Reiner Dizon
CpE 405

CpE 405 Final Project: CRC-16-CCITT Hardware Implementation

Table of Contents

Introduction

Cyclic Redundancy Check, a.k.a. CRC, is a channel coding algorithm which is a block code created using a generator polynomial. In order to generate a valid codeword or code polynomial, it has to be a multiple of the generator polynomial, g(x). In other words, a valid codeword will produce a remainder of zero when it is divided by this polynomial. To encode a message polynomial, it is multiplied by the size of the remainder (typically the degree of the generator polynomial) or length of CRC and then divided by the generator. Alternatively, it can also be multiplied by the check polynomial. However, the first method is primarily used to allow for hardware reuse of the CRC generation circuit for both encoder and decoder. To decode the message polynomial with the CRC, they are simply divided by the generator polynomial, and its remainder is called the syndrome. Whenever the syndrome produces zero, there is no error in the given encoded message which is consistent with the properties of the CRC.

Hardware Implementation

The main component of this circuit is the Linear Feedback Shift Registers or LFSR. This module contains several D flip-flops which is equal to the size of the CRC code. The data will be fed serially from the left, meaning that the most significant bit is the rightmost D flip-flop. All these flip-flops are initialized to zeroes, or in other words, the initial value of the CRC code is zero before any operations. Normally, in division operations, the subtraction operation is repeated multiple times to obtain the remainder. However, in Galois Field 2, there is no subtraction operation allowed in this system but an addition operation. This addition operation is implemented using XOR gates to perform the repeated operation necessary for the overall division operation to obtain either the CRC code or the syndrome. Based on the coefficients of the generator polynomial, the XOR gates are placed right before flip-flops with a '1' coefficient in the polynomial.

Simple CRC-4 Implementation Example

Its corresponding polynomial is $g(X) = X^4 + X + 1$

With a degree of 4, the size of this CRC code is four, which means this circuit requires 4 D flip-flops. Because the coefficients for the X and constant term is one, XOR gates will be placed before the 1st and 2nd flip flops. The other input to these XOR gates is the output of the MSB flip-flop. In the next page, the picture shows the CRC-4 hardware implementation:

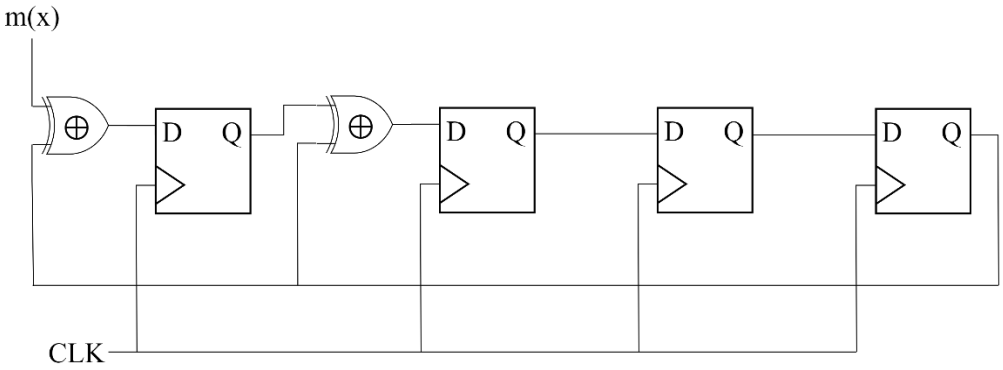

Fig. 1: CRC-4 Schematic

CRC-16-CCITT Implementation

Its corresponding polynomial is $g(X) = X^{16} + X^{12} + X^5 + 1$

Using the same principle as with the CRC-4 implementation, this circuit requires 16 D flip-flops and 3 XOR gates. The XOR gates are placed before the first, the fifth, and the twelfth leftmost D flip-flops with the other input coming from the rightmost D flip-flop. Figure 2 shows the schematic for this given CRC encoder and decoder circuit.

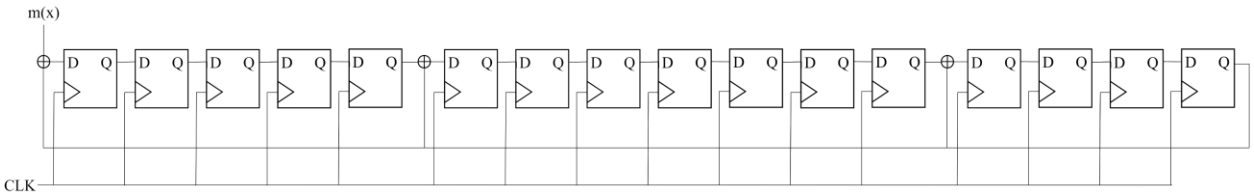

Fig. 2: CRC-16 Schematic

Encoding Simulation

Test Message: “The quick brown fox jumps over the lazy dog.”

```
VSIM 17> run -all
#                    5 << Starting the Simulation >>
# Original Message: The quick brown fox jumps over the lazy dog.  , Hex code: 546865207175696636b2062726f776e20666f78206a756d7073206f766572207468652065617a7920646f672e0000
#                 3702 i =         368, msg = 0, CRC = 1110101000001100, CRC = ea0c
```

Before encoding this message, I appended 16 zeroes to the end of this test message in order to get the proper CRC value. From the picture above, the CRC value for this message is

0xEA0C. In the next section, I will append this CRC value to see the syndrome and to confirm the proper operation of this circuit.

Decoding Simulation

To check the correct operation of the CRC module, the previously calculated CRC code is appended to the original message from the previous section. The results of this operation are in the following picture:

```
VSIM 19> run -all
#                    5 << Starting the Simulation >>
# Received Message (hex): 54686520717569636b2062726f776e20666f78206a756d7073206f766572207468652065206c617a7920646f672eea0c
#                 3702 i =         368, msg = 0, Syndrome = 0000000000000000, Syndrome = 0000
# Result: Code is good.
```

As shown above, the calculated syndrome or the remainder of the division operation is zero. This result indicates that there are no errors in the message received which is consistent with the properties of the CRC. In this next simulation, I have altered the received message to check if the calculated syndrome becomes non-zero. The CRC value is changed from 0xEA0C to 0x1111. The results of the decoding operation are shown below:

```
VSIM 21> run -all
#                    5 << Starting the Simulation >>
# Received Message (hex): 54686520717569636b2062726f776e20666f78206a756d7073206f766572207468652065206c617a7920646f672e1111
#                 3702 i =         368, msg = 1, Syndrome = 1111101100011101, Syndrome = fb1d
# Result: There's an error here
```

From the transcript above, it shows that the syndrome calculated is 0xFB1D which is non-zero. It is also indicated by the error message “Result: There’s an error here” in the above picture.

References

[1] S. N.V.N. P. Kumar, S. B. Jyothi, G. K. S. Tejaswi, “FPGA Based Design of Parallel CRC Generation For High-Speed Application,” in *International Journal of Scientific Research Engineering & Technology*, vol. 6, no. 3, pp. 258-264. Mar. 2017.
[2] M. Bloch. (n.d.). *Hardware Encoding and Decoding of Cyclic Codes*. [PowerPoint]. Available FTP: bloch.ece.gatech.edu Directory: sp10_ece6606 File: hardware.pdf
[3] J. Geluso (2007). “CRC16-CCITT.” [Website]. Available: http://srecord.sourceforge.net/crc16-ccitt.html Accessed April 16, 2018.

Appendix:

Shift_Reg.v

```
////////////////////////////////////////////////////////////////////////////
// Shift Register Module
// Reiner Dizon
// CpE 405 Final Project - CRC-16-CCITT Hardware Implementation
//
// ----------------------------------------------------------------------------
// Description:
// ----------------------------------------------------------------------------
// This module is an individual D flip-flop or shift register for the LFSR
// with synchronous reset and enable signals
//
////////////////////////////////////////////////////////////////////////////

module Shift_Reg(iClock, iEn, iRST, in, out);

//=======================================================
//  Parameter declarations
//=======================================================
parameter INITIAL = 1'b1;

//=======================================================
//  PORT declarations
//=======================================================
input wire iClock, iEn, iRST, in;
output wire out;

//=======================================================
//  REG/Wire declarations
//=======================================================
reg data;

//=======================================================
//  Structural/Behavioral coding
//=======================================================
initial data <= INITIAL;

always @(posedge iClock) begin
      if(iRST)
            data <= INITIAL;
      else if(iEn)
```

```
            data <= in;
      else
            data <= data;
end

assign out = data;

endmodule
```

CRC_CCITT.v

```
////////////////////////////////////////////////////////////////////////////////
// CRC CCITT Module
// Reiner Dizon
// CpE 405 Final Project - CRC-16-CCITT Hardware Implementation
//
// ------------------------------------------------------------------------
// Description:
// ------------------------------------------------------------------------
// This module models the CRC-16-CCITT circuit with all 16 D flip-flops
connected
// as a Linear Feedback Shift Register (LFSR) with 3 XOR gates.
// The polynomial is g(x) = x^16 + x^12 + x^5 + 1.
//
////////////////////////////////////////////////////////////////////////////////

module CRC_CCITT(iClock, iEn, iRST, iMSG, oCRC);
//=======================================================
//  Parameter declarations
//=======================================================
parameter INITIAL_VALUE = 16'hFFFF;
// parameter INITIAL_VALUE = 16'h1D0F;

//=======================================================
//  PORT declarations
//=======================================================
input wire iClock, iEn, iRST, iMSG;
output wire [15:0] oCRC;

//=======================================================
//  REG/Wire declarations
//=======================================================
wire final;

//=======================================================
//  Structural/Behavioral coding
//=======================================================

// polynomial used: 0x1021

Shift_Reg #(INITIAL_VALUE[0] ) u0 (iClock, iEn, iRST, iMSG ^ oCRC[15],
oCRC[0]);
Shift_Reg #(INITIAL_VALUE[1] ) u1 (iClock, iEn, iRST, oCRC[0], oCRC[1]);
Shift_Reg #(INITIAL_VALUE[2] ) u2 (iClock, iEn, iRST, oCRC[1], oCRC[2]);
Shift_Reg #(INITIAL_VALUE[3] ) u3 (iClock, iEn, iRST, oCRC[2], oCRC[3]);
Shift_Reg #(INITIAL_VALUE[4] ) u4 (iClock, iEn, iRST, oCRC[3], oCRC[4]);
Shift_Reg #(INITIAL_VALUE[5] ) u5 (iClock, iEn, iRST, oCRC[4] ^ oCRC[15],
oCRC[5]);
Shift_Reg #(INITIAL_VALUE[6] ) u6 (iClock, iEn, iRST, oCRC[5], oCRC[6]);
Shift_Reg #(INITIAL_VALUE[7] ) u7 (iClock, iEn, iRST, oCRC[6], oCRC[7]);
Shift_Reg #(INITIAL_VALUE[8] ) u8 (iClock, iEn, iRST, oCRC[7], oCRC[8]);
Shift_Reg #(INITIAL_VALUE[9] ) u9 (iClock, iEn, iRST, oCRC[8], oCRC[9]);
Shift_Reg #(INITIAL_VALUE[10]) u10(iClock, iEn, iRST, oCRC[9], oCRC[10]);
Shift_Reg #(INITIAL_VALUE[11]) u11(iClock, iEn, iRST, oCRC[10], oCRC[11]);
```

```
Shift_Reg #(INITIAL_VALUE[12]) u12(iClock, iEn, iRST, oCRC[11] ^ oCRC[15],
oCRC[12]);
Shift_Reg #(INITIAL_VALUE[13]) u13(iClock, iEn, iRST, oCRC[12], oCRC[13]);
Shift_Reg #(INITIAL_VALUE[14]) u14(iClock, iEn, iRST, oCRC[13], oCRC[14]);
Shift_Reg #(INITIAL_VALUE[15]) u15(iClock, iEn, iRST, oCRC[14], oCRC[15]);

endmodule
```

TestBench.v

```
////////////////////////////////////////////////////////////////////////////////
// Testbench Module
// Reiner Dizon
// CpE 405 Final Project - CRC-16-CCITT Hardware Implementation
//
// ----------------------------------------------------------------------------
// Description:
// ----------------------------------------------------------------------------
// This module tests and simulates the operation of the CRC-16-CCITT circuit.
//
////////////////////////////////////////////////////////////////////////////////

module TestBench;
//=========================================================
//  Parameter declarations
//=========================================================
parameter LENGTH = 368;

//=========================================================
//  REG/Wire declarations
//=========================================================
reg iClock, iEn, iRST, iMSG;
wire [15:0] oCRC;
integer i;

reg [LENGTH-1:0] msg;

//=========================================================
//  Structural/Behavioral coding
//=========================================================

// DUT
CRC_CCITT dut(iClock, iEn, iRST, iMSG, oCRC);

initial begin
      /* The quick brown fox jumps over the lazy dog. */
      // msg <=
368'h54686520717569636b2062726f776e20666f78206a756d7073206f76657220746865206c
617a7920646f672e0000;   // original
      // msg <=
368'h54686520717569636b2062726f776e20666f78206a756d7073206f76657220746865206c
617a7920646f672eea0c;   // received - correct
      // msg <=
368'h54686520717569636b2062726f776e20666f78206a756d7073206f76657220746865206c
617a7920646f672e1111;   // error test
```

```
	#5;

	$display($time, " << Starting the Simulation >>");
	$display("Original Message: %s, Hex code: %x", msg, msg);
	// $display("Received Message (hex): %x", msg);

	iEn <= 0;
	iRST <= 1;
	iMSG <= 0;
	#7;

	iEn <= 1;
	iRST <= 0;

	for(i = 0; i < LENGTH; i = i + 1) begin
		iMSG <= msg[LENGTH-1 - i]; #10;
	end

	iEn <= 0; #10;

	$display($time, " i = %d, msg = %b, CRC = %b, CRC = %x", i, iMSG, oCRC,
oCRC);
	// $display($time, " i = %d, msg = %b, Syndrome = %b, Syndrome = %x",
i, iMSG, oCRC, oCRC);

	if(oCRC == 0)
		$display("Result: Code is good.");
	else
		$display("Result: There's an error here");

	$stop;
end


always begin
	iClock <= 0; #5;
	iClock <= 1; #5;
end

always @(posedge iClock) begin
	// $display($time, " i = %d, msg = %b, CRC = %b, CRC = %x", i, iMSG,
oCRC, oCRC);
end


endmodule
```