ISSN 2278-3091
Volume 9, No.4, July – August 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse232942020.pdf
https://doi.org/10.30534/ijatcse/2020/232942020

Design and Verification of novel classical error control codes using VERILOG Hardware Description Language

Hieu V. Dang [1], **Tung V. Nguyen**[2], **Manh Hoang**[2], **Viet Q. Tran**[2], **Nhan D. Nguyen**[3], **and Duc M. Nguyen**[4]
[1]Dept. of Information Technology Specialization, FPT University, Hanoi, Vietnam, hieudv2@fe.edu.vn
[2]ICT Department, FPT University, Hanoi, Vietnam, tungnvhe130151@fpt.edu.vn, manhhhe130294@fpt.edu.vn,viettqse06178@fpt.edu.vn
[3]Dept. of Biomedical Engineering, Sungkyunkwan University, Suwon, South Korea, nhannd@skku.edu
[4]Design and Development Center, G2Touch Company, PankyoSilicon Valley, Seongnam, Korea, nguyenmanhduc18@gmail.com

ABSTRACT

Nowadays, intelligent devices can be found anywhere around us, and which help our living more comfortable. Communication systems are their main modules, and which have been attracting many scientists around the world. However, communication systems are performed under many errors from environments and obstacles around us. To overcome obstacles and make trustable and stable transmission, error control codes are invented and to be a great research fields to improve the performance such as detecting and correcting capability. In this paper, various types of error control code such as three popular error detecting and correcting codes: parity-check, CRC, and Hamming codes are introduced detailly on those structures and applications. In addition, their system on chip design are implemented on VERILOG HDL. Finally, the test benches with many test cases are mentioned to verify the function of those codes.

Key words: Error control code, Parity check code, Cyclic redundancy code, Hamming code, Verilog, FPGA, System on Chip design.

1. INTRODUCTION

Communication systems play an important role on many modern systems which can be found anywhere around us [1]. A communication system consists of some basic modules such as source encoder, source decoder, channel encoder, channel decoder, modulator, demodulator as be shown in Figure 1 [2]. Here, we will cover hardware design and implementation of some major error control codes (ECC). Based on each types of storage medium, we will use different type of ECC. Basically, the idea of ECC is to insert more parity part into data message to construct a codeword which has ability to against errors. The most basic ECCs are parity-check [3], cyclic redundancy code [4], and Hamming code [5], and they are popularly used in many systems around us [6].
In this paper, we will introduce the construction and the applications of three above ECCs. In addition, the designs of those ECC in hardware system are proposed. We use VERILOG HDL (Hardware description language) for simulation and verification. The VERILOGcodes are conducted by Model Simulation 10.4a tool student version.

The paper is organized as follows. In Section 2, we review parity-check method and its VERILOGimplementation. Design of CRC code is given in Section 3. And in Section 4, Hamming code is explained. Finally, conclusion is discussed at Section 5.

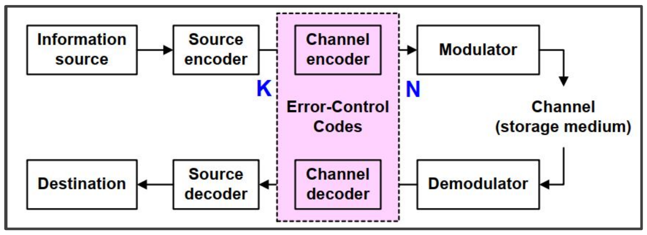
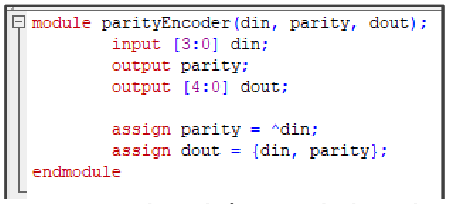


Figure 1: Basic modules of a communication system.

2. DESIGN AND VERIFICATION OF PARITY CHECK CODE

A. Parity-check code

In communication protocol, the simplest method for detecting error in binary string is parity-code. Theoretically, parity-code is to insert parity-bit to check the number of bits '1' or '0' in binary string is even or odd. In this section, we are going to design the parity-encoder and parity-checker.


```
module parityEncoder(din, parity, dout);
        input [3:0] din;
        output parity;
        output [4:0] dout;

        assign parity = ^din;
        assign dout = {din, parity};
endmodule
```
Figure 2: Verilog code for parity-check encoder.

B. Design of Parity-check encoder module

The purpose of parity-encoder module is to output the codeword from an input binary string, the input of parity-encoder has a fixed-length, and the output of parity-encoder will be the concatenation of the input and its corresponding parity-check bit. Since the parity-check bit is 0 if the number of bits '1' in binary string is even and is 1 if the number of bits '1' in binary string is odd, we can use XOR operation of all bits in binary string to calculate parity-check bit. The VERILOGimplementation is given in Figure 2 where 'din' stands for input message, 'parity' stands for the parity-bit, and 'dout' will be output codeword. Figure 3 is the schematic trace corresponding to parity-encoder. In addition, we also made a testbench as Figure 4 with some testcases and the timing diagram which help us to verify our implementation.

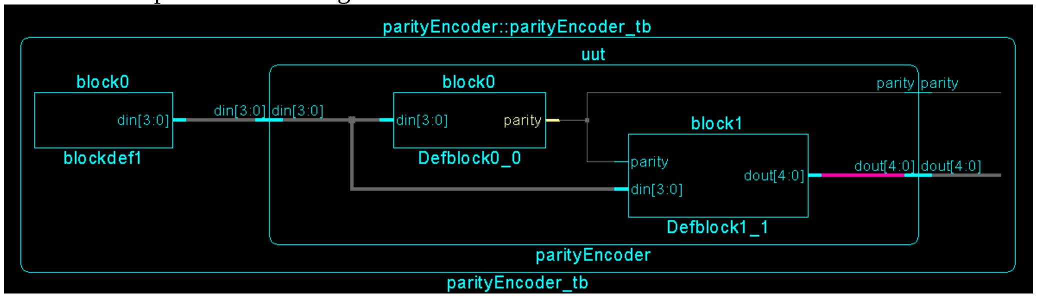


Figure 3: Schematic tracer of parity-check encoder.

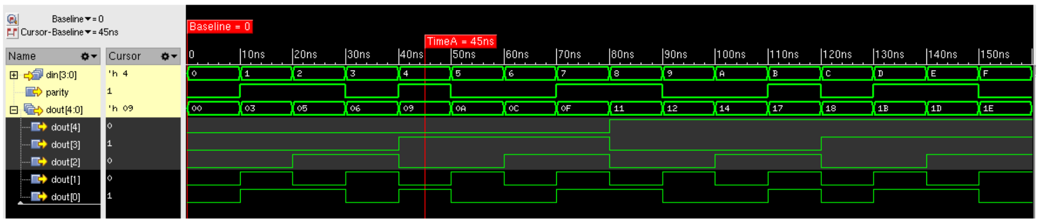


Figure 4: Simulation results for parity-check encoder with a testbench.

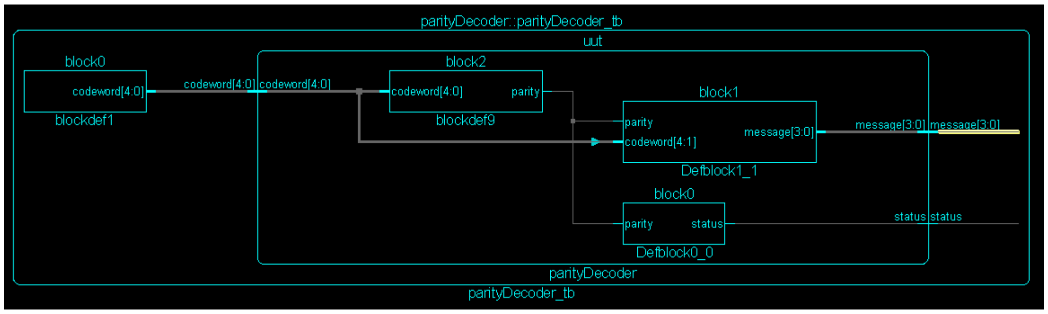


Figure 5: Schematic tracer of parity-check decoder.

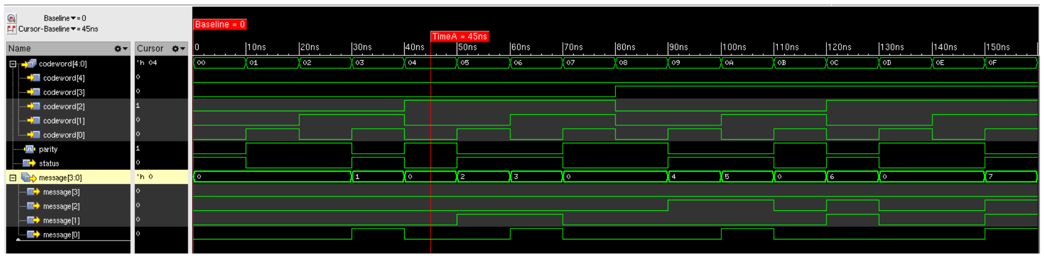


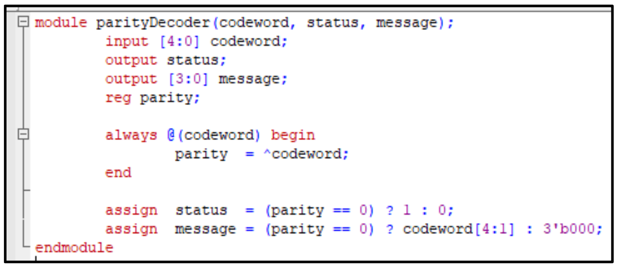
Figure 6: Simulation results for parity-check decoder with a testbench.

C. Design of Parity-check decoder module

The purpose of parity-checker is to verify a codeword is correct or not and output the correct binary string. The input of parity-checker is a codeword and the output is status bit to determine codeword is valid or not and a decoded binary string, which will be set to all zeros if codeword is not correct. Since as definition, the codeword is combination of binary string and its parity-check bit, a codeword is a valid if the output of parity-checker is zero, and the decoded message is just subpart by removed a parity part from codeword. The VERILOG implementation of parity-decoder is as Figure 7.

Figure 5 is the schematic trace corresponding to parity-decoder. In addition, we also made a testbench as Figure 6 with some testcases and the timing diagram which help us to verify the VERILOG implementation.

As the implementation result and analysis, the error in codeword can be in the bits from data or bit from parity-check and it only can detect error in case of the number of error bits is odd (1,3, or 5... error bits). Hence, parity-check is very simple method but not much efficient. Hence, parity-method is suitable for some applications where its protocol has slow transfer speed and the number of bits are not much such as UART protocol, used in many hardware applications where an operation can be repeated in case of difficulty for example SCSI, PCI buses, or in many microprocessor instruction caches such as I-cache.



```
module parityDecoder(codeword, status, message);
        input [4:0] codeword;
        output status;
        output [3:0] message;
        reg parity;

        always @(codeword) begin
                parity  = ^codeword;
        end

        assign  status  = (parity == 0) ? 1 : 0;
        assign  message = (parity == 0) ? codeword[4:1] : 3'b000;
endmodule
```

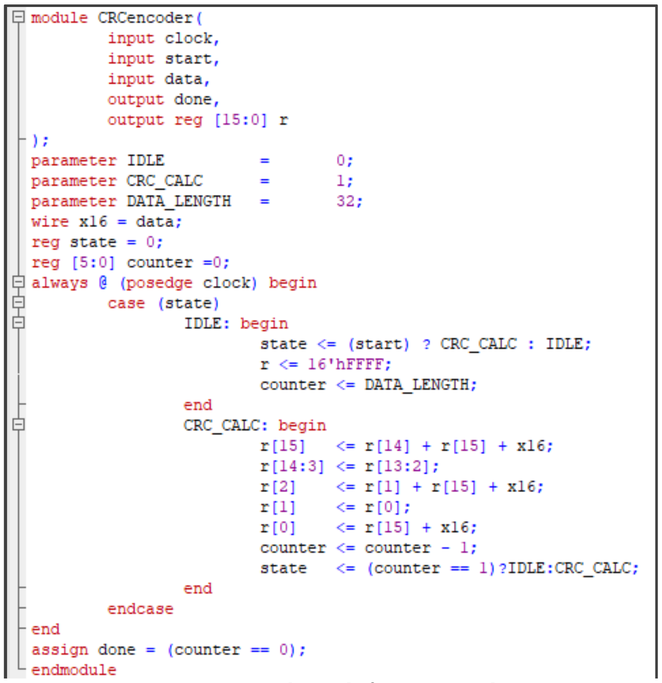
Figure 7: Verilog code for parity-check decoder.

3. DESIGN AND VERIFICATION OF CRC CODE

A. Cyclic Redundancy code

To general the parity-check method as explanation in Section 2, Cyclic redundant code (CRC) will be explained in this section. In CRC, the redundancy is attached to message to make the codeword, redundancy is calculated based on the reminder of polynomial division.

CRC is more efficient method than parity-check and it is popular since simply to implement in hardware, easy to analysis mathematically, and very good at detecting common errors caused by noise in transmission channel. Consequently, we can find CRC in many communication protocols with high transfer data or high speed of transfer such as CAN, Ethernet, or RF15693, ... The CRC-16 is used in USB device, CRC32 is used in Ethernet.



```
module CRCencoder(
          input clock,
          input start,
          input data,
          output done,
          output reg [15:0] r
);
parameter IDLE          =       0;
parameter CRC_CALC      =       1;
parameter DATA_LENGTH   =       32;
wire x16 = data;
reg state = 0;
reg [5:0] counter =0;
always @ (posedge clock) begin
          case (state)
                    IDLE: begin
                              state <= (start) ? CRC_CALC : IDLE;
                              r <= 16'hFFFF;
                              counter <= DATA_LENGTH;
                    end
                    CRC_CALC: begin
                              r[15]    <= r[14] + r[15] + x16;
                              r[14:3]  <= r[13:2];
                              r[2]     <= r[1] + r[15] + x16;
                              r[1]     <= r[0];
                              r[0]     <= r[15] + x16;
                              counter <= counter - 1;
                              state    <= (counter == 1)?IDLE:CRC_CALC;
                    end
          endcase
end
assign done = (counter == 0);
endmodule
```

Figure 8: Verilog code for CRC encoder.

B. Design of CRC encoder module

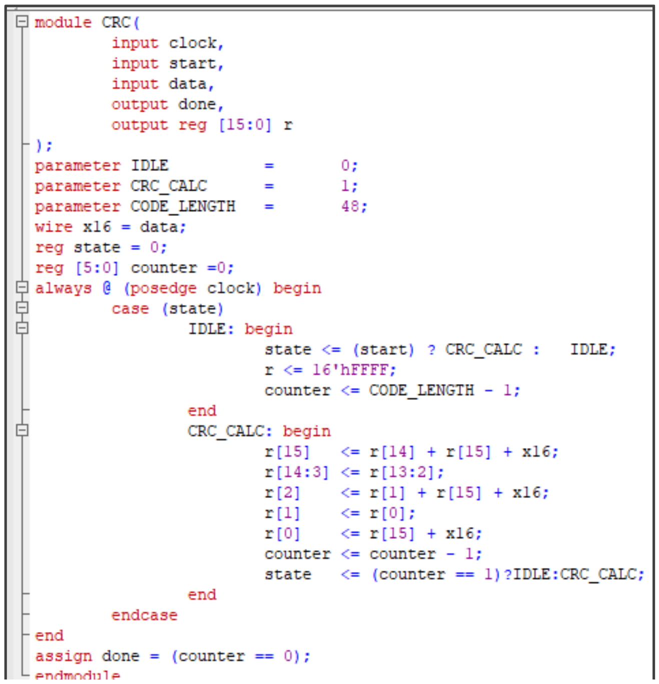
In this section, we implement CRC-16 encoder where generator polynomial is x16+x15+x2+1 as Figure 8. Since the degree of generator polynomial is 16, the output has length of 16 and is denoted as 'r'. The pipeline method will be used for inputs with one bit coming each timing clock, input bit is denoted as 'data'. In this implementation, we use Finite State Machine method with two states 'IDLE' and 'CRC_CALC'. In 'IDLE' state, we initialize 'r' as 16'hFFFF (length 16 of all '1'-bit binary string). In 'CRC-CAL' state, the state is updated for each coming time of data and the counter will be used to determine the full length of message we want to encode. When all the bits of message coming, the 'done' will be set to 1 and the value of 'r' at that time will be the final redundancy to be attached into message to construct the codeword.

Figure 10 is the schematic trace corresponding to CRC-encoder. In addition, we also made a testbench as Figure 11 with message is 32'h03_01_02_03 (32 bits length), the redundancy result 'r' will be 16'h303A when done = 1.

C. Design of CRC decoder module

Since the CRC-checker and CRC-encoder are based on the polynomial division with pre-defined generator polynomial, the implementation of CRC-checker is almost same as CRC-encoder as Figure 9. The full length of input message and the expected output values are the different. In this implementation, we use '48' as the full length of input message, and the expected value 'r' is all zeros. Whenever 'done' is set to 1, if 'r' is not all zeros, the codeword is invalid.

Figure 12 is the schematic trace corresponding to CRC-checker. In addition, we also made a testbench as Figure 13 with codeword input is 48'h03_01_02_03_30_3a, the redundancy result 'r' will be 16'h0000 when done = 1.



```
module CRC(
          input clock,
          input start,
          input data,
          output done,
          output reg [15:0] r
);
parameter IDLE          =       0;
parameter CRC_CALC      =       1;
parameter CODE_LENGTH   =       48;
wire x16 = data;
reg state = 0;
reg [5:0] counter =0;
always @ (posedge clock) begin
          case (state)
                    IDLE: begin
                              state <= (start) ? CRC_CALC :   IDLE;
                              r <= 16'hFFFF;
                              counter <= CODE_LENGTH - 1;
                    end
                    CRC_CALC: begin
                              r[15]    <= r[14] + r[15] + x16;
                              r[14:3]  <= r[13:2];
                              r[2]     <= r[1] + r[15] + x16;
                              r[1]     <= r[0];
                              r[0]     <= r[15] + x16;
                              counter <= counter - 1;
                              state    <= (counter == 1)?IDLE:CRC_CALC;
                    end
          endcase
end
assign done = (counter == 0);
endmodule
```

Figure 9: Verilog code for CRCchecker.

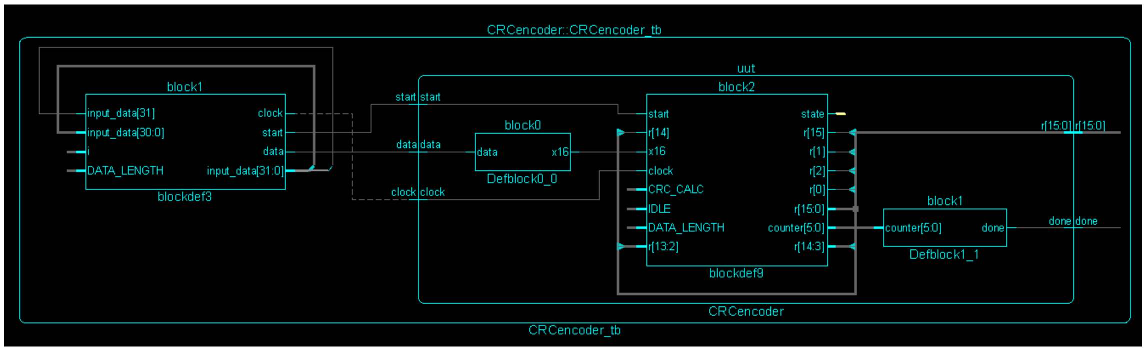
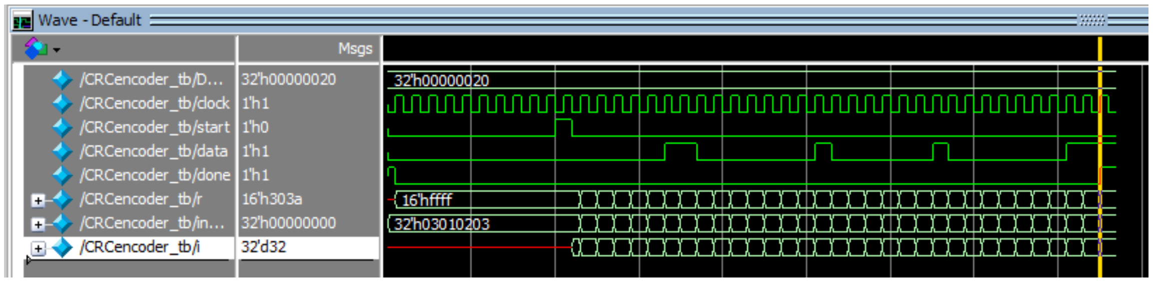


Figure 10: Schematic tracer of CRC encoder.


Wave - Default

	Msgs
/CRCencoder_tb/D...	32'h00000020
/CRCencoder_tb/clock	1'h1
/CRCencoder_tb/start	1'h0
/CRCencoder_tb/data	1'h1
/CRCencoder_tb/done	1'h1
/CRCencoder_tb/r	16'h303a
/CRCencoder_tb/in...	32'h00000000
/CRCencoder_tb/i	32'd32

Figure 11: Simulation results for CRC encoder with a testbench.

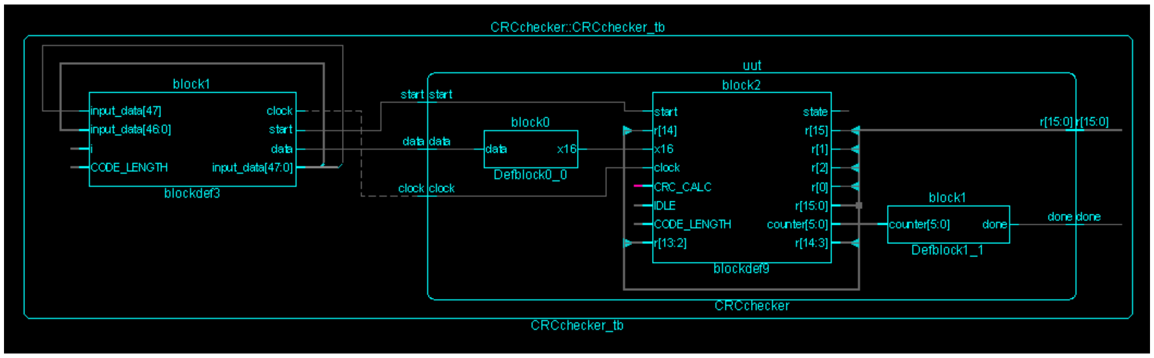
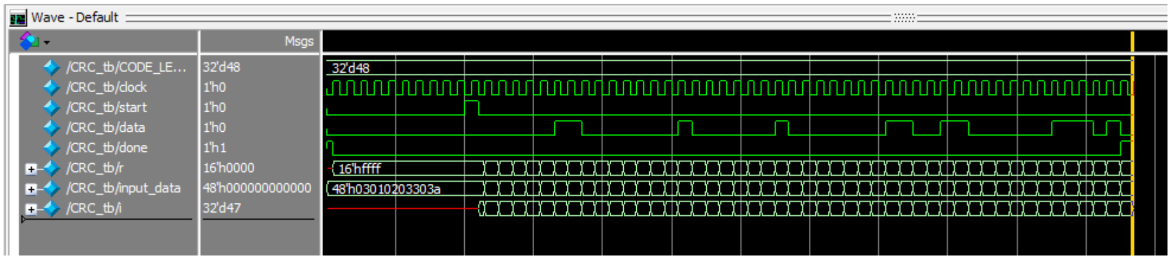


Figure 12: Schematic tracer of CRC checker.


Wave - Default

	Msgs
/CRC_tb/CODE_LE...	32'd48
/CRC_tb/clock	1'h0
/CRC_tb/start	1'h0
/CRC_tb/data	1'h0
/CRC_tb/done	1'h1
/CRC_tb/r	16'h0000
/CRC_tb/input_data	48'h000000000000
/CRC_tb/i	32'd47

Figure 13: Simulation results for CRC checker with a testbench.

4. DESIGN AND VERIFICATION OF HAMMING CODE

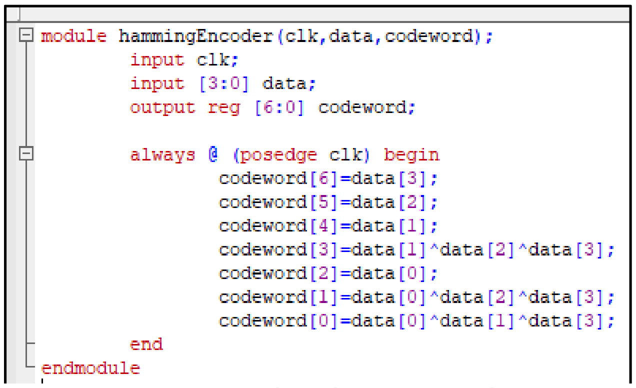
A. Hamming code

Since the Parity-check and CRC code are error detecting codes and unable to correct any error, Hamming code is invented by Richard Hamming to be the first error correcting code by based on syndrome calculation of error. Further analysis shows that Hamming code is a linear code, perfect 1-error correction, and 2-errors detection. Mathematically, Hamming code is constructed based on a generator matrix and the syndrome will be calculated based on parity-check matrix which has strong relation to generator matrix, and they can be transformed from each other. There is a class of Hamming code, its applications can be found everywhere in modern communication system such as Punched card reader, Modems, Computer memory, Embedded system, and processor, ... In this section, we discuss the smallest length [7,4,3] code.

B. Design of Hamming encoder module

In [7,4,3] Hamming code, '7' stands for codeword length, '4' stands for data length, and 3 is Hamming distance which is error detecting and correcting capable. Consequently, the input of module is 'data' with length 4, the output 'codeword' has length '7' and is calculated based on 'data' and parity equations as given in Figure 14.

Figure 15 is the schematic trace corresponding to Hamming encoder. In addition, we also made a testbench as Figure 16 with all the possible case for 'data' and 'codeword'.



```
module hammingEncoder(clk,data,codeword);
        input clk;
        input [3:0] data;
        output reg [6:0] codeword;

        always @ (posedge clk) begin
                codeword[6]=data[3];
                codeword[5]=data[2];
                codeword[4]=data[1];
                codeword[3]=data[1]^data[2]^data[3];
                codeword[2]=data[0];
                codeword[1]=data[0]^data[2]^data[3];
                codeword[0]=data[0]^data[1]^data[3];
        end
endmodule
```

Figure 14: Verilog code Hamming encoder.

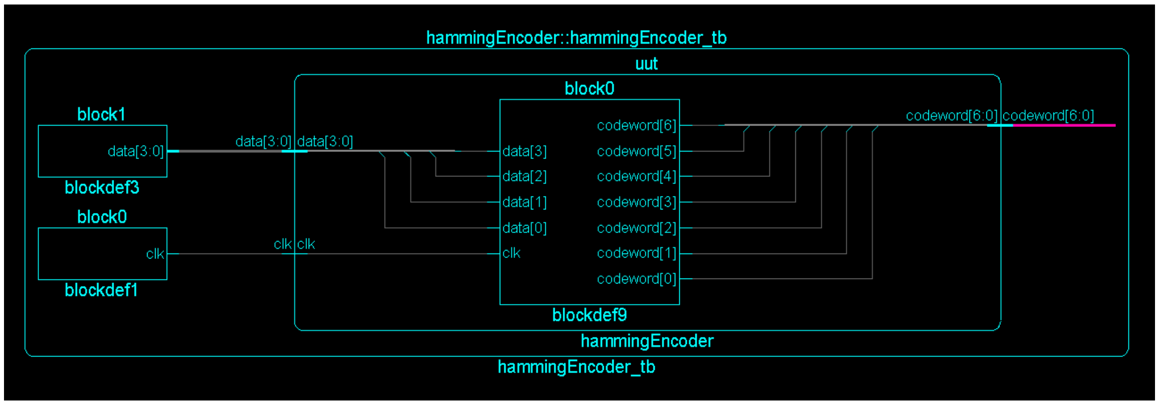


Figure 15: Schematic tracer of Hamming encoder.

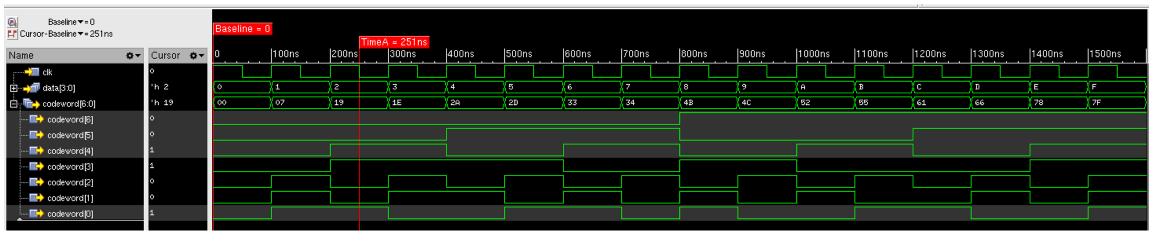


Figure 16: Simulation results for Hamming encoder with a testbench.

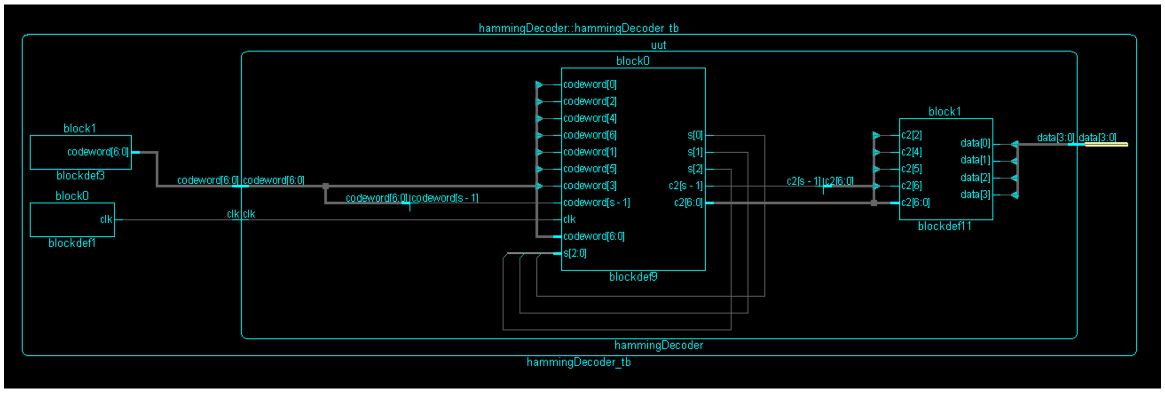


Figure 17: Schematic tracer of Hamming decoder.

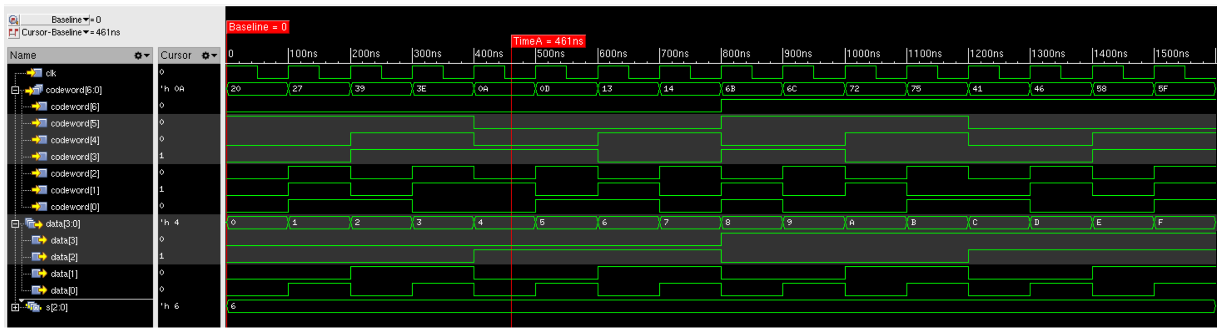
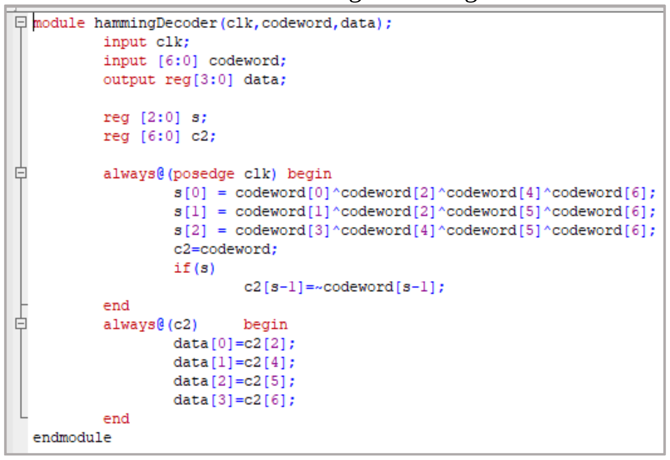


Figure 18: Simulation results for Hamming decoder with a testbench.

C. Design of Hamming decoder module

In Hamming decoder, the input will be the codeword message with length '7' and the output will be the correct data with length '4'. Inside the module, syndrome is denoted by 's' and based on value of 's', 'data' will be created suitably from input codeword. The VERILOG implementation is given as Figure 19.

Figure 17is the schematic trace corresponding to Hamming decoder. In addition, we also made a testbench as Figure 18 with two test cases. First test case is with correct codewords, and second test case is with the codewords with 1-bit error. The results of two test cases are given in Figure 20.

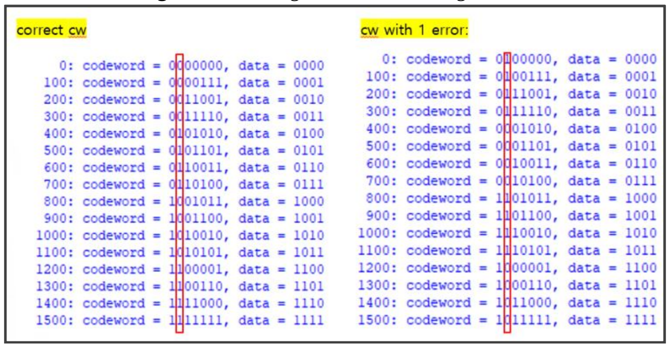


```
module hammingDecoder(clk,codeword,data);
        input clk;
        input [6:0] codeword;
        output reg[3:0] data;

        reg [2:0] s;
        reg [6:0] c2;

        always@(posedge clk) begin
                s[0] = codeword[0]^codeword[2]^codeword[4]^codeword[6];
                s[1] = codeword[1]^codeword[2]^codeword[5]^codeword[6];
                s[2] = codeword[3]^codeword[4]^codeword[5]^codeword[6];
                c2=codeword;
                if(s)
                        c2[s-1]=~codeword[s-1];
        end
        always@(c2)     begin
                data[0]=c2[2];
                data[1]=c2[4];
                data[2]=c2[5];
                data[3]=c2[6];
        end
endmodule
```

Figure 19: Verilog code for Hamming decoder.



```
correct cw                                         cw with 1 error:

   0: codeword = 0000000, data = 0000                 0: codeword = 0100000, data = 0000
 100: codeword = 0000111, data = 0001               100: codeword = 0100111, data = 0001
 200: codeword = 0011001, data = 0010               200: codeword = 0111001, data = 0010
 300: codeword = 0011110, data = 0011               300: codeword = 0111110, data = 0011
 400: codeword = 0101010, data = 0100               400: codeword = 0001010, data = 0100
 500: codeword = 0101101, data = 0101               500: codeword = 0001101, data = 0101
 600: codeword = 0110011, data = 0110               600: codeword = 0010011, data = 0110
 700: codeword = 0110100, data = 0111               700: codeword = 0010100, data = 0111
 800: codeword = 1001011, data = 1000               800: codeword = 1101011, data = 1000
 900: codeword = 1001100, data = 1001               900: codeword = 1101100, data = 1001
1000: codeword = 1010010, data = 1010              1000: codeword = 1110010, data = 1010
1100: codeword = 1010101, data = 1011              1100: codeword = 1110101, data = 1011
1200: codeword = 1100001, data = 1100              1200: codeword = 1000001, data = 1100
1300: codeword = 1100110, data = 1101              1300: codeword = 1000110, data = 1101
1400: codeword = 1111000, data = 1110              1400: codeword = 1011000, data = 1110
1500: codeword = 1111111, data = 1111              1500: codeword = 1011111, data = 1111
```

Figure 20: Two testbenches for Hamming decoder.

5. CONCLUSION

The paper has discussed about three basic but important error detection codes: Parity-check, CRC, and correction code: Hamming. Their mathematical explanation, application, implementation, as well as test bench for verification are given in detail.

Insystem on chip designs, error detection and correction codes are popularly used in many communication and processor systems. The implementation based on VERILOGare given to help students, researcher, and engineers a lot as a good reference in their works. The testbench files are too long to be attached, authors are ready to share under request.

ACKNOWLEDGEMENT

This work is supported by the FPT University, Hanoi, Vietnam; Sungkyunkwan University, Suwon, Republic of Korea; and G2Touch Company, Seongnam City, Republic of Korea.

Conflict of Interest

On behalf of all authors, the corresponding author declares that there is no conflict of interest.

REFERENCES

1. Nguyen, D.M., Kim, S. "**Quantum Key Distribution Protocol Based on Modified Generalization of Deutsch-Jozsa Algorithm in d-level Quantum System**", Int. J. Theor. Phys. vol. 58(1), pp. 71-82, 2019.
2. Kiran, K., et. al. "**Design and Implementation of Efficient Cryptographic Arithmetic based on Reversible logic and Vedic Mathematics**", Int. J. of Advanced Trendsin Computer Science and Engineering. vol. 9(2), 2020. https://doi.org/10.30534/ijatcse/2020/21922020
3. Nguyen, D.M., Kim, S. "**A novel construction for quantum stabilizer codes based on binary formalism**". Int. J. of Modern Phys. B. vol. 33(8), 2020.
4. Dhijaj, J., et. al. "**Design and verification of an automated CRC engine using VERILOG HDL**". Int. J. of Advanced Research in Electrical, Electronics and Instrumentation Engineering. vol. 2(6), 2013.
5. Achmad, F., et. al. "**Bit Error Detection and Correction with Hamming Code Algorithm**". Int. J. of Scientific Research in Science, Engineering and Technology. vol. 3(1), 2017.
6. Binh, A. N., et. al. "**A Novel Framework for Simulation of Quantum Information System**",Int. J. of Advanced Trends in Computer Science and Engineering. vol. 9(2), 2020 https://doi.org/10.30534/ijatcse/2020/129922020