Hamming (7, 4) Code Encoder

ECE 2330

By Liam Timmins

13 December 2022

Table of Contents

Table of Figures

Table of Tables

Introduction

The goal of this assignment was to develop a (7, 4) Hamming code encoder that returns a specific combination of three parity bits using four sequentially delivered data bits. The purpose of said device would be to detect when a bit in the codeword becomes corrupted. Corruptions, where bits of data erroneously change, may occur when information is transmitted in the presence of noise that inhibits.

Assuming a collection of bits would be represented in polynomial form as $b_{n-1}x^{n-1}+b_{n-2}x^{n-2}+...+b_1x+b_0$, a generator matrix can be created for a Hamming (7, 4) code encoder which, when multiplied by a matrix of the four polynomial data bits, will return a matrix that has both a specific set of parity bits as well as the initial data bits. The generator matrix can be created with the following:

$$G(x) = \begin{bmatrix} g(x) \\ g(x)x \\ g(x)(x^2+1) \\ g(x)(x^3+x+1) \end{bmatrix} = \begin{bmatrix} 1\ 1\ 0\ 1\ 0\ 0\ 0 \\ 0\ 1\ 1\ 0\ 1\ 0\ 0 \\ 1\ 1\ 1\ 0\ 0\ 1\ 0 \\ 1\ 0\ 1\ 0\ 0\ 0\ 1 \end{bmatrix}$$

The four rightmost columns of this matrix are an identity matrix, meaning that these bits will return what they are multiplied by, in this case the four initial data bits. The other bits of this system are the parity bits, and will produce a specific result for every combination of data bits. As these two types of bits are easy to distinguish from each other, this is a separable codeword. The product of the data bits and the generator matrix is shown below:

$$V(x) = (d_0+d_2+d_3) + (d_0+d_1+d_2)x + (d_1+d_2+d_3)x^2 + (d_0+d_1x+d_2x^2+d_3x^3)x^3$$

With the power of x representing the location of the bit and the coefficient representing the value of the bit, the parity bits are able to be calculated for each combination of data bits as the first three bits in the resulting sequence, assuming the sequence is $(g_0,g_1,g_2,g_3,g_4,g_5,g_6)$.

Determining Logic

Table 1: Hamming (7,4) Code Truth Table

d3	d2	d1	d0	p2	p1	p0
0	0	0	0	0	0	0
0	0	0	1	0	1	1
0	0	1	0	1	1	0
0	0	1	1	1	0	1
0	1	0	0	1	1	1
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	1	1
1	0	1	1	0	0	0
1	1	0	0	0	1	0
1	1	0	1	0	0	1
1	1	1	0	1	0	0
1	1	1	1	1	1	1

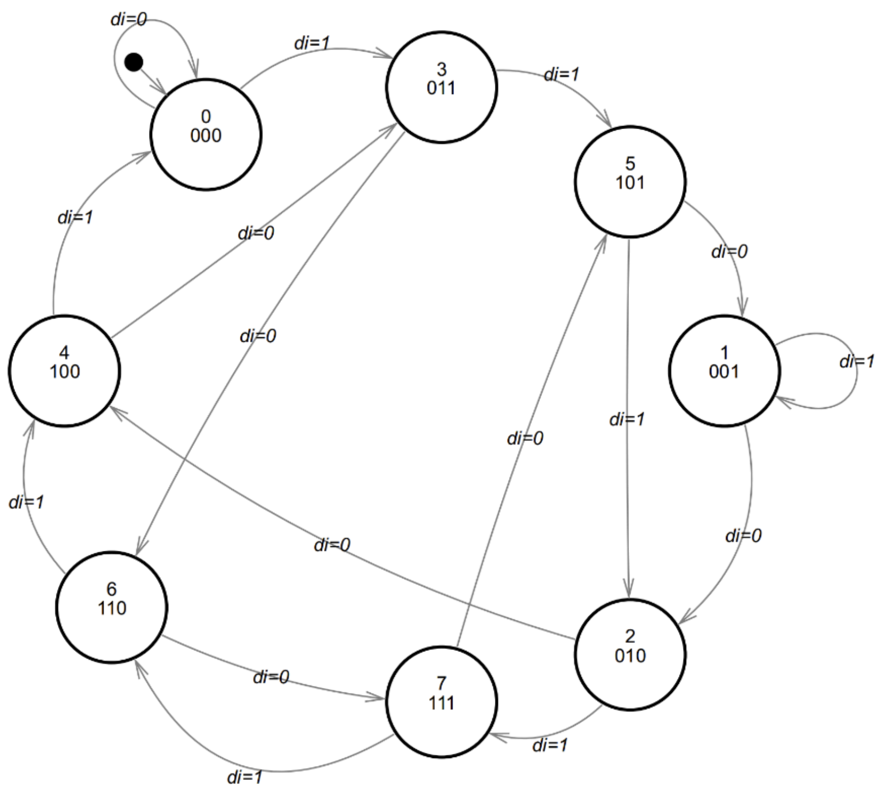


Figure 1. State Transition Diagram

Table 1 above depicts the expected outputs of the hamming (7,4) encoder given every possible combination of di inputs. This table was given alongside the state transition diagram in figure 1 in the problem assignment.

Table 2: di Next State Table

di	p2	p1	p0	p2next	p1next	p0next
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	1	1
0	1	0	1	0	0	1
0	1	1	0	1	1	1
0	1	1	1	1	0	1
1	0	0	0	0	1	1
1	0	0	1	0	0	1
1	0	1	0	1	1	1
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	0	1	0	1	0
1	1	1	0	1	0	0
1	1	1	1	1	1	0

Table 2 was generated with the use of the hamming (7, 4) encoder truth table and the state transition diagram in table 1 and figure 1 respectively. This next state table uses the current state data bit, represented by di, and the current state parity bits, represented by p2, p1, and p0, to determine the next set of parity bits. Starting with the parity bits 000, this process will repeat for all four data bits before displaying the expected output in table 1.

Karnaugh Maps

The following Karnaugh maps in figures 2, 3, and 4 use the next state table in table 2 to best organize the data such that equations can be developed for each of the three next state parity bits using the three current state parity bits and the current di bit as inputs. Following each figure, work is shown depicting how the logic of the Karnaugh maps is simplified to create more efficient hardware further in the development process.

$(p1, p0)$ / $(di, p2)$	00	01	11	10
00	0 (0)	0 (1)	0 (3)	0 (2)
01	1 (4)	1 (5)	1 (7)	1 (6)
11	0 (12)	0 (13)	0 (15)	0 (14)
10	1 (8)	1 (9)	1 (11)	1 (10)

Figure 2. p_0^{next} Karnaugh Map

p_0^{next} = (di' and p2) or (di and p2')

p_0^{next} = <u>di XOR p2</u>

$(p1, p0)$ \ $(di, p2)$	00	01	11	10
00	0 $_{0}$	1 $_{1}$	1 $_{3}$	0 $_{2}$
01	1 $_{4}$	0 $_{5}$	0 $_{7}$	1 $_{6}$
11	0 $_{12}$	1 $_{13}$	1 $_{15}$	0 $_{14}$
10	1 $_{8}$	0 $_{9}$	0 $_{11}$	1 $_{10}$

Figure 3. p_1^{next} Karnaugh Map

p_1^{next} = (di' and p2' and p0) or (di' and p2 and p0') or (di and p2 and p0) or (di and p2' and p0')

p_1^{next} = p0(di'*p2' + di*p2) + p0'(di'*p2 + di*p2')

p_1^{next} = p0 AND (di XNOR p2) OR p0' AND (di XOR p2)

p_1^{next} = <u>p0 XOR di XOR p2</u>

$(p1, p0)$ \ $(di, p2)$	00	01	11	10
00	0 $_{0}$	0 $_{1}$	1 $_{3}$	1 $_{2}$
01	0 $_{4}$	0 $_{5}$	1 $_{7}$	1 $_{6}$
11	0 $_{12}$	0 $_{13}$	1 $_{15}$	1 $_{14}$
10	0 $_{8}$	0 $_{9}$	1 $_{11}$	1 $_{10}$

Figure 4. p_2^{next} Karnaugh Map

p_2^{next} = <u>(p1)</u>

Simulation Results

Following the simplification of the logic for each next state parity bit in the Karnaugh maps, a block diagram was generated in figure 5 below.

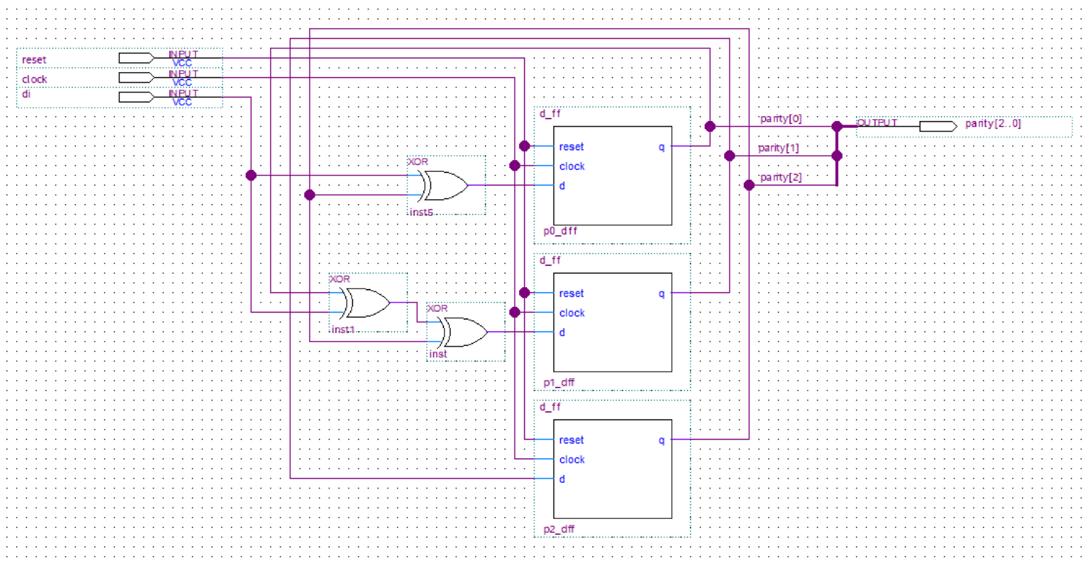


Figure 5. Block Diagram of Hamming (7, 4) Code Encoder

This diagram shows the three inputs being taken in, reset, clock, and di, as well as the three parity bits as an output. The logic simplified above is implemented as the input into three d-flip flop circuits, each representing one next state parity bit. Following the clock's rising edge, the d-flip flop pushes the next state logic for each bit forward, becoming the new current state. This data is then used to determine new next state logic with the di input after the clock's falling edge. The output simply labels the output of each of the d-flip flops and combines them into one standard logic vector to be displayed. This block diagram was then used to automatically generate the VHDL code to be tested.

To verify the generated code, a testbench needed to be made which tested every possible input of data bits and their respective expected output parity bits. In this endeavor, the following testbench process was developed:

```
reset_signal <= '1';
wait for 1 ns;
reset_signal <= '0';
wait for 1 ns;

di_signal <= '0';
wait until falling_edge(clock_signal);
di_signal <= '0';
wait until falling_edge(clock_signal);
di_signal <= '0';
wait until falling_edge(clock_signal);
di_signal <= '0';
wait until rising_edge(clock_signal);
report "Current data bits: 0000";

wait for 10 ns;
assert (parity_signal = "000") report "Unexpected parity achieved for data bits = 0000";
assert (parity_signal /= "000") report "Expected parity achieved for data bits = 0000";

wait until falling_edge(clock_signal);
```

Before the testbench process executes, the reset and clock inputs are assigned a value of ‘0’, and a clock process is created where the clock signal switches values every 50 ns. After the testbench process begins, the above code triggers a reset of the d-flip flops, resulting in an initial output of 0 for all parity bits. Next, the di_signal is changed after each clock cycle falling edge to represent each of the four input data bits. While this occurs, the encoder updates the parity bits after every rising clock edge. Following all four data bit assignments, the current data bits are declared, and the parity bits are compared with their corresponding expected values. The process then waits for the next falling clock edge before repeating the process for the other 15 potential data bits, with each being adjusted for their respective data and expected parity bits.

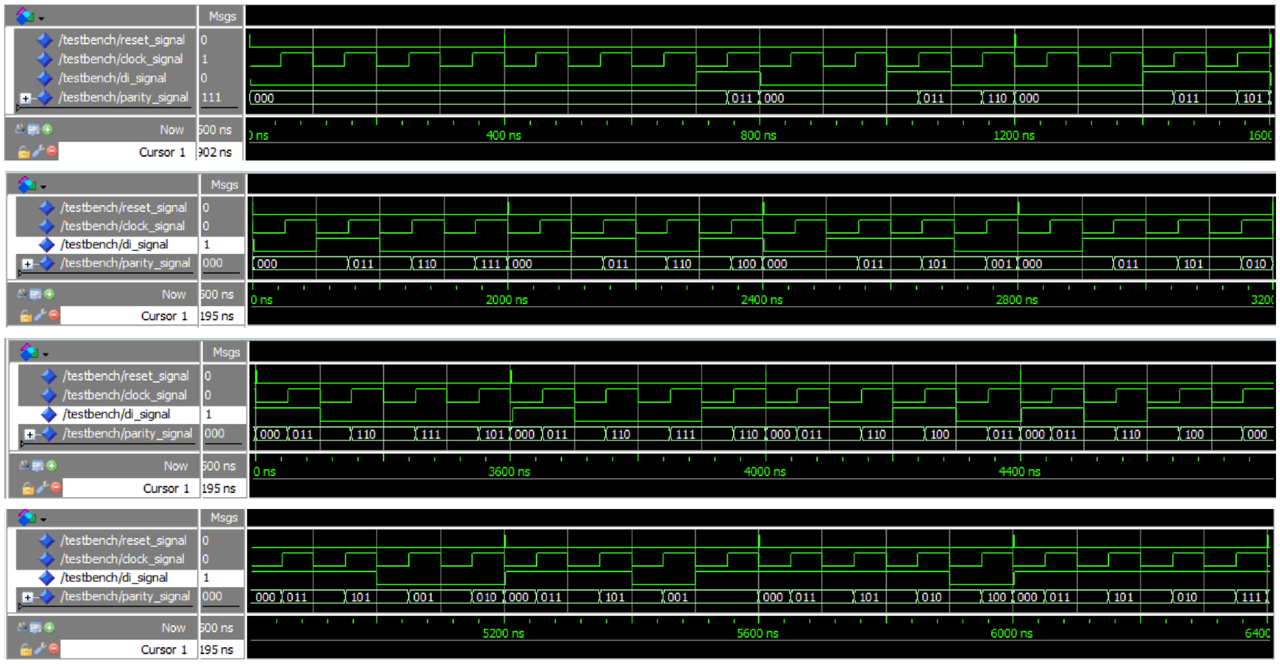
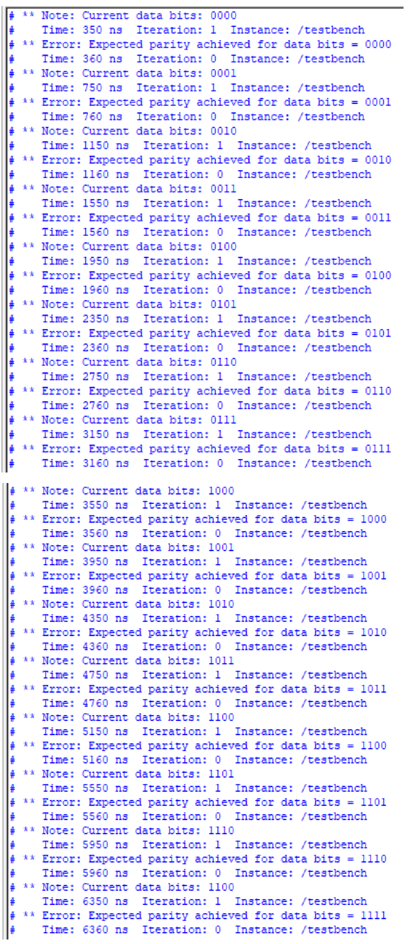


Figure 6. Signal Inputs and Outputs of Hamming (7, 4) Code Encoder



```
# ** Note: Current data bits: 0000
#    Time: 350 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0000
#    Time: 360 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0001
#    Time: 750 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0001
#    Time: 760 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0010
#    Time: 1150 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0010
#    Time: 1160 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0011
#    Time: 1550 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0011
#    Time: 1560 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0100
#    Time: 1950 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0100
#    Time: 1960 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0101
#    Time: 2350 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0101
#    Time: 2360 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0110
#    Time: 2750 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0110
#    Time: 2760 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 0111
#    Time: 3150 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 0111
#    Time: 3160 ns  Iteration: 0  Instance: /testbench

# ** Note: Current data bits: 1000
#    Time: 3550 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1000
#    Time: 3560 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1001
#    Time: 3950 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1001
#    Time: 3960 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1010
#    Time: 4350 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1010
#    Time: 4360 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1011
#    Time: 4750 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1011
#    Time: 4760 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1100
#    Time: 5150 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1100
#    Time: 5160 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1101
#    Time: 5550 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1101
#    Time: 5560 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1110
#    Time: 5950 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1110
#    Time: 5960 ns  Iteration: 0  Instance: /testbench
# ** Note: Current data bits: 1100
#    Time: 6350 ns  Iteration: 1  Instance: /testbench
# ** Error: Expected parity achieved for data bits = 1111
#    Time: 6360 ns  Iteration: 0  Instance: /testbench
```

Figure 7. Transcript with Assert Statements for Verification of Hamming (7, 4) Code Encoder

After simulating the testbench, figures 6 and 7 above are produced. Figure 6 shows the transition of the parity bits for every combination of data inputs. The tested data bits are shown in order from 0000 to 1111, with the reset bit designating each new test case. In figure 7, every combination of data bit inputs returns their expected output parity bits. As such, it can be shown that the Hamming (7, 4) code encoder operates as expected.

Summary

Given a set of four data bits, a corresponding set of three parity bits were calculated for the purpose of detecting any erroneous corruption during the transmission of data. Hardware and a testbench were developed in order to calculate these parity bits and verify they operated as expected for every possible input of data bits. Considering that both the simulated values generated in figure 6 and the assert statements in figure 7 verify this, we can assume that the encoder will operate as expected within any larger system should it be implemented within those contexts.