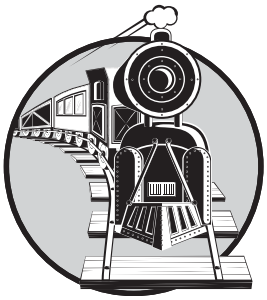
3

BINARY ARITHMETIC AND BIT OPERATIONS



Understanding how computers represent data in binary is a prerequisite to writing software that works well on those computers. Of equal importance, of course, is understanding how computers operate on binary data. Exploring arithmetic, logical, and bit operations on binary data is the purpose of this chapter.

3.1 Arithmetic Operations on Binary and Hexadecimal Numbers

Because computers use binary representation, programmers who write great code often have to work with binary (and hexadecimal) values. Often, when writing code, you may need to manually operate on two binary values in order to use the result in your source code. Although calculators are available to compute such results, you should be able to perform simple arithmetic operations on binary operands by hand.

Hexadecimal arithmetic is sufficiently painful that a hexadecimal calculator belongs on every programmer's desk (or, at the very least, use a software-based calculator that supports hexadecimal operations, such as the Windows calculator). Arithmetic operations on binary values, however, are actually easier than decimal arithmetic. Knowing how to manually compute binary arithmetic results is essential because several important algorithms use these operations (or variants of them). Therefore, the next several subsections describe how to manually add, subtract, multiply, and divide binary values, and how to perform various logical operations on them.

3.1.1 Adding Binary Values

Adding two binary values is easy; there are only eight rules to learn. (If this sounds like a lot, just realize that you had to memorize approximately 200 rules for decimal addition!) Here are the rules for binary addition:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0 with carry
- Carry + 0 + 0 = 1
- Carry + 0 + 1 = 0 with carry
- Carry + 1 + 0 = 0 with carry
- Carry + 1 + 1 = 1 with carry

Once you know these eight rules you can add any two binary values together. Here are some complete examples of binary addition:

```
  0101
+ 0011
------

Step 1: Add the LO bits (1 + 1 = 0 + carry).

   c
  0101
+ 0011
------
   0

Step 2: Add the carry plus the bits in bit position one (carry + 0 + 1 = 0 +
carry).

   c
  0101
+ 0011
-------
   00
```

```
Step 3: Add the carry plus the bits in bit position two (carry + 1 + 0 = 0 +
carry).

          c
          0101
        + 0011
        ------
           000

Step 4: Add the carry plus the bits in bit position three (carry + 0 + 0 = 1).

          0101
        + 0011
        ------
          1000
```

Here are some more examples:

```
  1100_1101       1001_1111       0111_0111
+ 0011_1011     + 0001_0001     + 0000_1001
-----------     -----------     -----------
1_0000_1000       1011_0000       1000_0000
```

3.1.2 Subtracting Binary Values

Binary subtraction is also easy; like addition, binary subtraction has eight rules:

- 0 – 0 = 0
- 0 – 1 = 1 with a borrow
- 1 – 0 = 1
- 1 – 1 = 0
- 0 – 0 – borrow = 1 with a borrow
- 0 – 1 – borrow = 0 with a borrow
- 1 – 0 – borrow = 0
- 1 – 1 – borrow = 1 with a borrow

Here are some complete examples of binary subtraction:

```
          0101
        – 0011
        ------

Step 1: Subtract the LO bits (1 – 1 = 0).

          0101
        – 0011
        ------
             0
```

```
Step 2: Subtract the bits in bit position one (0 – 1 = 1 + borrow).

          0101
        – 0011
           b
        ------
            10

Step 3: Subtract the borrow and the bits in bit position two (1 – 0 – b = 0).

          0101
        – 0011
        ------
           010

Step 4: Subtract the bits in bit position three (0 – 0 = 0).

          0101
        – 0011
        ------
          0010
```

Here are some more examples:

```
  1100_1101       1001_1111       0111_0111
– 0011_1011     – 0001_0001     – 0000_1001
-----------     -----------     -----------
  1001_0010       1000_1110       0110_1110
```

3.1.3 Multiplying Binary Values

Multiplication of binary numbers is also very easy. It's just like decimal multiplication involving only zeros and ones (which is trivial). Here are the rules you need to know for binary multiplication:

- $0 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$

Using these four rules, multiplication is done the same way you'd do decimal multiplication (in fact, if you just follow the rules for decimal multiplication on your binary values you'll actually get the correct results, because the rules for decimal multiplication involving the zero and one digits are identical). Here are some examples of binary multiplication:

```
     1010
   ×  0101
   -------

Step 1: Multiply the LO bit of the multiplier times the multiplicand.

     1010
   ×  0101
   -------
     1010    (1 × 1010)

Step 2: Multiply bit one of the multiplier times the multiplicand.

     1010
   ×  0101
   -------
     1010    (1 × 1010)
    0000     (0 × 1010)
   -------
    01010    (partial sum)

Step 3: Multiply bit two of the multiplier times the multiplicand.

     1010
   ×  0101
   -------
   001010    (previous partial sum)
    1010     (1 × 1010)
   -------
   110010    (partial sum)

Step 4: Multiply bit three of the multiplier times the multiplicand.

     1010
   ×  0101
   -------
   110010    (previous partial sum)
  0000       (0 × 1010)
   -------
  0110010    (product)
```

3.1.4 Dividing Binary Values

Like multiplication of binary numbers, binary division is actually easier than decimal division. You use the same (longhand) division algorithm, but binary division is easier because you can trivially determine whether the divisor goes into the dividend during each step of the longhand division algorithm. Figure 3-1 on the next page shows the steps in a decimal division problem.

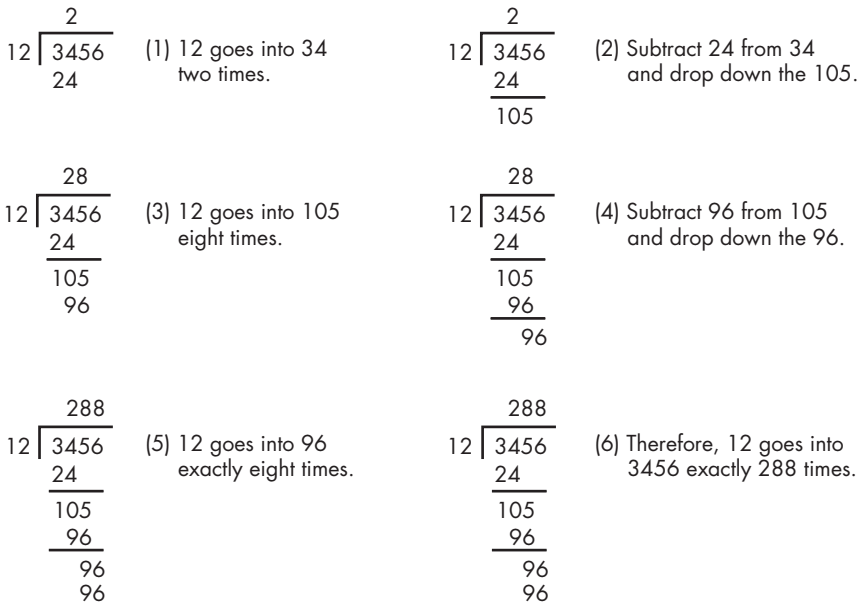
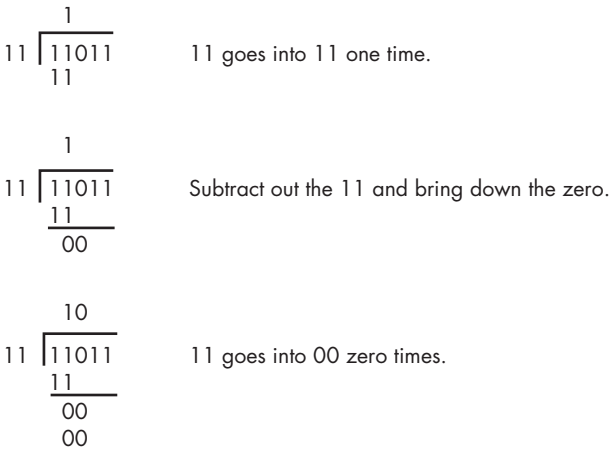


Figure 3-1: Decimal division (3456/12)

This algorithm is actually easier in binary because 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) as shown in Figure 3-2.



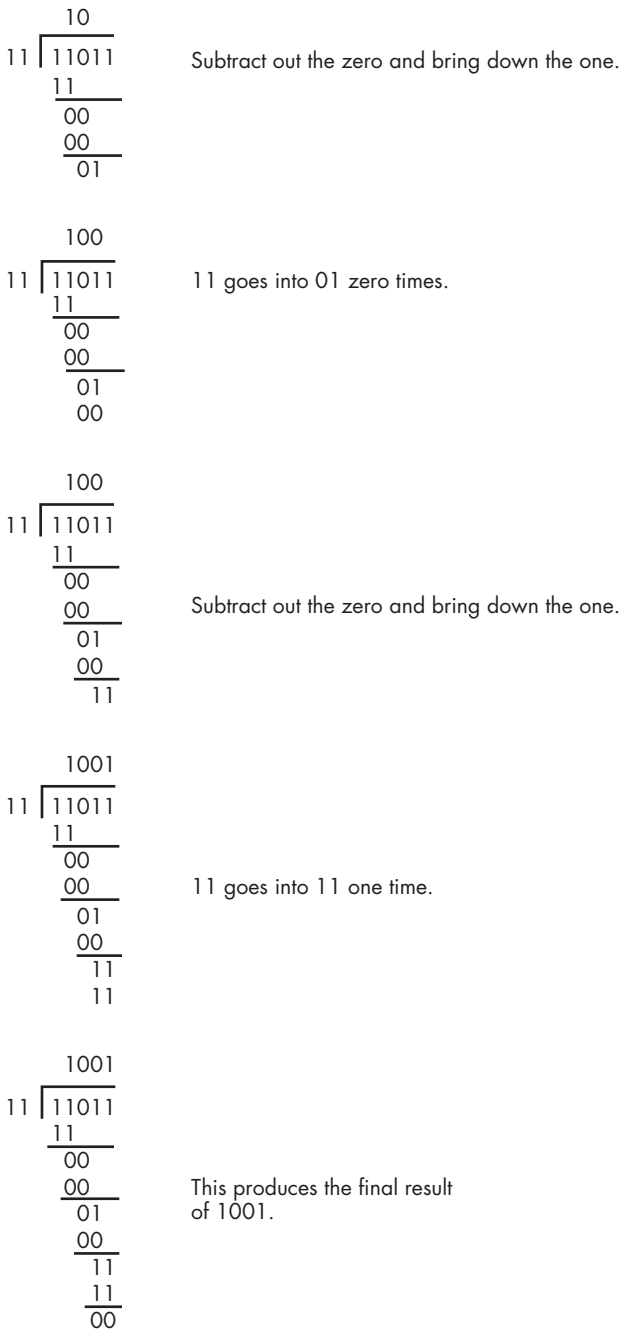


Figure 3-2: Longhand division in binary

3.2 Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND, OR, XOR (exclusive-or), and NOT. Unlike the arithmetic operations, a hexadecimal calculator isn't necessary to perform these operations.

The logical AND, OR, and XOR operations accept two single-bit operands and compute the following results:

```
AND:
            0 and 0 = 0
            0 and 1 = 0
            1 and 0 = 0
            1 and 1 = 1

OR:
            0 or 0 = 0
            0 or 1 = 1
            1 or 0 = 1
            1 or 1 = 1

XOR:
            0 xor 0 = 0
            0 xor 1 = 1
            1 xor 0 = 1
            1 xor 1 = 0
```

Table 3-1, Table 3-2, and Table 3-3 show the *truth tables* for the AND, OR, and XOR operations. A truth table is just like the multiplication tables you encountered in elementary school. The values in the left column correspond to the left operand of the operation. The values in the top row correspond to the right operand of the operation. The value located at the intersection of the row and column (for a particular pair of input values) is the result.

Table 3-1: AND truth table

AND	0	1
0	0	0
1	0	1

Table 3-2: OR truth table

OR	0	1
0	0	1
1	1	1

Table 3-3: XOR truth table

XOR	0	1
0	0	1
1	1	0

In plain English, the logical AND operation translates as, "If the first operand is one and the second operand is one, the result is one; otherwise the result is zero." We could also state this as "If either or both operands are zero, the result is zero." The logical AND operation is useful for forcing a zero result. If one of the operands is zero, the result is always zero regardless of the value of the other operand. If one of the operands contains one, then the result is the value of the other operand.

Colloquially, the logical OR operation is, "If the first operand or the second operand (or both) is one, the result is one; otherwise the result is zero." This is also known as the *inclusive-OR* operation. If one of the operands to the logical-OR operation is one, the result is always one. If an operand is zero, the result is always the value of the other operand.

In English, the logical XOR operation is, "If the first or second operand, but not both, is one, the result is one; otherwise the result is zero." If one of the operands is a one, the result is always the *inverse* of the other operand.

The logical NOT operation is unary (meaning it accepts only one operand). The truth table for the NOT operation appears in Table 3-4. This operator simply inverts (reverses) the value of its operand.

Table 3-4: NOT truth table

NOT	0	1
	1	0

3.3 Logical Operations on Binary Numbers and Bit Strings

The logical functions work on single-bit operands. Because most programming languages manipulate groups of 8, 16, or 32 bits, we need to extend the definition of these logical operations beyond single-bit operands. We can easily extend logical functions to operate on a *bit-by-bit* (or *bitwise*) basis. Given two values, a bitwise logical function operates on bit zero of both operands producing bit zero of the result; it operates on bit one of both operands producing bit one of the result, and so on. For example, if you want to compute the bitwise logical AND of two 8-bit numbers, you would logically AND each pair of bits in the two numbers:

```
%1011_0101
%1110_1110
----------
%1010_0100
```

This bit-by-bit execution also applies to the other logical operations, as well. The ability to force bits to zero or one using the logical AND and OR operations, and the ability to invert bits using the logical XOR operation, is very important when working with strings of bits (such as binary numbers). These operations let you selectively manipulate certain bits within a value while leaving other bits unaffected. For example, if you have an 8-bit binary value *X* and you want to guarantee that bits four through seven contain zeros, you could logically AND the value *X* with the binary value %0000_1111. This bitwise logical AND operation would force the HO four bits of *X* to zero and leave the LO four bits of *X* unchanged. Likewise, you could force the LO bit of *X* to one and invert bit number two of *X* by logically ORing *X* with %0000_0001 and then logically exclusive ORing (XORing) *X* with %0000_0100. Using the logical AND, OR, and XOR operations to manipulate bit strings in this fashion is known as *masking* bit strings. We use the term *masking* because we can use certain values (one for AND, zero for OR and XOR) to "mask out" or "mask in" certain bits in an operand while forcing other bits to zero, one, or their inverse.

Several languages provide operators that let you compute the bitwise AND, OR, XOR, and NOT of their operands. The C/C++/Java language family uses the ampersand (`&`) operator for bitwise AND, the pipe (`|`) operator for bitwise OR, the caret (`^`) operator for bitwise XOR, and the tilde (`~`) operator for bitwise NOT. The Visual Basic and Delphi/Kylix languages let you use the `and`, `or`, `xor`, and `not` operators with integer operands. From 80x86 assembly language, you can use the AND, OR, NOT, and XOR instructions to do these bitwise operations.

```
// Here's a C/C++ example:

    i = j & k;    // Bitwise AND
    i = j | k;    // Bitwise OR
    i = j ^ k;    // Bitwise XOR
    i = ~j;       // Bitwise NOT
```

3.4 Useful Bit Operations

Although bit operations may seem a bit abstract, they are quite useful for many non-obvious purposes. The following subsections describe some of their useful properties of using the logical operations in various languages.

3.4.1 Testing Bits in a Bit String Using AND

You can use the bitwise AND operator to test individual bits in a bit string to see if they are zero or one. If you logically AND a value with a bit string that contains a one in a certain bit position, the result of the logical AND will be zero if the corresponding bit contains a zero, and the result will be nonzero

if that bit position contains one. Consider the following C/C++ code that checks an integer value to see if it is odd or even by testing if bit zero of the integer:

```
IsOdd = (ValueToTest & 1) != 0;
```

In binary form, here's what this bitwise AND operation is doing:

```
xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx  // Assuming ValueToTest is 32 bits
0000_0000_0000_0000_0000_0000_0000_0001  // Bitwise AND with the value one
---------------------------------------
0000_0000_0000_0000_0000_0000_0000_000x  // Result of bitwise AND
```

The result is zero if the LO bit of `ValueToTest` contains a zero in bit position zero. The result is one if `ValueToTest` contains a one in bit position one. This calculation ignores all other bits in `ValueToTest`.

3.4.2 Testing a Set of Bits for Zero/Not Zero Using AND

You can also use the bitwise AND operator to check a set of bits to see if they are all zero. For example, one way to check to see if a number is evenly divisible by 16 is to see if the LO four bits of the value are all zeros. The following Delphi/Kylix statement uses the bitwise AND operator to accomplish this:

```
IsDivisibleBy16 := (ValueToTest and $f) = 0;
```

In binary form, here's what this bitwise AND operation is doing:

```
xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx  // Assuming ValueToTest is 32 bits
0000_0000_0000_0000_0000_0000_0000_1111  // Bitwise AND with $F
---------------------------------------
0000_0000_0000_0000_0000_0000_0000_xxxx  // Result of bitwise AND
```

The result is zero if and only if the LO four bits of `ValueToTest` are all zero, because `ValueToTest` is evenly divisible by 16 only if its LO four bits all contain zero.

3.4.3 Comparing a Set of Bits Within a Binary String

The AND and OR operations are particularly useful if you need to compare a subset of the bits in a binary value against some other value. For example, you might want to compare two 6-bit values found in bits 0, 1, 10, 16, 24, and 31 of a pair of 32-bit values. The trick is to set all the uninteresting bits to zero and then compare the two results.[1]

[1] It's also possible to set all the uninteresting bits to ones via the OR operation, but the AND operator is often more convenient.

Consider the following three binary values; the "x" bits denote bits whose values we don't care about:

```
%1xxxxxx0xxxxxxx1xxxxx0xxxxxxxx10
%1xxxxxx0xxxxxxx1xxxxx0xxxxxxxx10
%1xxxxxx1xxxxxxx1xxxxx1xxxxxxxx11
```

If we compare the first and second binary values (assuming we're only interested in bits 31, 16, 10, 1, and 0), we should find that the two values are equal. If we compare either of the first two values against the third value, we'll find that they are not equal. Furthermore, if we compare either of the first two values against the third, we should discover that the third value is greater than the first two. In C/C++ and assembly, this is how we could compare these values:

```
// C/C++ example

    if( (value1 & 0x81010403) == (value2 & 0x81010403))
    {
        // Do something if bits 31, 24, 16, 10, 1, and 0 of
        //  value1 and value2 are equal
    }

    if( (value1 & 0x81010403) != (value3 & 0x81010403))
    {
        // Do something if bits 31, 24, 16, 10, 1, and 0 of
        //  value1 and value3 are not equal
    }

// HLA/x86 assembly example:

    mov( value1, eax );       // EAX = value1
    and( $8101_0403, eax );   // Mask out unwanted bits in EAX
    mov( value2, edx );       // EDX = value2
    and( $8101_0403, edx );   // Mask out the same set of unwanted bits in EDX
    if( eax = edx ) then      // See if the remaining bits match

        // Do something if bits 31, 24, 16, 10, 1, and 0 of
        //  value1 and value2 are equal

    endif;

    mov( value1, eax );      // EAX = value1
    and( $8101_0403, eax );  // Mask out unwanted bits in EAX
    mov( value3, edx );      // EDX = value2
    and( $8101_0403, edx );  // Mask out the same set of unwanted bits in EDX
```

```
if( eax <> edx ) then    // See if the remaining bits do not match

    // Do something if bits 31, 24, 16, 10, 1, and 0 of
    //  value1 and value3 are not equal

endif;
```

3.4.4 Creating Modulo-n Counters Using AND

The AND operation lets you create efficient *modulo-n counters*. A modulo-n counter counts from zero[2] to some maximum value and then resets to zero. Modulo-n counters are great for creating repeating sequences of numbers such as 0, 1, 2, 3, 4, 5, . . . n–1, 0, 1, 2, 3, 4, 5, . . . n–1, 0, 1, ... You can use such sequences to create circular queues and other objects that reuse array elements upon encountering the end of the data structure. The normal way to create a modulo-n counter is to add one to the counter, divide the result by n, and then keep the remainder. The following code examples demonstrate the implementation of a modulo-n counter in C/ C++, Pascal, and Visual Basic:

```
cntr = (cntr + 1 ) % n;    // C/C++
cntr := (cntr + 1) mod n;  // Pascal/Delphi/Kylix
cntr = (cntr + 1) Mod n    ` Visual Basic
```

The problem with this particular implementation is that division is an expensive operation, requiring far more time to execute than operations such as addition. In general, you'll find it more efficient to implement modulo-n counters using a comparison rather than the remainder operator. Here's a Pascal example:

```
cntr := cntr + 1;      // Pascal example
if( cntr >= n ) then
    cntr := 0;
```

For certain special cases, however, you can increment a modulo-n counter more efficiently and conveniently using the AND operation. You can use the AND operator to create a modulo-n counter when n is a power of two. To create such a modulo-n counter, increment your counter and then logically AND it with the value $n = 2^m–1$ ($2^m–1$ contains ones in bit positions 0..m–1 and zeros everywhere else). Because the AND operation is usually much faster than a division, AND-driven modulo-n counters are much more efficient than those using the remainder operator. Indeed, on most CPUs, using the AND operator is quite a bit faster than using an `if` statement. The following examples show how to implement a modulo-n counter for $n = 32$ using the AND operation:

[2] Actually, they could count down to zero as well, but usually they count up.

```
//Note: 0x3f = 31 = 2^5 – 1, so n = 32 and m = 5

    cntr = (cntr + 1) & 0x3f;    // C/C++ example
    cntr := (cntr + 1) and $3f;  // Pascal/Delphi/Kylix example
    cntr = (cntr + 1) And &h3f   ` Visual Basic example
```

The assembly language code is especially efficient:

```
inc( eax );                      // Compute (eax + 1) mod 32
and( $3f, eax );
```

3.5 Shifts and Rotates

Another set of logical operations on bit strings are the *shift* and *rotate* operations. These functions can be further broken down into *shift lefts, rotate lefts, shift rights,* and *rotate rights*. These operations turn out to be very useful in many programs.

The shift left operation moves each bit in a bit string one position to the left, as shown in Figure 3-3. Bit zero moves into bit position one, the previous value in bit position one moves into bit position two, and so on.

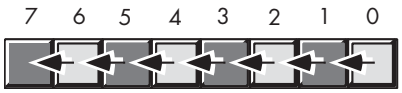


Figure 3-3: Shift left operation (on a byte)

There are two questions that arise: "What goes into bit zero?" and "Where does the HO bit wind up?" We'll shift a zero into bit zero, and the previous value of the HO bit will be the *carry* out of this operation.

Several high-level languages (such as C/C++/C#, Java, and Delphi/Kylix) provide a shift left operator. In the C language family, this operator is <<. In Delphi/Kylix, you use the `shl` operator. Here are some examples:

```
// C:

        cLang = d << 1;     // Assigns d shifted left one position to
                            //  variable "cLang"

// Delphi:

        Delphi := d shl 1;  // Assigns d shifted left one position to
                            //  variable "Delphi"
```

Shifting the binary representation of a number one position to the left is equivalent to multiplying that value by two. Therefore, if you're using a programming language that doesn't provide an explicit shift left operator,

you can usually simulate this by multiplying a binary integer value by two. Although the multiplication operation is usually slower than the shift left operation, most compilers are smart enough to translate a multiplication by a constant power of two into a shift left operation. Therefore, you could write code like the following in Visual Basic to do a shift left:

```
vb = d * 2
```

A shift right operation is similar to a shift left, except we're moving the data in the opposite direction. Bit seven moves into bit six; bit six moves into bit five; bit five moves into bit four; and so on. During a shift right, we'll move a zero into bit seven, and bit zero will be the carry out of the operation (see Figure 3-4). C, C++, C#, and Java use the >> operator for a shift right operation. Delphi/Kylix uses the `shr` operator. Most assembly languages also provide a shift right instruction (`shr` on the 80x86).

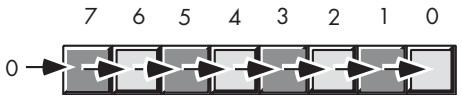


Figure 3-4: The shift right operation (on a byte)

Shifting an unsigned binary value right divides that value by two. For example, if you shift the unsigned representation of 254 ($FE) one place to the right, you get 127 ($7F), exactly as you would expect. However, if you shift the 8-bit two's complement binary representation of –2 ($FE) one position to the right, you get 127 ($7F), which is *not* correct. To divide a signed number by two using a shift, we must define a third shift operation: *arithmetic shift right*. An arithmetic shift right operation does not modify the value of the HO bit. Figure 3-5 shows the arithmetic shift right operation for an 8-bit operand.

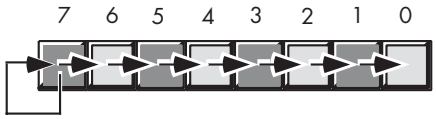


Figure 3-5: Arithmetic shift right operation (on a byte)

This generally produces the result you expect for two's complement signed operands. For example, if you perform the arithmetic shift right operation on –2 ($FE), you get –1 ($FF). Note, however, that this operation always rounds the numbers to the closest integer that is *less than or equal to the actual result*. If you arithmetically shift right –1 ($FF), the result is –1, not zero. Because –1 is less than zero, the arithmetic shift right operation rounds towards –1. This is not a "bug" in the arithmetic shift right operation; it just uses a different (though valid) definition of integer division. The bottom line, however, is that you probably won't be able to use a signed division

operator as a substitute for arithmetic shift right in languages that don't support arithmetic shift right, because most integer division operators round towards zero.

One problem with the shift right operation in high-level languages is that it's rare for a high-level language to support both the logical shift right and the arithmetic shift right. Worse still, the specifications for certain languages leave it up to the compiler's implementer to decide whether to use an arithmetic shift right or a logical shift right operation. Therefore, it's only safe to use the shift right operator on values whose HO bit will cause both forms of the shift right operation to produce the same result. If you need to guarantee that a shift right is a logical shift right or an arithmetic shift right operation, then you'll either have to drop down into assembly language or you'll have to handle the HO bit manually. Obviously, the high-level code gets ugly really fast, so a quick in-line assembly statement might be a better solution if your program doesn't need to be portable across different CPUs. The following code demonstrates how to simulate a 32-bit logical shift right and arithmetic shift right in languages that don't guarantee the type of shift they use:

```
// Written in C/C++, assuming 32-bit integers, logical shift right:
    // Compute bit 30.
    Bit30 = ((ShiftThisValue & 0x800000000) != 0) ? 0x40000000 : 0;
    // Shifts bits 0..30.
    ShiftThisValue = (ShiftThisValue & 0x7fffffff) >> 1;
    // Merge in Bit #30.
    ShiftThisValue = ShiftThisValue | Bit30;

// Arithmetic shift right operation

    Bits3031 = ((ShiftThisValue & 0x800000000) != 0) ? 0xC0000000 : 0;
    // Shifts bits 0..30.
    ShiftThisValue = (ShiftThisValue & 0x7fffffff) >> 1;
    // Merge bits 30/31.
    ShiftThisValue = ShiftThisValue | Bits3031;
```

Many assembly languages also provide various rotate instructions that recirculate bits through an operand by taking the bits shifted out of one end of the operation and shifting them into the other end of the operand. Few high-level languages provide this operation; fortunately, you won't need it very often. If you do, you can synthesize this operation using the shift operators available in your high-level language:

```
// Pascal/Delphi/Kylix Rotate Left, 32-bit example:
    // Puts bit 31 into bit 0, clears other bits.
    CarryOut := (ValueToRotate shr 31);
    ValueToRotate := (ValueToRotate shl 1) or CarryOut;
```

Assembly language programmers typically have access to a wide variety of shift and rotate instructions. For more information on the type of shift and rotate operations that are possible, consult my assembly language programming book, *The Art of Assembly Language* (No Starch Press).

3.6 Bit Fields and Packed Data

CPUs generally operate most efficiently on byte, word, and double-word data types;[3] but occasionally you'll need to work with a data type whose size is something other than 8, 16, or 32 bits. In such cases, you may be able to save some memory by *packing* different strings of bits together as compactly as possible, without wasting any bits to align a particular data field on a byte or other boundary.

Consider a date of the form "04/02/01." It takes three numeric values to represent this date: month, day, and year values. Months, of course, use the values 1..12. It will require at least four bits (a maximum of 16 different values) to represent the month. Days use the range 1..31. Therefore, it will take five bits (a maximum of 32 different values) to represent the day entry. The year value, assuming that we're working with values in the range 0..99, requires seven bits (representing up to 128 different values). Four plus five plus seven is 16 bits, or two bytes. In other words, we can pack our date data into two bytes rather than the three that would be required if we used a separate byte for each of the month, day, and year values. This saves one byte of memory for each date stored, which could be a substantial saving if you need to store many dates. You might arrange the bits as shown in Figure 3-6.

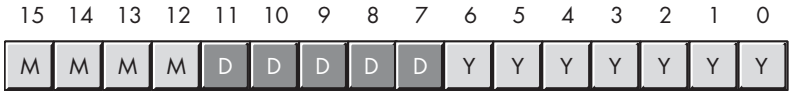


Figure 3-6: Short packed date format (16 bits)

`MMMM` represents the four bits making up the month value, `DDDDD` represents the five bits making up the day, and `YYYYYYY` is the seven bits that hold the year. Each collection of bits representing a data item is a *bit field*. We could represent April 2, 2001, with $4101:

```
0100     00010     0000001     = %0100_0001_0000_0001 or $4101
 04       02        01
```

Although packed values are *space efficient* (that is, they use little memory), they are *computationally inefficient* (slow!). The reason? It takes extra instructions to unpack the data from the various bit fields. These extra instructions take time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data

[3] Some RISC CPUs only operate efficiently on double-word values, so the concept of bit fields and packed data may apply to any object less than 32 bits in size on such CPUs.

fields will save you anything. The following sample HLA/x86 code demonstrates the effort that must go into packing and unpacking this 16-bit date format.

```
program dateDemo;

#include( "stdlib.hhf" )

static
    day:        uns8;
    month:      uns8;
    year:       uns8;

    packedDate: word;

begin dateDemo;

    stdout.put( "Enter the current month, day, and year: " );
    stdin.get( month, day, year );

    // Pack the data into the following bits:
    //
    //  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    //   m  m  m  m  d  d  d  d  d  y  y  y  y  y  y  y

    mov( 0, ax );
    mov( ax, packedDate );  //Just in case there is an error.
    if( month > 12 ) then

        stdout.put( "Month value is too large", nl );

    elseif( month = 0 ) then

        stdout.put( "Month value must be in the range 1..12", nl );

    elseif( day > 31 ) then

        stdout.put( "Day value is too large", nl );

    elseif( day = 0 ) then

        stdout.put( "Day value must be in the range 1..31", nl );

    elseif( year > 99 ) then

        stdout.put( "Year value must be in the range 0..99", nl );
```

```
    else

        mov( month, al );
        shl( 5, ax );
        or( day, al );
        shl( 7, ax );
        or( year, al );
        mov( ax, packedDate );

    endif;

    // Okay, display the packed value:

    stdout.put( "Packed data = $", packedDate, nl );

    // Unpack the date:

    mov( packedDate, ax );
    and( $7f, al );         // Retrieve the year value.
    mov( al, year );

    mov( packedDate, ax );  // Retrieve the day value.
    shr( 7, ax );
    and( %1_1111, al );
    mov( al, day );

    mov( packedDate, ax );  // Retrieve the month value.
    rol( 4, ax );
    and( %1111, al );
    mov( al, month );

    stdout.put( "The date is ", month, "/", day, "/", year, nl );

end dateDemo;
```

Keeping in mind the Y2K[4] problem, adopting a date format that only supports a two-digit year is rather foolish. So consider a better date format, shown in Figure 3-7.

Figure 3-7: Long packed date format (32 bits)

[4] Year 2000, a software engineering disaster that occurred because programmers in the 1900s encoded dates using only two digits and then discovered they couldn't differentiate 1900 and 2000 when the year 2000 came along.

Because there are more bits in a 32-bit variable than are needed to hold the date, even accounting for years in the range 0–65,535, this format allots a full byte for the `month` and `day` fields. Because these two fields are bytes, an application can easily manipulate them as byte objects, reducing the overhead to pack and unpack these fields on those processors that support byte access. This leaves fewer bits for the year, but 65,536 years is probably sufficient (you can probably assume that your software will not be in use 63,000 years from now).

Of course, you could argue that this is no longer a packed date format. After all, we needed three numeric values, two of which fit just nicely into one byte each and one that should probably have at least two bytes. Because this "packed" date format consumes the same four bytes as the unpacked version, what is so special about this format? Well, in this example packed effectively means *packaged* or *encapsulated*. This particular packed format does not use as few bits as possible; by packing the data into a double-word variable the program can treat the date value as a single data value rather than as three separate variables. This generally means that it requires only a single machine instruction to operate on this data rather than three separate instructions.

Another difference you will note between this long packed date format and the short date format appearing in Figure 3-6 is the fact that this long date format rearranges the `Year`, `Month`, and `Day` fields. This is important because it allows you to easily compare two dates using an unsigned integer comparison. Consider the following HLA/assembly code:

```
mov( Date1, eax );        // Assume Date1 and Date2 are double-word variables
if( eax > Date2 ) then    //  using the Long Packed Date format.

    << do something if Date1 > Date2 >>

endif;
```

Had you kept the different date fields in separate variables, or organized the fields differently, you would not have been able to compare `Date1` and `Date2` in such a straightforward fashion. This example demonstrates another reason for packing data, even if you don't realize any space savings — it can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

Some high-level languages provide built-in support for packed data. For example, in C you can define structures like the following:

```
struct
{
    unsigned bits0_3   :4;
    unsigned bits4_11  :8;
    unsigned bits12_15 :4;
    unsigned bits16_23 :8;
    unsigned bits24_31 :8;
} packedData;
```

This structure specifies that each field is an unsigned object that holds four, eight, four, eight, and eight bits, respectively. The ":n" item appearing after each declaration specifies the minimum number of bits the compiler will allocate for the given field.

Unfortunately, it is not possible to provide a diagram that shows how a C/C++ compiler will allocate the values from a 32-bit double word among the fields. No (single) diagram is possible because C/C++ compiler implementers are free to implement these bit fields any way they see fit. The arrangement of the bits within the bit string is arbitrary (for example, the compiler could allocate the `bits0_3` field in bits 28..31 of the ultimate object). The compiler can also inject extra bits between fields as it sees fit. The compiler can use a larger number of bits for each field if it so desires (this is actually the same thing as injecting extra padding bits between fields). Most C compilers attempt to minimize the injection of extraneous padding, but different C compilers (especially on different CPUs) do have their differences. Therefore, any use of C/C++ struct bit field declarations is almost guaranteed to be nonportable, and you can't really count on what the compiler is going to do with those fields.

The advantage of using the compiler's built-in data-packing capabilities is that the compiler automatically handles packing and unpacking the data for you. For example, you could write the following C/C++ code, and the compiler would automatically emit the necessary machine instructions to store and retrieve the individual bit fields for you:

```
struct
{
    unsigned year  :7;
    unsigned month :4;
    unsigned day   :5;
} ShortDate;
        . . .
    ShortDate.day = 28;
    ShortDate.month = 2;
    ShortDate.year = 3;  // 2003
```

3.7 Packing and Unpacking Data

The advantage of packed data types is efficient memory use. Consider the Social Security identification number in use in the United States. This is a nine-digit code that normally takes the following form (each "X" represents a single decimal digit):

```
XXX-XX-XXXX
```

If we encode a Social Security number using three separate (32-bit) integers, it will take 12 bytes to represent this value. That's actually more than the 11 bytes needed to represent the number using an array of characters. A better solution is to encode each field using short (16-bit) integers. Now it takes only 6 bytes to represent the Social Security number. Because the middle field in the Social Security number is always between 0 and 99, we can actually shave one more byte off the size of this structure by encoding the middle field with a single byte. Here's a sample Delphi/Kylix record structure that defines this data structure:

```
SSN :record

        FirstField:  smallint;  // smallints are 16 bits in Delphi/Kylix
        SecondField: byte;
        ThirdField:  smallint;

end;
```

If we drop the hyphens in the Social Security number, you'll notice that the result is a nine-digit number. Because we can exactly represent all values between 0 and 999,999,999 (nine digits) using 30 bits, it should be clear that we could actually encode any legal Social Security number using a 32-bit integer. The problem is that some software that manipulates Social Security numbers may need to operate on the individual fields. This means that you have to use expensive division, modulo, and multiplication operators in order to extract fields from a Social Security number you've encoded in a 32-bit integer format. Furthermore, it's a bit more painful to convert Social Security numbers to and from strings when using the 32-bit format. The advantage of using bit fields to hold a value is that it's relatively easy to insert and extract individual bit fields using fast machine instructions, and it's also less work to create a standard string representation (including the hyphens) of one of these fields. Figure 3-8 provides a straightforward implementation of the Social Security number packed data type using a separate string of bits for each field (note that this format uses 31 bits and ignores the HO bit).

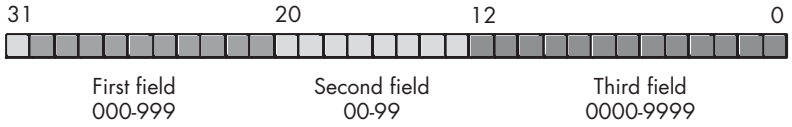


Figure 3-8: Social Security number packed fields encoding

As you'll soon see, fields that begin at bit position zero in a packed data object are the ones you can most efficiently access. So it's generally a good idea to arrange the fields in your packed data type so that the field you access most often begins at bit zero. Of course, you'll have to determine which field you access most often on an application-by-application basis. If you have no idea which field you'll access most often, you should try to assign the fields so they begin on a byte boundary. If there are unused bits in your packed type, you should attempt to spread them throughout the structure so that individual fields begin on a byte boundary and have those fields consume multiples of eight bits.

We've only got one unused bit in the Social Security example shown in Figure 3-8, but it turns out that we can use this extra bit to align two fields on a byte boundary and ensure that one of those fields occupies a bit string whose length is a multiple of eight bits. Consider Figure 3-9, which shows a rearranged version of our Social Security number data type.

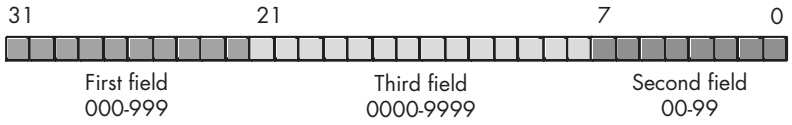


Figure 3-9: A (possibly) improved encoding of the Social Security number

One problem with the data format shown in Figure 3-9 is that we can't sort Social Security numbers in an intuitive fashion by simply comparing 32-bit unsigned integers.[5] Therefore, if you intend to do a lot of sorting based on the entire Social Security number, the format in Figure 3-8 is probably a better format.

If this type of sorting isn't important to you, the format in Figure 3-9 has some advantages. This packed type actually uses eight bits (rather than seven) to represent `SecondField` (along with moving `SecondField` down to bit position zero); the extra bit will always contain zero. This means that `SecondField` consumes bits 0..7 (a whole byte) and `ThirdField` begins on a byte boundary (bit position eight). `ThirdField` doesn't consume a multiple of eight bits, and `FirstField` doesn't begin on a nice byte boundary, but we've done fairly well with this encoding, considering we only had one extra bit to play around with.

[5] "Intuitive" meaning that the first field is the most significant portion of the value, the second field is the next most significant, and the third field is the least significant component of the number.

The next question is, "How do we access the fields of this packed type?" There are two separate activities here. We need the ability to retrieve, or *extract*, the packed fields, and we need to be able to *insert* data into these fields. The AND, OR, and SHIFT operations provide the tools for this.

When actually operating on these fields, it's convenient to work with three separate variables rather than working directly with the packed data. For our Social Security number example, we can create the three variables `FirstField`, `SecondField`, and `ThirdField`. We can then extract the actual data from the packed value into these three variables, operate on these variables, and then insert the data from these three variables back into their fields when we're done.

Extracting the `SecondField` data from the packed format shown in Figure 3-9 is easy (remember, the field aligned to bit zero in our packed data is the easiest one to access). All you have to do is copy the data from the packed representation to the `SecondField` variable and then mask out all but the `SecondField` bits using the AND operation. Because `SecondField` is a 7-bit value, we can create the mask as an integer containing all one bits in positions zero through six and zeros everywhere else. The following C/C++ code demonstrates how to extract this field into the `SecondField` variable (assuming `packedValue` is a variable holding the 32-bit packed Social Security number):

```
SecondField = packedValue & 0x7f;   // 0x7f = %0111_1111
```

Extracting fields that are not aligned at bit zero takes a little more work. Consider the `ThirdField` entry in Figure 3-9. We can mask out all the bits associated with the first and second fields by logically ANDing the packed value with %_11_1111_1111_1111_0000_0000 ($3F_FF00). However, this leaves the `ThirdField` value sitting in bits 8 through 21, which is not convenient for various arithmetic operations. The solution is to shift the masked value down eight bits so that it's aligned at bit zero in our working variable. The following Pascal/Delphi/Kylix code shows how one might do this:

```
SecondField := (packedValue and $3fff00) shr 8;
```

As it turns out, you can also do the shift first and then do the logical AND operation (though this requires a different mask, $11_1111_1111_1111 or $3FFF). Here's the C/C++ code that extracts `SecondField` using that technique:

```
SecondField = (packedValue >> 8) & 0x3FFF;
```

Extracting a field that is aligned against the HO bit, as the first field is in our Social Security packed data type is almost as easy as accessing the data aligned at bit zero. All you have to do is shift the HO field down so that it's aligned at bit zero. The logical shift right operation automatically fills in the HO bits of the result with zeros, so no explicit masking is necessary. The following Pascal/Delphi code demonstrates this:

```
FirstField := packedValue shr 18; // Delphi's shift right is a logical
                                  //  shift right.
```

In HLA/x86 assembly language, it's actually quite easy to access the second and third fields of the packed data format in Figure 3-9. This is because we can easily access data at any arbitrary byte boundary in memory. That allows us to treat both the second and third fields as though they both are aligned at bit zero in the data structure. In addition, because the `SecondField` value is an 8-bit value (with the HO bit always containing zero), it only takes a single machine instruction to unpack the data, as shown here:

```
movzx( (type byte packedValue), eax );
```

This instruction fetches the first byte of `packedValue` (which is the LO 8 bits of `packedValue` on the 80x86), and it zero extends this value to 32 bits in EAX (`movzx` stands for "move with zero extension"). The EAX register, therefore, contains the `SecondField` value after this instruction executes.

Extracting the `ThirdField` value from our packed format isn't quite as easy, because this field isn't an even multiple of eight bits long. Therefore, we'll still need a masking operation to clear the unused bits from the 32-bit result we produce. However, because `ThirdField` is aligned on a byte (8-bit) boundary in our packed structure, we'll be able to avoid the shift operation that was necessary in the high-level code. Here's the HLA/x86 assembly code that extracts the third field from our `packedValue` object:

```
mov( (type word packedValue[1]), ax );  // Extracts bytes 1 & 2
                                        //  from packedValue.
and( $3FFF, eax );                      // Clears all the undesired bits.
```

Extracting `FirstField` from the `packedValue` object in HLA/x86 assembly code is identical to the high-level code; we'll simply shift the upper ten bits (which comprise `FirstField`) down to bit zero:

```
mov( packedValue, eax );
shr( 21, eax );
```

Inserting a field into a packed structure is only a little more complicated than extracting a field. Assuming the data you want to insert appears in some variable and contains zeros in the unused bits, inserting a field into a packed object requires three operations. First, if necessary, you shift the field's data to the left so its alignment matches the corresponding field in the packed object. The second step is to clear the corresponding bits in the packed structure. The final step is to logically OR the shifted field into the packed object. Figure 3-10 on the next page displays the details of this operation.

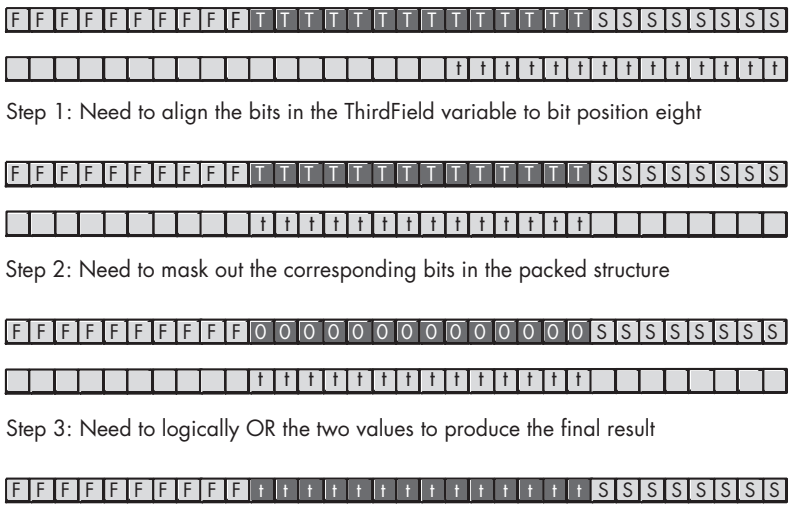


Figure 3-10: Inserting ThirdField into the Social Security packed type

Here's the C/C++ code that accomplishes the operation shown in Figure 3-10:

```
packedValue = (packedValue & 0xFFc000FF) | (ThirdField << 8 );
```

You'll note that $FFC000FF is the hexadecimal value that corresponds to all zeros in bit positions 8 through 21 and ones everywhere else.

3.8 For More Information

My book, *The Art of Assembly Language*, provides additional information on bit processing, including several algorithms for counting bits, reversing the bits in an object, merging two bit strings, coalescing sets of bits, and spreading bits out across some value. Please see that text for more details on these low-level bit operations. Donald Knuth's *The Art of Computer Programming, Volume Two: Seminumerical Algorithms* provides a discussion of various arithmetic operations (addition, subtraction, multiplication, and division) that you may find of interest.