[0] );    // Store away the L.O. dword of the quotient.

        mov( Remainder, edi );      // Get the pointer to the remainder's value.
        mov( edx, [edi] );          // Store away the remainder value.

        pop( edi );
        pop( edx );
        pop( eax );

    end div128;

    static
        op1:    t128    := [$2222_2221, $4444_4444, $6666_6666, $8888_8888];
        op2:    dword   := 2;
        quo:    t128;
        rmndr:  dword;


    begin testDiv128;

        div128( op1, op2, quo, rmndr );

        stdout.put
        (
            nl
            nl
            "After the division: " nl
            nl
            "Quotient = $",
            quo[12], "_",
            quo[8], "_",
            quo[4], "_",
            quo[0], nl

            "Remainder = ", (type uns32 rmndr )
        );

    end testDiv128;
```

Program 4.2 Unsigned 128/32 Bit Extended Precision Division

You can extend this code to any number of bits by simply adding additional MOV / DIV / MOV instructions to the sequence. Like the extended multiplication the previous section presents, this extended precision division algorithm works only for unsigned operands. If you need to divide two signed quantities, you must note their signs, take their absolute values, do the unsigned division, and then set the sign of the result based on the signs of the operands.

If you need to use a divisor larger than 32 bits 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).

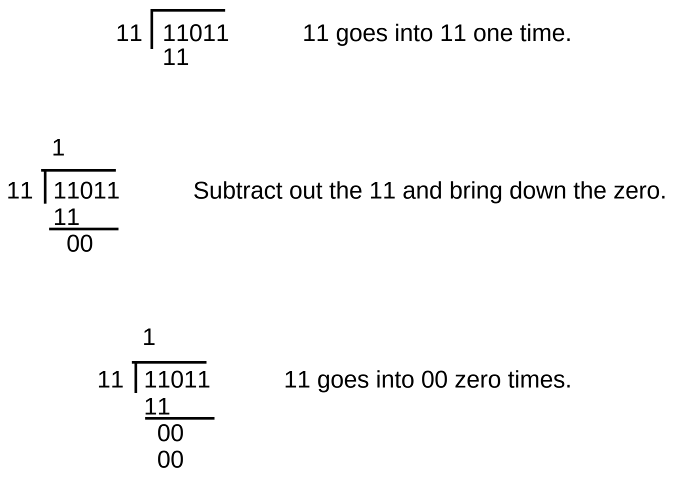
As for multiplication, the best way to understand how the computer performs division is to study how you were taught to perform long division by hand. Consider the operation 3456/12 and the steps you would take to manually perform this operation:



(1) 12 goes into 34 two times.
(2) Subtract 24 from 35 and drop down the 105.
(3) 12 goes into 105 eight times.
(4) Subtract 96 from 105 and drop down the 96.
(5) 12 goes into 96 exactly eight times.
(6) Therefore, 12 goes into 3456 exactly 288 times.

Figure 4.3 Manual Digit-by-digit Division Operation

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 goes into 11 one time.
Subtract out the 11 and bring down the zero.
11 goes into 00 zero times.

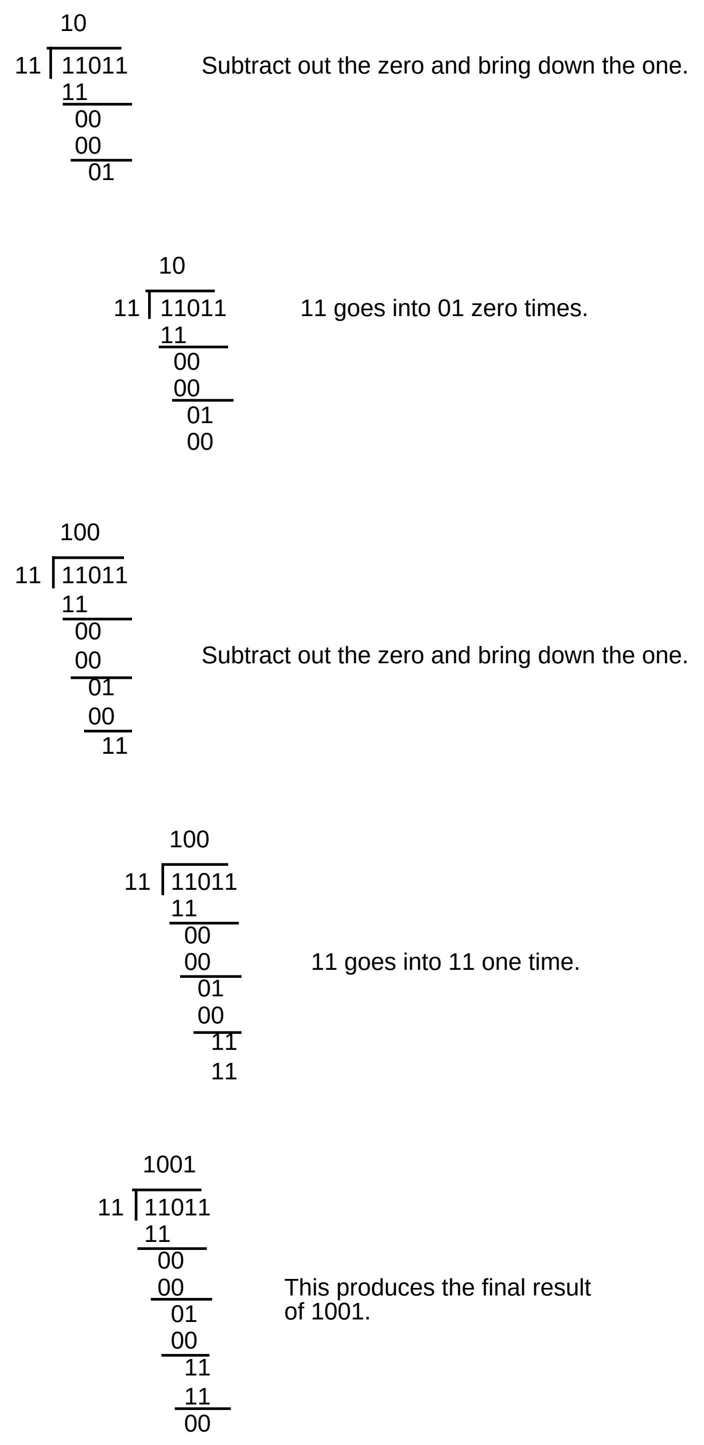

Subtract out the zero and bring down the one.
11 goes into 01 zero times.
Subtract out the zero and bring down the one.
11 goes into 11 one time.
This produces the final result of 1001.

Figure 4.4 Longhand Division in Binary

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 following program implements this algorithm

```
program testDiv128b;
#include( "stdlib.hhf" )

type
    t128:dword[4];



// div128-
//
// This procedure does a general 128/128 division operation
// using the following algorithm:
// (all variables are assumed to be 128 bit objects)
//
// 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
//

procedure div128
(
        Dividend:   t128;
        Divisor:    t128;
    var QuotAdrs:   t128;
    var RmndrAdrs:  t128
);  @nodisplay;

const
    Quotient: text := "Dividend";   // Use the Dividend as the Quotient.

var
    Remainder: t128;

begin div128;

    push( eax );
    push( ecx );
    push( edi );

    mov( 0, eax );              // Set the remainder to zero.
    mov( eax, Remainder[0] );
    mov( eax, Remainder[4] );
    mov( eax, Remainder[8] );
    mov( eax, Remainder[12]);

    mov( 128, ecx );            // Count off 128 bits in ECX.
    repeat

        // Compute Remainder:Quotient := Remainder:Quotient SHL 1:

        shl( 1, Dividend[0] );  // See the section on extended
        rcl( 1, Dividend[4] );  // precision shifts to see how
        rcl( 1, Dividend[8] );  // this code shifts 256 bits to
        rcl( 1, Dividend[12]);  // the left by one bit.
        rcl( 1, Remainder[0] );
        rcl( 1, Remainder[4] );
        rcl( 1, Remainder[8] );
        rcl( 1, Remainder[12]);

        // Do a 128-bit comparison to see if the remainder
        // is greater than or equal to the divisor.

        if
        ( #{
            mov( Remainder[12], eax );
            cmp( eax, Divisor[12] );
            ja true;
            jb false;

            mov( Remainder[8], eax );
            cmp( eax, Divisor[8] );
            ja true;
            jb false;

            mov( Remainder[4], eax );
            cmp( eax, Divisor[4] );
            ja true;
            jb false;

            mov( Remainder[0], eax );
            cmp( eax, Divisor[0] );
            jb false;
        }# ) then

            // Remainder := Remainder - Divisor

            mov( Divisor[0], eax );
            sub( eax, Remainder[0] );

            mov( Divisor[4], eax );
            sbb( eax, Remainder[4] );

            mov( Divisor[8], eax );
            sbb( eax, Remainder[8] );

            mov( Divisor[12], eax );
            sbb( eax, Remainder[12] );
```

```
                // Quotient := Quotient + 1;

                add( 1, Quotient[0] );
                adc( 0, Quotient[4] );
                adc( 0, Quotient[8] );
                adc( 0, Quotient[12] );

            endif;
            dec( ecx );

        until( @z );


        // Okay, copy the quotient (left in the Dividend variable)
        // and the remainder to their return locations.

        mov( QuotAdrs, edi );
        mov( Quotient[0], eax );
        mov( eax, [edi] );
        mov( Quotient[4], eax );
        mov( eax, [edi+4] );
        mov( Quotient[8], eax );
        mov( eax, [edi+8] );
        mov( Quotient[12], eax );
        mov( eax, [edi+12] );

        mov( RmndrAdrs, edi );
        mov( Remainder[0], eax );
        mov( eax, [edi] );
        mov( Remainder[4], eax );
        mov( eax, [edi+4] );
        mov( Remainder[8], eax );
        mov( eax, [edi+8] );
        mov( Remainder[12], eax );
        mov( eax, [edi+12] );


        pop( edi );
        pop( ecx );
        pop( eax );

    end div128;



    // Some simple code to test out the division operation:

    static
        op1:    t128    := [$2222_2221, $4444_4444, $6666_6666, $8888_8888];
        op2:    t128    := [2, 0, 0, 0];
        quo:    t128;
        rmndr:  t128;


    begin testDiv128b;

        div128( op1, op2, quo, rmndr );

        stdout.put
        (
            nl
            nl
            "After the division: " nl
            nl
            "Quotient = $",
            quo[12], "_",
            quo[8], "_",
            quo[4], "_",
            quo[0], nl

            "Remainder = ", (type uns32 rmndr )
        );

    end testDiv128b;
```

Program 4.3 Extended Precision Division

This code looks simple but there are a few problems with it. First, it does not check for division by zero (it will produce the value $FFFF_FFFF_FFFF_FFFF 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 (or RAISE the ex.DivisionError exception). Dealing with signed values is the same as the earlier division algorithm, this problem appears as a programming exercise. The performance of this algorithm, however, leaves a lot to be desired. It's around an order of magnitude or two worse than the DIV/IDIV instructions on the x86 and they are among the slowest instructions on the CPU.

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 uses only 32 bits. Often, even though the divisor is a 128 bit variable, the value itself fits just fine into 32 bits (i.e., the H.O. double words of Divisor are zero). In this special case, that occurs frequently, you can use the DIV instruction which is much faster.

4.2.6 Extended Precision NEG Operations

Although there are several ways to negate an extended precision value, the shortest way for smaller values (96 bits or less) 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 instruction would if you subtracted the destination value from zero. This code takes the following form (assuming you want to negate the 64-bit value in EDX:EAX):

```
neg( edx );
neg( eax );
sbb( 0, edx );
```

The SBB instruction decrements EDX if there is a borrow out of the L.O. word of the negation operation (which always occurs unless EAX 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:

```
static
    Value: dword[4];
      .
      .
      .
    neg( Value[12] );      // Negate the H.O. double word.
    neg( Value[8] );       // Neg previous dword in memory.
    sbb( 0, Value[12] );   // Adjust H.O. dword.

    neg( Value[4] );       // Negate the second dword in the object.
    sbb( 0, Value[8] );    // Adjust third dword in object.
    sbb( 0, Value[12] );   // Adjust the H.O. dword.

    neg( Value );          // Negate the L.O. dword.
    sbb( 0, Value[4] );    // Adjust second dword in object.
    sbb( 0, Value[8] );    // Adjust third dword in object.
    sbb( 0, Value[12] );   // Adjust the H.O. dword.
```

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

```
static
    Value: dword[5];   // 160-bit value.
      .
      .
      .
    mov( 0, eax );
    sub( Value, eax );
    mov( eax, Value );

    mov( 0, eax );
    sbb( Value[4], eax );
    mov( eax, Value[4] );

    mov( 0, eax );
    sbb( Value[8], eax );
    mov( eax, Value[8] );

    mov( 0, eax );
    sbb( Value[12], eax );
    mov( eax, Value[12] );

    mov( 0, eax );
    sbb( Value[16], eax );
    mov( eax, Value[16] );
```

4.2.7 Extended Precision AND Operations

Performing an n-byte AND operation is very easy – simply AND the corresponding bytes between the two operands, saving the result. For example, to perform the AND operation where all operands are 64 bits long, you could use the following code:

```
    mov( (type dword source1), eax );
    and( (type dword source2), eax );
    mov( eax, (type dword dest) );

    mov( (type dword source1[4]), eax );
    and( (type dword source2[4]), eax );
    mov( eax, (type dword dest[4]) );
```

This technique easily extends to any number of words, all you need to is logically AND the corresponding bytes, words, or double words together in the operands. Note that this sequence sets the flags according to the value of the last AND operation. If you AND the H.O. double words last, this sets all but the zero flag correctly. If you need to test the zero flag after this sequence, you will need to logically OR the two resulting double words together (or otherwise compare them both against zero).

4.2.8 Extended Precision OR Operations

Multi-byte logical OR operations are performed in the same way as multi-byte AND operations. You simply OR the corresponding bytes in the two operand together. For example, to logically OR two 96 bit values, use the following code:

```
mov( (type dword source1), eax );
or( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
or( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

mov( (type dword source1[8]), eax );
or( (type dword source2[8]), eax );
mov( eax, (type dword dest[8]) );
```

As for the previous example, this does not set the zero flag properly for the entire operation. If you need to test the zero flag after a multiprecision OR, you must logically OR the resulting double words together.

4.2.9 Extended Precision XOR Operations

Extended precision XOR operations are performed in a manner identical to AND/OR – simply XOR the corresponding bytes 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.

```
mov( (type dword source1), eax );
xor( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
xor( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );
```

The comment about the zero flag in the previous two sections applies here.

4.2.10 Extended Precision NOT Operations

The NOT instruction inverts all the bits in the specified operand. An extended precision NOT is performed by simply executing the NOT instruction on all the affected operands. For example, to perform a 64 bit NOT operation on the value in (edx:eax), all you need to do is execute the instructions:

```
not( eax );
not( edx );
```

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 ($FF, $FFFF, or $FFFF_FFFF) performs the same operation as the NOT instruction.

4.2.11 Extended Precision Shift Operations

Extended precision shift operations require a shift and a rotate instruction. Consider what must happen to implement a 64 bit SHL using 32 bit operations:

1) A zero must be shifted into bit zero.
2) Bits zero through 30 are shifted into the next higher bit.
3) Bit 31 is shifted into bit 32.
4) Bits 32 through 62 must be shifted into the next higher bit.
5) Bit 63 is shifted into the carry flag.

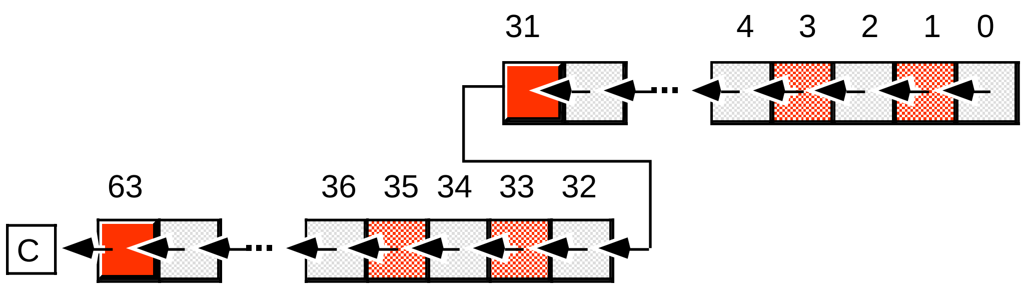


Figure 4.5 64-bit Shift Left Operation

The two instructions you can use to implement this 32 bit shift are SHL and RCL. For example, to shift the 64 bit quantity in (EDX:EAX) one position to the left, you'd use the instructions:

```
    shl( 1, eax );
    rcl( 1, eax );
```

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. Nor can you specify a constant value greater than one 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 64 bit operand and shifts bit 31 into the carry flag. The RCL instruction then shifts the carry flag into bit 32 and then shifts bit 63 into the carry flag. The result is exactly what we want.

To perform a shift left on an operand larger than 64 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 96 bit shift left operation on a memory location you could use the following instructions:

```
    shl( 1, (type dword Operand[0]) );
    rcl( 1, (type dword Operand[4])  );
    rcl( 1, (type dword Operand[8])  );
```

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 96 bit value *Operand* to the left the number of bits specified in ECX:

```
ShiftLoop:
    shl( 1, (type dword Operand[0]) );
    rcl( 1, (type dword Operand[4]) );
    rcl( 1, (type dword Operand[8]) );
    dec( ecx );
    jnz 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:

```
// Double precision SAR:

    sar( 1, (type dword Operand[8]) );
    rcr( 1, (type dword Operand[4]) );
    rcr( 1, (type dword Operand[0]) );

// Double precision SHR:

    shr( 1, (type dword Operand[8]) );
    rcr( 1, (type dword Operand[4]) );
    rcr( 1, (type dword Operand[0]) );
```

There is one major difference between the extended precision shifts described here and their 8/16/32 bit counterparts – the extended precision shifts set the flags differently than the single precision operations. This is because the rotate instructions affect the flags differently than the shift instructions. Fortunately, the carry is the flag most often tested after a shift operation and the extended precision shift operations (i.e., rotate instructions) properly set this flag.

The SHLD and SHRD instructions let you efficiently implement multiprecision shifts of several bits. These instructions have the following syntax:

```
shld( constant, Operand1, Operand2 );
shld( cl, Operand1, Operand2 );
shrd( constant, Operand1, Operand2 );
shrd( cl, Operand1, Operand2 );
```

The SHLD instruction does the following:

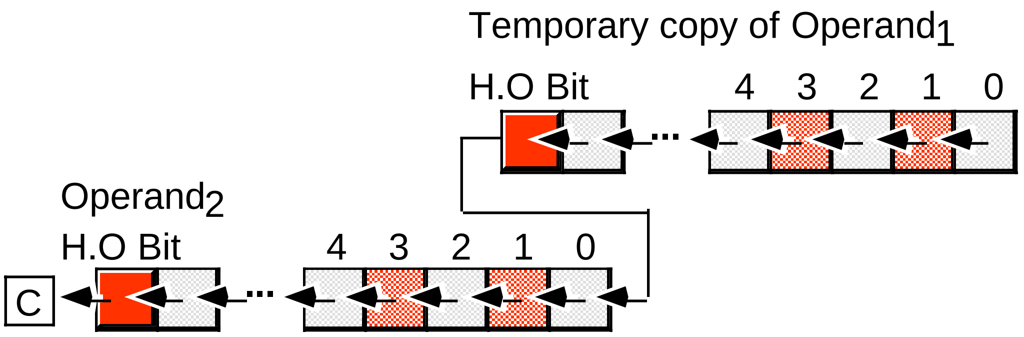


Figure 4.6 SHLD Operation

Operand$_1$ must be a 16 or 32 bit register. *Operand$_2$* can be a register or a memory location. Both operands must be the same size. The immediate operand can be a value in the range zero through n-1, where n is the number of bits in the two operands; it specifies the number of bits to shift.

The SHLD instruction shifts bits in *Operand$_2$* to the left. The H.O. bits shift into the carry flag and the H.O. bits of *Operand1* shift into the L.O. bits of *Operand$_2$*. Note that this instruction does not modify the value of *Operand$_1$*, it uses a temporary copy of *Operand$_1$* during the shift. The immediate operand specifies the number of bits to shift. If the count is n, then SHLD shifts bit n-1 into the carry flag. It also shifts the H.O. n bits of *Operand$_1$* into the L.O. n bits of *Operand$_2$*. The SHLD instruction sets the flag bits as follows:

- If the shift count is zero, the SHLD instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the *Operand*$_2$.
- If the shift count is one, the overflow flag will contain one if the sign bit of *Operand*$_2$ changes during the shift. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

The SHRD instruction is similar to SHLD except, of course, it shifts its bits right rather than left. To get a clear picture of the SHRD instruction, consider Figure 4.7

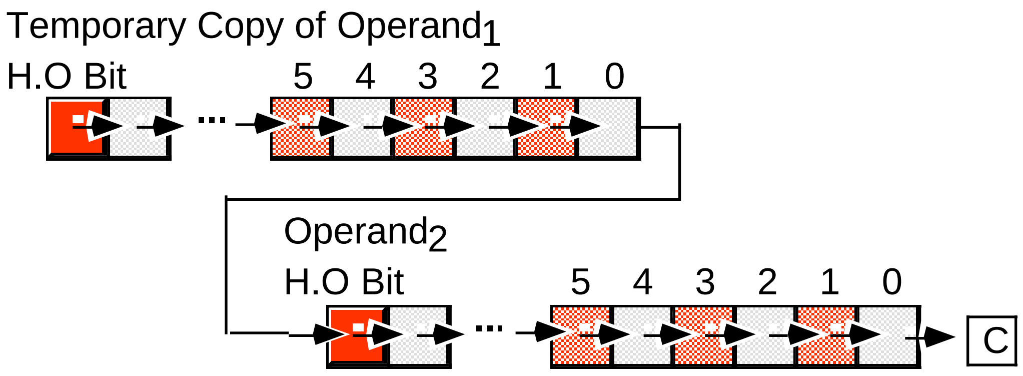


Figure 4.7 SHRD Operation

The SHRD instruction sets the flag bits as follows:

- If the shift count is zero, the SHRD instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the *Operand*$_2$.
- If the shift count is one, the overflow flag will contain one if the H.O. bit of *Operand*$_2$ changes. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

Consider the following code sequence:

```
static
    ShiftMe: dword[3] := [ $1234, $5678, $9012 ];
      .
      .
      .
    mov( ShiftMe[4], eax )
    shld( 6, eax, ShiftMe[8] );
    mov( ShiftMe[0], eax );
    shld( 6, eax, ShiftMe[4] );
    shl( 6, ShiftMe[0] );
```

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.

4.2.12 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 96 bit RCL and RCR operations, use the following instructions:

```
rcl( 1, (type dword Operand[0]) );
rcl( 1, (type dword Operand[4])  );
rcl( 1, (type dword Operand[8])  );

rcr( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4])  );
rcr( 1, (type dword Operand[0])  );
```

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. You can use the BT, SHLD, and SHRD instructions to 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( 4, EDX:EAX );

    mov( edx, ebx );
    shld, 4, eax, edx );
    shld( 4, ebx, eax );
    bt( 0, eax );        // 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.

4.2.13 Extended Precision I/O

Once you have the ability to compute using extended precision arithmetic, the next problem is how do you get those extended precision values into your program and how do you display those extended precision values to the user? HLA's Standard Library provides routines for unsigned decimal, signed decimal, and hexadecimal I/O for values that are eight, 16, 32, or 64 bits in length. So as long as you're working with values whose size is less than or equal to 64 bits in length, you can use the Standard Library code. If you need to input or output values that are greater than 64 bits in length, you will need to write your own procedures to handle the operation. This section discusses the strategies you will need to write such routines.

The examples in this section work specifically with 128-bit values. The algorithms are perfectly general and extend to any number of bits (indeed, the 128-bit algorithms in this section are really nothing more than an extension of the algorithms the HLA Standard Library uses for 64-bit values). If you need a set of 128-bit unsigned I/O routines, you will probably be able to use the following code as-is. If you need to handle larger values, simple modifications to the following code is all that should be necessary.

The following examples all assume a common data type for 128-bit values. The HLA type declaration for this data type is one of the following depending on the type of value

```
type
    bits128: dword[4];
    uns128: bits128;
    int128: bits128;
```

4.2.13.1 Extended Precision Hexadecimal Output

Extended precision hexadecimal output is very easy. All you have to do is output each double word component of the extended precision value from the H.O. double word to the L.O. double word using a call to the *stdout.putd* routine. The following procedure does exactly this to output a *bits128* value:

```
procedure putb128( b128: bits128 ); nodisplay;
begin putb128;

    stdout.putd( b128[12] );
    stdout.putd( b128[8] );
    stdout.putd( b128[4] );
    stdout.putd( b128[0] );

end putb128;
```

Since HLA provides the *stdout.putq* procedure, you can shorten the code above by calling *stdout.putq* just twice:

```
procedure putb128( b128: bits128 ); nodisplay;
begin putb128;

    stdout.putq( (type qword b128[8]) );
    stdout.putq( (type qword b128[0]) );

end putb128;
```

Note that this code outputs the two quad words with the H.O. quad word output first and L.O. quad word output second.

4.2.13.2 Extended Precision Unsigned Decimal Output

Decimal output is a little more complicated than hexadecimal output because the H.O. bits of a binary number affect the L.O. digits of the decimal representation (this was not true for hexadecimal values which is why hexadecimal output is so easy). Therefore, we will have to create the decimal representation for a binary number by extracting one decimal digit at a time from the number.

The most common solution for unsigned decimal output is to successively divide the value by ten until the result becomes zero. The remainder after the first division is a value in the range 0..9 and this value corresponds to the L.O. digit of the decimal number. Successive divisions by ten (and their corresponding remainder) extract successive digits in the number.

Iterative solutions to this problem generally allocate storage for a string of characters large enough to hold the entire number. Then the code extracts the decimal digits in a loop and places them in the string one by one. At the end of the conversion process, the routine prints the characters in the string in reverse order (remember, the divide algorithm extracts the L.O. digits first and the H.O. digits last, the opposite of the way you need to print them).

In this section, we will employ a recursive solution because it is a little more elegant. The recursive solution begins by dividing the value by 10 and saving the remainder in a local variable. If the quotient was not zero, the routine recursively calls itself to print any leading digits first. On return from the recursive call (which prints all the leading digits), the recursive algorithm prints the digit associated with the remainder to complete the operation. Here's how the operation works when printing the decimal value "123":

- (1) Divide 123 by 10. Quotient is 12, remainder is 3.
- (2) Save the remainder (3) in a local variable and recursively call the routine with the quotient.
- (3) [Recursive Entry 1] Divide 12 by 10. Quotient is 1, remainder is 2.
- (4) Save the remainder (2) in a local variable and recursively call the routine with the quotient.
- (5) [Recursive Entry 2] Divide 1 by 10. Quotient is 0, remainder is 1.
- (6) Save the remainder (1) in a local variable. Since the Quotient is zero, don't call the routine recursively.
- (7) Output the remainder value saved in the local variable (1). Return to the caller (Recursive Entry 1).
- (8) [Return to Recursive Entry 1] Output the remainder value saved in the local variable in recursive entry 1 (2). Return to the caller (original invocation of the procedure).
- (9) [Original invocation] Output the remainder value saved in the local variable in the original call (3). Return to the original caller of the output routine.

The only operation that requires extended precision calculation through this entire algorithm is the "divide by 10" requirement. Everything else is simple and straight-forward. We are in luck with this algorithm, since we are dividing an extended precision value by a value that easily fits into a double word, we can use the fast (and easy) extended precision division algorithm that uses the DIV instruction (see "Extended Precision Division" on page 864). The following program implements a 128-bit decimal output routine utilizing this technique.

```
program out128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    uns128: dword[4];



// DivideBy10-
//
//  Divides "divisor" by 10 using fast
//  extended precision division algorithm
//  that employs the DIV instruction.
//
//  Returns quotient in "quotient"
//  Returns remainder in eax.
//  Trashes EBX, EDX, and EDI.

procedure DivideBy10( dividend:uns128; var quotient:uns128 ); @nodisplay;
begin DivideBy10;

    mov( quotient, edi );
    xor( edx, edx );
    mov( dividend[12], eax );
    mov( 10, ebx );
    div( ebx, edx:eax );
    mov( eax, [edi+12] );

    mov( dividend[8], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+8] );

    mov( dividend[4], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+4] );
    mov( dividend[0], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+0] );
    mov( edx, eax );

end DivideBy10;



// Recursive version of putu128.
// A separate "shell" procedure calls this so that
// this code does not have to preserve all the registers
// it uses (and DivideBy10 uses) on each recursive call.

procedure recursivePutu128( b128:uns128 ); @nodisplay;
var
    remainder: byte;

begin recursivePutu128;

    // Divide by ten and get the remainder (the char to print).

    DivideBy10( b128, b128 );
    mov( al, remainder );       // Save away the remainder (0..9).

    // If the quotient (left in b128) is not zero, recursively
    // call this routine to print the H.O. digits.

    mov( b128[0], eax );    // If we logically OR all the dwords
    or( b128[4], eax );     // together, the result is zero if and
    or( b128[8], eax );     // only if the entire number is zero.
    or( b128[12], eax );
    if( @nz ) then

        recursivePutu128( b128 );

    endif;

    // Okay, now print the current digit.

    mov( remainder, al );
    or( '0', al );          // Converts 0..9 -> '0..'9'.
    stdout.putc( al );

end recursivePutu128;


// Non-recursive shell to the above routine so we don't bother
// saving all the registers on each recursive call.

procedure putu128( b128:uns128 ); @nodisplay;
begin putu128;

    push( eax );
    push( ebx );
    push( edx );
    push( edi );

    recursivePutu128( b128 );
    pop( edi );
    pop( edx );
    pop( ebx );
    pop( eax );

end putu128;



// Code to test the routines above:

static
    b0: uns128 := [0, 0, 0, 0];             // decimal = 0
    b1: uns128 := [1234567890, 0, 0, 0];    // decimal = 1234567890
    b2: uns128 := [$8000_0000, 0, 0, 0];    // decimal = 2147483648
    b3: uns128 := [0, 1, 0, 0 ];            // decimal = 4294967296

    // Largest uns128 value
    // (decimal=340,282,366,920,938,463,463,374,607,431,768,211,455):

    b4: uns128 := [$FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF ];

begin out128;

    stdout.put( "b0 = " );
    putu128( b0 );
    stdout.newln();

    stdout.put( "b1 = " );
    putu128( b1 );
    stdout.newln();

    stdout.put( "b2 = " );
    putu128( b2 );
    stdout.newln();

    stdout.put( "b3 = " );
    putu128( b3 );
    stdout.newln();

    stdout.put( "b4 = " );
    putu128( b4 );
    stdout.newln();

end out128;
```

Program 4.4 128-bit Extended Precision Decimal Output Routine

4.2.13.3 Extended Precision Signed Decimal Output

Once you have an extended precision unsigned decimal output routine, writing an extended precision signed decimal output routine is very easy. The basic algorithm takes the following form:

- Check the sign of the number. If it is positive, call the unsigned output routine to print it.

- If the number is negative, print a minus sign. Then negate the number and call the unsigned output routine to print it.

To check the sign of an extended precision integer, of course, you simply test the H.O. bit of the number. To negate a large value, the best solution is to probably subtract that value from zero. Here’s a quick version of *puti128* that uses the *putu128* routine from the previous section.

```
procedure puti128( i128: int128 ); nodisplay;
begin puti128;

    if( (type int32 i128[12]) < 0 ) then

        stdout.put( ’-’ );

        // Extended Precision Negation:

        push( eax );
        mov( 0, eax );
        sub( i128[0], eax );
        mov( eax, i128[0] );

        mov( 0, eax );
        sbb( i128[4], eax );
        mov( eax, i128[4] );

        mov( 0, eax );
        sbb( i128[8], eax );
        mov( eax, i128[8] );

        mov( 0, eax );
        sbb( i128[12], eax );
        mov( eax, i128[12] );
        pop( eax );

    endif;
    putu128( (type uns128 i128));

end puti128;
```

4.2.13.4 Extended Precision Formatted I/O

The code in the previous two sections prints signed and unsigned integers using the minimum number of necessary print positions. To create nicely formatted tables of values you will need the equivalent of a *puti128Size* or *putu128Size* routine. Once you have the "unformatted" versions of these routines, implementing the formatted versions is very easy.

The first step is to write an "i128Size" and a "u128Size" routine that computes the minimum number of digits needed to display the value. The algorithm to accomplish this is very similar to the numeric output routines. In fact, the only difference is that you initialize a counter to zero upon entry into the routine (e.g., the non-recursive shell routine) and you increment this counter rather than outputting a digit on each recursive call. (Don’t forget to increment the counter inside "i128Size" if the number is negative; you must allow for the output of the minus sign.) After the calculation is complete, these routines should return the size of the operand in the EAX register.

Once you have the "i128Size" and "u128Size" routines, writing the formatted output routines is very easy. Upon initial entry into *puti128Size* or *putu128Size*, these routines call the corresponding "size" routine to determine the number of print positions for the number to display. If the value that the "size" routine returns is greater than the absolute value of the minimum size parameter (passed into *puti128Size* or *putu128Size*) all you need to do is call the put routine to print the value, no other formatting is necessary. If the absolute value of the parameter size is greater than the value *i128Size* or *u128Size* returns, then the program must compute the difference between these two values and print that many spaces (or other filler character) before printing the number (if the parameter size value is positive) or after printing the number (if the parameter size value is negative). The actual implementation of these two routines is left as an exercise at the end of the volume. If you have any further questions about how to do this, you can take a look at the HLA Standard Library code for routines like *stdout.putu32Size*.

4.2.13.5 Extended Precision Input Routines

There are a couple of fundamental differences between the extended precision output routines and the extended precision input routines. First of all, numeric output generally occurs without possibility of error[3]; numeric input, on the other hand, must handle the very real possibility of an input error such as illegal characters and numeric overflow. Also, HLA's Standard Library and run-time system encourages a slightly different approach to input conversion. This section discusses those issues that differentiate input conversion from output conversion.

Perhaps the biggest difference between input and output conversion is the fact that output conversion is *unbracketed*. That is, when converting a numeric value to a string of characters for output, the output routine does not concern itself with characters preceding the output string nor does it concerning itself with the characters following the numeric value in the output stream. Numeric output routines convert their data to a string and print that string without considering the context (i.e., the characters before and after the string representation of the numeric value). Numeric input routines cannot be so cavalier; the contextual information surrounding the numeric string is very important.

A typical numeric input operation consists of reading a string of characters from the user and then translating this string of characters into an internal numeric representation. For example, a statement like "stdin.get(i32);" typically reads a line of text from the user and converts a sequence of digits appearing at the beginning of that line of text into a 32-bit signed integer (assuming *i32* is an *int32* object). Note, however, that the *stdin.get* routine skips over certain characters in the string that may appear before the actual numeric characters. For example, *stdin.get* automatically skips any leading spaces in the string. Likewise, the input string may contain additional data beyond the end of the numeric input (for example, it is possible to read two integer values from the same input line), therefore the input conversion routine must somehow determine where the numeric data ends in the input stream. Fortunately, HLA provides a simple mechanism that lets you easily determine the start and end of the input data: the *Delimiters* character set.

The *Delimiters* character set is a variable, internal to HLA, that contains the set of legal characters that may precede or follow a legal numeric value. By default, this character set includes the end of string marker (a zero byte), a tab character, a line feed character, a carriage return character, a space, a comma, a colon, and a semicolon. Therefore, HLA's numeric input routines will automatically ignore any characters in this set that occur on input before a numeric string. Likewise, characters from this set may legally follow a numeric string on input (conversely, if any non-delimiter character follows the numeric string, HLA will raise an *ex.ConversionError* exception).

The *Delimiters* character set is a private variable inside the HLA Standard Library. Although you do not have direct access to this object, the HLA Standard Library does provide two accessor functions, *conv.setDelimiters* and *conv.getDelimiters* that let you access and modify the value of this character set. These two functions have the following prototypes (found in the "conv.hhf" header file):

```
procedure conv.setDelimiters( Delims:cset );
procedure conv.getDelimiters( var Delims:cset );
```

The *conv.SetDelimiters* procedure will copy the value of the *Delims* parameter into the internal *Delimiters* character set. Therefore, you can use this procedure to change the character set if you want to use a different set of delimiters for numeric input. The *conv.getDelimiters* call returns a copy of the internal

3. Technically speaking, this isn't entirely true. It is possible for a device error (e.g., disk full) to occur. The likelihood of this is so low that we can effectively ignore this possibility.

Delimiters character set in the variable you pass as a parameter to the *conv.getDelimiters* procedure. We will use the value returned by *conv.getDelimiters* to determine the end of numeric input when writing our own extended precision numeric input routines.

When reading a numeric value from the user, the first step will be to get a copy of the *Delimiters* character set. The second step is to read and discard input characters from the user as long as those characters are members of the *Delimiters* character set. Once a character is found that is not in the *Delimiters* set, the input routine must check this character and verify that it is a legal numeric character. If not, the program should raise an *ex.IllegalChar* exception if the character's value is outside the range $00..$7f or it should raise the *ex.ConversionError* exception if the character is not a legal numeric character. Once the routine encounters a numeric character, it should continue reading characters as long as they valid numeric characters; while reading the characters the conversion routine should be translating them to the internal representation of the numeric data. If, during conversion, an overflow occurs, the procedure should raise the *ex.ValueOutOfRange* exception.

Conversion to numeric representation should end when the procedure encounters the first delimiter character at the end of the string of digits. However, it is very important that the procedure does not consume the delimiter character that ends the string. That is, the following is incorrect:

```
static
    Delimiters: cset;
         .
         .
         .
    conv.getDelimiters( Delimiters );

    // Skip over leading delimiters in the string:

    while( stdin.getc() in Delimiters ) do  /* getc did the work */ endwhile;
    while( al in {'0'..'9'}) do

        // Convert character in AL to numeric representation and
        // accumulate result...

        stdin.getc();

    endwhile;
    if( al not in Delimiters ) then

        raise( ex.ConversionError );

    endif;
```

The first WHILE loop reads a sequence of delimiter characters. When this first WHILE loop ends, the character in AL is not a delimiter character. So far, so good. The second WHILE loop processes a sequence of decimal digits. First, it checks the character read in the previous WHILE loop to see if it is a decimal digit; if so, it processes that digit and reads the next character. This process continues until the call to *stdin.getc* (at the bottom of the loop) reads a non-digit character. After the second WHILE loop, the program checks the last character read to ensure that it is a legal delimiter character for a numeric input value.

The problem with this algorithm is that it consumes the delimiter character after the numeric string. For example, the colon symbol is a legal delimiter in the default *Delimiters* character set. If the user types the input "123:456" and executes the code above, this code will properly convert "123" to the numeric value one hundred twenty-three. However, the very next character read from the input stream will be the character "4" not the colon character (":"). While this may be acceptable in certain circumstances, Most programmers expect numeric input routines to consume only leading delimiter characters and the numeric digit characters. They do not expect the input routine to consume any trailing delimiter characters (e.g., many programs will read the next character and expect a colon as input if presented with the string "123:456"). Since *stdin.getc* consumes an input character, and there is no way to "put the character back" onto the input stream, some other way of reading input characters from the user, that doesn't consume those characters, is needed[4].

The HLA Standard Library comes to the rescue by providing the *stdin.peekc* function. Like *stdin.getc*, the *stdin.peekc* routine reads the next input character from HLA's internal buffer. There are two major differences between *stdin.peekc* and *stdin.getc*. First, *stdin.peekc* will not force the input of a new line of text from the user if the current input line is empty (or you've already read all the text from the input line). Instead, *stdin.peekc* simply returns zero in the AL register to indicate that there are no more characters on the input line. Since #0 is (by default) a legal delimiter character for numeric values, and the end of line is certainly a legal way to terminate numeric input, this works out rather well. The second difference between *stdin.getc* and *stdin.peekc* is that *stdin.peekc* does not consume the character read from the input buffer. If you call *stdin.peekc* several times in a row, it will always return the same character; likewise, if you call *stdin.getc* immediately after *stdin.peekc*, the call to *stdin.getc* will generally return the same character as returned by *stdin.peekc* (the only exception being the end of line condition). So although we cannot put characters back onto the input stream after we've read them with *stdin.getc*, we can peek ahead at the next character on the input stream and base our logic on that character's value. A corrected version of the previous algorithm might be the following:

```
static
    Delimiters: cset;
         .
         .
         .
    conv.getDelimiters( Delimiters );

    // Skip over leading delimiters in the string:

    while( stdin.peekc() in Delimiters ) do

        // If at the end of the input buffer, we must explicitly read a
        // new line of text from the user.  stdin.peekc does not do this
        // for us.

        if( al = #0 ) then

            stdin.ReadLn();

        else

            stdin.getc();  // Remove delimiter from the input stream.

        endif;

    endwhile;
    while( stdin.peekc in {'0'..'9'}) do

        stdin.getc();     // Remove the input character from the input stream.

        // Convert character in AL to numeric representation and
        // accumulate result...

    endwhile;
    if( al not in Delimiters ) then

        raise( ex.ConversionError );

    endif;
```

4. The HLA Standard Library routines actually buffer up input lines in a string and process characters out of the string. This makes it easy to "peek" ahead one character when looking for a delimiter to end the input value. Your code can also do this, however, the code in this chapter will use a different approach.

Note that the call to *stdin.peekc* in the second WHILE does not consume the delimiter character when the expression evaluates false. Hence, the delimiter character will be the next character read after this algorithm finishes.

The only remaining comment to make about numeric input is to point out that the HLA Standard Library input routines allow arbitrary underscores to appear within a numeric string. The input routines ignore these underscore characters. This allows the user to input strings like "FFFF_F012" and "1_023_596" which are a little more readable than "FFFFF012" or "1023596". To allow underscores (or any other symbol you choose) within a numeric input routine is quite simple; just modify the second WHILE loop above as follows:

```
while( stdin.peekc in {'0'..'9', '_'}) do

    stdin.getc();  // Read the character from the input stream.

    // Ignore underscores while processing numeric input.

    if( al <> '_' ) then

        // Convert character in AL to numeric representation and
        // accumulate result...

    endif;

endwhile;
```

4.2.13.6 Extended Precision Hexadecimal Input

As was the case for numeric output, hexadecimal input is the easiest numeric input routine to write. The basic algorithm for hexadecimal string to numeric conversion is the following:

- Initialize the extended precision value to zero.
- For each input character that is a valid hexadecimal digit, do the following:
    - Convert the hexadecimal character to a value in the range 0..15 ($0..$F).
    - If the H.O. four bits of the extended precision value are non-zero, raise an exception.
    - Multiply the current extended precision value by 16 (i.e., shift left four bits).
    - Add the converted hexadecimal digit value to the accumulator.
- Check the last input character to ensure it is a valid delimiter. Raise an exception if it is not.

The following program implements this extended precision hexadecimal input routine for 128-bit values.

```
program Xin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    b128: dword[4];



procedure getb128( var inValue:b128 ); @nodisplay;
const
    HexChars  := {'0'..'9', 'a'..'f', 'A'..'F', '_'};
var
```

```
        Delimiters: cset;
        LocalValue: b128;

    begin getb128;

        push( eax );
        push( ebx );

        // Get a copy of the HLA standard numeric input delimiters:

        conv.getDelimiters( Delimiters );

        // Initialize the numeric input value to zero:

        xor( eax, eax );
        mov( eax, LocalValue[0] );
        mov( eax, LocalValue[4] );
        mov( eax, LocalValue[8] );
        mov( eax, LocalValue[12] );

        // By default, #0 is a member of the HLA Delimiters
        // character set.  However, someone may have called
        // conv.setDelimiters and removed this character
        // from the internal Delimiters character set.  This
        // algorithm depends upon #0 being in the Delimiters
        // character set, so let's add that character in
        // at this point just to be sure.

        cs.unionChar( #0, Delimiters );


        // If we're at the end of the current input
        // line (or the program has yet to read any input),
        // for the input of an actual character.

        if( stdin.peekc() = #0 ) then

            stdin.readLn();

        endif;



        // Skip the delimiters found on input.  This code is
        // somewhat convoluted because stdin.peekc does not
        // force the input of a new line of text if the current
        // input buffer is empty.  We have to force that input
        // ourselves in the event the input buffer is empty.

        while( stdin.peekc() in Delimiters ) do

            // If we're at the end of the line, read a new line
            // of text from the user; otherwise, remove the
            // delimiter character from the input stream.

            if( al = #0 ) then

                stdin.readLn(); // Force a new input line.

            else
```

```
                stdin.getc();   // Remove the delimiter from the input buffer.

            endif;

        endwhile;

        // Read the hexadecimal input characters and convert
        // them to the internal representation:

        while( stdin.peekc() in HexChars ) do

            // Actually read the character to remove it from the
            // input buffer.

            stdin.getc();

            // Ignore underscores, process everything else.

            if( al <> '_' ) then

                if( al in '0'..'9' ) then

                    and( $f, al );  // '0'..'9' -> 0..9

                else

                    and( $f, al );  // 'a'/'A'..'f'/'F' -> 1..6
                    add( 9, al );   // 1..6 -> 10..15

                endif;

                // Conversion algorithm is the following:
                //
                // (1) LocalValue := LocalValue * 16.
                // (2) LocalValue := LocalValue + al
                //
                // Note that "* 16" is easily accomplished by
                // shifting LocalValue to the left four bits.
                //
                // Overflow occurs if the H.O. four bits of LocalValue
                // contain a non-zero value prior to this operation.

                // First, check for overflow:

                test( $F0, (type byte LocalValue[15]));
                if( @nz ) then

                    raise( ex.ValueOutOfRange );

                endif;

                // Now multiply LocalValue by 16 and add in
                // the current hexadecimal digit (in EAX).

                mov( LocalValue[8], ebx );
                shld( 4, ebx, LocalValue[12] );
                mov( LocalValue[4], ebx );
                shld( 4, ebx, LocalValue[8] );
                mov( LocalValue[0], ebx );
                shld( 4, ebx, LocalValue[4] );
                shl( 4, ebx );
                add( eax, ebx );
                mov( ebx, LocalValue[0] );

            endif;

        endwhile;

        // Okay, we've encountered a non-hexadecimal character.
        // Let's make sure it's a valid delimiter character.
        // Raise the ex.ConversionError exception if it's invalid.

        if( al not in Delimiters ) then

            raise( ex.ConversionError );

        endif;

        // Okay, this conversion has been a success.  Let's store
        // away the converted value into the output parameter.

        mov( inValue, ebx );
        mov( LocalValue[0], eax );
        mov( eax, [ebx] );

        mov( LocalValue[4], eax );
        mov( eax, [ebx+4] );

        mov( LocalValue[8], eax );
        mov( eax, [ebx+8] );

        mov( LocalValue[12], eax );
        mov( eax, [ebx+12] );

        pop( ebx );
        pop( eax );

    end getb128;



    // Code to test the routines above:

    static
        b1:b128;

    begin Xin128;

        stdout.put( "Input a 128-bit hexadecimal value: " );
        getb128( b1 );
        stdout.put
        (
            "The value is: $",
            b1[12], '_',
            b1[8],  '_',
            b1[4],  '_',
            b1[0],
            nl
        );

    end Xin128;
```

Program 4.5 Extended Precision Hexadecimal Input

Extending this code to handle objects that are not 128 bits long is very easy. There are only three changes necessary: you must zero out the whole object at the beginning of the getb128 routine; when checking for overflow (the "test($F, (type byte LocalValue[15]));" instruction) you must test the H.O. four bits of the new object you're processing; and you must modify the code that multiplies LocalValue by 16 (via SHLD) so that it multiplies your object by 16 (i.e., shifts it to the left four bits).

4.2.13.7 Extended Precision Unsigned Decimal Input

The algorithm for extended precision unsigned decimal input is nearly identical to that for hexadecimal input. In fact, the only difference (beyond only accepting decimal digits) is that you multiply the extended precision value by 10 rather than 16 for each input character (in general, the algorithm is the same for any base; just multiply the accumulating value by the input base). The following code demonstrates how to write a 128-bit unsigned decimal input routine.

```
program Uin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    u128: dword[4];



procedure getu128( var inValue:u128 ); @nodisplay;
var
    Delimiters: cset;
    LocalValue: u128;
    PartialSum: u128;

begin getu128;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

    // Get a copy of the HLA standard numeric input delimiters:

    conv.getDelimiters( Delimiters );

    // Initialize the numeric input value to zero:

    xor( eax, eax );
    mov( eax, LocalValue[0] );
    mov( eax, LocalValue[4] );
    mov( eax, LocalValue[8] );
    mov( eax, LocalValue[12] );
```

```
            // By default, #0 is a member of the HLA Delimiters
            // character set.  However, someone may have called
            // conv.setDelimiters and removed this character
            // from the internal Delimiters character set.  This
            // algorithm depends upon #0 being in the Delimiters
            // character set, so let's add that character in
            // at this point just to be sure.

            cs.unionChar( #0, Delimiters );


            // If we're at the end of the current input
            // line (or the program has yet to read any input),
            // for the input of an actual character.

            if( stdin.peekc() = #0 ) then

                stdin.readLn();

            endif;



            // Skip the delimiters found on input.  This code is
            // somewhat convoluted because stdin.peekc does not
            // force the input of a new line of text if the current
            // input buffer is empty.  We have to force that input
            // ourselves in the event the input buffer is empty.

            while( stdin.peekc() in Delimiters ) do

                // If we're at the end of the line, read a new line
                // of text from the user; otherwise, remove the
                // delimiter character from the input stream.

                if( al = #0 ) then

                    stdin.readLn(); // Force a new input line.

                else

                    stdin.getc();   // Remove the delimiter from the input buffer.

                endif;

            endwhile;

            // Read the decimal input characters and convert
            // them to the internal representation:

            while( stdin.peekc() in '0'..'9' ) do

                // Actually read the character to remove it from the
                // input buffer.

                stdin.getc();

                // Ignore underscores, process everything else.

                if( al <> '_' ) then
```

```
and( $f, al );              // '0'..'9' -> 0..9
mov( eax, PartialSum[0] );  // Save to add in later.

// Conversion algorithm is the following:
//
// (1) LocalValue := LocalValue * 10.
// (2) LocalValue := LocalValue + al
//
// First, multiply LocalValue by 10:

mov( 10, eax );
mul( LocalValue[0], eax );
mov( eax, LocalValue[0] );
mov( edx, PartialSum[4] );

mov( 10, eax );
mul( LocalValue[4], eax );
mov( eax, LocalValue[4] );
mov( edx, PartialSum[8] );

mov( 10, eax );
mul( LocalValue[8], eax );
mov( eax, LocalValue[8] );
mov( edx, PartialSum[12] );

mov( 10, eax );
mul( LocalValue[12], eax );
mov( eax, LocalValue[12] );

// Check for overflow.  This occurs if EDX
// contains a none zero value.

if( edx /* <> 0 */ ) then

    raise( ex.ValueOutOfRange );

endif;

// Add in the partial sums (including the
// most recently converted character).

mov( PartialSum[0], eax );
add( eax, LocalValue[0] );

mov( PartialSum[4], eax );
adc( eax, LocalValue[4] );

mov( PartialSum[8], eax );
adc( eax, LocalValue[8] );

mov( PartialSum[12], eax );
adc( eax, LocalValue[12] );

// Another check for overflow.  If there
// was a carry out of the extended precision
// addition above, we've got overflow.

if( @c ) then

    raise( ex.ValueOutOfRange );
```

```
                endif;

            endif;

        endwhile;

        // Okay, we've encountered a non-decimal character.
        // Let's make sure it's a valid delimiter character.
        // Raise the ex.ConversionError exception if it's invalid.

        if( al not in Delimiters ) then

            raise( ex.ConversionError );

        endif;

        // Okay, this conversion has been a success.  Let's store
        // away the converted value into the output parameter.

        mov( inValue, ebx );
        mov( LocalValue[0], eax );
        mov( eax, [ebx] );

        mov( LocalValue[4], eax );
        mov( eax, [ebx+4] );

        mov( LocalValue[8], eax );
        mov( eax, [ebx+8] );

        mov( LocalValue[12], eax );
        mov( eax, [ebx+12] );

        pop( edx );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end getu128;



    // Code to test the routines above:

    static
        b1:u128;

    begin Uin128;

        stdout.put( "Input a 128-bit decimal value: " );
        getu128( b1 );
        stdout.put
        (
            "The value is: $",
            b1[12], '_',
            b1[8],  '_',
            b1[4],  '_',
            b1[0],
            nl
        );

    end Uin128;
```

Program 4.6 Extended Precision Unsigned Decimal Input

As for hexadecimal input, extending this decimal input to some number of bits beyond 128 is fairly easy. All you need do is modify the code that zeros out the *LocalValue* variable and the code that multiplies *LocalValue* by ten (overflow checking is done in this same code, so there are only two spots in this code that require modification).

4.2.13.8 Extended Precision Signed Decimal Input

Once you have an unsigned decimal input routine, writing a signed decimal input routine is easy. The following algorithm describes how to accomplish this:

- Consume any delimiter characters at the beginning of the input stream.
- If the next input character is a minus sign, consume this character and set a flag noting that the number is negative.
- Call the unsigned decimal input routine to convert the rest of the string to an integer.
- Check the return result to make sure it's H.O. bit is clear. Raise the *ex.ValueOutOfRange* exception if the H.O. bit of the result is set.
- If the sign flag was set in step two above, negate the result.

The actual code is left as a programming exercise at the end of this volume.

4.3 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:

```
static
    var1: byte;
    var2: word;
         .
         .
         .
// Unsigned addition:

    movzx( var1, ax );
    add( var2, ax );

// Signed addition:
```

```
movsx( var1, ax );
add( var2, ax );
```

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( var2, ax );      // Load 16 bit value  into AX
         .               // Do some other operations leaving
         .               //  a 16-bit quantity in AX.
        add( var1, al );  // Add in the eight-bit value
        adc( 0, ah );     // 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 $00 or $FF. If a register is available, the best thing to do is the following:

```
    mov( ax, bx );     // BX is the available register.
    movsx( var1, ax );
    add( bx, ax );
```

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

```
        push( ax );          // Save word value.
        movsx( var1, ax );   // Sign extend 8-bit operand to 16 bits.
        add( [esp], ax );   // Add in previous word value
        add( 2, esp );      // Pop junk from stack
```

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

```
    mov( ax, temp );
    movsx( var1, ax );
    add( temp, ax );
```

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.

As a last example, consider adding an eight bit signed value to a quadword (64 bit) value:

```
static
    QVal:qword;
    BVal:int8;
     .
     .
     .
    movsx( BVal, eax );
    cdq();
    add( (type dword QVal), eax );
    adc( (type dword QVal[4]), edx );
```

4.4 Decimal Arithmetic

The 80x86 CPUs use the binary numbering system for their native internal representation. The binary numbering system is, by far, the most common numbering system in use in computer systems today. In days long since past, however, there were computer systems that were based on the decimal (base 10) numbering system rather than the binary numbering system. Consequently, their arithmetic system was decimal based rather than binary. Such computer systems were very popular in systems targeted for business/commercial systems[5]. Although systems designers have discovered that binary arithmetic is almost always better than decimal arithmetic for general calculations, the myth still persists that decimal arithmetic is better for money calculations than binary arithmetic. Therefore, many software systems still specify the use of decimal arithmetic in their calculations (not to mention that there is lots of legacy code out there whose algorithms are only stable if they use decimal arithmetic). Therefore, despite the fact that decimal arithmetic is generally inferior to binary arithmetic, the need for decimal arithmetic still persists.

Of course, the 80x86 is not a decimal computer; therefore we have to play tricks in order to represent decimal numbers using the native binary format. The most common technique, even employed by most so-called decimal computers, is to use the *binary coded decimal*, or BCD representation. The BCD representation (see “Nibbles” on page 56) uses four bits to represent the 10 possible decimal digits. The binary value of those four bits is equal to the corresponding decimal value in the range 0..9. Of course, with four bits we can actually represent 16 different values. The BCD format ignores the remaining six bit combinations.

Table 1: Binary Code Decimal (BCD) Representation

BCD Representation	Decimal Equivalent
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	Illegal
1011	Illegal

5. In fact, until the release of the IBM 360 in the middle 1960’s, most scientific computer systems were binary based while most commercial/business systems were decimal based. IBM pushed their system\360 as a single purpose solution for both business and scientific applications. Indeed, the model designation (360) was derived from the 360 degrees on a compass so as to suggest that the system\360 was suitable for computations "at all points of the compass" (i.e., business and scientific).

Table 1: Binary Code Decimal (BCD) Representation

BCD Representation	Decimal Equivalent
1100	Illegal
1101	Illegal
1110	Illegal
1111	Illegal

Since each BCD digit requires four bits, we can represent a two-digit BCD value with a single byte. This means that we can represent the decimal values in the range 0..99 using a single byte (versus 0..255 if we treat the value as an unsigned binary number). Clearly it takes a bit more memory to represent the same value in BCD as it does to represent the same value in binary. For example, with a 32-bit value you can represent BCD values in the range 0..99,999,999 (eight significant digits) but you can represent values in the range 0..4,294,967,295 (better than nine significant digits) using the binary representation.

Not only does the BCD format waste memory on a binary computer (since it uses more bits to represent a given integer value), but decimal arithmetic is slower. For these reasons, you should avoid the use of decimal arithmetic unless it is absolutely mandated for a given application.

Binary coded decimal representation does offer one big advantage over binary representation: it is fairly trivial to convert between the string representation of a decimal number and the BCD representation. This feature is particularly beneficial when working with fractional values since fixed and floating point binary representations cannot exactly represent many commonly used values between zero and one (e.g., $^1/_{10}$). Therefore, BCD operations can be efficient when reading from a BCD device, doing a simple arithmetic operation (e.g., a single addition) and then writing the BCD value to some other device.

4.4.1 Literal BCD Constants

HLA does not provide, nor do you need, a special literal BCD constant. Since BCD is just a special form of hexadecimal notation that does not allow the values $A..$F, you can easily create BCD constants using HLA's hexadecimal notation. Of course, you must take care not to include the symbols 'A'..'F' in a BCD constant since they are illegal BCD values. As an example, consider the following MOV instruction that copies the BCD value '99' into the AL register:

```
mov( $99, al );
```

The important thing to keep in mind is that you must not use HLA literal decimal constants for BCD values. That is, "mov(95, al);" does not load the BCD representation for ninety-five into the AL register. Instead, it loads $5F into AL and that's an illegal BCD value. Any computations you attempt with illegal BCD values will produce garbage results. Always remember that, even though it seems counter-intuitive, you use hexadecimal literal constants to represent literal BCD values.

4.4.2 The 80x86 DAA and DAS Instructions

The integer unit on the 80x86 does not directly support BCD arithmetic. Instead, the 80x86 requires that you perform the computation using binary arithmetic and use some auxiliary instructions to convert the binary result to BCD. To support packed BCD addition and subtraction with two digits per byte, the 80x86 provides two instructions: decimal adjust after addition (DAA) and decimal adjust after subtraction (DAS). You would execute these two instructions immediately after an ADD/ADC or SUB/SBB instruction to correct the binary result in the AL register.

Two add a pair of two-digit (i.e., single-byte) BCD values together, you would use the following sequence:

```
mov( bcd_1, al );    // Assume that bcd1 and bcd2 both contain
add( bcd_2, al );    // value BCD values.
daa();
```

The first two instructions above add the two byte values together using standard binary arithmetic. This may not produce a correct BCD result. For example, if *bcd_1* contains $9 and *bcd_2* contains $1, then the first two instructions above will produce the binary sum $A instead of the correct BCD result $10. The DAA instruction corrects this invalid result. It checks to see if there was a carry out of the low order BCD digit and adjusts the value (by adding six to it) if there was an overflow. After adjusting for overflow out of the L.O. digit, the DAA instruction repeats this process for the H.O. digit. DAA sets the carry flag if the was a (decimal) carry out of the H.O. digit of the operation.

The DAA instruction only operates on the AL register. It will not adjust (properly) for a decimal addition if you attempt to add a value to AX, EAX, or any other register. Specifically note that DAA limits you to adding two decimal digits (a single byte) at a time. This means that for the purposes of computing decimal sums, you have to treat the 80x86 as though it were an eight-bit processor, capable of adding only eight bits at a time. If you wish to add more than two digits together, you must treat this as a multiprecision operation. For example, to add four decimal digits together (using DAA), you must execute a sequence like the following:

```
    // Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

    mov( bcd_1[0], al );
    add( bcd_2[0], al );
    daa();
    mov( al, bcd_3[0] );
    mov( bcd_1[1], al );
    adc( bcd_2[1], al );
    daa();
    mov( al, bcd_3[1], al );

// Carry is set at this point if there was unsigned overflow.
```

Since a binary addition of a word requires only three instructions, you can see why decimal arithmetic is so expensive[6].

The DAS (decimal adjust after subtraction) adjusts the decimal result after a binary SUB or SBB instruction. You use it the same way you use the DAA instruction. Examples:

```
// Two-digit (one byte) decimal subtraction:

mov( bcd_1, al );    // Assume that bcd1 and bcd2 both contain
sub( bcd_2, al );    // value BCD values.
das();

// Four-digit (two-byte) decimal subtraction.
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

mov( bcd_1[0], al );
sub( bcd_2[0], al );
das();
mov( al, bcd_3[0] );
mov( bcd_1[1], al );
sbb( bcd_2[1], al );
das();
mov( al, bcd_3[1], al );
```

6. You'll also soon see that it's rare to find decimal arithmetic done this way. So it hardly matters.

```
// Carry is set at this point if there was unsigned overflow.
```

Unfortunately, the 80x86 only provides support for addition and subtraction of packed BCD values using the DAA and DAS instructions. It does not support multiplication, division, or any other arithmetic operations. Because decimal arithmetic using these instructions is so limited, you'll rarely see any programs use these instructions.

4.4.3 The 80x86 AAA, AAS, AAM, and AAD Instructions

In addition to the *packed decimal* instructions (DAA and DAS), the 80x86 CPUs support four *unpacked decimal* adjustment instructions. Unpacked decimal numbers store only one digit per eight-bit byte. As you can imagine, this data representation scheme wastes a considerable amount of memory. However, the unpacked decimal adjustment instructions support the multiplication and division operations, so they are marginally more useful.

The instruction mnemonics AAA, AAS, AAM, and AAD stand for "ASCII adjust for Addition, Subtraction, Multiplication, and Division" (respectively). Despite their name, these instructions do not process ASCII characters. Instead, they support an unpacked decimal value in AL whose L.O. four bits contain the decimal digit and the H.O. four bits contain zero. Note, though, that you can easily convert an ASCII decimal digit character to an unpacked decimal number by simply ANDing AL with the value $0F.

The AAA instruction adjusts the result of a binary addition of two unpacked decimal numbers. If the addition of those two values exceeds 10, then AAA will subtract 10 from AL and increment AH by one (as well as set the carry flag). AAA assumes that the two values you add together were legal unpacked decimal values. Other than the fact that AAA works with only one decimal digit at a time (rather than two), you use it the same way you use the DAA instruction. Of course, if you need to add together a string of decimal digits, using unpacked decimal arithmetic will require twice as many operations and, therefore, twice the execution time.

You use the AAS instruction the same way you use the DAS instruction except, of course, it operates on unpacked decimal values rather than packed decimal values. As for AAA, AAS will require twice the number of operations to add the same number of decimal digits as the DAS instruction. If you're wondering why anyone would want to use the AAA or AAS instructions, keep in mind that the unpacked format supports multiplication and division, while the packed format does not. Since packing and unpacking the data is usually more expensive than working on the data a digit at a time, the AAA and AAS instruction are more efficient if you have to work with unpacked data (because of the need for multiplication and division).

The AAM instruction modifies the result in the AX register to produce a correct unpacked decimal result after multiplying two unpacked decimal digits using the MUL instruction. Because the largest product you may obtain is 81 (9*9 produces the largest possible product of two single digit values), the result will fit in the AL register. AAM unpacks the binary result by dividing it by 10, leaving the quotient (H.O. digit) in AH and the remainder (L.O. digit) in AL. Note that AAM leaves the quotient and remainder in different registers than a standard eight-bit DIV operation.

Technically, you do not have to use the AAM instruction immediately after a multiply. AAM simply divides AL by ten and leaves the quotient and remainder in AH and AL (respectively). If you have need of this particular operation, you may use the AAM instruction for this purpose (indeed, that's about the only use for AAM in most programs these days).

If you need to multiply more than two unpacked decimal digits together using MUL and AAM, you will need to devise a multiprecision multiplication that uses the manual algorithm from earlier in this chapter. Since that is a lot of work, this section will not present that algorithm. If you need a multiprecision decimal multiplication, see the next section; it presents a better solution.

The AAD instruction, as you might expect, adjusts a value for unpacked decimal division. The unusual thing about this instruction is that you must execute it *before* a DIV operation. It assumes that AL contains the least significant digit of a two-digit value and AH contains the most significant digit of a two-digit unpacked decimal value. It converts these two numbers to binary so that a standard DIV instruction will produce the correct unpacked decimal result. Like AAM, this instruction is nearly useless for its intended purpose as extended precision operations (e.g., division of more than one or two digits) are extremely inefficient. However, this instruction is actually quite useful in its own right. It computes AX = AH*10+AL (assuming that AH and AL contain single digit decimal values). You can use this instruction to easily convert a two-character string containing the ASCII representation of a value in the range 0..99 to a binary value. E.g.,

```
mov( ’9’, al );
mov( ’9’, ah );    // "99" is in AH:AL.
and( $0F0F, ax );  // Convert from ASCII to unpacked decimal.
aad();             // After this, AX contains 99.
```

The decimal and ASCII adjust instructions provide an extremely poor implementation of decimal arithmetic. To better support decimal arithmetic on 80x86 systems, Intel incorporated decimal operations into the FPU. The next section discusses how to use the FPU for this purpose. However, even with FPU support, decimal arithmetic is inefficient and less precise than binary arithmetic. Therefore, you should carefully consider whether you really need to use decimal arithmetic before incorporating it into your programs.

4.4.4 Packed Decimal Arithmetic Using the FPU

To improve the performance of applications that rely on decimal arithmetic, Intel incorporated support for decimal arithmetic directly into the FPU. Unlike the packed and unpacked decimal formats of the previous sections, the FPU easily supports values with up to 18 decimal digits of precision, all at FPU speeds. Furthermore, all the arithmetic capabilities of the FPU (e.g., transcendental operations) are available in addition to addition, subtraction, multiplication, and division. Assuming you can live with *only* 18 digits of precision and a few other restrictions, decimal arithmetic on the FPU is the right way to go if you must use decimal arithmetic in your programs.

The first fact you must note when using the FPU is that it doesn’t really support decimal arithmetic. Instead, the FPU provides two instruction, FBLD and FBSTP, that convert between packed decimal and binary floating point formats when moving data to and from the FPU. The FBLD (float/BCD load) instruction loads an 80-bit packed BCD value unto the top of the FPU stack after converting that BCD value to the IEEE binary floating point format. Likewise, the FBSTP (float/BCD store and pop) instruction pops the floating point value off the top of stack, converts it to a packed BCD value, and stores the BCD value into the destination memory location.

Once you load a packed BCD value into the FPU, it is no longer BCD. It’s just a floating point value. This presents the first restriction on the use of the FPU as a decimal integer processor: calculations are done using binary arithmetic. If you have an algorithm that absolutely positively depends upon the use of decimal arithmetic, it may fail if you use the FPU to implement it[7].

The second limitation is that the FPU supports only one BCD data type: a ten-byte 18-digit packed decimal value. It will not support smaller values nor will it support larger values. Since 18 digits is usually sufficient and memory is cheap, this isn’t a big restriction.

A third consideration is that the conversion between packed BCD and the floating point format is not a cheap operation. The FBLD and FBSTP instructions can be quite slow (more than two orders of magnitude slower than FLD and FSTP, for example). Therefore, these instructions can be costly if you’re doing simple additions or subtractions; the cost of conversion far outweighs the time spent adding the values a byte at a time using the DAA and DAS instructions (multiplication and division, however, are going to be faster on the FPU).

You may be wondering why the FPU’s packed decimal format only supports 18 digits. After all, with ten bytes it should be possible to represent 20 BCD digits. As it turns out, the FPU’s packed decimal format uses the first nine bytes to hold the packed BCD value in a standard packed decimal format (the first byte contains the two L.O. digits and the ninth byte holds the H.O. two digits). The H.O. bit of the tenth byte

7. An example of such an algorithm might by a multiplication by ten by shifting the number one digit to the left. However, such operations are not possible within the FPU itself, so algorithms that misbehave inside the FPU are actually quite rare.

holds the sign bit and the FPU ignores the remaining bits in the tenth byte. If you're wondering why Intel didn't squeeze in one more digit (i.e., use the L.O. four bits of the tenth byte to allow for 19 digits of precision), just keep in mind that doing so would create some possible BCD values that the FPU could not exactly represent in the native floating point format. Hence the limitation to 18 digits.

The FPU uses a one's complement notation for negative BCD values. That is, the sign bit contains a one if the number is negative or zero and it contains a zero if the number is positive or zero (like the binary one's complement format, there are two distinct representations for zero).

HLA's *tbyte* type is the standard data type you would use to define packed BCD variables. The FBLD and FBSTP instructions require a tbyte operand. Unfortunately, the current version of HLA does not let you (directly) provide an initializer for a tbyte variable. One solution is to use the @NOSTORAGE option and initialize the data following the variable declaration. For example, consider the following code fragment:

```
static
    tbyteObject: tbyte; @nostorage
                    byte  $21, $43, $65, 0, 0, 0, 0, 0, 0, 0;
```

This *tbyteObject* declaration tells HLA that this is a tbyte object but does not explicitly set aside any space for the variable (see "The Static Sections" on page 167). The following BYTE directive sets aside ten bytes of storage and initializes these ten bytes with the value $654321 (remember that the 80x86 organizes data from the L.O. byte to the H.O. byte in memory). While this scheme is inelegant, it will get the job done. The chapters on Macros and the Compile-Time Language will discuss a better way to initialize tbyte and qword data.

Because the FPU converts packed decimal values to the internal floating point format, you can mix packed decimal, floating point, and (binary) integer formats in the same calculation. The following program demonstrate how you might achieve this:

```
program MixedArithmetic;
#include( "stdlib.hhf" )

static
    tb: tbyte; @nostorage;
        byte $21,$43,$65,0,0,0,0,0,0,0;

begin MixedArithmetic;

    fbld( tb );
    fmul( 2.0 );
    fiadd( 1 );
    fbstp( tb );
    stdout.put( "bcd value is " );
    stdout.puttb( tb );
    stdout.newln();

end MixedArithmetic;
```

Program 4.7   Mixed Mode FPU Arithmetic

The FPU treats packed decimal values as integer values. Therefore, if your calculations produce fractional results, the FBSTP instruction will round the result according to the current FPU rounding mode. If you need to work with fractional values, you need to stick with floating point results.

4.5 Sample Program

The following sample program demonstrates BCD I/O. The following program provides two procedures, BCDin and BCDout. These two procedures read an 18-digit BCD value from the user (with possible leading minus sign) and write a BCD value to the standard output device.

```
program bcdIO;
#include( "stdlib.hhf" )


// The following is equivalent to TBYTE except it
// lets us easily gain access to the individual
// components of a BCD value.

type
    bcd:record

            LO8:    dword;
            MID8:   dword;
            HO2:    byte;
            Sign:   byte;

        endrecord;





// BCDin-
//
// This function reads a BCD value from the standard input
// device.  The number can be up to 18 decimal digits long
// and may contain a leading minus sign.
//
// This procedure stores the BCD value in the variable passed
// by reference as a parameter to this routine.


procedure BCDin( var input:tbyte ); @nodisplay;
var
    bcdVal:     bcd;
    delimiters: cset;

begin BCDin;

    push( eax );
    push( ebx );

    // Get a copy of the input delimiter characters and
    // make sure that #0 is a member of this set.

    conv.getDelimiters( delimiters );
    cs.unionChar( #0, delimiters );

    // Skip over any leading delimiter characters in the text:

    while( stdin.peekc() in delimiters ) do
```

```
            // If we're at the end of an input line, read a new
            // line of text from the user, otherwise remove the
            // delimiter character from the input stream.

            if( stdin.peekc() = #0 ) then

                stdin.readLn(); // Get a new line of input text.

            else

                stdin.getc();   // Remove the delimeter.

            endif;

        endwhile;


        // Initialize our input accumulator to zero:

        xor( eax, eax );
        mov( eax, bcdVal.LO8 );
        mov( eax, bcdVal.MID8 );
        mov( al, bcdVal.HO2 );
        mov( al, bcdVal.Sign );


        // If the first character is a minus sign, then eat it and
        // set the sign bit to one.

        if( stdin.peekc() = '-' ) then

            stdin.getc();             // Eat the sign character.
            mov( $80, bcdVal.Sign );  // Make this number negative.

        endif;

        // We must have at least one decimal digit in this number:

        if( stdin.peekc() not in '0'..'9' ) then

            raise( ex.ConversionError );

        endif;

        // Okay, read in up to 18 decimal digits:

        while( stdin.peekc() in '0'..'9' ) do

            stdin.getc();   // Read this decimal digit.
            shl( 4, al );   // Move digit to H.O. bits of AL

            mov( 4, ebx );
            repeat

                // Cheesy way to SHL bcdVal by four bits and
                // merge in the new character.

                shl( 1, al );
                rcl( 1, bcdVal.LO8 );
                rcl( 1, bcdVal.MID8 );
                rcl( 1, bcdVal.HO2 );
                // If the user has entered more than 18
                // decimal digits, the carry will be set
                // after the RCL above.  Test that here.

                if( @c ) then

                    raise( ex.ValueOutOfRange );

                endif;
                dec( ebx );

            until( @z );

        endwhile;

        // Be sure that the number ends with a proper delimiter:

        if( stdin.peekc() not in delimiters ) then

            raise( ex.ConversionError );

        endif;


        // Okay, store the ten-byte input result into
        // the location specified by the parameter.

        mov( input, ebx );
        mov( bcdVal.LO8, eax );
        mov( eax, [ebx] );
        mov( bcdVal.MID8, eax );
        mov( eax, [ebx+4] );
        mov( (type word bcdVal.HO2), ax ); // Grabs "Sign" too.
        mov( ax, [ebx+8] );

        pop( ebx );
        pop( eax );

    end BCDin;




    // BCDout-
    //
    // The converse of the above.  Prints the string representation
    // of the packed BCD value to the standard output device.

    procedure BCDout( output:tbyte ); @nodisplay;
    var
        q:qword;

    begin BCDout;

        // This code cheats *big time*.
        // It converts the BCD value to a 64-bit integer
        // and then calls the stdout.puti64 routine to
        // actually print the number.  In theory, this is
        // a whole lot slower than converting the BCD value
        // to ASCII and printing the ASCII chars, however,
        // I/O is so much slower than the conversion that
        // no one will notice the extra time.

        fbld( output );
        fistp( q );
        stdout.puti64( q );


    end BCDout;




    static
        tb1:    tbyte;
        tb2:    tbyte;
        tbRslt: tbyte;

    begin bcdIO;

        stdout.put( "Enter a BCD value: " );
        BCDin( tb1 );
        stdout.put( "Enter a second BCD value: " );
        BCDin( tb2 );

        fbld( tb1 );
        fbld( tb2 );
        fadd();
        fbstp( tbRslt );

        stdout.put( "The sum of " );
        BCDout( tb1 );
        stdout.put( " + " );
        BCDout( tb2 );
        stdout.put( " is " );
        BCDout( tbRslt );
        stdout.newln();

    end bcdIO;
```

Program 4.8 BCD I/O Sample Program

4.6 Putting It All Together

Extended precision arithmetic is one of those activities where assembly language truly shines. It's much easier to perform extended precision arithmetic in assembly language than in most high level languages; it's far more efficient to do it in assembly language, as well. Extended precision arithmetic was, perhaps, the most important subject that this chapter teaches.

Although extended precision arithmetic and logical calculations are important, what good are extended precision calculations if you can't get the extend precision values in and out of the machine? Therefore, this chapter devotes a fair amount of space to describing how to write your own extended precision I/O routines. Between the calculations and the I/O this chapter describes, you're set to perform those really hairy calculations you've always dreamed of!

Although decimal arithmetic is nowhere near as prominent as it once was, the need for decimal arithmetic does arise on occasion. Therefore, this chapter spends some time discussing BCD arithmetic on the 80x86.