

# Elevator Controller Verilog Code:

```

1 'timescale 1ns/1ns
2
3 // Module to control elevator movement, direction, and door operations based on
4 // requests
5 module elevator_controller(
6   input wire clk, // Clock signal for synchronization
7   input wire rst, // Reset signal to initialize or reset the system
8   input wire [3:0] f_req, // Floor requests coming from external inputs
9   input wire [3:0] c_req, // Cabin requests made from inside the elevator
10  output reg [3:0] request, // Combined request signal for tracking all floor
11    and cabin requests
12  output reg [1:0] current_floor, // Current floor where the elevator is
13    located
14  output reg direction, // Direction of the elevator movement (1: up, 0: down)
15  output reg door_open // Door state (1: open, 0: closed)
16 );
17
18 // State encoding for idle, movement, and door operations
19 // Local parameters defining the states of the elevator
20 // IDLE states represent when the elevator is stationary at each floor
21 // MOVE_UP and MOVE_DOWN states represent the elevator's movement
22 // OPEN_DOOR states represent door operations for respective floors
23 localparam [3:0] IDLE_0 = 4'b0000,
24           IDLE_1 = 4'b0001,
25           IDLE_2 = 4'b0010,
26           IDLE_3 = 4'b0011,
27           MOVE_UP = 4'b0100,
28           MOVE_DOWN = 4'b0101,
29           OPEN_DOOR_0 = 4'b0110,
30           OPEN_DOOR_1 = 4'b0111,
31           OPEN_DOOR_2 = 4'b1000,
32           OPEN_DOOR_3 = 4'b1001;
33
34 reg [3:0] state, next_state;
35 reg [3:0] next_request;
36
37 // Handles reset and state transitions on clock edges
38 always @ (posedge clk or posedge rst) begin
39   if (rst) begin
40     state <= IDLE_0;
41     request <= 4'b0000;
42     current_floor <= 2'b00;
43     direction <= 1'b1;
44     door_open <= 1'b0;
45   end else begin
46     state <= next_state;
47     request <= next_request;
48   end
49 end
50
51 // Updates elevator's current floor, door state, and direction based on state
52 always @ (posedge clk) begin
53   if (!rst) begin
54     case (state)
55       MOVE_UP: begin
56         current_floor <= current_floor + 2'b01;
57         door_open <= 1'b0;
58         direction <= 1'b1;
59       end
60       MOVE_DOWN: begin
61         current_floor <= current_floor - 2'b01;
62         door_open <= 1'b0;
63         direction <= 1'b0;
64       end
65       OPEN_DOOR_0: door_open <= 1'b1;
66       OPEN_DOOR_1: door_open <= 1'b1;
67       OPEN_DOOR_2: door_open <= 1'b1;
68       OPEN_DOOR_3: door_open <= 1'b1;
69       IDLE_0, IDLE_1, IDLE_2, IDLE_3: door_open <= 1'b0;
70     default: ;
71   endcase
72 end
73 end
74
75 // Combines floor and call requests into a single signal
76 wire [3:0] new_requests = f_req | c_req;
77
78 // Determines the next request and clears it when doors open at the requested
79 // floor
80 always @(*) begin
81   next_request = request | new_requests;
82   case (state)
83     OPEN_DOOR_0: next_request = request & ~(4'b0001);
84     OPEN_DOOR_1: next_request = request & ~(4'b0010);
85     OPEN_DOOR_2: next_request = request & ~(4'b0100);
86     OPEN_DOOR_3: next_request = request & ~(4'b1000);
87     default: next_request = request | new_requests;
88   endcase
89 end
90
91 // Checks if there are any pending requests
92 wire any_requests = !request;
93
94 // Checks if there are any requests above the current floor
95 wire up_req_exists = ((request[1] && (current_floor < 2'b01)) ||
96                       (request[2] && (current_floor < 2'b10)) ||
97                       (request[3] && (current_floor < 2'b11)));
98
99 // Checks if there are any requests below the current floor
100 wire down_req_exists = ((request[2] && (current_floor > 2'b01)) ||
101                          (request[1] && (current_floor > 2'b10)) ||
102                          (request[0] && (current_floor > 2'b00)));
103
104 // State transition logic based on current requests and elevator status
105 always @(*) begin
106   next_state = state;
107   case (state)
108     IDLE_0: begin
109       if (!any_requests) next_state = IDLE_0;
110       else if (request[0]) next_state = OPEN_DOOR_0;
111       else if (direction && up_req_exists) next_state = MOVE_UP;
112       else if (!direction && down_req_exists) next_state = MOVE_DOWN;
113       else if (direction && !up_req_exists && down_req_exists) next_state =
114         MOVE_DOWN;
115       else if (!direction && !down_req_exists && up_req_exists) next_state =
116         MOVE_UP;
117     end
118     IDLE_1: begin
119       if (!any_requests) next_state = IDLE_1;
120       else if (request[1]) next_state = OPEN_DOOR_1;
121       else if (direction && up_req_exists) next_state = MOVE_UP;
122       else if (!direction && down_req_exists) next_state = MOVE_DOWN;
123       else if (direction && !up_req_exists && down_req_exists) next_state =
124         MOVE_DOWN;
125       else if (!direction && !down_req_exists && up_req_exists) next_state =
126         MOVE_UP;
127     end
128     IDLE_2: begin
129       if (!any_requests) next_state = IDLE_2;
130       else if (request[2]) next_state = OPEN_DOOR_2;
131       else if (direction && up_req_exists) next_state = MOVE_UP;
132       else if (!direction && down_req_exists) next_state = MOVE_DOWN;
133       else if (direction && !up_req_exists && down_req_exists) next_state =
134         MOVE_DOWN;
135       else if (!direction && !down_req_exists && up_req_exists) next_state =
136         MOVE_UP;
137     end
138     IDLE_3: begin
139       if (!any_requests) next_state = IDLE_3;
140       else if (request[3]) next_state = OPEN_DOOR_3;
141       else if (direction && up_req_exists) next_state = MOVE_UP;
142       else if (!direction && down_req_exists) next_state = MOVE_DOWN;
143       else if (direction && !up_req_exists && down_req_exists) next_state =
144         MOVE_DOWN;
145       else if (!direction && !down_req_exists && up_req_exists) next_state =
146         MOVE_UP;
147     end
148     MOVE_UP: begin
149       case (current_floor + 2'b01)
150         2'b00: next_state = IDLE_0;
151         2'b01: next_state = IDLE_1;
152         2'b10: next_state = IDLE_2;
153         2'b11: next_state = IDLE_3;
154       endcase
155     end
156     MOVE_DOWN: begin
157       case (current_floor - 2'b01)
158         2'b00: next_state = IDLE_0;
159         2'b01: next_state = IDLE_1;
160         2'b10: next_state = IDLE_2;
161         2'b11: next_state = IDLE_3;
162       endcase
163     end
164     OPEN_DOOR_0: next_state = IDLE_0;
165     OPEN_DOOR_1: next_state = IDLE_1;
166     OPEN_DOOR_2: next_state = IDLE_2;
167     OPEN_DOOR_3: next_state = IDLE_3;
168
169     default: next_state = IDLE_0;
170   endcase
171 end
172
173 endmodule

```

## Verilog Testbench Code:

```
1  `timescale ins/ins
2  module testbench;
3    reg clk;
4    reg rst;
5    reg [3:0] f_req;
6    reg [3:0] c_req;
7    wire [3:0] request;
8    wire [1:0] current_floor;
9    wire direction;
10   wire door_open;
11   elevator_controller test_instance (
12     .clk(clk),
13     .rst(rst),
14     .f_req(f_req),
15     .c_req(c_req),
16     .request(request),
17     .current_floor(current_floor),
18     .direction(direction),
19     .door_open(door_open)
20   );
21   initial begin
22     // Generate a continuous clock signal with a 10 ns period
23     clk = 0;
24     forever #5 clk = ~clk;
25   end
26   initial begin
27     $dumpfile("dump.vcd");
28     $dumpvars(0, testbench);
29     // Activate reset and initialize request signals
30     rst = 1;
31     f_req = 4'b0000;
32     c_req = 4'b0000;
33     #20 rst = 0;
34     // Simulate a floor request for the highest floor
35     f_req = 4'b1000;
36     #10 f_req = 4'b0000;
37     #100;
38     c_req = 4'b0010;
39     #10 c_req = 4'b0000;
40     #100;
41     c_req = 4'b0100;
42     #10 c_req = 4'b0000;
43     #100;
44     f_req = 4'b0001;
45     #10 f_req = 4'b0000;
46     #100;
47     // Terminate the simulation after test cases are complete
48     #50 $finish;
49   end
50 endmodule
```

## WaveForms Result:

