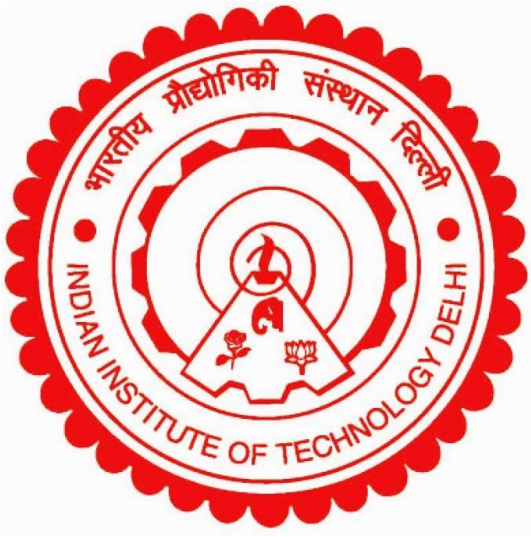
Indian Institute of Technology,Delhi

ELL 1401 Project Report

Mano-Machine: A CPLD-Based FSM Engine



ADEEB ALI ISLAM

2025EE11232

Table 1 and 3, Friday

Adeeb Ali Islam
2025EE11232

Kushagra Nath
2025EE11241

Ekatva Kushvaha
2025EE11673

Vedant Somwanshi
2025EE11707

1. Abstract

This project is not a single FSM, but a reusable "meta-machine" designed to run, test, and debug *any* Finite State Machine on a CPLD.

This project presents a complete Finite State Machine (FSM) validation workflow, which is composed of two primary innovations:

1. A **High-Level Synthesis (HLS) Tool:** built in Python, which intelligently parses an Excel-based state table and auto-generates robust, synthesis-ready Verilog code.

2. **CPLD-Based Hardware Debugger:** that runs the generated code. This debugger features a novel Clock Gating mechanism, allowing a user to either run the FSM in real-time or freeze it in "Step Mode" to advance the logic *one clock cycle at a time* for precise, physical validation.

Also, we can observe the present state number on a 7-segment display, allowing us to visualise the FSM's internal state. This "Mano-Machine" transforms the CPLD from a "black box" into a fully transparent, "meta-machine" for hardware debugging.

2. Introduction (The Problem & Vision)

Standard FSM design involves two pain points:

1. **Tedious Coding:** Manually translating a large state table into case statements is slow and highly prone to human error.
2. **Difficult Debugging:** A simulation (software) doesn't prove hardware validity, and a "black box" CPLD (hardware) gives no insight into *why* it's failing.

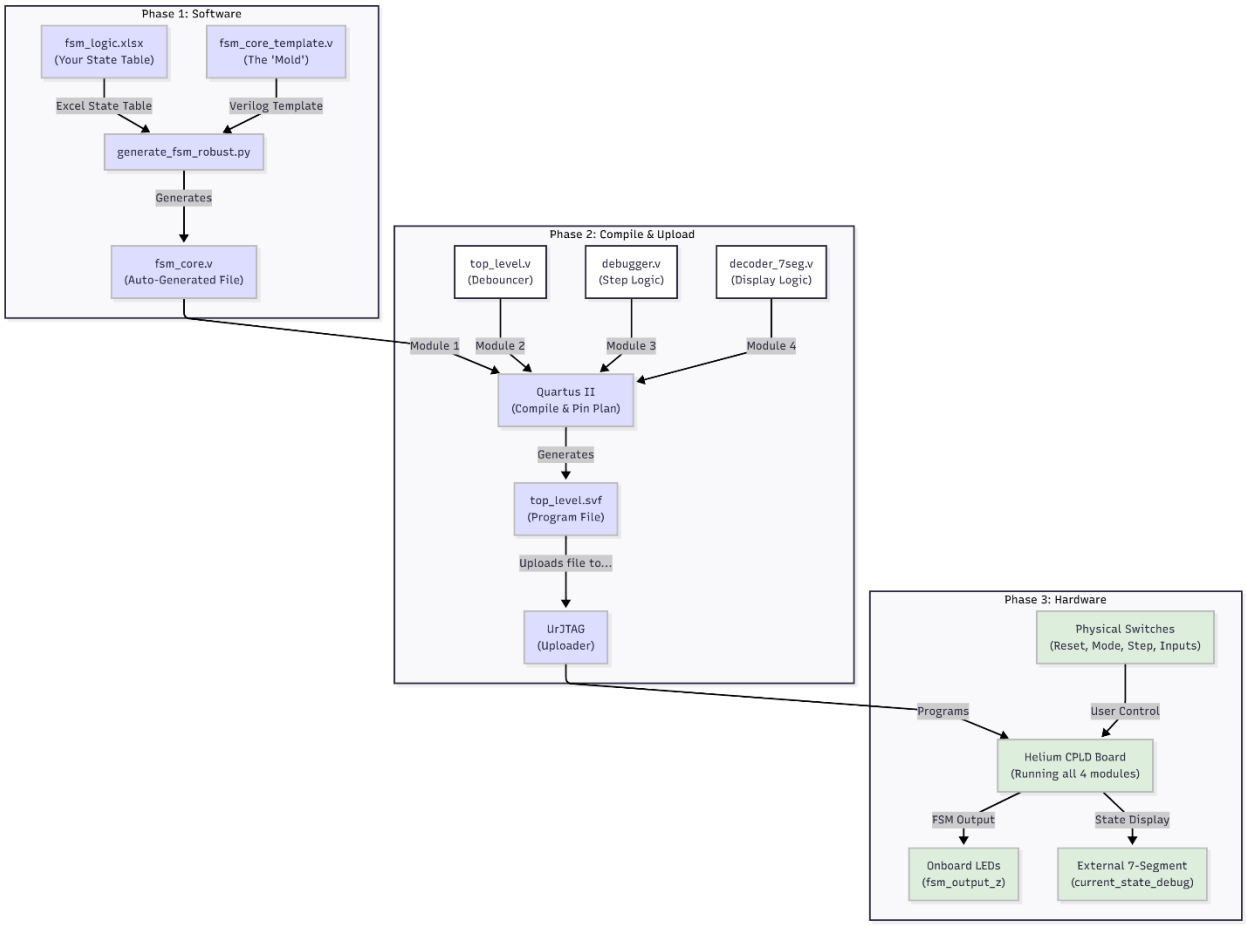
Our project solves both problems. The "Mano-Machine" is a complete engine. A user first defines their FSM in a simple Excel file. A Python script then auto-generates the Verilog. Finally, our CPLD hardware runs that FSM on two modes:

1) Run Mode: The FSM runs continuously along with the inbuilt 1 Hz clock
2) Step Mode: The FSM freezes at the state user wants to, the user could change the inputs to observe the present output and update the state (through a step button) to go to the next state.

While the present state is visible on a 7 segment display. Allowing for transparent access to the FSM's every state.(Max Input width = 2, Max output width = 4, Max number of states = 16)

3. System Architecture (The Blueprint)

Our architecture is split int o a software "synthesis" phase and a hardware "implementation" phase.



Basically the following steps happen,

User writes output and next state in a template excel sheet(shown below) →

Runs python program from terminal (python fsm_generator.py <excel sheet name>) →

Obtains a fsm_core file with case statements implementing fsm →

Compiles whole code and runs on CPLD →

Mode Switch : If high, working in run mode; If low, working in step mode(freezes in current state until step switch used)

Step Switch : In step mode, used to update to next state by changing switch from low to high (edge detection)

4. Implementation Methodology & Code

Our project contains two "killer features" that make it an innovative tool.

Innovation 1: The High-Level Synthesis (HLS) Python Tool

We did not write the main FSM Verilog code by hand. We wrote a Python script (generate_fsm_robust.py) to do it for us.

- **Methodology** : The script uses the pandas library to read an E xcel (.xlsx) state table. It intelligently loops through every row and auto -generates the Verilog case statements for the *fsm_core.v* module.
- **Smart "Anchor" System** : The script injects this logic into the `fsm_core_template.v` file by searching for two "anchor" comments: `//PYTHON-GEN-START` and `//PYTHON-GEN-END`. It replaces *only* the code between these anchors.
- **Robustness (Error Handling):** The script is "bulletproof" and handles the two biggest user errors:
    1. **Unused Inputs:** If a user leaves an input (like `10` or `11`) blank in Excel, the script skips it. The generated code *automatically* adds a `default:` case (e.g., `default: next_state = S0;`) to catch these inputs and force a safe state.
    2. **Unused States:** If a user only defines 10 states (S0-S9), the Python script only writes those 10. The *template file itself* has a master `default:` block to catch the 6 unused states (S10-S15) and force a safe reset, preventing unknown `XXXX` states.

Innovation 2: The Hardware Debugger (Clock Gating & Edge Detection)

This is the "magic" that makes our hardware work. It's built from two parts in `debugger.v`.

A) The Clock Gating (Step Mode): We control the FSM's clock with a `clk_enable` signal. The FSM's main register can *only* update when this signal is `1`. As seen in fsm_core.v :

```
always @(posedge clk or negedge reset) begin
  if (reset == 1'b0) // <-- Check for low level (button PRESSED)
    present_state <= S0;
  else if (clk_enable)
    present_state <= next_state;
end
```

The `debugger.v` module generates this "gate"(clk_enable) signal.

In "Run Mode" (`SW1`=UP), it's always `1`.

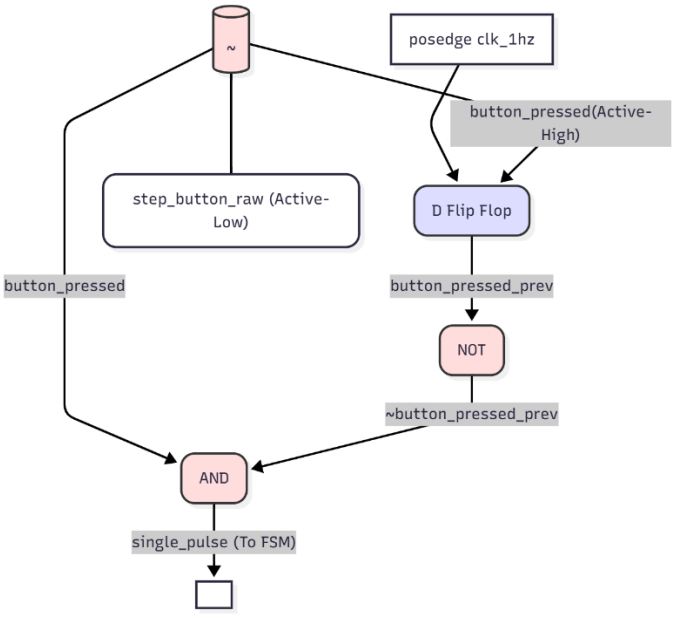
In "Step Mode" (`SW1`=DOWN), it is 0 (freezing the FSM) *until* it receives a `single_pulse`.

```
//Decides whether in Run Mode or Step Mode
    always @(*) begin
        //In Run Mode
        if (mode_switch_raw == 1'b1)
            fsm_clk_enable = 1'b1;
        else // In Step Mode
            fsm_clk_enable = single_pulse;
    end
```

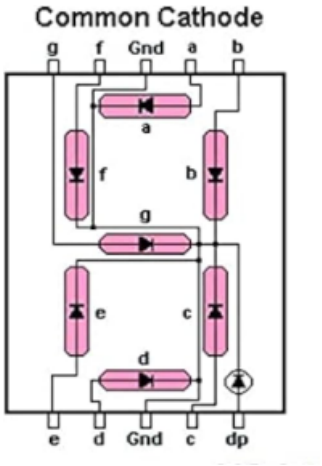
B) The Edge Detector (The "Single Pulse" Circuit): To make the "Step" button work, we can't just pass the button press. A user holds a button for 1-2 seconds (1-2 clock ticks), which would cause 1-2 state changes. We need to turn that *long* press into a *single, 1-cycle pulse*. We do this with a "Rising Edge Detector."

- **Logic:** assign single_pulse = button_pressed & ~button_pressed_prev; (in debugger.v)
- **Explanation:** This logic is "true" for *only one clock cycle*: the exact moment when the button is pressed (button_pressed=1) but the "history" register from the last cycle (button_pressed_prev) is still 0. On the very next clock tick, both are 1, so the pulse goes back to 0, freezing the FSM again.

Circuit Diagram (Logic for the Edge Detector):



Common Cathode Display:



4.2. Excel Template + HLS Code + Verilog Template (HLS Toolkit)

(explanation for codes is given above, comments in the codes will highlight more)

1) Excel Template: (The following template goes from S0 to S15)

	A	B	C	D
1	**Present State**	**Input**	**Next State**	**Output**
2	S0	00		
3	S0	01		
4	S0	10		
5	S0	11		
6	S1	00		
7	S1	01		
8	S1	10		
9	S1	11		
10	S2	00		
11	S2	01		
12	S2	10		
13	S2	11		
14	S3	00		
15	S3	01		
16	S3	10		
17	S3	11		
18	S4	00		
19	S4	01		
20	S4	10		
21	S4	11		
22	S5	00		
23	S5	01		
24	S5	10		
25	S5	11		
26	S6	00		
27	S6	01		

2) fsm_generator.py (HSL code, reads excel, rewrites/molds template to final fsm)

```
3) import pandas as pd
4) import os
5) import sys
6)
7) VERILOG_TEMPLATE = 'fsm_core_template.v'
8) OUTPUT_FILE = 'fsm_core.v'
9)
10) def generate_verilog_from_excel(excel_file_path):
11)     print(f"Reading FSM table from: {excel_file_path}...")
12)     try:
13)         # Read Excel, force Input/Output to be STRINGS
14)         df = pd.read_excel(excel_file_path, dtype={'Input': str, 'Output':
    str})
15)     except Exception as e:
16)         print(f"Error: Could not read Excel file. Check file or path.")
17)         print(f"Details: {e}")
18)         return
19)
20)     # This is how we handle empty/template rows. Drop any row which has one
    of the cells empty
21)     original_count = len(df)
22)     df.dropna(subset=['Present State', 'Input', 'Next State', 'Output'],
    how='any', inplace=True)
23)     final_count = len(df)
24)     print(f"Found {final_count} valid transitions. {original_count -
    final_count} empty/partial rows were skipped.")
25)
26)     print("Generating ROBUST logic (forcing 2-bit I/O)...")
27)
28)     logic_buffer = ""
29)     grouped = df.groupby('Present State')
30)
31)     for state, group in grouped:
32)         logic_buffer += f"\t\t{state}: begin\n"
33)         logic_buffer += "\t\t\tcase (x_in)\n"
34)
35)
36)         for index, row in group.iterrows():
37)             inp_val = str(row['Input']).zfill(2) # Pad with 0 if user just
    wrote "1"
```

```
38)             next_s = row['Next State']
39)             out_val = str(row['Output']).zfill(4) # Pad with 000 if user
    just wrote "1"
40)
41)
42)             logic_buffer += f"\t\t\t\t2'b{inp_val}: begin next_state =
    {next_s}; z = 4'b{out_val}; end\n"
43)
44)         # DEFAULT PREVENTS LOCK-OUT IN CASE OF UNUSED INPUTS FOR A STATE
45)         logic_buffer += "\t\t\t\tdefault: begin next_state = S0; z =
    4'b0000; end\n"
46)
47)         logic_buffer += "\t\t\tendcase\n"
48)         logic_buffer += "\t\tend\n\n"
49)
50)     # Read Template
51)     try:
52)         with open(VERILOG_TEMPLATE, 'r') as f:
53)             template_lines = f.readlines()
54)     except FileNotFoundError:
55)         print(f"Error: Template file '{VERILOG_TEMPLATE}' not found.")
56)         return
57)
58)     # COMPILING THE WHOLE CONTENT TO WRITE
59)     final_lines = []
60)     in_gen_block = False
61)
62)     for line in template_lines:
63)         if "//PYTHON-GEN-START" in line:
64)             final_lines.append(line)
65)             final_lines.append(logic_buffer)
66)             in_gen_block = True
67)         elif "//PYTHON-GEN-END" in line:
68)             final_lines.append(line)
69)             in_gen_block = False
70)         elif not in_gen_block:
71)             final_lines.append(line)
72)     # REWROTE fsm_core.v
73)     with open(OUTPUT_FILE, 'w') as f:
74)         f.writelines(final_lines)
75)
76)     print(f"Success! Overwrote {OUTPUT_FILE} with logic from
    {excel_file_path}.")
77)
78) if __name__ == "__main__":
```

```
79)
80)
81)     if len(sys.argv) != 2:
82)         print("Error: You must provide exactly one Excel file name.")
83)         print(f"Usage: python {sys.argv[0]} my_fsm_table.xlsx")
84)         sys.exit(1)
85)
86)
87)     excel_file_path = sys.argv[1]
88)
89)
90)     if not os.path.exists(excel_file_path):
91)         print(f"Error: File not found. No file named '{excel_file_path}'
   exists in this directory.")
92)         sys.exit(1)
93)
94)
95)     generate_verilog_from_excel(excel_file_path)
```

3) fsm_core_template.v

```
4)
5)  module fsm_core(
6)      input clk,
7)      input clk_enable,
8)      input reset,
9)      input [1:0] x_in,
10)     output [3:0] z_out,
11)     output [3:0] current_state_debug
12)   );
13)
14)   // State Definitions
15)   parameter S0 = 4'd0;
16)   parameter S1 = 4'd1;
17)   parameter S2 = 4'd2;
18)   parameter S3 = 4'd3;
19)   parameter S4 = 4'd4;
20)   parameter S5 = 4'd5;
21)   parameter S6 = 4'd6;
22)   parameter S7 = 4'd7;
23)   parameter S8 = 4'd8;
24)   parameter S9 = 4'd9;
25)   parameter S10 = 4'd10;
26)   parameter S11 = 4'd11;
27)   parameter S12 = 4'd12;
```

```
28)  parameter S13 = 4'd13;
29)  parameter S14 = 4'd14;
30)  parameter S15 = 4'd15;
31)
32)  // State Registers
33)  reg [3:0] present_state;
34)  reg [3:0] next_state;
35)  reg [3:0] z;
36)
37)  // State Updation when clock available(the clock signal[clk_enable] will
   be controlled by debugger)
38)  always @(posedge clk or negedge reset) begin
39)    if (reset == 1'b0)
40)      present_state <= S0;
41)    else if (clk_enable)
42)      present_state <= next_state;
43)  end
44)
45)  // The whole FSM in form of case blocks and if-else
46)  always @(*) begin
47)    // Safe Defaults
48)    next_state = S0;
49)    z = 4'b0000;
50)
51)    case (present_state)
52)
53)        //PYTHON-GEN-START
54)        // This area will be overwritten by the Python script.
55)        // (Example logic below is ignored by the script,
56)        //  but is useful for stand-alone testing)
57)        /*
58)        S0: begin
59)            case(x_in)
60)                2'b00: begin next_state = S1; z = 4'b0001; end
61)                default: begin next_state = S0; z = 4'b0000; end
62)            endcase
63)        end
64)        S1: begin
65)            case(x_in)
66)                2'b00: begin next_state = S0; z = 4'b0010; end
67)                default: begin next_state = S1; z = 4'b0000; end
68)            endcase
69)        end
70)        */
71)        //PYTHON-GEN-END
```

```
72)
73)
74)         // To prevent lock out for unused states
75)         default: begin
76)             next_state = S0;
77)             z = 4'b0000;
78)         end
79)     endcase
80)   end
81)
82)   assign current_state_debug = present_state;
83)   assign z_out = z;
84)
85)endmodule
```

4.3. The Core Mano - Machine Modules (Main Verilog Codes)

The system is a modular, hierarchical design implemented in Verilog. It consists of four main modules:

1. top_level **(The Motherboard):** This module connects all other modules and maps them to the physical CPLD pins.
2. fsm_core **(The FSM):** This is the "Device Under Test." It's a template that can hold any FSM logic. It takes a clk_enable signal from the debugger.
3. debugger **(The Debug Core):** This is the "magic" of the project. It controls *when* the fsm_core is allowed to clock.
4. decoder_7seg **(The Display Driver):** A simple combinational module to translate the 4-bit present_state from the FSM into 7 signals for the external displ ay.

1) top_level.v (The Motherboard)

```
2)
3) module top_level(
4)     // Physical Pin Declarations
5)     // Inputs (from Helium Board)
6)     input clk_1hz,          // Pin 43: Onboard 1Hz Clock
7)     input reset_button,     // Pin 6 (SW3): Global Asynchronous Reset
8)     input mode_switch,      // Pin 4 (SW1): Run=0, Step=1
9)     input [1:0] fsm_input_x,  // Pin 5 (SW2), Pin 8 (SW4)
10)    input step_button,      // Pin 14 (SW8): Single-Step button
11)
12)    // Outputs (to Helium Board & Breadboard)
13)    output [6:0] segments_out, // Pins 16,18,19,34,37,39,40 (To 7-Seg)
14)    output [3:0] fsm_output_z  // Pin 33 (LED8), Pin 31 (LED7)
```

```
15)  );
16)
17)  // Internal Wires
18)
19)  wire fsm_enable_wire;
20)  wire [3:0] state_wire;
21)
22)  // FSM Core
23)  fsm_core FSM_INSTANCE (
24)    .clk(clk_1hz),
25)    .clk_enable(fsm_enable_wire), // <-- connected to debugger's output
26)    .reset(reset_button),
27)    .x_in(fsm_input_x),
28)    .z_out(fsm_output_z),       // --> connected to physical LEDs
29)    .current_state_debug(state_wire) // --> connected to decoder's input
30)  );
31)
32)  // Debugger
33)  debugger DEBUGGER_INSTANCE (
34)    .clk_1hz(clk_1hz),
35)    .reset(reset_button),
36)    .mode_switch_raw(mode_switch),
37)    .step_button_raw(step_button),
38)    .fsm_clk_enable(fsm_enable_wire) // <-- connected to FSM's input
39)  );
40)
41)  // 7-Segment Decoder
42)  decoder_7seg DECODER_INSTANCE (
43)    .bcd_in(state_wire),      // <-- connected to FSM's debug output
44)    .segments_out(segments_out) // --> connected to 7 Seg output
45)  );
46)
47)endmodule
```

2) fsm_core.v (The FSM) (for Jhonson Count er)

```
module fsm_core(
    input clk,
    input clk_enable,
    input reset,
    input [1:0] x_in,
    output [3:0] z_out,
    output [3:0] current_state_debug
  );
```

```
  // State Definitions
  parameter S0 = 4'd0;
  parameter S1 = 4'd1;
  parameter S2 = 4'd2;
  parameter S3 = 4'd3;
  parameter S4 = 4'd4;
  parameter S5 = 4'd5;
  parameter S6 = 4'd6;
  parameter S7 = 4'd7;
  parameter S8 = 4'd8;
  parameter S9 = 4'd9;
  parameter S10 = 4'd10;
  parameter S11 = 4'd11;
  parameter S12 = 4'd12;
  parameter S13 = 4'd13;
  parameter S14 = 4'd14;
  parameter S15 = 4'd15;

  // State Registers
  reg [3:0] present_state;
  reg [3:0] next_state;
  reg [3:0] z;

  // State Updation when clock available(the clock signal[clk_enable] will be
controlled by debugger)
  always @(posedge clk or negedge reset) begin
    if (reset == 1'b0)
      present_state <= S0;
    else if (clk_enable)
      present_state <= next_state;
  end

  // The whole FSM in form of case blocks and if-else
  always @(*) begin
    // Safe Defaults
    next_state = S0;
    z = 4'b0000;

    case (present_state)

        //PYTHON-GEN-START
        S0: begin
            case (x_in)
                2'b00: begin next_state = S0; z = 4'b0000; end
```

```
            2'b01: begin next_state = S1; z = 4'b0000; end
            2'b10: begin next_state = S3; z = 4'b0000; end
            2'b11: begin next_state = S0; z = 4'b0000; end
            default: begin next_state = S0; z = 4'b0000; end
        endcase
    end

    S1: begin
        case (x_in)
            2'b00: begin next_state = S1; z = 4'b0001; end
            2'b01: begin next_state = S8; z = 4'b0001; end
            2'b10: begin next_state = S0; z = 4'b0001; end
            2'b11: begin next_state = S0; z = 4'b0001; end
            default: begin next_state = S0; z = 4'b0000; end
        endcase
    end

    S10: begin
        case (x_in)
            2'b01: begin next_state = S13; z = 4'b1010; end
            default: begin next_state = S0; z = 4'b0000; end
        endcase
    end

    S11: begin
        case (x_in)
            2'b01: begin next_state = S5; z = 4'b1011; end
            default: begin next_state = S0; z = 4'b0000; end
        endcase
    end

    S12: begin
        case (x_in)
            2'b01: begin next_state = S6; z = 4'b1100; end
            default: begin next_state = S0; z = 4'b0000; end
        endcase
    end

    S13: begin
        case (x_in)
            2'b01: begin next_state = S14; z = 4'b1101; end
            default: begin next_state = S0; z = 4'b0000; end
        endcase
    end
```

```
        S14: begin
            case (x_in)
                2'b01: begin next_state = S15; z = 4'b1110; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S15: begin
            case (x_in)
                2'b01: begin next_state = S7; z = 4'b1111; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S2: begin
            case (x_in)
                2'b00: begin next_state = S2; z = 4'b0010; end
                2'b01: begin next_state = S9; z = 4'b0010; end
                2'b10: begin next_state = S1; z = 4'b0010; end
                2'b11: begin next_state = S0; z = 4'b0010; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S3: begin
            case (x_in)
                2'b00: begin next_state = S3; z = 4'b0011; end
                2'b01: begin next_state = S1; z = 4'b0011; end
                2'b10: begin next_state = S2; z = 4'b0011; end
                2'b11: begin next_state = S0; z = 4'b0011; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S4: begin
            case (x_in)
                2'b01: begin next_state = S2; z = 4'b0100; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S5: begin
            case (x_in)
                2'b01: begin next_state = S10; z = 4'b0101; end
                default: begin next_state = S0; z = 4'b0000; end
```

```
            endcase
        end

        S6: begin
            case (x_in)
                2'b01: begin next_state = S11; z = 4'b0110; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S7: begin
            case (x_in)
                2'b01: begin next_state = S3; z = 4'b0111; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S8: begin
            case (x_in)
                2'b01: begin next_state = S4; z = 4'b1000; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        S9: begin
            case (x_in)
                2'b01: begin next_state = S12; z = 4'b1001; end
                default: begin next_state = S0; z = 4'b0000; end
            endcase
        end

        //PYTHON-GEN-END


        // To prevent lock out for unused states
        default: begin
            next_state = S0;
            z = 4'b0000;
        end
    endcase
end

assign current_state_debug = present_state;
assign z_out = z;
```

```
endmodule
```

3) debugger.v (The Debug Core)

```
module debugger(
    input clk_1hz,
    input reset,
    input mode_switch_raw,
    input step_button_raw,
    output reg fsm_clk_enable
);

    wire button_pressed;
    assign button_pressed = ~step_button_raw;

    // Edge Detector
    // This turns a long press into a single 1-cycle pulse.

    reg button_pressed_prev;
    wire single_pulse;

    always @(posedge clk_1hz or negedge reset) begin
        if (reset == 1'b0)
            button_pressed_prev <= 1'b0;
        else
            button_pressed_prev <= button_pressed;
    end

    //Edge Detection
    assign single_pulse = button_pressed & ~button_pressed_prev;

    //Decides whether in Run Mode or Step Mode
    always @(*) begin
        //In Run Mode
        if (mode_switch_raw == 1'b1)
            fsm_clk_enable = 1'b1;
        else // In Step Mode
            fsm_clk_enable = single_pulse;
    end

endmodule
```

Note : the port names are mode_swtich_raw and step_button_raw as before the code was based on a “debouncer” logic which caused a lot of delay as 1 Hz is a very slow frequency. Hence, when we scrapped it, we appended raw with the port names to signify direct signal from the switches.

4) decoder_7seg.v (The Display Driver)

```
//For Common Anode (For Common Cathode everything would be complemented)
module decoder_7seg(
    input [3:0] bcd_in,
    output reg [6:0] segments_out
  );

  always @(*) begin
    case (bcd_in)
      //           abcdefg
      4'd0: segments_out = 7'b0000001; // 0
      4'd1: segments_out = 7'b1001111; // 1
      4'd2: segments_out = 7'b0010010; // 2
      4'd3: segments_out = 7'b0000110; // 3
      4'd4: segments_out = 7'b1001100; // 4
      4'd5: segments_out = 7'b0100100; // 5
      4'd6: segments_out = 7'b0100000; // 6
      4'd7: segments_out = 7'b0001111; // 7
      4'd8: segments_out = 7'b0000000; // 8
      4'd9: segments_out = 7'b0000100; // 9
      4'd10: segments_out = 7'b0001000; // A
      4'd11: segments_out = 7'b1100000; // b
      4'd12: segments_out = 7'b0110001; // C
      4'd13: segments_out = 7'b1000010; // d
      4'd14: segments_out = 7'b0110000; // E
      4'd15: segments_out = 7'b0111000; // F
      default: segments_out = 7'b1111111; // Off
    endcase
  end

endmodule
```

4.4) Pin Diagram (which was used by us)

Function	Verilog Name	Helium Label	CPLD Pin
Clock	clk_1hz	Onboard Clock	43
Reset	reset_button_raw	SW5	9
Mode	mode_switch_raw	SW1	4
FSM Input [0]	fsm_input_x_raw_0	SW2	5
FSM Input [1]	fsm_input_x_raw_1	SW3	6
Step Button	step_button_raw	SW4	8
FSM Output [0]	fsm_output_z[0]	LED1	24
FSM Output [1]	fsm_output_z[1]	LED2	25
FSM Output [2]	fsm_output_z[2]	LED3	26
FSM Output [3]	fsm_output_z[3]	LED4	27
7-Seg 'a'	segments_out[0]	External I/O	16
7-Seg 'b'	segments_out[1]	External I/O	18
7-Seg 'c'	segments_out[2]	External I/O	19
7-Seg 'd'	segments_out[3]	External I/O	34
7-Seg 'e'	segments_out[4]	External I/O	37
7-Seg 'f'	segments_out[5]	External I/O	39
7-Seg 'g'	segments_out[6]	External I/O	40

5. Validation

The project was validated using two methods: software simulation and hardware demonstration.

For validation we use a simple demo as a 2 bit counter:

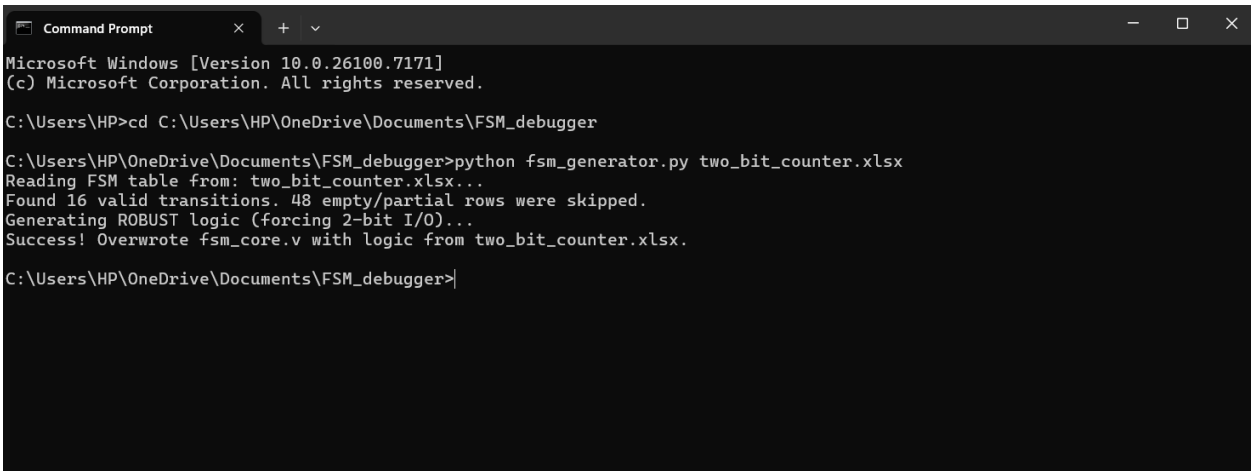
two_bit_counter.xlsx

(Input: 00 (remains same), 01(next state[mod4]), 10(previous state[mode4]), 11(revert to S0))

	A	B	C	D
1	**Present State**	**Input**	**Next State**	**Output**
2	S0	00	S0	0000
3	S0	01	S1	0000
4	S0	10	S3	0000
5	S0	11	S0	0000
6	S1	00	S1	0001
7	S1	01	S2	0001
8	S1	10	S0	0001
9	S1	11	S0	0001
10	S2	00	S2	0010
11	S2	01	S3	0010
12	S2	10	S1	0010
13	S2	11	S0	0010
14	S3	00	S3	0011
15	S3	01	S0	0011
16	S3	10	S2	0011
17	S3	11	S0	0011
18	S4	00		

(This table represents a moore FSM here)

Afterwards we generate the fsm_core through cmd:



(the file path is where the excel sheet, python program and all stuff(mano modules) are present)

5.1. Software Validation (Testbench)

To prove the logic, a testbench module (tb_top_level.v) was written to simulate the user's actions.

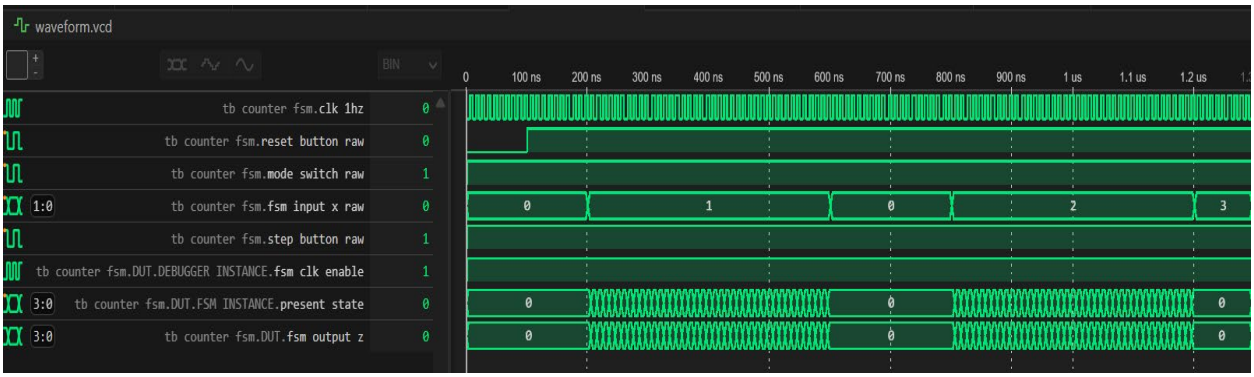
Testbench Code (tb_top_level.v):

```
`timescale 1ns / 1ps

module tb_counter_fsm;

    // 1. Inputs
    reg clk_1hz;
    reg reset_button_raw;
    reg mode_switch_raw;
    reg [1:0] fsm_input_x_raw;
    reg step_button_raw;

    // 2. Outputs
    wire [3:0] fsm_output_z;
    wire [6:0] segments_out;

    top_level DUT (
        .clk_1hz(clk_1hz),
        .reset_button(reset_button_raw),
        .mode_switch(mode_switch_raw),
        .fsm_input_x(fsm_input_x_raw),
```

```
        .step_button(step_button_raw),
        .fsm_output_z(fsm_output_z),
        .segments_out(segments_out)
    );

    always begin
        #5 clk_1hz = ~clk_1hz;
    end

    //  The Test Script
    initial begin
        $dumpfile("waveform.vcd");
        $dumpvars(0, tb_counter_fsm);


        clk_1hz = 0;
        reset_button_raw = 1'b1;
        mode_switch_raw  = 1'b1;
        fsm_input_x_raw  = 2'b00;
        step_button_raw  = 1'b1;

        $display("--- Counter FSM Test Start ---");

        //Test 1
        $display("T= %0t: Resetting System...", $time);
        reset_button_raw = 1'b0;
        #100;
        reset_button_raw = 1'b1;
        #100;

        //Test 2
        $display("T= %0t: Testing Count UP (Input 01)...", $time);
        fsm_input_x_raw = 2'b01;
        #400;

        //Test 3
        $display("T= %0t: Testing HOLD (Input 00)...", $time);
        fsm_input_x_raw = 2'b00;
        #200;

        // Test 4
        $display("T= %0t: Testing Count DOWN (Input 10)...", $time);
        fsm_input_x_raw = 2'b10;
        #400;
```

```
        // Test 5
        $display("T= %0t: Testing Soft Reset (Input 11)...", $time);
        fsm_input_x_raw = 2'b11;
        #100;

        $display("--- Test Finished ---");
        $stop;
    end

endmodule
```

Testbench Waveforms:



Note on Simulation Methodology : The software simulation was performed using **Icarus Verilog** , an industry-standard open-source Verilog compiler and simulator. The testbench (tb_counter_fsm.v) was compiled and executed to generate a Value Change Dump (.vcd) file, which records all signal transitions during the test.

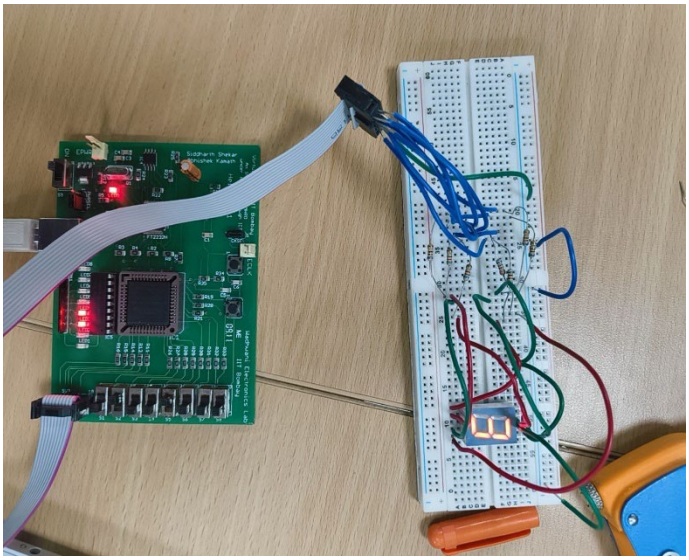
The resulting waveforms were visualized and captured using a **VCD Waveform Viewer** (GTKWave / VS Code WaveTraceextension). This methodology serves as a robust functional validation of the FSM logic, equivalent to the ModelSim -Altera workflow originally intended for the lab environment, which was unavailable on the workstation.

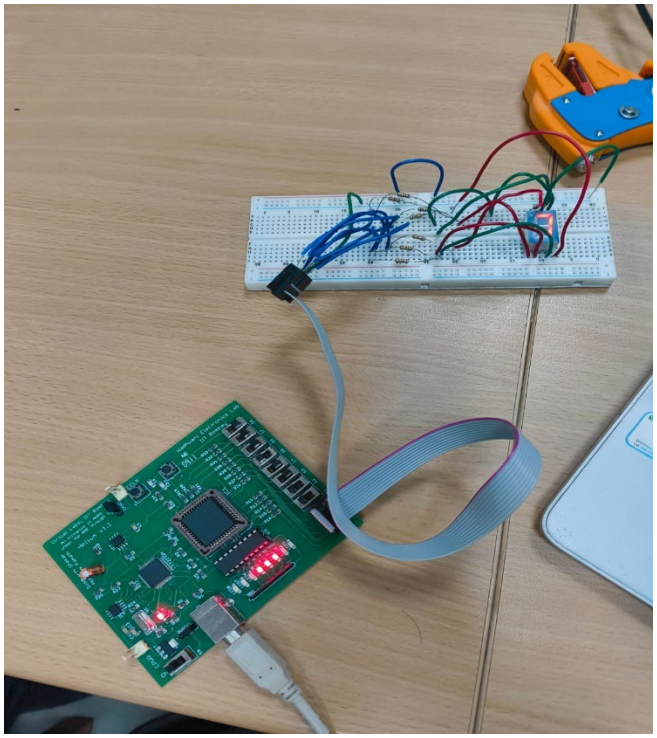
5.2. Hardware Validation (The Demo)

The system was successfully synthesized and downloaded to the Helium CPLD board. The hardware demonstration validated all project goals:

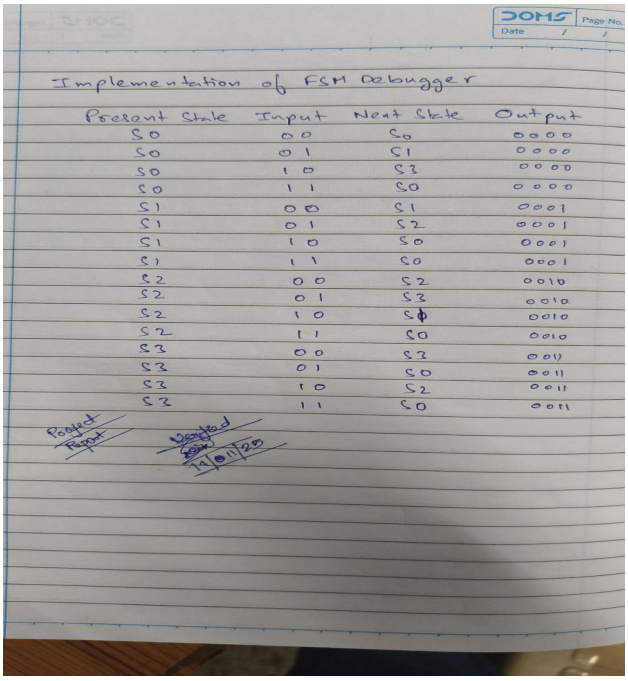
1. **Reset:** Pressing SW5 (Reset) correctly forced the FSM to S0 and the 7-segment display to "0".
2. **Run Mode:** Setting SW1 (Mode) to "Run" (UP) and x_in to "01" (SW2=DOWN, SW3=UP) correctly caused the 7-segment display to count up (0, 1, 2...) once per second.
3. **Step Mode :** Setting SW1 (Mode) to "Step" (DOWN) successfully froze the FSM.
4. **Step Button:** While in Step Mode, pressing SW4 (Step) caused the FSM to advance by

exactly one state per press, proving the edge -detector and clock-gating logic were successful.





(Final Images of the hardware implementation)



Implementation of FSM Debugger

Present State	Input	Next State	Output
S0	00	S0	0000
S0	01	S1	0000
S0	10	S3	0000
S0	11	S0	0000
S1	00	S1	0001
S1	01	S2	0001
S1	10	S0	0001
S1	11	S0	0001
S2	00	S2	0010
S2	01	S3	0010
S2	10	S0	0010
S2	11	S0	0010
S3	00	S3	0011
S3	01	S0	0011
S3	10	S2	0011
S3	11	S0	0011

Verified 14/11/25

(Verified By TA)

6. Conclusion

This project successfully achieved its goal of creating a "meta-machine" on a CPLD. The "CPLD Hardware Debugger" is a robust and reusable tool that transforms FSM design from a "black box" exercise into a transparent, hands-on validation process. The "Step Mode" function, in particular, proved to be an invaluable tool for physically and visually confirming the logic of a state table, one clock cycle at a time.

7. Acknowledgements

We would like to thank the TAs and laboratory staff for their guidance and support with the CPLD hardware and troubleshooting. We also utilized AI assistance (Google Gemini) for debugging Verilog syntax errors, generating initial code templates for the testbench and Python script, and clarifying concepts regarding asynchronous resets and switch debouncing. All final logic design, hardware implementation, and system integration were performed by the team.