CS322M: Finite State Machines (FSM)

Instructor: *Satyajit Das*

CSE, IIT Guwahati

Due: **19th August, 2025**

Learning Outcomes

- Model control problems using **Moore** and **Mealy** FSMs.
- Draw correct **state diagrams** and derive **transition logic**.
- Implement clean **synchronous** RTL in verilog with active-high reset.
- Build **self-checking testbenches** and generate waveforms.
- Compose systems from **multiple interacting FSMs** (req/ack handshake).
- Read the entire document before starting. Follow the instructions and remember the teachings from class.

What to Submit (Per Problem)

- **State Diagram:** labeled states, transitions, outputs.
- **Waveforms:** annotated screenshots of key events.
- **Code:** synthesizable RTL + testbench.
- **README:** compile/run/visualize steps; expected behavior.
- **GitHub:** repo URL and final commit hash.

Recommended Repo Layout

```
fsm-assignments/
  problem1_seqdet/
    seq_detect_mealy.v
    tb_seq_detect_mealy.v
    README.md
    waves/   (VCD/PNG)
  problem2_traffic/
    traffic_light.v
    tb_traffic_light.v
    README.md
    waves/
  problem3_vending/
    vending_mealy.v
    tb_vending_mealy.v
    README.md
    waves/
  problem4_link/
    master_fsm.v
    slave_fsm.v
    link_top.v
    tb_link_top.v
    README.md
```

Tools & How to Run (Examples)

Icarus Verilog + GTKWave:

```
iverilog -o sim.out <tb>.v
vvp sim.out
gtkwave dump.vcd
```

Verilator:

```
verilator --cc --exe --build tb_*.cpp *.v
./obj_dir/Vtb_*
```

ModelSim/Questa:

```
vlib work
vlog *.v
vsim -c work.tb_top -do "run -all; quit"
```

Each README must include exact commands and how to open waveforms.

Problem 1 (Mealy): Overlapping Sequence Detector

Goal: Detect serial bit pattern `1101` on `din` with **overlap**. Output `y` is a **1-cycle pulse** when the last bit arrives.

Type: Mealy **Reset:** synchronous, active-high.

Deliverables

- State diagram with proper fallback edges (overlap).
- Waveform with `din` and `y`; mark detection cycles.
- README: streams tested and expected pulse indices.

Problem 1: Module & Testbench Constructs

Module (interface only):

```
module seq_detect_mealy(
  input  wire clk,
  input  wire rst,  // sync active-high
  input  wire din,  // serial input bit per clock
  output wire y     // 1-cycle pulse when pattern ...1101 seen
);
// Implement states & Mealy output
endmodule
```

Testbench (skeleton):

```
module tb_seq_detect_mealy;
  reg  clk, rst, din;
  wire y;

  seq_detect_mealy dut(.clk(clk), .rst(rst), .din(din), .y(y));

  // 1) Clock gen (e.g., 100 MHz) and sync reset
  // 2) Drive a bitstream with overlaps, e.g.: 11011011101
  // 3) Log time, din, y
  // 4) $dumpfile("dump.vcd"); $dumpvars(0, tb_seq_detect_mealy);
```

Problem 2 (Moore): Two-Road Traffic Light

Goal: Control NS/EW with shared `tick` (1 Hz).
Timing: NS G 5 ticks → NS Y 2 → EW G 5 → EW Y 2 → repeat.
Outputs: `ns_g, ns_y, ns_r, ew_g, ew_y, ew_r`. Exactly one of {g,y,r} high per road.
Type: Moore **Reset:** synchronous, active-high.

Deliverables

- State diagram with four phases and tick-based transitions.
- Waveform showing durations (5/2/5/2 ticks).
- README: how 1 Hz tick was generated and verified.

Problem 2: Module & Testbench Constructs

Module (interface only):

```
module traffic_light(
  input  wire clk,
  input  wire rst,   // sync active-high
  input  wire tick,  // 1-cycle per-second pulse
  output wire ns_g, ns_y, ns_r,
  output wire ew_g, ew_y, ew_r
);
// Implement state machine and per-phase tick counter
endmodule
```

Testbench (skeleton):

```
module tb_traffic_light;
  reg clk, rst, tick;
  wire ns_g, ns_y, ns_r, ew_g, ew_y, ew_r;

  traffic_light dut(
    .clk(clk), .rst(rst), .tick(tick),
    .ns_g(ns_g), .ns_y(ns_y), .ns_r(ns_r),
    .ew_g(ew_g), .ew_y(ew_y), .ew_r(ew_r)
  );
```

Problem 3 (Mealy): Vending Machine with Change

Goal: Price = 20. Accept coins 5 or 10. When total $\geq$ 20: `dispense`=1 (1 cycle). If total=25: also `chg5`=1 (1 cycle). Reset total after vend.
coin[1:0]: `01`=5, `10`=10, `00`=idle (ignore `11`). One coin max per cycle.
Type: Mealy **Reset:** synchronous, active-high.

Deliverables

- State diagram + brief justification for Mealy.
- Waveform highlighting vend & change pulses.

Problem 3: Module & Testbench Constructs

Module (interface only):

```
module vending_mealy(
  input  wire       clk,
  input  wire       rst,     // sync active-high
  input  wire [1:0] coin,    // 01=5, 10=10, 00=idle
  output wire       dispense,// 1-cycle pulse
  output wire       chg5     // 1-cycle pulse when returning 5
);
// Implement states for totals (0,5,10,15) and Mealy outputs
endmodule
```

Testbench (skeleton):

```
module tb_vending_mealy;
  reg  clk, rst;
  reg  [1:0] coin;
  wire dispense, chg5;

  vending_mealy dut(.clk(clk), .rst(rst), .coin(coin),
                    .dispense(dispense), .chg5(chg5));

  // 1) Clock + reset
```

Problem 4 (Two FSMs): Master–Slave Handshake

Goal: Two FSMs (**Master**, **Slave**) with 4-phase **req/ack** and 8-bit data bus.
Per byte:

1. Master drives `data`, raises `req`.
2. Slave latches `data` on `req`, asserts `ack` (hold 2 cycles).
3. Master sees `ack`, drops `req`; Slave then drops `ack`.
4. Repeat for 4 bytes; Master asserts `done` (1 cycle) at end.

Reset: synchronous, active-high **Clock:** common `clk`.

Deliverables

- Two state diagrams (Master, Slave) + timing diagram (req/ack/data).
- Waveform showing 4 handshakes and `done` pulse.

Problem 4: Module Constructs

Master (4-byte burst, e.g., `A0..A3`):

```
module master_fsm(
  input  wire       clk,
  input  wire       rst,   // sync
  input  wire       ack,
  output wire       req,
  output wire [7:0] data,
  output wire       done   // 1-cycle when burst completes
);
// Implement handshake states and byte index
endmodule
```

Slave (latch on req, assert ack 2 cycles):

```
module slave_fsm(
  input  wire       clk,
  input  wire       rst,     // sync
  input  wire       req,
  input  wire [7:0] data_in,
  output wire       ack,
  output wire [7:0] last_byte // observable for TB
);
```

Problem 4: Top & Testbench Constructs

Top (connects both FSMs):

```
module link_top(
  input  wire clk,
  input  wire rst,
  output wire done
);
// Instantiate master_fsm and slave_fsm, wire req/ack/data
endmodule
```

Testbench (skeleton):

```
module tb_link_top;
  reg  clk, rst;
  wire done;

  link_top dut(.clk(clk), .rst(rst), .done(done));

  // 1) Clock + reset
  // 2) Monitor handshake via hierarchical refs:
  //    dut.<inst>.req, dut.<inst>.ack, dut.<inst>.data
  // 3) Run until 'done' high, then stop
  // 4) VCD and annotate the 4 handshakes
```

Design Guidance: Mealy vs. Moore

- **Mealy**: output can change immediately with input (fewer states; watch glitches).
- **Moore**: output depends only on state (clean timing; often 1-cycle delayed).
- Use **synchronous** active-high reset; transition on `posedge clk`.
- One-hot or binary encodings are acceptable; justify your choice.

Waveform Expectations & Dumping

In every testbench:

- `$dumpfile("dump.vcd"); $dumpvars(0, <tb_name>);` (or tool equivalent).
- Include key signals: inputs, outputs, and optionally state (via debug wires).
- Mark expected cycles in README (detection pulses, tick edges, ack windows).

Submission Checklist

- Four folders (one per problem) with RTL, TB, README, waves/.
- GitHub URL + commit hash (`git rev-parse HEAD`) in `SUBMIT.md`.

What is `tick` in Problem 2?

`tick` is a **slow, 1-cycle pulse** that marks elapsed real time (e.g., **1 second**). The traffic-light FSM still runs on the fast `clk` (e.g., 50 MHz), but it advances its phase counter *only* when `tick=1`. This avoids counting millions of `clk` cycles inside the FSM.

Why use `tick`?

- Clean separation of **time-keeping** (prescaler) from **control logic** (FSM).
- Easy to change timing (make `tick` 2 Hz, 10 Hz, etc.) without touching the FSM.
- Simulation becomes faster and clearer (you can generate artificial `tick` pulses).

Prescaler Overview (Tick Generator)

Prescaler takes the fast `clk` and emits a **single-cycle pulse** `tick` at a lower rate (`TICK_HZ`).
Example: with `CLK_FREQ_HZ`=50 000 000 and `TICK_HZ`=1, `tick` asserts once per second.

Design notes

- Use a counter that rolls over every $\frac{\texttt{CLK\_FREQ\_HZ}}{\texttt{TICK\_HZ}}$ cycles.
- Make `tick` exactly **one** `clk` cycle wide at rollover.
- Keep reset **synchronous, active-high**.

Prescaler (Tick Generator) — Module Construct

Interface only (students implement internals):

```
module tick_prescaler #(
  parameter integer CLK_FREQ_HZ = 50_000_000, // fast clock rate
  parameter integer TICK_HZ     = 1           // desired tick rate
)(
  input  wire clk,   // fast system clock
  input  wire rst,   // synchronous active-high reset
  output wire tick   // 1-cycle pulse at TICK_HZ
);
// Implement a counter that asserts 'tick' for 1 clk on rollover
endmodule
```

How it connects to Problem 2:

```
// Top-level (example wiring)
wire tick_1hz;
tick_prescaler #(.CLK_FREQ_HZ(50_000_000), .TICK_HZ(1)) u_div (
  .clk(clk), .rst(rst), .tick(tick_1hz)
);

traffic_light u_tl (
  .clk(clk), .rst(rst), .tick(tick_1hz),
```

Testbench-Friendly Tick (Fast Simulation)

In simulation, you usually don't want to wait real-time seconds. Generate a **faster** `tick` (e.g., every 20 `clk` cycles) directly in the TB:

```
// Inside tb_traffic_light (example)
reg clk, rst, tick;
integer cyc;

always @(posedge clk) begin
  cyc  <= cyc + 1;
  tick <= (cyc % 20 == 0); // 1-cycle pulse every 20 cycles (fast sim)
end

// Drive clk and rst as usual
```

Tip: Document your simulation tick period in the README and scale the phase durations accordingly.

Hardware Tick: Parameterizable Divider (Construct)

Alternative construct (students may prefer explicit terminal count):

```
module tick_divider #(
  parameter integer CLK_FREQ_HZ = 50_000_000,
  parameter integer TICK_HZ     = 1
)(
  input  wire clk,
  input  wire rst,   // sync active-high
  output wire tick   // 1-cycle pulse each 1/TICK_HZ seconds
);
// localparam integer TERMINAL = CLK_FREQ_HZ / TICK_HZ;
// Count 0..TERMINAL-1; assert 'tick' when counter == TERMINAL-1
endmodule
```

Usage: Replace `tick_prescaler` with `tick_divider` in your top-level if you prefer this style. Both are acceptable solutions as long as `tick` is a clean, single-cycle pulse.

Reminder for Students (Tick Integration)

- **FSM runs on `clk`**; phase counter increments **only on `tick`**.
- Verify in waveforms that `tick` is **one-cycle wide**, evenly spaced.
- Show in README: clock frequency, chosen `TICK_HZ`, and how you verified 5/2/5/2 tick durations.

Good Luck!

Build, simulate, visualize, and reason.

Questions? Post on the course forum.