Our discussion so far has focused on logic design as it applies to hardware implementation. Frequently software design also requires the use of binary logic. This section presents some higher-level binary applications, ones that might be found in software. These applications are mostly for error checking and correction, but the techniques used should not be limited to these areas.

9.1 Bitwise Operations

Most software performs data manipulation using mathematical operations such as multiplication or addition. Some applications, however, may require the examination or manipulation of data at the bit level. For example, what might be the fastest way to determine whether an integer is odd or even?

The method most of us are usually taught to distinguish odd and even values is to divide the integer by two discarding any remainder then multiply the result by two and compare it with the original value. If the two values are equal, the original value was even because a division by two would not have created a remainder. Inequality, however, would indicate that the original value was odd. Below is an if-statement in the programming language C that would have performed this check.

```
if(((iVal/2)*2) == iVal)
    // This code is executed for even values
else
    // This code is executed for odd values
```

Let's see if we can't establish another method. As we discussed in Chapter 3, a division by two can be accomplished by shifting all of the bits of an integer one position to the right. A remainder occurs when a one is present in the rightmost bit, i.e., the least significant bit. A zero in this position would result in no remainder. Therefore, if the LSB is one, the integer is odd. If the LSB is zero, the integer is even. This is shown with the following examples.

$35_{10} = 00100011_2$ $124_{10} = 01111100_2$
$93_{10} = 01011101_2$ $30_{10} = 00011110_2$

This reduces our odd/even detection down to an examination of the LSB. The question is can we get the computer to examine only a single bit, and if we can, will it be faster than our previous example?

There is in fact a way to manipulate data at the bit level allowing us to isolate or change individual bits. It is based on a set of functions called ***bitwise operations***, and the typical programming language provides operators to support them.

The term bitwise operation refers to the setting, clearing, or toggling of individual bits within a binary number. To do this, all processors are capable of executing logical operations (AND, OR, or XOR) on the individual pairs of bits within two binary numbers. The bits are paired up by matching their bit position, performing the logical operation, then placing the result in the same bit position of the destination value.

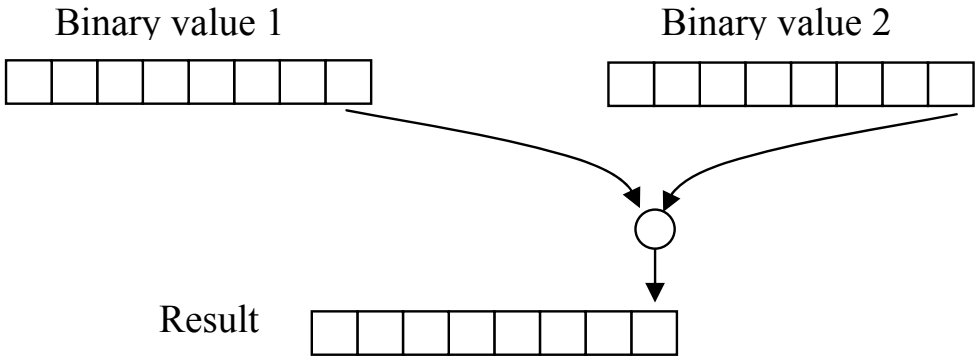


Figure 9-1 Graphic of a Bitwise Operation Performed on LSB

As an example, Figure 9-2 presents the bitwise AND of the binary values 01101011_2 and 11011010_2.

Value 1	0	1	1	0	1	0	1	1
Value 2	1	1	0	1	1	0	1	0
Resulting AND	0	1	0	0	1	0	1	0

Figure 9-2 Bitwise AND of 01101011_2 and 11011010_2

Remember that the output of an AND is one if and only if all of the inputs are one. Examining Figure 9-2, we see that the only ones that appear in the result are in the columns where both of the original values

have ones. In the C programming language, the bitwise AND is indicated with the operator '&'. The example in Figure 9-2 can then be represented in C with the following code.

```
int iVal1 = 0b01101011;
int iVal2 = 0b11011010;
int result = iVal1 & iVal2;
```

Note that the prefix '0b' is a non-standard method of declaring a binary integer and is not supported by all C compilers. If your compiler does not support this type of declaration, use the hex prefix '0x' and declare iVal1 to be 0x6B and iVal2 to be 0xDA. As for the other bitwise operators in C, the bitwise operator for OR is '|' while the bitwise operator for the XOR is '^'.

Typically, bitwise operations are intended to manipulate the bits of a single variable. In order to do this, we must know two things: what operation needs to be done to the bits and what is the "mask" we need to use.

As for the first item, there are three operations: clearing bits to zero, setting bits to one, and toggling bits from one to zero or from zero to one. Clearing bits is taken care of with the bitwise AND operation while setting bits is done with the bitwise OR. The bitwise XOR will toggle specific bits.

The mask is a binary value that is of the same length as the original value. It has a pattern of ones and zeros that defines which bits of the original value are to be changed and which bits are to be left alone.

The next three sections discuss each of the three types of bitwise operations: clearing bits, setting bits, and toggling bits.

9.1.1 Clearing/Masking Bits

Clearing individual bits, also known as bit masking, uses the bitwise logical AND to clear individual bits while leaving the other bits untouched. The mask that is used will have ones in the bit positions that are to be left alone while zeros are in the bit positions that need to be cleared.

This operation is most commonly used when we want to isolate a bit or a group of bits. It is the perfect operation for distinguishing odd and even numbers where we want to see what the LSB is set to and ignore the remaining bits. The bitwise AND, can be used to clear all of the bits above the LSB but leave the LSB alone. The mask we want to use will

have a one in the LSB and zeros in all of the other positions. In Figure 9-3, the results of four bitwise ANDs are given, two for odd numbers and two for even numbers. By ANDing a binary mask of 00000001_2, the odd numbers have a non-zero result while the even numbers have a zero result.

35_{10} (odd)	0	0	1	0	0	0	1	1
Odd/Even Mask	0	0	0	0	0	0	0	1
Resulting AND	0	0	0	0	0	0	0	1
124_{10} (even)	0	1	1	1	1	1	0	0
Odd/Even Mask	0	0	0	0	0	0	0	1
Resulting AND	0	0	0	0	0	0	0	0
93_{10} (odd)	0	1	0	1	1	1	0	1
Odd/Even Mask	0	0	0	0	0	0	0	1
Resulting AND	0	0	0	0	0	0	0	1
30_{10} (even)	0	0	0	1	1	1	1	0
Odd/Even Mask	0	0	0	0	0	0	0	1
Resulting AND	0	0	0	0	0	0	0	0

Figure 9-3 Four Sample Bitwise ANDs

This shows that by using a bitwise AND with a mask of 00000001_2, we can distinguish an odd integer from an even integer. Since bitwise operations are one of the fastest operations that can be performed on a processor, it is the preferred method. In fact, if we use this bitwise AND to distinguish odd and even numbers on a typical processor, it can be twice as fast as doing the same process with a right shift followed by a left shift and over ten times faster than using a divide followed by a multiply. Below is an if-statement in the programming language C that uses a bitwise AND to distinguish odd and even numbers.

```
if(!(iVal&0b00000001))
    // This code is executed for even values
else
    // This code is executed for odd values
```

The bitwise AND can also be used to clear specific bits. For example, assume we want to separate the nibbles of a byte into two different variables. The following process can be used to do this:

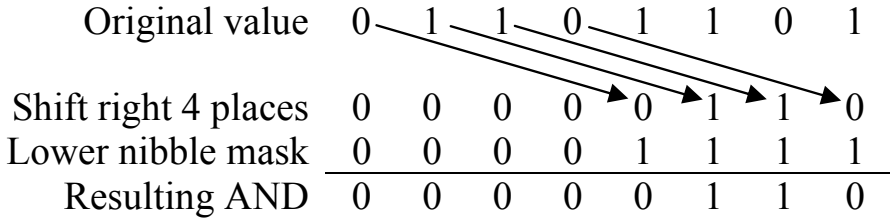
- Copy the original value to the variable meant to store the lower nibble, then clear all but the lower four bits
- Copy the original value to the variable meant to store the upper nibble, then shift the value four bits to the right. (See Section 3.7, "Multiplication and Division by Powers of Two," to see how to shift right using C.) Lastly, clear all but the lower four bits.

This process is demonstrated below using the byte 01101101_2.

Isolating the lower nibble

Original value	0	1	1	0	1	1	0	1
Lower nibble mask	0	0	0	0	1	1	1	1
Resulting AND	0	0	0	0	1	1	0	1

Isolating the upper nibble



The following C code will perform these operations.

```
lower_nibble = iVal & 0x0f;
upper_nibble = (iVal>>4) & 0x0f;
```

Example

Using bitwise operations, write a function in C to determine if an IPv4 address is a member of the subnet 192.168.12.0 with a subnet mask of 255.255.252.0. Return a 1 (true) if the IP address is a member and a 0 (false) if it is not.

Solution

An IPv4 address consists of four bytes or octets separated from one another with periods or "dots". When converted to binary representation, an IPv4 address becomes a 32 bit number.

The address is divided into two parts: a subnet id and a host id. All of the computers that are connected to the same subnet, e.g., a company or a school network, have the same subnet id. Each computer on a subnet, however, has a unique host id. The host id allows the computer to be uniquely identified among all of the computers on the subnet.

The subnet mask identifies the bits that represent the subnet id. When we convert the subnet mask in this example, 255.255.252.0, to binary, we get:

11111111.11111111.11111100.00000000

The bits that identify the subnet id of an IP address correspond to the positions with ones in the subnet mask. The positions with zeros in the subnet mask identify the host id. For this example, the first 22 bits of any IPv4 address going to this subnet should be the same, specifically they should equal the address 192.168.12.0 or in binary:

11000000.10101000.00001100.00000000

So how can we determine if an IPv4 address is a member of this subnet? If we could clear the bits of the host id, then the remaining bits should equal 192.168.12.0. This sounds like the bitwise AND. If we perform a bitwise AND on an IP address using the subnet mask 255.255.252.0, then the result should be 192.168.12.0 because the host id will be cleared. Let's do this by hand for one address inside the subnet, 192.168.15.23, and one address outside the subnet, 192.168.31.23. First, convert these two addresses to binary.

192.168.15.23 = 11000000.10101000.00001111.00010111
192.168.31.23 = 11000000.10101000.00011111.00010111

Now perform a bitwise AND with each of these addresses to come up with their respective subnets.

IP Address	11000000.10101000.00001111.00010111
Subnet mask	11111111.11111111.11111100.00000000
Bitwise AND	11000000.10101000.00001100.00000000

IP Address	11000000.10101000.00011111.00010111
Subnet mask	11111111.11111111.11111100.00000000
Bitwise AND	11000000.10101000.00011100.00000000

Notice that the result of the first bitwise AND produces the correct subnet address while the second bitwise AND does not. Therefore, the first address is a member of the subnet while the second is not.

The code to do this is shown below. It assumes that the type **`int`** is defined to be at least four bytes long. The left shift operator '<<' used in the initialization of **`sbnt_ID`** and **`sbnt_mask`** pushes each octet of the IP address or subnet mask to the correct position.

```
int subnetCheck(int IP_address)
{
  int sbnt_ID = (192<<24)+(168<<16)+(12<<8)+0;
  int sbnt_mask = (255<<24)+(255<<16)+(252<<8)+0;
  if((sbnt_mask & IP_address) == sbnt_ID)
    return 1;
  else return 0;
}
```

9.1.2 Setting Bits

Individual bits within a binary value can be set to one using the bitwise logical OR. To do this, OR the original value with a binary mask of the same size with ones in all the positions to be set and zeros in all the positions to be left alone. As an example, the operation below sets bit positions 1, 3, and 5 of the binary value 10010110_2.

Original value	1	0	0	1	0	1	1	0
Mask	0	0	1	0	1	0	1	0
Bitwise OR	1	0	1	1	1	1	1	0

Note that bit position 1 was already set, and therefore, this operation should appear to have no affect on that bit.

In the C programming language, the bitwise OR is performed using the operator '|'.

Example

Assume that a control byte is used to control eight sets of lights in an auditorium. Each bit controls a set of lights as follows:

bit 7 – House lighting	bit 3 – Emergency lighting
bit 6 – Work lighting	bit 2 – Stage lighting
bit 5 – Aisle lighting	bit 1 – Orchestra pit lighting
bit 4 – Exit lighting	bit 0 – Curtain lighting

For example, if the house lighting, exit lighting, and stage lighting are all on, the value of the control byte should be 10010100_2. What mask would be used with the bitwise OR to turn on the aisle lighting and the emergency lighting?

Solution

The bitwise OR uses a mask where a one is in each position that needs to be turned on and zeros are placed in the positions meant to be left alone. To turn on the aisle lighting and emergency lighting, bits 5 and 3 must be turned on while the remaining bits are to be left alone. This gives us a mask of 00101000_2.

9.1.3 Toggling Bits

We can also toggle or switch the value of individual bits from 1 to 0 or vice versa. This is done using the bitwise XOR. Let's begin our discussion by examining the truth table for a two-input XOR.

Table 9-1 Truth Table for a Two-Input XOR Gate

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

If we cover up the bottom two rows of this truth table leaving only the rows where A=0 visible, we see that the value of B is passed along to X, i.e., if A=0, then X equals B. If we cover up the rows where A=0 leaving only the rows where A=1 visible, it looks like the inverse of B is passed to X, i.e., if A=1, then X equals the inverse of B. This discussion makes a two-input XOR gate look like a programmable

inverter. If A is zero, B is passed through to the output untouched. If A is one, B is inverted at the output.

Therefore, if we perform a bitwise XOR, the bit positions in the mask with zeros will pass the original value through and bit positions with ones will invert the original value. The example below uses the mask 00101110_2 to toggle bits 1, 2, 3, and 5 of a binary value while leaving the others untouched.

Original value	1	0	0	1	0	1	1	0
Mask	0	0	1	0	1	1	1	0
Bitwise XOR	1	0	1	1	1	0	0	0

Example

Assume a byte is used to control the warning and indicator lights on an automotive dashboard. The following is a list of the bit positions and the dashboard lights they control.

bit 7 – Oil pressure light	bit 3 – Left turn indicator
bit 6 – Temperature light	bit 2 – Right turn indicator
bit 5 – Door ajar light	bit 1 – Low fuel light
bit 4 – Check engine light	bit 0 – High-beams light

Determine the mask to be used with a bitwise XOR that when used once a second will cause the left and right turn indicators to flash when the emergency flashers are on.

Solution

The bitwise XOR uses a mask where a one is in each position that needs to be toggled and zeros are placed in the positions meant to be left alone. To toggle bits 3 and 2 on and off, the mask should have ones in those positions and zeros everywhere else. Therefore, the mask to be used with the bitwise XOR is 00001100_2.

9.2 Comparing Bits with XOR

This brings us to our first method for detecting errors in data: comparing two serial binary streams to see if they are equal. Assume that one device is supposed to send a stream of bits to another device. An example of this might be a motion detector mounted in an upper corner of a room. The motion detector has either a zero output indicating the room is unoccupied or a one output indicating that

something in the room is moving. The output from this motion detector may look like that shown in Figure 9-4.

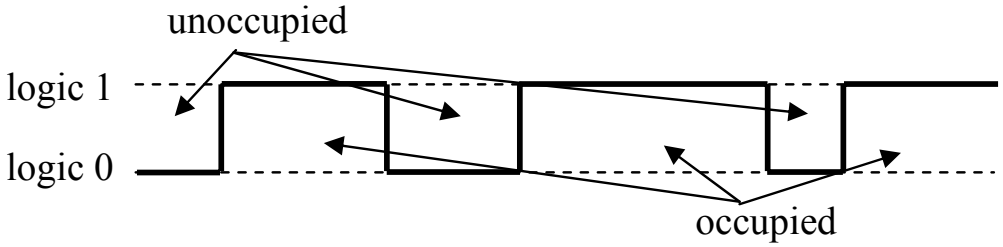


Figure 9-4 Possible Output from a Motion Detector

To verify the output of the motion detector, a second motion detector could be mounted in the room so that the two separate outputs could be compared to each other. If the outputs are the same, the signal can be trusted; if they are different, then one of the devices is in error. At this point in our discussion, we won't know which one.

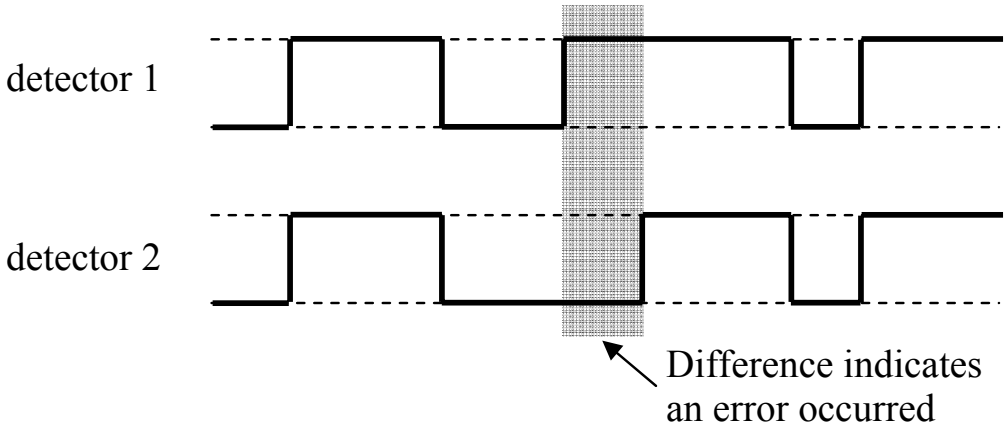


Figure 9-5 A Difference in Output Indicates an Error

A two-input XOR gate can be used here to indicate when an error has occurred. Remember that the output of a two-input XOR gate is a zero if both of the inputs are the same and a one if the inputs are different. This gives us a simple circuit to detect when two signals which should be identical are not.

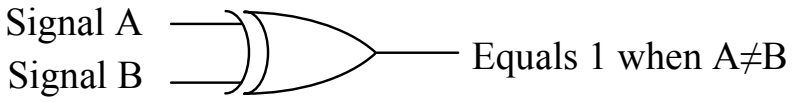


Figure 9-6 Simple Error Detection with an XOR Gate

This circuit will be used later in this chapter to support more complex error detection and correction circuits.

9.3 Parity

One of the most primitive forms of error detection is to add a single bit called a parity bit to each piece of data to indicate whether the data has an odd or even number of ones. It is considered a poor method of error detection as it sometimes doesn't detect multiple errors. When combined with other methods of error detection, however, it can improve their overall performance.

There are two primary types of parity: odd and even. Even parity means that the sum of the ones in the data element and the parity bit is an even number. With odd parity, the sum of ones in the data element and the parity bit is an odd number. When designing a digital system that uses parity, the designers decide in advance which type of parity they will be using.

Assume that a system uses even parity. If an error has occurred and one of the bits in either the data element or the parity bit has been inverted, then counting the number of ones results in an odd number. From the information available, the digital system cannot determine which bit was inverted or even if only one bit was inverted. It can only tell that an error has occurred.

One of the primary problems with parity is that if two bits are inverted, the parity bit appears to be correct, i.e., it indicates that the data is error free. Parity can only detect an odd number of bit errors.

Some systems use a parity bit with each piece of data in memory. If a parity error occurs, the computer will generate a non-maskable interrupt, a condition where the operating system immediately discontinues the execution of the questionable application.

Example

Assume the table below represents bytes stored in memory along with an associated parity bit. Which of the stored values are in error?

Data								Parity
1	0	0	1	0	1	1	0	0
0	0	1	1	1	0	1	0	1
1	0	1	1	0	1	0	1	1
0	1	0	1	1	0	0	1	0
1	1	0	0	0	1	0	1	1

Solution

To determine which data/parity combinations have an error, count the number of ones in each row. The rows with an odd sum have errors while the rows with an even sum are assumed to contain valid data.

Data								Parity	
1	0	0	1	0	1	1	0	0	4 ones – even → no error
0	0	1	1	1	0	1	0	1	5 ones – odd → Error!
1	0	1	1	0	1	0	1	1	6 ones – even → no error
0	1	0	1	1	0	0	1	0	4 ones – even → no error
1	1	0	0	0	1	0	1	1	5 ones – odd → Error!

9.4 Checksum

For digital systems that store or transfer multiple pieces of data in blocks, an additional data element can be added to each block to provide error detection for the block. This method of error detection is common, especially for the transmission of data across networks.

One of the simplest implementations of this method of error detection is the *checksum*. As a device transmits data, it takes the sum of all of the data elements it is transmitting to create an aggregate sum. The overflow carries for this sum are all discarded leaving a final value that contains the same number of bits as the data elements being transmitted. This sum is called the *datasum*. At the end of its transmission, the transmitting device sends some form of its calculated datasum. This new form of the datasum is called the *checksum*.

As the data elements come into the receiving device, they are added a second time in order to recreate the datasum. Once all of the data elements have been received, the receiving device compares its calculated datasum with the checksum sent by the transmitting device. The data is considered error free if the receiving device's datasum compares favorably with the transmitted checksum.

Figure 9-7 presents a sample data block and checksum that might have been transmitted by a device.

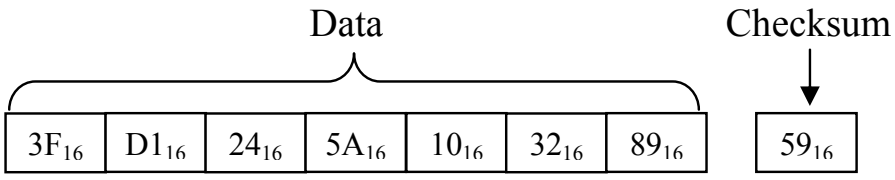


Figure 9-7 Sample Block of Data with Accompanying Checksum

Upon receiving this transmission, the datasum for this data block must be calculated. Begin by taking the sum of all the data elements.

$$3F_{16} + D1_{16} + 24_{16} + 5A_{16} + 10_{16} + 32_{16} + 89_{16} = 259_{16}$$

The final datasum is calculated by discarding any carries that went beyond the byte width defined by the data block. This is equivalent to stripping off all the bits that go past the lowest 8 bits. For the sample block, the datasum would equal 59_{16}.

There is another method of calculating the datasum called a ***one's complement sum***. In this case, the carries are added to the datasum instead of being stripped off. For the sample block, the one's complement datasum would be $59_{16} + 2 = 5B_{16}$.

The checksum shown for the data block in Figure 9-7 is only one of a number of different possible checksums for this data. In this case, the checksum was set equal to the expected datasum. If any of the data elements or if the checksum was in error, the datasum would not equal the checksum. If this happens, the digital system would know that an error had occurred. In the case of a network data transmission, it would request the data to be resent.

The only difference between different implementations of the checksum method is how the datasum and checksum are compared in order to detect an error. As with parity, it is the decision of the designer as to which method is used. The type of checksum used must be agreed upon by both the transmitting and receiving devices ahead of time. The following is a short list of some of the different types of checksum implementations:

- A block of data is considered error free if the datasum is *equal* to the checksum. In this case, the checksum element is calculated by taking the sum of all of the data elements and discarding any carries, i.e., setting the checksum equal to the datasum.
- A block of data is considered error free if the sum of the datasum and checksum results in a binary value with all ones. In this case, the checksum element is calculated by taking the 1's complement of the datasum. This method is called a *1's complement checksum*.
- A block of data is considered error free if the sum of the datasum and checksum results in a binary value with all zeros. In this case, the checksum element is calculated by taking the 2's complement of the datasum. This method is called a *2's complement checksum*.

As shown earlier, the basic checksum for the data block in Figure 9-7 is 59_{16} (01011001_2). The 1's complement checksum for the same data block is equal to the 1's complement of 59_{16}.

$$\text{1's complement of } 59_{16} = 10100110_2 = A6_{16}$$

The 2's complement checksum for the data block is equal to the 2's complement of 59_{16}.

$$\text{2s complement of } 59_{16} = 10100111_2 = A7_{16}$$

Example

Determine if the data block and accompanying checksum below are error free. The data block uses a 1's complement checksum.

Data						Checksum
06_{16}	00_{16}	$F7_{16}$	$7E_{16}$	01_{16}	52_{16}	31_{16}

Solution

First, we need to calculate the datasum. This is done by adding all of the data elements in the data block.

$$\begin{array}{r} 06_{16} \\ +\ 00_{16} \\ \hline 06_{16} \end{array} \quad \begin{array}{r} 06_{16} \\ +\ F7_{16} \\ \hline FD_{16} \end{array} \quad \begin{array}{r} FD_{16} \\ +\ 7E_{16} \\ \hline 17B_{16} \end{array} \quad \begin{array}{r} 7B_{16} \\ +\ 01_{16} \\ \hline 7C_{16} \end{array} \quad \begin{array}{r} 7C_{16} \\ +\ 52_{16} \\ \hline CE_{16} \end{array}$$

This gives us a datasum of CE_{16}. If we add this to 31_{16}, we get:

$$\begin{array}{r} CE_{16} \\ +\ 31_{16} \\ \hline FF_{16} \end{array}$$

That means that the data block is error free.

There is a second way to check this data. Instead of adding the datasum to the checksum, you can use the datasum to recalculate the checksum and compare the result with the received checksum. Taking the 1's complement of CE_{16} gives us:

$$CE_{16} = 11001110_2$$
$$\text{1's complement of } CE_{16} = 001100001_2 = 31_{16}$$

Example

Write a C program to determine the basic checksum, 1's complement checksum, and 2's complement checksum for the data block 07_{16}, 01_{16}, 20_{16}, 74_{16}, 65_{16}, 64_{16}, $2E_{16}$.

Solution

Before we get started on this code, it is important to know how to take a 1's complement and a 2's complement in C. The 1's complement uses a bitwise not operator '~'. By placing a '~' in front of a variable or constant, the bitwise inverse or 1's complement is returned. Since most computers represent negative numbers with 2's complement notation, the 2's complement is calculated by placing a negative sign in front of the variable or constant.

The code below begins by calculating the datasum. It does this with a loop that adds each value from the array of data values to a variable labeled *datasum*. After each addition, any potential carry is stripped off using a bitwise AND with 0xff. This returns the byte value.

Once the datasum is calculated, the three possible checksum values can be calculated. The first one is equal to the datasum, the second is equal to the bitwise inverse of the datasum, and the third is equal to the 2's complement of the datasum.

```
int datasum=0;
int block[] = {0x07, 0x01, 0x20, 0x74,
               0x65, 0x64, 0x2E};

// This for-loop adds all of the data elements
for(int i=0; i < sizeof(block)/sizeof(int); i++)
     datasum += block[i];

// The following line discards potential carries
datasum &= 0xff;
// Compute each of the three types of checksums
int basic_checksum = datasum;
int ones_compl_checksum = 0xff&(~datasum);
int twos_compl_checksum = 0xff&(-datasum);
```

If we execute this code with the appropriate output statements, we get the following three values for the checksums.

```
The basic checksum is 93
The 1's complement checksum is 6c
The 2's complement checksum is 6d
```

The next section presents a type of error checking used for long sequences of bits.

9.5 Cyclic Redundancy Check

The problem with using a checksum for error correction lies in its simplicity. If multiple errors occur in a data stream, it is possible that they may cancel each other out, e.g., a single bit error may subtract 4 from the checksum while a second error adds 4. If the width of the checksum character is 8 bits, then there are $2^8 = 256$ possible checksums for a data stream. This means that there is a 1 in 256 chance that multiple errors may not be detected. These odds could be reduced by increasing the size of the checksum to 16 or 32 bits thereby increasing the number of possible checksums to $2^{16} = 65{,}536$ or $2^{32} = 4{,}294{,}967{,}296$ respectively.

Assume Figure 9-8 represents a segment of an integer number line where the result of the checksum is identified. A minor error in one of the values may result in a small change in the checksum value. Since the erroneous checksum is not that far from the correct checksum, it is easy for a second error to put the erroneous checksum back to the correct value indicating that there hasn't been an error when there actually has been one.

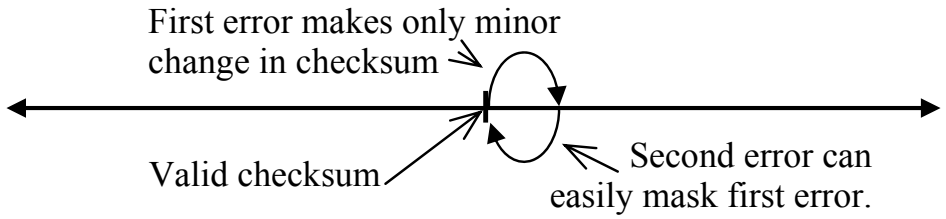


Figure 9-8 Small Changes in Data Canceling in Checksum

What we need is an error detection method that generates vastly different values for small errors in the data. The checksum algorithm doesn't do this which makes it possible for two bit changes to cancel each other in the sum.

A cyclic redundancy check (CRC) uses a basic binary algorithm where each bit of a data element modifies the checksum across its entire length regardless of the number of bits in the checksum. This

means that an error at the bit level modifies the checksum so significantly that an equal and opposite bit change in another data element cannot cancel the effect of the first.

First, calculation of the CRC checksum is based on the remainder resulting from a division rather than the result of an addition. For example, the two numbers below vary only by one bit.

$$0111\ 1010\ 1101\ 1100_2 = 31{,}452_{10}$$
$$0111\ 1011\ 1101\ 1100_2 = 31{,}708_{10}$$

The checksums at the nibble level are:

$$0111 + 1010 + 1101 + 1100 = 1010_2 = 10_{10}$$
$$0111 + 1011 + 1101 + 1100 = 1011_2 = 11_{10}$$

These two values are very similar, and a bit change from another nibble could easily cancel it out.

If, on the other hand, we use the remainder from a division for our checksum, we get a wildly different result for the two values. For the sake of an example, let's divide both values by 9_{10}.

$$31{,}452 \div 9 = 3{,}494 \text{ with a remainder of } 6 = 0110_2$$
$$31{,}708 \div 9 = 3{,}523 \text{ with a remainder of } 1 = 0001_2$$

This is not a robust example due to the fact that 4 bits only have 16 possible bit patterns, but the result is clear. A single bit change in one of the data elements resulted in a single bit change in the addition result. The same change, however, resulted in three bits changing in the division remainder.

The problem is that division in binary is not a quick operation. For example, Figure 9-9 shows the long division in binary of $31{,}452_{10} = 0111101011011100_2$ by $9_{10} = 1001_2$. The result is a quotient of $110110100110_2 = 3{,}494_{10}$ with a remainder of $110_2 = 6_{10}$.

Remember that the goal is to create a checksum that can be used to check for errors, not to come up with a mathematically correct result. Keeping this in mind, the time it takes to perform a long division can be reduced by removing the need for "borrows". This would be the same as doing an addition while ignoring the carries. The truth table in Table 9-2 shows the single bit results for both addition and subtraction when carries and borrows are ignored.

```
          110110100110
      ____________________
1001 | 0111101011011100
       -1001
        1100
       -1001
          1110
         -1001
           1011
          -1001
             1010
            -1001
                1111
               -1001
                 1100
                -1001
                   110
```

Figure 9-9 Example of Long Division in Binary

Table 9-2 Addition and Subtraction Without Carries or Borrows

A	B	A+B	A – B
0	0	0	0
0	1	1	1 (no borrow)
1	0	1	1
1	1	0 (no carry)	0

The A + B and A – B columns of the truth table in Table 9-2 should look familiar; they are equivalent to the XOR operation. This means that a borrow-less subtraction is nothing more than a bitwise XOR. Figure 9-10 shows an example of an addition and a subtraction where there is no borrowing. Note that an addition without carries produces the same result as a subtraction without borrows.

```
 11011010          11011010
+01101100          -01101100
 10110110          10110110
```

Figure 9-10 Examples of XOR Subtraction and Addition

There is a problem when trying to apply this form of subtraction to long division: an XOR subtraction doesn't care whether one number is larger than another. For example, 1111_2 could be subtracted from 0000_2 with no ill effect. In long division, you need to know how many digits to pull down from the dividend before subtracting the divisor.

To solve this, the assumption is made that one value can be considered "larger" than another if the bit position of its highest one is the same or greater than the bit position of the highest one in the second number. Figure 9-11 shows examples of subtractions that would be valid or invalid for long division using this criterion.

10110 – 10011	Valid	0111 – 0011	Valid
0110 – 1001	Invalid	01011 – 10000	Invalid

Figure 9-11 Valid and Invalid Borrow-less Subtractions

Figure 9-12 repeats the long division of Figure 9-9 using borrow-less subtractions. It is a coincidence that the resulting remainder is the same for the long division of Figure 9-9. This is not usually true.

```
          111010001010
      ┌──────────────
1001  │ 0111101011011100
        -1001
          1100
         -1001
           1011
          -1001
             1001
            -1001
                01011
               -1001
                  1010
                 -1001
                    110
```

Figure 9-12 Example of Long Division Using XOR Subtraction

Since addition and subtraction without carries or borrows are equivalent to a bitwise XOR, we should be able to reconstruct the original value from the quotient and the remainder using nothing but

XORs. Table 9-3 shows the step-by-step process of this reconstruction. The leftmost column of the table is the bit-by-bit values of the binary quotient of the division of Figure 9-12.

Starting with a value of zero, 1001_2 is XOR'ed with the result in the second column when the current bit of the quotient is a 1. The result is XOR'ed with 0000_2 if the current bit of the quotient is a 0. The rightmost column is the result of this XOR. Before going to the next bit of the quotient, the result is shifted left one bit position. Once the end of the quotient is reached, the remainder is added. This process brings back the dividend using a multiplication of the quotient and divisor.

Table 9-3 Reconstructing the Dividend Using XORs

Quotient (Q)	Result from previous step shifted left one bit	XOR Value Q=0: 0000 Q=1: 1001	XOR result
1	0	1001	1001
1	10010	1001	11011
1	110110	1001	111111
0	1111110	0000	1111110
1	11111100	1001	11110101
0	111101010	0000	111101010
0	1111010100	0000	1111010100
0	11110101000	0000	11110101000
1	111101010000	1001	111101011001
0	1111010110010	0000	1111010110010
1	11110101100100	1001	11110101101101
0	111101011011010	0000	111101011011010
Add remainder to restore the dividend: 111101011011010 + 110 = **111101011011100**			

Example

Perform the long division of 1100110110101011_2 by 1011_2 in binary using the borrow-less subtraction, i.e., XOR function.

Solution

Using the standard "long-division" procedure with the XOR subtractions, we divide 1011_2 into 1100110110101011_2. Table 9-4 checks our result using the technique shown in Table 9-3. Since we were able to recreate the original value from the quotient and remainder, the division must have been successful.

```
          1110101001111
      ┌──────────────────
1011  │ 1100110110101011
       -1011
        1111
        -1011
         1001
         -1011
           1001
           -1011
             1010
             -1011
               1101
               -1011
                 1100
                 -1011
                   1111
                   -1011
                     1001
                     -1011
                       010
```

Table 9-4 Second Example of Reconstructing the Dividend

Quotient (Q)	Result from previous step shifted left one bit	XOR Value Q=0: 0000 Q=1: 1011	XOR result
1	0	1011	1011
1	10110	1011	11101
1	111010	1011	110001
0	1100010	0000	1100010
1	11000100	1011	11001111
0	110011110	0000	110011110
1	1100111100	1011	1100110111
0	11001101110	0000	11001101110
0	110011011100	0000	110011011100
1	1100110111000	1011	1100110110011
1	11001101100110	1011	11001101101101
1	110011011011010	1011	110011011010001
1	1100110110100010	1011	1100110110101001
Add remainder to restore the dividend: 1100110110101001 + 010 = **1100110110101011**			

Note that we are reconstructing the original value from the quotient here in order to demonstrate the application of the XOR in this modified division and multiplication. This is not a part of the CRC implementation. In reality, as long as the sending and receiving devices use the same divisor, the only result of the division that is of concern is the remainder. As long as the sending and receiving devices obtain the same results, the transmission can be considered error free.

9.5.1 CRC Process

The primary difference between different CRC implementations is the selection of the divisor or polynomial as it is referred to in the industry. In the example used in this discussion, we used 1001_2, but this is by no means a standard value. Divisors of different bit patterns and different bit lengths perform differently. The typical divisor is 17 or 33 bits, but this is only because of the standard bit widths of 16 and 32 bits in today's processor architectures. A few divisors have been selected as performing better than others, but a discussion of why they perform better is beyond the scope of this text.

There is, however, a relationship between the remainder and the divisor that we do wish to discuss here. We made an assumption earlier in this section about how to decide whether one value is larger than another with regards to XOR subtraction. This made it so that in an XOR division, a subtraction from an intermediate value is possible only if the most significant one is in the same bit position as the most significant one of the divisor. This is true all the way up to the final subtraction which produces the remainder. These most significant ones cancel leaving a zero in the most significant bit position of each result including the remainder. Since the MSB is always a zero for the result of every subtraction in an XOR division, each intermediate result along with the final remainder must always be at least one bit shorter in length than the divisor.

There is another interesting fact about the XOR division that is a direct result of the borrow-less subtraction, and the standard method of CRC implementation has come to rely on this fact. Assume that we have selected an n-bit divisor. The typical CRC calculation begins by appending n-1 zeros to the end of the data (dividend). After we divide this new data stream by the divisor to compute the remainder, the remainder is added to the end of the new data stream effectively replacing the n-1 zeros with the value of the remainder.

Remember that XOR addition and subtraction are equivalent. Therefore, by adding the remainder to the end of the data stream, we

have effectively subtracted the remainder from the dividend. This means that when we divide the data stream (which has the remainder added/subtracted) by the same divisor, **the new remainder should equal zero**. Therefore, if the receiving device generates a remainder of zero after dividing the entire data stream with the polynomial, the transmission was error-free. The following example illustrates this.

Example

Generate the CRC checksum to be transmitted with the data stream 1011011010010110_2 using the divisor 11011_2.

Solution

Since the divisor is 5 bits long, append 5 – 1 = 4 zeros to the end of the data.

New data stream = "1011011010010110" + "0000"
= "10110110100101100000"

Finish by computing the CRC checksum using XOR division.

```
          1100001101110101
      ____________________
11011 | 10110110100101100000
       -11011
        11011
        -11011
             010100
             -11011
               11111
               -11011
                  10001
                  -11011
                    10101
                    -11011
                      11100
                      -11011
                         11100
                         -11011
                            11100
                            -11011
                              0111
```

The data stream sent to the receiving device becomes the original data stream with the 4-bit remainder appended to it.

Transmitted data stream = "1011011010010110" + "0111"
= "10110110100101100111"

If the receiving device divides this entire data stream by the divisor 11011_2, the remainder will be zero. This is shown in the long division below.

```
          1100001101110101
      ____________________
11011 | 10110110100101100111
       -11011
         11011
        -11011
             010100
            -11011
              11111
             -11011
                10001
               -11011
                 10101
                -11011
                  11100
                 -11011
                    11101
                   -11011
                      11011
                     -11011
                          0
```

If this process is followed, the receiving device will calculate a zero remainder any time there is no error in the data stream.

9.5.2 CRC Implementation

Up to now, the discussion has focused on the mathematics behind creating and using CRC checksums. As for the implementation of a CRC checksum, programmers use the following process:

- A single n-bit divisor is defined. Both the sending and receiving devices use the same n-bit divisor.

- The sending device adds n-1 zeros to the end of the data being sent, and then performs the XOR division in order to obtain the remainder. The quotient is thrown away.
- The sending device takes the original data (without the n–1 zeros) and appends the n–1 bit remainder to the end. This is equivalent to subtracting the remainder from the data without removing the n-1 appended zeros.
- The data with the appended remainder is sent to the receiving device.
- The receiving device performs an XOR division on the received message and its appended n–1 bit remainder using the same divisor.
- If the result of the receiver's XOR division is zero, the message can be considered error free. Otherwise, an error was incurred during transmission.

A number of CRC divisors or polynomials have been defined for standard implementations. For example, the CRC-CCITT divisor is the 17-bit polynomial 11021_{16} while the divisor used in IEEE 802.3 Ethernet is the 33-bit polynomial $104C11DB7_{16}$.

As for implementing the XOR division, most data streams are far too large to be contained in a single processor register. Therefore, the data stream must be passed through a register that acts like a window revealing only the portion of the stream where the XOR subtraction is being performed. This is the second benefit of using the bitwise XOR. Without the XOR subtraction, the whole dividend would need to be contained in a register in order to support the borrow function.

Remember that the MSB of both the intermediate value and the divisor in an XOR subtraction are always 1. This means that the MSB of the subtraction is unnecessary as it always result in a zero. Therefore, for an n-bit divisor or polynomial, only an n-1 bit register is needed for the XOR operation.

The code presented in Figure 9-13 appends four zeros to the end of a 32-bit data stream (*data_stream*), then performs an XOR division on it with the 5-bit polynomial 10111_2 (*poly*). The division is done in a division register (*division_register*). This division register in theory should only be four bits wide, but since there is no four bit integer type in C, an 8-bit char is used. After every modification of the division register, a bitwise AND is performed on it with the binary mask 1111_2 in order to strip off any ones that might appear above bit 3. The binary mask is labeled *division_mask*.

```
// This code generates a four-bit CRC from a 32 bit
// data stream by passing it through a four-bit
// division register where it is XOR'ed with the last
// four bits of a five bit polynomial
__int32 data_stream = 0x48376dea; // Data stream
#define poly 0x17                 // Polynomial=10111

// The XOR is performed in a char variable which will
// then be AND'ed with a 4-bit mask to clear the fifth
// bit. A mask allowing us to check for a fifth bit is
// also defined here.
char division_register = 0;
#define division_mask 0xf
#define division_MSB 0x10

// We will need to count how many times we've shifted
// the data stream so that we know when we are done.
// For a 32 bit stream, we need to shift 32+4 times.
int shift_count = 0;
#define shift_total (32+4)

__int32 temp_ds = data_stream;
while (shift_count < shift_total)
{
// The following code shifts bits into the division
// register from the data stream until a bit overflows
// past the length of the division register. Once this
// bit overflows, we know we have loaded a value from
// which the polynomial can be subtracted.
   while ((!(division_register & division_MSB))
      &&(shift_count < shift_total))
   {
      division_register <<= 1;
      if((temp_ds & 0x80000000) != 0)
         division_register+=1;
      temp_ds <<= 1;
      shift_count++;
   }
   division_register &= division_mask;
// If we have a value large enough to XOR with the
// polynomial, then we should do a bitwise XOR
   if(shift_count < shift_total)
      division_register ^= (poly & division_mask);
}
printf("The four-bit CRC for the 32 bit data stream
        0x%x using the polynomial 0x%x is 0x%x.\n",
        data_stream, poly, division_register);
```

Figure 9-13 Sample Code for Calculating CRC Checksums

Running this code with a 32-bit constant assigned to the variable *data_stream* will produce the four-bit CRC checksum 0010_2 for the polynomial 10111_2.

There are better ways to implement the CRC algorithm. This code is presented only to show how the division register might work.

9.6 Hamming Code

Errors can also occur in data stored in memory. One possibility is that a manufacturing defect or a hardware failure could cause a specific memory cell to be un-writable. Random errors might also be caused by an electrical event such as static electricity or electromagnetic interference that causes one or more bits to flip. Whatever the cause, we need to be able to determine if the data we are reading is the same as the data we wrote.

One solution might be to store an additional bit with each data byte. This bit could act as a parity bit making it so that the total number of ones stored in each memory location along with the corresponding parity bit is always even. When it is time to read the data, the number of ones in the in the data and the parity bit are counted. If an odd result occurs, we know that there was an error. As mentioned before, however, parity is not a very robust error checking method. If two errors occur, the parity still appears correct. In addition, it might be nice to detect ***and*** correct the error.

One way to do this is to use multiple parity bits, each bit responsible for the parity of a smaller, overlapping portion of the data. For example, we could use four parity bits to represent the parity of four different groupings of the four bits of a nibble. Table 9-5 shows how this might work for the four-bit value 1011_2. Each row of the table groups three of the four bits of the nibble along with a parity bit, P_n. The value shown for the parity bit makes the sum of all the ones in the grouping of three bits plus parity an even number.

Table 9-5 Data Groupings and Parity for the Nibble 1011_2

	Data Bits				Parity Bits			
	D_3=1	D_2=0	D_1=1	D_0=1	P_0	P_1	P_2	P_3
Group A	1	0	1		0			
Group B	1		1	1		1		
Group C	1	0		1			0	
Group D		0	1	1				0

In memory, the nibble would be stored with its parity bits in an eight-bit location as 10110100_2.

Now assume that the bit in the D_1 position which was originally a 1 is flipped to a 0 causing an error. The new value stored in memory would be 10010100_2. Table 9-6 duplicates the groupings of Table 9-5 with the new value for D_1. The table also identifies groups that incur a parity error with the data change.

Table 9-6 Data Groupings with a Data Bit in Error

	Data Bits				Parity Bits				Parity Result
	D_3=1	D_2=0	D_1=0	D_0=1	P_0	P_1	P_2	P_3	
Group A	1	0	0		0				Odd – Error
Group B	1		0	1		1			Odd – Error
Group C	1	0		1			0		Even – Okay
Group D		0	0	1				0	Odd – Error

Note that parity is now in error for groups A, C, and D. Since the D_1 position is the only bit that belongs to all three of these groups, then a processor checking for errors would not only know that an error had occurred, but also in which bit it had occurred. Since each bit can only take on one of two possible values, then we know that flipping the bit D_1 will return the nibble to its original data.

If an error occurs in a parity bit, i.e., if P_3 is flipped, then only one group will have an error. Therefore, when the processor checks the parity of the four groups, a single group with an error indicates that it is a parity bit that has been changed and the original data is still valid.

Table 9-7 Data Groupings with a Parity Bit in Error

	Data Bits				Parity Bits				Parity Result
	D_3=1	D_2=0	D_1=1	D_0=1	P_0	P_1	P_2	P_3	
Group A	1	0	1		0				Even – Okay
Group B	1		1	1		1			Even – Okay
Group C	1	0		1			0		Even – Okay
Group D		0	1	1				1	Odd – Error

It turns out that not all four data groupings are needed. If we only use groups A, B, and C, we still have the same level of error detection, but we do it with one less parity bit. Continuing our example without

Group D, if our data is error-free or if a single bit error has occurred, one of the following eight situations is true.

Table 9-8 Identifying Errors in a Nibble with Three Parity Bits

Groups with bad parity	Bit in error
None	Error-free
A	P_0
B	P_1
C	P_2
A and B	D_1
A and C	D_2
B and C	D_0
A, B, and C	D_3

This use of multiple parity bits to "code" an error correction scheme for multiple bits is called the Hamming Code. It was developed by Richard Hamming during the late 1940's while he worked at Bell Laboratories.

The Hamming Code can be shown graphically using a Venn diagram. We begin by creating three overlapping circles, one circle for each group. Each of the parity bits P_n is placed in the portion of their corresponding circle that is not overlapped by any other circle. D_0 is placed in the portion of the diagram where circles B and C overlap, D_1 goes where circles A and B overlap, and D_2 goes where circles A and C overlap. Place D_3 in the portion of the diagram where all three circles overlap. Figure 9-14 presents just such an arrangement.

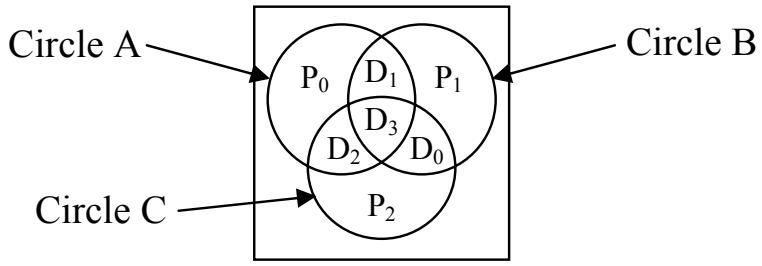
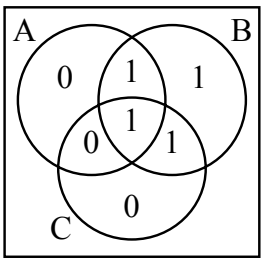
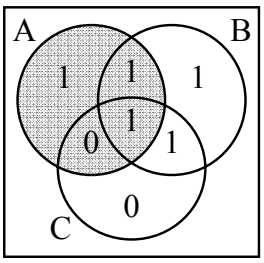


Figure 9-14 Venn Diagram Representation of Hamming Code

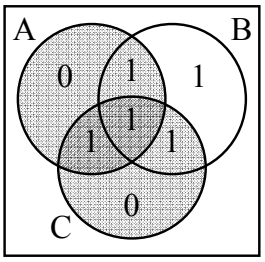
Figure 9-15a uses this arrangement to insert the nibble 1011_2 into a Venn diagram. Figures 9-15b, c, and d show three of the seven possible error conditions.



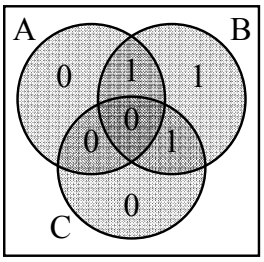
a.) Error-free condition



b.) Parity error in circle A



c.) Parity errors in A & C



d.) Parity errors in A, B, & C

Figure 9-15 Example Single-Bit Errors in Venn Diagram

In 9-15b, a single error in circle A indicates only the parity bit P_0 is in error. In 9-15c, since both circles A and C have errors, then the bit change must have occurred in the region occupied only by A and C, i.e., where D_2 is located. Therefore, D_2 should be 0. Lastly, in 9-15d, an error in all three circles indicates that there has been a bit change in the region shared by all three circles, i.e., in bit D_3. Therefore, we know that bit D_3 is in error. Each of these errors can be corrected by inverting the value of the bit found in error.

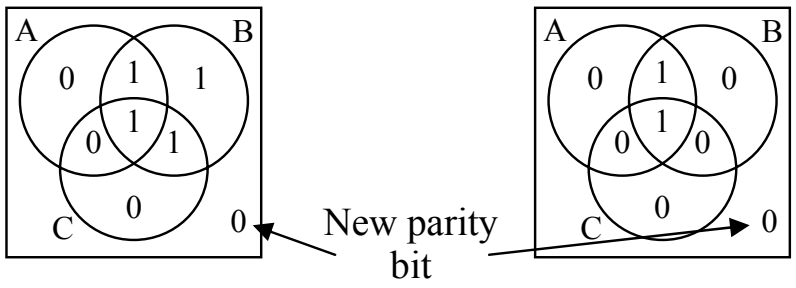
Two errors, however, cannot be detected correctly with this method. In Figure 9-16b, both the parity bit P_1 and the data bit D_0 are in error. If we do a parity check on each of the three circles in this Venn diagram, we find erroneous parity only in circle C. This would indicate that only the parity bit P_2 is in error. This is a problem because it incorrectly assumes the data 1010_2 is correct.

This is a problem. Apparently, this error checking scheme can detect when a double-bit error occurs, but if we try to correct it, we end up with erroneous data. We need to expand our error detection scheme to be able to detect and correct single bit errors and distinguish them from double bit errors.

a.) Error-free condition b.) Two-Bit Error Condition

Figure 9-16 Example of a Two-Bit Error

This can be done by adding one more bit that acts as a parity check for all seven data and parity bits. Figure 9-17 represents this new bit using the same example from Figure 9-16.



a.) Error-free condition b.) Two-Bit Error Condition

Figure 9-17 Using Parity to Check for Double-Bit Errors

If a single-bit error occurs, then after we go through the process of correcting the error, this new parity bit will be correct. If, however, after we go through the process of correcting the error and the new parity bit is in error, then it can be assumed that a double-bit error has occurred and that correction is not possible. This is called ***Single-Error Correction/Doubled-Error Detection***.

This error detection and correction scheme can be expanded to any number of bits. All we need to do is make sure there are enough parity bits to cover the error-free condition plus any possible single-bit error in the data or the parity. For example, in our four data bit and three parity bit example above, there can be one of seven single bit errors. Add the error-free condition and that makes eight possible conditions that must be represented with parity bits. Since there are three parity

bits, then there are $2^3 = 8$ possible bit patterns represented using the parity bits, one for each of the outcomes.

For the general case, we see that p parity bits can uniquely identify $2^p - 1$ single-bit errors. Note that the one is subtracted from 2^p to account for the condition where there are no errors. If $2^p - 1$ is less than the number of data bits, n, plus the number of parity bits, p, then we don't have enough parity bits. This relationship is represented with equation 9-1.

$$p + n \leq 2^p - 1 \qquad (9.1)$$

Table 9-9 presents a short list of the number of parity bits that are required for a specific number of data bits.

Table 9-9 Parity Bits Required for a Specific Number of Data Bits

Number of data bits (n)	Number of parity bits (p)	$p + n$	$2^p - 1$
4	3	7	7
8	4	12	15
16	5	21	31
32	6	38	63
64	7	71	127
128	8	136	255

To detect double-bit errors, an additional bit is needed to check the parity of all of the $p + n$ bits.

Let's develop the error-checking scheme for 8 data bits with 4 parity bits. Remember from the four-bit example that there were three parity checks:

- P_0 was the parity bit for data bits for D_1, D_2, and D_3;
- P_1 was the parity bit for data bits for D_0, D_1, and D_3; and
- P_2 was the parity bit for data bits for D_0, D_2, and D_3.

In order to check for a bit error, the sum of ones for each of these groups is taken. If all three sums result in even values, then the data is error-free. The implementation of a parity check is done with the XOR function. Remember that the XOR function counts the number of ones at the input and outputs a 1 for an odd count and a 0 for an even count.

This means that the three parity checks we use to verify our four data bits can be performed using the XOR function. Equations 9.2, 9.3, and 9.4 show how these three parity checks can be done. The XOR is represented here with the symbol ⊕.

$$\text{Parity check for group A} = P_0 \oplus D_1 \oplus D_2 \oplus D_3 \quad (9.2)$$
$$\text{Parity check for group B} = P_1 \oplus D_0 \oplus D_1 \oplus D_3 \quad (9.3)$$
$$\text{Parity check for group C} = P_2 \oplus D_0 \oplus D_2 \oplus D_3 \quad (9.4)$$

The data bits of our four-bit example were $D_3 = 1$, $D_2 = 0$, $D_1 = 1$, and $D_0 = 1$ while the parity bits were $P_0 = 0$, $P_1 = 1$, and $P_2 = 0$. Substituting these into equations 9.2, 9.3, and 9.4 gives us:

$$\text{Parity check for group A} = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$
$$\text{Parity check for group B} = 1 \oplus 1 \oplus 1 \oplus 1 = 0$$
$$\text{Parity check for group C} = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

Assume that a single-bit error has occurred. If the single-bit error was a parity bit, then exactly one of the parity checks will be one while the others are zero. For example, if P_0 changed from a 0 to a 1, we would get the following parity checks.

$$\text{Parity check for group A} = 1 \oplus 1 \oplus 0 \oplus 1 = 1$$
$$\text{Parity check for group B} = 1 \oplus 1 \oplus 1 \oplus 1 = 0$$
$$\text{Parity check for group C} = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

The single parity bit error reveals itself as a single parity check outputting a 1. If, however, a data bit changed, then we have more than one parity check resulting in a 1. Assume, for example, that D_1 changed from a 1 to a 0.

$$\text{Parity check for group A} = 0 \oplus 0 \oplus 0 \oplus 1 = 1$$
$$\text{Parity check for group B} = 1 \oplus 1 \oplus 0 \oplus 1 = 1$$
$$\text{Parity check for group C} = 0 \oplus 1 \oplus 0 \oplus 1 = 0$$

Since D_1 is the only bit that belongs to both the parity check of groups A and B, then D_1 must have been the one to have changed.

Using this information, we can go to the eight data bit example. With four parity bits, we know that there will be four parity check equations, each of which will have a parity bit that is unique to it.

Parity check A = $P_0 \oplus$ (XOR of data bits of group A)
Parity check B = $P_1 \oplus$ (XOR of data bits of group B)
Parity check C = $P_2 \oplus$ (XOR of data bits of group C)
Parity check D = $P_3 \oplus$ (XOR of data bits of group D)

The next step is to figure out which data bits, D_0 through D_7, belong to which groups. Each data bit must have a unique membership pattern so that if the bit changes, its parity check will result in a unique bit pattern. Note that all of the data bits must belong to at least two groups. This is because if there is an error with a bit that belongs to only one group, it will look like a parity bit error since only one parity check will output a 1.

Table 9-10 shows one way to group the bits in the different parity check equations or groups. It is not the only way to group them.

Table 9-10 Membership of Data and Parity Bits in Parity Groups

	Parity check group			
	A	B	C	D
P_0	×			
P_1		×		
P_2			×	
P_3				×
D_0	×	×		
D_1	×		×	
D_2		×	×	
D_3	×	×	×	
D_4	×			×
D_5		×		×
D_6	×	×		×
D_7			×	×

By using the grouping presented in Table 9-10, we can complete our four parity check equations.

$$\text{Parity check A} = P_0 \oplus D_0 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6 \quad (9.5)$$
$$\text{Parity check B} = P_1 \oplus D_0 \oplus D_2 \oplus D_3 \oplus D_5 \oplus D_6 \quad (9.6)$$
$$\text{Parity check C} = P_2 \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_7 \quad (9.7)$$
$$\text{Parity check D} = P_3 \oplus D_4 \oplus D_5 \oplus D_6 \oplus D_7 \quad (9.8)$$

When it comes time to store the data, we will need 12 bits, eight for the data and four for the parity bits. But how do we calculate the parity bits? Remember that the parity check must always equal zero. Therefore, the sum of the data bits of each parity group with the parity bit must be an even number. Therefore, if the sum of the data bits by themselves is an odd number, the parity bit must equal a 1, and if the sum of the data bits by themselves is an even number, the parity bit must equal a 0. This sounds just like the XOR function again. Therefore, we use equations 9.9, 9.10, 9.11, and 9.12 to calculate the parity bits before storing them.

$$P_0 = D_0 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6 \quad (9.9)$$
$$P_1 = D_0 \oplus D_2 \oplus D_3 \oplus D_5 \oplus D_6 \quad (9.10)$$
$$P_2 = D_1 \oplus D_2 \oplus D_3 \oplus D_7 \quad (9.11)$$
$$P_3 = D_4 \oplus D_5 \oplus D_6 \oplus D_7 \quad (9.12)$$

Now let's test the system. Assume we need to store the data 10011100_2. This gives us the following values for our data bits:

$$D_7 = 1 \quad D_6 = 0 \quad D_5 = 0 \quad D_4 = 1 \quad D_3 = 1 \quad D_2 = 1 \quad D_1 = 0 \quad D_0 = 0$$

The first step is to calculate our parity bits. Using equations 9.9, 9.10, 9.11, and 9.12 we get the following values.

$$P_0 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$P_1 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 0$$
$$P_2 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$$
$$P_3 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$

Once again, the XOR is really just a parity check. Therefore, if there is an odd number of ones, the result is 1 and if there is an even number of ones, the result is 0.

Now that the parity bits have been calculated, the data and parity bits can be stored. This means that memory will contain the following value:

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	P_0	P_1	P_2	P_3
1	0	0	1	1	1	0	0	0	0	1	0

If our data is error free, then when we read it and substitute the values for the data and parity bits into our parity check equations, all four results should equal zero.

Parity check A = $0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 0$
Parity check B = $0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 = 0$
Parity check C = $1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0$
Parity check D = $0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$

If, however, while the data was stored in memory, it incurs a single-bit error, e.g., bit D_6 flips from a 0 to a 1, then we should be able to detect it. If D_6 does flip, the value shown below is what will be read from memory, and until the processor checks the parity, we don't know that anything is wrong with it.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	P_0	P_1	P_2	P_3
1	1	0	1	1	1	0	0	0	0	1	0

Start by substituting the values for the data and parity bits read from memory into our parity check equations. Computing the parity for all four groups shows that an error has occurred.

Parity check A = $0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$
Parity check B = $0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$
Parity check C = $1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0$
Parity check D = $0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1$

Since we see from Table 9-10 that the only bit that belongs to parity check groups A, B, and D is D_6, then we know that D_6 has flipped and we need to invert it to return to our original value.

The same problem appears here as it did in the nibble case if there are two bit errors. It is solved here the same way as it was for the nibble application. By adding a parity bit representing the parity of all twelve

data and parity bits, then if one of the group parities is wrong but the overall parity is correct, we know that a double-bit error has occurred and correction is not possible.

9.7 What's Next?

In this chapter we've discussed how to correct errors that might occur in memory without having discussed the technologies used to store data. Chapter 10 begins our discussion of storing data by examining the memory cell, a logic element capable of storing a single bit of data.

Problems

1. Using an original value of 11000011_2 and a mask of 00001111_2, calculate the results of a bitwise AND, a bitwise OR, and a bitwise XOR for these values.

2. Assume that the "idiot lights" of an automotive dashboard are controlled with an eight-bit binary value named *dash_lights*. The table below describes the function of each bit. Assume that a '1' turns on the light corresponding to that bit position and a '0' turns it off.

D_0	Low fuel	D_4	Left turn signal
D_1	Oil pressure	D_5	Right turn signal
D_2	High temperature	D_6	Brake light
D_3	Check engine	D_7	Door open

   For each of the following situations, write the line of code that uses a bitwise operation to get the desired outcome.

   a.) Turn on the low fuel, oil pressure, high temperature, check engine, and brake lights without affecting any other lights. This operation would be done when the ignition key is turned to start.
   b.) Toggle both the right and left turn signals as if the flashers were on without affecting any other lights.
   c.) Turn off the door open light when the door is closed.

3. True or False: A checksum changes if the data within the data block is sorted differently.

4. Compute the basic checksum, the 1's complement checksum, and the 2's complement checksum for each of the following groups of 8-bit data elements using both the basic calculation of the datasum and the one's complement datasum. All of the data is represented in hexadecimal.

   a.) 34, 9A, FC, 28, 74, 45
   b.) 88, 65, 8A, FC, AC, 23, DC, 63
   c.) 00, 34, 54, 23, 5C, F8, F1, 3A, 34

5. Use the checksum to verify each of the following groups of 8-bit data elements. All of the data is represented in hexadecimal.

   a.) 54, 47, 82, CF, A9, 43 basic checksum = D8
   b.) 36, CD, 32, CA, CF, A8, 56, 88 basic checksum = 55
   c.) 43, A3, 1F, 8F, C5, 45, 43 basic checksum = E1

6. Identify the two reasons for using the XOR "borrow-less" subtraction in the long-division used to calculate the CRC.

7. What problem does the checksum error correction method have that is solved by using CRCs?

8. True or False: A CRC changes if the data within the data block is sorted differently.

9. True or False: By using the CRC process where the transmitting device appends the remainder to the end of the data stream, the remainder calculated by the receiving device should be zero.

10. How many possible CRC values (i.e., remainders) are possible with a 33-bit polynomial?

11. Assume each of the following streams of bits is received by a device and that each of the streams has appended to it a CRC checksum. Using the polynomial 10111, check to see which of the data streams are error free and which are not.

    a.) 10010111010110010 01
    b.) 10110101001010100100101
    c.) 11010110101010110111011011011

12. Compute the number of parity bits needed to provide single-bit error correction for 256 bits of data.

13. Using the error detection/correction equations 9.5 through 9.8, determine the single-bit error that would result from the following parity check values.

	Results of parity check			
	a.)	b.)	c.)	d.)
Parity check A	1	1	0	1
Parity check B	1	1	0	0
Parity check C	1	0	1	1
Parity check D	0	1	0	0

14. Using the programming language of your choice, implement the parity generating function of the single-bit error correction scheme for eight data bits discussed in this chapter. Use equations 9.9 through 9.12 to generate the parity bits. You may use the C prototype shown below as a starting point where the integer *data* represents the 8 data bits and the returned value will contain the four parity bits in the least significant four positions in the order P_0, P_1, P_2, and P_3.

```
int generateParityBits (int data)
```

15. Using the programming language of your choice, implement the parity checking function of the single-bit error correction scheme for eight data bits discussed in this chapter. Use equations 9.5 through 9.8 to verify the data read from memory. You may use the C prototype shown below as a starting point where the integer *data* represents the 8 data bits and the integer *parity* represents the four parity bits in the least significant four positions in the order P_0, P_1, P_2, and P_3. The returned value is the data, unmodified if no error was detected or corrected if a single-bit error was detected.

```
int generateCorrectedData (int data, parity);
```

16. Determine the set of parity check equations for eight data bits and four parity bits if we use the table to the right in place of the memberships defined by Table 9-10.

	Parity check group			
	A	B	C	D
P_0	×			
P_1		×		
P_2			×	
P_3				×
D_0	×	×		
D_1		×	×	
D_2	×		×	
D_3	×			×
D_4		×		×
D_5	×	×		×
D_6			×	×
D_7	×		×	×

17. Identify the error in the parity check equations below. Note that the expressions are supposed to represent a different grouping than those in equations 9.2, 9.3, and 9.4. There is still an error though with these new groupings.

Parity check for group A = $P_0 \oplus D_0 \oplus D_2 \oplus D_3$
Parity check for group B = $P_1 \oplus D_0 \oplus D_1$
Parity check for group C = $P_2 \oplus D_1 \oplus D_2$