

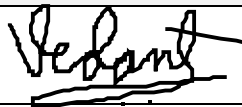


<b>Course Title:</b>	
<b>Course Number:</b>	
<b>Semester/Year (e.g.F2016)</b>	

<b>Instructor:</b>	
--------------------	--

<i>Assignment/Lab Number:</i>	
<i>Assignment/Lab Title:</i>	

<i>Submission Date:</i>	
<i>Due Date:</i>	

<b>Student LAST Name</b>	<b>Student FIRST Name</b>	<b>Student Number</b>	<b>Section</b>	<b>Signature*</b>
				

\*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: <http://www.ryerson.ca/senate/current/pol60.pdf>

## Introduction

In digital communications the transmitted message must travel through a noisy channel, whether wireless or otherwise, before reaching the receiver. This may induce errors in the message sequence which will make the received message unusable. This will cause the receiver to request a retransmission of the message which is obviously undesirable if a high bit rate is required. One way to recover the true message from the erroneous message is to use a technique called error control coding. The essence of the technique is to add redundant data bits to the message sequence at the transmitter side which contain enough information to recover the bits in the sequence which have been affected by errors. Error control codes are characterized by the length of the codeword, its error detection capability and its error correcting capabilities. One of the earliest error control codes that was conceived is the Hamming code which is a linear block code that can correct one-bit errors. Block codes break apart the initial message sequence into equal length blocks which are encoded separately. Linear codes have the property where if any two codewords are added together it will result in another valid codeword. Although Hamming codes are widely used in digital communications the same theory has been used in the field of digital systems testing. For instance, the theory behind error control coding has been used to design LFSR's, or linear feedback shift registers, and MISR's, or multiple input signature registers, which are crucial components in BIST, or built-in self-test, systems. The objective of this paper is to explore the theory of error control codes, particularly the Hamming code, and implement an encoder and decoder in VHDL. Two versions of the encoder and decoder will be developed and both implementations will be compared to determine their strengths and weaknesses.

## Theory

Linear block codes are denoted using  $(n, k)$  where  $n$  is the length of the codeword, and  $k$  is the number of information bits in the codeword. Since there are  $k$  bits in the message the number of possible binary sequences is  $2^k$ . Similarly, there are  $2^n$  possible codewords. The  $k$  bit message sequence is assigned an  $n$  bit codeword in a one-to-one fashion. The set of all  $n$  bit binary sequences is called the vector space  $V_n$  over the Galois field of two elements. Addition and multiplication in the Galois field of two elements, denoted as  $GF(2)$ , is different than the corresponding operations in the real vector space. Table 1 shows the addition and multiplication operations in  $GF(2)$ .

Table 1: Operations in GF(2)	
Addition	Multiplication
$0 + 0 = 0$	$0 \times 0 = 0$
$0 + 1 = 1$	$0 \times 1 = 0$
$1 + 0 = 1$	$1 \times 0 = 0$
$1 + 1 = 0$	$1 \times 1 = 1$

The subset of  $V_n$  is called a subspace, denoted by  $S$ , if it meets the following criteria. It contains the  $n$  bit sequence or vector with all zeros and the sum of any two vectors in  $S$  also exists in  $S$ . The  $(n, k)$  code is called a linear block code if the valid codewords form a subspace of the vector space  $V_n$ . Since  $k < n$ , the  $2^k$  valid codewords are chosen from the  $2^n$  possible  $n$  bit binary sequences with the aim of increasing the distance between the valid codewords and reducing the amount of redundancy required. To generate the codeword from the message vector a generator matrix is used. The generator matrix is constructed from the basis vectors of the subspace  $V_k$  of the vector space  $V_n$ . Basis vectors are the smallest number of linearly independent vectors that can span the entire subspace  $V_k$ , in other words every vector in the subspace can be expressed as a linear combination of basis vectors. The number of basis vectors is the dimension of the subspace  $V_k$  and is equal to  $k$ . Many generator matrices exist for the  $(n, k)$  linear block code so for the purposes of this project the generator matrix that will be used is given below. The first three columns taken together is called the parity matrix  $P$  and the last four columns is the identity matrix  $I_4$ .

$$G = [P \ I_4] = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The codeword for a message is obtained by multiplying the message vector by the generator matrix as shown below.

$$\begin{aligned} \vec{u} = \vec{m}G &= [m_1 m_2 m_3 m_4] \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \\ &= [(m_1 + m_3 + m_4)(m_1 + m_2 + m_3)(m_2 + m_3 + m_4)m_1 m_2 m_3 m_4] \\ &= [p_1 p_2 p_3 m_1 m_2 m_3 m_4] \end{aligned}$$

If the codewords are selected in such a way that the message bits appear together then this type of code is called systematic. In the expression above, the message bits are the last four bits of the codeword while the first three bits are the so-called parity bits. When the codeword is transmitted through the noisy channel some bits may be corrupted so the received codeword will not be a valid codeword. Let  $\vec{r} = \vec{u} + \vec{e}$  be the received codeword where  $\vec{u}$  is a valid codeword and  $\vec{e}$  is the error pattern that corrupts it. To obtain the uncorrupted codeword at the receiver side the received codeword is multiplied by the parity check matrix, which is shown below.

$$H^T = \begin{bmatrix} I^{n-k} \\ P \end{bmatrix} = \begin{bmatrix} I^3 \\ P \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The result of the multiplication is called the syndrome. A unique characteristic of the syndrome is that it is dependent only on the error pattern of the received codeword, as shown below.

$$\vec{s} = \vec{r}H^T = (\vec{u} + \vec{e})H^T = \vec{m}GH^T + \vec{e}H^T = \vec{0} + \vec{e}H^T = \vec{e}H^T$$

If the syndrome is zero it implies that the received codeword is valid and no error was made during transmission. Otherwise, there is an error in the codeword, and the error pattern has a one-to-one relationship with the syndrome. A look up table can be used to find the error pattern after calculating the syndrome at the receiver side. After the error pattern is found it can be added to the received codeword to flip the erroneous bits to obtain a good estimate for the valid codeword that was transmitted.

Cyclic codes are a subset of linear block codes. An  $(n, k)$  linear block code is called a cyclic code if all codewords in the subspace  $S$ , except for the zero codeword, can be obtained by shifting any codeword in  $S$  a certain number of bits. For instance, if  $U = (u_0, u_1, \dots, u_{n-1})$  is a codeword in  $S$  then  $U^1 = (u_{n-1}, u_0, u_1, \dots, u_{n-2})$  is also a codeword in  $S$ . The codeword can be expressed as a polynomial of order  $n - 1$  as shown below, where each term represents the presence of a one or zero binary digit.

$$U(x) = u_0 + u_1x + u_2x^2 + \dots + u_{n-1}x^{n-1}$$

The advantage of representing the codeword as a polynomial is that multiplying the codeword polynomial by  $x$  is equal to shifting the codeword one bit to the right. Let  $U^i(x)$  be the polynomial of the codeword shifted to the right by  $i$  bits, however this time the shifted bits will wrap around to the front of the codeword. The relationship between the original polynomial and the shifted polynomial is given below, where  $n$  is the length of the codeword.

$$U^i(x) = x^i U(x) \text{mod}(x^n + 1)$$

In order to obtain the codeword for the message it must first be represented as a message polynomial, as shown below.

$$m(x) = m_0 + m_1x + m_2x^2 + \dots + m_{k-1}x^{k-1}$$

The codeword polynomial is generated from the message polynomial by multiplying the latter with the generator polynomial  $g(x)$ .

$$U(x) = m(x)g(x)$$

This shows that all valid codeword polynomials must be divisible by  $g(x)$  without a remainder. The requirements for a generator polynomial to uniquely generate cyclic codes are that it must divide  $x^n + 1$  without a remainder and must have a degree of  $n - k$ . It is desirable to generate codewords systematically where the message bits are together in the codeword. To generate systematic cyclic codes the following formula will be used.

$$U(x) = x^{n-k}m(x) \text{mod}g(x) + x^{n-k}m(x)$$

There are many generator polynomials for (7, 4) and (15, 11) codes, however for the purposes of this paper the following polynomials will be used.

$$g(x) = 1 + x + x^3$$

$$g(x) = 1 + x + x^4$$

After obtaining the codeword polynomial it is necessary to recover the original message polynomial at the receiver side. Let  $Z(x) = U(x) + e(x)$  be the received polynomial, where  $U(x)$  is the valid codeword polynomial and  $e(x)$  is the error polynomial.  $Z(x)$  should be divisible by  $g(x)$  without a remainder, otherwise it is assumed that the received polynomial is erroneous. The remainder is called the syndrome of the received polynomial and it is given by the following equation.

$$S(x) = Z(x) \bmod g(x)$$

The codeword component of the received polynomial is divisible by  $g(x)$  without a remainder. Therefore, the syndrome is only dependent on the error pattern polynomial as shown below.

$$\begin{aligned} S(x) &= Z(x) \bmod g(x) = (U(x) + e(x)) \bmod g(x) = U(x) \bmod g(x) + e(x) \bmod g(x) \\ &= e(x) \bmod g(x) \end{aligned}$$

Hamming codes are a class of linear block codes where the length of the codeword and the number of information bits follows the structure given below.

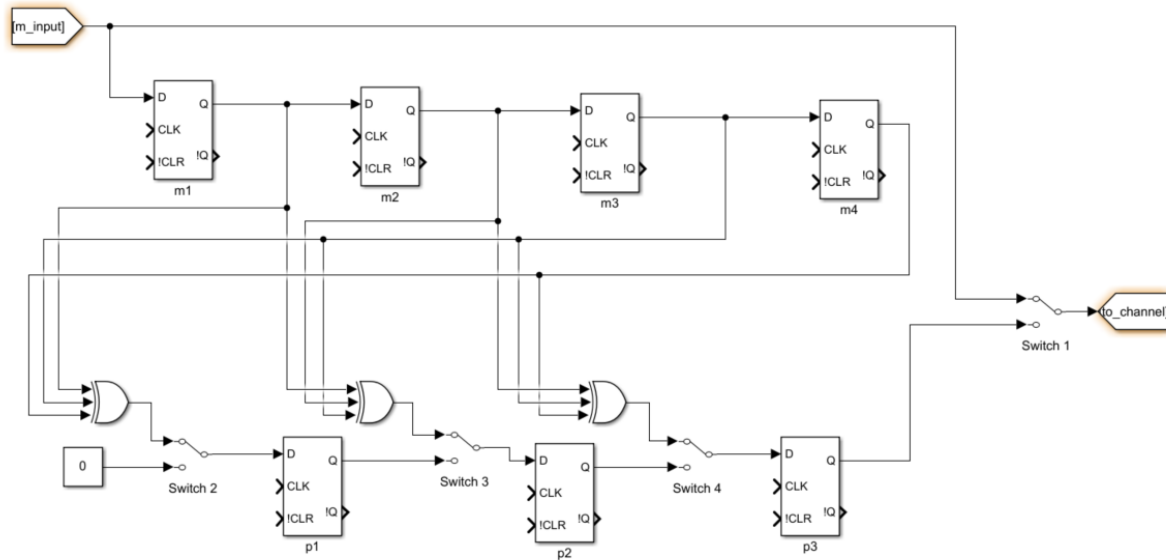
$$(n, k) = (2^m - 1, 2^m - 1 - m), m = 2, 3 \dots$$

In addition, Hamming codes are also perfect codes which means they are able to correct exactly one error bit in the received codeword at any location. This is because there are  $n - k = m$  bits in the syndrome. Which means there are  $2^m$  number of syndromes and by extension  $2^m$  error patterns because there is a one-to-one relationship between the two. Excluding all zero syndrome which indicates no error, the total number of error patterns is  $2^m - 1$  which is the same as the length of the codeword. Therefore, there are exactly enough error patterns to corrupt every bit in the codeword.

## Hardware Design

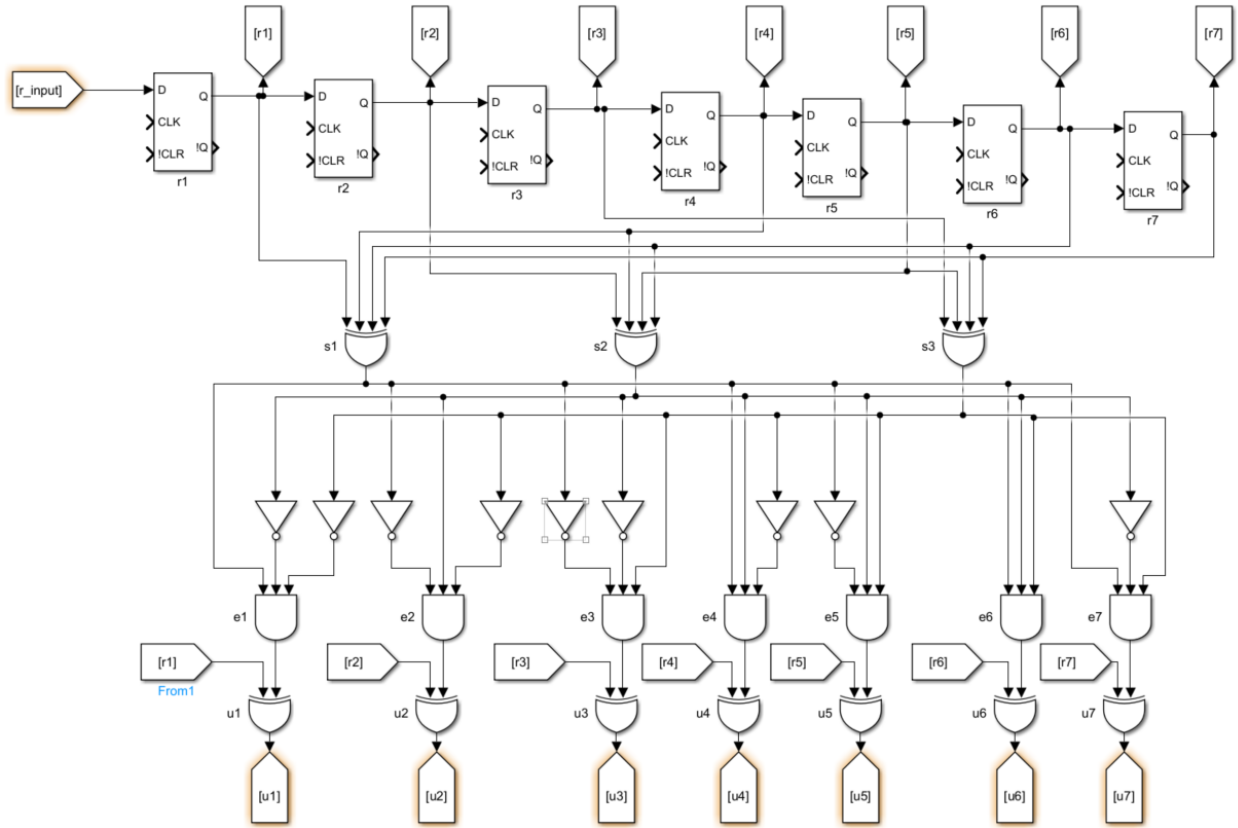
The encoder and decoders for two Hamming codes, namely (7, 4) and (15, 11), will be implemented. The following section will only describe the design of the encoders and decoders for the (7, 4) code since the design of the (15, 11) code is similar. Additionally, two different hardware design approaches will be used for each code. The first hardware design approach will be based off the matrix method of generating linear block codes described in the previous section. Figure 1 shows the schematic of the encoder which consists of four registers to store the message bits and three registers to store the parity bits. During operation the message bits will be simultaneously shifted into the message registers and shifted out onto the channel. The XOR gates are used to calculate the value of the parity bits and store them in the parity registers.

Inputs to the XOR gates are determined based on the generator matrix. For instance, the XOR gate feeding parity register  $p1$  is taking its inputs from message registers  $m1$ ,  $m3$  and  $m4$  which correspond to the first column of the generator matrix. After all message bits are shifted into the message registers Switch 1 will flip to the lower position allowing the parity bits to be shifted out.



**Figure 1: Matrix based encoder design**

Figure 2 shows the schematic of the decoder design which consists of seven registers to store the received codeword bits. During operation the received codeword bits are shifted into the register. The XOR gates that are labelled  $s1$ ,  $s2$  and  $s3$  will calculate the syndrome from the codeword bits currently stored in the register. The syndrome is fed into a logic circuit that will generate the corresponding error pattern. For example, syndrome  $\vec{s} = 100$  will result in the error pattern  $\vec{e} = 1000000$  because only the first AND gate will be turned on with this syndrome pattern. The error pattern is added back to the received codeword to generate the corrected codeword.



**Figure 2: Matrix based decoder design**

The second hardware design approach is based off the cyclic codes that were described in the second half of the theory section. The encoder shown in Figure 3 is implemented as a linear feedback shift register, or LFSR. The LFSR is designed directly from the coefficients of the generator polynomial  $g(x) = g_0 + g_1x + g_2x^2 + g_3x^3 = 1 + x + x^3$ . If the coefficient is a binary one, then there exists a feedback path. Since for the given generator polynomial only  $g_1 = 1$  there will be a feedback path for the second register from the left. During operation Switch 2 is lowered to the bottom position to allow the message bits to simultaneously shift into the LFSR and out onto the channel. After all message bits are shifted into the LFSR the bits remaining in the register are the parity bits. Switch 2 is then flipped to the upper position to shift the parity bits onto the channel following the message bits.





## VHDL Implementation

Hardware Descriptive Language, or HDL, is used to describe the design for a digital logic circuit in the form of a program. Two popular HDL languages are VHDL and Verilog. The encoders and decoders for both Hamming codes were implemented in VHDL. Furthermore, each code is implemented using the matrix approach and the cyclic shift approach. The VHDL code is provided at the end of this document. A VHDL description of a digital circuit consists of an entity declaration and architecture declaration. The former describes the input and output ports of the logic block while the latter describes the functionality of the logic block. Several architectures can be associated with a single entity. For example, the “hamming\_encoder\_7\_4” entity has the architectures “matrix\_arch\_7\_4” and “cyclic\_arch\_7\_4” associated with it. All encoder and decoder implementations require “clk” and “reset” ports since they are sequential circuits. The “hamming\_encoder\_7\_4” entity has two additional ports “m” and “u” which are the serial input and output ports respectively. The “temp\*” ports found in all the designs are for conveniently viewing the state of internal registers and are not mandatory. The “matrix\_arch\_7\_4” architecture contains three registers. The message and parity registers are for storing the message and parity bits. The third register is for counting the number of bits that have been received by the encoder. All statements within an architecture body will be executed concurrently except for those placed in process blocks. Statements within a process block are executed sequentially. The process block must include a sensitivity list which contains a list of signals which reevaluates the process block if any of them change. The first process block in the “matrix\_arch\_7\_4” architecture describes the functionality of the registers in the design. If the “reset” signal is logic one, then the registers will be set to their default states while a rising clock edge will set the registers to their next state values. The rest of the architecture body contains the code for the next state logic and the output logic. For instance, a conditional signal assignment statement is used to increment the counter unless it reaches its maximum value in which case it will wrap around. The next state value of the message register is always the incoming bit concatenated with all but the last bit of the current message register. The second process statement in the architecture is used to calculate the parity bits and produce the output of the encoder. The sensitivity list includes the “count\_reg” signal because the next state value of the parity bits must be recalculated after every bit received by the encoder. If the number of incoming bits captured by the encoder are fewer or equal to the number of message registers, then the output of the encoder is the incoming message bit. During this phase of operation, the next state values of the parity registers are being calculated using the equations from the generator matrix. After all message bits have been inputted into the encoder the parity registers will contain the parity bits of the codeword for the message. During this phase of operation, the output of the encoder will be the last bit stored in the parity registers. The next state value of the parity registers is the current value stored in the parity registers logically shifted to the right by one bit. The other encoder and decoder blocks are designed using a similar approach and detailed explanation of their functionality will not be included for the sake of brevity. After the VHDL code for the logic blocks are written they are simulated to verify that they are operating correctly. Simulating a VHDL description of a circuit requires implementing a so-called testbench that will input test vectors into the design and display the output of the logic block. All encoder and

decoder designs were simulated and the outputs they produced are compared with manually calculated results to verify their functionality.

## Experimental Results

The simulation waveforms for all logic blocks are shown in Figure 5 through Figure 12. Figure 5 shows the output of the encoder for the (7, 4) Hamming code when the input is the message  $\vec{m} = m_1m_2m_3m_4 = 1001$ . The message bits are inputted into the encoder last bit first so the encoder will also produce the codeword last bit first. The codeword is obtained by multiplying the message vector with the generator matrix, as shown below.

$$\vec{u} = \vec{m}G = [1001] \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} = [0111001]$$

The (7, 4) encoder produces the output  $\vec{u} = u_1u_2u_3u_4u_5u_6u_7 = 0111001$  which matches the codeword that was calculated manually. Figure 6 shows the output of the (7, 4) encoder using the cyclic shift architecture. The output of the cyclic shift encoder is  $\vec{u} = 0111001$  which also matches the codeword that was manually calculated earlier. Figure 7 shows the output of the encoder for the (15, 11) Hamming code when the input is the message  $\vec{m} = m_1m_2m_3m_4m_5m_6m_7m_8m_9m_{10}m_{11} = 10110011101$ . The manual calculation of the codeword is shown below.

$$\vec{u} = \vec{m}G = [10110011101] \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = [110110110011101]$$

The output of the encoder based on the matrix architecture is  $\vec{u} = 110110110011101$  which matches the codeword that was manually calculated. Figure 8 shows the output of the encoder based on the cyclic shift architecture which also produced the correct codeword. Figure 9 shows the output of the (7, 4) decoder based on the matrix architecture. The input to the decoder is  $\vec{r} = r_1r_2r_3r_4r_5r_6r_7 = 0101001$  which is the valid codeword  $\vec{u} = 0111001$  where the 3<sup>rd</sup> bit has been corrupted. The manual calculation of the syndrome is shown below.

$$\vec{s} = \vec{r}H^T = [0101001] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = [001]$$

It is possible to extrapolate the error pattern from the syndrome because the syndrome is only dependent on the error pattern portion of the received codeword. Therefore, the error pattern that corresponds to the syndrome  $\vec{s} = 001$  is  $\vec{e} = 0010000$ . The decoder based on the matrix architecture produces the corrected codeword once all received codeword bits have been shifted into the decoder. The output of the decoder after the seventh clock cycle is  $\vec{u} = 0111001$ . The error pattern and the syndrome stored in the internal registers of the decoder after the seventh clock cycle is  $\vec{e} = 0010000$  and  $\vec{s} = 001$  which matches the values that were calculated earlier. Figure 10 shows the output of the (7, 4) decoder, which is  $\vec{u} = 0111001$ , based on the cyclic shift architecture. The difference between this and the previous approach is that the output of the decoder is serial. This means more clock cycles are required to get the output of the decoder. In fact, the cyclic shift decoder requires three times as many clock cycles to get the output. For example, for the (7, 4) decoder the first seven clock cycles are needed to shift in the received codeword into the decoder's register. The next seven clock cycles are needed to correct the error and the final seven clock cycles are required to shift out the corrected codeword from the decoder's register. Figure 11 shows the output of the decoder for the (15, 11) code based on the matrix architecture. The input to the decoder was  $\vec{r} = r_1r_2r_3r_4r_5r_6r_7r_8r_9r_{10}r_{11}r_{12}r_{13}r_{14}r_{15} = 110110110010101$  which is the valid codeword  $\vec{u} = 110110110011101$  where the twelfth bit has been corrupted. The manual calculation of the syndrome is shown below.

$$\vec{s} = \vec{r}H^T = [110110110010101] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} = [0111]$$

Like what was done before the error pattern can be extrapolated from the syndrome. For the syndrome  $\vec{s} = 0111$  the error pattern is  $\vec{e} = 000000000001000$ . The decoder based on the matrix architecture yielded a corrected codeword of  $\vec{u} = 110110110011101$  which is the

expected result. In addition, during the fifteenth clock cycle the internal registers of the decoder also contains the same syndrome and error pattern as what was calculated above. Figure 12 shows the output of the decoder for the (15, 11) code using the cyclic shift architecture. The decoder eventually outputs the correct codeword, however it takes forty-five clock cycles to finish the operation.

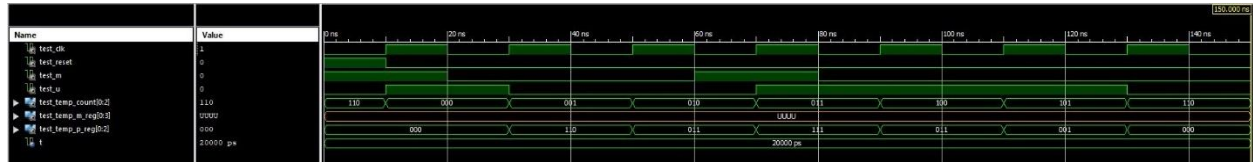


Figure 5: (7, 4) Hamming encoder matrix architecture waveforms

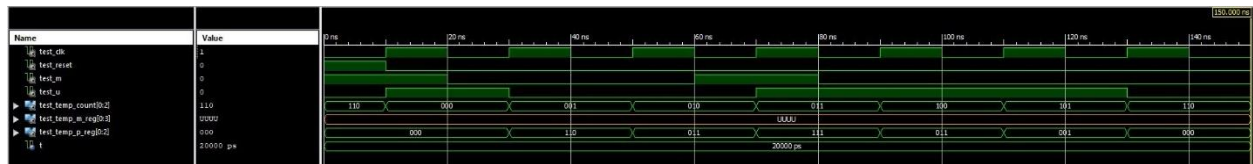


Figure 6: (7, 4) Hamming encoder cyclic shift architecture waveforms

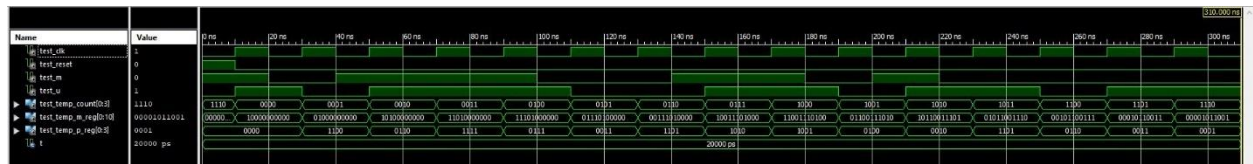


Figure 7: (15, 11) Hamming encoder matrix architecture waveforms



Figure 8: (15, 11) Hamming encoder cyclic shift architecture waveforms



Figure 9: (7, 4) Hamming decoder matrix architecture waveforms



Figure 10: (7, 4) Hamming decoder cyclic shift architecture waveforms

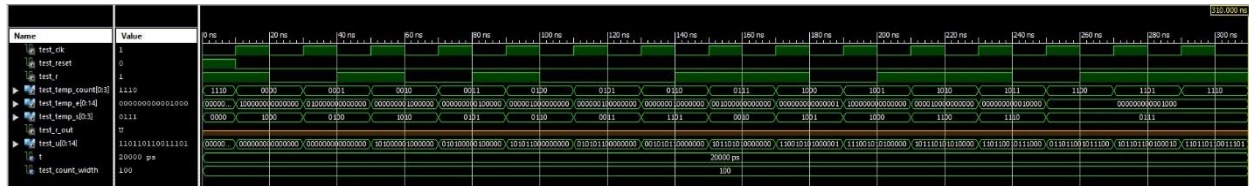


Figure 11: (15, 11) Hamming decoder matrix architecture waveforms

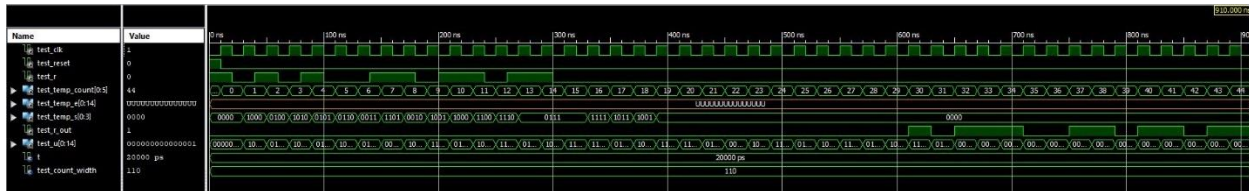


Figure 12: (15, 11) Hamming decoder cyclic shift architecture waveforms

## Conclusion

In conclusion, the objective of the project was accomplished because the encoder and decoder design for the (7, 4) and (15, 11) Hamming code were designed and implemented in VHDL. The VHDL descriptions of the logic circuits were then simulated, and the output waveforms were examined to ensure they were operating correctly. Two different approaches were followed when designing the encoders and decoders. The first was derived directly from the generator matrix and was dubbed matrix architecture. While the second was derived from the theory of cyclic codes and was subsequently called cyclic shift architecture. Although both approaches were different the encoders and decoders produced the same results for the same test vectors. However, the cyclic shift architecture had some significant advantages and some manageable disadvantages that made it preferable over the other approach for real life situations. For instance, the matrix architecture required a decoder to find the corresponding error pattern given the syndrome. This is problematic because as the length of the codeword increases from 7 to 15 more logic gates are needed to implement the syndrome to error pattern decoder. Furthermore, converting the VHDL code is also more difficult since the syndrome to error pattern decoder must be redesigned from scratch. On the other hand, the complexity of the cyclic shift architecture does not increase by much when the codeword length changes from 7 to 15. Only the sizes of the buffer register and LFSR needs to be updated. As a result, the VHDL code is easier to update as well. The main disadvantage of the cyclic shift approach is that it requires far more clock cycles, as much as three times, to decode the codeword than the matrix approach. However, due to advancements in technology the clock speeds are getting faster so making the trade between a slower device and lower hardware cost is a better strategy.

## References

- [1] EE8505 Lecture Slides
- [2] P. P. Chu, *FPGA prototyping by VHDL examples : Xilinx MicroBlaze MCS SoC*. Hoboken, Nj, Usa: Wiley, 2017.
- [3] B. Sklar, *Digital communications : : fundamentals and applications*. London: Pearson, 2018.
- [4] M. L. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. New York: Springer Science+Business Media, 2000.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
-- (7,4) hamming code
```

```
entity hamming_encoder_7_4 is
```

```
port(
    clk, reset: in std_logic;
    m: in std_logic;
    temp_count: out std_logic_vector(0 to 2);
    temp_m_reg: out std_logic_vector(0 to 3);
    temp_p_reg: out std_logic_vector(0 to 2);
    u: out std_logic
);
```

```
end hamming_encoder_7_4;
```

```
architecture matrix_arch_7_4 of hamming_encoder_7_4 is
```

```
    signal m_reg, m_next: std_logic_vector(0 to 3);
    signal p_reg, p_next: std_logic_vector(0 to 2);
    signal count_reg, count_next: unsigned(0 to 2);
```

```
begin
```

```
-- register
```

```
process(clk, reset)
```

```
begin
```

```
    if(reset = '1') then
```

```
        m_reg <= (others => '0');
```

```
        p_reg <= (others => '0');
```

```
        count_reg <= "110";
```

```
    elsif(clk'event and clk='1') then
```

```
        m_reg <= m_next;
```

```
        p_reg <= p_next;
```

```
        count_reg <= count_next;
```

```
    end if;
```

```
end process;
```

```
-- next state and output logic
```

```
count_next <= (others => '0') when count_reg = 6 else count_reg + 1;
```

```
m_next <= m & m_reg(0 to 2);
```

```
process(count_reg)
```

```
begin
```

```
-- shift-in the message bits into the message register
```

```
-- at the same time shift-out the incoming message bit to the channel
```

```
if(count_reg < 4) then
```

```
    u <= m;
```

```
    p_next(0) <= m_reg(0) xor m_reg(2) xor m_reg(3);
```

```
    p_next(1) <= m_reg(0) xor m_reg(1) xor m_reg(2);
```

```
    p_next(2) <= m_reg(1) xor m_reg(2) xor m_reg(3);
```

```
-- shift-out the parity bits into the channel
```

```
else
```

```
    u <= p_reg(2);
```

```
    p_next <= '0' & p_reg(0 to 1);
```

```
end if;
```

```
end process;
```

```

temp_count <= std_logic_vector(count_reg);
temp_m_reg <= m_reg;
temp_p_reg <= p_reg;
end matrix_arch_7_4;

```

architecture cyclic\_arch\_7\_4 of hamming\_encoder\_7\_4 is

```

signal count_reg, count_next: unsigned(0 to 2);
signal p_reg, p_next: std_logic_vector(0 to 2);
begin
-- register
process(clk, reset)
begin
if(reset = '1') then
p_reg <= (others => '0');
count_reg <= "110";
elsif(clk'event and clk = '1') then
p_reg <= p_next;
count_reg <= count_next;
end if;
end process;

-- next state and output logic
count_next <= (others => '0') when count_reg = 6 else count_reg + 1;

process(count_reg)
begin
-- shift the message bits onto the channel
-- while simultaneously shifting the message bits into the LFSR
if(count_reg < 4) then
u <= m;
p_next(0) <= p_reg(2) xor m;
p_next(1) <= p_reg(0) xor p_reg(2) xor m;
p_next(2) <= p_reg(1);
else
u <= p_reg(2);
p_next <= '0' & p_reg(0 to 1);
end if;
end process;

temp_count <= std_logic_vector(count_reg);
temp_p_reg <= p_reg;
end cyclic_arch_7_4;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

```

-- (15, 11) hamming encoder

entity hamming\_encoder\_15\_11 is

```

generic(
N: integer:= 15;
K: integer:= 11;
P: integer:= 4

```



```

);
port(
  clk, reset: in std_logic;
  m: in std_logic;
  temp_count: out std_logic_vector(0 to 3);
  temp_m_reg: out std_logic_vector(0 to 10);
  temp_p_reg: out std_logic_vector(0 to 3);
  u: out std_logic
);
end hamming_encoder_15_11;

```

architecture matrix\_arch\_15\_11 of hamming\_encoder\_15\_11 is

```

  signal m_reg, m_next: std_logic_vector(0 to 10);
  signal p_reg, p_next: std_logic_vector(0 to 3);
  signal count_reg, count_next: unsigned(0 to 3);
begin

```

```

  -- register

```

```

  process(clk, reset)

```

```

  begin

```

```

    if(reset = '1') then

```

```

      m_reg <= (others => '0');

```

```

      p_reg <= (others => '0');

```

```

      count_reg <= "1110";

```

```

    elsif(clk'event and clk='1') then

```

```

      m_reg <= m_next;

```

```

      p_reg <= p_next;

```

```

      count_reg <= count_next;

```

```

    end if;

```

```

  end process;

```

```

  -- next state and output logic

```

```

  count_next <= (others => '0') when count_reg = 14 else count_reg + 1;

```

```

  m_next <= m & m_reg(0 to 9);

```

```

  process(count_reg)

```

```

  begin

```

```

    -- shift-in the message bits into the message register

```

```

    -- at the same time shift-out the incoming message bit to the channel

```

```

    if(count_reg < 11) then

```

```

      u <= m;

```

```

      p_next(0) <= m_reg(0) xor m_reg(3) xor m_reg(4) xor m_reg(6) xor m_reg(8) xor m_reg(9) xor m_reg(10);

```

```

      p_next(1) <= m_reg(0) xor m_reg(1) xor m_reg(3) xor m_reg(5) xor m_reg(6) xor m_reg(7) xor m_reg(8);

```

```

      p_next(2) <= m_reg(1) xor m_reg(2) xor m_reg(4) xor m_reg(6) xor m_reg(7) xor m_reg(8) xor m_reg(9);

```

```

      p_next(3) <= m_reg(2) xor m_reg(3) xor m_reg(5) xor m_reg(7) xor m_reg(8) xor m_reg(9) xor m_reg(10);

```

```

    -- shift-out the parity bits into the channel

```

```

  else

```

```

    u <= p_reg(3);

```

```

    p_next <= '0' & p_reg(0 to 2);

```

```

  end if;

```

```

  end process;

```

```

  temp_count <= std_logic_vector(count_reg);

```

```

  temp_m_reg <= m_reg;

```

```

  temp_p_reg <= p_reg;

```

```

end matrix_arch_15_11;

```

architecture cyclic\_arch\_15\_11 of hamming\_encoder\_15\_11 is

signal count\_reg, count\_next: unsigned(0 to 3);

signal p\_reg, p\_next: std\_logic\_vector(0 to 3);

begin

-- register

process(clk, reset)

begin

if(reset = '1') then

p\_reg <= (others => '0');

count\_reg <= "1110";

elsif(clk'event and clk = '1') then

p\_reg <= p\_next;

count\_reg <= count\_next;

end if;

end process;

-- next state and output logic

count\_next <= (others => '0') when count\_reg = 14 else count\_reg + 1;

process(count\_reg)

begin

-- shift the message bits onto the channel

-- while simultaneously shifting the message bits into the LFSR

if(count\_reg < 11) then

u <= m;

p\_next(0) <= p\_reg(3) xor m;

p\_next(1) <= p\_reg(0) xor p\_reg(3) xor m;

p\_next(2) <= p\_reg(1);

p\_next(3) <= p\_reg(2);

else

u <= p\_reg(3);

p\_next <= '0' & p\_reg(0 to 2);

end if;

end process;

temp\_count <= std\_logic\_vector(count\_reg);

temp\_p\_reg <= p\_reg;

end cyclic\_arch\_15\_11;

-- (7, 4) hamming decoder

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity hamming\_decoder\_7\_4 is

generic(

COUNT\_WIDTH: integer:= 3

);

port(

clk, reset: in std\_logic;

r: in std\_logic;

temp\_e: out std\_logic\_vector(0 to 6);

temp\_count: out std\_logic\_vector(0 to COUNT\_WIDTH-1);

```

temp_s: out std_logic_vector(0 to 2);
r_out: out std_logic;
u: out std_logic_vector(0 to 6)
);
end hamming_decoder_7_4;

```

architecture matrix\_arch\_7\_4 of hamming\_decoder\_7\_4 is

```

signal r_reg, r_next: std_logic_vector(0 to 6);
signal s: std_logic_vector(0 to 2);
signal count_reg, count_next: unsigned(0 to COUNT_WIDTH-1);

```

```
begin
```

```
-- register
```

```
process(clk, reset)
```

```
begin
```

```
if(reset = '1') then
```

```
    r_reg <= (others => '0');
```

```
    count_reg <= "110";
```

```
elsif(clk'event and clk='1') then
```

```
    r_reg <= r_next;
```

```
    count_reg <= count_next;
```

```
end if;
```

```
end process;
```

```
-- next state and output logic
```

```
count_next <= (others => '0') when count_reg = 6 else count_reg + 1;
```

```
r_next <= r & r_reg(0 to 5);
```

```
-- calculate the syndrome
```

```
s(0) <= r_reg(0) xor r_reg(3) xor r_reg(5) xor r_reg(6);
```

```
s(1) <= r_reg(1) xor r_reg(3) xor r_reg(4) xor r_reg(5);
```

```
s(2) <= r_reg(2) xor r_reg(4) xor r_reg(5) xor r_reg(6);
```

```
-- decode the syndrome to get the error pattern
```

```
-- add the error pattern to the received codeword to get the correct codeword
```

```
process(s, r_reg)
```

```
begin
```

```
case(s) is
```

```
when "000" =>
```

```
    u <= r_reg xor "0000000";
```

```
    temp_e <= "0000000";
```

```
when "001" =>
```

```
    u <= r_reg xor "0010000";
```

```
    temp_e <= "0010000";
```

```
when "010" =>
```

```
    u <= r_reg xor "0100000";
```

```
    temp_e <= "0100000";
```

```
when "011" =>
```

```
    u <= r_reg xor "0000010";
```

```
    temp_e <= "0000010";
```

```
when "100" =>
```

```
    u <= r_reg xor "1000000";
```

```
    temp_e <= "1000000";
```

```
when "101" =>
```

```
    u <= r_reg xor "0000100";
```

```
    temp_e <= "0000100";
```

```

when "110" =>
    u <= r_reg xor "0001000";
    temp_e <= "0001000";
-- s = 111
when others =>
    u <= r_reg xor "0000001";
    temp_e <= "0000001";
end case;
end process;

temp_count <= std_logic_vector(count_reg);
temp_s <= s;
end matrix_arch_7_4;

architecture cyclic_arch_7_4 of hamming_decoder_7_4 is
    signal count_reg, count_next: unsigned(0 to COUNT_WIDTH-1);
    signal s_reg, s_next: std_logic_vector(0 to 2);
    signal buf_reg, buf_next: std_logic_vector(0 to 6);
begin
    -- register
    process(clk, reset)
    begin
        if(reset = '1') then
            count_reg <= "10100";
            s_reg <= (others => '0');
            buf_reg <= (others => '0');
        elsif(clk'event and clk = '1') then
            count_reg <= count_next;
            s_reg <= s_next;
            buf_reg <= buf_next;
        end if;
    end process;

    -- next state and output logic
    count_next <= (others => '0') when count_reg = 20 else count_reg + 1;

    process(count_reg)
    begin
        -- simultaneously shift the received bits into the LFSR for calculating the syndrome
        -- and into the buffer register
        if(count_reg < 7) then
            buf_next <= r & buf_reg(0 to 5);
            s_next(0) <= s_reg(2) xor r;
            s_next(1) <= s_reg(0) xor s_reg(2);
            s_next(2) <= s_reg(1);
            r_out <= '0';
        -- calculate the output of the AND gate and feed it back into the LFSR and into the buffer register
        elsif(count_reg >= 7 and count_reg < 14) then
            buf_next <= (buf_reg(6) xor (s_reg(0) and not s_reg(1) and s_reg(2))) & buf_reg(0 to 5);
            s_next(0) <= s_reg(2) xor (s_reg(0) and not s_reg(1) and s_reg(2));
            s_next(1) <= s_reg(0) xor s_reg(2);
            s_next(2) <= s_reg(1);
            r_out <= '0';
        -- shift out the corrected codeword stored in the buffer register onto the channel
        else

```

```

    buf_next <= '0' & buf_reg(0 to 5);
    s_next <= (others => '0');
    r_out <= buf_reg(6);
end if;
end process;

```

```

temp_count <= std_logic_vector(count_reg);
temp_s <= s_reg;
end cyclic_arch_7_4;

```

-- (15, 11) hamming decoder

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

```

```

entity hamming_decoder_15_11 is
generic(
    COUNT_WIDTH: integer:= 4
);
port(
    clk, reset: in std_logic;
    r: in std_logic;
    temp_e: out std_logic_vector(0 to 14);
    temp_count: out std_logic_vector(0 to COUNT_WIDTH-1);
    temp_s: out std_logic_vector(0 to 3);
    r_out: out std_logic;
    u: out std_logic_vector(0 to 14)
);
end hamming_decoder_15_11;

```

```

architecture matrix_arch_15_11 of hamming_decoder_15_11 is
    signal r_reg, r_next: std_logic_vector(0 to 14);
    signal s: std_logic_vector(0 to 3);
    signal count_reg, count_next: unsigned(0 to COUNT_WIDTH-1);
begin
    -- register
    process(clk, reset)
    begin
        if(reset = '1') then
            r_reg <= (others => '0');
            count_reg <= "1110";
        elsif(clk'event and clk='1') then
            r_reg <= r_next;
            count_reg <= count_next;
        end if;
    end process;

```

```

    -- next state and output logic
    count_next <= (others => '0') when count_reg = 14 else count_reg + 1;
    r_next <= r & r_reg(0 to 13);

```

```

    -- calculate the syndrome
    s(0) <= r_reg(0) xor r_reg(4) xor r_reg(7) xor r_reg(8) xor r_reg(10) xor r_reg(12) xor r_reg(13) xor r_reg(14);
    s(1) <= r_reg(1) xor r_reg(4) xor r_reg(5) xor r_reg(7) xor r_reg(9) xor r_reg(10) xor r_reg(11) xor r_reg(12);

```

```
s(2) <= r_reg(2) xor r_reg(5) xor r_reg(6) xor r_reg(8) xor r_reg(10) xor r_reg(11) xor r_reg(12) xor r_reg(13);
s(3) <= r_reg(3) xor r_reg(6) xor r_reg(7) xor r_reg(9) xor r_reg(11) xor r_reg(12) xor r_reg(13) xor r_reg(14);
```

```
-- decode the syndrome to get the error pattern
-- add the error pattern to the received codeword to get the correct codeword
```

```
process(s, r_reg)
begin
  case(s) is
    when "0000" =>
      u <= r_reg xor "0000000000000000";
      temp_e <= "0000000000000000";
    when "0001" =>
      u <= r_reg xor "0001000000000000";
      temp_e <= "0001000000000000";
    when "0010" =>
      u <= r_reg xor "0010000000000000";
      temp_e <= "0010000000000000";
    when "0011" =>
      u <= r_reg xor "0000001000000000";
      temp_e <= "0000001000000000";
    when "0100" =>
      u <= r_reg xor "0100000000000000";
      temp_e <= "0100000000000000";
    when "0101" =>
      u <= r_reg xor "0000000001000000";
      temp_e <= "0000000001000000";
    when "0110" =>
      u <= r_reg xor "0000010000000000";
      temp_e <= "0000010000000000";
    when "0111" =>
      u <= r_reg xor "0000000000010000";
      temp_e <= "0000000000010000";
    when "1000" =>
      u <= r_reg xor "1000000000000000";
      temp_e <= "1000000000000000";
    when "1001" =>
      u <= r_reg xor "0000000000000001";
      temp_e <= "0000000000000001";
    when "1010" =>
      u <= r_reg xor "0000000010000000";
      temp_e <= "0000000010000000";
    when "1011" =>
      u <= r_reg xor "0000000000000010";
      temp_e <= "0000000000000010";
    when "1100" =>
      u <= r_reg xor "0000100000000000";
      temp_e <= "0000100000000000";
    when "1101" =>
      u <= r_reg xor "0000000100000000";
      temp_e <= "0000000100000000";
    when "1110" =>
      u <= r_reg xor "0000000000100000";
      temp_e <= "0000000000100000";
    when others =>
      u <= r_reg xor "0000000000001000";
```

```

    temp_e <= "000000000000100";
end case;
end process;

temp_count <= std_logic_vector(count_reg);
temp_s <= s;
end matrix_arch_15_11;

architecture cyclic_arch_15_11 of hamming_decoder_15_11 is
    signal count_reg, count_next: unsigned(0 to COUNT_WIDTH-1);
    signal s_reg, s_next: std_logic_vector(0 to 3);
    signal buf_reg, buf_next: std_logic_vector(0 to 14);
begin
    -- register
    process(clk, reset)
    begin
        if(reset = '1') then
            count_reg <= "101100";
            s_reg <= (others => '0');
            buf_reg <= (others => '0');
        elsif(clk'event and clk = '1') then
            count_reg <= count_next;
            s_reg <= s_next;
            buf_reg <= buf_next;
        end if;
    end process;

    -- next state and output logic
    count_next <= (others => '0') when count_reg = 44 else count_reg + 1;

    process(count_reg)
    begin
        -- simultaneously shift the received bits into the LFSR for calculating the syndrome
        -- and into the buffer register
        if(count_reg < 15) then
            buf_next <= r & buf_reg(0 to 13);
            s_next(0) <= s_reg(3) xor r;
            s_next(1) <= s_reg(0) xor s_reg(3);
            s_next(2) <= s_reg(1);
            s_next(3) <= s_reg(2);
            r_out <= '0';
            -- calculate the output of the AND gate and feed it back into the LFSR and into the buffer register
        elsif(count_reg >= 15 and count_reg < 30) then
            buf_next <= (buf_reg(14) xor (s_reg(0) and not s_reg(1) and not s_reg(2) and s_reg(3))) & buf_reg(0 to 13);
            s_next(0) <= s_reg(3) xor (s_reg(0) and not s_reg(1) and not s_reg(2) and s_reg(3));
            s_next(1) <= s_reg(0) xor s_reg(3);
            s_next(2) <= s_reg(1);
            s_next(3) <= s_reg(2);
            r_out <= '0';
            -- shift out the corrected codeword stored in the buffer register onto the channel
        else
            buf_next <= '0' & buf_reg(0 to 13);
            s_next <= (others => '0');
            r_out <= buf_reg(14);
        end if;
    end process;
end architecture cyclic_arch_15_11;

```

```
end process;
```

```
temp_count <= std_logic_vector(count_reg);
```

```
temp_s <= s_reg;
```

```
u <= buf_reg;
```

```
end cyclic_arch_15_11;
```



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity hamming_encoder_7_4_tb is
end hamming_encoder_7_4_tb;
```

```
architecture matrix_arch_7_4_tb of hamming_encoder_7_4_tb is
```

```
constant T: time:= 20ns;
signal test_clk, test_reset: std_logic;
signal test_m, test_u: std_logic;
signal test_temp_count: std_logic_vector(0 to 2);
signal test_temp_m_reg: std_logic_vector(0 to 3);
signal test_temp_p_reg: std_logic_vector(0 to 2);
```

```
begin
-- instantiate the device under test
--dut: entity work.hamming_encoder_7_4(matrix_arch_7_4)
--port map(clk => test_clk, reset => test_reset, m => test_m, temp_count => test_temp_count, temp_m_reg => test_
temp_m_reg, temp_p_reg => test_temp_p_reg, u => test_u);
```

```

dut: entity work.hamming_encoder_7_4(cyclic_arch_7_4)
port map(clk => test_clk, reset => test_reset, m => test_m, temp_count => test_temp_count, temp_p_reg => test_te
mp_p_reg, u => test_u);
```

```
-- create a clock signal that runs forever
```

```
process
begin
test_clk <= '0';
wait for T/2;
test_clk <= '1';
wait for T/2;
end process;
```

```
-- toggle the reset to clear all registers of the circuit before operation
```

```
test_reset <= '1', '0' after T/2;
```

```
-- other stimuli
```

```
process
begin
-- m = 1001
-- first message vector
-- input message bit 4
test_m <= '1';
wait until falling_edge(test_clk);
-- input message bit 3
test_m <= '0';
wait until falling_edge(test_clk);
-- input message bit 2
test_m <= '0';
wait until falling_edge(test_clk);
-- input message bit 1
test_m <= '1';
wait until falling_edge(test_clk);
```

```
test_m <= '0';
for i in 0 to 2 loop
```

```

    wait until falling_edge(test_clk);
end loop;
wait until rising_edge(test_clk);

-- terminate the simulation
assert false
    report "Simulation Completed"
    severity failure;
end process;
end matrix_arch_7_4_tb;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hamming_encoder_15_11_tb is
end hamming_encoder_15_11_tb;

architecture matrix_arch_15_11_tb of hamming_encoder_15_11_tb is
    constant T: time:= 20ns;
    signal test_clk, test_reset: std_logic;
    signal test_m, test_u: std_logic;
    signal test_temp_count: std_logic_vector(0 to 3);
    signal test_temp_m_reg: std_logic_vector(0 to 10);
    signal test_temp_p_reg: std_logic_vector(0 to 3);
begin
    -- instantiate the device under test
    --dut: entity work.hamming_encoder_15_11(matrix_arch_15_11)
    --port map(clk => test_clk, reset => test_reset, m => test_m, temp_count => test_temp_count, temp_m_reg => test_
    temp_m_reg, temp_p_reg => test_temp_p_reg, u => test_u);

    dut: entity work.hamming_encoder_15_11(cyclic_arch_15_11)
    port map(clk => test_clk, reset => test_reset, m => test_m, temp_count => test_temp_count, temp_p_reg => test_te
    mp_p_reg, u => test_u);

    -- create a clock signal that runs forever
    process
    begin
        test_clk <= '0';
        wait for T/2;
        test_clk <= '1';
        wait for T/2;
    end process;

    -- toggle the reset to clear all registers of the circuit before operation
    test_reset <= '1', '0' after T/2;

    -- other stimuli
    process
    begin
        -- m = 10110011101
        -- input message bit 11
        test_m <= '1';
        wait until falling_edge(test_clk);
        -- input message bit 10
        test_m <= '0';
    end process;
end architecture matrix_arch_15_11_tb;

```

```

wait until falling_edge(test_clk);
-- input message bit 9
test_m <= '1';
wait until falling_edge(test_clk);
-- input message bit 8
test_m <= '1';
wait until falling_edge(test_clk);
-- input message bit 7
test_m <= '1';
wait until falling_edge(test_clk);
-- input message bit 6
test_m <= '0';
wait until falling_edge(test_clk);
-- input message bit 5
test_m <= '0';
wait until falling_edge(test_clk);
-- input message bit 4
test_m <= '1';
wait until falling_edge(test_clk);
-- input message bit 3
test_m <= '1';
wait until falling_edge(test_clk);
-- input message bit 2
test_m <= '0';
wait until falling_edge(test_clk);
-- input message bit 1
test_m <= '1';
wait until falling_edge(test_clk);

test_m <= '0';
for i in 0 to 3 loop
    wait until falling_edge(test_clk);
end loop;
wait until rising_edge(test_clk);

-- terminate the simulation
assert false
    report "Simulation Completed"
    severity failure;
end process;
end matrix_arch_15_11_tb;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

entity hamming_decoder_matrix_7_4_tb is
end hamming_decoder_matrix_7_4_tb;

```

```

architecture matrix_arch_7_4_tb of hamming_decoder_matrix_7_4_tb is
    constant T: time:= 20ns;
    constant TEST_COUNT_WIDTH: integer:= 3;
    signal test_clk, test_reset: std_logic;
    signal test_r: std_logic;
    signal test_temp_count: std_logic_vector(0 to TEST_COUNT_WIDTH-1);
    signal test_temp_e: std_logic_vector(0 to 6);

```

```

signal test_temp_s: std_logic_vector(0 to 2);
signal test_u: std_logic_vector(0 to 6);
begin
-- instantiate the device under test
dut: entity work.hamming_decoder_7_4(matrix_arch_7_4)
generic map(COUNT_WIDTH => TEST_COUNT_WIDTH)
port map(clk => test_clk, reset => test_reset, r => test_r, temp_e => test_temp_e, temp_count => test_temp_count, t
emp_s => test_temp_s, u => test_u);

-- create a clock signal that runs forever
process
begin
test_clk <= '0';
wait for T/2;
test_clk <= '1';
wait for T/2;
end process;

-- toggle the reset to clear all registers of the circuit before operation
test_reset <= '1', '0' after T/2;

-- other stimuli
process
begin
-- original message: u = 0111001
-- corrupted message: r = 0101001
-- input received codeword bit 7
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 6
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 5
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 4
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 3
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 2
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 1
test_r <= '0';
wait until falling_edge(test_clk);
wait until rising_edge(test_clk);

-- terminate the simulation
assert false
report "Simulation Completed"
severity failure;
end process;
end matrix_arch_7_4_tb;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hamming_decoder_7_4_cyclic_tb is
end hamming_decoder_7_4_cyclic_tb;

architecture cyclic_arch_7_4_tb of hamming_decoder_7_4_cyclic_tb is
constant T: time:= 20ns;
constant TEST_COUNT_WIDTH: integer:= 5;
signal test_clk, test_reset: std_logic;
signal test_r: std_logic;
signal test_temp_count: std_logic_vector(0 to TEST_COUNT_WIDTH-1);
signal test_temp_e: std_logic_vector(0 to 6);
signal test_temp_s: std_logic_vector(0 to 2);
signal test_r_out: std_logic;
signal test_u: std_logic_vector(0 to 6);
begin
-- instantiate the device under test
dut: entity work.hamming_decoder_7_4(cyclic_arch_7_4)
generic map(COUNT_WIDTH => TEST_COUNT_WIDTH)
port map(clk => test_clk, reset => test_reset, r => test_r, temp_e => test_temp_e, temp_count => test_temp_count, t
emp_s => test_temp_s, r_out => test_r_out, u => test_u);

-- create a clock signal that runs forever
process
begin
test_clk <= '0';
wait for T/2;
test_clk <= '1';
wait for T/2;
end process;

-- toggle the reset to clear all registers of the circuit before operation
test_reset <= '1', '0' after T/2;

-- other stimuli
process
begin
-- original message: u = 0111001
-- corrupted message: r = 0101001
-- input received codeword bit 7
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 6
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 5
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 4
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 3
test_r <= '0';

```

```

wait until falling_edge(test_clk);
-- input received codeword bit 2
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 1
test_r <= '0';
wait until falling_edge(test_clk);

```

```

-- wait for 14 more clock cycles
test_r <= '0';
for i in 1 to 14 loop
    wait until falling_edge(test_clk);
end loop;
wait until rising_edge(test_clk);

```

```

-- terminate the simulation
assert false
    report "Simulation Completed"
    severity failure;
end process;
end cyclic_arch_7_4_tb;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

entity hamming_decoder_15_11_matrix_tb is
end hamming_decoder_15_11_matrix_tb;

```

```

architecture matrix_arch_15_11_tb of hamming_decoder_15_11_matrix_tb is
    constant T: time:= 20ns;

```

```

-- must change this constant when switching between matrix and cyclic implementations

```

```

    constant TEST_COUNT_WIDTH: integer:= 4;

```

```

    signal test_clk, test_reset: std_logic;

```

```

    signal test_r: std_logic;

```

```

    signal test_temp_count: std_logic_vector(0 to TEST_COUNT_WIDTH-1);

```

```

    signal test_temp_e: std_logic_vector(0 to 14);

```

```

    signal test_temp_s: std_logic_vector(0 to 3);

```

```

    signal test_r_out: std_logic;

```

```

    signal test_u: std_logic_vector(0 to 14);

```

```

begin

```

```

-- instantiate the device under test

```

```

    dut: entity work.hamming_decoder_15_11(matrix_arch_15_11)

```

```

    generic map(COUNT_WIDTH => TEST_COUNT_WIDTH)

```

```

    port map(clk => test_clk, reset => test_reset, r => test_r, temp_e => test_temp_e, temp_count => test_temp_count, t
emp_s => test_temp_s, r_out => test_r_out, u => test_u);

```

```

-- create a clock signal that runs forever

```

```

process

```

```

begin

```

```

    test_clk <= '0';

```

```

    wait for T/2;

```

```

    test_clk <= '1';

```

```

    wait for T/2;

```

```

end process;

```

-- toggle the reset to clear all registers of the circuit before operation

test\_reset <= '1', '0' after T/2;

-- other stimuli

process

begin

-- original message: u = 110110110011101

-- corrupted message: r = 110110110010101

-- input received codeword bit 15

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 14

test\_r <= '0';

wait until falling\_edge(test\_clk);

-- input received codeword bit 13

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 12

test\_r <= '0';

wait until falling\_edge(test\_clk);

-- input received codeword bit 11

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 10

test\_r <= '0';

wait until falling\_edge(test\_clk);

-- input received codeword bit 9

test\_r <= '0';

wait until falling\_edge(test\_clk);

-- input received codeword bit 8

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 7

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 6

test\_r <= '0';

wait until falling\_edge(test\_clk);

-- input received codeword bit 5

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 4

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 3

test\_r <= '0';

wait until falling\_edge(test\_clk);

-- input received codeword bit 2

test\_r <= '1';

wait until falling\_edge(test\_clk);

-- input received codeword bit 1

test\_r <= '1';

wait until falling\_edge(test\_clk);

wait until rising\_edge(test\_clk);

```

-- terminate the simulation
assert false
report "Simulation Completed"
severity failure;
end process;
end matrix_arch_15_11_tb;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hamming_decoder_15_11_cyclic_tb is
end hamming_decoder_15_11_cyclic_tb;

architecture cyclic_arch_15_11_tb of hamming_decoder_15_11_cyclic_tb is
constant T: time:= 20ns;
-- must change this constant when switching between matrix and cyclic implementations
constant TEST_COUNT_WIDTH: integer:= 6;
signal test_clk, test_reset: std_logic;
signal test_r: std_logic;
signal test_temp_count: std_logic_vector(0 to TEST_COUNT_WIDTH-1);
signal test_temp_e: std_logic_vector(0 to 14);
signal test_temp_s: std_logic_vector(0 to 3);
signal test_r_out: std_logic;
signal test_u: std_logic_vector(0 to 14);
begin
-- instantiate the device under test
dut: entity work.hamming_decoder_15_11(cyclic_arch_15_11)
generic map(COUNT_WIDTH => TEST_COUNT_WIDTH)
port map(clk => test_clk, reset => test_reset, r => test_r, temp_e => test_temp_e, temp_count => test_temp_count, t
emp_s => test_temp_s, r_out => test_r_out, u => test_u);

-- create a clock signal that runs forever
process
begin
test_clk <= '0';
wait for T/2;
test_clk <= '1';
wait for T/2;
end process;

-- toggle the reset to clear all registers of the circuit before operation
test_reset <= '1', '0' after T/2;

-- other stimuli
process
begin
-- original message: u = 110110110011101
-- corrupted message: r = 110110110010101
-- input received codeword bit 15
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 14
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 13

```



```

test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 12
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 11
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 10
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 9
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 8
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 7
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 6
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 5
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 4
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 3
test_r <= '0';
wait until falling_edge(test_clk);
-- input received codeword bit 2
test_r <= '1';
wait until falling_edge(test_clk);
-- input received codeword bit 1
test_r <= '1';
wait until falling_edge(test_clk);

-- wait for 30 more clock cycles
test_r <= '0';
for i in 1 to 30 loop
    wait until falling_edge(test_clk);
end loop;
wait until rising_edge(test_clk);

-- terminate the simulation
assert false
    report "Simulation Completed"
    severity failure;
end process;
end cyclic_arch_15_11_tb;

```