International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 04 Issue: 11 | Nov -2020 ISSN: 2582-3930

Comprehensive study on 8/12-bit Cyclic Redundancy Check

Neelanjan Goswami[1], Moka Saicharan[2], Mohd Ahmed Siddique[3], Sudarshan Kumar[4]

[1,2,3,4] *Department of Electronics and Communication Engineering, Bangalore Institute of Technology*

Abstract - Our understanding of Cyclic Redundancy Check mainly revolves around the operation performed on bits of data to be used in various stages of communication and many other applications to check the accuracy of the bits received. Cyclic Redundancy Check (CRC) is that very device/operation performed on cyclic codes of binary data to check the accuracy (validity) of bits received at any stage after transmission i.e. whether the bits received, match/correspond to the bits transmitted from a transmitter through a channel. The operation which brings this checking mechanism into reality was first analysed. This operation required to add certain additional bits (check bits) at the end of the message. Once added, it was added/XOR'ed with the bits of generator polynomial appropriately. The generation of these certain check bits which now represent a unique sequence of 0s and 1s was constructed in the form of a Verilog Code through the use of required statements accordingly as and when required. In this paper, the language chosen for CRC synthesis is Verilog and the RTL Schematic and Technology View of the code has been displayed along with the CRC output waveforms. Message bits along with the CRC output too has been shown which verifies the CRC check bits and their operation.

Key Words: Redundancy, MSB, XOR, Checking mechanism Message and Check Bits

1.INTRODUCTION

A cyclic redundancy check (CRC) is a code which identifies errors that are commonly used in digital networks and storage devices to detect unintended changes to raw data. CRCs are named because of the consistency of the test (data verification) value (it extends the message without adding information) and the algorithm is based on cyclic codes. CRCs are popular because they are simple to implement in binary hardware, easy to analyse mathematically, and especially good for detecting common noise-related transmission channel errors. Since the check value has a fixed length, the function that generates it is sometimes used as a hash function.

The CRC was invented by W. Wesley Peterson, 1961; a 32-bit CRC feature, being used Ethernet and several other protocols, is the result of a group of researchers and was documented in 1975.

The CRCs are based on the notion of cyclic error-codes correction. W first formulated the use of systematic cyclic codes that encode messages by adding a fixed-length check value for detecting errors in communication networks., was first proposed by W. Wesley Peterson in 1961.

Cyclic codes are not always simple to implement but have the benefit of becoming ideally suited for the detection of burst errors: contiguous sequences of invalid data symbols in messages. This is crucial as burst errors are common transmission errors in several modes of communication, which includes magnetic or optical storage devices.

The n-bit CRC implemented inside an arbitrary length data block detects any single error burst not over n bits, and the fraction of any longer error bursts that it will detect is (1 – 2 − n).

2. CRC operations

2.1. Working of a basic CRC

Specifying a CRC code involves the interpretation of a so-called polynomial generator. This polynomial is the divisor in a long division of polynomials, which takes the message as the dividend and discards the quotient and transforms the remainder into the product. The important caveat is that the polynomial coefficients are calculated on the basis of a finite field arithmetic (Binary calculations), so that the addition operation can always be performed bitwise-parallel (there is no carry between digits).

Throughout reality, all widely used CRCs employ two components commonly named 0 and 1 that suit comfortably the device architecture.

A n-bit CRC is considered a CRC if its check value is n bits wide. Multiple CRCs are possible for a given n, with each one having a different polynomial. Such a polynomial has the highest degree n, meaning it has terms n + 1. In other words, the polynomial has a n + 1 length; its encoding takes n + 1 bits. Remember that most specifications of polynomials either drop MSB or LSB, as they are still 1.

The easiest system which detects error, the parity bit, is really a 1-bit CRC which utilizes the x + 1 polynomial generator (two terms) and also has the name CRC-1.

2.2. Functionality of CRC

CRC-enabled device computes a short, fixed-length binary sequence, recognized as the check value or CRC, for each block of data being sent or stored and appends it to the data, forming a codeword.

2.3. Mechanism of CRC

When a device receives or reads a codeword, it either compares its check value to a value that has been newly determined from the data block, or performs a CRC on an equivalent basis on the entire codeword and compares the resulting check value with the predicted residue constant. If the

values for the CRC do not match, then there is a data error in the block.

The app may take corrective steps, such as re-reading the block or asking it to be forwarded again. Therefore, the data is believed to be error-free (though it can contain undetected errors with a limited likelihood; this is inherent in the essence of error checking).

2.4. Computation of CRC codeword (Division algorithm used)

Calculation of the n-bit binary CRC requires to line the bits that represent the input in a row and also to place the (n + 1)-bit pattern that represents the CRC divisor (called the "polynomial") below the left end of the row.

In this instance, we encode 14 bits of a 3-bit CRC message with a polynomial x3 + x + 1. The polynomial is expressed in binary as a coefficient; the third-degree polynomial has four coefficients (1x3 + 0x2 + 1x + 1). In this case, the coefficients are 1, 0, 1 and 1. The outcome of the calculation is 3 bits long.

Start with the message to be encoded:
11010011101100

It is first padded with zeros corresponding to the length of bit n of the CRC. This is achieved in such a way that the resulting code word is in structured form. Here is the first computation for the 3-bit CRC:

2.4.1. Pre-computation

11010011101100 000	<--- input right padded by 3 bits
1011	<--- divisor (4 bits) = $x^3 + x + 1$
01100011101100 000	<--- result

In each step, the algorithm acts on the bits directly above the divider. The result of this iterative process is the bitwise XOR of the polynomial divider with the bits above it. For this stage, the bits not above the divisor are merely copied directly below. The divisor is then moved one bit to the right, and now the process is repeated until the divisor reaches the right end of the source row.

$$a + b = \gamma \quad (1)$$

2.4.2. Calculation

Here is the entire calculation:

Long Division: -

```
11010011101100 000                (A)
1011                              (B)
----------------------------
01100011101100 000                (C)
 01011
 ----------------------------
 00111011101100 000
  001011
  ----------------------------
  00010111101100 000
   0001011
   ----------------------------
   00000001101100 000             (D)
    00000001011
    ----------------------------
    00000000110100 000
     000000001011
     ----------------------------
     00000000011000 000
      0000000001011
      ----------------------------
      00000000001110 000
       00000000001011
       ----------------------------
       00000000000101 000
        00000000000101 1
        ----------------------------
        00000000000000 100        (E)
        ----------------------------
```

(A) <--- input with check value

(B) <--- divisor

(C) <--- result (note the first four bits are the XOR with the divisor beneath, the rest are unchanged)

(D) <--- note that the divisor moves over to align with the next 1 in the dividend, the rest are unchanged in other words, it doesn't necessarily move one per iteration

(E) <--- remainder (3 bits).

Division algorithm terminates here as the dividend is equal to zero. Although the leftmost bit of the divisor has zeroed each bit of input it has touched, only other bits in the input row that can be null are now the n bits on the right end of the row. Such n bits are now the remainder of the division step, which will also be the evaluation of the CRC function (except if any of the chosen CRC specification is requested for post-processing).

2.4.3. Post-Calculation

Long Division: -

```
11010011101100 100          (A)
1011                        (B)
----------------------------
01100011101100 100          (C)
01011
----------------------------
00111011101100 100
001011
----------------------------
00010111101100 100
0001011
----------------------------
00000001101100 100
00000001011
----------------------------
00000000110100 100
000000001011
----------------------------
00000000011000 100
0000000001011
----------------------------
00000000001110 100
00000000001011
----------------------------
00000000000101 100
00000000000101 1
----------------------------
00000000000000 000          (D)
----------------------------
(A) <--- input with check value
(B) <--- divisor
(C) <--- result
(D) <--- remainder
```

3. Model Code based on above example

3.1. Verilog Code

```
module crc(msg,gp,crcf);
input [13:0]msg;
input [3:0]gp;
output [16:0]crcf;
reg [16:0]rem;
integer i,b,cnt=0;

always@(msg or gp)
begin
cnt=0;
for(i=16;i>=0;i=i-1)
begin
if(i<3)
rem[i]=1'b0;
else
rem[i]=msg[i-3];
end
begin: blk
for(i=16;i>=3;i=i-1)
begin
if(rem[i])
begin
b=i;
disable blk;
end
end
end
end

always@(cnt)
begin: blk2
for(i=16;i>=3;i=i-1)
begin
if(rem[i]==1'b1)
begin
b=i;
disable blk2;
end
end
end
always@(b)
begin: blk3
if(rem[16-:14]==14'b00_0000_0000_0000)
disable blk3;
else
begin
rem[b-:4]=rem[b-:4]^gp[3:0];
cnt=cnt+1;
end
end
```

```
assign crcf[16-:14]=msg[13-:14];
assign crcf[2:0]=rem[2:0];
endmodule
```

3.2. Modelsim output waveform

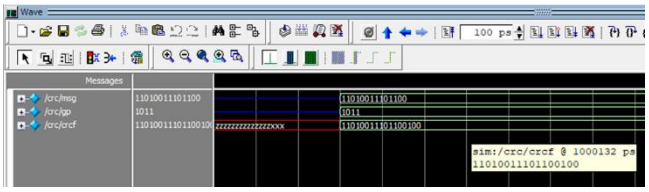


Figure 1. Model code output waveform.

In the above simulation, msg is the message input coefficients = 11010011101100

gp is the generator polynomial coefficients = 1011

crcf is the message input appended with the check bits or check value (3 bits) = 11010011101100 100

4. VERILOG code for 8 bit /12 bit Redundancy Check

Since in the computation example we used a 14-bit message input for encoding and a 3rd degree generator polynomial (i.e. 4-bit generator polynomial), for the actual implementation of an 8 bit or a 12-bit cyclic redundancy check we need to choose a 9 bit and 13-bit generator polynomial coefficients input respectively. Hence the above code is modified accordingly for a 9 bit or a 13-bit generator polynomial input along with a 16-bit message input as shown below:

4.1. Verilog Code

```
module crc(msg,gp,crcf);

input [15:0]msg;   /* 16 bit message input */

/* [12:0] to be used for a 13-bit generator polynomial (12 bit CRC) output [23:0] crcf; /* 16 + (8-1) = 23 bit CRC output for an 8 bit CRC and 16 + (12-1) = 27 bit CRC output for a 12 bit CRC */

input [8:0]gp;

reg [23:0]rem;

/* Appending zeroes at the end of the message input */
integer i,b,cnt=0;
integer i,b,cnt=0;

always@(msg or gp)
begin
cnt=0;
for(i=23;i>=0;i=i-1)       /* i=27 for a 12 bit CRC */

begin
if(i<8)                    /* i<12 for a 12 bit CRC */



rem[i]=1'b0;
else
rem[i]=msg[i-8];           /* [i-12] for a 12 bit CRC */

end
begin: blk
for(i=23;i>=8;i=i-1)       /* i=27;i>=12 for a 12 bit CRC */

begin
if(rem[i])
begin
b=i;
disable blk;
end
end
end
end

/* Finding the position of logic 1 bit (i.e. b) from left of appended message */

always@(cnt)
begin: blk2
for(i=23;i>=8;i=i-1)
begin

/* i=27;i>=12 for a 12 bit CRC */

if(rem[i])
begin
b=i;
disable blk2;
end
end
end

/* Performing bitwise XOR */
always@(b)
begin: blk3
if(rem[23-:16]==16'b0)              /* rem[27-:16] for 12 bit CRC */

disable blk3;
else
begin
rem[b-:9]=rem[b-:9]^gp[8:0];
```

```
/*rem[b-:13], gp[12:0] for a 12 bit CRC*/

cnt=cnt+1;
end
end

assign crcf[23-:16]=msg[15-:16];   /*crcf[27-:16] for a 12 bit CRC */

assign crcf[7:0]=rem[7:0];

endmodule
```

4.2. Modelsim output Waveform

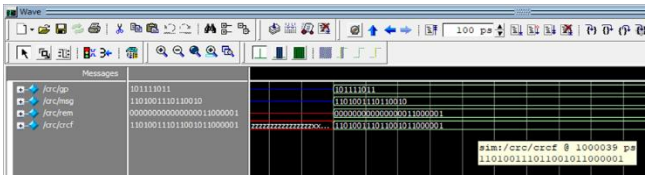


Figure 2. Verilog code output waveform.

In the above output,

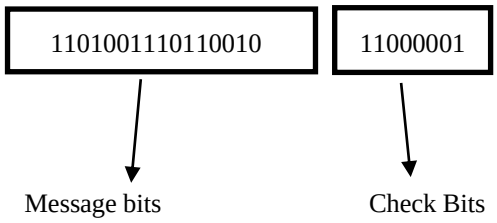
- gp (generator polynomial – 8th degree (9 bits)) = 1x8 + 0x7 + 1x6 + 1x5 + 1x4 + 1x3 + 0x2 + 1x1 + 1x0 = 101111011
- msg (input message coefficients – 16 bit) = 1101001110110010
- rem(remainder) = 0000000000000000 11000001
- crcf (CRC output i.e. input message appended with rem (check bits)) = 1101001110110010 11000001

5. Analytical Solution

```
1101001110110010 00000000          (A)
101111011                          (B)
-----------------------------------
0110111000110010 00000000          (C)
0101111011
-----------------------------------
0011000011110010 00000000
00101111011
-----------------------------------
0001111110010010 00000000
000101111011
-----------------------------------
0000100000100010 00000000
0000101111011
-----------------------------------
0000001111111010 00000000
000000101111011
-----------------------------------
0000000100001100 00000000
0000000101111011
-----------------------------------
0000000001110111 00000000
0000000001011110 11
-----------------------------------
0000000000101001 11000000
                  101111 011
-----------------------------------
0000000000000110 10100000
0000000000000101 111011
-----------------------------------
0000000000000011 01001100
0000000000000010 1111011
-----------------------------------
0000000000000001 10111010
0000000000000001 01111011
-----------------------------------
0000000000000000 11000001          (D)
```

Remainder (Check Bits) = 11000001
(A) Message with zero padding
(B) gp
(C)Message xor gp
(D)Remainder

CRC encoded output =



6. CRC WITH Error Verification Using Remainder

6.1. Verilog Code

```
/*
 working:

 1.) obtaining check bit for given msg input
 2.) passing message without error and verifying
 3.) passing message with error and verifying
 4.) assigning a new signal output=
0; if no error
1; if error

1010101010101100
101111011

*/

module
crc(msg,msg_retrieved,gp,crcf_msg,crcf_retrieved,error);
```

```
input [15:0]msg;
input [15:0]msg_retrieved;
output error;
reg error;

/* 16 bit message input */

input [8:0]gp;

/*using same gp for msg and msg_retrieved

[12:0] to be used for a 13 bit generator polynomial(12 bit CRC ) */
output [23:0]crcf_msg;
output [23:0]crcf_retrieved;

/* 16 + (8-1) = 23 bit CRC output for an 8 bit CRC and 6 + (12-1) = 27 bit CRC output for a 12 bit CRC */

reg [23:0]rem;
reg [23:0]rem1;
integer i,b,cnt=0,cnt1=0;

/*Appending zeroes at the end of the message input */

always@(msg or gp)
begin
cnt=0;
for(i=23;i>=0;i=i-1)

/* i=27 for a 12 bit CRC */
begin if(i<8)

/* i<12 for a 12 bit CRC */

rem[i]=1'b0;
else rem[i]=msg[i-8];

/* [i-12] for a 12 bit CRC */

cnt=cnt+1;
end
end

/* Finding the position of logic 1 bit (i.e. b) from left of appended message */

always@(cnt)
begin: blk1
for(i=23;i>=8;i=i-1)
begin

/* i=27;i>=12 for a 12 bit CRC */

if(rem[i])
begin b=i;
disable blk1;
end
end

end

/* Performing bitwise XOR */

always@(b)
begin: blk2
    if(rem[23-:16]==16'b0)
/* rem[27-:16] for 12 bit CRC */
disable blk2;
else
begin rem[b-:9]=rem[b-:9]^gp[8:0];

/* rem[b-:13], gp[12:0] for a 12 bit CRC*/

cnt=cnt+1;
end
end
assign crcf_msg[23-:16]=msg[15:16];

/*crcf[27-:16] for a 12 bit CRC */

assign crcf_msg[7:0]=rem[7:0];

/*same process for msg_retrieved
Appending zeroes at the end of the message input */

always@(msg_retrieved or gp)
begin
cnt1=0;
for(i=23;i>=0;i=i-1)

/* i=27 for a 12 bit CRC */

begin
  if(i<8)
  rem1[i]=rem[i];
  else
  begin
    rem1[i]=msg_retrieved[i-8];

/* [i-12] for a 12 bit CRC */ cnt1=cnt1+1;
end
end
end

/* Finding the position of logic 1 bit (i.e. b) from left of appended message */

always@(cnt1)
begin: blk3
for(i=23;i>=8;i=i-1)
begin

/* i=27;i>=12 for a 12 bit CRC */

if(rem1[i])
begin b=i;
disable blk3;
end
end
```

```
end

/* Performing bitwise XOR */

always@(b)
begin: blk4
 if(rem1[23:0]==24'b0)

/* rem[27-:16] for 12 bit CRC */
 disable blk4;
 else if(b>=8)
 begin rem1[b-:9]=rem1[b-:9]^gp[8:0];

/* rem[b-:13], gp[12:0] for a 12 bit CRC*/

 cnt1=cnt1+1;
 end
 end
 assign crcf_retrieved[23-:16]=msg_retrieved[15-:16];

/*crcf[27-:16] for a 12 bit CRC */
assign crcf_retrieved[7:0]=rem1[7:0];
/*Error verification*/
always@(crcf_msg & crcf_retrieved)
begin
if(rem1[23:0]!=24'b0)
assign error=1;
else
assign error=0;
end
endmodule
```

6.2. Modelsim output Waveforms

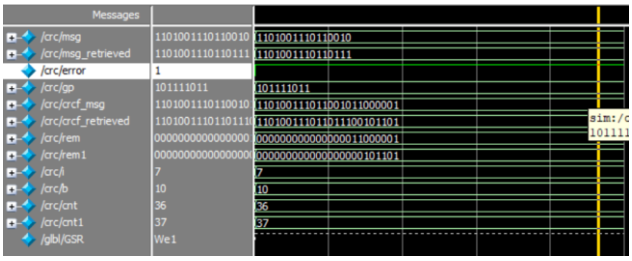


Figure 3. Verilog code output waveform with no error in message received variable

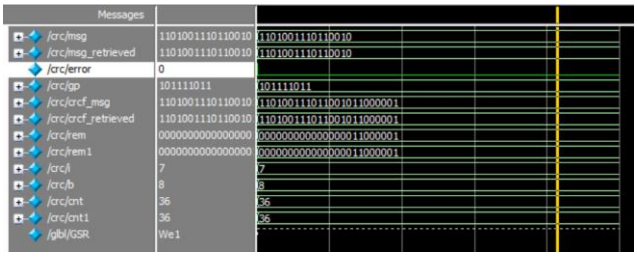


Figure 4. Verilog code output waveform with Error in received message, last 3 bits as 111 instead of 010

7. Obtaining RTL Schematic

RTL schematic cannot be formed when a variable which is not pre-initialized cannot have an operation upon itself. Here the 'rem' variable is being used in initialization in the first always block {always (msg, gp)}, 'rem=msg[i-8]' and in the next always block below it, i.e. (always@(b), 'rem[b-:9] =rem[b-:9] ^gp [8:0]'. Since all always blocks run parallel, the RTL designer gives an error as rem is not initialized "rem<23> is already riven by msg <15>"

To overcome this problem, we used to variables of net and reg datatype. The output variable would be of reg type and the other would be net type.

7.1. Code A: Simpler Verilog code

```
module crc(msg,gp,crf);
input [15:0]msg; // 16bit i/p
input [8:0]gp; // 8 bit crc
output [23:0] crf; // 16+9-1 = 24 bits
//reg [23:0] crcf;
wire [23:0] crcf;
reg [8:0] rem;
integer i,cnt,b,c,ct;
reg [23:0] a;
// appending zeros
always @ (msg,gp)
begin
for(i=23;i>=0;i=i-1)
begin
if(i>8)
a[i] = msg[(i-8)];
else
a[i] = 0;
end
end

//finding leftmost 1 and then find no. of xors and xor it
always @ (msg,gp)
begin : blk1
c=23;   // c is 1 leftmost
for(i=23;i>=0;i=i-1)
begin : blk3
if(a[i])


begin:blk2
for(cnt=0;cnt<=14;cnt=cnt+1)
begin

if(cnt==(c-8))
disable blk2;
b=c;
rem = a[b-:9]^gp[8:0];
b=b-1;


end
end


c=c-1;
```

```
if(c<=8)
disable blk1;
end


end

assign crcf[23:9]=msg[15:0];
assign crcf[8:0]=rem[8:0];
assign crf = crcf;
endmodule
```

The drawback of above code: But this code runs for finite number of times. To make it more volatile and reduce its computation time, we may change the code as following:

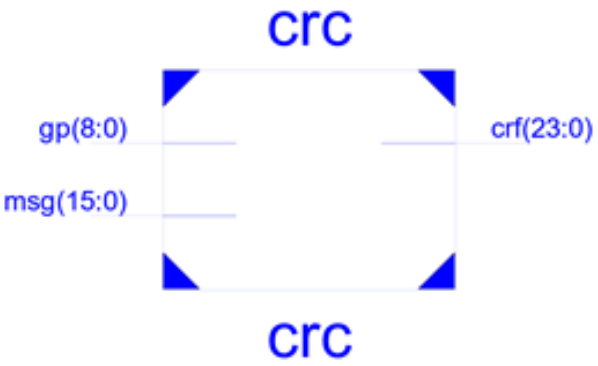
7.2. Code B: Adaptive Verilog code for CRC

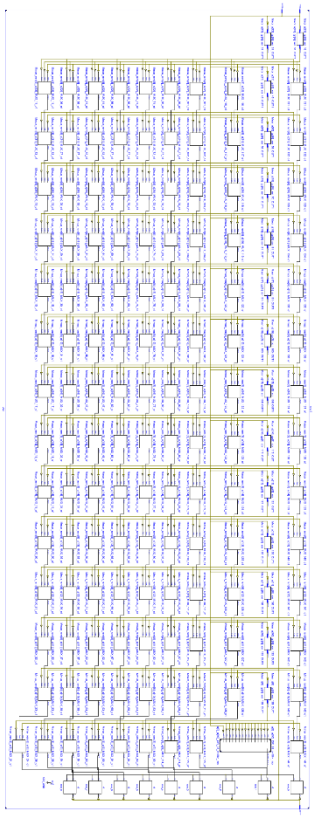
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Bangalore Institute of Technology
// Engineer: Neelanjan Goswami, Moka Saicharan & Mohd Ahmed Siddique
//
// Create Date:    17:39:46 03/26/2020
// Design Name:    CRC
// Module Name:    crc (Cyclic Redundancy Check)
// Project Name:   CRC
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//

// NEELANJAN, SAICHARAN AND AHMED
//////////////////////////////////////////////////////////////////////////////////
module crc(msg,gp,crf);
input [15:0]msg; // 16bit i/p
input [8:0]gp; // 8 bit crc
output [23:0] crf; // 16+9-1 = 24 bits

//reg [23:0] crcf;
wire [23:0] crcf;
reg [8:0] rem;
integer i,cnt,b,c,ct;
reg [23:0] a;

// appending zeros
always @ (msg,gp)
begin
for(i=23;i>=0;i=i-1)
begin
if(i>8)
a[i] = msg[(i-8)];
else
a[i] = 0;
end

/*Finding 1, then XORing with gp, till 9th bit*/
for(i=23;i>=8;i=i-1)
begin

if(a[i])
begin
a[i-:9]=a[i-:9]^gp[8:0];
end

end
end


assign crcf[23-:16]=msg[15:0];
assign crcf[7:0]=a[7:0];
assign crf = crcf;
endmodule
```

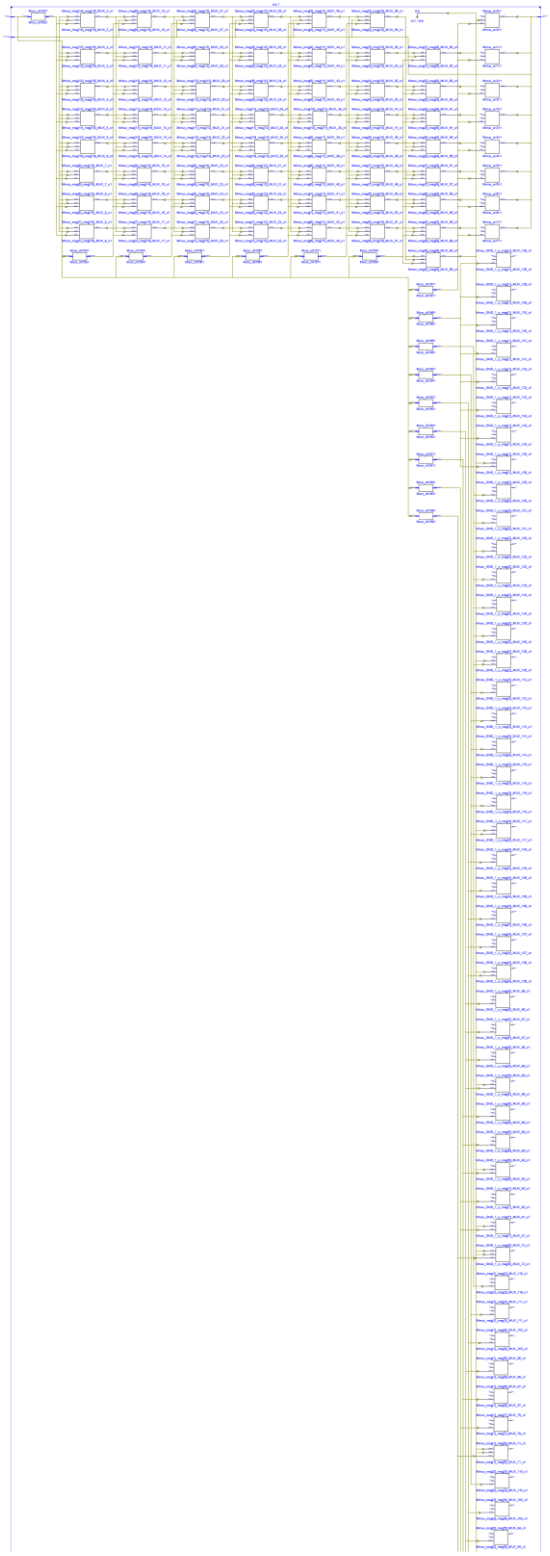
8. Schematics

8.1. RTL Schematic

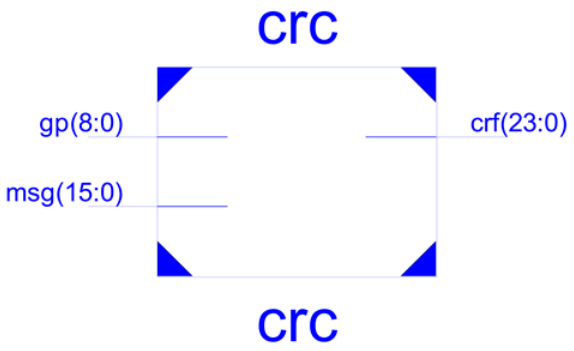
Code A: -

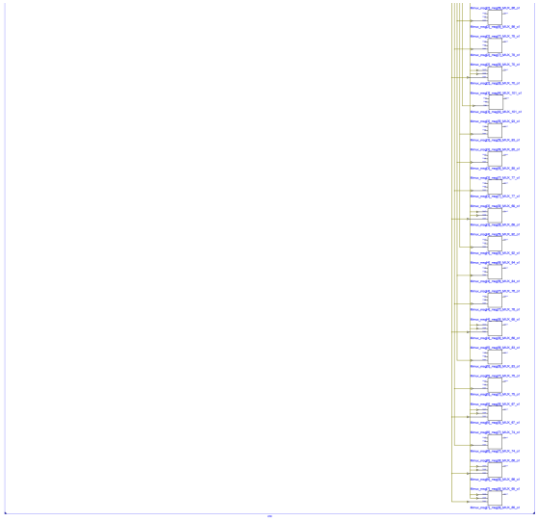


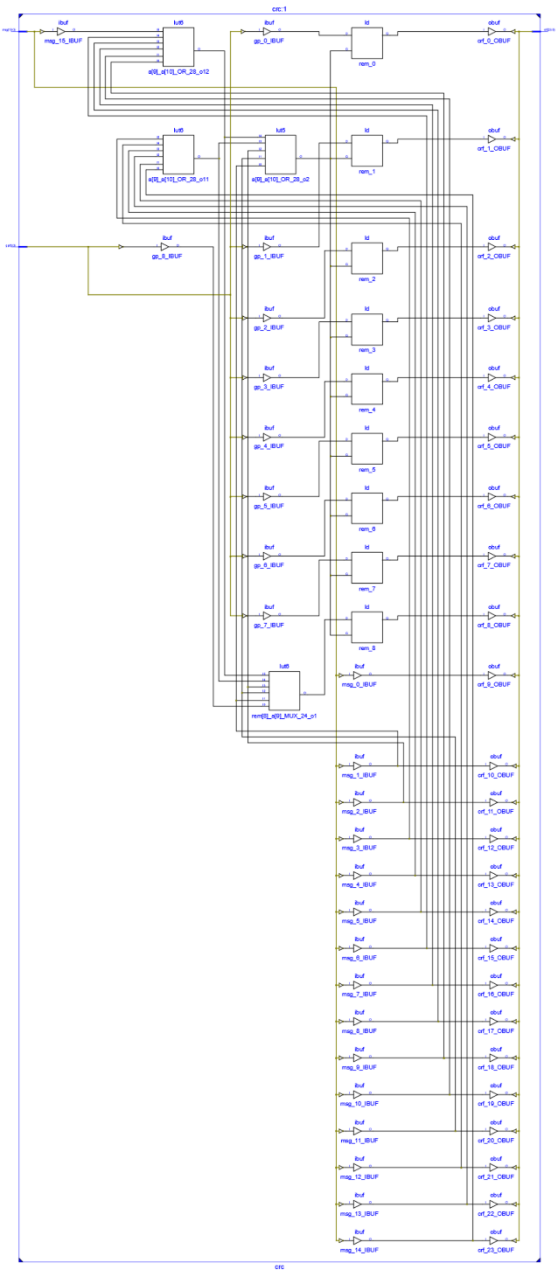




Code B: -

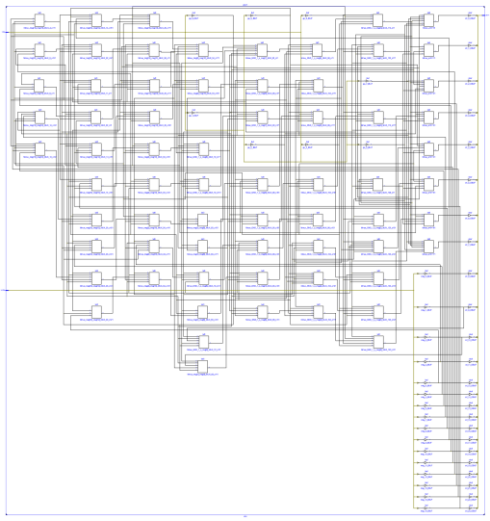






8.2. Technology Schematic

Code B:



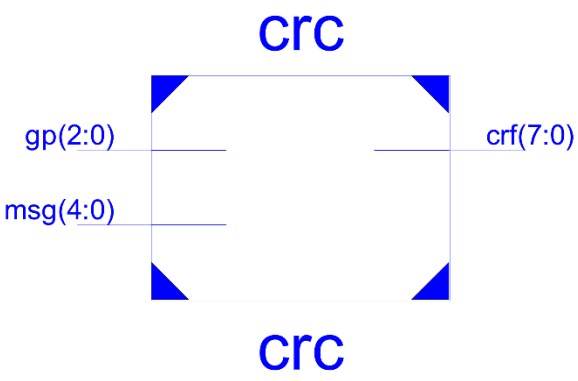
Code A:

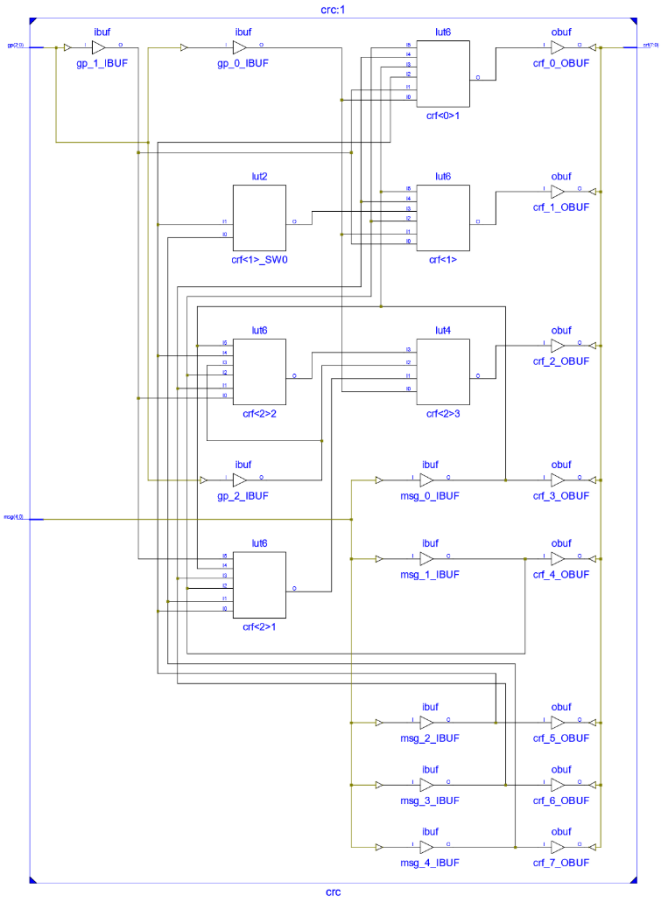
Simplification: -

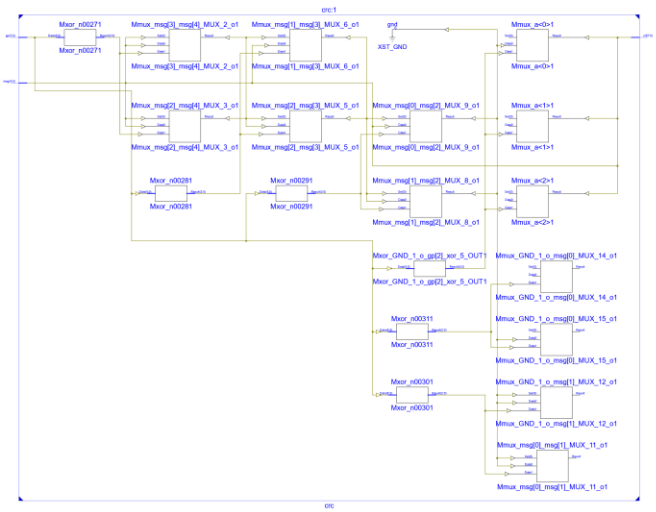
For scaling down, a 2-bit CRC's schematics too are shown: -

i.e. 5-bit Message bits and 3-bit Check bits

8.3. RTL Schematic







8.4. Technology Schematic

9. Applications

1. A multiple bits error correction method based on cyclic redundancy check codes.
2. In the RFID system, the application of error correction.
3. Detection and recovery of memory-resident corrupted data of mobile communication billing system based on cyclic redundancy check.
4. Enhanced error correction for satellite navigation message based on CRC codes.
5. Study of CRC-p Code Efficiency and Evaluation of Optimal CRC Code for VHF Maritime Ad-hoc wireless network communications.

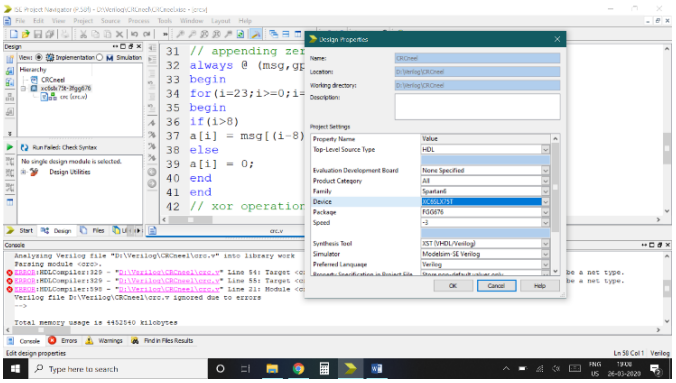
10. CONCLUSIONS

A Verilog code for an 8/12-bit cyclic redundancy check was created and tested. This code could also be extended for any number bit cyclic redundancy check by changing the bit size of registers taken as message input and generator polynomial

in the Verilog code. Different versions of the Verilog code for the same were also provided that could be used appropriately by the designer/student. Our study mainly emphasizes on bringing the operation of a CRC (Cyclic Redundancy Check) onto a hardware by writing a Verilog code that performs that very operation. The code was then translated into a basic circuit level representation with the use of RTL (Register Transfer Level) tools provided by Xilinx. Our study and code may or may not be the most efficient or compact hardware for a CRC but was rather done to make the viewers reading this paper understand the simplicity and the use of basic Verilog language to bring any real-life hardware/circuit for any kind of operation/application into reality. Software like Xilinx provide an easy to understand interface and variety of options to execute and implement any given Verilog code through simulations and RTL schematics respectively. They have single handedly brought the time and expense required to create any hardware/electronic circuits through the use of popular and easy to understand languages like Verilog, VHDL and technologies like RTL. Our main motive through this paper is to show this very creation of a CRC circuit/hardware through the steps we followed:

- Starting by understanding the need and the essence of this hardware,
- Analyzing the operation performed by this hardware,
- Preparing a model code with the use of simple Verilog statements,
- Generalizing the code and making it more flexible,
- Testing it through simulations (provided by ModelSim),
- Finally bringing the hardware into reality through the use of RTL technology.

11. System Configuration



ACKNOWLEDGEMENT

The paper and the work behind it would not have been poss ible without the outstanding guidance of my supervisor, Dr. A. B. Kalpana. Her enthusiasm, knowledge and exacting attention to detail have been an inspiration and kept my work on track from my first encounter with Verilog coding to the final draft of this paper. Dr. Sree Ranga Raju M. N., my professor at Bangalore Institute of Technology, have also looked over my transcriptions and answered with unfailing patience numerous questions about the hardware description language and semantics. I am also grateful for the insightful comments offered by the anonymous peer reviewers at my college's research block. The generosity and expertise of one and all have improved this study in innumerable ways and saved me from many errors; those that inevitably remain are entirely my own responsibility.

REFERENCES

1. Samir Palnitkar, ―Verilog HDL: A Guide to Digital Design and Synthesis", Pearson Education, Second Edition.
2. Kevin Skahill, ―VHDL for Programmable Logic‖, PHI/Pearson education, 2006.I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
3. Donald E. Thomas, Philip R. Moorby, ―The Verilog Hardware Description Language, Springer Science+Business Media, LLC, Fifth edition.R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.
4. Michael D. Ciletti, ―Advanced Digital Design with the Verilog HDL Pearson (Prentice Hall), Second edition.M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 198.
5. Padmanabhan, Tripura Sundari, ―Design through Verilog HDL, Wiley, 2016 Kass, R. E. and A. E. Raftery (1995). Bayes Factors. Journal of the American Statistical Association 90, 773–794.
6. The Verilog Hardware Description Language by Thomas, D . E . / Moorby, Philip R ., Fourth Edition, Published by Kluwer Academic Publishers, date Published: 05/1998, ISBN: 0792381661.
7. Burns C (1996) An architecture for a Verilog hardware accelerator. In: Proceedings of the IEEE international Verilog HDL conference.
8. Charlton C, Jackson D, Leng PH, Russell PC (1990) Modelling circuit delays in a demand driven simulator. Comput Electr Eng 20(4):309.
9. Cummings CE (2000) Nonblocking assignments in Verilog Synthesis, coding styles that kill! Synopsys User Group, San Jose (SNUG).
10. Gordon M (1995) The semantic challenge of Verilog HDL 1995. In: Proceedings of the 10th annual IEEE symposium on logic in computer science (LICS).
11. M. Ahmad, "Importance of Modeling and Simulation of Materials in Research", J. Mod. Sim. Mater., vol. 1, no. 1, pp. 1-2, Jan. 2018. DOI: https://doi.org/10.21467/jmsm.1.1.1-2