논리설계 및 실험 003반
프로젝트 최종 보고서

Verilog를 사용한 error correction이 가능한 serial communication link의 구현

30조

2021-15818 자유전공학부 강채림

2022-13874 전기정보공학부 유준상

2023년 12월 19일

1. 개요

본 프로젝트는 verilog를 통해 자가 오류 정정이 가능한 serial communication link를 구현한 TRX_TOP의 합성을 목표로 한다. 4bit의 데이터에 4bit의 parity bit을 삽입한, 총 8bit의 코드를 직렬화하여 송수신하는 UART 통신을 진행한다. 이 때, 송신하는 과정에서 1bit의 error가 발생해도 parity bit을 통해 수신하는 과정에 error을 정정해 알맞은 4bit의 데이터를 수신할 수 있도록 한다. 전체 과정은 크게 5가지로 나눌 수 있는데, data_in이 Hamming Encoder -〉 Error Injection -〉 UART Transmitter -〉 UART Receiver -〉 Hamming Decoder 순으로 거쳐 data_out으로 나온다.

2. 코드 설명 (주석에 코드 설명 있음)

2.1 Hamming Encoder

작동원리: 클록 신호마다 4bit 입력 데이터에 4bit의 parity bit를 추가해 8bit 데이터를 출력한다.

```
1
2  module Hamming_Encoder(
3      output [7:0] data_out,
4      input [3:0] data_in,
5      input clk, rstn
6      );
7
8  // **** TODO **** //
9  reg [7:0] c;
10
11 always @ (posedge clk or negedge rstn) begin
12     if (~rstn) begin // if reset, set every bit to 0
13         c = 8'b00000000;
14     end
15     else begin
16         c[0] = (data_in[0] ^ data_in[1] ^ data_in[3]); // P1
17         c[1] = (data_in[0] ^ data_in[2] ^ data_in[3]); // P2
18         c[2] = data_in[0]; // D1
19         c[3] = (data_in[1] ^ data_in[2] ^ data_in[3]); // P3
20         c[4] = data_in[1]; // D2
21         c[5] = data_in[2]; // D3
22         c[6] = data_in[3]; // D4
23         c[7] = (data_in[0] ^ data_in[1] ^ data_in[2]); // P4
24     end
25 end
26
27 assign data_out = c;
28 // ************** //
29
30 endmodule
31
```

그림 1 Hamming_Encoder.v 코드

2.2 Error Injection

작동원리: 8bit 데이터, btn1, btn2를 입력으로 받고 btn1,btn2의 조작에 따라 데이터에 에러를 주입한다. btn1과 btn2가 모두 1일 경우, c의 첫 번째 비트와 다섯 번째 비트를 반전시키고 btn1만 1일 경우 c의 첫 번째 비트를 반전, btn2만 1일 경우 c의 다섯 번째 비트를 반전시킨다. 두 버튼 모두 0인 경우 데이터에는 변경이 없다.

```
1
2  module Error_Injection(
3          output [7:0] data_out,
4          input [7:0] data_in,
5          input btn1, btn2,
6          input clk, rstn
7      );
8
9  // **** TODO **** //
10 reg [7:0] c;
11
12 always @ (posedge clk or negedge rstn) begin
13     if (~rstn)  c = 8'b00000000; // if reset, set every bit to 0
14     else begin
15         c = data_in;
16         if (btn1 && btn2) begin // bit1, bit5 error
17             c[0] = ~c[0];
18             c[4] = ~c[4];
19         end else if (btn1) begin // bit1 error
20             c[0] = ~c[0];
21         end else if (btn2) begin // bit5 error
22             c[4] = ~c[4];
23         end else ; // no error
24     end
25 end
26 assign data_out = c;
27 // ************** //
28
29 endmodule
30
```

그림 2 Error_Injection.v 코드

2.3 UART Transmitter

작동원리: 입력으로 데이터 data_bus, 전송 가능 여부 b_ready, 데이터 로드 여부 load_data, 시작 신호 여부 t_init, 클럭 clk, 리셋 rstn을 받는다. 이 모듈은 idle, waiting, sending 세 가지 상태를 가지며 상태에 따라 데이터를 로드하고 전송한다. shift_reg는 데이터와 함께 시작비트를 포함하는 레지스터이다. 또한 bit_cnt를 통해 전송된 비트의 수를 추적하고 모든 비트가 전송되면 clear 신호를 통해 bit_cnt를 초기화하고 idle 상태로 돌아간다.

```
module UART_Transmitter (
    output serial_out,
    input [7:0] data_bus,
    input b_ready,
    input load_data,
    input t_init,
    input clk,
    input rstn
);

parameter idle = 2'b00;
parameter waiting = 2'b01;
parameter sending = 2'b10;

reg [7:0] data_reg;
reg [8:0] shift_reg; // shift_reg[0] is initialized as 1 and changes to start_bit(0) during start signal
reg load_shift; // Signal to move data_reg into shift_reg
reg [1:0] state, next_state;
reg [3:0] bit_cnt; // Count of transmitted bits; switches to idle state when it reaches 1001
reg clear; // Signal to reset bit_cnt to 0 after transmitting the last bit
reg shift; // Signal to shift the contents of shift_reg


assign serial_out = shift_reg[0];

initial begin
    clear = 0;
    load_shift = 0;
    shift_reg = 9'b111111111; //Initializing shift_reg with 1, means no data input
    bit_cnt = 4'b0000;
end

 assign serial_out = shift_reg[0];

// Combinational logic to determine next_state based on the current state
always @(*) begin

    load_shift = 0;
    clear = 0;
    shift = 0;

    next_state = state;
    case(state)
        idle: begin

            if (b_ready) begin
                next_state = waiting;
            end
        end
        waiting: begin
            if (t_init) begin
                load_shift = 1;
                next_state = sending;
            end
        end
```

```
        sending: begin
            if (bit_cnt < 8) begin
                shift = 1;
            end else begin
                next_state = idle;
                clear = 1;
            end
        end
        default: next_state = idle;
    endcase
end

//State transition logic
always @(posedge clk or negedge rstn) begin

    if (!rstn) begin
        state <= idle;
    end else begin
        state <= next_state;
    end
end

//Logic to control the shift_reg
always @(posedge clk or negedge rstn) begin

    if (!rstn) begin
        shift_reg = 9'b111111111;
        //Reset shift_reg with 1
```

```
always @(posedge clk or negedge rstn) begin

   if (!rstn) begin
       shift_reg = 9'b111111111;
       //Reset shift_reg with 1
   end else begin
       if(load_data) begin
               data_reg = data_bus;
               //Load data_bus into data_reg when load_data is asserted
       end

       if (load_shift) begin
           shift_reg = {data_reg, 1'b0};
           //Load data_reg into shift_reg along with the start bit
       end else if (shift) begin
           shift_reg = {1'b1,shift_reg[8:1]};
           //Shift data for transmission
           bit_cnt = bit_cnt + 1;
           //Increment bit count upon shifting
       end else if (clear) begin
           bit_cnt = 0;
           //Reset bit count to 0 after transmitting all bits
       end
   end
end

endmodule
```

2.4 UART Receiver

작동원리: 이 모듈은 idle, starting, receiving 세 가지 상태로 구성된다. 각 상태에 따른 동작은 다음과 같다.

idle: 수신기가 데이터를 받기 위해 대기하는 상태이다. 시작 비트를 기다리며, serial_in이 0으로 감지되면 다음 상태로 전환된다.

starting: 시작 상태로, 시작 비트를 받고 있는 상태이다. sample_cnt가 3'b011이고, serial_in이 0인 경우를 데이터 입력이 들어오는 것이라고 판단하여 다음 상태로 전환한다.

receiving: 데이터를 수신하고 있는 상태로, 시리얼로부터 비트를 받아 shift_reg에 저장한다. sample_cnt가 특정 패턴(3)을 감지하면, 비트를 받아들이고, 9비트를 받은 후에는 not_ready_out 신호를 활성화하여 데이터가 수신되었음을 표시하고, 다음 clk 신호에서 data_reg에 수신한 데이터를 저장하게 된다.

```
1  module UART_Receiver(
2      output reg [7:0] data_reg,
3      output reg not_ready_out, error1, error2,
4      input serial_in,
5      input not_ready_in,
6      input sample_clk,
7      input rstn
8      );
9
10 parameter idle = 2'b00;
11 parameter starting = 2'b01;
12 parameter receiving = 2'b10;
13
14 reg [7:0] shift_reg;
15 reg [2:0] sample_cnt;
16 reg [3:0] bit_cnt;
17 reg [1:0] state, next_state;
18 reg shift, load;
19
20 initial begin
21     load = 0;
22     error1 = 1'b0;
23     error2 = 1'b0;
24     not_ready_out = 1'b1; //Initialize not_ready_out as high
25     data_reg = 8'b00000000; //Initialize data_reg to 0
26 end
27
```

```
always @(*) begin
    next_state = state;

    case (state)
        idle: begin
            //Transition to the starting state if the incoming serial input is 0
            if (serial_in == 0) next_state = starting;
            else next_state = idle;
        end
        starting: begin
            //Transition to the receiving state after a sequence of samples and valid start bit
            if (sample_cnt == 3'b011 && serial_in == 0) next_state = receiving;
            else if (serial_in == 1) next_state = idle;
            else next_state = starting;
        end
        receiving: begin
            //Stay in the receiving state until load signal becomes active
            if (load==1) begin
                next_state = idle;
            end else next_state = receiving;
        end
        default: next_state = idle;
    endcase
end

always @(posedge sample_clk or negedge rstn) begin
    if(!rstn) begin
        state <= idle;


        sample_cnt <= 3'b000;
        bit_cnt <= 4'b0000;

    end
    else begin
        state <= next_state;

        if (state == idle) begin
            load <=0;
        end

        else if (state == starting) begin
            sample_cnt = sample_cnt + 3'b001;
            //Increment sample count by 1 to move towards the receiving state
            load <=0;
        end



        else begin //state == receiving
            sample_cnt = sample_cnt + 3'b001;
            if (sample_cnt == 3) begin //Receive data when sample_cnt is 3.
                bit_cnt = bit_cnt + 4'b1;
                if (bit_cnt == 4'b1001) begin//Received stop bit (1) after receiving 9 bits
                    not_ready_out = 1'b0;
                    load = 1; //Transition to idle on the next clock cycle
                    data_reg <= shift_reg; //Load shift_reg into data_reg when all data is received
```

```
                    bit_cnt = 0;
                    sample_cnt = 3'b000;
                end else begin
                    shift_reg = {serial_in, shift_reg[7:1]};
                end
            end
        end

    end
end

endmodule
```

2.5 Hamming Decoder

작동원리: 해밍 코드화 후 UART 통신이 진행된 8bit 데이터를 입력으로 받아 지정한 레지스터인 A,B,C,D의 값에 따라 error의 개수-1bit or 2bit-를 판단한다. 그 후, error가 1bit일 경우, A, B, C, D를 이용해 error를 자가 수정한 후, 최종적으로 송신부에서 송신하려던 4bit의 데이터를 data_out으로 출력한다.

```
1  module Hamming_Decoder(
2      output [3:0] data_out,
3      output err_uncorrectable, err_correctable,
4      input [7:0] data_in,
5      input clk, rstn
6      );
7  reg A, B, C, D;
8  reg err_single_bit, err_double_bit;
9  reg [3:0] c;
10
11 // Syndrome(A, B, C, D) calculation
12 always @(posedge clk or negedge rstn) begin
13
14     if (~rstn) begin
15         c = 4'b0000;
16         err_double_bit = 0;
17         err_single_bit = 0;
18     end
19     else begin
20         c = {data_in[6], data_in[5], data_in[4], data_in[2]};
21         //Syndrome calculation
22         A = data_in[0] ^ data_in[2] ^ data_in[4] ^ data_in[6];
23         B = data_in[1] ^ data_in[2] ^ data_in[5] ^ data_in[6];
24         C = data_in[3] ^ data_in[4] ^ data_in[5] ^ data_in[6];
25         D = data_in[0] ^ data_in[1] ^ data_in[2] ^ data_in[3] ^ data_in[4] ^ data_in[5] ^ data_in[6] ^ data_in[7];
26         //Error number check
27         err_single_bit = (D == 1'b1);
28         err_double_bit = (!err_single_bit && (A | B | C));
29         //Error correctable check
```

```
29          //Error correctable check
30          if (err_single_bit) begin // only correctable error
31              if (A && B && !C) begin // D1 error: 1 1 0
32                  c[0] = ~c[0];
33              end else if (A && !B && C) begin //D2 error : 1 0 1
34                  c[1] = ~c[1];
35              end else if (!A && B && C) begin //D3 error : 0 1 1
36                  c[2] = ~c[2];
37              end else if (A && B && C) begin //D4 error : 1 1 1
38                  c[3] = ~c[3];
39              end
40          end
41      end
42  end
43
44  assign err_uncorrectable = err_double_bit;
45  assign err_correctable = err_single_bit;
46  assign data_out = c;
47
48  endmodule
49
```

3. 결과

3.1 Before synthesis

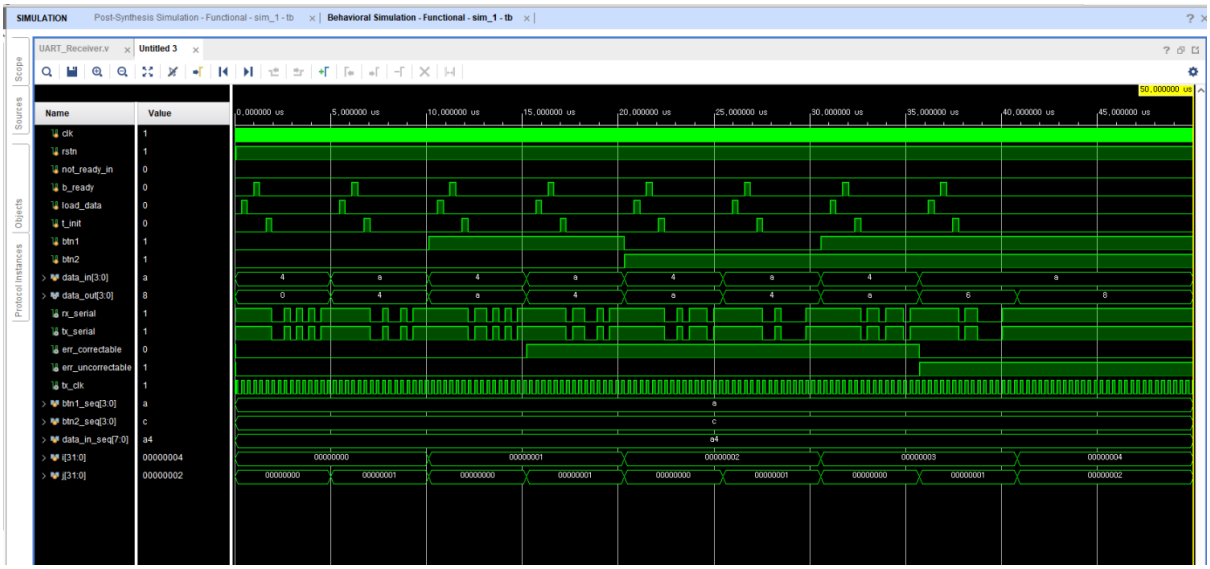

그림 3 Behavioral simulation waveform(50μs)

위 behavioral simulation은 합성 전 알고리즘 수준에서 시스템을 묘사한 것이다. 이를 통해 tb.v에서 제시된 8가지 경우의 data_in 대한 data_out 값의 출력을 알 수 있다.

0us에서 reset이 진행된 직후, data_in으로 4'h4의 값이 입력된다. Hamming encoder로 4 bit의 parity bit까지 입력되면 10101010의 8bit이 송신되게 된다. 이 과정에서 error가 발생되지 않으므로, 그대로 UART 통신이 진행되게 된다. UART_transmitter에서 data sending이 시작되게 하는 신호인 t_init이 1로 들어온 후 다음 tx_clk이 positive edge가 되는 시점부터 start_bit(0), 01010101 – 송신

된 데이터의 LSB부터 MSB 순- , stop_bit(1)이 순차적으로 tx_serial로 들어온다. Stop_bit(1)를 받은 후, 8bit가 hamming decoder를 거쳐 2 clk cycle 이후 data_out으로 4'h4가 제대로 나오게 된다.

5us에서 data_out이 나옴과 동시에 data_in으로 4'hA 값이 입력된다. 위와 마찬가지로 hamming encoder를 거치면 총 11010010의 8bit가 송신되고, error injection에서도 button이 누르지 않기 때문에 위와 마찬가지로 UART 통신이 0,0,1,0,0,1,0,1,1,1 (start_bit, LSB->MSB, stop_bit) 순으로 진행된다. 마찬가지로 hamming decoder을 거쳐 data_out으로 4'hA 값이 출력되는 것을 볼 수 있다.

10us와 15us에서는 각각 data_in으로 4'h4, 4'hA 값이 입력되어 Hamming encoder를 거친 후 나오는 8bit의 값은 각각 0us와 5us와 같지만, error injection 과정에서 btn1이 눌려 bit1에 error가 생긴다. 따라서 UART 통신을 할 때는 각각 10101011과 11010011로 통신이 진행된다. 하지만 수신부의 Hamming decoder에서 이 error을 감지해 자가 오류 수정을 거쳐 제대로 된 data값인 4'h4와 4'hA가 data_out으로 출력된다.

20us와 25us는 바로 전과 마찬가지로 진행되지만, error injection 과정에서 btn2가 눌려 bit5에 error가 생긴다. 따라서 UART 통신 시 각각 10111010과 11000010으로 진행된다. Error가 1bit에서만 일어났으므로 hamming decoder에서 이를 감지해 제대로 된 값인 4'h4와 4'hA가 출력된다.

30us와 35us에서도 전과 동일하게 진행되지만, error injection 과정에서 btn1, btn2가 모두 눌려 bit1, bit5에 error가 생긴다. 따라서 UART 통신 시 각각 10111011과 11000011으로 통신이 진행된다. 이 경우에는 error가 2bit에서 일어났으므로, 수신부의 hamming decoder에서 이를 감지할 수는 있지만, error correction을 못한다. 따라서 err_uncorrectable 신호가 1이 되지만, data_out 값은 우리가 지정한 값이 아닌 무작위 값인 4'h6과 4'h8이 나오는 것을 볼 수 있다.

3.2 Post synthesis

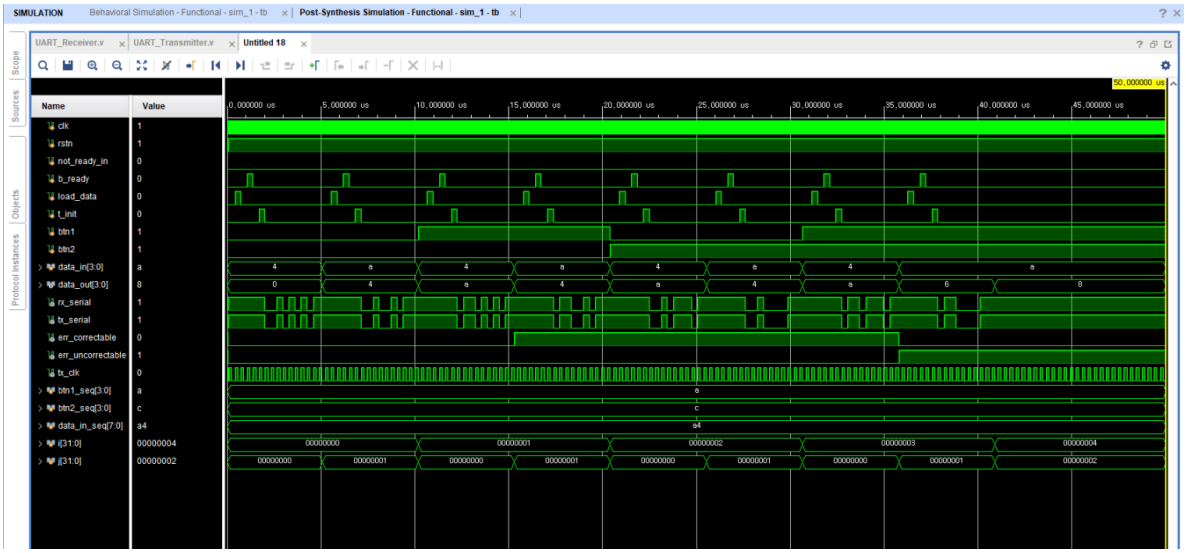

그림 4 Post-synthesis functional simulation waveform (50μs)

Post-synthesis functional simulation waveform이 behavioral simulation waveform과 동일한 것으로 보아, TRX_TOP으로 합성 후에도 모듈이 제대로 동작함을 확인할 수 있다.

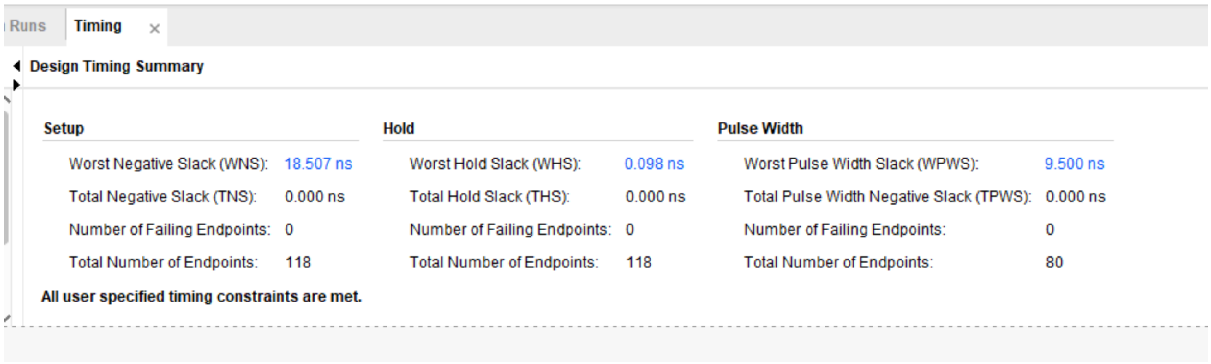

그림 5 Report timing summary

위 report timing summary를 통해 setup timing violation이 나지 않았음을 확인할 수 있다.