Brandon Starks & Barrett Durtschi

Dr. Kantabutra

ECE 5508

Final Project Report

**Introduction:**

A Cyclic Redundancy Check (CRC) is an Error Detection (ED) method for determining if a bit-flip has occurred in data transmission through an unreliable channel. The goal of this project was to produce mathematical proofs that show a CRC works on the BASYS3 board using SystemVerilog. The application of an unreliable channel was modeled by forcibly changing one or more bits at the received checksum value. This change in bits allowed the CRC to show that it was capable of both illustrating that it could correctly decode a message, and whether each piece of data was correct or not. The scope of this project included simulating the CRC in Python to understand the operations of the CRC before FPGA implementation. Once the simulations were complete, the code would be referenced to create a program in SystemVerilog that would be capable of illustrating how the CRC works in simulation, and on the BASYS3 board.

**Background:**

A CRC is made up of an encoder and a decoder module. The encoder takes a data word input (in our case 8-bits) that is pushed through a generator. The generator’s main function is to XOR the input data word with the divisor. The divisor, in other words, is the shared polynomial used to encode and decode the message being sent. The generator outputs the remainder of this XOR operation, and it is appended to the original data word. This output is the message that will be sent through a communication channel and decoded. Once the data has reached the receiver, the process is flipped. The code word is sent through a checker function that will output another reminder. This remainder is the key factor in determining if there has been a bit-flip. If there has been a bit-flip, the remainder will be none zero and some type of flag will indicate there has been an error in the transmission. If the remainder is zero, the transmission was successful. Our project expanded on typical modulo-2 division CRCs. Instead of only computing a remainder given a polynomial and data, we perform these calculations using CRC register that is initialized to a specific state. Figures 1A, 1B, and 1C show how a typical modulo-2 division CRC works in calculation. A more extended analysis will be given to our additions to the modulo-2 type CRC in the next section.

A picture containing text, screenshot, font, diagram

Description automatically generated

Fig 1C. Final remainder shows that message has been sent with errors because it is a non-zero value.

A picture containing text, screenshot, font, diagram

Description automatically generated

Fig 1B. Illustration of Receiving side calculations. Same process as sender but data is appended with sender’s remainder.

A picture containing text, screenshot, font, diagram

Description automatically generated

Fig 1A. Illustration of modulo-2 division Encoder of CRC. Data is appended with zeros and XORed with divisor to yield the checksum/remainder.

**Python Simulations:**

Using a combination of Google Colab and Visual Studio Code, Python was chosen to be the language of choice to implement the CRC simulations. Two separate functions were created to model both the encoder and decoder modules we would make in SystemVerilog. Each module was tested separately to ensure its own operations were working before combining them into a single test function. Once the two functions were combined, the final test ensured that the result flag would trigger whether the output was the correct data sent from the sender. This flag was tested with several different data entries, all of which showed a passing test (data had not been corrupted). To ensure the tests would fail if data were to be corrupted, we changed the value of the checksum; this caused the result flag to trigger that the message had been received incorrectly. The encoder and decoder module are represented in the Python code snippets pictured in Fig 2A. and Fig 2B.

module crc\_8\_encoder(

    input logic [7:0] data,  // Data is used to calculate checksum with poly.

    output logic [7:0] checksum  // Checksum calculated for 0 result of XOR.

    );

    int unsigned poly = 8'h9b;  // Internal variable for polynomial.

    int unsigned crc = 16'h00ff; // Internal variable for CRC register.

    always @(data) begin

        crc = 16'h00ff;  // Reinitialize the CRC register for the modulo 2 division.

        for (int i = 0; i < 8; i++) begin   // Iterate over each literal in dataword.

            if (crc & 8'h80)                // If current bit of crc register is 1.

                crc = (crc << 1) ^ poly;    // XOR CRC variable with polynomial.

            else                            //

                crc = crc << 1;             // Shift CRC register to left.

                                            //

            if (data & (1 << (7-i)))        // If current literal in dataword is 1 ...

                crc = crc ^ 8'h01;          // XOR CRC register with dataword.

            end                             //

                                            //

        checksum = crc & 8'hff;             // Only take 8 LSBs for checksum

    end

endmodule

polynomial.

            else                            //

                crc = crc << 1;             // Shift CRC register to left.

                                            //

            if (data & (1 << (7-i)))        // If current literal in dataword is 1 ...

                crc = crc ^ 8'h01;          // XOR CRC register with dataword.

            end                             //

                                            //

        checksum = crc & 8'hff;             // Only take 8 LSBs for checksum

    end

endmodule

Fig 2A. Encoder function from Python code

module crc\_8\_decoder(

    input logic [7:0] data, checksum,  //  Data and checksum used to calculate result.

    output logic [7:0] result  // Result indicates if data received contains error (nonzero).

    );

    int unsigned poly = 8'h9b;  // Internal variable for polynomial.

    int unsigned crc = 16'h00ff; // Internal variable for CRC register.

    always @(data, checksum) begin

        crc = 16'h00ff;  // Reinitialize the CRC register for the modulo 2 division.

        for (int i = 0; i < 8; i++) begin   // Iterate over each literal in dataword.

            if (crc & 8'h80)                // If current bit of crc register is 1.

                crc = (crc << 1) ^ poly;    // XOR CRC variable with polynomial.

            else                            //

                crc = crc << 1;             // Shift CRC register to left.

                                            //

            if (data & (1 << (7 - i)))      // If current literal in dataword is 1 ...

                crc = crc ^ 8'h01;          // XOR CRC register with dataword.

            end                             //

                                            //

        result = checksum - (crc & 8'hff);  // Compare calculated checksum with supplied.

    end

endmodule

Fig 2B. Decoder Function from Python code

**GitHub work:**

GitHub was used as the choice repository for our work. This allowed for a better flow of work to be assigned to each person in the team. GitHub also made it so ‘collisions’ didn’t occur in our code when both of us had revisions that were made at or around the same time. A fair amount of time was spent learning how to use Git, but as GitHub is becoming ever more prominent for any type of project, it was extremely beneficial for any future projects involving a team. In total there were around thirty branches that were resolved and closed out, which shows the extent of our project. Fig 3A. shows our GitHub closed issues page and Fig 3B. shows a snippet of the commits

A screenshot of a computer

Description automatically generated with medium confidence

Fig 3B. Snippet of commits page from GitHub Repository

A screenshot of a computer

Description automatically generated with medium confidence

Fig 3A. GitHub Repository page of issues resolved.

**SystemVerilog Simulations/Implementation:**

In our SystemVerilog program, the Python code was easily transferred with a few changes to get the output the same as in the simulations. Additional modifications were made to show that the CRC register was being updated as the data input was being XORed with the polynomial. This made it easier to visualize how the CRC was being changed as the math operations were being applied. Some more noticeable changes included specifying which variables were designated as inputs/outputs and integer values needed to be unsigned to prevent an overflow condition. At first, we were able to showcase the data input along with the output checksum. To verify the results were correct, we ran a few examples from our working Python code and verified that the results for the checksum (encoder) and result (decoder) were the same in SystemVerilog. To better illustrate how the result is calculated, we also put the polynomial and CRC register (the temp value that was used to shift after XORing), on top of each other in the waveform window. Going through a couple of examples, it can be visually seen how the values are calculated which creates the checksum in the encoder and the result in the decoder. Fig 4A. and Fig 4B. show the waveform simulations output from Vivado. It is good to take note that the figures are only representative of a single test to better see the results.

A screenshot of a computer

Description automatically generated

Fig 4A: Encoder Testbench waveform with the input dataword, output checksum, and calculations of the CRC register

The BASYS3 board without any other hardware was chosen to be used as the FPGA for this project. To visually understand what the CRC was doing, with the I/O limitations, we chose to reduce our CRC from 16-bit to 8-bit. This reduction made it possible to show the checksum on the board along with the data input using all 8 switches and LED. Because the value of the CRC register consisted of 16-bits, it made it a little harder to show every aspect of the calculations, and that is why we provide the simulation discussed above. The result flag logic is shown on the HEX display as a zero or one depending on if the data was received intact. Below are the specific areas of the board we decided to utilize all the switches and LEDs. The bottom (W13-V17) switches are used as the data inputs while the top (R2-V2) switches are used for bit flipping. This bit flipping is incorporated to simulate that the result will accurately ‘detect’ if a bit has been flipped in the communication channel. The bottom (V14-U16) LEDs are used to show the checksum and the top (L1-V13) LEDs are used to show the result.

A screenshot of a computer

Description automatically generated

Fig 4B: Decoder Testbench waveform with input dataword and checksum. Outputs include the results and calculations of the CRC register

Our implementation can be seen through the link provided in our PowerPoint presentation. We can see as the data inputs are changed the Checksum and Result change to give a correct output (the top LEDs should all not be lit for a correct result). If one of the bit flip switches is activated, there should start to be LEDs on the top section of the BASYS3 board lighting up signifying the checksum has changed from the sender to receiver. This may seem simple, but there is a lot going on behind the code that is making these signals output as they should.

**Conclusion:**

In conclusion, this project successfully implemented an 8-bit CRC in the SystemVerilog simulation window and on the BASYS3 board. The CRC is a great beginning ED method to learn for both wireless engineers and digital designers because of its ease of use and usability in the industry. Using GitHub as our repository also increased our knowledge of team projects and how to collaborate while not having collisions in code. This project increased our knowledge level in both the wireless communications world and digital design spaces especially in the realm of SystemVerilog and FPGA programming.