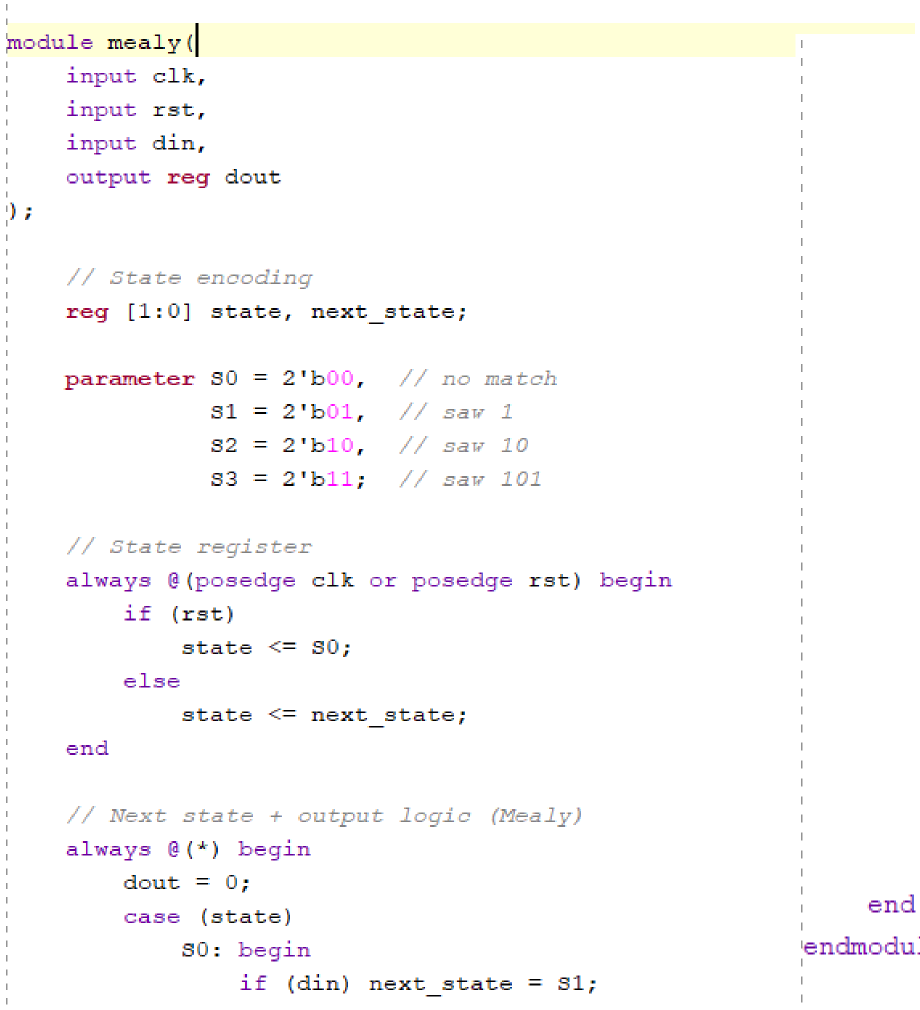
Day 2: Skill Integration & Modern Tool Exploration

VHDL FSM Design & NVIDIA Transformer Engine

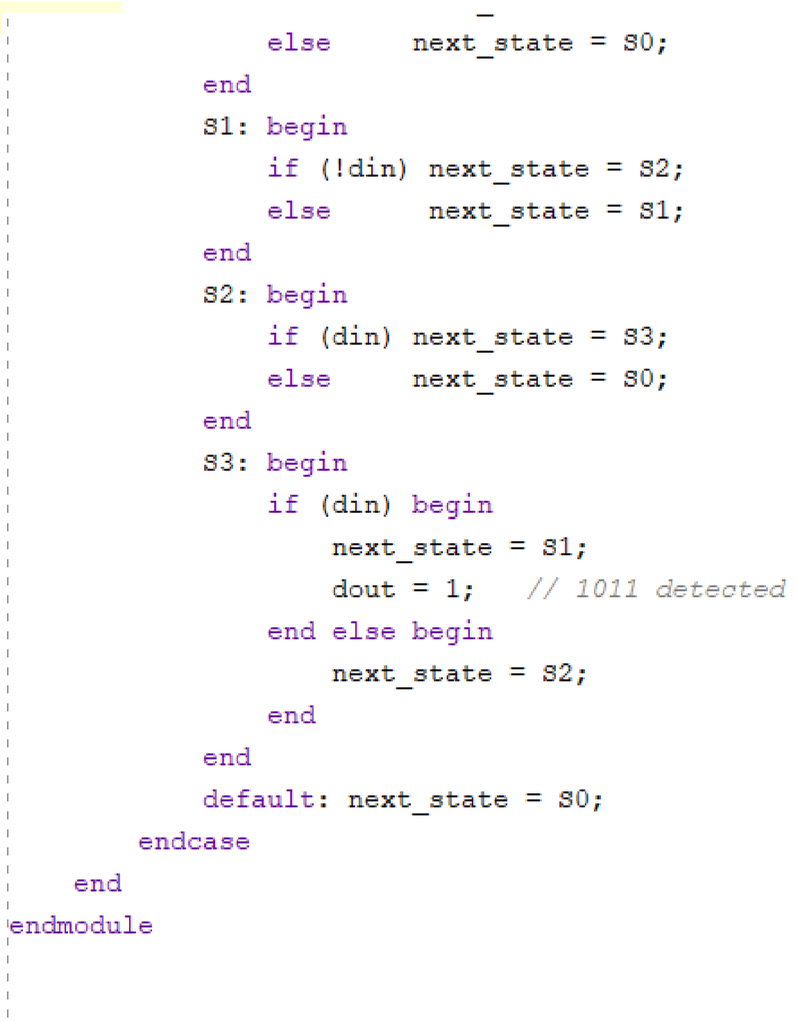
Core Philosophy

- Bridge foundational digital design with modern AI hardware
- Strengthen VHDL fundamentals through hands-on FSM design
- Develop a performance-oriented engineering mindset

Mealy FSM – code

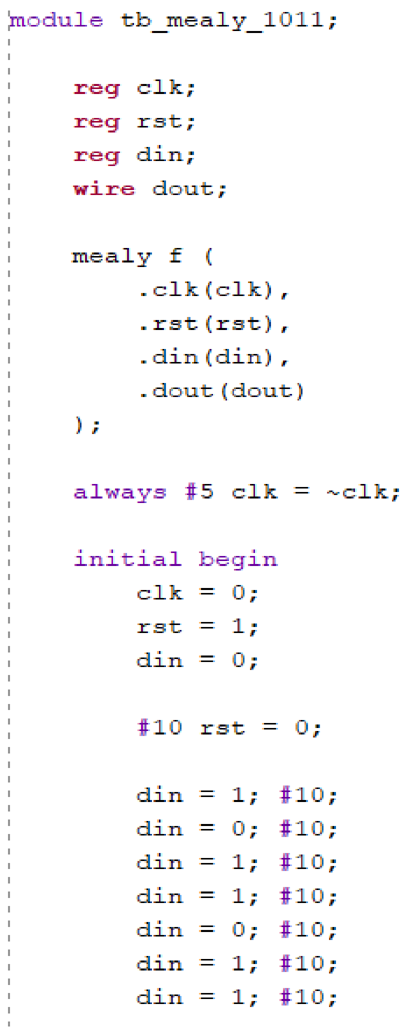


```verilog
module mealy(
    input clk,
    input rst,
    input din,
    output reg dout
);

    // State encoding
    reg [1:0] state, next_state;

    parameter S0 = 2'b00,  // no match
              S1 = 2'b01,  // saw 1
              S2 = 2'b10,  // saw 10
              S3 = 2'b11;  // saw 101

    // State register
    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= S0;
        else
            state <= next_state;
    end

    // Next state + output logic (Mealy)
    always @(*) begin
        dout = 0;
        case (state)
            S0: begin
                if (din) next_state = S1;
```

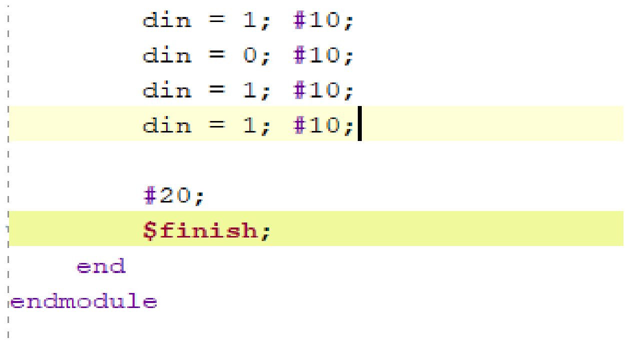


```verilog
                else     next_state = S0;
            end
            S1: begin
                if (!din) next_state = S2;
                else      next_state = S1;
            end
            S2: begin
                if (din) next_state = S3;
                else     next_state = S0;
            end
            S3: begin
                if (din) begin
                    next_state = S1;
                    dout = 1;   // 1011 detected
                end else begin
                    next_state = S2;
                end
            end
            default: next_state = S0;
        endcase
    end
endmodule
```

Mealy FSM – Testbench



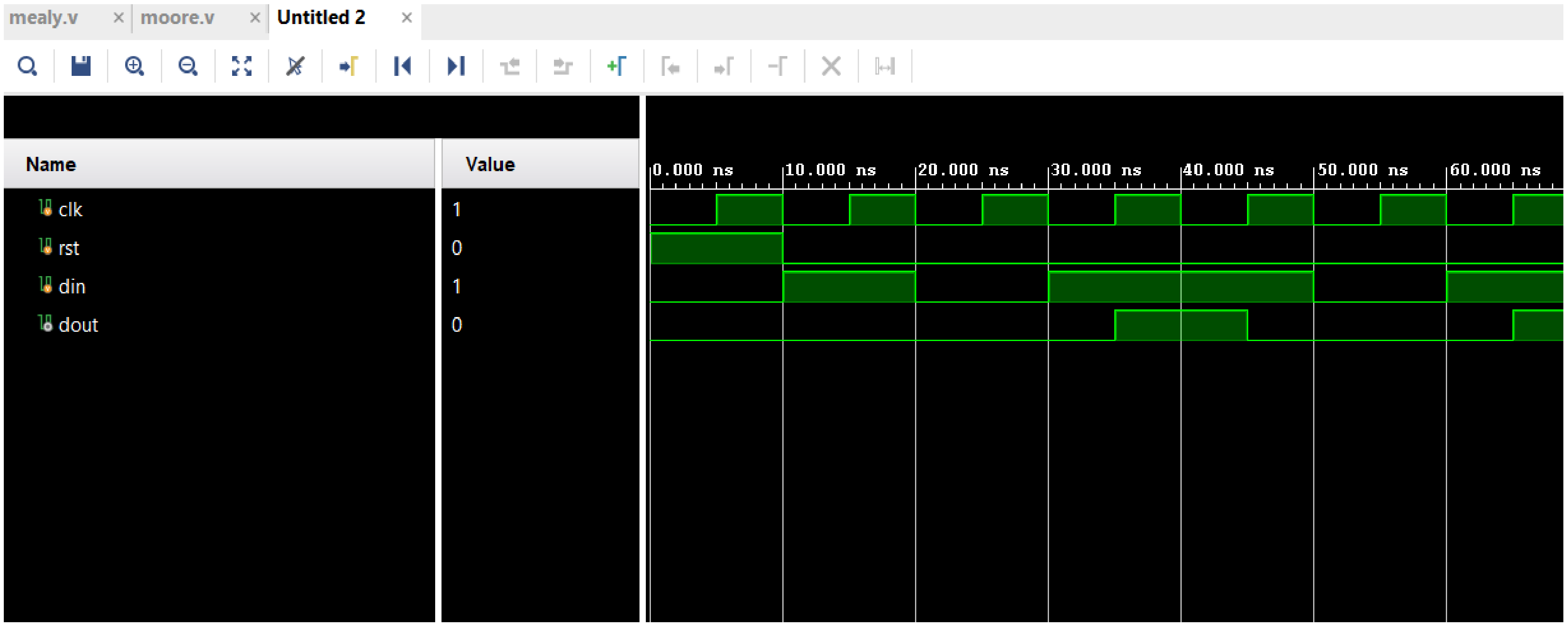
```
module tb_mealy_1011;

    reg clk;
    reg rst;
    reg din;
    wire dout;

    mealy f (
        .clk(clk),
        .rst(rst),
        .din(din),
        .dout(dout)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        rst = 1;
        din = 0;

        #10 rst = 0;

        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;
```



```
        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;

        #20;
        $finish;
    end
endmodule
```

Mealy FSM – Output

- Output depends on current state and input
- Faster output response
- Requires fewer states than Moore FSM



Name	Value
clk	1
rst	0
din	1
dout	0

Moore FSM – Code

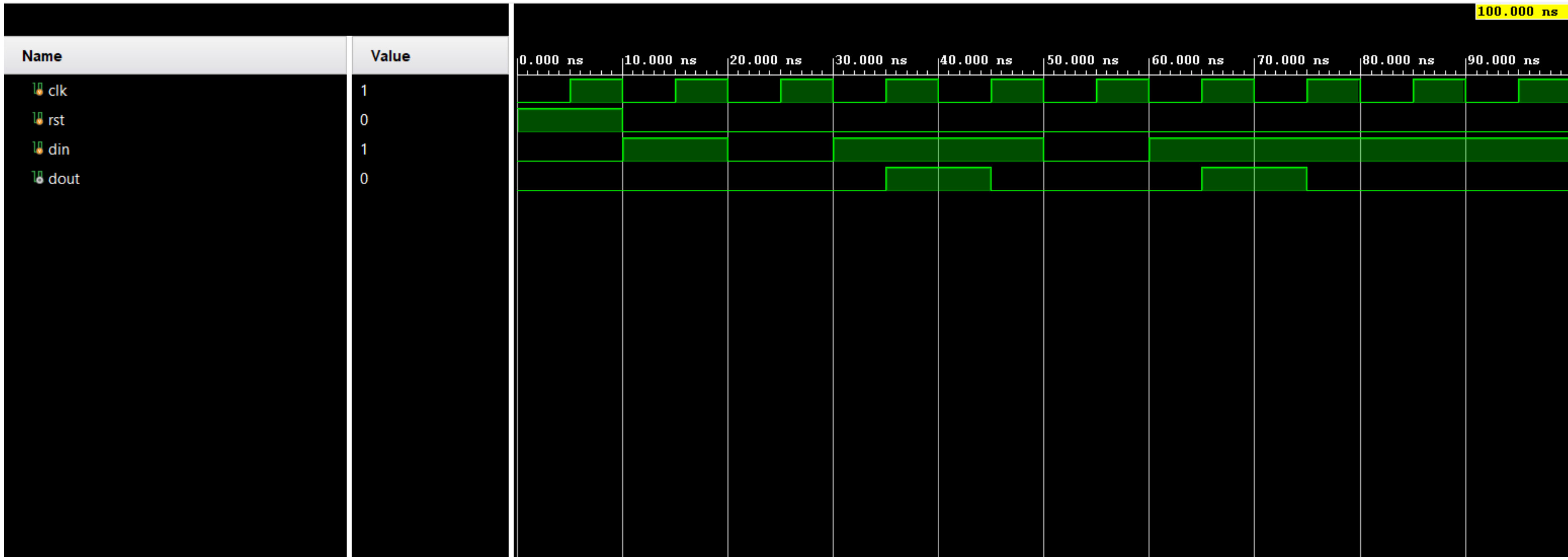
```verilog
module moore (
    input clk,
    input rst,
    input din,
    output reg dout
);

    // State encoding
    reg [2:0] state, next_state;

    parameter S0 = 3'b000,   // no match
              S1 = 3'b001,   // saw 1
              S2 = 3'b010,   // saw 10
              S3 = 3'b011,   // saw 101
              S4 = 3'b100;   // saw 1011 (output state)

    // State register
    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= S0;
        else
            state <= next_state;
    end

    // Next state logic
    always @(*) begin
        case (state)
            S0: if (din) next_state = S1; else next_state = S0;
            S1: if (!din) next_state = S2; else next_state = S1;
            S2: if (din) next_state = S3; else next_state = S0;
            S3: if (din) next_state = S4; else next_state = S2;
            S4: next_state = S1;  // allow overlapping
            default: next_state = S0;
        endcase
    end

    // Output logic (Moore → depends only on state)
    always @(*) begin
        if (state == S4)
            dout = 1;
        else
            dout = 0;
    end
endmodule
```

Moore FSM - Testbench

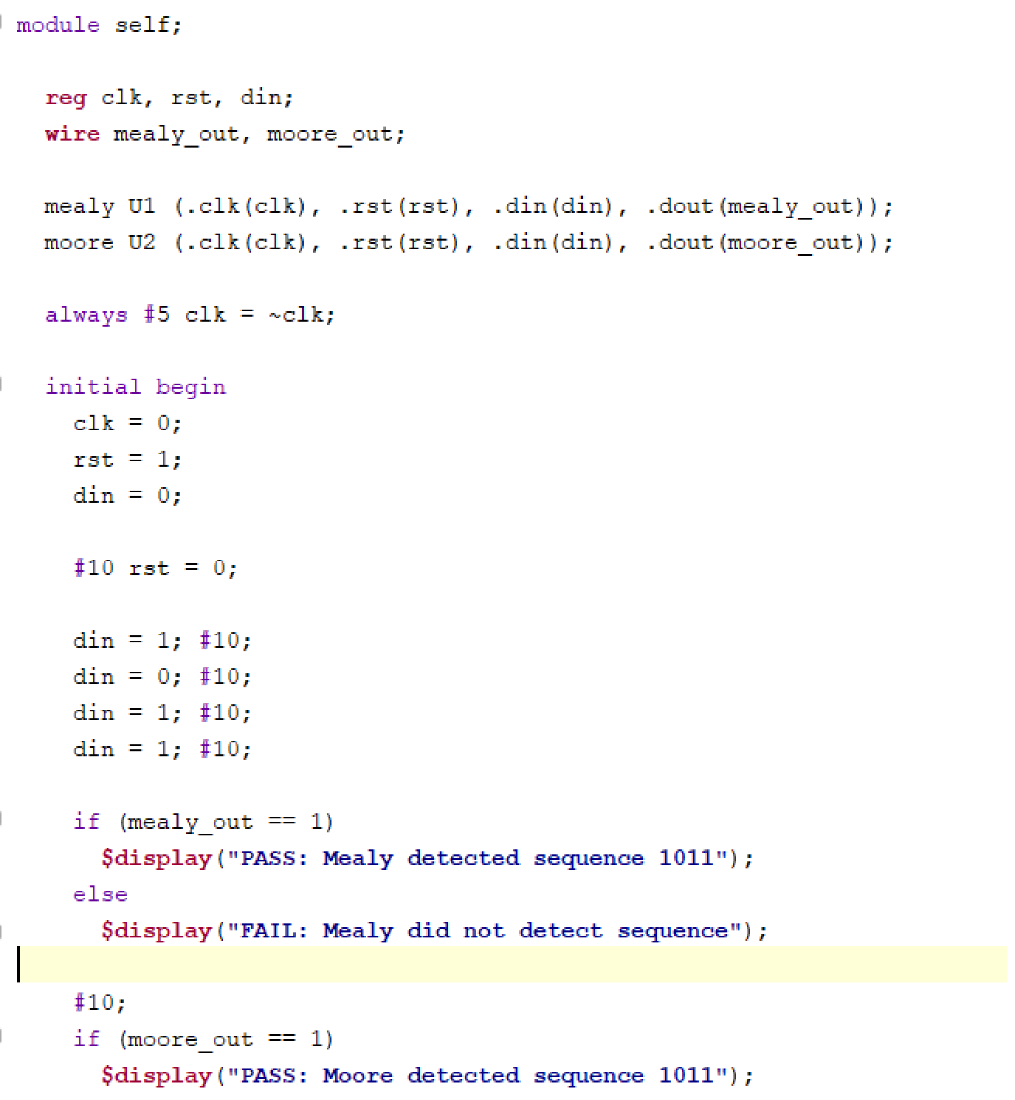
```
module tb_moore_1011;

    reg clk;
    reg rst;
    reg din;
    wire dout;

    moore f (
        .clk(clk),
        .rst(rst),
        .din(din),
        .dout(dout)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        rst = 1;
        din = 0;

        #10 rst = 0;

        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;
        din = 0; #10;
        din = 1; #10;
        din = 1; #10;
        din = 1; #10;

        #20;
        $finish;
    end
endmodule
```

Moore FSM - Output

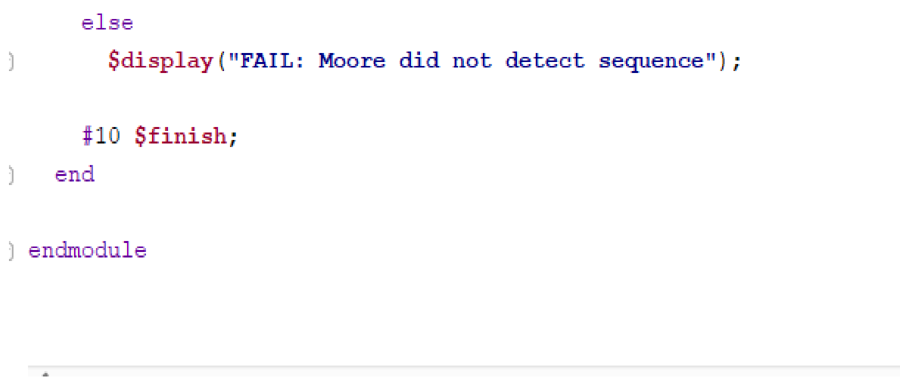
- Output depends only on current state
- Glitch-free and stable output
- Uses an extra state for output generation



Self Checking Testbench



```
module self;

  reg clk, rst, din;
  wire mealy_out, moore_out;

  mealy U1 (.clk(clk), .rst(rst), .din(din), .dout(mealy_out));
  moore U2 (.clk(clk), .rst(rst), .din(din), .dout(moore_out));

  always #5 clk = ~clk;

  initial begin
    clk = 0;
    rst = 1;
    din = 0;

    #10 rst = 0;

    din = 1; #10;
    din = 0; #10;
    din = 1; #10;
    din = 1; #10;

    if (mealy_out == 1)
      $display("PASS: Mealy detected sequence 1011");
    else
      $display("FAIL: Mealy did not detect sequence");


    #10;
    if (moore_out == 1)
      $display("PASS: Moore detected sequence 1011");
```



```
    else
      $display("FAIL: Moore did not detect sequence");

    #10 $finish;
  end

endmodule
```

A self-checking testbench automatically verifies the DUT output against expected results using conditional checks.
It eliminates manual waveform analysis by clearly reporting PASS or FAIL during simulation.

Blocking vs Non-Blocking Assignments

```
module blocking_swap (
    input  wire clk,
    input  wire rst,
    output reg  a,
    output reg  b
);

always @(posedge clk or posedge rst) begin
    if (rst) begin
        a = 0;
        b = 1;
    end else begin
        a = b;
        b = a;
    end
end

endmodule
```

Blocking assignments execute sequentially, so the second assignment sees the updated value.
This causes incorrect swapping when used in clocked logic.

```
module nonblocking_swap (
    input  wire clk,
    input  wire rst,
    output reg  a,
    output reg  b
);

always @(posedge clk or posedge rst) begin
    if (rst) begin
        a <= 0;
        b <= 1;
    end else begin
        a <= b;
        b <= a;
    end
end

endmodule
```

Non-blocking assignments update registers in parallel at the clock edge.
This correctly models hardware behavior and enables proper swapping.

VHDL Timing Concepts

Single-cycle vs Multi-cycle Paths

- A single-cycle path must complete all logic within one clock period, making timing constraints tight.
  A multi-cycle path is allowed to take multiple clock cycles, relaxing timing requirements and improving design flexibility.

False Paths

- False paths are signal paths that never occur during normal operation of the circuit.
  They are excluded from timing constraints to prevent unnecessary timing violations during analysis.

Critical Path Reduction Using Pipelining

- Pipelining breaks long combinational logic into smaller stages separated by registers.
  This reduces critical path delay and allows the circuit to run at a higher clock frequency.

NVIDIA Transformer Engine

•NVIDIA Transformer Engine is a high-performance software library designed to accelerate Transformer-based deep learning models.
•It enables the use of **FP8 precision**, which reduces memory usage while maintaining model accuracy.
•The library dynamically selects precision formats to balance performance and numerical stability.
•Transformer Engine is tightly optimized for **NVIDIA Hopper GPU architecture**.
•It improves training and inference speed for large language models and AI workloads.
•The engine integrates seamlessly with popular frameworks like **PyTorch**.
•FP8 computation is enabled using an **autocast mechanism with FP8 recipes**.
•It requires CUDA and cuDNN, making it a high-performance, hardware-aware library.
•Transformer Engine demonstrates strong **hardware–software co-design principles**.
•Its goal is to maximize throughput, reduce latency, and improve efficiency in modern AI systems.