CRC IMPLEMENTATION USING VERILOG HDL

DIGITAL SYSTEM DESIGN – MINI PROJECT

19/ENG/059 K.M.A.P. KULASINGHE

19/ENG/112 WICKRAMAARACHCHI U.H.D.A.I.

19/ENG/11 DAMITHA I.N.

19/ENG/28 GURUPARAN U.

CONTENT

INTRODUCTION

The cyclic redundancy check algorithm (CRC) is a common error detection algorithm used in many digital transmission and storage protocols to find problems as they are transmitted via the communication channel. It is a mathematical algorithm that generates a checksum (a numerical value) from a block of data. The checksum is used to compare the data that is transmitted with the data that is received, to detect any errors or discrepancies.

CRC works by dividing the data into segments and using a mathematical formula to calculate a numerical value, known as the checksum, for each segment. The checksum is then sent along with the data to the receiver. Upon receiving the data, the receiver calculates the checksum of the data and compares it with the one sent with the data. If the two checksums match, then the data is assumed to be valid and accepted. If the two checksums do not match, then the data is assumed to be corrupted and the receiver rejects the data.

The main advantage of using CRC is that its calculations are relatively fast, and it is relatively easy to implement. It is also relatively robust, meaning that it can detect a wide variety of errors. In addition, CRC is widely used in many communication protocols such as Ethernet, Wi-Fi, and USB. It is also used in many storage formats such as CDs, DVDs, and hard drives. CRC is a powerful tool for detecting errors in digital data transmission and is a popular choice for many applications.

The aim of this project is to use Verilog to create and construct a Cyclic Redundancy Check Error Detection Code. A quick and effective solution for detecting burst faults in digital data transmission and storage is provided by the CRC code.

SAMPLE CALCULATION

- Data bit to be send = 100100
- Given polynomial equation = x^3+x^2+1
- Divisor (k) = 1101
- Appending zeros = 4 – 1 = 3
- Dividend = 100100000

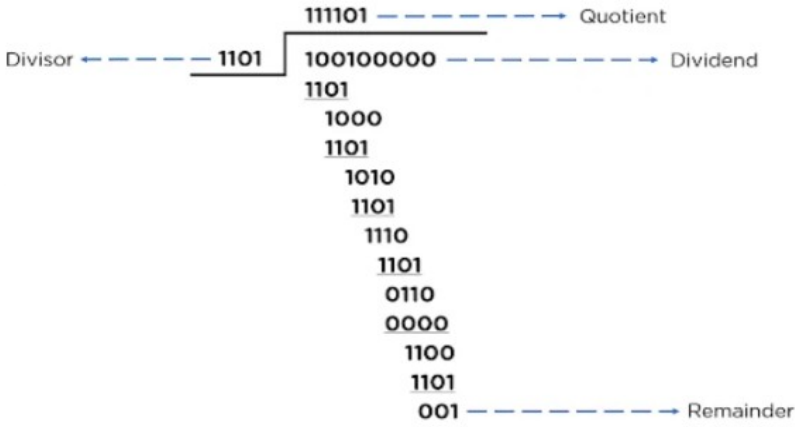


Figure 1: MODULO-2 DIVISION.

- The new message bit that has to be transmitted at the sender's end = 100100001

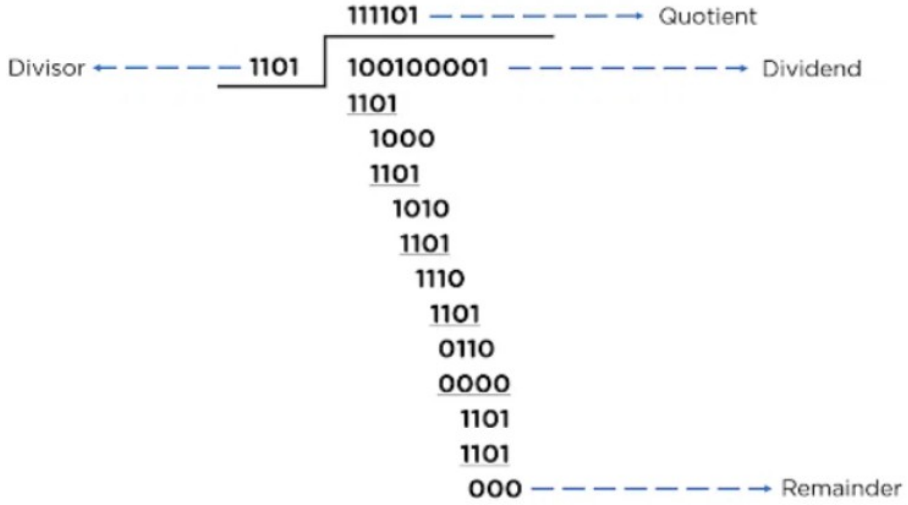
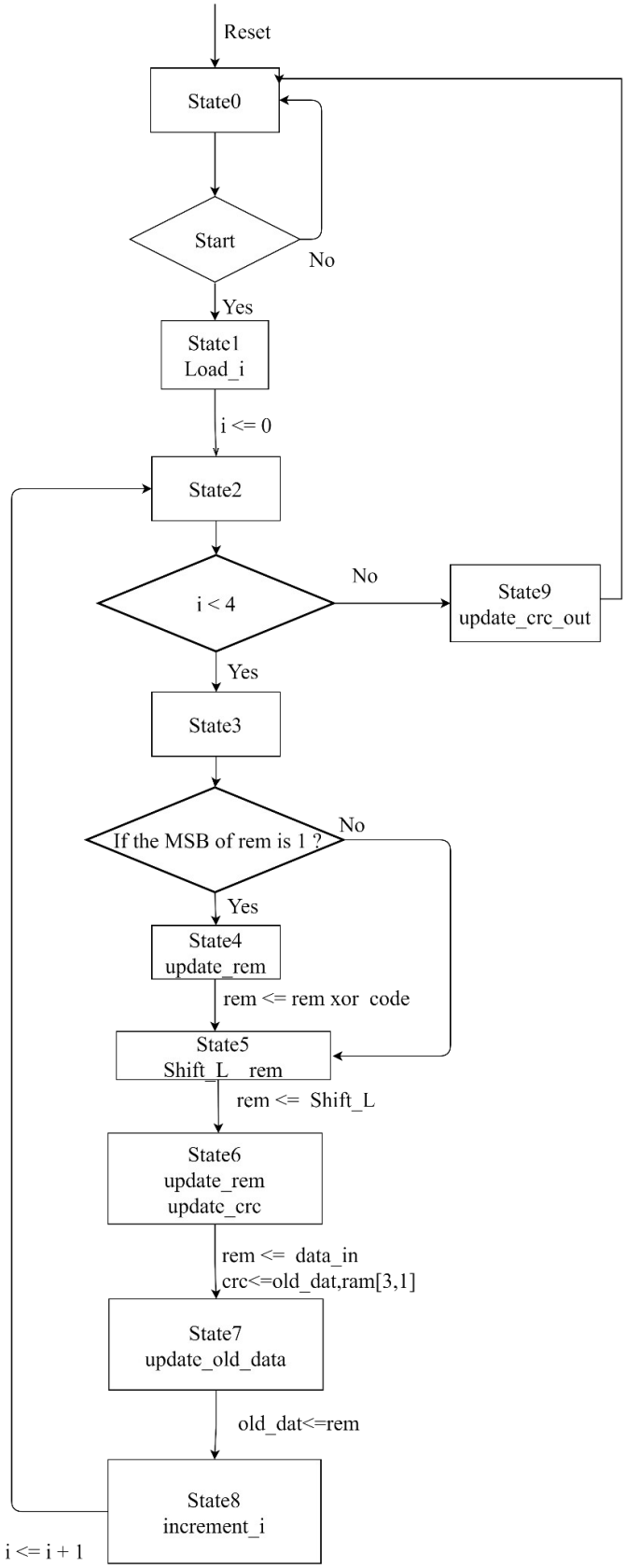


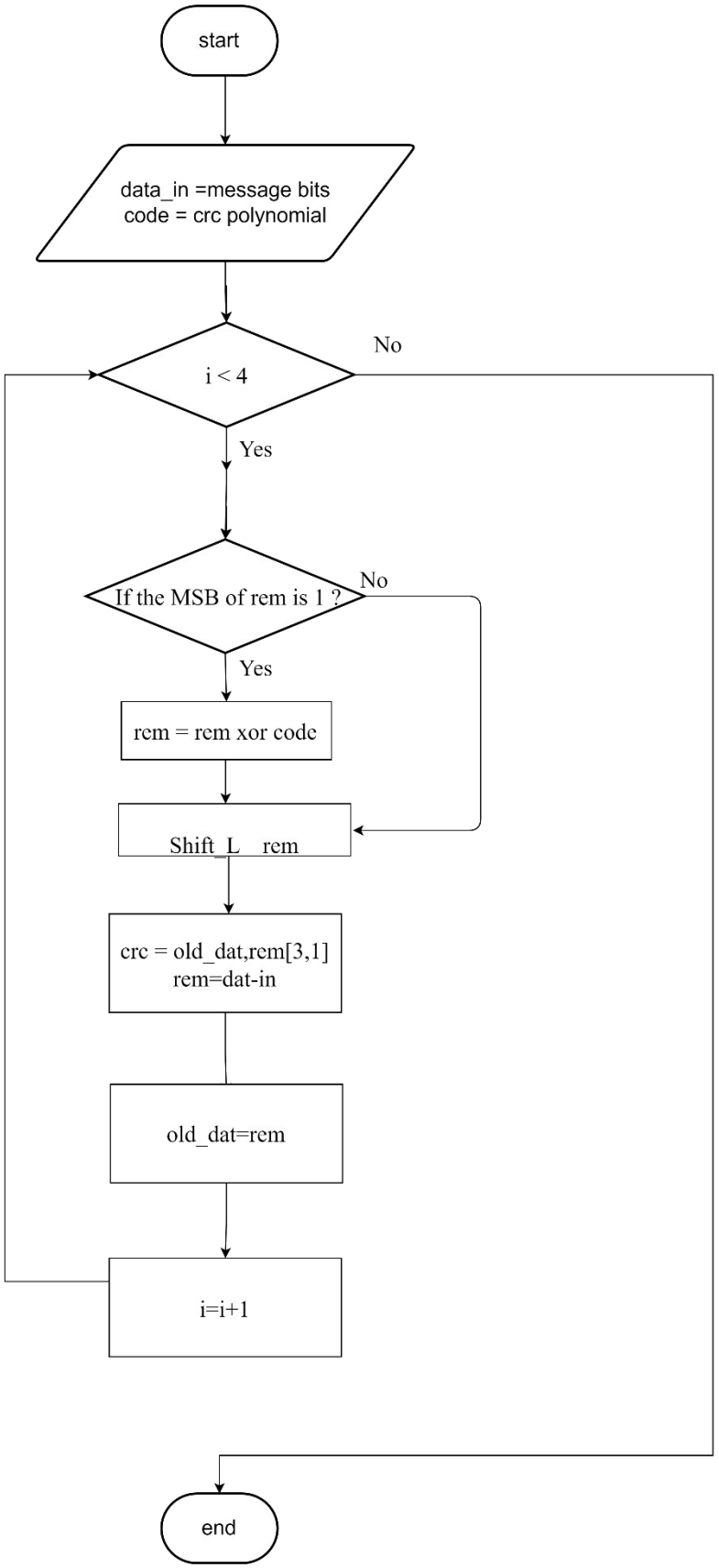
Figure 2: MODULO-2 DIVISION.

At the receiving end, when the division is performed, if the remainder equals to zero then the message bit is transmitted without an error. But if the remainder is not zero then the message bit is not send correctly.

DEVELOPMENT OF ASMD



FLOW CHART


start
data_in =message bits
code = crc polynomial
i < 4
No
Yes
If the MSB of rem is 1 ?
No
Yes
rem = rem xor code
Shift_L rem
crc = old_dat,rem[3,1]
rem=dat-in
old_dat=rem
i=i+1
end

VERILOG CODE

- Testbed

```
`timescale 1ns / 1ps
module Testbed();
    reg clk;
    reg [3:0] data;
    reg rst;
    reg en;
    wire [6:0] output_crc;

    //clock generation
    initial clk = 0;
    always #5 clk = ~clk;

    //enable
    initial en = 1;

    //reset
    initial rst = 0;

   //simulation duration
    initial
    #50 $finish;

    initial
    begin
           data = 4'b1001;
        #10 data = 4'b1011;
        #10 data = 4'b1000;
        #10 data = 4'b1010;

    end

    crc74 c1 (.crc_en(en),
            .rst(rst),
            .clk(clk),
            .code(4'b1011),
            .data_in(data),
            .crc_out(output_crc));

endmodule
```

- *Testbed* module code explanation

  - Module *Testbed* serves as both the testbed and the top module.
  - The time scale is specified as “the time unit 1ns, and precision is 1ps”. The variables are declared as,
    - *clk* as a register, which is used in clock generation
    - *data* as a 4-bit register ,which holds dataword
    - *rst* as a register, which holds the reset status
    - *en* as a register, which holds the enable status
    - *output_crc* as 7-bit wire, which holds the calculated codeword

  - A clock signal is generated, which is used to synchronize the modules. The variable *clk* is initialized, which inverts its value every 5 nanoseconds.
  - The *en* and *rst* variables are initialized.
  - The simulation duration is set to 50 nanoseconds.
  - The *data* variable is initialized with different values, at different time intervals.
    - At t = 0ns , *data* is set to 1001
    - At t = 10ns, *data* is set to 1011
    - At t = 20ns, *data* is set to 1000
    - At t = 30ns, *data* is set to 1010

  - The *crc74* module is instantiated, with above configured values.

- Module crc74

```
`timescale 1ns / 1ps
module crc74(crc_out, data_in, crc_en, rst, clk, code);

//inputs and outputs
output [6:0] crc_out;
input [3:0] data_in;
input [3:0] code;
input crc_en;
input rst;
input clk;

//register initialization
reg [3:0] rem = {4{1'b0}};
reg [3:0] old_dat = {4{1'b0}};
reg [6:0] crc = {6{1'b0}};;

//assigning codeword to output
assign crc_out = crc;

//always at positive edge of clock
always @(posedge clk)
begin
    if (clk)
        rem = data_in;  //new data input
        old_dat = rem;  //keeping a copy
end

//always at negative edge of clock
always @(negedge clk)
begin
    if (crc_en)
        repeat (4)
        begin
            if (rem[3] == 1)
            begin
                rem = rem ^ code;   //XOR
            end
            rem = rem << 1;     //left shift
         end
         crc = {old_dat,rem[3:1]};  //creating codeword
end
```

```
//always at positive edge of reset
always @(posedge rst)
begin
    if (rst)
    begin
        rem <= {4{1'b0}};
        crc <= {6{1'b0}};
    end
end

endmodule
```

- *crc74* module code explanation

  - Module *crc74* implements the crc(7,4) algorithm. The codeword is 7-bit long, and the dataword is 4-bit long.
  - The same timescale as the topmodule is specified.
  - The *crc74* module is declared with the,
    - Outputs
      - *crc_out* as a 7-bit register output, which outputs the codeword after calculation
    - Inputs
      - *data_in* as a 3-bit input, which receives the dataword
      - *code* as a 3-bit input, which receives the divisor
      - *crc_en* as an input, which receives the enable status
      - *rst* as an input, which receives the reset status
      - *clk* as an input, which receives the clock signal for synchronization
  - Three registers are declared for holding temporary values.
    - *Rem* as a 4-bit register, to hold remainder of the division
    - *old_dat* as a 4-bit register, to hold data from the previous iteration
    - *crc* as a 7-bit register, to hold the codeword until it is assign to the *crc_out*
  - The *crc* variable is assigned to *crc_out* variable, which is sent outward to the testbed.
  - When a positive edge of the clock is sensed, the always block, uses blocking assignments to assign *data_in* to *rem* and *rem* to *old_dat*.

- When a negative edge of the *clk* is sensed, the always block runs to calculate the crc.
  - If the *crc_en* is asserted, the modulo-2 division is performed, through a repeat loop.
  - If the first bit of the *rem* register is 1, *rem* is XORed with the *code* (divisor).
  - A single left bit shift is performed on *rem* register.
  - This is performed 4 times.
  - After the loop, the register *crc* is assigned with the concatenated register of *old_dat* and *rem*.

- When a positive edge of the *rst* is sensed, the always block runs and resets the registers, *rem* and *crc* into zeros.

SIMULATION

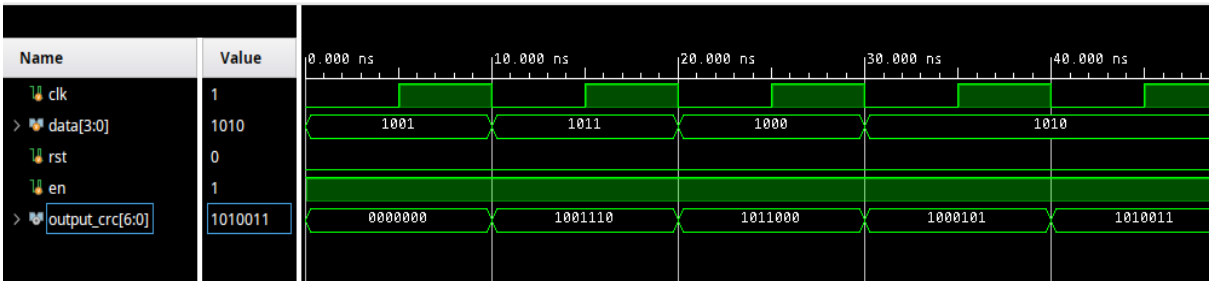


Figure 3: SIMULATION OF THE CODE.

- The reset signal is set to zero, and enable is set to high, and the clock signal is flipped every 5 nanoseconds.
- During the positive edge of the clock, the dataword in the *data* register in topmodule, is captured by the *crc74* module.
- During the negative edge of the clock, the calculated codeword is sent out by the *crc74* module, to the *output_crc* register in the topmodule.

Table 1: SIMULATION DATA.

Dataword (*data*)	**Codeword (*output_crc*)**
1001	1001110
1011	1011000
1000	1000101
1010	1010011

REFERENCES

[1] M, SAIPAVANKUMAR, et al. “FPGA Implementation of CRC Error Detection Code.” 1 Jan. 2015.