University of East London

1-Wire Protocol Project
Final Submission

Algorithms for Electronic Design Automation - CSE324

Wednesday, 20 August2025

Submitted to:

Dr. Haytham Azmi

Eng. Abdelrahman Sherif

Submitted by (Group 10):

- Naira Wasseem Ebraheem 23P0373
- Sara Mahmoud Ibrahim 23P0172

Table of Contents

Introduction

The 1-Wire protocol is a communication protocol that allows multiple devices to communicate over a single data line, reducing wiring complexity. It operates using master-slave architecture, where a single master device controls the communication with one or more slave devices.

The protocol is particularly well-suited for low-speed, low-power applications such as temperature sensors, RFID tags, and memory devices. Despite its simplicity, 1-Wire supports various advanced features, such as device addressing and data transfer integrity, making it ideal for distributed sensor networks and embedded systems where minimizing wiring is crucial.

In this project, we are designing a basic 1-Wire protocol setup with a single master and one slave. This will serve as a foundational step before expanding to more complex configurations.

Literature Review

The One Wire protocol, developed by Dallas Semiconductor (now Maxim Integrated), is widely used for communication with low-speed peripherals such as temperature sensors (e.g., DS18B20) using a single data line. Multiple studies have proposed various methods for implementing the One Wire master controller in Verilog or VHDL, each focusing on optimizing timing precision, resource utilization, and compatibility with FPGA platforms.

Key approaches include:

1. State Machine-Based Designs:
    - Most implementations use finite state machines (FSMs) to manage the timing-critical reset, presence detect, read, and write operations.
    - These designs emphasize strict adherence to the One Wire timing diagram, with careful clock cycle counting.

2. Clock Domain Management:
    - Some work focuses on using oversampling techniques (e.g., 1MHz or higher) to ensure precise timing, especially when the system clock does not match One Wire timing requirements.
3. Bit-Banging vs. Hardware Control:
    - Simpler designs use bit-banging techniques in software-controllable HDL modules.
    - More sophisticated designs implement full hardware control, allowing autonomous One Wire transactions with minimal CPU involvement.
4. Parameterization and Reusability:
    - Several implementations support configurable timing parameters to accommodate different One Wire devices or clock speeds, improving module reusability across platforms.
5. Testing and Verification:
    - A few papers present simulation frameworks or FPGA testbenches to validate protocol compliance using known devices like DS18B20 or DS2401.

In conclusion, while many approaches exist, the most robust designs balance timing accuracy, low logic usage, and flexibility, often leveraging FSMs and carefully designed timing controls in Verilog or VHDL.

System Design

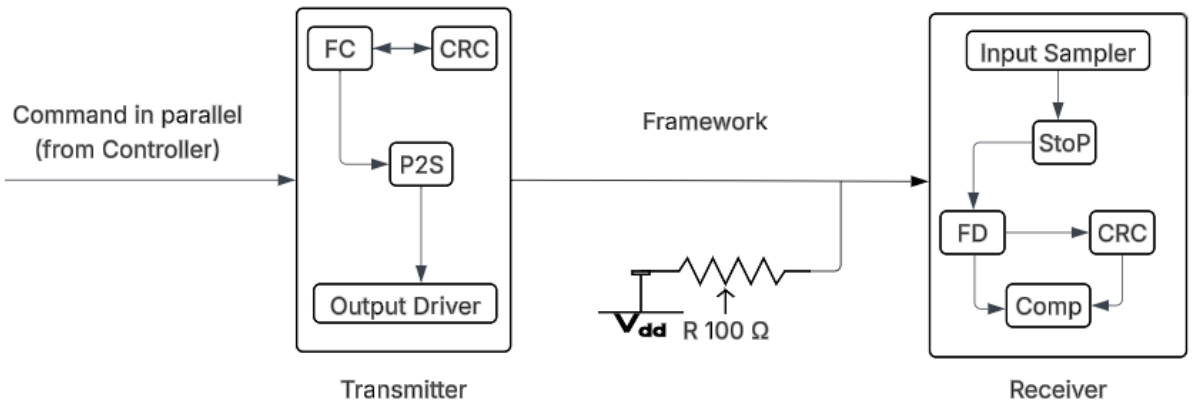


Figure 1 1-wire architecture.

Transmitter Module

Takes the command from the controller and constructs a framework and sends it to the slave, it contains the following modules:

CRC: it works as a validator (helps in checking that the framework is sent correctly) that makes calculations over the command to be sent to the slave to compare.

FC: the framework constructor concatenates the command along with the CRC bits.

P2S: it converts the parallel framework to a series one and sends it to the output driver to be sent to the slave.

Output driver: it sends the signals over the wire taking into consideration the timing of each signal (sends reset, send 0, send 1 and reads presence pulse).

Receiver Module

Takes the framework sent by the master, compares it's CRC bits with the one generated using the slave's CRC module and makes the action required in the command (according to the application). It contains the following modules:

Input Sampler: it decodes the signal into 1 or 0 bit.

S2P: it stores the decoded bit to produce a 64-bit frame.

FD: the framework destructor splits the framework received to command and CRC bits.

CRC: same as the one used in master, it recalculates the CRC from the received command.

Comparator: it compares the CRC bits sent with the calculated one.

Full System:

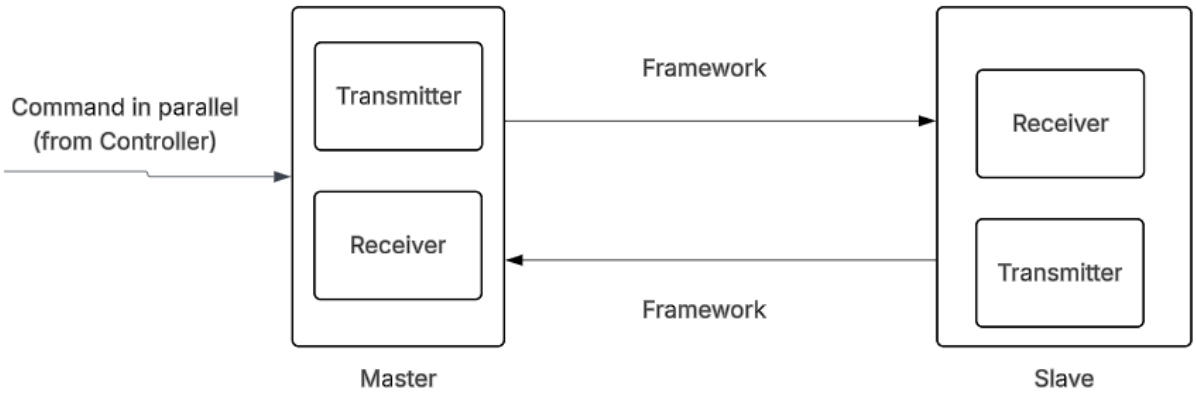


Figure 2 Full 1-wire system

Both master and slave are capable of sending and receiving frameworks according to scenario

First the controller sends the command to the master which send it to the slave according to the application (assume heat sensor) and the command (0xAA55AA55AA55AA) asks for the reading, then the slave transmitter sends it (assume it is equals to 0xffffffffffffff)

If another command is sent to the slave, it does not send any frameworks (commands)

Sending Signal specs

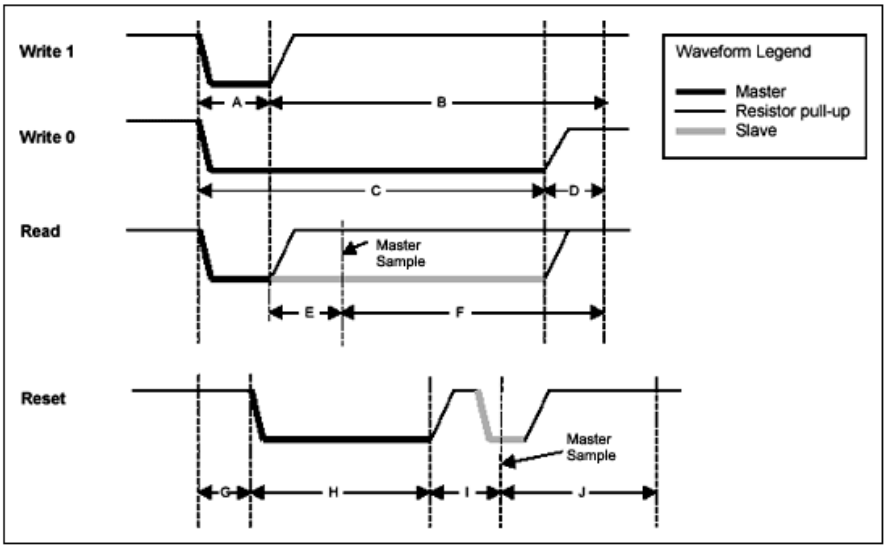


Figure 3 Timing graph.

Operation	Description	Implementation
Write 1 bit	Send a '1' bit to the 1-Wire slaves (Write 1 time slot)	Drive bus low, delay A Release bus, delay B
Write 0 bit	Send a '0' bit to the 1-Wire slaves (Write 0 time slot)	Drive bus low, delay C Release bus, delay D
Read bit	Read a bit from the 1-Wire slaves (Read time slot)	Drive bus low, delay A Release bus, delay E Sample bus to read bit from slave Delay F
Reset	Reset the 1-Wire bus slave devices and ready them for a command	Delay G Drive bus low, delay H Release bus, delay I Sample bus, 0 = device(s) present, 1 = no device present Delay J

Parameter	us (micro seconds)
A	6
B	64
C	60
D	10
E	9
F	55
G	0
H	480
I	70
J	410

Transmitter

Transmitter Verilog Code

Transmitter Top module:

Controls the data flow between submodules using FSM discussed before.

```
module onewire_tx (
    input clk,
    input reset,
    input [55:0] i_tx_data,
    input i_tx_start,
    inout bus,
    output wire o_tx_busy,      // Transmission in progress
    output wire o_tx_done      // Transmission complete
);

    // Internal signals
    wire [63:0] s_frame;
    wire [7:0] s_crc_out;
    wire s_serial_out;
    wire s_bit_strobe;
    wire s_p2s_done; // Transmition is done
    wire s_crc_ready;

    // Control signals
    reg s_send_reset; // start sending reset signal
    wire s_reset_done; // the reset signal is sent
    wire s_all_bits_sent;
    wire s_1bit_sent;
```

```
    // Module instances
    frame_constructor fc(
        .i_data(i_tx_data),
        .i_crc(s_crc_out),
        .o_frame(s_frame)
    );

    crc8 crc(
        .clk(clk),
        .reset(reset),
        .i_data(i_tx_data),
        .i_enable(i_tx_start),
        .o_crc(s_crc_out),
        .o_done(s_crc_ready)
    );

    p2s ptos(
        .clk(clk),
        .reset(reset),
        .i_parallel(s_frame),
        .i_start(s_reset_done),
        .o_serial_out(s_serial_out),
        .o_bit_strobe(s_bit_strobe),
        .o_done(s_p2s_done)
    );
```

```
    output_driver driver(
        .clk(clk),
        .reset(reset),
        .i_serial(s_serial_out),
        .i_bit_strobe(s_bit_strobe),
        .i_start(s_send_reset),
        .bus(bus),
        .o_busy(o_tx_busy),
        .o_done_64bits(s_all_bits_sent),
        .o_done_reset(s_reset_done),
        .o_done_1bit(s_1bit_sent)
    );
```

```
    // Control FSM
    parameter IDLE = 2'b00;
    parameter RESET = 2'b01;
    parameter TRANSMIT = 2'b10;
    parameter FINISH = 2'b11;

    reg [1:0] state;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            state <= IDLE;
            s_send_reset <= 1'b0;
        end else begin
            case (state)
                IDLE: begin
                    if (i_tx_start && s_crc_ready) begin
                        state <= RESET;
                        s_send_reset <= 1'b1;
                    end
                end

                RESET: begin
                    if (s_reset_done) begin
                        s_send_reset <= 1'b0;
                        state <= TRANSMIT;
                    end
                end

                TRANSMIT: begin
                    if (s_all_bits_sent) begin
                        state <= FINISH;
                    end
                end

                FINISH: begin
                    state <= IDLE;
                end
            endcase
        end
    end

    // assign    o_tx_busy = (state == IDLE || state == FINISH) ? 1'b0 : 1'b1;
    assign    o_tx_done = (state == FINISH);

endmodule
```

CRC Module:

The CRC8 module implements a cyclic redundancy check algorithm using the Dallas/Maxim 1-Wire polynomial ($x^8 + x^5 + x^4 + 1 = 0x31$) to generate an 8-bit error detection code. Operating sequentially, it processes 56 bits of input data one bit at a time over multiple clock cycles, comparing the most significant bits of the current CRC register and data shift register to determine whether to apply polynomial division. This provides robust error detection capability for 1-Wire communications.

```
6  module crc8 (
7      input clk,
8      input reset, // high async
9      input [55:0] i_data,
10     input i_enable,
11     output reg [7:0] o_crc,
12     output reg o_done
13 );
14
15     reg [7:0] s_crc_reg;
16     reg [5:0] s_bit_counter;
17     reg [55:0] s_data_shift;
18     reg s_calculating;
19
20
21     parameter CRC_POLY = 8'h31;
```

```
23     always @(posedge clk or posedge reset) begin
24         if (reset) begin
25             s_crc_reg <= 8'h00;
26             o_crc <= 8'h00;
27             o_done <= 1'b0;
28             s_bit_counter <= 6'h0;
29             s_calculating <= 1'b0;
30             s_data_shift <= 56'h0;
31         end
32         else begin
33             if (i_enable && !s_calculating) begin
34                 s_data_shift <= i_data;
35                 s_bit_counter <= 6'd56;
36                 s_calculating <= 1'b1;
37                 o_done <= 1'b0;
38                 s_crc_reg <= 8'h00;
39             end
40             else if (s_calculating) begin
41                 if (s_bit_counter > 0) begin
42
43                     if (s_crc_reg[7] ^ s_data_shift[55]) begin
44                         s_crc_reg <= (s_crc_reg << 1) ^ CRC_POLY;
45                     end
46                     else begin
47                         s_crc_reg <= s_crc_reg << 1;
48                     end
49                     s_data_shift <= s_data_shift << 1;
50                     s_bit_counter <= s_bit_counter - 1;
51                 end
52                 else begin
```

```
53                     o_crc <= s_crc_reg;
54                     o_done <= 1'b1;
55                     s_calculating <= 1'b0;
56                 end
57             end
58             else begin
59                 o_done <= 1'b0;
60             end
61         end
62     end
63
64 endmodule
```

P2S Module:

Sends a bit per clock from the 64-bit framework to the output driver to be sent starting from the LSB to the MSB and outputs a flag showing that a bit is ready.

```
module p2s (
    input clk,
    input reset,              // Active-high reset
    input [63:0] i_parallel,
    input i_start,             // Start conversion
    output reg o_serial_out,
    output reg o_bit_strobe,   // Indicates new bit is output
    output reg o_done          // All bits are out
);

    reg [6:0] s_bit_counter;
    reg [6:0] timer;
    reg [63:0] s_shift_reg;
```

```
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            s_bit_counter <= 7'b0;
            s_shift_reg <= 64'b0;
            o_serial_out <= 1'b0;
            o_bit_strobe <= 1'b0;
            o_done <= 1'b0;
            timer <= 1'b0;
        end
        else begin
            o_bit_strobe <= 1'b0;
            o_done <= 1'b0;

            if (i_start && s_bit_counter == 0) begin
                s_bit_counter <= 7'b1000001;
                timer <= 7'b1000110;
                o_serial_out <= i_parallel[0];
                s_shift_reg <= i_parallel >> 1;
                o_bit_strobe <= 1'b1;
            end
            else if (s_bit_counter > 0) begin
```

```
                if (timer == 0) begin
                    o_serial_out <= s_shift_reg[0];
                    s_shift_reg <= s_shift_reg >> 1;
                    if (s_bit_counter == 1 ) begin
                        o_done <= 1'b1;
                        s_bit_counter <= 7'b0;
                        o_bit_strobe <= 1'b0;
                    end else begin
                        o_bit_strobe <= 1'b1;
                        s_bit_counter <= s_bit_counter - 1;
                    end
                    timer <= 7'b1000110;
                end
                else timer <= timer - 1;
            end
        end
    end
endmodule
```

FC Module:

operates as a purely combinational circuit that concatenates the 56-bit data (command) with the 8-bit CRC checksum to form a complete 64-bit transmission frame.

```
module frame_constructor (
    input [55:0] i_data,
    input [7:0] i_crc,
    output [63:0] o_frame
);


    assign o_frame = {i_data, i_crc};

endmodule
```

Output driver:

transmit each bit to the slave considering the time specs mentioned before.

```
module output_driver (
    input clk,
    input reset,
    input i_serial,
    input i_bit_strobe,
    input i_start,
    inout bus,
    output reg o_busy,
    output wire o_done_64bits,
    output wire o_done_reset,
    output wire o_done_1bit
);

    parameter IDLE            = 4'b0000;
    parameter RESET_PULSE     = 4'b0001;
    parameter WAIT_PRESENCE   = 4'b0010;
    parameter SAMPLE_PRESENCE = 4'b0011;
    parameter TX_WRITE_0      = 4'b0100;
    parameter TX_WRITE_1      = 4'b0101;
    parameter TX0_RECOVERY    = 4'b0110;
    parameter TX1_RECOVERY    = 4'b0111;
    parameter DONE_ALL        = 4'b1000;
    parameter DONE_RESET      = 4'b1001;
    parameter DONE_1BIT       = 4'b1010;
```

```
    parameter RESET_LOW        = 480;
    parameter PRESENCE_WAIT    = 70;
    parameter PRESENCE_TIMEOUT = 410;
    parameter WRITE_0_LOW      = 60;
    parameter WRITE_1_LOW      = 6;
    parameter RECOVERY0_TIME   = 10;
    parameter RECOVERY1_TIME   = 64;

    reg [3:0] current_state, next_state;
    reg [15:0] counter;
    reg s_bus_out, bus_controller;

    assign bus = bus_controller ? s_bus_out : 1'bz;
```

```
    // Next state logic
    always @(*) begin
        case(current_state)
            IDLE:
                if (i_start) next_state = RESET_PULSE;
                else next_state = IDLE;

            RESET_PULSE:     next_state = (counter == 1) ? WAIT_PRESENCE : RESET_PULSE;
            WAIT_PRESENCE:   next_state = (counter == 1) ? SAMPLE_PRESENCE : WAIT_PRESENCE;
            SAMPLE_PRESENCE: begin
                if (counter == 0 && bus === 0) begin
                next_state =  DONE_RESET;
                end
                else if (counter == 0) begin
                next_state = IDLE;
                end
                else next_state = SAMPLE_PRESENCE;
            end
            TX_WRITE_0:      next_state = (counter == 1) ? TX0_RECOVERY : TX_WRITE_0;
            TX_WRITE_1:      next_state = (counter == 1) ? TX1_RECOVERY : TX_WRITE_1;
            TX0_RECOVERY:    next_state = (counter == 1) ? DONE_1BIT : TX0_RECOVERY;
            TX1_RECOVERY:    next_state = (counter == 1) ? DONE_1BIT : TX1_RECOVERY;
            DONE_RESET:      next_state = i_serial ? TX_WRITE_1 : TX_WRITE_0;
            DONE_1BIT: begin
                if (i_bit_strobe) begin
                            if (i_serial)
                                next_state = TX_WRITE_1;
                            else next_state = TX_WRITE_0;
                        end
                else next_state = DONE_ALL;
```

```
            end
            DONE_ALL: begin
                if (i_start)
                    next_state = RESET_PULSE;
                else
                    next_state = IDLE;
            end
            default:         next_state = IDLE;
        endcase
    end
```

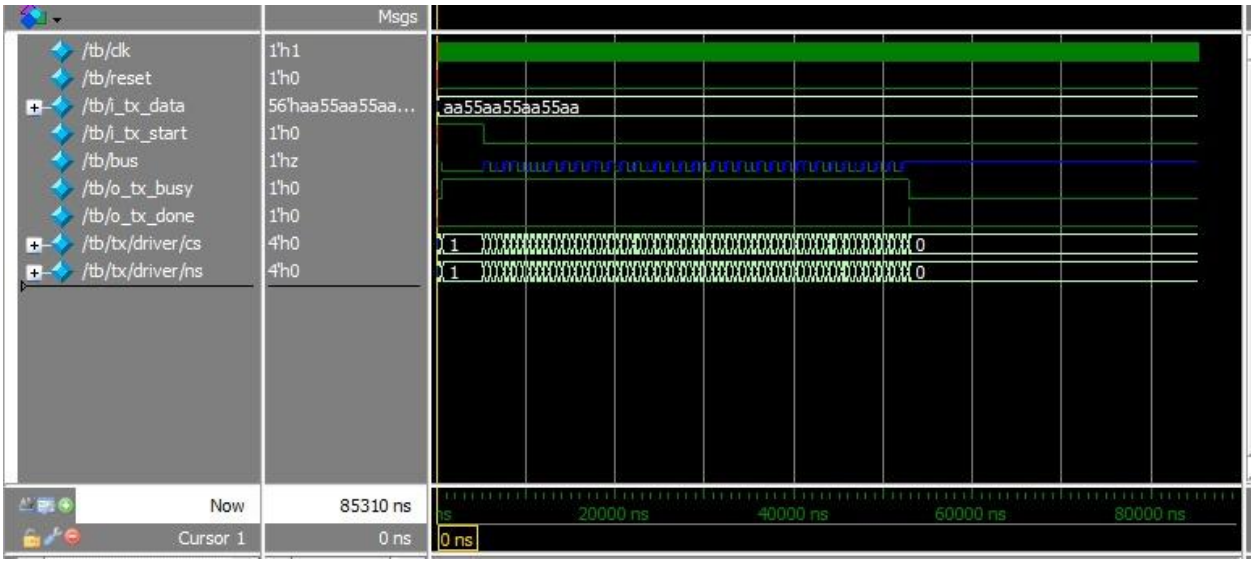
```
86      // State register, counter and output logic
87      always @(posedge clk or posedge reset) begin
88          if (reset) begin
89              current_state <= IDLE;
90              counter <= 0;
91              s_bus_out <= 1'b1;
92              bus_controller <= 1'b0;
93              o_busy <= 0;
94          end else begin
95              current_state <= next_state;
96
97              // Set counter on state entry
98              if (current_state != next_state) begin
99                  case(next_state)
100                     RESET_PULSE:     counter <= RESET_LOW;
101                     WAIT_PRESENCE:   counter <= PRESENCE_WAIT;
102                     SAMPLE_PRESENCE: counter <= PRESENCE_TIMEOUT;
103                     TX_WRITE_0:      counter <= WRITE_0_LOW;
104                     TX_WRITE_1:      counter <= WRITE_1_LOW;
105                     TX0_RECOVERY:     counter <= RECOVERY0_TIME;
106                     TX1_RECOVERY:     counter <= RECOVERY1_TIME;
107                     default:         counter <= 0;
108                 endcase
109             end else if (counter > 0) begin
110                 counter <= counter - 1;
111             end
113             // Bus control logic
114             case(next_state)
115                 RESET_PULSE, TX_WRITE_0, TX_WRITE_1: begin
116                     s_bus_out <= 1'b0;
117                     bus_controller  <= 1'b1;
118                 end
119                 default: begin
120                     s_bus_out <= 1'b1;
121                     bus_controller  <= 1'b0;
122                 end
123             endcase
124
125             // Output signals as reg
126             o_busy <= (next_state != IDLE) && (next_state != DONE_ALL);
127         end
128     end
129
130     assign o_done_reset = (current_state == DONE_RESET);
131     assign o_done_1bit = (current_state == DONE_1BIT);
132     assign o_done_64bits = (current_state == DONE_ALL);
133
134 endmodule
```

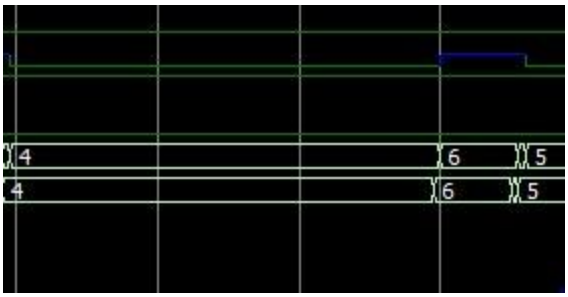
Test bench

Test case sending 1 command

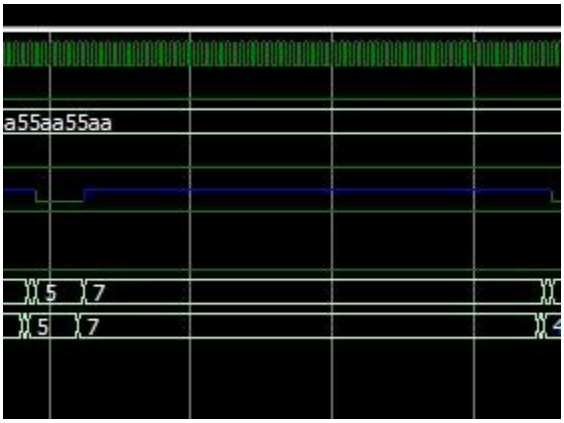
```
// Clock generation: 10ns period (100MHz)
initial begin
    clk = 1;
    forever #500 clk = ~clk;
end

initial begin
    reset = 1;
    i_tx_start = 0;
    i_tx_data = 56'hAA55AA55AA55AA;
    #10;

    reset = 0;
    i_tx_start = 1;
    #53000;
    i_tx_start = 0;
    $stop;
end
```

output of test case 1:





sending 0 bit



sending 1 bit

Time analysis using linting

Transcript

```
# Top level modules:
#
#      crc8
#
# End time: 22:14:01 on Jul 29,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Transcript

```
# Top level modules:
#
#      frame_constructor
#
# End time: 22:14:47 on Jul 29,2025, Elapsed time: 0:00:01
#
# Errors: 0, Warnings: 0
#

Visualizer>
```

Transcript

```
# Top level modules:
#
#      onewire_master_tx
#
# End time: 22:16:15 on Jul 29,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Transcript

```
# Top level modules:
#
#      output_driver
#
# End time: 22:15:32 on Jul 29,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Transcript

```
# Top level modules:
#
#      p2s
#
# End time: 22:06:15 on Jul 29,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Synthesis

Elaboration design:

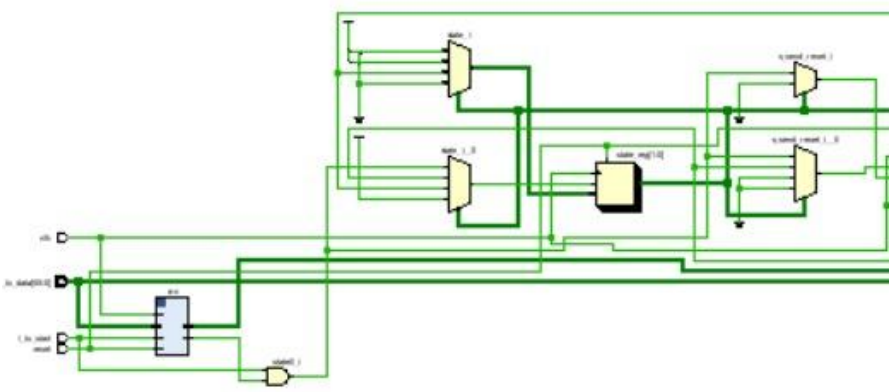


Figure 4 Elaboration design part 1

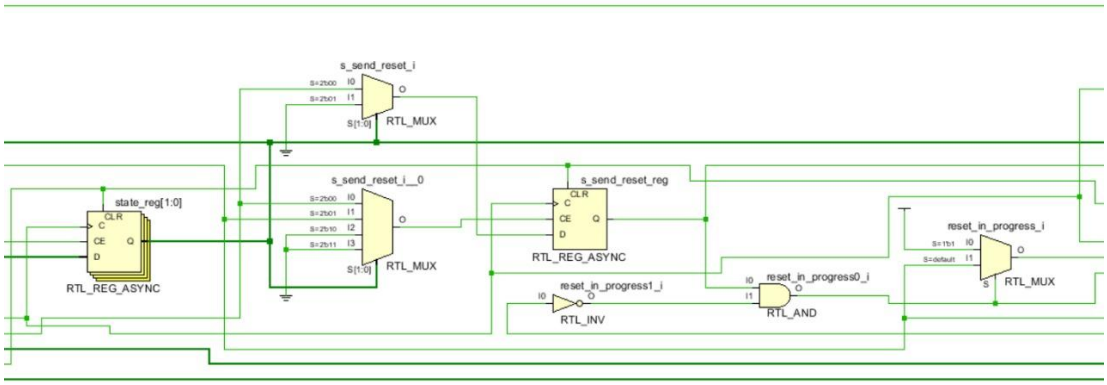


Figure 5 Elaboration design part 2

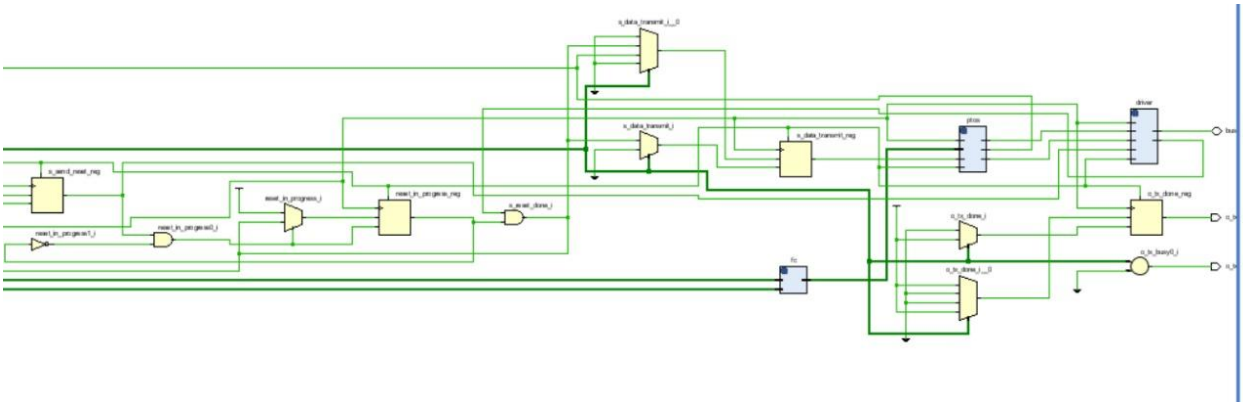
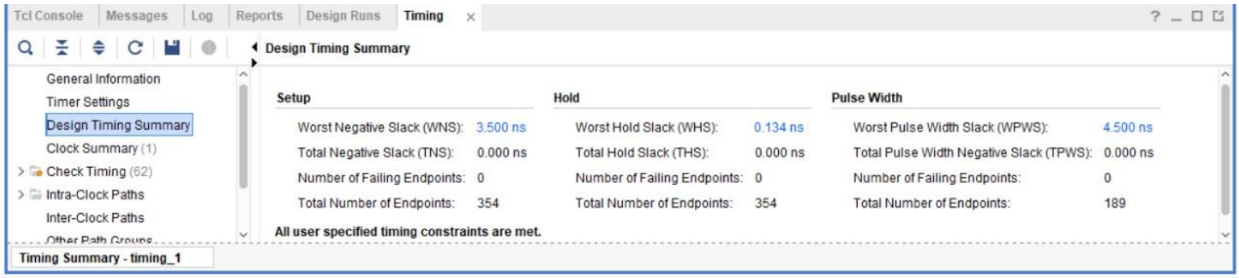
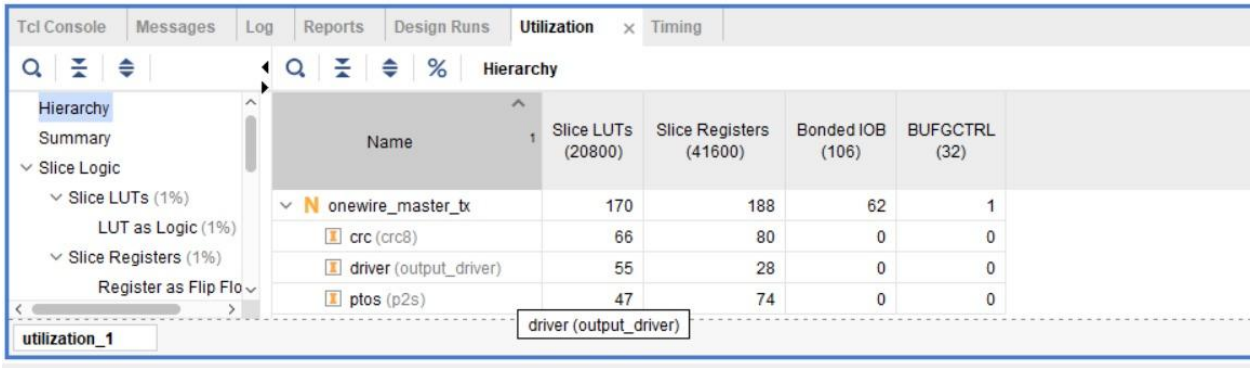


Figure 6 Elaboration design part 3

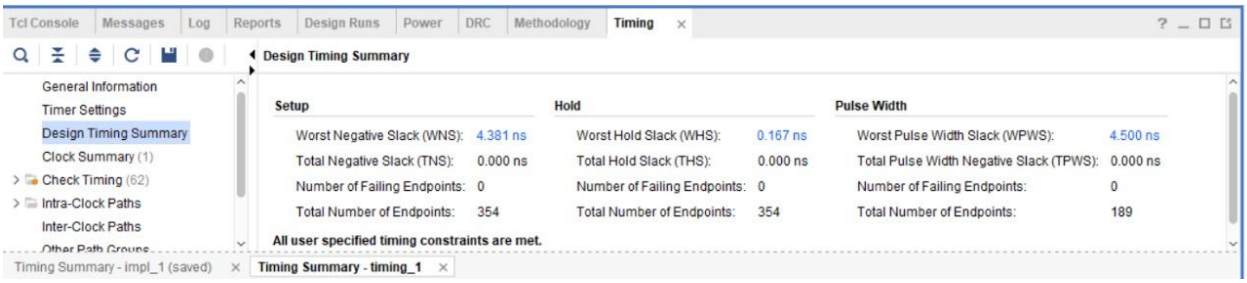
Timing summary:



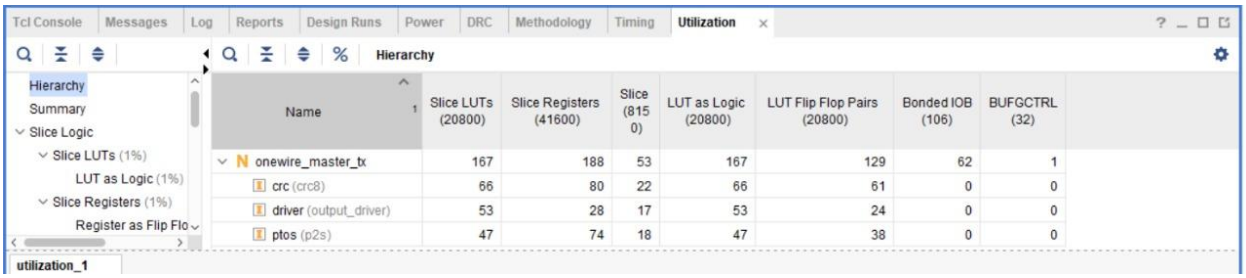
utilization report



Implementation time summary



Implementation report utilization



Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
onewire_master_tx	167	188	53	167	129	62	1
crc (crc8)	66	80	22	66	61	0	0
driver (output_driver)	53	28	17	53	24	0	0
ptos (p2s)	47	74	18	47	38	0	0

Receiver

Receiver Verilog Code

Slave Module:

Top Level Module instantiate the following modules and manage signals between them.

```
6   module onewire_rx (
7       input clk,
8       input reset,
9       inout bus,
10      output [55:0] o_command,
11      output o_error
12  );
13
14      // Input sampling
15      wire s_reset_det, s_bit_val, s_bit_ready, s_presence;
16
17      // Frame processing
18      wire [63:0] s_parallel_out;
19      wire s_frame_ready;
20
21      // Frame analysis
22      wire [55:0] s_cmd;
23      wire [7:0] s_received_crc;
24
25      // CRC checking
26      wire [7:0] s_calculated_crc;
27      wire s_crc_done;
28      wire s_crc_valid;
```

```
30      // Module instances
31      Input_Sampler sampler(
32          .clk(clk),
33          .reset(reset),
34          .bus(bus),
35          .o_reset_det(s_reset_det),
36          .o_presence_pusle(s_presence),
37          .o_bit_val(s_bit_val),
38          .o_bit_ready(s_bit_ready)
39      );
40
41      s2p deserializer(
42          .clk(clk),
43          .reset(reset),
44          .i_bit_val(s_bit_val),
45          .i_bit_ready(s_bit_ready),
46          .o_parallel(s_parallel_out),
47          .o_frame_ready(s_frame_ready)
48      );
49
50      frame_destructor destructor(
51          .i_frame(s_parallel_out),
52          .i_reset(reset),
53          .i_enable (s_frame_ready),
54          .o_command(s_cmd),
55          .o_crc(s_received_crc)
56      );
```

```
58      crc8 slave_crc (
59          .clk(clk),
60          .reset(reset),
61          .i_data(s_cmd),
62          .i_enable (s_frame_ready),
63          .o_crc(s_calculated_crc),
64          .o_done(s_crc_done)
65      );
66
67      CRC_Comparator checker(
68          .i_enable(s_crc_done),
69          .i_received_crc(s_received_crc),
70          .i_calculated_crc(s_calculated_crc),
71          .o_crc_valid(s_crc_valid)
72      );
73
74
75      // Tri-state buffer for bus line
76      assign bus = (s_presence) ? 1'b0 : 1'bz;
77      assign o_command = s_cmd;
78
79      assign o_error = s_crc_done ? ~ s_crc_valid : 0;
80
81  endmodule
```

Input Sampler:

Process each pulse and detect if it is a reset, 0 or 1 signal according to the FSM mentioned before.

```
6   module Input_Sampler (clk, reset, bus,o_reset_det, o_presence_pusle , o_bit_val, o_bit_ready);
7
8
9   input clk, reset, bus;
10  output reg o_reset_det, o_presence_pusle, o_bit_val, o_bit_ready;
11
12  //timing parameters
13  parameter RESET_LOW = 480; // 480us for reset pulse
14  parameter PRESENCE_WAIT = 70; // 70us for presence pulse
15  parameter PRESENCE_TIMEOUT = 410; // 410us for presence timeout
16  parameter SAMPLE_DELAY = 70; // 70us for sampling delay
17  parameter RECOVERY_TIME = 50; // 50us for recovery time
18
19  //FSM states
20  parameter IDLE = 4'b0000;
21  parameter WAIT_RESET = 4'b0001;
22  parameter RESET_DETECTED = 4'b0010;
23  parameter WAIT_PRESENCE = 4'b0011;
24  parameter READY_PRESENCE = 4'b0100;
25  parameter SEND_PRESENCE = 4'b0101;
26  parameter PRESENCE_DETECTED = 4'b0110;
27  parameter WAIT_SAMPLE = 4'b0111;
28  parameter SAMPLE_BIT = 4'b1000;
29  parameter DONE = 4'b1001;
```

```
31  //internal signals
32  reg [15:0] timer;  //keep track of timing durations
33  reg [6:0] s_frame_count;
34  reg [3:0] cs,ns;
35
36
37  //state memory logic
38  always @(posedge clk or posedge reset) begin
39      if (reset)
40          cs <= IDLE;
41      else
42          cs <= ns;
43  end
```

```
45  //next state logic
46  always @(*) begin
47      ns = cs;  //default to current state
48      case(cs)
49          IDLE : begin
50              if (bus == 0) begin //detect reset pulse
51                  ns = WAIT_RESET;
52              end
53          end
54
55      WAIT_RESET : begin
56      if (bus === 1'bz) begin
57          if (timer >= RESET_LOW) begin  // Changed from RESET_LOW-2
58              ns = RESET_DETECTED;
59          end else begin
60              ns = IDLE;
61          end
62      end
63      end
64
65      RESET_DETECTED : begin
66          ns = WAIT_PRESENCE; //move to wait for presence pulse
67      end
68      WAIT_PRESENCE : begin
69          if (timer >= PRESENCE_WAIT) //wait to check if a slave is present
70          ns = READY_PRESENCE;
71      end
72
73      READY_PRESENCE: begin
74          ns = SEND_PRESENCE;
75      end
```

```
77      SEND_PRESENCE: begin
78          if (timer >= PRESENCE_TIMEOUT)
79          ns = PRESENCE_DETECTED;
80      end
81
82       PRESENCE_DETECTED: begin
83              ns = WAIT_SAMPLE;
84          end
85
86      WAIT_SAMPLE: begin
87          if (timer >= SAMPLE_DELAY)
88              ns = SAMPLE_BIT;
89      end
90
91      SAMPLE_BIT: begin
92          if (s_frame_count == 7'd64) begin
93              ns = DONE;
94          end
95          else ns = WAIT_SAMPLE;
96      end
97
98      DONE: begin
99          ns = IDLE;
100     end
101
102     default: ns = IDLE;
103 endcase
104 end
```

```
106  //output logic
107  always @(posedge clk or posedge reset) begin
108      if (reset) begin
109          timer <= 1;
110          o_presence_pusle <= 0;
111          o_reset_det <= 0;
112          o_bit_val <= 0;
113          o_bit_ready <= 0;
114          s_frame_count <= 0;
115      end else begin
116          // Default outputs
117          o_presence_pusle <= 0;
118          o_bit_ready <= 0;
119          o_reset_det <= 0;
120          case (cs)
121              IDLE: timer <= 1;
122
123              WAIT_RESET: timer <= timer + 1;
124
125              RESET_DETECTED: begin
126                  timer <= 1;
127                  o_reset_det <= 1;
128              end
129
130              WAIT_PRESENCE: timer <= timer + 1;
131
132              READY_PRESENCE:  timer <= 1;
133
134              SEND_PRESENCE: begin
135                  o_presence_pusle <= 1;
136                  timer <= timer + 1;
137              end
139              PRESENCE_DETECTED: begin
140                  timer <= 1;
141              end
142
143              WAIT_SAMPLE: begin
144                  timer <= timer + 1;
145                  if (timer == RECOVERY_TIME) begin
146                      if (bus == 0) begin
147                          o_bit_val <= 0;
148                      end
149                      else o_bit_val <= 1;
150                      o_bit_ready <= 1;
151                      s_frame_count <= s_frame_count + 1;
152                  end
153
154              end
155
156              SAMPLE_BIT: begin
157                  timer <= 1;
158              end
159
160              DONE: timer <= timer + 1;
161
162              default: timer <= 1;
163          endcase
164      end
165  end
166  endmodule
```

S2P:

After input sampler detect the sent bit, S2P combines each 64 bits forming a framework.

```
6   module s2p (
7       input clk,
8       input reset,
9       input i_bit_val,
10      input i_bit_ready,
11      output reg [63:0] o_parallel,
12      output reg o_frame_ready
13  );
14      reg [6:0] s_count;
15      reg [63:0] s_temp_frame;
```

```
18      always @(posedge clk or posedge reset) begin
19          if (reset) begin
20              s_temp_frame <= 64'b0;
21              o_frame_ready <= 1'b0;
22              s_count = 0;
23          end else begin
24              if (i_bit_ready) begin
25                  s_temp_frame <= {i_bit_val, s_temp_frame[63:1]};
26                  s_count = s_count + 1;
27
28                  if (s_count == 7'b1000000 ) begin
29                      o_frame_ready <= 1'b1;
30                  end else begin
31                      o_frame_ready <= 1'b0;
32                  end
33              end else begin
34                  s_temp_frame <= s_temp_frame;
35                  s_count <= s_count;
36                  o_frame_ready <= 1'b0;
37              end
38          end
39      end
40
41      always @(s_temp_frame) begin
42          o_parallel = s_temp_frame;
43      end
44
45  endmodule
```

Frame Destructor:

Divide the framework into command and CRC bits.

```
module frame_destructor (
    input         clk,
    input         i_reset,      // async, active-high
    input         i_enable,
    input  [63:0] i_frame,
    output [55:0] o_command,
    output [7:0]  o_crc,
    output reg o_done
);
    reg [55:0] r_cmd;
    reg [7:0]  r_crc;

    always @(posedge clk or posedge i_reset) begin
        if (i_reset) begin
            r_cmd <= 56'd0;
            r_crc <= 8'd0;
            o_done <= 0;
        end else if (i_enable) begin
            r_cmd <= i_frame[63:8];
            r_crc <= i_frame[7:0];
            o_done <= 1;
        end
        // else: hold previous values (flip-flops do this naturally)
    end

    assign o_command = r_cmd;
    assign o_crc     = r_crc;
endmodule
```

Comparator:

Checks that the CRC send is equal to the CRC calculated in the receiver module.

```
7   module CRC_Comparator ( i_enable, i_received_crc, i_calculated_crc, o_crc_valid );
8       input i_enable;
9       input  [7:0] i_received_crc;
10      input  [7:0] i_calculated_crc;
11      output o_crc_valid;
12
13
14  assign o_crc_valid = (i_enable && i_received_crc == i_calculated_crc) ? 1'b1 : 1'b0;
15
16  endmodule
```

TestBench

Receving a commad of 101010… pattern

```
28   // Task for sending a single bit
29   task send_bit0;
30     begin
31       i_dq = 0;
32       #60000;
33       i_dq = 1'bz;
34       #10000;
35     end
36   endtask
37
38   task send_bit1;
39     begin
40       i_dq = 0;
41       #6000;
42       i_dq = 1'bz;
43       #64000;
44     end
45   endtask
```

```
48   initial begin
49
50     // Initial conditions
51     reset = 1;
52     i_dq = 1'bz;
53     #10000 // reset module
54     reset = 0;
55     i_dq = 0;
56     #480000; // pull down of reset pulse
57     i_dq = 1'b1; // release
58     #30000
59     i_dq = 0; // presence
60     #120000
61     i_dq = 1'bz; // wait to detect presence pulse
62     #50000;
63
64     // Test Case: Send 64-bit frame (alternating 1 and 0 pattern)
65     repeat(8) begin
66       send_bit1;  send_bit0; send_bit1;  send_bit0;
67       send_bit1;  send_bit0; send_bit1;  send_bit0;
68     end
69
70     $display("\n64-bit frame transmission complete");
71     #100000;
72     $finish;
73   end
74
75   endmodule
```

Time analysis using linting

Transcript

```
# Top level modules:
#
#      s2p
#
# End time: 22:27:33 on Aug 06,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Transcript

```
# Top level modules:
#
#      onewire_slave_rx
#
# End time: 22:27:14 on Aug 06,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Transcript

```
# Top level modules:
#
#      Input_Sampler
#
# End time: 22:26:52 on Aug 06,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Transcript

```
# Top level modules:
#
#      frame_destructor
#
# End time: 22:26:33 on Aug 06,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#

Visualizer>
```

Transcript

```
# Top level modules:
#
#      crc8
#
# End time: 22:26:13 on Aug 06,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#

Visualizer>
```

Transcript

```
# Top level modules:
#
#      CRC_Comparator
#
# End time: 22:20:31 on Aug 06,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#

Visualizer>
```

Synthesis

Elaboration design:

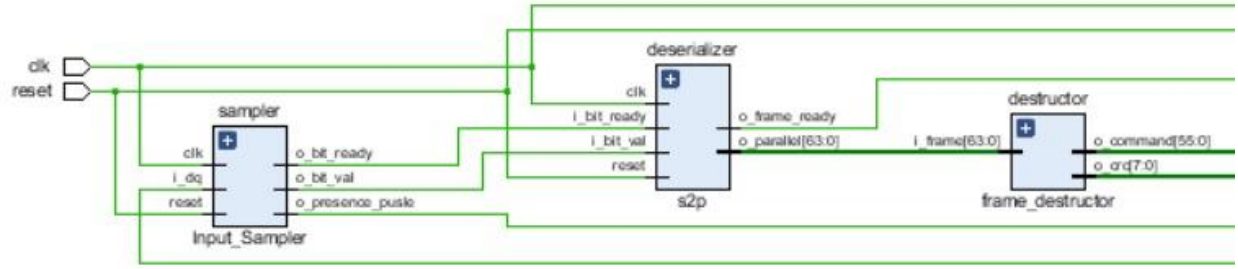


Figure 7 Elaboration design part 1

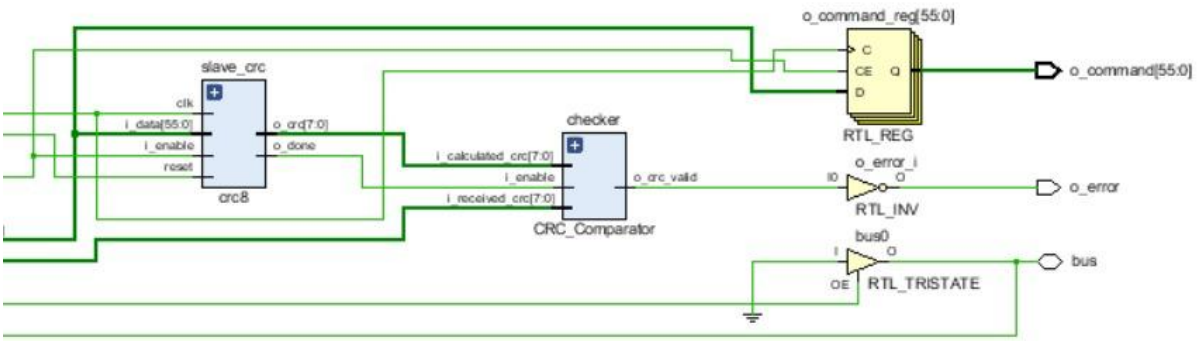
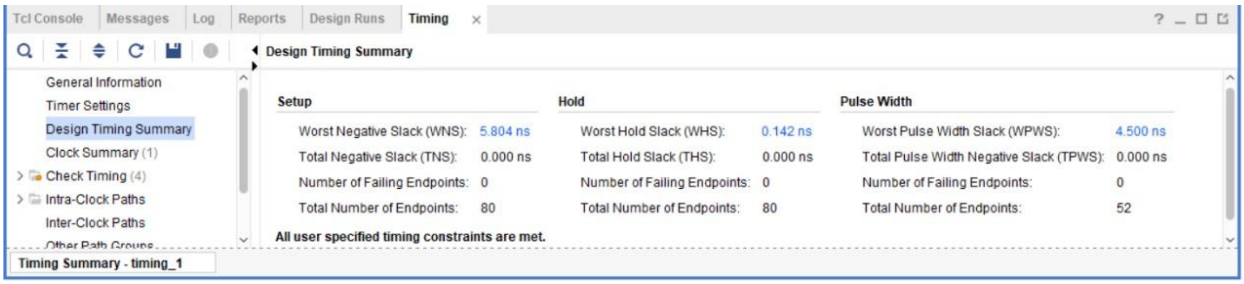


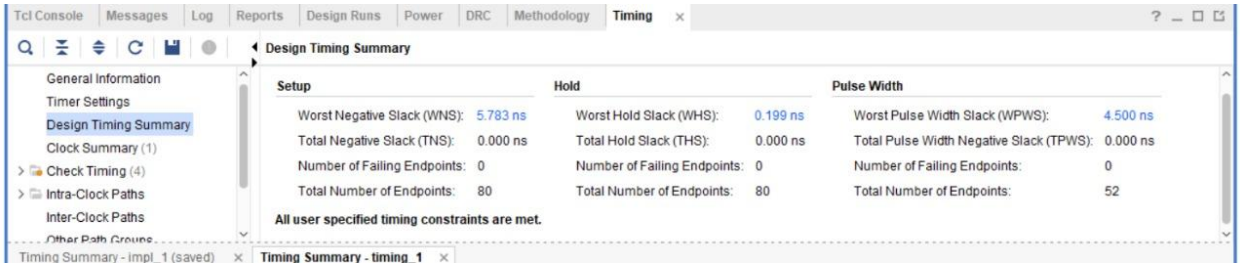
Figure 8 Elaboration design part 2

Timing summary:

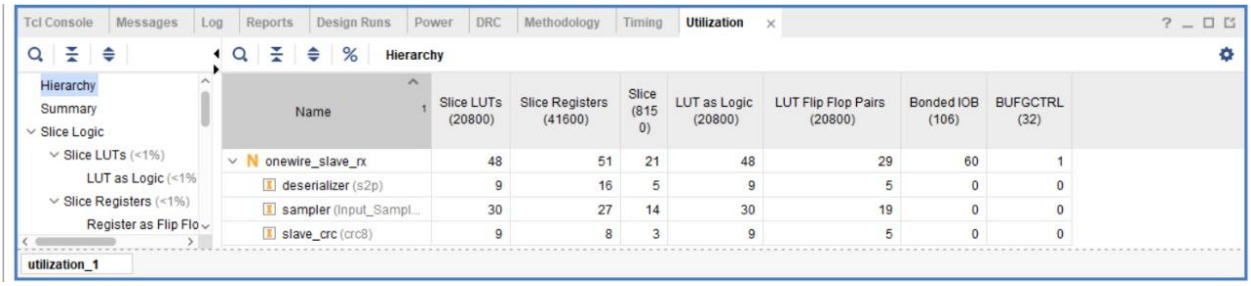


utilization report

Implementation time summary



Implementation report utilization



1 Wire protocol

One Wire Verilog Code

onewire Module:

Top Level Module instantiate the slave and master.

```
6   module onewire (
7       input clk,
8       input reset,
9       input [55:0] i_tx_data,   // from controller
10      input i_tx_start,
11      output [55:0] o_master_rx_command,
12      output o_master_rx_error,
13      output [55:0] o_slave_rx_command,
14      output o_slave_rx_error
15  );
16
17  wire bus1, bus2;
18
19  master m1 (
20      clk,
21      reset,
22      i_tx_data,
23      i_tx_start,
24      bus1,
25      bus2,
26      o_master_rx_command,
27      o_master_rx_error
28  );
29  slave  s1 (
30      clk,
31      reset,
32      bus1,
33      bus2,
34      o_slave_rx_command,
35      o_slave_rx_error
36  );
37
38  endmodule
```

Master module:

Includes both receiver and transmitter modules each has it's on bus.

```
6   module master (
7       input clk,
8       input reset,
9       input [55:0] i_tx_data,   // from controller
10      input i_tx_start,
11      inout bus1,
12      inout bus2,
13      output [55:0] o_command,
14      output o_error
15  );
16
17  wire s_track_master_busy;
18  wire s_track_master_done;
19
20  wire [55:0] s_master_data;
21  wire s_master_start;
22  wire [55:0] s_master_command;
23  wire s_master_error;
```

```
25  onewire_tx master_tx(
26      .clk(clk),
27      .reset(reset),
28      .i_tx_data(s_master_data),
29      .i_tx_start(s_master_start),
30      .bus(bus1),
31      .o_tx_busy(s_track_master_busy),
32      .o_tx_done(s_track_master_done)
33  );
34
35  onewire_rx master_rx(
36      .clk(clk),
37      .reset(reset),
38      .bus(bus2),
39      .o_command(s_master_command),
40      .o_error(s_master_error)
41  );
42
43  assign s_master_data = i_tx_data;
44  assign s_master_start = i_tx_start;
45  assign o_command = s_master_command;
46  assign o_error = s_master_error;
47
48  endmodule
```

Slave module:

Includes both receiver and transmitter modules each has it's on bus.

```
module slave (
    input clk,
    input reset,
    inout bus1,
    inout bus2,
    output [55:0] o_command,
    output o_error
);

wire s_track_slave_busy;
wire s_track_slave_done;

reg [55:0] s_slave_data;
reg s_slave_start;
wire [55:0] s_slave_command;
wire s_slave_error;

onewire_rx slave_rx(
    .clk(clk),
    .reset(reset),
    .bus(bus1),
    .o_command(s_slave_command),
    .o_error(s_slave_error)
);
```

```
onewire_tx slave_tx(
    .clk(clk),
    .reset(reset),
    .i_tx_data(s_slave_data),
    .i_tx_start(s_slave_start),
    .bus(bus2),
    .o_tx_busy(s_track_slave_busy),
    .o_tx_done(s_track_slave_done)
);


assign o_command = s_slave_command;
assign o_error = s_slave_error;

always @(*) begin
    if (s_slave_command == 56'hAA55AA55AA55AA) begin
        s_slave_data = 56'hFFFFFFFFFFFFFF;
        s_slave_start = 1;
    end
    else begin
        s_slave_data = 56'h0;
        s_slave_start = 0;
    end
end

endmodule
```

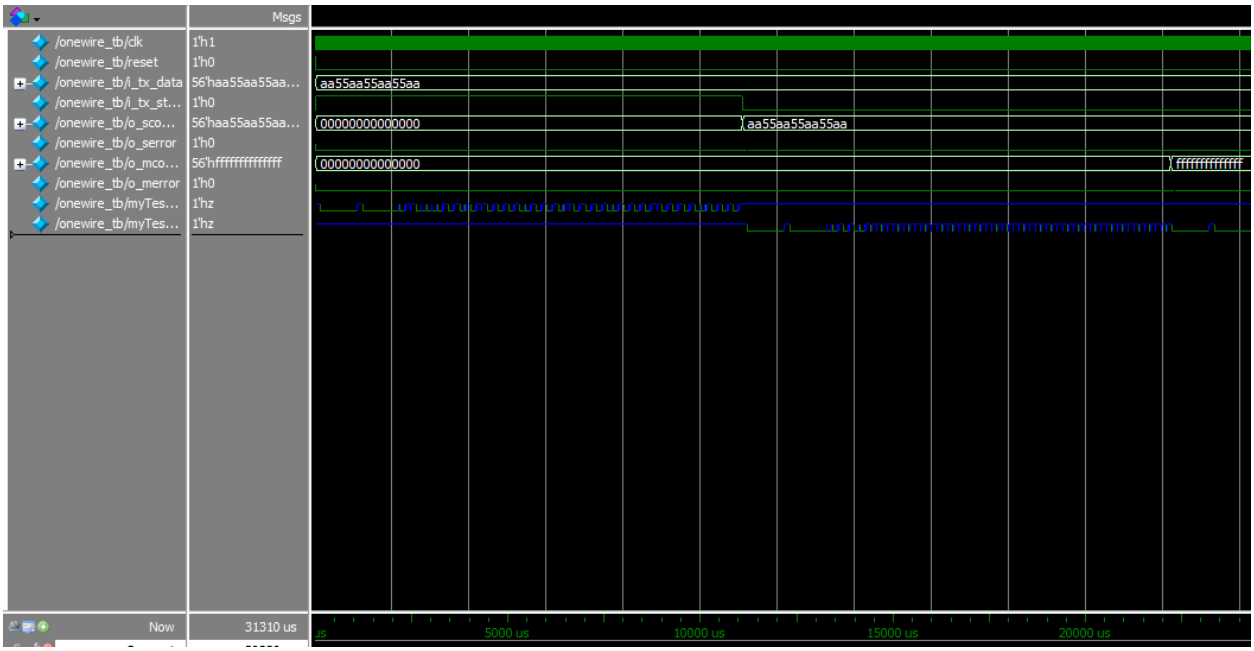
TestBench

Full scenario test case code:

```
`timescale 1us / 1us

module onewire_tb;

reg clk;
reg reset;
reg [55:0] i_tx_data;
reg i_tx_start;
wire [55:0] o_scommand;
wire o_serror;
wire [55:0] o_mcommand;
wire o_merror;

onewire myTest (
    clk,
    reset,
    i_tx_data,
    i_tx_start,
    o_mcommand,
    o_merror,
    o_scommand,
    o_serror
);


initial begin
    clk = 0;
    forever #1 clk = ~clk;
end
```

```
initial begin
    reset = 1;
    i_tx_start = 0;
    i_tx_data = 56'hAA55AA55AA55AA; // crc = 0x0C
    #10;

    reset = 0;
    i_tx_start = 1;
    #11100;
    i_tx_start = 0;
    #200

    reset = 0;
    #22900
    $stop;
end

initial begin
    $monitor("Time = %0t | o_mcommand = %h | o_merror = %h | o_scommand = %h | o_serror = %h",
     $time, o_mcommand, o_merror, o_scommand, o_serror);
end

endmodule
```

Output of testcase 1:



Outputs

```
# Time = 0 | o_mcommand = 00000000000000 | o_merror = 0 | o_mcommand = 00000000000000 | o_merror = 0
# Time = 11105 | o_mcommand = 00000000000000 | o_merror = 0 | o_mcommand = aa55aa55aa55aa | o_merror = 0

# Time = 22201 | o_mcommand = ffffffffffffff | o_merror = 0 | o_mcommand = aa55aa55aa55aa | o_merror = 0
```

slave received command = master sent command

master received command = slave build in command that was sent

error signals = 0

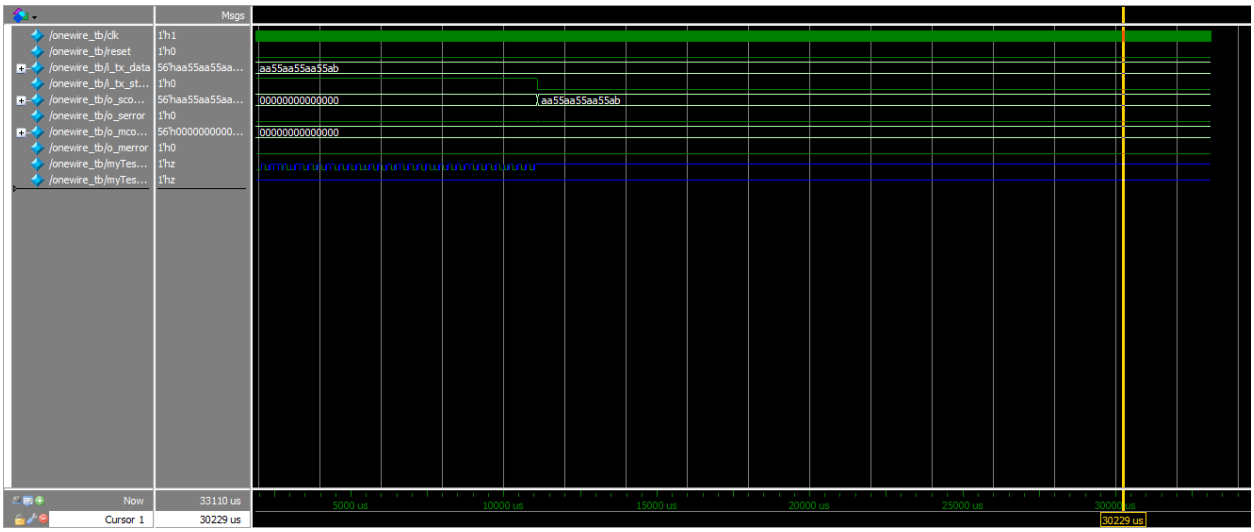
therefore the test case succeeded

Test case 2

just the first bus should work (same as pevious one but the command is different)

```
36  initial begin
37      reset = 1;
38      i_tx_start = 0;
39      i_tx_data = 56'hAA55AA55AA55AB;
40      #10;
41
42      reset = 0;
43      i_tx_start = 1;
44      #11100;
45      i_tx_start = 0;
46      #2000
47      $stop;
48  end
```

Output of Testcase 2:



Outputs

```
# Time = 0 | o_mcommand = 00000000000000 | o_merror = 0 | o_mcommand = 00000000000000 | o_merror = 0
# Time = 11105 | o_mcommand = 00000000000000 | o_merror = 0 | o_mcommand = aa55aa55aa55ab | o_merror = 0
```

slave received command = master sent command

slave does not transmitt any command

error signals = 0

therefore the test case succeeded

Synthesis

Elaboration design:

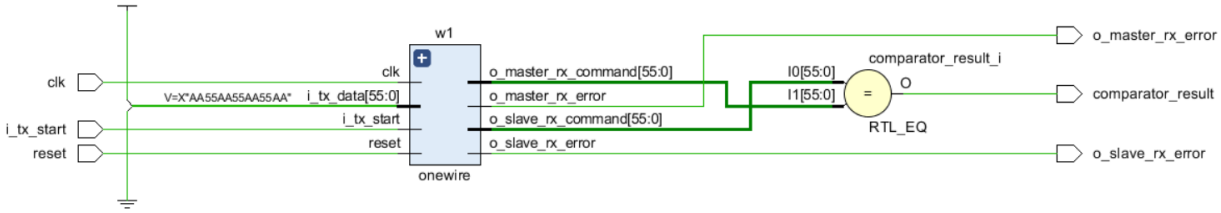


Figure 8 Elaboration design wrapper module

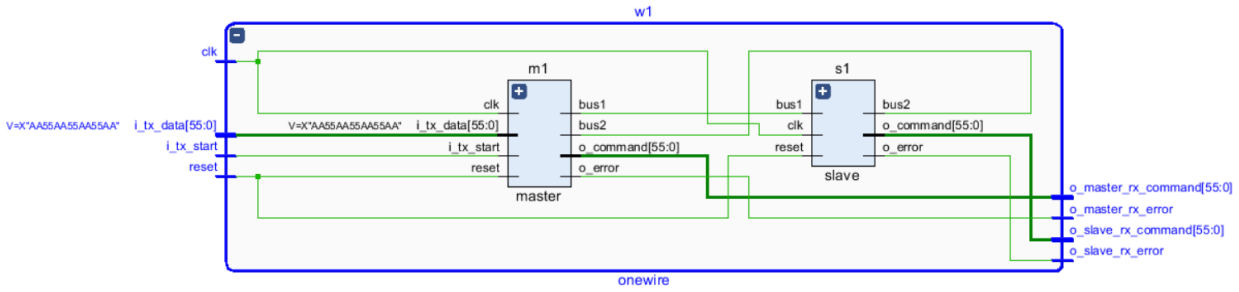


Figure 9 Elaboration design onewire module

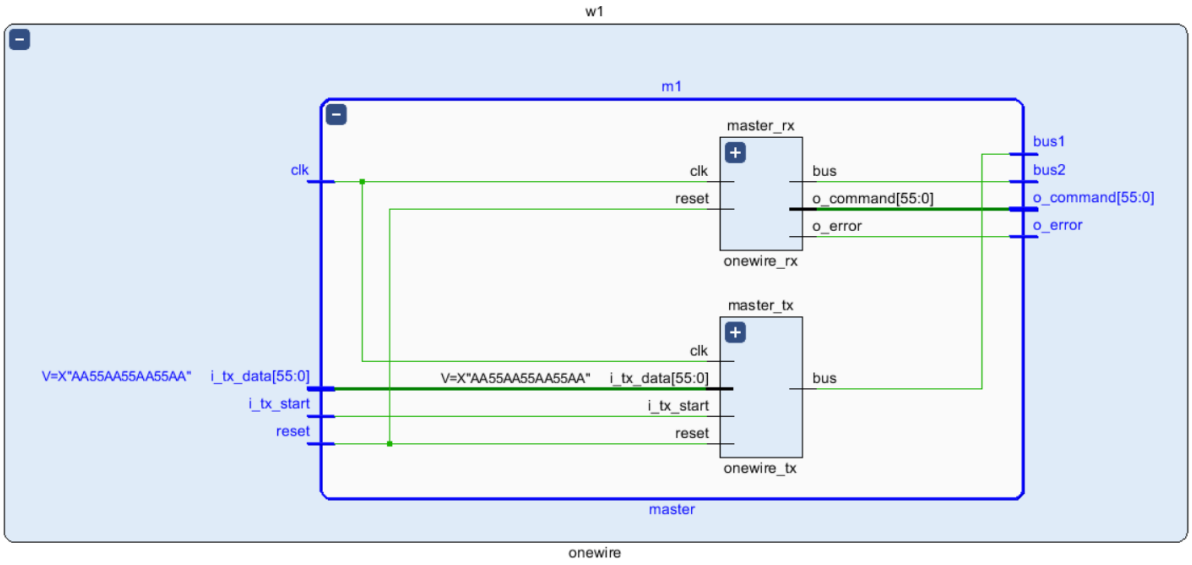


Figure 10 Elaboration design master module

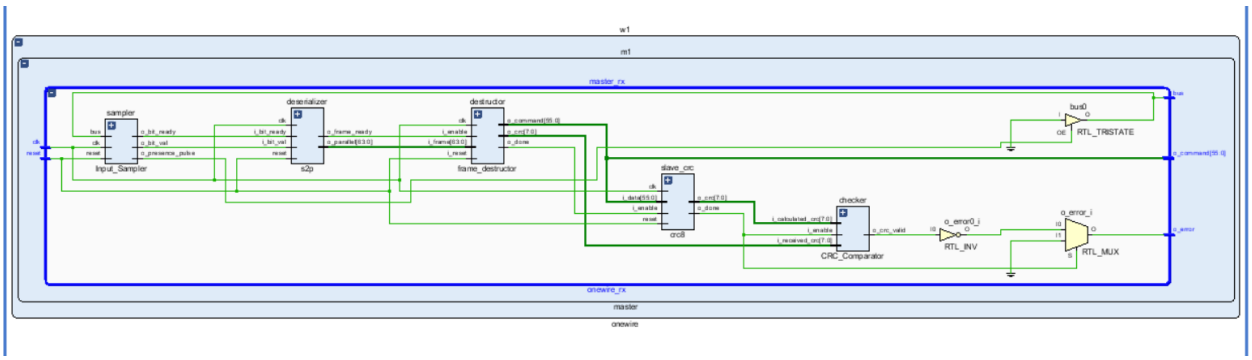


Figure 11 Elaboration design master_rx module

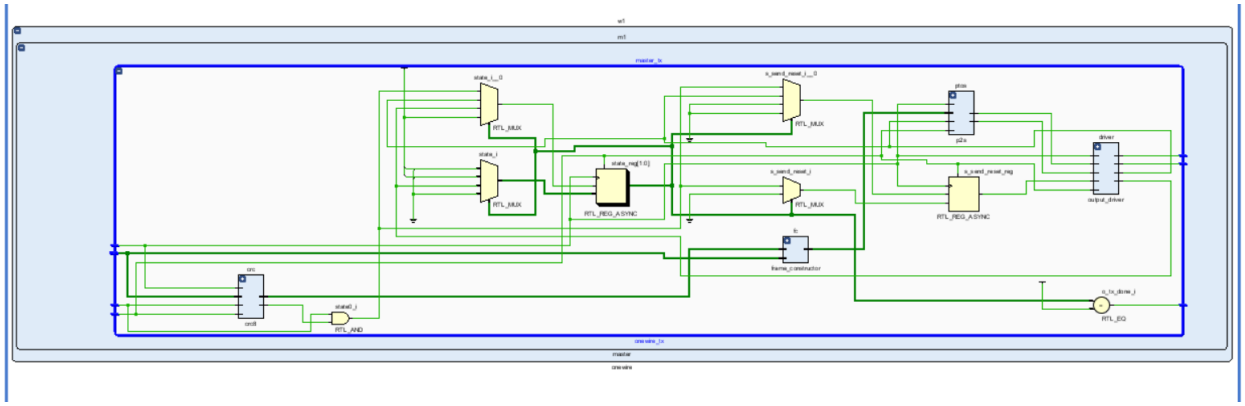


Figure12 Elaboration design part master_tx module

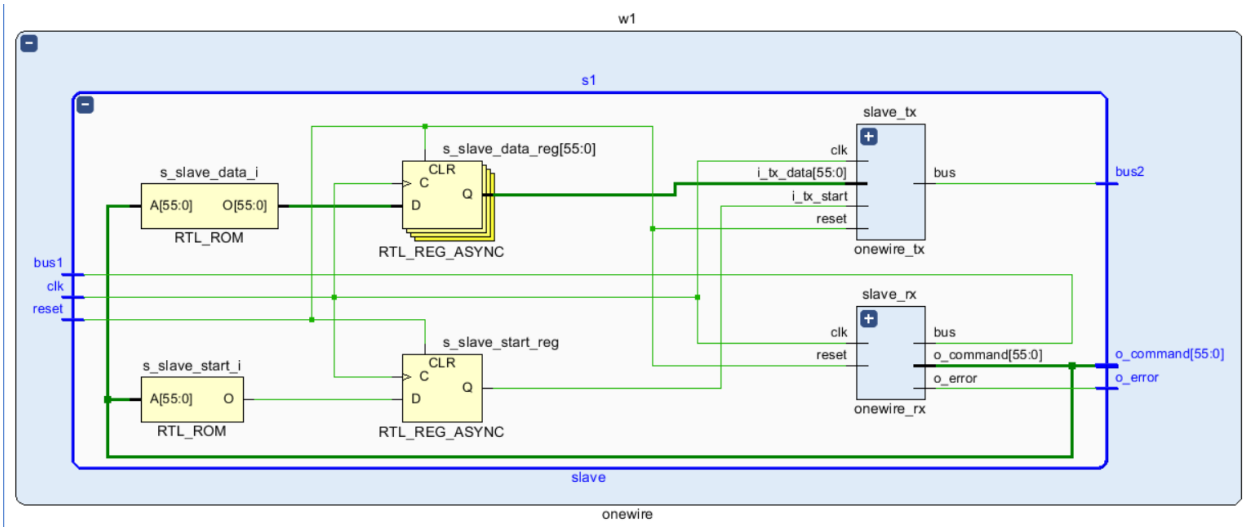


Figure 13 Elaboration design slave module

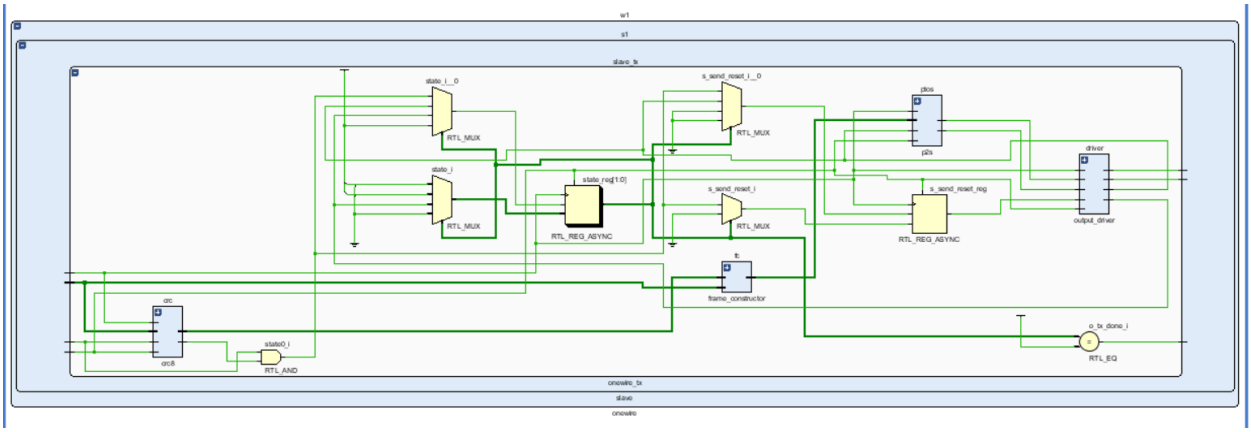


Figure 14 Elaboration design slave_rx

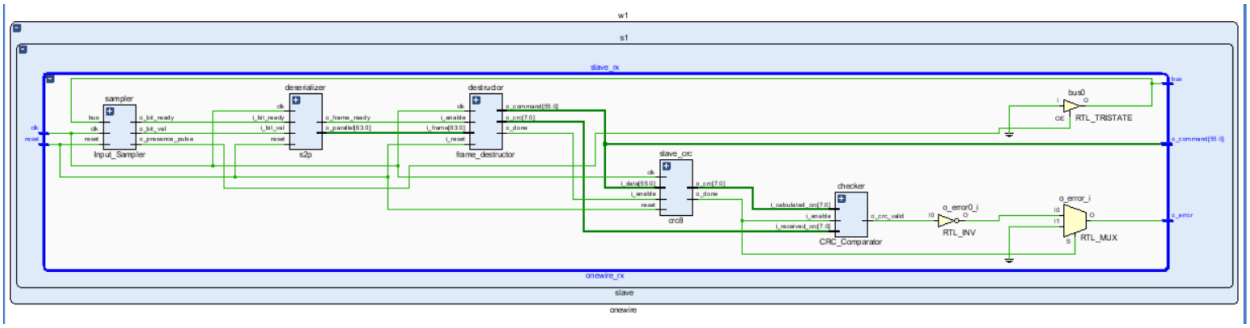


Figure 15 Elaboration design slave_tx

Synthesis timing summary:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.219 ns	Worst Hold Slack (WHS):	0.134 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1671	Total Number of Endpoints:	1671	Total Number of Endpoints:	877

All user specified timing constraints are met.

Synthesis utilization report:

Reports | Design Runs | **Utilization** | Timing | Debug

Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	Bonded IOB (106)	BUFGCTRL (32)
N wrapper	631	876	6	1
w1 (onewire)	631	876	0	0

Implementation timing summary:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.489 ns	Worst Hold Slack (WHS):	0.109 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1671	Total Number of Endpoints:	1671	Total Number of Endpoints:	877

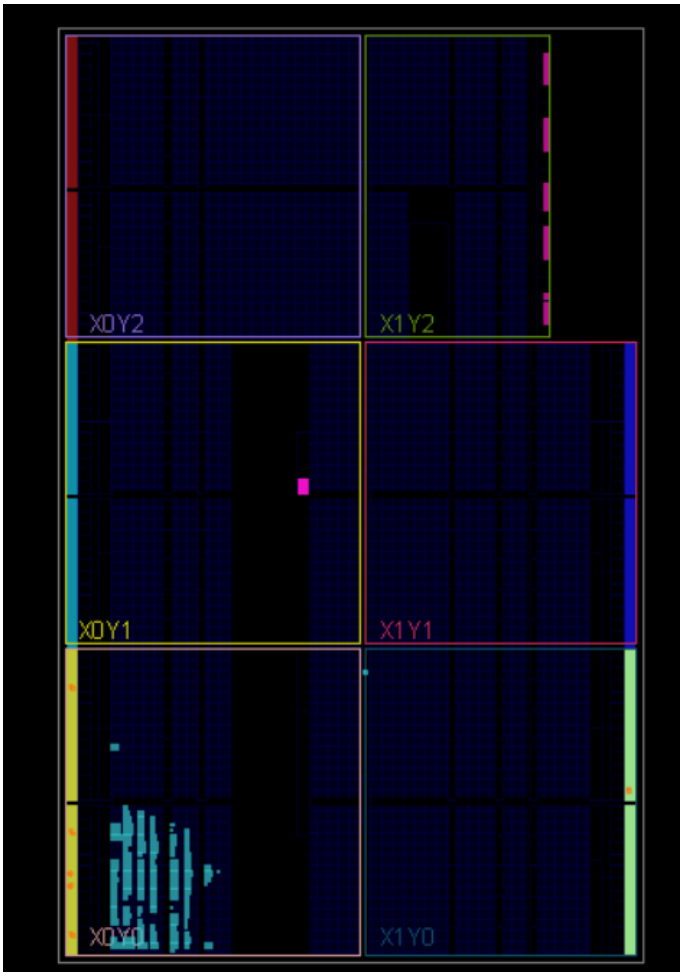
All user specified timing constraints are met.

Implementation utilization report:

Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
wrapper	627	876	235	627	461	6	1
w1 (onewire)	627	876	235	627	461	0	0

Device:



Linting

Master Compilation

Transcript

```
# Top level modules:
#
#      onewire_tx
#
# End time: 03:21:02 on Aug 21,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Lint checks

async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	p2s	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	crc8	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	onewire_tx	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	output_driver	Clock	open	unassig…	2.3.6.2

Slave Compilation

Transcript

```
# Top level modules:
#
#      onewire_rx
#
# End time: 03:15:59 on Aug 21,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Lint checks

Severity	Status	Check	Alias	Message	Module	Category	State	Owner	STARC Reference
		async_reset_active_high		Asynchronous reset is active high. Reset reset, Modul…	crc8	Clock	open	unassig…	2.3.6.2
		async_reset_active_high		Asynchronous reset is active high. Reset reset, Modul…	Input_Sampler	Clock	open	unassig…	2.3.6.2
		async_reset_active_high		Asynchronous reset is active high. Reset reset, Modul…	s2p	Clock	open	unassig…	2.3.6.2
		async_reset_active_high		Asynchronous reset is active high. Reset i_reset, Mod…	frame_destru…	Clock	open	unassig…	2.3.6.2
		inout_port_exists		Inout port exists in the design. Signal bus, Module one…	onewire_rx	Connectivity	open	unassig…	
		always_signal_assign_large		Always block has more signal assignments than the s…	crc8	Rtl Design Style	open	unassig…	2.6.1.3

Top module Compilation

Transcript

```
# Top level modules:
#
#      wrapper
#
# End time: 03:26:47 on Aug 21,2025, Elapsed time: 0:00:00
#
# Errors: 0, Warnings: 0
#
```

Lint checks

async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	p2s	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	crc8	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	onewire_tx	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	output_driver	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	Input_Sampler	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	s2p	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset i_reset, Mod…	frame_destru…	Clock	open	unassig…	2.3.6.2
async_reset_active_high	Asynchronous reset is active high. Reset reset, Modul…	slave	Clock	open	unassig…	2.3.6.2
inout_port_exists	Inout port exists in the design. Signal bus, Module one…	onewire_tx	Connectivity	open	unassig…	