Lab 7: Hamming Code

ECE 211: Digital Circuits I — Spring 2025

Abstract. This lab assignment will introduce you to error-detecting and error-correcting codes for data reliability. Error correction codes extend a data value with an extra bit, termed a ***parity bit***, that makes the total number of 1's in the word even (or odd). This parity bit is transmitted alongside the original data value. If the signal is damaged during transmission, the redundancy provided by the parity bits can be used to detect and correct these errors! In this lab, you will design and build a full Hamming code encoder and decoder that is able to detect and correct single-bit errors. You will use both structural and behavioral modeling in this lab, and use testbenches to validate your models.

Objectives.

- Understand the use of error detecting and error correcting codes for data reliability
- Use ***parity checks*** to add redundancy to data in order to detect and correct single errors
- Design and build a full Hamming code circuit using these concepts
- Test the operation of individual circuit components using testbenches
- Test the operation of the entire circuit using the input and output components on the Nexys A7

Materials and Equipment.

- Nexys A7-100T FPGA board with USB cable

Deliverables.

- In-person lab demonstration (per team)
- Lab report (per team)

Part 0: Set Up a Vivado Project

To begin the lab, we will set up a new Vivado project and configure the Nexys A7 board. You can reference Lab 0 for a more detailed description of these steps.

1. Create a Xilinx Vivado project and import the provided constraints file and SystemVerilog modules. This lab will use both simulation and design sources. The file **`lab07_top.sv`** should be uploaded as a design source. The file **`lab07_tb.sv`** should be uploaded as a simulation source. You will use this file as a testbench to write test cases for each module.

2. In this lab, you will primarily use Vivado's simulator to verify your design at each part, and then use the FPGA board at the end to demonstrate the entire circuit. If you would like, you can connect your FPGA board and open the hardware target at this point in the lab.

Part 1: Theory of Error Detection Codes

Data moving through or stored in a circuit may be unreliable! Consider a 4-bit binary number as an example. If any of the bits in the 4-bit word `0001` were erroneously changed, we could not detect this error since this bit flip will create another valid word (*e.g.,* `0000` or `0011` or `0101` or `1001`).

Different error-detecting and error-correcting codes have been proposed to address this problem. For example, we could extend our 4-bit word with an extra bit to make the number of 1's in the word even. With this design, we could detect any single-bit error by counting the number of 1's. If the number of 1's were odd, then we would know an error occurred! This added bit is called a ***parity bit***. (*Hint: What logic gate can we use to generate a parity bit?)*

Original Word	Word with Parity Bit
0001	00011

Received Word	Parity-Based Error Detection
00011	No, Even number of 1s
00010	Yes, Odd number of 1s
00001	Yes, Odd number of 1s
00111	Yes, Odd number of 1s
01011	Yes, Odd number of 1s
10011	Yes, Odd number of 1s

Table 1. Detecting Single-Bit Errors in the Message "0001" via a Parity Bit

A parity bit for the word `0001` would be `1`, giving the transmitted signal, `00011`, an even number of 1s. If this transmitted signal was received as `00001`, the number of 1's is counted to check for evenness. We would know that an error existed. We cannot, however, correct this error with this simple parity bit. We also cannot detect an error if there are *two* bit flips in the transmitted signal.

The principle of adding extra bits is known as adding ***redundancy***, defined as the total number of bits transmitted divided by the data bits. If the number of data bits is (n-1), then, for a single parity system, redundancy is n/(n-1).

In this lab, you will build a ***Hamming code*** encoder and decoder that can detect and correct single-bit errors in a 4-bit data signal. The data value is given as four bits `D3`, `D2`, `D1`, `D0`, where `D3` is the most significant bit. The Hamming code adds three parity bits, `P2`, `P1`, and `P0`, which depend on different combinations of the data bits.

Parity Bit	Based on Data Bits	Check Code
P2	D3, D2, D0	C
P1	D3, D1, D0	B
P0	D2, D1, D0	A

Table 2. Parity Bits for Hamming Code

The specific Hamming code we will use is given in Table 3. Our circuit will transmit binary-coded decimals (BCD), using four binary bits to represent the decimal numbers 0-9. Throughout each part of this lab, you will build each of the following components: (1) A ***Hamming code encoder*** that generates the three parity bits for `D3`, `D2`, `D1`, and `D0`; (2) An ***error position indicator*** that checks a received 7-bit signal and indicates the position of a flipped bit (if one exists); and (3) An ***error corrector*** that corrects a data bit given an error position signal.

	6	5	4	3	2	1	0	← *Bit Position*
Decimal Number	**P2**	**P1**	**D3**	**P0**	**D2**	**D1**	**D0**	← *Transmitted Signal with Interleaved Parity Bits*
0	0	0	0	0	0	0	0	
1	1	1	0	1	0	0	1	
2	0	1	0	1	0	1	0	
3	1	0	0	0	0	1	1	
4	1	0	0	1	1	0	0	
5	0	1	0	0	1	0	1	
6	1	1	0	0	1	1	0	
7	0	0	0	1	1	1	1	
8	1	1	1	0	0	0	0	
9	0	0	1	1	0	0	1	

Table 3. Hamming Code Representation for Decimal Numbers 0-9

Part 2: Design and Build a Parity Generator Module (Hamming Code Encoder)

The Hamming code encoding process is shown in Figure 1. The Parity Generator in this diagram is responsible for calculating parity bits `P0`, `P1`, and `P2`. The value for each parity bit is based on the combination of data bit values given in Table 2. For example, the first parity bit, `P2`, is generated by counting the 1's in data bits `D3`, `D2`, and `D0` and adding a 1 if the number of 1's within these three positions is odd or a 0 if the number is even. A similar process occurs for parity check bits `P1` and `P0`.

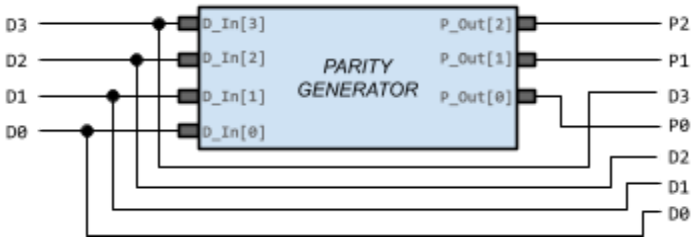

Figure 1. Parity Generator Module

3. Design and implement a parity generator module.
   a. Add a new design source file to your project for `parity_generator.sv`
   b. Your module should take a 4-bit vector input, **`D_In`**
   c. Your module should produce a 3-bit vector output, **`P_Out`**
   d. Design and implement the module to meet the description above (*Hint: The parity bit is 1 when there are an odd number of ones and 0 when there are an even number of ones. What single logic gate can we use to generate a parity bit?)*

4. Test your parity generator module using a testbench.
   a. In `lab07_tb.sv`, instantiate a parity generator
   b. Create a series of at least four unique test cases for your module. A single test is a specific set of input signals that you are looking to evaluate (*e.g.,* `0000`)
   c. Remember to add delays between each test case!
   d. Simulate your circuit and verify your circuit's operation using the Waveform Viewer or by printing messages to the Tcl Console.
   e. If you are stuck on extending `lab07_tb.sv` with tests for this module, refer to Lab 6 for more details about creating test cases!

5. Verify your test cases and record your testbench code and results before moving to the next part of this lab assignment.

Stop and Check: Using the Vivado Simulator, ensure that your module works correctly for at least four unique test cases.

Record: Screenshot the waveform for your test cases of your parity generator module to include in your Lab Report. Record your test cases and testbench code as well.

Part 3: Design and Build an Error Position Indicator Module

With our parity generator module, we can encode a 4-bit decimal value in Hamming code and transmit the resulting 7-bit signal. Because of the Hamming code representation, we can detect and correct any single-bit error in the transmitted value! In Part 3 of the lab, we will perform the first step of this decoding process—error detection.

To check whether an error occurred in the transmission process, the parity checks are performed on the received word, in reverse order. In other words, does `P2`, `D3`, `D2`, and `D0` have an even number of ones (giving a 1 for "correct") or an odd number of ones (giving a 0 for "incorrect"). The result of this check is named **`C`**, as listed in Table 2.

Each check code pinpoints the location of the error. Combining the check bits **`ABC`**, where **`A`**, **`B`**, and **`C`** are defined in Table 2, gives the position of the error. For example, if **`ABC`** = `110`, this indicates an error in parity bit `P2` in Table 1. If **`ABC`** = `010`, this indicates an error data bit `D2` in Table 1. If **`ABC`** = `111`, then there is no error.

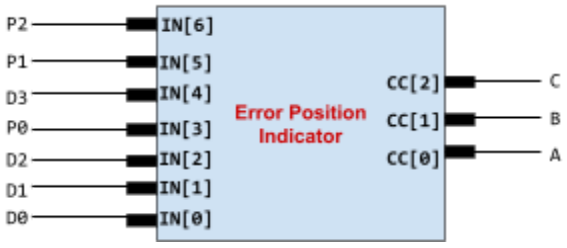

Figure 2. Error Position Indicator Module

Design and build the error position indicator module in SystemVerilog. A block diagram of this module is given above. The seven inputs to the error position indicator module represent the seven bits of the received word. The error position is indicated by the 3-bit output which corresponds to the binary value of the bit position of the bit in error. The outputs **A**, **B**, and **C**, are the check codes listed in Table 2.

6. Design and implement an error position indicator module.
   a. Add a new design source file to your project for `error_position_indicator.sv`
   b. Your module should take a 7-bit vector input, called **`IN`**
   c. Your module should produce a 3-bit vector output, called **`CC`** `(Check Code A, B, C)`
   d. Design and implement the module to meet the description above

7. Test your error position indicator module using a testbench.
   a. In `lab07_tb.sv`, instantiate an error position indicator
   b. Create a series of at least four unique test cases for your module. A single test is a specific set of input signals that you are looking to evaluate (*e.g.,* 0000101)
   c. Remember to add delays between each test case!
   d. Simulate your circuit and verify your circuit's operation using the Wave view or by printing messages to the Tcl Console
   e. If you are stuck on extending `lab07_tb.sv` with tests for this module, refer to Lab 6 for more details about creating test cases!

8. Verify your test cases and record your testbench code and results before moving to the next part of this lab assignment.

Stop and Check: Using the Vivado Simulator, ensure that your module works correctly for at least four unique test cases.

Record: Screenshot the waveform for your test cases of your error position indicator module to include in your Lab Report. Record your test cases and testbench code as well.

Part 4: Design and Build an Error Corrector Module

Using the error position indicator, we can correct the error in our received signal! In this part of the lab, you will design and build an error corrector module that corrects an error in the received word based on the output from the error position indicator module. The inputs to this module are the check codes (`A`, `B`, and `C`) generated from your error position indicator module module, and the data bits from the received signal. The four outputs of this module are `D3`, `D2`, `D1`, and `D0` which represent the 4 data bits with no errors. A block diagram of this module is given below.

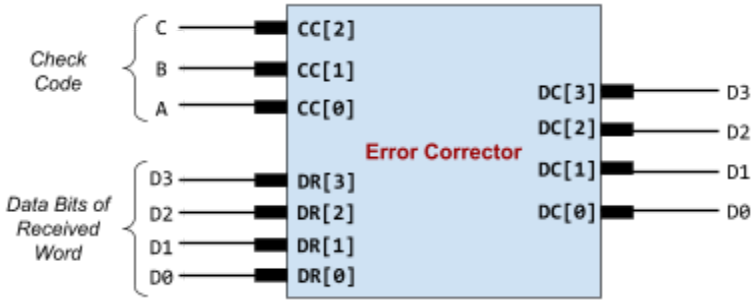

Figure 3. Error Corrector Module

You can implement your error correction module using a ***SystemVerilog if-statement***! Recall that if-statements must always be placed within an `always_comb` block. The following code snippet gives an example of how you might use an if-statement to implement the error correction functionality. The text shown in red in angled-brackets should be replaced with valid SystemVerilog expressions.

```
always_comb begin
      if (<Position Indicator is 0>) begin
           // Correct Bit 0 (D0) by flipping it
           DC[0] = ~DR[0];
      end
end
```

9. Design and implement an error corrector indicator module.
   a. Add a new design source file to your project for `error_corrector.sv`
   b. Your module should take a 3-bit vector for **`CC`** (check code) and a 4-bit vector for the received data, named **`DR`**
   c. Your module should produce the 4-bit vector of corrected data, named **`DC`**
   d. Using behavioral modeling, design and implement the module to meet the description above

10. Test your error corrector module using a testbench.
    a. In `lab07_tb.sv`, instantiate an error corrector
    b. Create a series of at least four unique test cases for your module. A single test is a specific set of input signals that you are looking to evaluate (*e.g.,* 0000101)
    c. Remember to add delays between each test case!

d. Simulate your circuit and verify your circuit's operation using the Wave view or by printing messages to the Tcl Console

11. Verify your test cases and record your testbench code and results before moving to the next part of this lab assignment.

Stop and Check: Using the Vivado Simulator, ensure that your module works correctly for at least four unique test cases.

Record: Screenshot the waveform for your test cases of your error correction module to include in your Lab Report. Record your test cases and testbench code as well.

Part 5: Connect the Circuit with an Error Injector Module

In the final part of this lab assignment, you will connect your Hamming code encoder and decoder modules together to test if they are working correctly. It is more interesting to see how our modules can correct errors in the transmitted signal. To test this, you will use a provided ***error injector*** module that flips a bit of the transmitted signal based on some additional inputs. This module is shown in orange in Figure 4. The error injector takes a one-hot 7-bit input signal and flips the bit in the corresponding position. For example, the input 0000001 would flip the least significant bit (`D0`) of the transmitted signal, whereas the input 0000010 would flip the next bit (`D1`). The entire configuration for the circuit is shown in Figure 4.

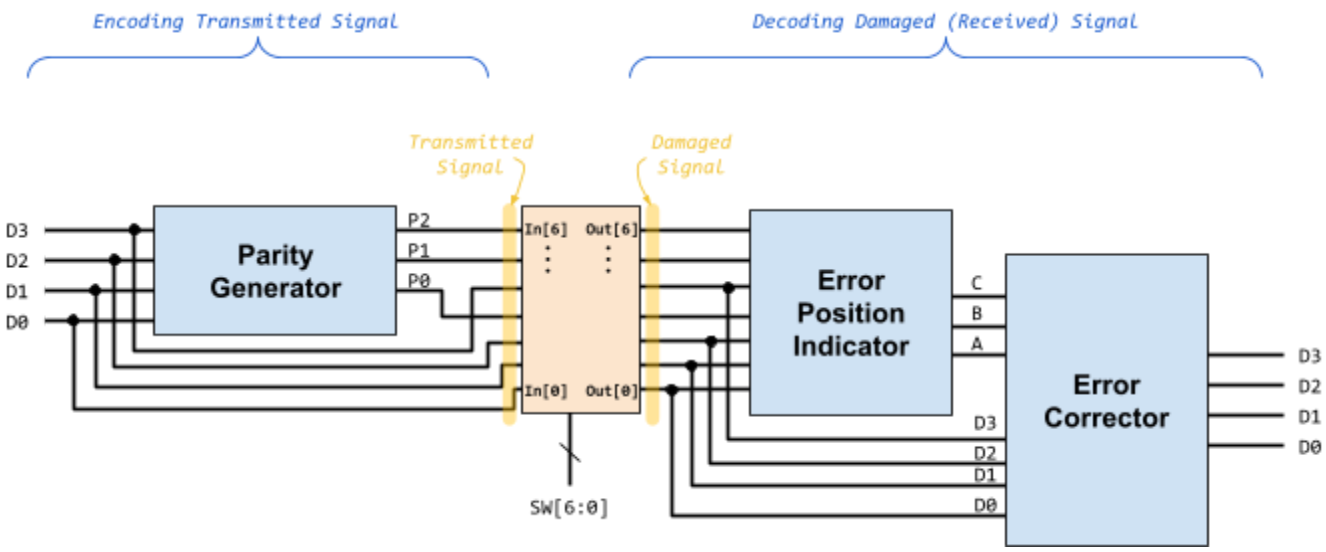

Figure 4. Configuration in Top-Level File, with Error Injector

In this step, you will synthesize your SystemVerilog code and implement it on the Nexys A7 FPGA. Switches 15-12 encode the 4-bit binary number you wish to transmit (D3-D0) and switches 6-0 will be used to inject errors that damage your transmitted signal (*i.e.,* connected to the error injector module).

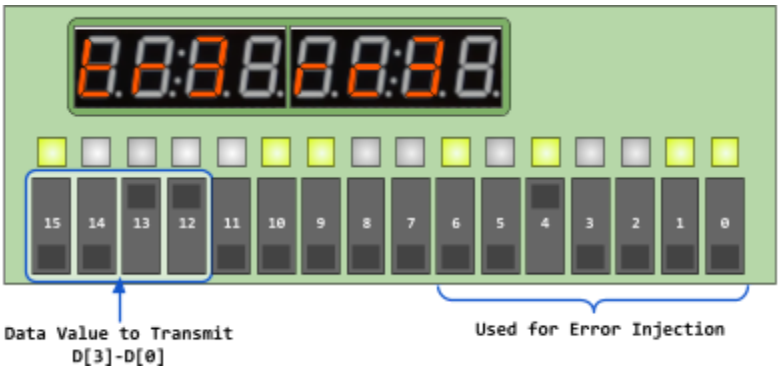

Figure 5. Input Combination (Switches) for Lab 7

Your board will display the 7-bit transmitted and received signals on the LEDs. The received signal is *before* error correction and therefore should indicate if there were any errors. The seven-segment display will be used to show the data value that was transmitted (before error injection) and the data value that was received (after error correction). In other words, the seven-segment display is used to show the data values displayed to an end user and the LEDs are used to display the transmitted and received signals with the parity bits and added redundancy. This configuration is shown in Figure 6.

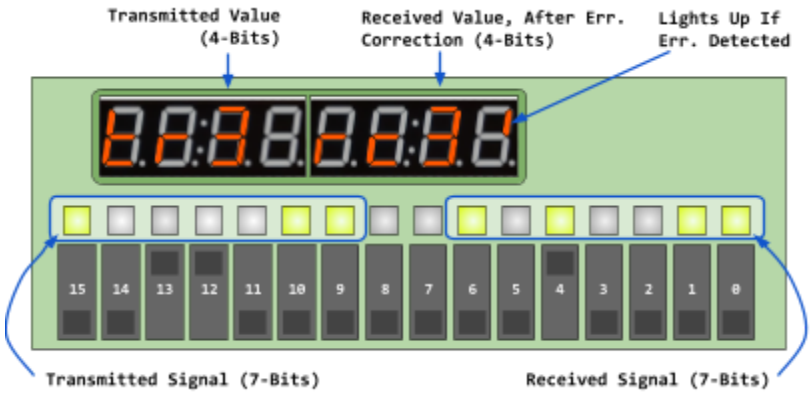

Figure 6. Output Combination (LEDs) for Lab 7

12. Implement the design shown in Figure 4 in the top-level module (`lab07_top`).

    a. Instantiate your parity generator module to encode the “transmitted signal,” consisting of the data and parity bits.

    b. Instantiate the error injector module, provided in `error_injector.sv`, to damage the transmitted signal. This module takes two inputs, the 7-bit transmitted data signal (**`In`**) and a 7-bit signal to indicate which bit will be flipped (**`Pos`**). The module outputs a 7-bit signal (**`Out`**) with an error at the indicated position. **`Out`** is the “damaged signal” that needs to be corrected by your Hamming code decoder components.

    c. Finally, to decode the “damaged signal,” instantiate your error position indicator and error corrector modules according to Figure 4.

13. Connect the inputs of your design to the switches as shown in Figures 5, where switches 15-12 will encode the 4-bit binary input D3-D0.

14. Connect the output of your design to the LEDs and the seven-segment display as shown in Figure 6. The seven-segment display is programmed automatically via the `hamming_display` module. You will need to connect the relevant ports of this module, as shown in the following code snippet:

```
hamming_display DISP(
    .d_tr( ),           /* Connect original, transmitted D3-D0 */
    .d_rc( ),           /* Connect received, corrected D3-D0   */
    .a( ),              /* Connect check code bit A */
    .b( ),              /* Connect check code bit B */
    .c( ),              /* Connect check code bit C */
    .clk(clk), .seg_n(SEG), .an(AN));
```

15. Verify that the circuit schematic matches your proposed design. You can view the circuit schematic under “RTL Analysis/Open Elaborated Design/Schematic” in the Flow Navigator.

16. Generate a bitstream and program the FPGA. Test that your circuit functions correctly and demonstrate it to an instructor.

Stop and Check: Using the switches for your error injector, inject single-bit errors into your transmitted data signal. Ensure that the final output on the seven-segment display is correct!

Record: Screenshot a schematic for each module for inclusion in your lab report (parity generator, error position indicator, and error corrector).

Questions: What happens if there are two errors in the transmitted signal? Can your circuit detect this? Can it correct it?

Part 6: Wrap-Up

Before leaving the lab, make sure to complete the following steps!

17. ***Demonstrate the operation of your circuit to the instructor.*** A portion of your lab grade will come from the lab report in addition to this demonstration.

18. Turn the Nexys board OFF and disconnect it from the computer. Make sure to save your Vivado project and sign out of your machine!

Lab Report

Lab Report. In collaboration with your lab partner, submit one lab report on Moodle that describes the lab activity and your team's circuit design. Your written report is worth 60% of your grade for each lab assignment. Your lab report should include the following sections:

- **Title Page:** Your lab report should begin with a title block that includes the course number, the title of the lab assignment, the names of each team member, and the date of the lab assignment. The title page should also include a "Statement of Collaboration" (detailed below) and an estimate of the total amount of time spent on the lab assignment. The inclusion of these components are graded, but the content of each statement is not used in determining your grade.

- **Statement of Collaboration:** Describe the contributions of each team member to the lab assignment, including writing of the lab report. What is specified in this section will not affect your grade, however it is expected that you periodically rotate roles within your lab group if you do divide work, including writing the report.

- **Introduction:** Include a brief paragraph summarizing the objectives of the lab and what your team achieved. This section should be written in your own words; it is not acceptable to copy text for this from the lab handout.

- **Design:** Include a concise yet descriptive explanation of the circuit design, following all steps in the process—from the problem specification and initial truth table, to equations and circuit schematics. Tables and equations should be typeset and clearly labeled. Technical information should be accompanied by some text narrating each step or aspect of the design.

- **Testing & Results:** Describe how you tested your circuit, including rationale for why you chose specific test cases if the circuit is not tested exhaustively. If you tested your circuit at multiple points (*i.e.,* using a testbench and then on the Nexys board), this section should include all test cases at each point. Test cases can be included in the form of tables or simulation waveforms. Results should be neatly formatted and labeled, indicating what step is being tested. This section should also include any recorded measurements, tests, diagrams, or images specified by the lab document.

- **Discussion/Conclusion:** Your discussion/conclusion section should comment on any observations about the technique you employed to get your results. Briefly describe any difficulties that you encountered and how you resolved them. Then, summarize what you concluded from your work. (It may also be helpful to think negatively, *i.e.* what your results did not show. Remember that a negative result is often as valuable as a positive result!)

  A number of technical questions may be posed in the assignment sheet. If questions are given, include your answers to them in this section.

Your lab report should be typed and submitted as a .pdf document. Only one member per team needs to submit the lab report.