
**CS 223 Lab Assignment IV**
**Preliminary Work**
**Sıla Bozkurt**
**22401775**
**Section: 3**
**Instructor: Sinem Sav**
**December 7, 2025**

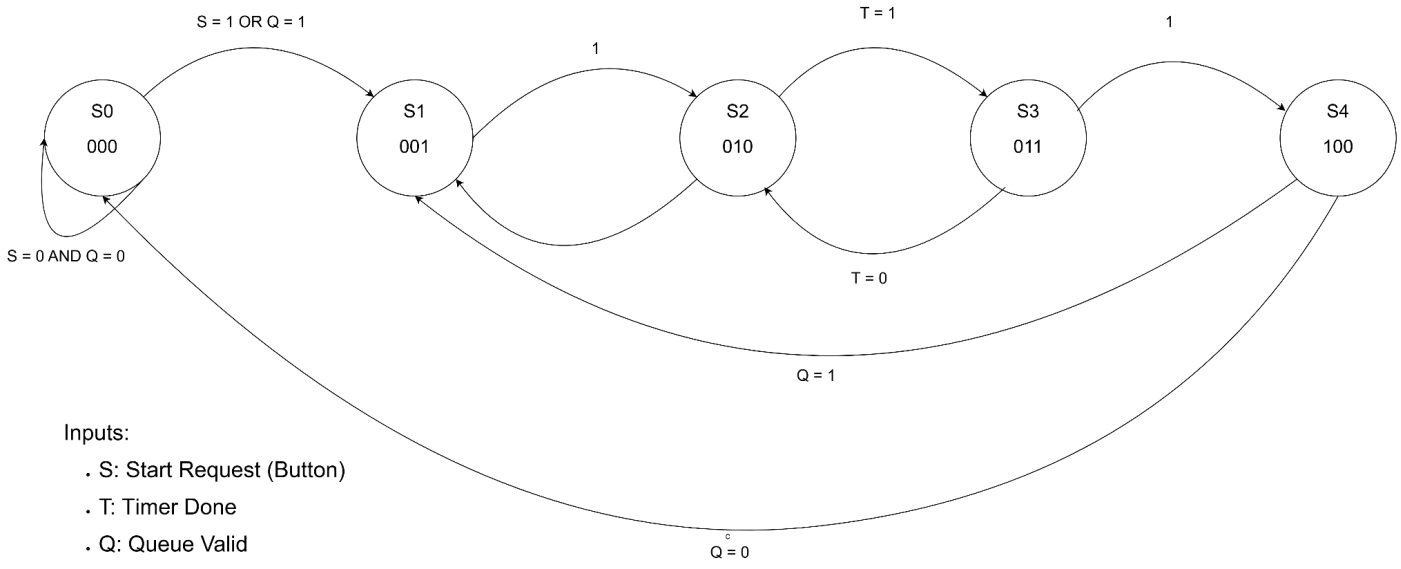
# Smart Parking Management System

**Indicate which type of FSM (Moore or Mealy) you will implement and justify your choice. Note: Consider factors such as timing behavior, output stability, and implementation complexity when making your decision.**

For the Smart Parking Management System, a Moore FSM is the optimal choice primarily due to its output stability. Since Moore outputs depend solely on the current state, they change synchronously with the clock, effectively preventing combinational glitches that can occur in Mealy machines when inputs fluctuate. This ensures that the LED indicators and timer signals remain stable and glitch-free, which is critical for meeting the visual display requirements of the lab.

Furthermore, the Moore architecture simplifies the implementation complexity for this design. The system's logic naturally divides into distinct operational modes (such as BUSY counting down or CHECK_QUEUE processing arrivals), and mapping outputs directly to these states is more intuitive than making them dependent on transient input signals. This approach also aligns perfectly with the requirement to process vehicle admissions sequentially, allowing the controller to stabilize decisions within a clock cycle.

**Provide the state transition diagram, state encoding, state transition and output tables, next-state and output equations, and the FSM schematic.**




Inputs:

- S: Start Request (Button)
- T: Timer Done
- Q: Queue Valid

Outputs (Inside Circle):

- Bit 2 (LED): 0=Solid, 1=Blink
- Bit 1 (Timer): 0=Off, 1=Run
- Bit 0 (Pop): 0=Hold, 1=Pop

| Current State | Current State Encoding | Inputs | Next State | Next State Encoding | Outputs (led, timer, pop) |
|---|---|---|---|---|---|
| S0 | 000 | $req\_entry = 1$ OR $queue\_valid = 1$ | S1 | 001 | 0, 0, 0 |
| S0 | 000 | $req\_entry = 0$ AND $queue\_valid = 0$ | S0 | 000 | 0, 0, 0 |
| S1 | 001 | 1 | S2 | 010 | 1, 1, 0 |
| S2 | 010 | $timer\_done = 1$ | S3 | 011 | 1, 1, 0 |
| S2 | 010 | $timer\_done = 0$ | S2 | 010 | 1, 1, 0 |
| S3 | 011 | 1 | S4 | 100 | 1, 0, 0 |
| S4 | 100 | $queue\_valid = 1$ | S1 | 001 | 1, 0, 1 |
| S4 | 100 | $queue\_valid = 0$ | S0 | 000 | 0, 0, 0 |

Inputs & Outputs

- Inputs:
  - $req\_entry$: Button press or vehicle arrival signal.
  - $timer\_done$: Signal from the timer indicating service time is over.
  - $queue\_valid$: Signal indicating the waiting queue is not empty.
- Outputs:
  - $led\_control$: Controls LED pattern (0=Solid/Free, 1=Blink/Occupied).
  - $timer\_en$: Enables/Resets the countdown timer.
  - $queue\_pop$: Signals the queue to release the next vehicle.

Binary encoding is used for the state variables:

- S0 (IDLE): 000
- S1 (START_TIMER): 001
- S2 (BUSY): 010
- S3 (DONE): 011
- S4 (CHECK_Q): 100

Next State Equations:

$$D_0 = (\overline{Q_2 Q_1 Q_0}.\ (req\_entry + queue\_valid)) + (Q_2 \overline{Q_1 Q_0}.\ queue\_valid)$$

$$D_1 = (\overline{Q_2 Q_1} Q_0) + (\overline{Q_2} Q_1 \overline{Q_0})$$

$$D_2 = \overline{Q_2} Q_1 Q_0$$

Output Equations:

Inputs: State Bits $Q_2$, $Q_1$, $Q_0$

$$Out_{led} = Q_2 + Q_1 + Q_0$$
$$Out_{timer} = \overline{Q_2}.\ (Q_1 \oplus Q_0)$$
$$Out_{pop} = Q_2$$

**Determine the minimum number of flip-flops required by your encoding.**

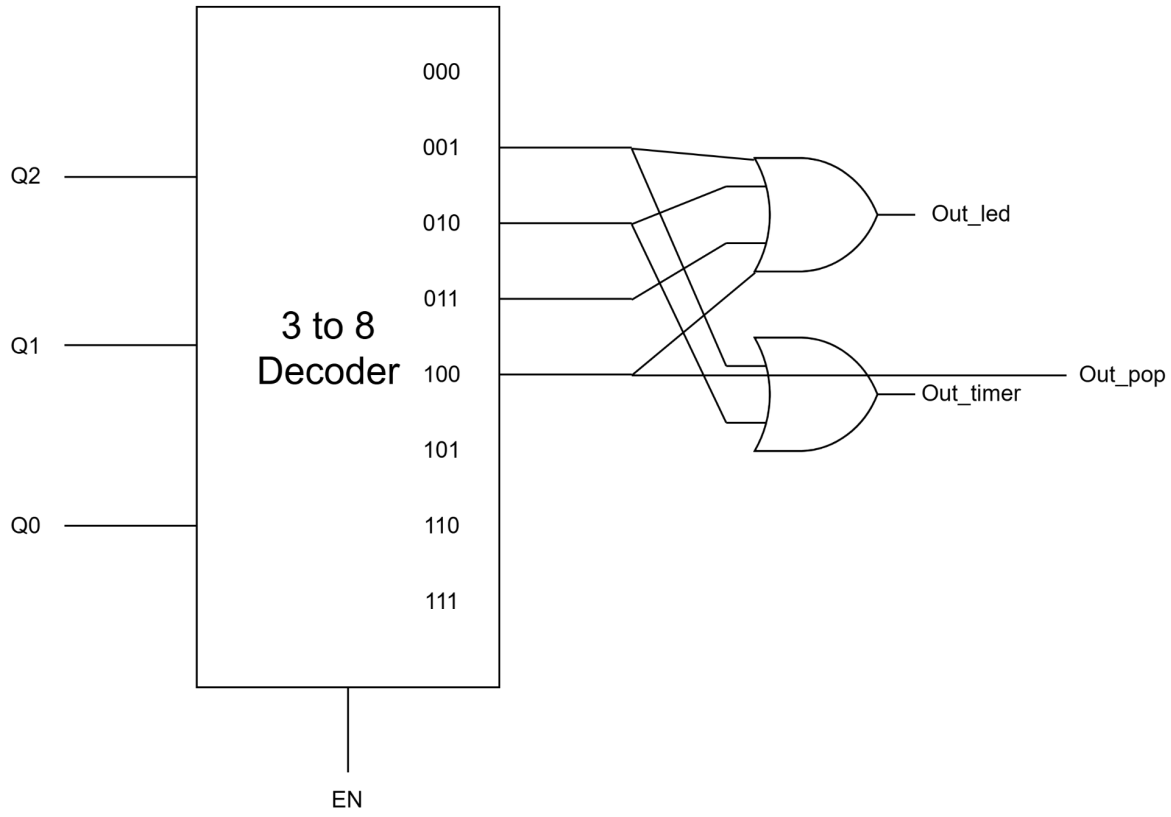
Number of total states: 5

In order to calculate the minimum number of bits n that should satisfy $2^n \geq number\ of\ states$

$$2^2 = 4 < 5 < 2^3 = 8 \Rightarrow n = 3$$

Therefore, 3 flip flops are required

**Redesign your outputs using decoders to simplify the combinational logic. Use a 3-to-8 decoder to generate one-hot encoded state signals from your state register bits, then express your output functions in terms of these decoded signals. This approach often leads to simpler sum-of-products expressions compared to direct Boolean minimization of state bits.**




Inputs: State Bits $Q_2$, $Q_1$, $Q_0$

Decoded Outputs: ($Y_n$ corresponds to state $n$):

$Y_0 = IDLE\ (000)$
$Y_1 = START\_TIMER\ (001)$
$Y_2 = BUSY\ (010)$
$Y_3 = DONE(011)$
$Y_4 = CHECK\_Q(100)$
$Out_{led} = Y_1 + Y_2 + Y_3 + Y_4$
$Out_{timer} = Y_1 + Y_2$
$Out_{pop} = Y_4$

**Write a SystemVerilog module implementing your FSM and a self-checking testbench that verifies functionality.**

```
`timescale 1ns / 1ps

module ParkingZoneFSM (
    input logic clk,
    input logic reset,
    input logic req_entry,     // Button press
    input logic timer_done,    // Timer finished signal
    input logic queue_valid,   // Queue has waiting cars
    output logic led_occupied, // 1 = Blink (Occupied), 0 = Solid (Free)
    output logic timer_start,  // 1 = Start/Run Timer
    output logic queue_pop     // 1 = Pop next car from queue
);

    // State Encoding
    typedef enum logic [2:0] {
        IDLE       = 3'b000,
        START_TIMER = 3'b001,
        BUSY       = 3'b010,
        DONE       = 3'b011,
        CHECK_Q    = 3'b100
    } state_t;

    state_t current_state, next_state;

    // State Register
    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            current_state <= IDLE;
        else
            current_state <= next_state;
    end

    // Next State Logic
    always_comb begin
        next_state = current_state; // Default
        case (current_state)
            IDLE: begin
                if (req_entry || queue_valid)
                    next_state = START_TIMER;
                else
                    next_state = IDLE;
            end
            START_TIMER: begin
```

```
            next_state = BUSY; // Unconditional transition
        end
        BUSY: begin
            if (timer_done)
                next_state = DONE;
            else
                next_state = BUSY;
        end
        DONE: begin
            next_state = CHECK_Q; // Unconditional transition
        end
        CHECK_Q: begin
            if (queue_valid)
                next_state = START_TIMER; // Loop back for queued car
            else
                next_state = IDLE;
        end
        default: next_state = IDLE;
      endcase
   end

   // Output Logic
   always_comb begin
      led_occupied = (current_state == START_TIMER) ||
                     (current_state == BUSY) ||
                     (current_state == DONE) ||
                     (current_state == CHECK_Q);

      timer_start  = (current_state == START_TIMER) || (current_state == BUSY);

      queue_pop    = (current_state == CHECK_Q) && queue_valid;
   end

endmodule

`timescale 1ns / 1ps

module tb_ParkingZoneFSM;
   logic clk, reset;
   logic req_entry, timer_done, queue_valid;
   logic led_occupied, timer_start, queue_pop;
   ParkingZoneFSM uut (.*);


   always #5 clk = ~clk;
```

```
    initial begin
        clk = 0; reset = 1;
        req_entry = 0; timer_done = 0; queue_valid = 0;

        #15 reset = 0;

        $display("Test 1: Normal Vehicle Arrival");
        #10 req_entry = 1;
        #10 req_entry = 0;

        #10;
        assert(led_occupied == 1) else $error("Error: LED should be ON (Blinking) in BUSY
state");
        assert(timer_start == 1)  else $error("Error: Timer should be enabled");

        #20 timer_done = 1;
        #10 timer_done = 0;

        #20;
        assert(uut.current_state == uut.IDLE) else $error("Error: Should return to IDLE when
queue is empty");
        assert(led_occupied == 0) else $error("Error: LED should be OFF (Solid) in IDLE");

        $display("Test 2: Queue Logic");
        queue_valid = 1;

        #10;
        assert(uut.current_state == uut.START_TIMER) else $error("Error: Should auto-start from
IDLE if Queue is valid");

        #10;
        timer_done = 1;
        #10 timer_done = 0;

        #10;
        assert(uut.current_state == uut.START_TIMER) else $error("Error: Should loop back to
START if Queue valid");
        assert(queue_pop == 0) else $error("Error: Queue pop logic timing check"); // Pop happens
in Check_Q state specifically

        queue_valid = 0;
        timer_done = 1; #10 timer_done = 0;

        #20;
        $display("All Tests Passed.");
        $finish;
```

```
    end
endmodule
```