Course Name: System Design using Verilog

Name: D Nelson Eugene

Roll No: KVLSI2501121

Date of Submission: 31-05-2025 (Saturday)

Q1. Design a single State Transition Diagram (STD) for a Mealy-based sequence detector that detects overlapping occurrences of the following binary sequences:

a) 01011

b) 11001

c) 1100

Write down the Verilog code for the generated STD for the given sequences.

Soln: **RTL CODE**

```
module sequence_detector (

  input clk,

  input rst,

  input in,

  output reg out

);

  parameter S0 = 4'd0,

            S1 = 4'd1,

            S2 = 4'd2,

            S3 = 4'd3,

            S4 = 4'd4,

            S5 = 4'd5,
          S6 = 4'd6,
          S7 = 4'd7,
          S8 = 4'd8;
reg [3:0] current_state, next_state;
always @(*) begin
    next_state = current_state;
    out = 0;
    case (current_state)
        S0: begin
            if (in == 1) next_state = S5;
            else         next_state = S1;
        end
        S1: begin
            if (in == 1) next_state = S2;
            else         next_state = S1;
        end
        S2: begin
            if (in == 1) next_state = S5;
            else         next_state = S3;
        end
        S3: begin
            if (in == 1) next_state = S4;
            else         next_state = S1;
        end
        S4: begin
            if (in == 1) begin
                next_state = S0;
                out = 1; // 01011 detected
            end else begin
                next_state = S3;
            end
        end
        S5: begin
            if (in = = 1) next_state = S6;
            else          next_state = S1;
        end
        S6: begin
            if (in == 1) next_state = S6;
            else          next_state = S7;
        end
        S7: begin
            if (in == 1) next_state = S8;
            else begin
                next_state = S0;
                out = 1; // 1100 detected
            end
        end
        S8: begin
            if (in == 1) next_state = S6;
            else begin
                next_state = S0;
                out = 1; // 11001 detected
            end
        end
        default: begin
            next_state = S0;
```

```
            out = 0;
        end
    endcase
  end
  // Sequential block: state register
  always @(posedge clk or posedge rst) begin
    if (rst)
      current_state <= S0;
    else
      current_state <= next_state;
  end
endmodule
```

Testbench

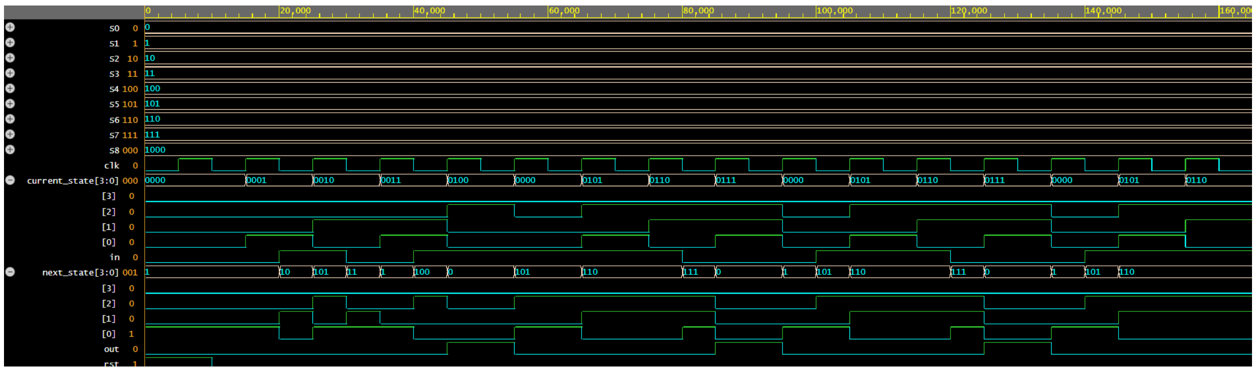
```
‘timescale 1ns/1ps
module sequence_detector_tb;
  reg clk;
  reg rst;
  reg in;
  wire out;
  sequence_detector dut (
    .clk(clk),
    .rst(rst),
    .in(in),
    .out(out)
  );
```

```
always #5 clk = ~clk;

initial begin

    $dumpfile("sequence_detector.vcd");

    $dumpvars(0, sequence_detector_tb);

    clk = 0;

    rst = 1;

    in = 0;

    #10;

    rst = 0;

    $display("Time\tin\tout");

    // Sequence: 01011

    in = 0; #10;

    $display("%0t\t%b\t%b", $time, in, out);

    in = 1; #10;

    $display("%0t\t%b\t%b", $time, in, out);

    in = 0; #10;

    $display("%0t\t%b\t%b", $time, in, out);

    in = 1; #10;

    $display("%0t\t%b\t%b", $time, in, out);

    in = 1; #10; // Should produce out = 1

    $display("%0t\t%b\t%b", $time, in, out);
```

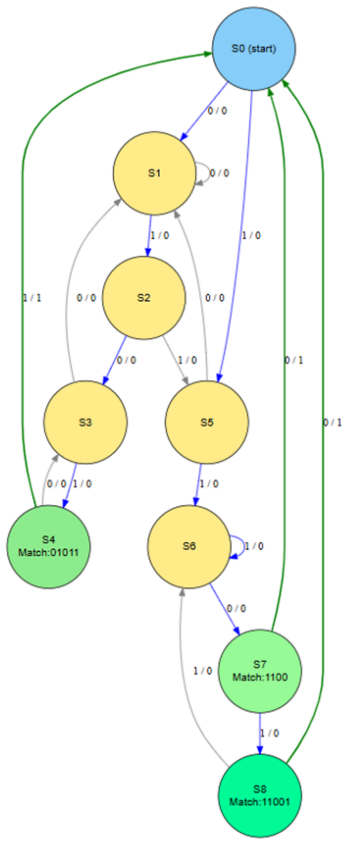
```
// Sequence: 1100

in = 1; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 1; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 0; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 0; #10; // Should produce out = 1

$display("%0t\t%b\t%b", $time, in, out);

// Sequence: 11001

in = 1; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 1; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 0; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 0; #10;

$display("%0t\t%b\t%b", $time, in, out);

in = 1; #10; // Should produce out = 1

$display("%0t\t%b\t%b", $time, in, out);

$display("Test completed");

#20;
```

```
        $finish;
    end
endmodule
```

Waveform



Final Mealy State diagram that detects all the input sequences



Q2. Design a Verilog module for a Washing Machine with the following specifications: Functional Requirements: The machine has six states: a) IDLE: Waiting for user to start b) FILL: Filling water (2 minutes) c) WASH: Washing clothes (20 minutes) d) RINSE: Clothes Rinse (10 minutes) e) DRAIN: Draining water (3 minutes) f) SPIN: Spin clothes (5 minutes) After completing all these steps, a DONE/BEEP signal should be high for 30 seconds.

Soln: RTL Code

```
module washing_machine (

    input clk,

    input rst,

    input start,

    output reg fill,

    output reg wash,

    output reg rinse,

    output reg drain,

    output reg spin,

    output reg beep

);

    // State encoding

    parameter IDLE  = 3'd0;

    parameter FILL  = 3'd1;

    parameter WASH  = 3'd2;
    parameter RINSE = 3'd3;

    parameter DRAIN = 3'd4;

    parameter SPIN  = 3'd5;

    parameter DONE  = 3'd6;

    reg [2:0] current_state, next_state;

    reg [15:0] timer; // Enough to hold time in seconds

    // State update logic

    always @(posedge clk or posedge rst) begin

        if (rst) begin

            current_state <= IDLE;

            timer <= 0;

        end else begin

            current_state <= next_state;

            if (current_state != next_state)

                timer <= 0;

            else

                timer <= timer + 1;

        end

    end
```

```
// Next state and output logic

always @(current_state or start or timer) begin

    // Default outputs

    fill  = 0;

    wash  = 0;

    rinse = 0;

    drain = 0;

    spin  = 0;

    beep  = 0;

    next_state = current_state;

    case (current_state)

        IDLE: begin

            if (start)

                next_state = FILL;

        end

        FILL: begin

            fill = 1;

            if (timer >= 120) // 2 minutes

                next_state = WASH;

        end
```

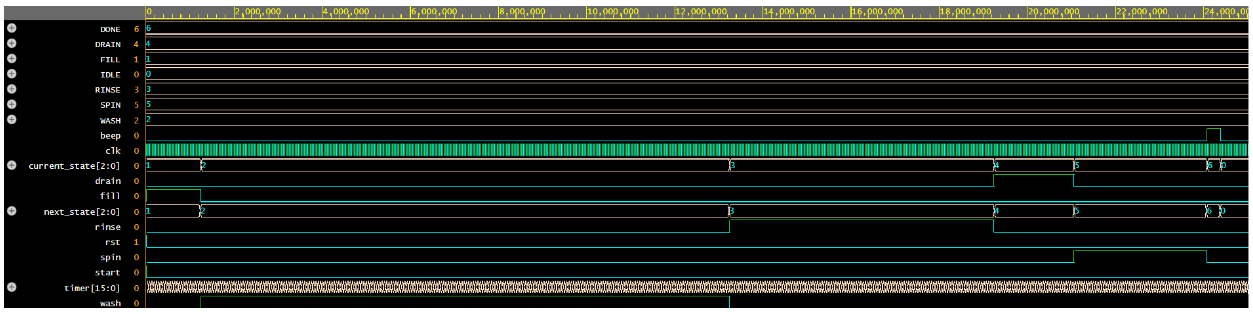
```
WASH: begin
    wash = 1;
    if (timer >= 1200) // 20 minutes
        next_state = RINSE;
end
RINSE: begin
    rinse = 1;
    if (timer >= 600) // 10 minutes
        next_state = DRAIN;
end
DRAIN: begin
    drain = 1;
    if (timer >= 180) // 3 minutes
        next_state = SPIN;
end
SPIN: begin
    spin = 1;
    if (timer >= 300) // 5 minutes
        next_state = DONE;
```

```
            end
            DONE: begin
                beep = 1;
                if (timer >= 30) // Beep for 30 seconds
                    next_state = IDLE;
            end
        endcase
    end
endmodule
```

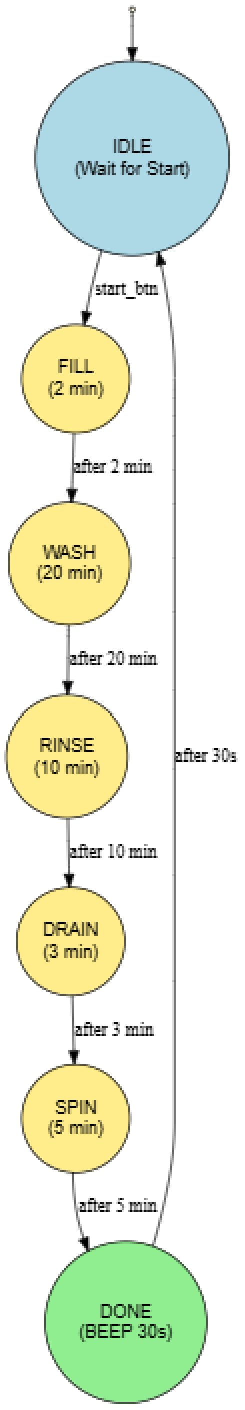
Test bench

```
`timescale 1ns / 1ps
module washing_machine_tb;
    reg clk;
    reg rst;
    reg start;
    wire fill, wash, rinse, drain, spin, beep;
    washing_machine dut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .fill(fill),

        .wash(wash),

        .rinse(rinse),

        .drain(drain),

        .spin(spin),

        .beep(beep)

    );

     // Clock generation: 1Hz clock => 1 cycle = 1 second (for simulation purpose,
use faster clock in real sim)

    always #5 clk = ~clk; // 10ns period = 100MHz clock

    initial begin

        $dumpfile("washing_machine.vcd");

        $dumpvars(0, washing_machine_tb);

        clk = 0;

        rst = 1;

        start = 0;

        // Reset pulse

        #10;

        rst = 0;

        // Wait a bit then start machine
        #10;

        start = 1;

        #10;

        start = 0; // Pulse

        // Simulate enough time to go through all states

        // Total simulated time:

        // FILL (120) + WASH (1200) + RINSE (600) + DRAIN (180) + SPIN (300) + DONE (30) = 2430 sec

        // We reduce scale to make simulation fast: 1 simulated second = 10ns

        #25000; // ~2.5k simulated seconds

        $display("Simulation finished.");

        $finish;

    end

endmodule
```

Waveform



State Diagram of its working



Q3.Design and implement the I2C Bus Protocol using Verilog on Modelsim.

RTL Code and its Tb

```
// I2C Master Module

module i2c_master (

    input wire clk,

    input wire reset,

    input wire start,

    input wire [6:0] slave_addr,

    input wire [7:0] data,

    output reg sda,

    output reg scl,

    output reg done

);

    // State encoding using parameter

    parameter IDLE    = 3'b000;

    parameter START   = 3'b001;

    parameter ADDRESS = 3'b010;

    parameter DATA    = 3'b011;

    parameter STOP    = 3'b100;
```

```
reg [2:0] current_state, next_state;

reg [3:0] bit_count; // to count up to 8 bits

// Clock divider to slow down SCL relative to clk might be needed in practice

// For simplicity, toggle scl every clk posedge for now

always @(posedge clk or posedge reset) begin

    if (reset) begin

        current_state <= IDLE;

        done <= 0;

        sda <= 1;

        scl <= 1;

        bit_count <= 0;

    end else begin

        current_state <= next_state;

    end

end

// Control sda, scl and state transitions

always @(posedge clk or posedge reset) begin

    if (reset) begin

        sda <= 1;

        scl <= 1;
        bit_count <= 0;

        done <= 0;

    end else begin

        case (current_state)

            IDLE: begin

                done <= 0;

                sda <= 1;

                scl <= 1;

                bit_count <= 0;

                if (start)

                    next_state <= START;

                else

                    next_state <= IDLE;

            end

            START: begin

                // Start condition: SDA goes low while SCL is high

                sda <= 0;

                scl <= 1;

                bit_count <= 0;

                next_state <= ADDRESS;
```

```
end
ADDRESS: begin
    scl <= ~scl; // toggle clock
    if (scl == 0) begin
        // On low edge, set SDA with address bit
        sda <= slave_addr[6 - bit_count];
    end else begin
        if (bit_count == 6) begin
            bit_count <= 0;
            next_state <= DATA;
        end else begin
            bit_count <= bit_count + 1;
            next_state <= ADDRESS;
        end
    end
end
DATA: begin
    scl <= ~scl; // toggle clock
    if (scl == 0) begin
        // On low edge, set SDA with data bit
        sda <= data[7 - bit_count];
    end else begin
        if (bit_count == 7) begin
            bit_count <= 0;
            next_state <= STOP;
        end else begin
            bit_count <= bit_count + 1;
            next_state <= DATA;
        end
    end
end
STOP: begin
    // Stop condition: SDA goes high while SCL is high
    scl <= 1;
    sda <= 0; // first maintain SDA low
    #1; // wait some time unit before releasing SDA
    sda <= 1;
    done <= 1;
    next_state <= IDLE;
end
```

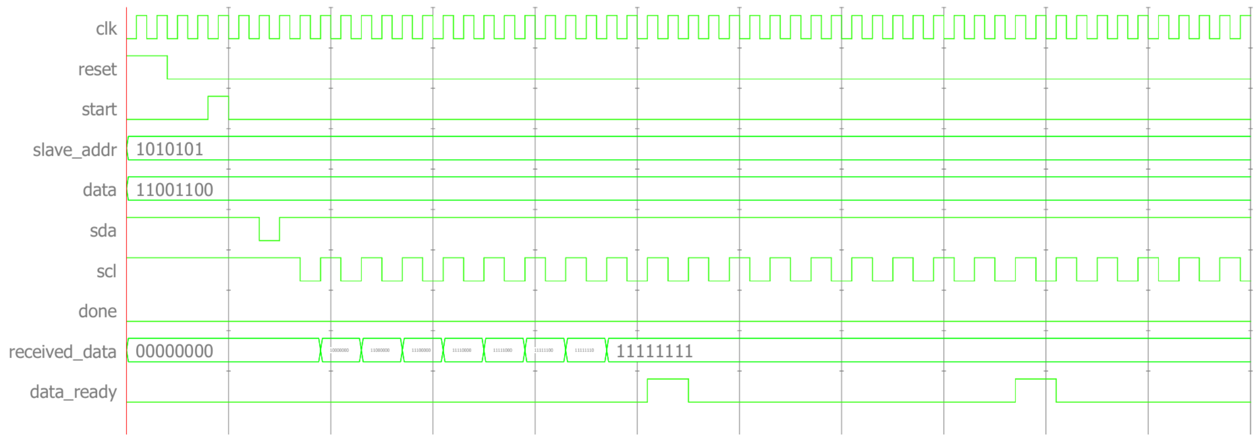
```
            default: begin
                sda <= 1;
                scl <= 1;
                done <= 0;
                next_state <= IDLE;
            end
        endcase
    end
end
endmodule
// I2C Slave Module
module i2c_slave (
    input wire clk,
    input wire reset,
    input wire sda,
    input wire scl,
    output reg [7:0] received_data,
    output reg data_ready
);
    reg [3:0] bit_count;
    always @(posedge scl or posedge reset) begin

    if (reset) begin

        received_data <= 0;

        data_ready <= 0;

        bit_count <= 0;

    end else begin

        if (bit_count < 8) begin

            received_data[7 - bit_count] <= sda; // Receive data

            bit_count <= bit_count + 1;

            data_ready <= 0;

        end else begin

            data_ready <= 1; // Data received

            bit_count <= 0;

        end

    end

    end

endmodule

// Testbench: tb_i2c

module tb_i2c;

    reg clk;
    reg reset;

    reg start;

    reg [6:0] slave_addr;

    reg [7:0] data;

    wire sda;

    wire scl;

    wire done;

    wire [7:0] received_data;

    wire data_ready;

    i2c_master master (

    .clk(clk),

    .reset(reset),

    .start(start),

    .slave_addr(slave_addr),

    .data(data),

    .sda(sda),

    .scl(scl),

    .done(done)

    );

    i2c_slave slave (
    .clk(clk),
    .reset(reset),
    .sda(sda),
    .scl(scl),
    .received_data(received_data),
    .data_ready(data_ready)
);
initial begin
    clk = 0;
    reset = 1;
    start = 0;
    slave_addr = 7'b1010101;
    data = 8'b11001100;
    #20 reset = 0;
    #20 start = 1;
    #10 start = 0;
    #500; // enough time for transfer
    $stop;
end
always #5 clk = ~clk; // 100MHz clock
```

```
    initial begin
$monitor("%t  |  sda=%b  scl=%b  done=%b  received_data=%b  data_ready=%b",
$time, sda, scl, done, received_data, data_ready);
    end
endmodule
```

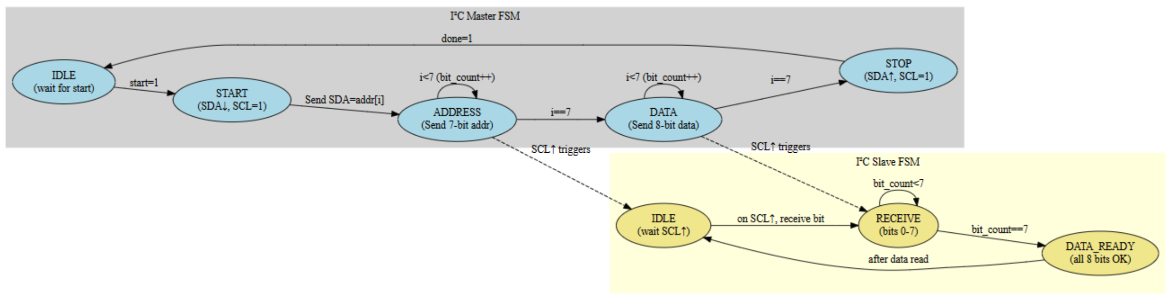
Simulation Output

```
#                    0 | sda=1 scl=1 done=0 received_data=00000000 data_ready=0
#                   65 | sda=0 scl=1 done=0 received_data=00000000 data_ready=0
#                   75 | sda=1 scl=1 done=0 received_data=00000000 data_ready=0
#                   85 | sda=1 scl=0 done=0 received_data=00000000 data_ready=0
#                   95 | sda=1 scl=1 done=0 received_data=10000000 data_ready=0
#                  105 | sda=1 scl=0 done=0 received_data=10000000 data_ready=0
#                  115 | sda=1 scl=1 done=0 received_data=11000000 data_ready=0
#                  125 | sda=1 scl=0 done=0 received_data=11000000 data_ready=0
#                  135 | sda=1 scl=1 done=0 received_data=11100000 data_ready=0
#                  145 | sda=1 scl=0 done=0 received_data=11100000 data_ready=0
#                  155 | sda=1 scl=1 done=0 received_data=11110000 data_ready=0
#                  165 | sda=1 scl=0 done=0 received_data=11110000 data_ready=0
#                  175 | sda=1 scl=1 done=0 received_data=11111000 data_ready=0
#                  185 | sda=1 scl=0 done=0 received_data=11111000 data_ready=0
#                  195 | sda=1 scl=1 done=0 received_data=11111100 data_ready=0
#                  205 | sda=1 scl=0 done=0 received_data=11111100 data_ready=0
#                  215 | sda=1 scl=1 done=0 received_data=11111110 data_ready=0
#                  225 | sda=1 scl=0 done=0 received_data=11111110 data_ready=0
#               235 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               245 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               255 | sda=1 scl=1 done=0 received_data=11111111 data_ready=1
#               265 | sda=1 scl=0 done=0 received_data=11111111 data_ready=1
#               275 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               285 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               295 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               305 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               315 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               325 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               335 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               345 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               355 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               365 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               375 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               385 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               395 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               405 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               415 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               425 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               435 | sda=1 scl=1 done=0 received_data=11111111 data_ready=1
#               445 | sda=1 scl=0 done=0 received_data=11111111 data_ready=1
#               455 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#               465 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0
#               475 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0
#           485 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0

#           495 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0

#           505 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0

#           515 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0

#           525 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0

#           535 | sda=1 scl=1 done=0 received_data=11111111 data_ready=0

#           545 | sda=1 scl=0 done=0 received_data=11111111 data_ready=0

# ** Note: $stop    : D:/Xilinx/work/i2c_bus.v(186)
```

Waveform



FSM BLOCK OF I2C BUS


I²C Master FSM
done=1
IDLE
(wait for start)
start=1
START
(SDA↓, SCL=1)
Send SDA=addr[i]
i<7 (bit_count++)
ADDRESS
(Send 7-bit addr)
i==7
i<7 (bit_count++)
DATA
(Send 8-bit data)
i==7
STOP
(SDA↑, SCL=1)
SCL↑ triggers
SCL↑ triggers
I²C Slave FSM
bit_count<7
IDLE
(wait SCL↑)
on SCL↑, receive bit
RECEIVE
(bits 0-7)
bit_count==7
DATA_READY
(all 8 bits OK)
after data read