AMERICAN UNIVERSITY OF BEIRUT

MSFEA EECE 320 DIGITAL SYSTEMS DESIGN

Verilog Project

Students:

Soheil Haroun/ 202405922

Tarek Bshinnati/ 202400161

December 12th, 2024

Introduction:

This report is about making an elevator controller for a three-story building. The controller works using a Moore finite state machine. We used Verilog to design it and ran tests to make sure it works. The elevator follows specific rules; it handles requests, moves in the right direction, and opens and closes its doors when needed. Our job was to draw out the design, write the code, and test it to see if it does everything correctly.

Current State	Condition	Next State
I0	no requests	I0
I0	R(0)	OD0
I0	dir=up & up_req	MU
I0	dir=down & down_req	MD
I0	dir=up & !up_req & down_req	MD
I0	dir=down & !down_req & up_req	MU
I1	no requests	I1
I1	R(1)	OD1
I1	dir=up & up_req	MU
I1	dir=down & down_req	MD
I1	dir=up & !up_req & down_req	MD
I1	dir=down & !down_req & up_req	MU
I2	no requests	I2
I2	R(2)	OD2
I2	dir=up & up_req	MU
I2	dir=down & down_req	MD
I2	dir=up & !up_req & down_req	MD
I2	dir=down & !down_req & up_req	MU
I3	no requests	I3
I3	R(3)	OD3
I3	dir=up & up_req	MU
I3	dir=down & down_req	MD
I3	dir=up & !up_req & down_req	MD
I3	dir=down & !down_req & up_req	MU

Current State	Condition	Next State
MU	after move up from I0	I1
MU	after move up from I1	I2
MU	after move up from I2	I3
MD	after move down from I3	I2
MD	after move down from I2	I1
MD	after move down from I1	I0
OD0	after 1 cycle	I0
OD1	after 1 cycle	I1
OD2	after 1 cycle	I2
OD3	after 1 cycle	I3

Legend:
I0, I1, I2, I3 = Idle states at floors 0,1,2,3 (doors closed)
MU = Move Up (one floor up)
MD = Move Down (one floor down)
OD0, OD1, OD2, OD3 = Open Door states at floors 0,1,2,3
R(x) = Request at floor x
dir=up/dir=down = Current direction
up_req = Requests above current floor
down_req = Requests below current floor
no requests = No pending requests

Elevator Controller Design:

This design controls an elevator of a three-story house using Moore FSM. It does the basic actions which include moving the elevator, determining its direction, and performing door operations. Requests from the floors and the cabin are encoded using a 4-bit system ensuring reasonable prioritization according to the direction (up or down) the elevator is moving to. Then, once this request has been satisfied, the system automatically resets it. Between any two floors, the elevator takes one clock cycle and pauses briefly and opens its doors at the end position. Because it has well-defined states and transitions, the FSM ensures that the operation runs smoothly, and it performs well under many conditions.

Elevator Controller Verilog Code

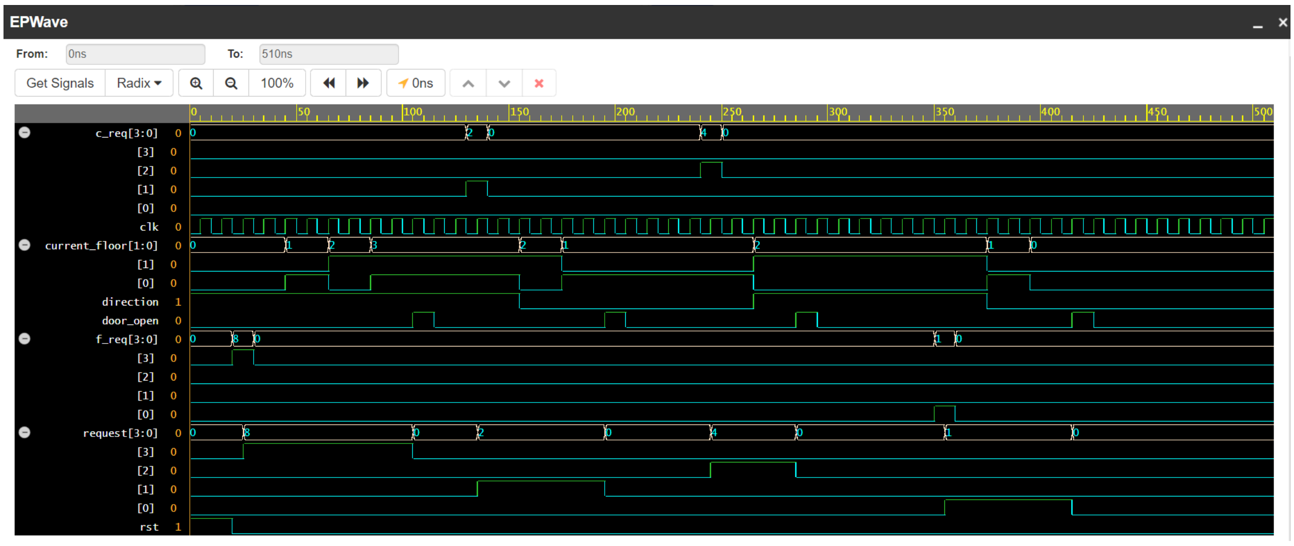
```verilog
`timescale 1ns/1ns

// Module to control elevator movement, direction, and door operations based on
      requests
module elevator_controller(
    input wire clk, // Clock signal for synchronization
    input wire rst, // Reset signal to initialize or reset the system
    input wire [3:0] f_req, // Floor requests coming from external inputs
    input wire [3:0] c_req, // Cabin requests made from inside the elevator
    output reg [3:0] request, // Combined request signal for tracking all floor
          and cabin requests
    output reg [1:0] current_floor, // Current floor where the elevator is
          located
    output reg direction, // Direction of the elevator movement (1: up, 0: down)
    output reg door_open // Door state (1: open, 0: closed)
);

// State encoding for idle, movement, and door operations
// Local parameters defining the states of the elevator
// IDLE states represent when the elevator is stationary at each floor
// MOVE_UP and MOVE_DOWN states represent the elevator's movement
// OPEN_DOOR states represent door operations for respective floors
localparam [3:0] IDLE_0 = 4'b0000,
                     IDLE_1 = 4'b0001,
                     IDLE_2 = 4'b0010,
                     IDLE_3 = 4'b0011,
                     MOVE_UP = 4'b0100,
                     MOVE_DOWN = 4'b0101,
                     OPEN_DOOR_0 = 4'b0110,
                     OPEN_DOOR_1 = 4'b0111,
                     OPEN_DOOR_2 = 4'b1000,
                     OPEN_DOOR_3 = 4'b1001;

reg [3:0] state, next_state;
reg [3:0] next_request;

// Handles reset and state transitions on clock edges
always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE_0;
        request <= 4'b0000;
        current_floor <= 2'b00;
        direction <= 1'b1;
        door_open <= 1'b0;
    end else begin
        state <= next_state;
        request <= next_request;
    end
end

// Updates elevator's current floor, door state, and direction based on state
always @(posedge clk) begin
    if (!rst) begin
        case (state)
            MOVE_UP: begin
                current_floor <= current_floor + 2'b01;
                door_open <= 1'b0;
                direction <= 1'b1;
            end

            MOVE_DOWN: begin
                current_floor <= current_floor - 2'b01;
                door_open <= 1'b0;
                direction <= 1'b0;
            end

            OPEN_DOOR_0: door_open <= 1'b1;
            OPEN_DOOR_1: door_open <= 1'b1;
            OPEN_DOOR_2: door_open <= 1'b1;
            OPEN_DOOR_3: door_open <= 1'b1;

            IDLE_0, IDLE_1, IDLE_2, IDLE_3: door_open <= 1'b0;

            default: ;
        endcase
    end
end
```

```verilog
// Combines floor and call requests into a single signal
wire [3:0] new_requests = f_req | c_req;

// Determines the next request and clears it when doors open at the requested
      floor
always @(*) begin
    next_request = request | new_requests;
    case (state)
        OPEN_DOOR_0: next_request = request & ~(4'b0001);
        OPEN_DOOR_1: next_request = request & ~(4'b0010);
        OPEN_DOOR_2: next_request = request & ~(4'b0100);
        OPEN_DOOR_3: next_request = request & ~(4'b1000);
        default: next_request = request | new_requests;
    endcase
end

// Checks if there are any pending requests
wire any_requests = |request;

// Checks if there are any requests above the current floor
wire up_req_exists = ((request[1] && (current_floor < 2'b01)) ||
                      (request[2] && (current_floor < 2'b10)) ||
                      (request[3] && (current_floor < 2'b11)));

// Checks if there are any requests below the current floor
wire down_req_exists = ((request[2] && (current_floor > 2'b10)) ||
                        (request[1] && (current_floor > 2'b01)) ||
                        (request[0] && (current_floor > 2'b00)));

// State transition logic based on current requests and elevator status
always @(*) begin
    next_state = state;
    case (state)
        IDLE_0: begin
            if (!any_requests) next_state = IDLE_0;
            else if (request[0]) next_state = OPEN_DOOR_0;
            else if (direction && up_req_exists) next_state = MOVE_UP;
            else if (!direction && down_req_exists) next_state = MOVE_DOWN;
            else if (direction && !up_req_exists && down_req_exists) next_state =
                  MOVE_DOWN;
            else if (!direction && !down_req_exists && up_req_exists) next_state
                 = MOVE_UP;
        end

        IDLE_1: begin
            if (!any_requests) next_state = IDLE_1;
            else if (request[1]) next_state = OPEN_DOOR_1;
            else if (direction && up_req_exists) next_state = MOVE_UP;
            else if (!direction && down_req_exists) next_state = MOVE_DOWN;
            else if (direction && !up_req_exists && down_req_exists) next_state =
                  MOVE_DOWN;
            else if (!direction && !down_req_exists && up_req_exists) next_state
                 = MOVE_UP;
        end

        IDLE_2: begin
            if (!any_requests) next_state = IDLE_2;
            else if (request[2]) next_state = OPEN_DOOR_2;
            else if (direction && up_req_exists) next_state = MOVE_UP;
            else if (!direction && down_req_exists) next_state = MOVE_DOWN;
            else if (direction && !up_req_exists && down_req_exists) next_state =
                  MOVE_DOWN;
            else if (!direction && !down_req_exists && up_req_exists) next_state
                 = MOVE_UP;
        end

        IDLE_3: begin
            if (!any_requests) next_state = IDLE_3;
            else if (request[3]) next_state = OPEN_DOOR_3;
            else if (direction && up_req_exists) next_state = MOVE_UP;
            else if (!direction && down_req_exists) next_state = MOVE_DOWN;
            else if (direction && !up_req_exists && down_req_exists) next_state =
                  MOVE_DOWN;
            else if (!direction && !down_req_exists && up_req_exists) next_state
                 = MOVE_UP;
        end

        MOVE_UP: begin
            case (current_floor + 2'b01)
                2'b00: next_state = IDLE_0;
                2'b01: next_state = IDLE_1;
                2'b10: next_state = IDLE_2;
                2'b11: next_state = IDLE_3;
            endcase
        end

        MOVE_DOWN: begin
            case (current_floor - 2'b01)
                2'b00: next_state = IDLE_0;
                2'b01: next_state = IDLE_1;
                2'b10: next_state = IDLE_2;
                2'b11: next_state = IDLE_3;
            endcase
        end

        OPEN_DOOR_0: next_state = IDLE_0;
        OPEN_DOOR_1: next_state = IDLE_1;
        OPEN_DOOR_2: next_state = IDLE_2;
        OPEN_DOOR_3: next_state = IDLE_3;

        default: next_state = IDLE_0;
    endcase
end

endmodule
```

Verilog Testbench Code:

```verilog
'timescale 1ns/1ns
module testbench;
    reg clk;
    reg rst;
    reg [3:0] f_req;
    reg [3:0] c_req;
    wire [3:0] request;
    wire [1:0] current_floor;
    wire direction;
    wire door_open;
    elevator_controller test_instance (
        .clk(clk),
        .rst(rst),
        .f_req(f_req),
        .c_req(c_req),
        .request(request),
        .current_floor(current_floor),
        .direction(direction),
        .door_open(door_open)          );
    initial begin
        // Generate a continuous clock signal with a 10 ns period
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0, testbench);
        // Activate reset and initialize request signals
        rst = 1;
        f_req = 4'b0000;
        c_req = 4'b0000;
        #20 rst = 0;
        // Simulate a floor request for the highest floor
        f_req = 4'b1000;
        #10 f_req = 4'b0000;
        #100;
        c_req = 4'b0010;
        #10 c_req = 4'b0000;
        #100;
        c_req = 4'b0100;
        #10 c_req = 4'b0000;
        #100;
        f_req = 4'b0001;
        #10 f_req = 4'b0000;
        #100;
        // Terminate the simulation after test cases are complete
        #50 $finish;
    end
endmodule
```

WaveForms Result:


EPWave
From: 0ns
To: 510ns
Get Signals
Radix
100%
0ns
c_req[3:0] 0
[3] 0
[2] 0
[1] 0
[0] 0
clk 0
current_floor[1:0] 0
[1] 0
[0] 0
direction 1
door_open 0
f_req[3:0] 0
[3] 0
[2] 0
[1] 0
[0] 0
request[3:0] 0
[3] 0
[2] 0
[1] 0
[0] 0
rst 1