9 ENCODER CIRCUITS

What to Expect

Encoders and decoders are used to change a coded byte from one form to another. For example, a binary-to-BCD encoder changes a binary byte to its BCD equivalent. Encoders and decoders are very common in digital circuits and are used, for example, to change a binary number to a visual display that uses an Light Emitting Diode (LED). The following topics are included in this chapter.

- Developing circuits the use multiplexers and demultiplexers
- Creating a minterm generator using a multiplexers
- Creating a ten-line priority encoder
- Using a seven-segment display for a decoded binary number
- Employing a decoder as a function generator
- Explaining the theory and process of error detection and correction
- Detecting errors in a transmitted byte using the Hamming Code

9.1 MULTIPLEXERS/DEMULTIPLEXERS

9.1.1 *Multiplexer*

A multiplexer is usually called a "mux" and a demultiplexer is called a "dmux."

A multiplexer is used to connect one of several input lines to a single output line. Thus, it selects which input to pass to the output. This function is similar to a rotary switch where several potential inputs to the switch can be sent to a single output. A demultiplexer is a multiplexer in reverse, so a single input can be routed to any of several outputs. While mux/dmux circuits were originally built for transmission systems (like using a single copper wire to carry several different telephone calls simultaneously), today they are used

as "decision-makers" in virtually every digital logic system and are, therefore, one of the most important devices for circuit designers.

To help clarify this concept, Figure 9.1 is a simple schematic diagram that shows two rotary switches set up as a mux/dmux pair. As the switches are set in the diagram, a signal would travel from INPUT B to OUTPUT 2 through a connecting wire.

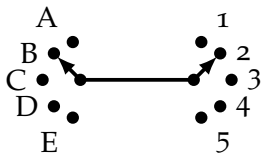

Figure 9.1: Multiplexer Using Rotary Switches

Imagine that the switches could somehow be synchronized so they rotated among the setting together; that is, INPUT A would always connect to OUTPUT 1 and so forth. That would mean a single wire could carry five different signals. For example, imagine that the inputs were connected to five different intrusion sensors in a building and the five outputs were connected to lamps on a guard's console in a remote building. If something triggered sensor *A* then as soon as the mux/dmux pair rotated to that position it would light lamp one on the console. Carrying all of these signals on a single wire saves a lot of expense. Of course, a true alarm system would be more complex than this, but this example is only designed to illustrate how a mux/dmux pair works in a transmission system.

Figure 9.2 is the logic diagram for a simple one-bit two-to-one multiplexer. In this circuit, an input is applied to input ports *A* and *B*. Port *Sel* is the selector and if that signal is zero then Port *A* will be routed to output *Y*; if, though, *Sel* is one then Port *B* will be routed to *Y*.

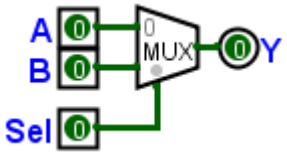

Figure 9.2: Simple Mux

Truth Table 9.1, below, is for a multiplexer:

Inputs			Output
A	B	Sel	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 9.1: Truth Table for a Multiplexer

In this multiplexer, the data input ports are only a single bit wide; however, in a normal circuit those ports could be a full 32-bit or 64-bit word and the selected word would be passed to the output port. Moreover, a multiplexer can have more than two input ports so a very versatile switch can be built to handle switching full words from one of eight or even sixteen different inputs. Because of its ability to channel a selected data stream to a single bus line from many different sub-circuits, the multiplexer is one of the workhorses for digital logic circuits and is frequently found in complex devices like CPUs.

9.1.2 *Demultiplexer*

A demultiplexer is the functional opposite of a multiplexer: a single input is routed to one of several potential outputs. Figure 9.3 is the logic diagram for a one-bit one-to-two demultiplexer. In this circuit, an input is applied to input port *A*. Port *Sel* is a control signal and if that signal is zero then input *A* will be routed to output *Y*, but if the control signal is one then input *A* will be routed to output *Z*.

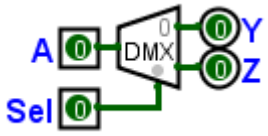

Figure 9.3: Simple Dmux

Truth Table 9.2, below, is for a demultiplexer.

Inputs		Outputs	
A	Sel	Y	Z
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Table 9.2: Truth Table for a Demultiplexer

In this demultiplexer the data input port is only a single bit wide; however, in a normal circuit that port could be a full 32-bit or 64-bit word and that entire word would be passed to the selected output port. Moreover, a demultiplexer can have more than two outputs so a very versatile switch can be built to handle switching full words to one of eight or even sixteen different outputs. Because of its ability to switch a data stream to different sub-circuits, the demultiplexer is one of the workhorses for digital logic circuits and is frequently found in complex devices like CPUs.

9.1.3 *Minterm Generators*

Demultiplexers can be combined with an OR gate and be used as a minterm generator. Consider the circuit for this two-variable equation.

$$\int(A, B) = \sum(1, 2) \tag{9.1}$$

Since there are two input variables, A and B, the dmux needs to have two select bits, one for each variable, and that would generate four potential dmux outputs W, X, Y, and Z. This circuit could be constructed using a four-output dmux with a two-bit control signal.

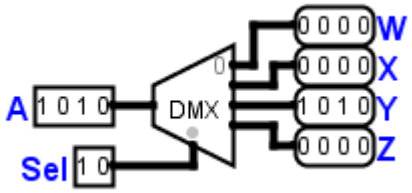

Figure 9.4: 1-to-4 Dmux

In Figure 9.4, a four-bit input (A) is routed to one of four output ports: W, X, Y, or Z, depending on the setting of the select, *Sel*. Figure 9.4 shows the data input of 1010 being routed to Y by a select of 10.

However, the equation specifies that the only outputs that would be used are when *Sel* is 01 or 10. Thus, output ports X and Y must be

sent to an OR gate and the other two outputs ignored. The output of the OR gate would only activate when *Sel* is set to 01 or 10, as shown in Figure 9.5.

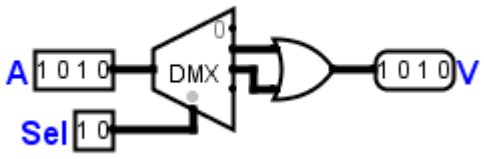

Figure 9.5: 1-to-4 Dmux As Minterm Generator

9.2 ENCODERS/DECODERS

9.2.1 *Introduction*

Encoders and decoders are used to convert some sort of coded data into a different code. For example, it may be necessary to convert the code created by a keyboard into ASCII for use in a word processor. By definition, the difference between an encoder and a decoder is the number of inputs and outputs: Encoders have more inputs than outputs, while decoders have more outputs than inputs.

As an introduction to encoders, consider Figure 9.6, which is designed to encode three single line inputs (maybe three different push buttons on a control panel) into a binary number for further processing by the computer. In this circuit, the junction between the two OR gates and the output (Y) is a *joiner* that combines two bit streams into a single bus. Physically, two wires (one from *U1* and one from *U2*) would be spliced together into a single cable (a *bus*) that contains two strands. The figure shows that when input *C* is active the output of the encoder is 11.

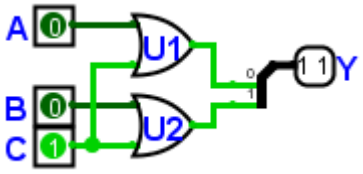

Figure 9.6: Three-line to 2-Bit Encoder

As an introduction to decoders, consider Figure 9.7, which is designed to decode a two-bit binary input and drive a single output line high. A circuit like this may be used to light an LED used as a warning on a console if a particular binary code is generated elsewhere in a circuit. In this circuit, the input is a two-bit number (10 in the illustration), but those two bits are separated through a splitter and each is applied to one of the inputs of a series of four AND gates. Imagine

that the MSB was placed on the wire on the left of the grid and wired to the bottom input of each of the AND gates. If $A = 10$, then AND gate three would activate and output X would go high.

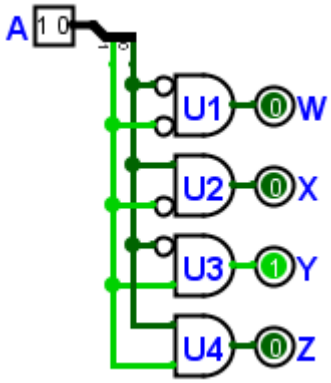

Figure 9.7: Four-Bit to 4-Line Decoder

Both encoders and decoders are quite common and are used in many electronic devices. However, it is not very common to build these circuits out of discrete components (like in the circuits above). Rather, inexpensive integrated circuits are available for most encoder/decoder operations and these are much easier, and more reliable, to use.

9.2.2 *Ten-Line Priority*

This encoder is sometimes called Ten-Line to Four-Line.

Consider a ten-key keypad containing the numbers zero through nine, like a keypad that could be used for numeric input from some hand-held device. In order to be useful, a key press would need to be encoded to a binary number for further processing by a logic circuit.

The keypad outputs a nine-bit number such that a single bit goes high to indicate which key was. For example, when key number two is pressed, 0_0000_0010 is output from the device. A priority encoder would accept that nine-bit number and output a binary number that could be used in computer circuits. Truth Table 9.3 is for the Priority Encoder that meets the specification. The device is called a "priority" encoder since it will respond to only the highest value key press. For example, if someone pressed the three and five keys simultaneously the encoder would ignore the three key press and transmit 0101, or binary five.

Dashes are normally used to indicate "don't care" on a Truth Table.

Inputs									Outputs			
9	8	7	6	5	4	3	2	1	Y1	Y2	Y3	Y4
0	0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	1	-	0	0	1	0
0	0	0	0	0	0	1	-	-	0	0	1	1
0	0	0	0	0	1	-	-	-	0	1	0	0
0	0	0	0	1	-	-	-	-	0	1	0	1
0	0	0	1	-	-	-	-	-	0	1	1	0
0	0	1	-	-	-	-	-	-	0	1	1	1
0	1	-	-	-	-	-	-	-	1	0	0	0
1	-	-	-	-	-	-	-	-	1	0	0	1

Table 9.3: Truth Table for Priority Encoder

This circuit can be realized by using a grid input and routing the various lines to an appropriate *AND* gate. This is one of the circuits built in the lab manual that accompanies this text.

9.2.3 *Seven-Segment Display*

A seven-segment display is commonly used in calculators and other devices to show hexadecimal numbers. To create the numeric shapes, various segments are activated while others remain off, so binary numbers must be decoded to turn on the various segments for any given combination of inputs. A seven-segment display has eight input ports and, when high, each of those ports will activate one segment of the display.

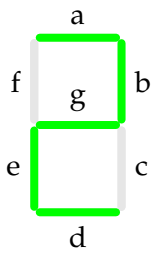

Figure 9.8: Seven-Segment Display

Usually an eighth "segment" is available —a decimal point in the lower right corner.

In Figure 9.8 the seven segments are labeled and it shows that the number "2" for example, is made by activating segments a, b, g, e, and d. Table 9.4 shows the various segments that must be activated for each of the 16 possible input values.

Hex	—	Binary				—	Display						
	—	**3**	**2**	**1**	**0**	—	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	—	0	0	0	0	—	1	1	1	1	1	1	0
1	—	0	0	0	1	—	0	1	1	0	0	0	0
2	—	**0**	**0**	**1**	**0**	—	**1**	**1**	**0**	**1**	**1**	**0**	**1**
3	—	0	0	1	1	—	1	1	1	1	0	0	1
4	—	0	1	0	0	—	0	1	1	0	0	1	1
5	—	0	1	0	1	—	1	0	1	1	0	1	1
6	—	0	1	1	0	—	1	0	1	1	1	1	1
7	—	0	1	1	1	—	1	1	1	0	0	0	0
8	—	1	0	0	0	—	1	1	1	1	1	1	1
9	—	1	0	0	1	—	1	1	1	1	0	1	1
A	—	1	0	1	0	—	1	1	1	0	1	1	1
B	—	1	0	1	1	—	0	0	1	1	1	1	1
C	—	1	1	0	0	—	1	0	0	1	1	1	0
D	—	1	1	0	1	—	0	1	1	1	1	0	1
E	—	1	1	1	0	—	1	0	0	1	1	1	1
F	—	1	1	1	1	—	1	0	0	0	1	1	1

Table 9.4: Truth Table for Seven-Segment Display

Notice that to display the number "2" (in bold font), segments a, b, d, e, and g must be activated.

A decoder circuit, as in Figure 9.9, uses a demultiplexer to activate a one-bit line based on the value of the binary input. Note: to save space, two parallel decoders are used in this circuit.

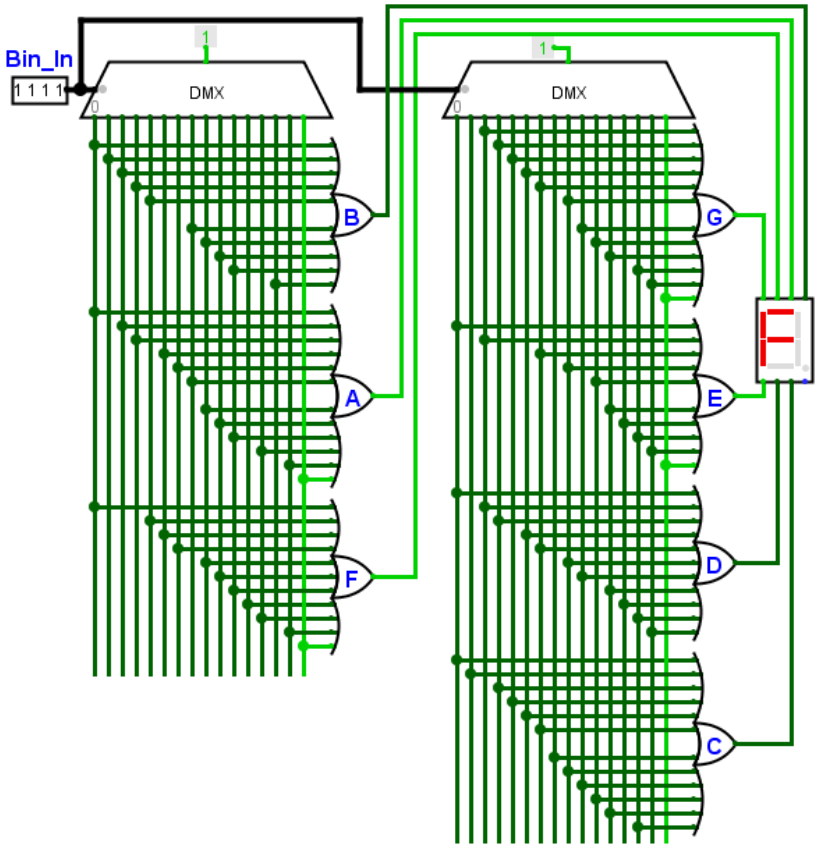

Figure 9.9: 7-Segment Decoder

Figure 9.9 shows an input of 1111_2 so the last line on each demultiplexer is activated. Those lines are used to activate the necessary inputs on the seven-segment display to create an output of "F".

A Hex Digit Display has a single port that accepts a four-bit binary number and that number is decoded into a digital display. Figure 9.10 shows a hex digit display.

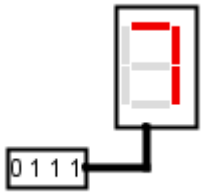

Figure 9.10: Hex Decoder

The *Logisim-evolution* simulator used in this class includes both a *7-Segment Display* and a *Hex Digit Display*. One of the strengths of using a seven-segment display rather than a hex digit display is that the circuit designer has total control over which segments are displayed. It is common, for example, to activate each of the outer segments in

a rapid sequence to give the illusion of a rotating circle. As opposed to a seven-segment display, a hex digit display is very simple to wire and use, as Figures 9.9 and 9.10 make clear. Both of these two types of displays are available on the market and a designer would chose whichever type meets the needs.

9.2.4 *Function Generators*

Decoders provide an easy way to create a circuit when given a minterm function. Imagine that a circuit is needed for the function defined in Equation 9.2.

$$\int(A, B, C) = \sum(0, 2, 7) \tag{9.2}$$

Whatever this circuit is designed to do, it should activate an output only when the input is zero, two, or seven. The circuit in Figure 9.11 illustrates a simple minterm generator using a demultiplexer and an OR gate. When input A is zero, two, or seven then output Y will go high, otherwise it is low.

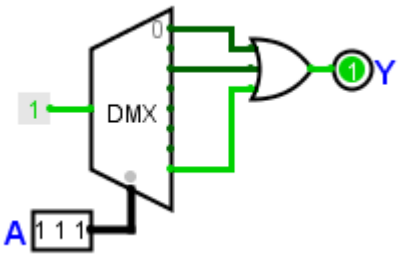

Figure 9.11: Minterm Generator

9.3 ERROR DETECTION

9.3.1 *Introduction*

Whenever a byte (or any other group of bits) is transmitted or stored, there is always the possibility that one or more bits will be accidentally complemented. Consider these two binary numbers:

0110 1010 0011 1010
0110 1010 0111 1010

They differ by only one bit (notice Group Three). If the top number is what is supposed to be in a memory location but the bottom number is what is actually there then this would, obviously, create a problem. There could be any number of reasons why a bit would be wrong,

but the most common is some sort of error that creeps in while the byte is being transmitted between two stores, like between a Universal Synchronous Bus (USB) drive and memory or between two computers sharing a network. It is desirable to be able to detect that a byte contains a bad bit and, ideally, even know which bit is wrong so it can be corrected.

Parity is a common method used to check data for errors and it can be used to check data that has been transmitted, held in memory, or stored on a hard drive. The concept of parity is fairly simple: A bit (called the *parity bit*) is added to each data byte and that extra bit is either set to zero or one in order to make the bit-count of that byte contain an even or odd number of ones. For example, consider this binary number:

1101

There are three ones in this number, which is an odd number. If odd parity is being used in this circuit, then the parity bit would be zero so there would be an odd number of ones in the number. However, if the circuit is using even parity, then the parity bit would be set to one in order to have four ones in the number, which is an even number. Following is the above number with both even and odd parity bits (those parity bits are in the least significant position and are separated from the original number by a space for clarity):

1101 0 (Odd Parity)
1101 1 (Even Parity)

Table 9.5 shows several examples that may help to clarify this concept. In each case, a parity bit is used to make the data byte even parity (spaces were left in the data byte for clarity).

Data Byte	Parity Bit
0000 0000	0
0000 0001	1
0000 0011	0
0000 0100	1
1111 1110	1
1111 1111	0

Table 9.5: Even Parity Examples

Generating a parity bit can be done with a series of cascading *XOR* gates but *Logisim-evolution* had two parity gates, one that outputs high when the inputs have an odd number of ones and the other when there are an even number of ones. Figure 9.12 illustrates using an odd

parity gate (labeled "2K+1"). In this circuit, if input *A* has an odd number of ones, as illustrated, then the parity generator will output a one to indicate input *A* has an odd number of ones. That parity bit is added as the most significant bit to output *Y*. Since output *Y* will always have an even number of bits this is an even parity circuit.

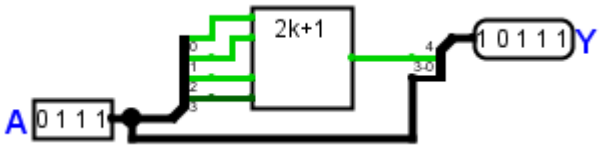

Figure 9.12: Parity Generator

Parity is a simple concept and is the foundation for one of the most basic methods of error checking. As an example, if some byte is transmitted using even parity but the data arrives with an odd number of ones then one of the bits was changed during transmission.

9.3.2 *Iterative Parity Checking*

One of the problems with using parity for error detection is that while it may be known that *something* is wrong, there is no way to know which of the bits is wrong. For example, imagine an eight-bit system is using even parity and receives this data and parity bit:

1001 1110 PARITY: 0

There is something wrong with the byte. It is indicating even parity but has an odd number of ones in the byte. It is impossible to know which bit changed during transmission. In fact, it may be that the byte is correct but the parity bit itself changed (a *false error*). It would be nice if the parity error detector would not only indicate that there was an error, but could also determine which bit changed so it could be corrected.

One method of error correction is what is known as *Iterative Parity Checking*. Imagine that a series of eight-bit bytes were being transmitted. Each byte would have a parity bit attached; however, there would also be a parity byte that contains a parity bit for each bit in the preceding five bytes. It is easiest to understand this by using a table (even parity is being used):

Byte	Data								Parity
1	0	0	0	0	0	0	0	0	0
2	1	0	1	1	0	0	0	0	1
3	1	0	1	1	0	0	1	1	1
4	1	1	1	0	1	0	1	0	1
5	0	1	0	0	0	0	0	0	1
P	1	0	1	0	1	0	0	1	0

Table 9.6: Iterative Parity

In Table 9.6, Byte one is 0000 0000. Since the system is set for even parity, and it is assumed that a byte with all zeros is even, then the parity bit is zero. Each of the five bytes has a parity bit that is properly set such that each byte (with the parity bit) includes an even number of bits. Then, after a group of five bytes a *parity byte* is inserted into the data stream so that each column of five bits also has a parity check; and that parity bit is found in row P on the table. Thus, the parity bit at the bottom of the first column is one since that column has three other ones. As a final check, the parity byte itself also has a parity bit added.

Table 9.7 is the same as Table 9.6, but Bit Zero, the least significant bit, in Byte One has been changed from a zero to a one (that number is highlighted).

Byte	Data								Parity
1	0	0	0	0	0	0	0	1	0
2	1	0	1	1	0	0	0	0	1
3	1	0	1	1	0	0	1	1	1
4	1	1	1	0	1	0	1	0	1
5	0	1	0	0	0	0	0	0	1
P	1	0	1	0	1	0	0	1	0

Table 9.7: Iterative Parity With Error

In Table 9.7 the parity for Byte One is wrong, and the parity for Bit Zero in the parity byte is wrong; therefore, Bit Zero in Byte One needs to be changed. If the parity bit for a row is wrong, but no column parity bits are wrong, or a column is wrong but no rows are wrong, then the parity bit itself is incorrect. This is one simple way to not only detect data errors, but correct those errors.

There are two weaknesses with iterative parity checking. First, it is restricted to only single-bit errors. If more than one bit is changed in a group then the system fails. This, though, is a general weakness for

most parity checking schemes. The second weakness is that a parity byte must be generated and transmitted for every few data bytes (five in the example). This increases the transmission time dramatically and normally makes the system unacceptably slow.

9.3.3 *Hamming Code*

9.3.3.1 *Introduction*

Richard Hamming worked at Bell labs in the 1940s and he devised a way to not only detect that a transmitted byte had changed, but exactly which bit had changed by interspersing parity bits within the data itself. Hamming first defined the "distance" between any two binary words as the number of bits that were different between them. As an example, the two binary numbers 1010 and 1010 has a distance of zero between them since there are no different bits, but 1010 and 1011 has a distance of one since one bit is different. This concept is called the *Hamming Distance* in honor of his work.

The circuit illustrated in Figure 9.13 calculates the Hamming distance between two four-bit numbers. In the illustration, 0100 and 1101 are compared and two bits difference in those two numbers is reported.

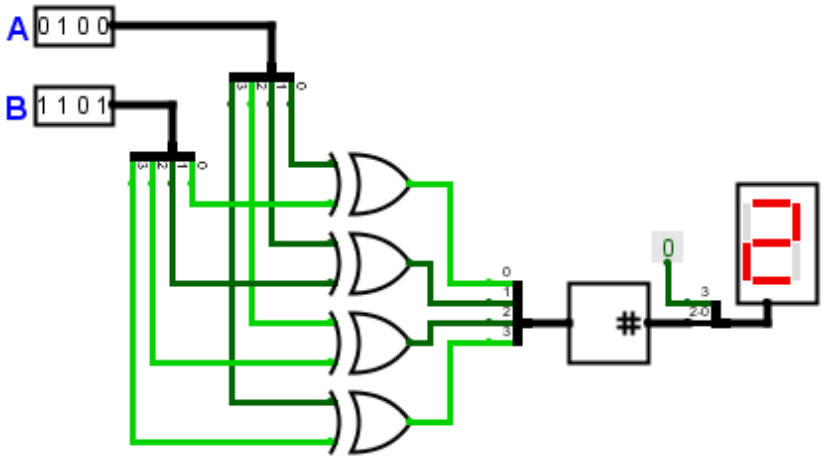

Figure 9.13: Hamming Distance

The four bits for input *A* and input *B* are wired to four XOR gates then the output of those gates is wired to a *Bit Adder* device. The XOR gates will output a one if the two input bits are different then the bit adder will total how many ones are present at its input. The output of the bit adder is a three-bit number but to make it easier to read that number it is wired to a hex digit display. Since that display needs a four-bit input a constant zero is wired to the most significant bit of the input of the hex digit display.

9.3.3.2 *Generating Hamming Code*

Hamming parity is designed so a parity bit is generated for various combinations of bits within a byte in such a way that every data bit is linked to at least three different parity bits. This system can then determine not only that the parity is wrong but which bit is wrong. The cost of a Hamming system is that it adds five parity bits to an eight-bit byte to create a 13-bit word. Consider the bits for the 13-bit word in Table 9.8.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	0	0	0	0	0	0	0	0	0	0	0

Table 9.8: Hamming Parity Bits

The bits numbered P_0 to P_4 are Hamming parity bits and the bits numbered d_0 to d_7 are the data bits. The Hamming parity bits are interspersed with the data but they occur in positions zero, one, three, and seven (counting right to left). The following chart shows the data bits that are used to create each parity bit:

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	X	0	X	0	X	0	X	0	X	0	P
0	0	X	X	0	0	X	X	0	0	X	P	0
0	X	0	0	0	0	X	X	X	P	0	0	0
0	X	X	X	X	P	0	0	0	0	0	0	0
P	X	0	X	X	0	0	X	X	0	X	0	0

Table 9.9: Hamming Parity Cover Table

From Table 9.9, line one shows the data bits that are used to set parity bit zero (P_0). If data bits d0, d1, d3, d4, and d6 are all one then P_0 would be one (even parity is assumed). The data bits needed to create the Hamming parity bit are marked in all five lines. A note is necessary about parity bit P_4. In order to detect transmission errors that are two bits large (that is, two bits were flipped), each data bit needs to be covered by three parity bits. Parity bit P_4 is designed to provide the third parity bit for any data bits that have only two others. For example, look down the column containing data bit d_0 and notice that it has only two parity bits (P_0 and P_1) before P_4. By adding P_4 to the circuit that data bit gets a third parity bit.

As an example of a Hamming code, imagine that this byte needed to be transmitted: 0110 1001. This number could be placed in the data bit positions of the Hamming table.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	0	1	0	0

Table 9.10: Hamming Example - Iteration 1

Bit zero, P0, is designed to generate even parity for data bits d_0, d_1, d_3, d_4, and d_6. Since there are three ones in that group, then P_0 must be one. That has been filled in below (for convenience, the Hamming parity bit pattern for P_0 is included in the last row of the table).

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	0	1	0	1
0	0	X	0	X	0	X	0	X	0	X	0	P

Table 9.11: Hamming Example - Iteration 2

Bit one, P_1, is designed to generate even parity for data bits d_0, d_2, d_3, d_5, and d_6. Since there are four ones in that group, then P_1 must be zero. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	0	1	0	1
0	0	X	X	0	0	X	X	0	0	X	P	0

Table 9.12: Hamming Example - Iteration 3

Bit three, P_2, is designed to generate even parity for data bits d_1, d_2, d_3, and d_7. Since there is one one in that group, then P_2 must be one. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	1	1	0	1
0	X	0	0	0	0	X	X	X	P	0	0	0

Table 9.13: Hamming Example - Iteration 4

Bit seven, P_3, is designed to generate even parity for data bits d_4, d_5, d_6, and d_7. Since there are two ones in that group, then P_3 must be zero. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	1	1	0	1
0	X	X	X	X	P	0	0	0	0	0	0	0

Table 9.14: Hamming Example - Iteration 5

Bit eight, P_4, is designed to generate even parity for data bits d_0, d_1, d_2, d_4, d_5, and d_7. Since there are two ones in that group, then P_4 must be zero. That has been filled in below.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	1	1	0	0	1	0	0	1	1	0	1
P	X	0	X	X	0	0	X	X	0	X	0	0

Table 9.15: Hamming Example - Iteration 6

When including Hamming parity, the byte 0110 1001 is converted to: 0 0110 0100 1101.

In Figure 9.14, a 11-bit input, *A*, is used to create a 16-bit word that includes Hamming parity bits. In the illustration, input 010 0111 0111 is converted to 1010 0100 1110 1111.

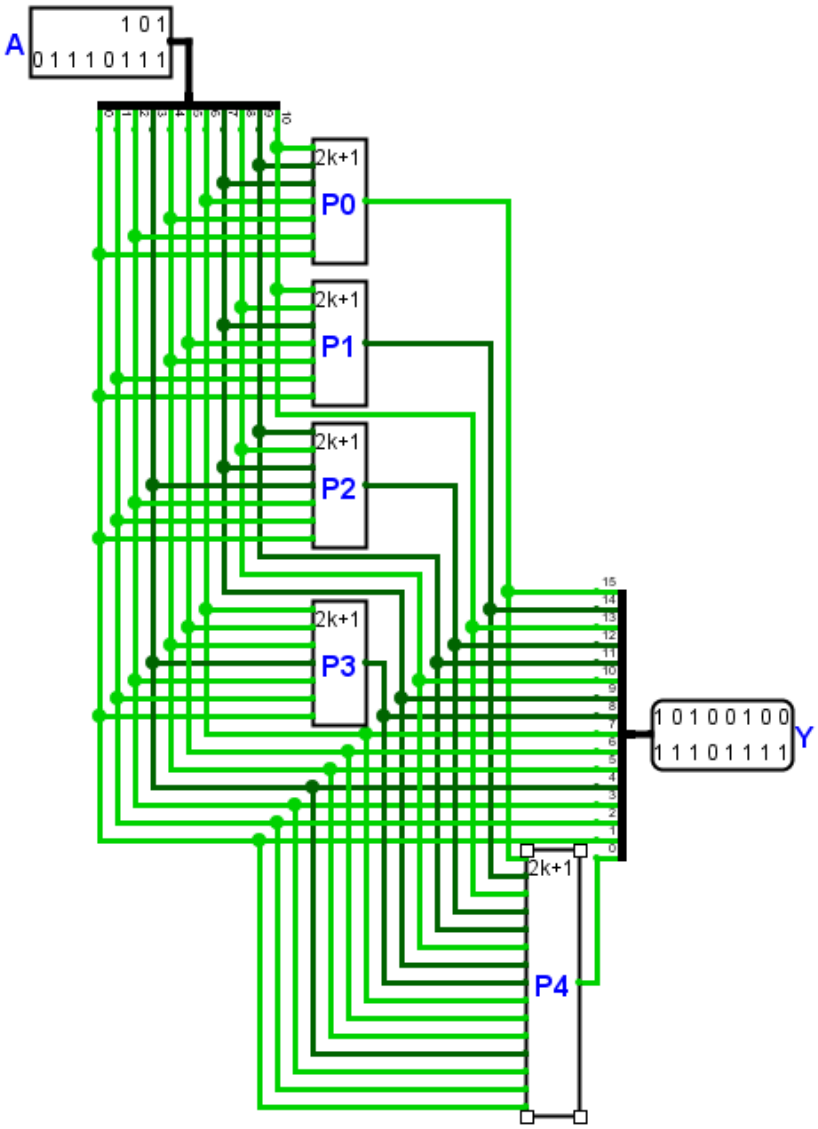

Figure 9.14: Generating Hamming Parity

The process used by the circuit in Figure 9.14 is to wire each of the input bits to various parity generators and then combine the outputs of those parity generators, along with the original bits, into a single 16-bit word. While the circuit has a lot of wired connections the concept is fairly simple. P0 calculates the parity for input bits $0, 1, 3, 4, 6, 8, 10$. That is then wired to the least significant bit of output Y.

9.3.3.3 *Checking Hamming Code*

To check the accuracy of the data bits in a word that contains Hamming parity bits the following general process is used:

1. Calculate the Hamming Parity Bit for each of the bit groups exactly like when the parity was first calculated.

2. Compare the calculated Hamming Parity bits with the parity bits found in the original binary word.

3. If the parity bits match then there is no error. If the parity bits do not match then the bad bit can be corrected by using the pattern of parity bits that do not match.

As an example, imagine that bit eight (last bit in the first group of four) in the Hamming code created above was changed from zero to one: 0 011$\underline{1}$ 0100 1101 (this is bit d_4). Table 9.16 shows that Hamming Bits P_0, P_3, and P_4 would now be incorrect since d_4 is used to create those parity bits.

P_4	d_7	d_6	d_5	d_4	P_3	d_3	d_2	d_1	P_2	d_0	P_1	P_0
0	0	X	0	X	0	X	0	X	0	X	0	P
0	0	X	X	0	0	X	X	0	0	X	P	0
0	X	0	0	0	0	X	X	X	P	0	0	0
0	X	X	X	X	P	0	0	0	0	0	0	0
P	X	0	X	X	0	0	X	X	0	X	0	0

Table 9.16: Hamming Parity Cover Table Reproduced

Since the only data bit that uses these three parity bits is d_4 then that one bit can be inverted to correct the data in the eight-bit byte.

The circuit illustrated in Figure 9.15 realizes a Hamming parity check. Notice that the input is the same 16-bit word generated in the circuit in Figure 9.14 except bit eight (the last bit on the top row of the input) has been complemented. The circuit reports that bit eight is in err so it would not only alert an operator that something is wrong with this data but it would also be able to automatically correct the wrong bit.

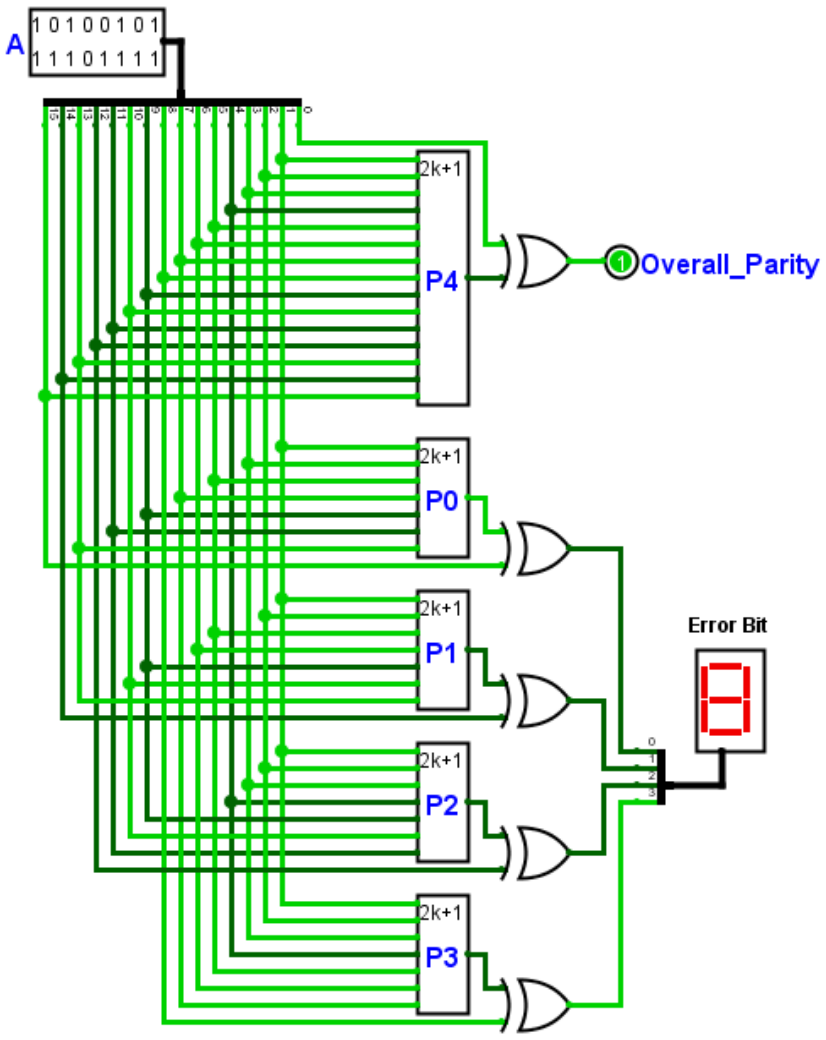

Figure 9.15: Checking Hamming Parity

9.3.4 *Hamming Code Notes*

- When a binary word that includes Hamming parity is checked to verify the accuracy of the data bits using three overlapping parity bits, as developed in this book, one-bit errors can be corrected and two-bit errors can be detected. This type of system is often called Single Error Correction, Double Error Detection (SECDED) and is commonly used in computer memories to ensure data integrity.

- While it seems wasteful to add five Hamming bits to an eight-bit byte (a 62.5% increase in length), the number of bits needed for longer words does not continue to increase at that rate. Hamming bits are added to a binary word in multiples of powers of two. For example, to cover a 32-bit word only seven Hamming bits are needed, an increase of only about 22.%; and to cover a 256-

bit word only 10 Hamming bits are needed, an increase of just under 4%.

- This lesson counts bits from right-to-left and considers the first position as bit zero, which matches with the bit counting pattern used throughout the book. However, many authors and online resources count Hamming bits from left-to-right and consider the left-most position as bit one because that is a natural way to count.

9.3.5 *Sample Problems*

The following problems are provided for practice.

8-Bit Byte	With Hamming
11001010	1110011011001
10001111	1100001110111
01101101	0011011100111
11100010	0111000010000
10011011	1100111011100

Table 9.17: Hamming Parity Examples

Hamming With Error	Error Bit
0110101011001	1
1100000100110	3
0000100001100	5
1110111011010	9
1110010100110	12

Table 9.18: Hamming Parity Errors