Fundamentals of Digital System Design

Finite State Machines

Supun Kuruppu

About Me

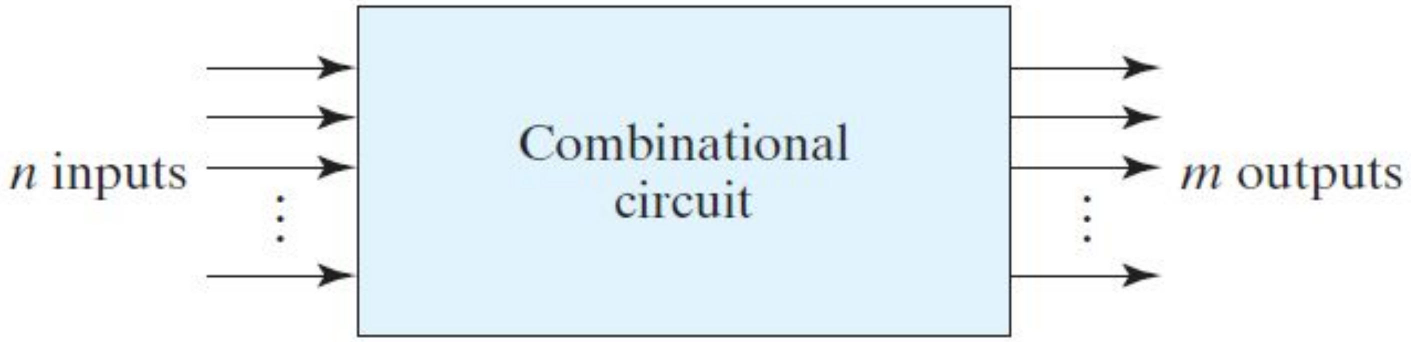
- Engineering Consultant at Analog Inference (Santa Clara, California)
- Associate Electronics Engineer at Paraqum Technologies
- B.Sc. in Electronic & Telecom. Eng. from the University of Moratuwa, Sri Lanka (2024)
- Former Research Affiliate at the University of Sydney
- Former Research Team member at The AI Team
- https://www.linkedin.com/in/supun-dasantha-kuruppu/

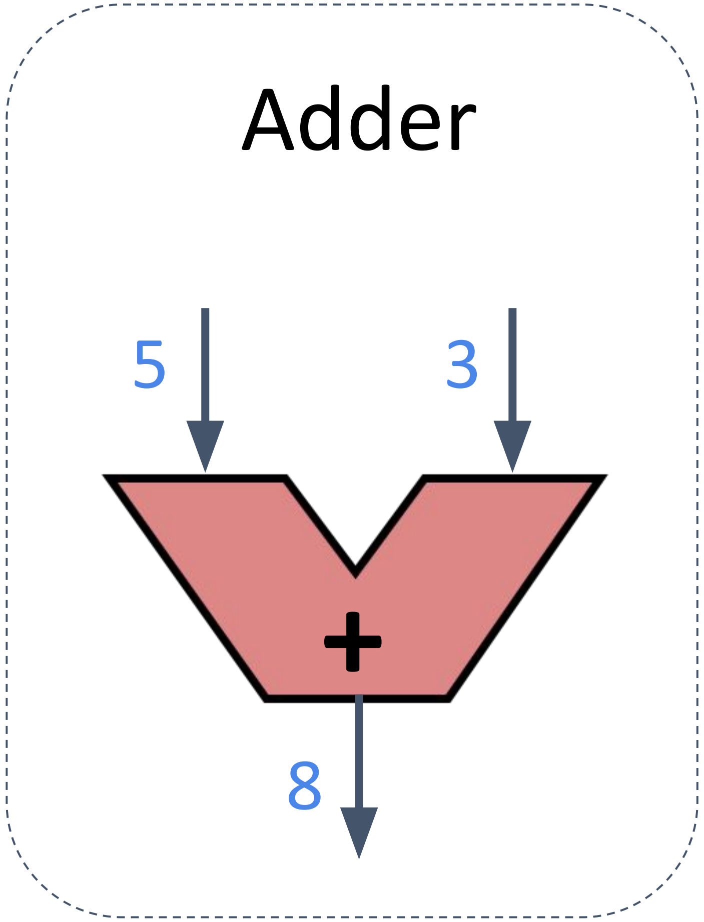
What you'll learn today

- What Finite State Machines (FSMs) are
- How to describe a sequential circuit using FSMs
- How to design sequential circuits using FSMs with SystemVerilog

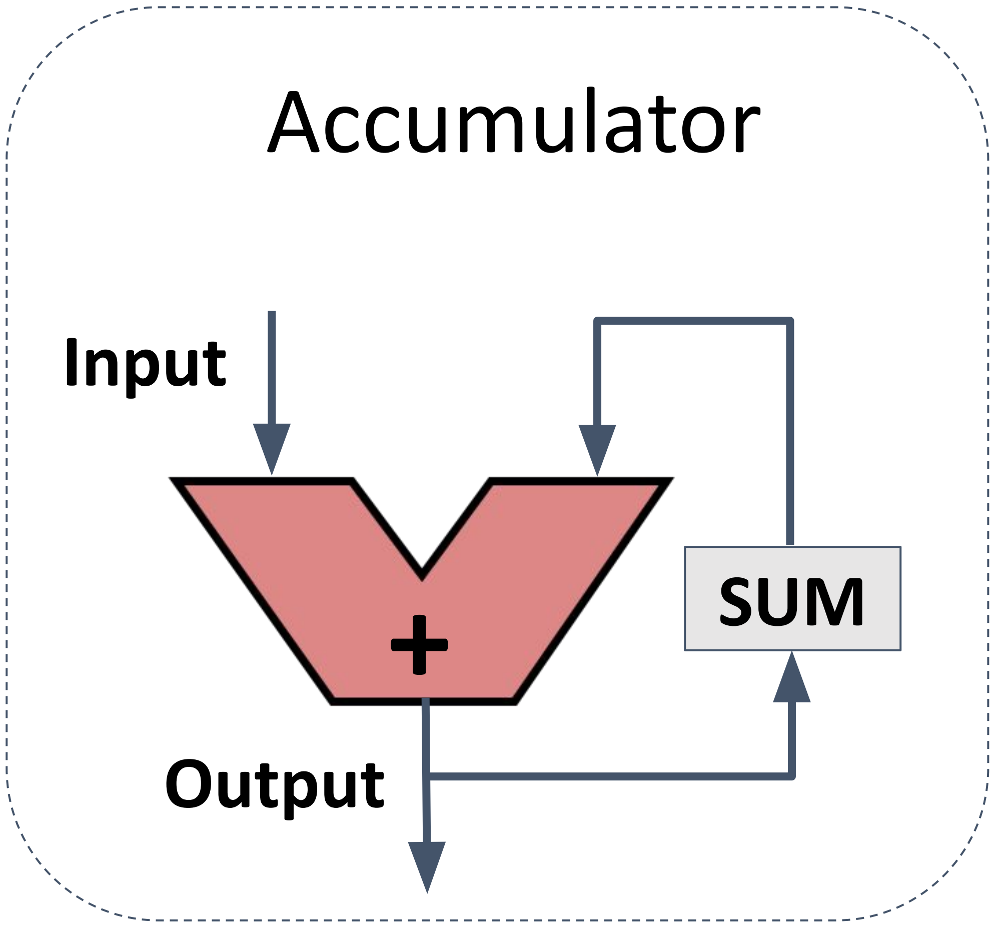
Recap: Combinational Logic

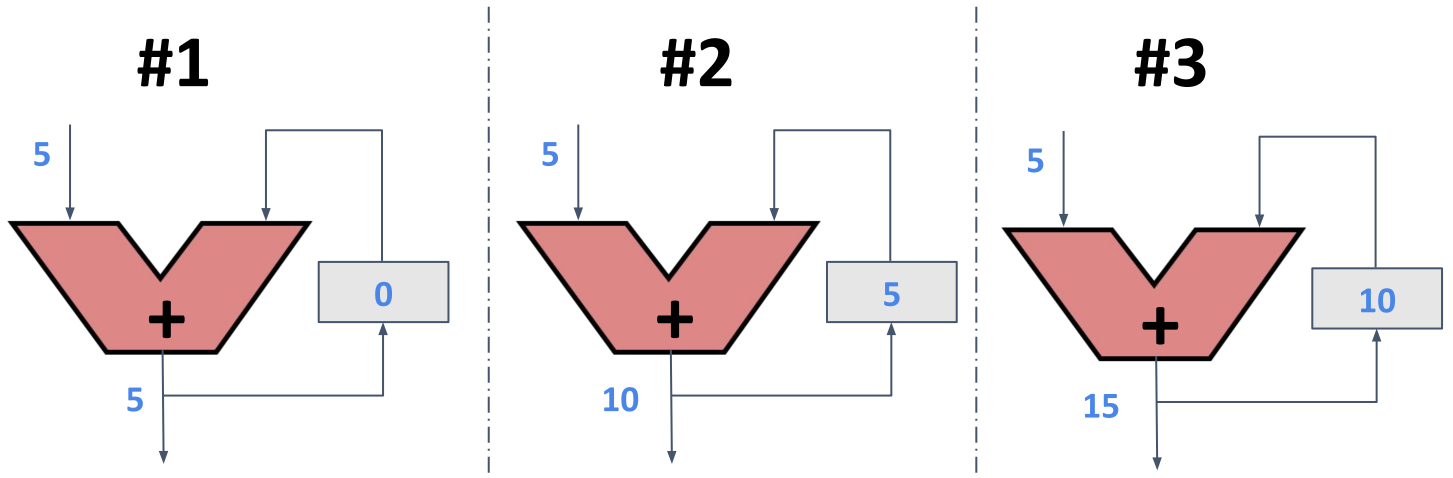
“A type of digital circuit where the output is determined solely by the current inputs





Recap





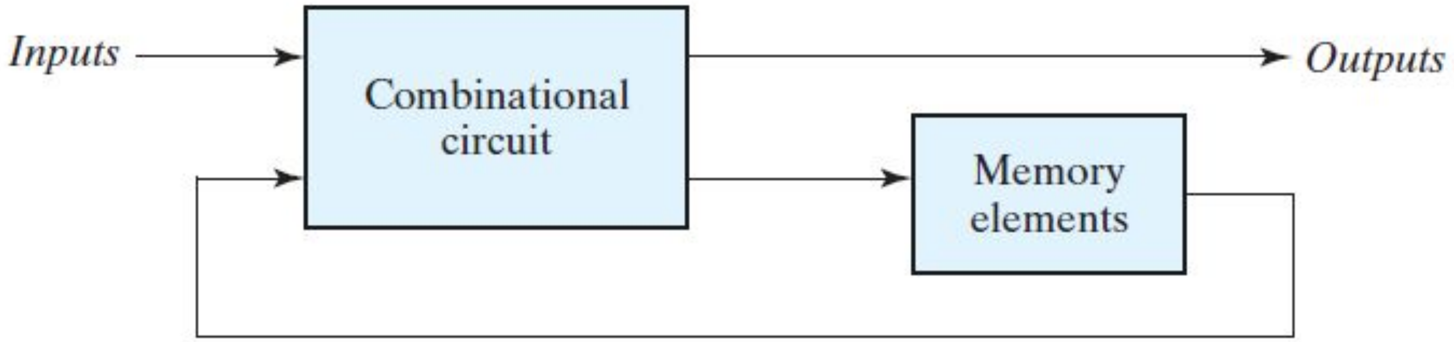
Same Input

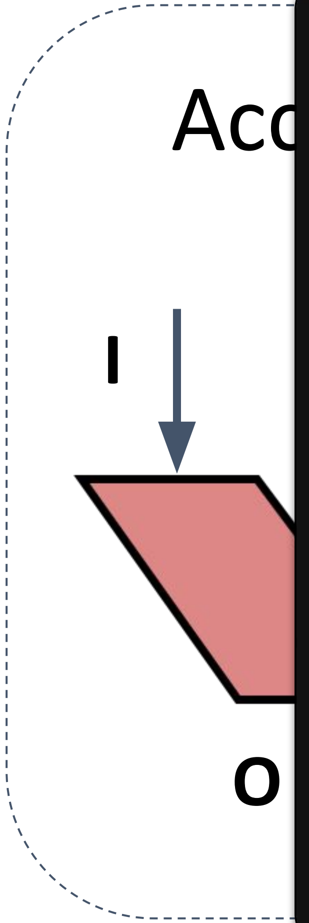
Different Output

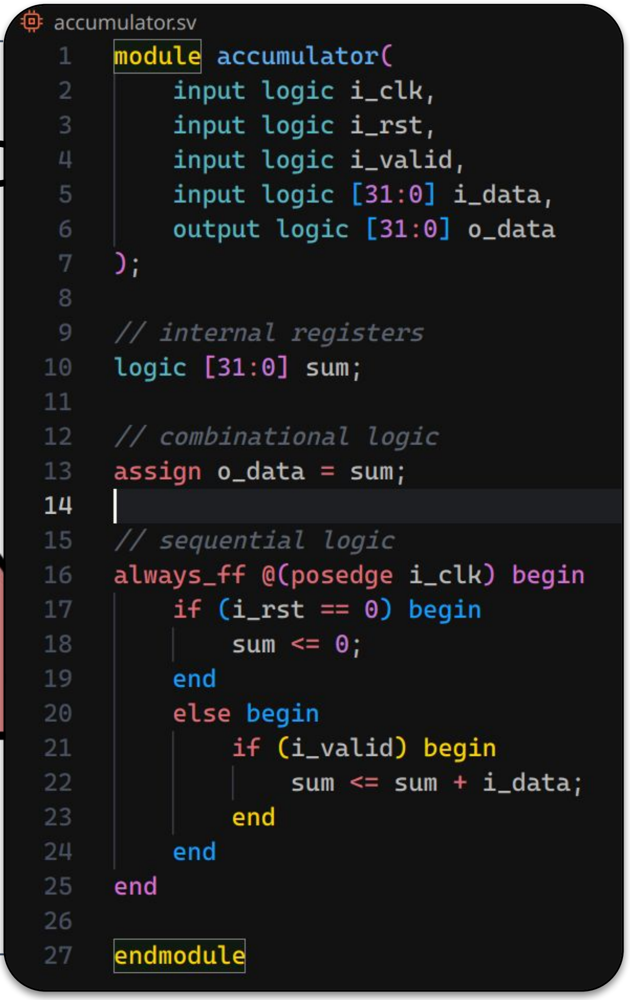
	Input	utput
#1	5	5
#2	5	10
#3	5	15

Recap: Sequential Logic

A type of digital circuit where the output depends not only on the current inputs but also on the history of previous inputs



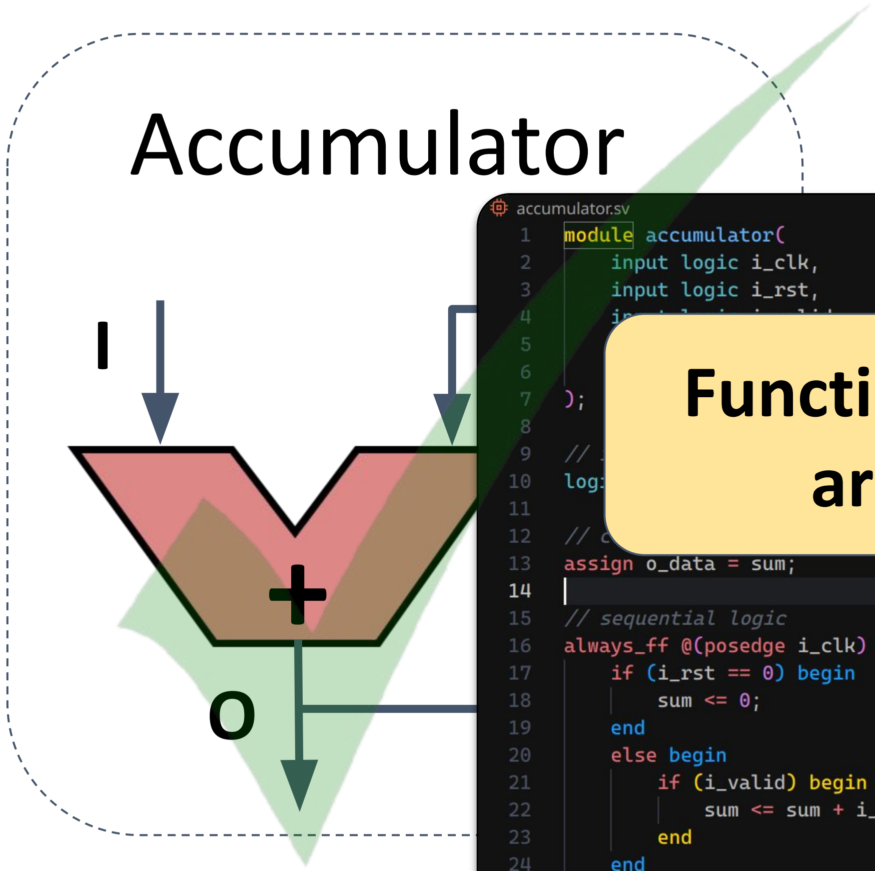


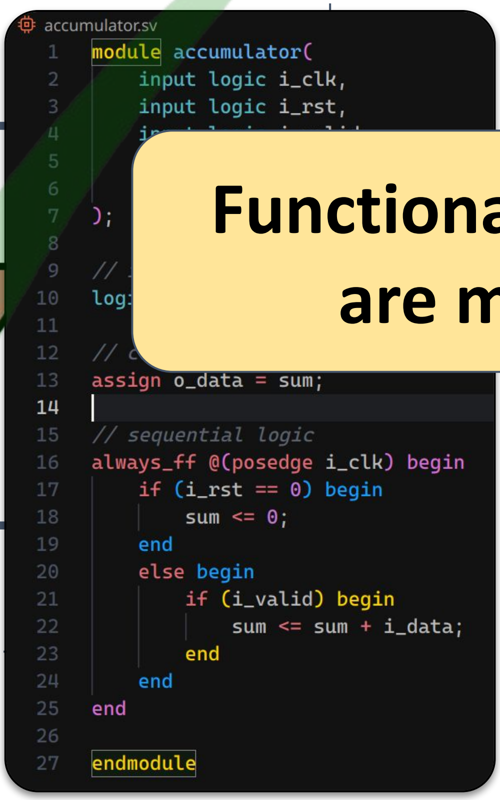

accumulator.sv

```
module accumulator(
    input logic i_clk,
    input logic i_rst,
    input logic i_valid,
    input logic [31:0] i_data,
    output logic [31:0] o_data
);

// internal registers
logic [31:0] sum;

// combinational logic
assign o_data = sum;

// sequential logic
always_ff @(posedge i_clk) begin
    if (i_rst == 0) begin
        sum <= 0;
    end
    else begin
        if (i_valid) begin
            sum <= sum + i_data;
        end
    end
end

endmodule
```

Need for FSMs

Simple Sequential Circuits




accumulator.sv
```
module accumulator(
    input logic i_clk,
    input logic i_rst,
);

// 
log

// 
assign o_data = sum;

// sequential logic
always_ff @(posedge i_clk) begin
    if (i_rst == 0) begin
        sum <= 0;
    end
    else begin
        if (i_valid) begin
            sum <= sum + i_data;
        end
    end
end

endmodule
```

Complex Systems

Central Processing Unit

Graphics Processing Unit

AI Accelerators

Control Systems

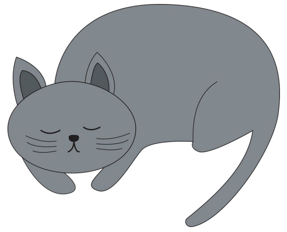
Functionality of complex systems are modelled using FSMs

What is an FSM?

- High-level description of a sequential logic circuit
- You can describe the functionality of any sequential logic circuit using an FSM
  - Typically used to model complex systems

It describes the relationship between the sequential circuit’s inputs, states and outputs

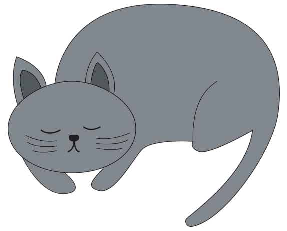
Example: Inputs, States, Outputs



Hungry

Hungry

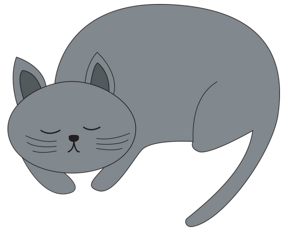
Example: Inputs, States, Outputs



Hungry

Full

Example: Inputs, States, Outputs



Full

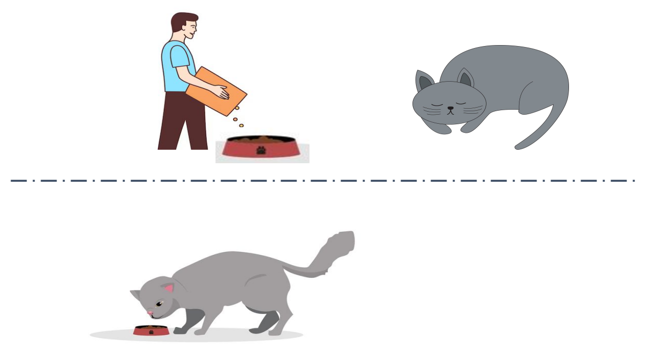
Full

Example: Inputs, States, Outputs

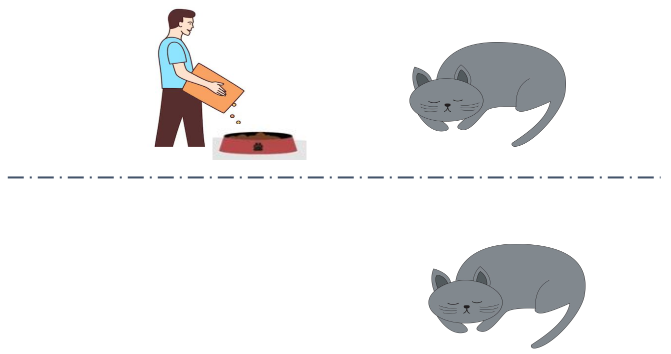
Interaction between the cat owner and the cat

Input	Owner giving food
States	Cat is hungry/full
Output	Cat eating/not eating

Cat is **hungry**

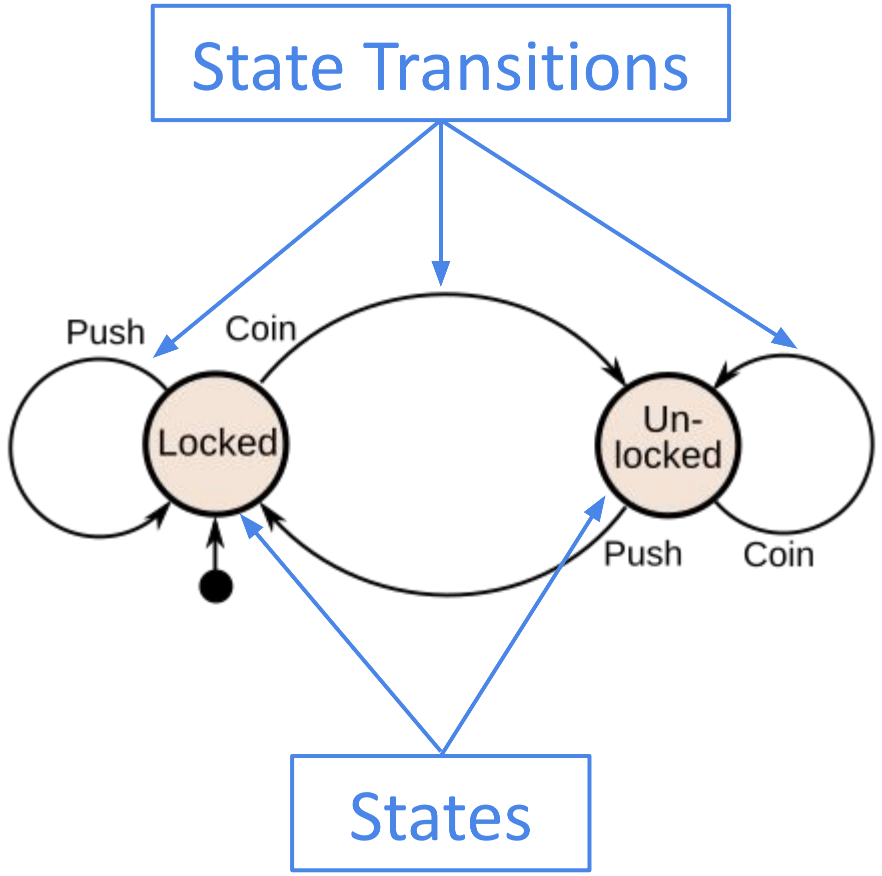


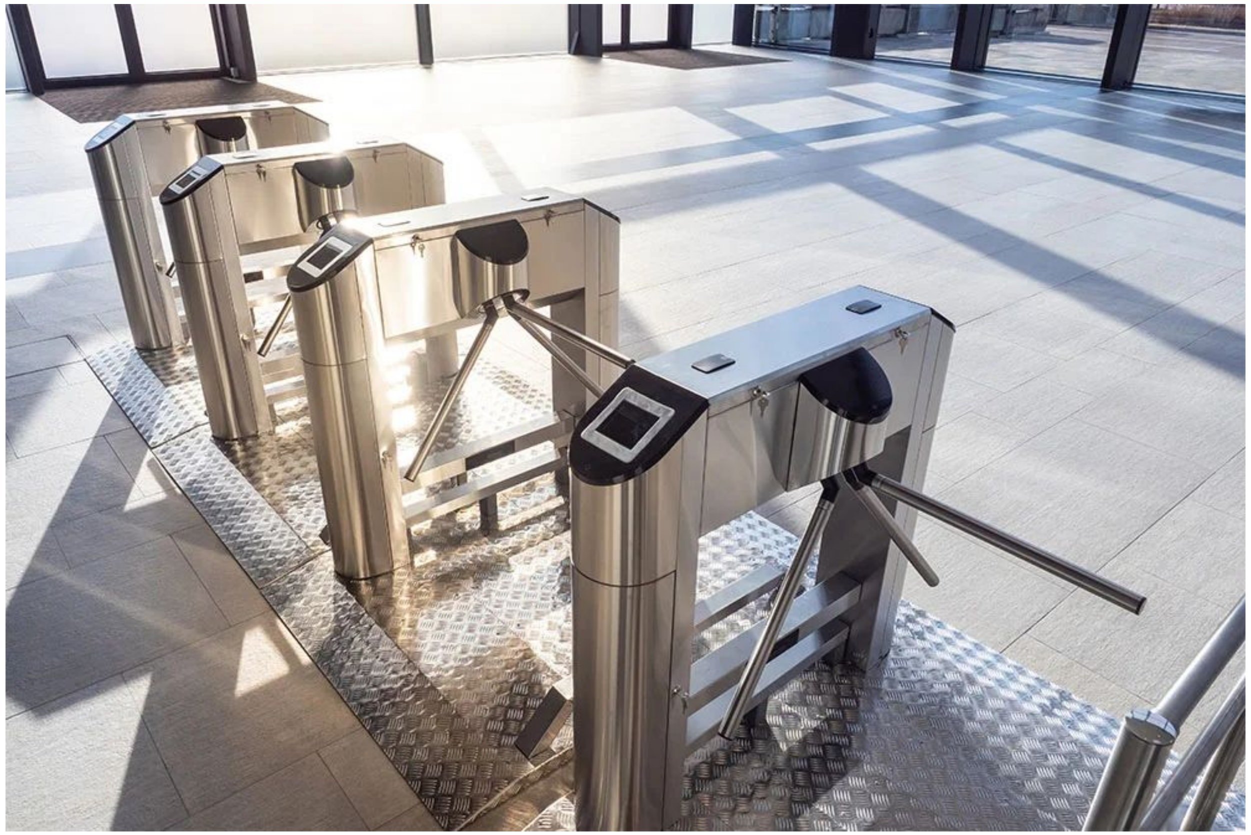
Cat is **full**



State in a system is a collection of additional data that influences the output apart from the input

Example: Turnstile

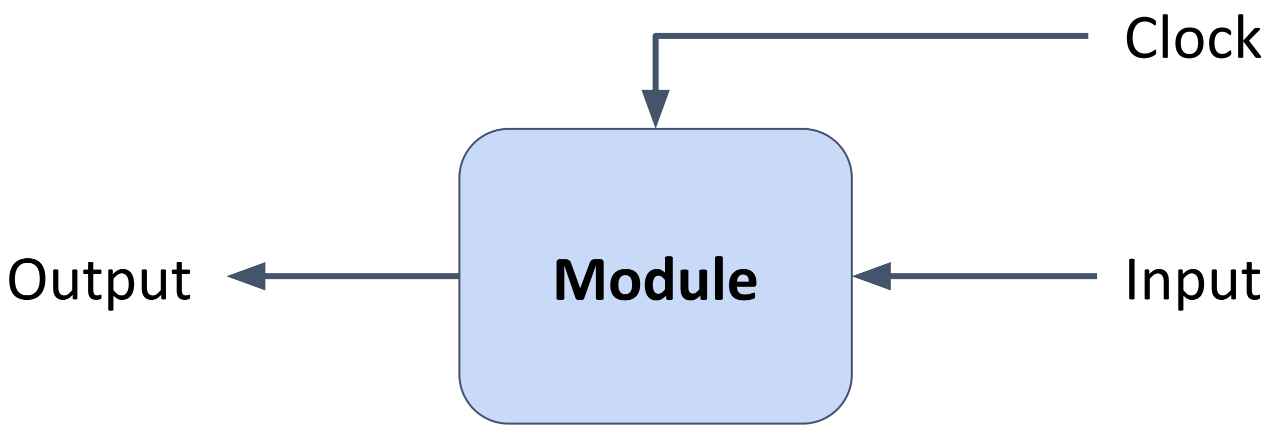




What is a turnstile?
https://youtu.be/m4HPARVUWag?si=Y1WrAE_UdGtStAhA

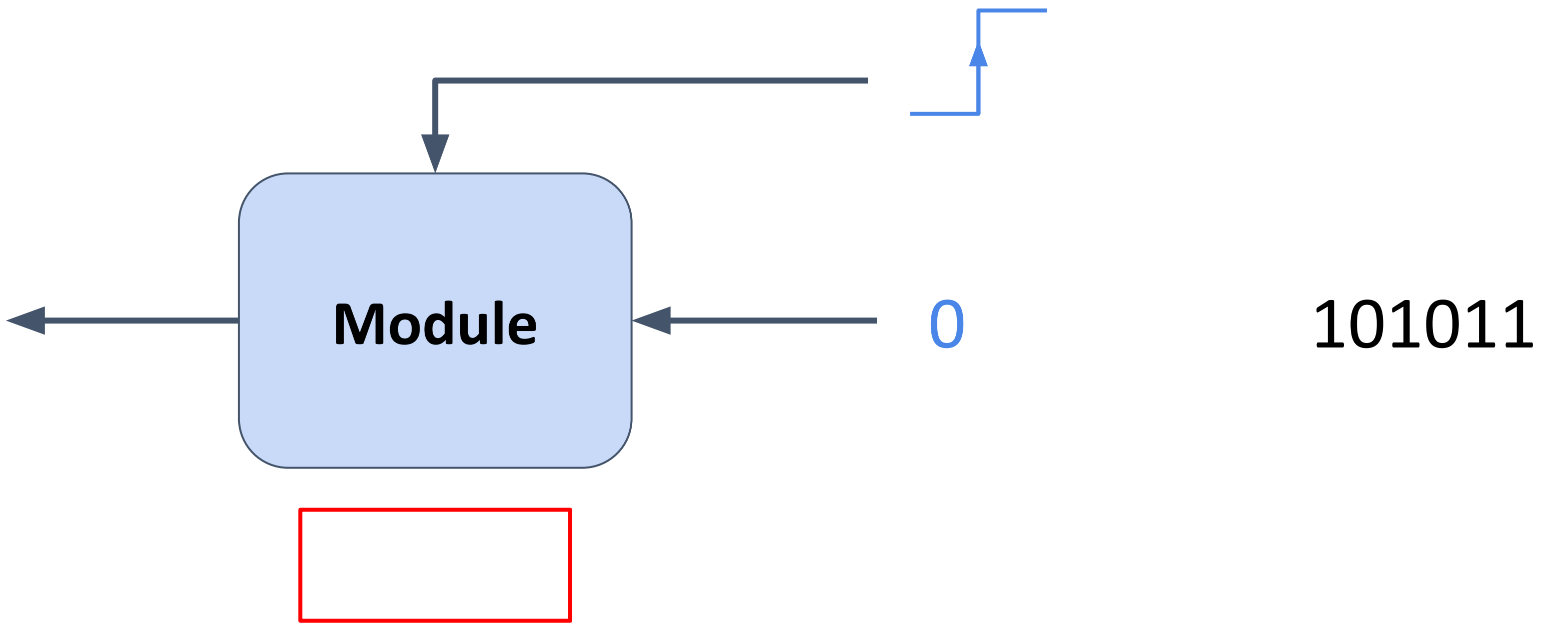
Finite State Machine

Example:
Build a synchronous system that can identify the “101” bit patterns in a series of incoming bits



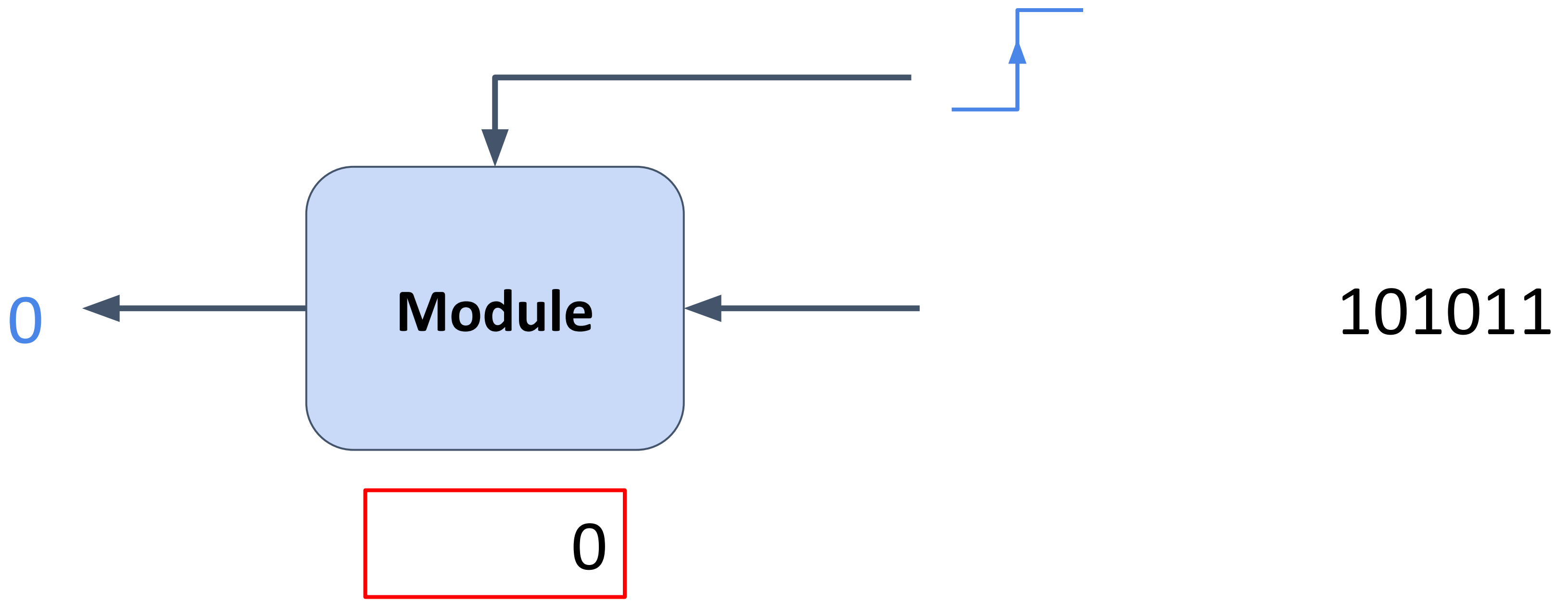
Finite State Machine

- Module demonstration with 0101011 input



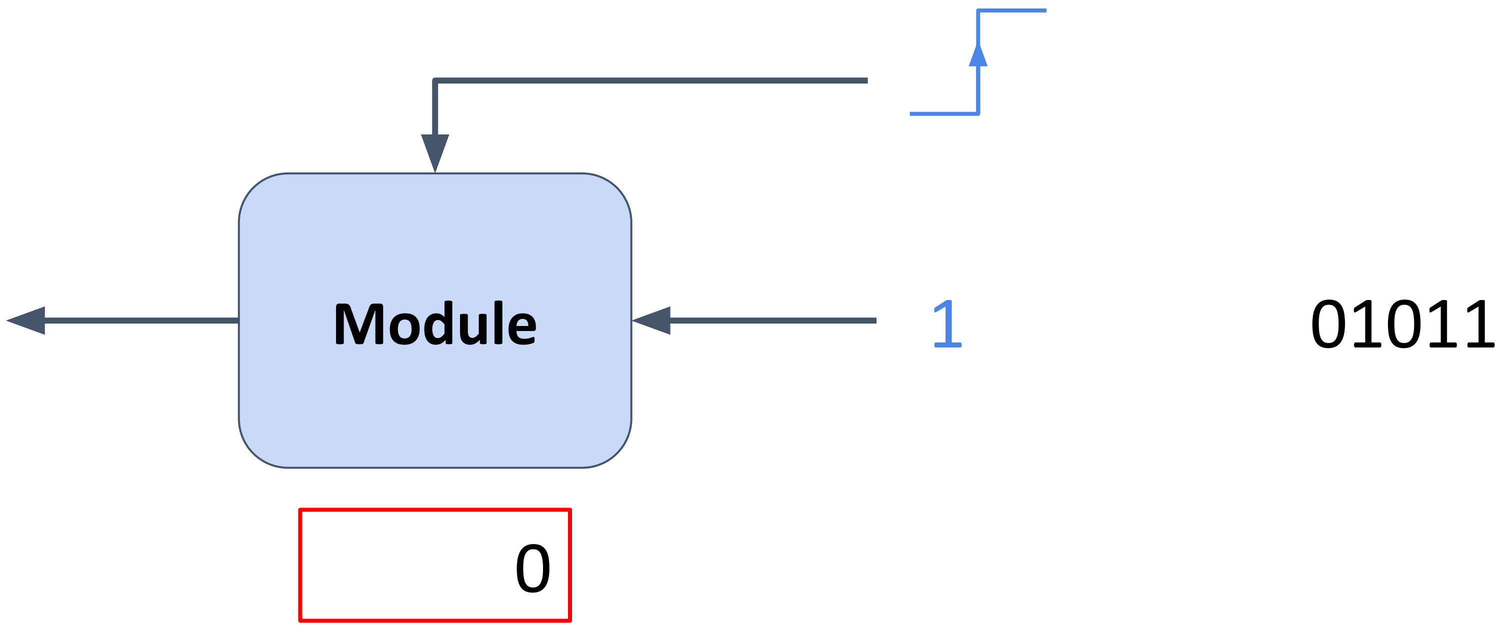
Finite State Machine

- Module demonstration with 0101011 input



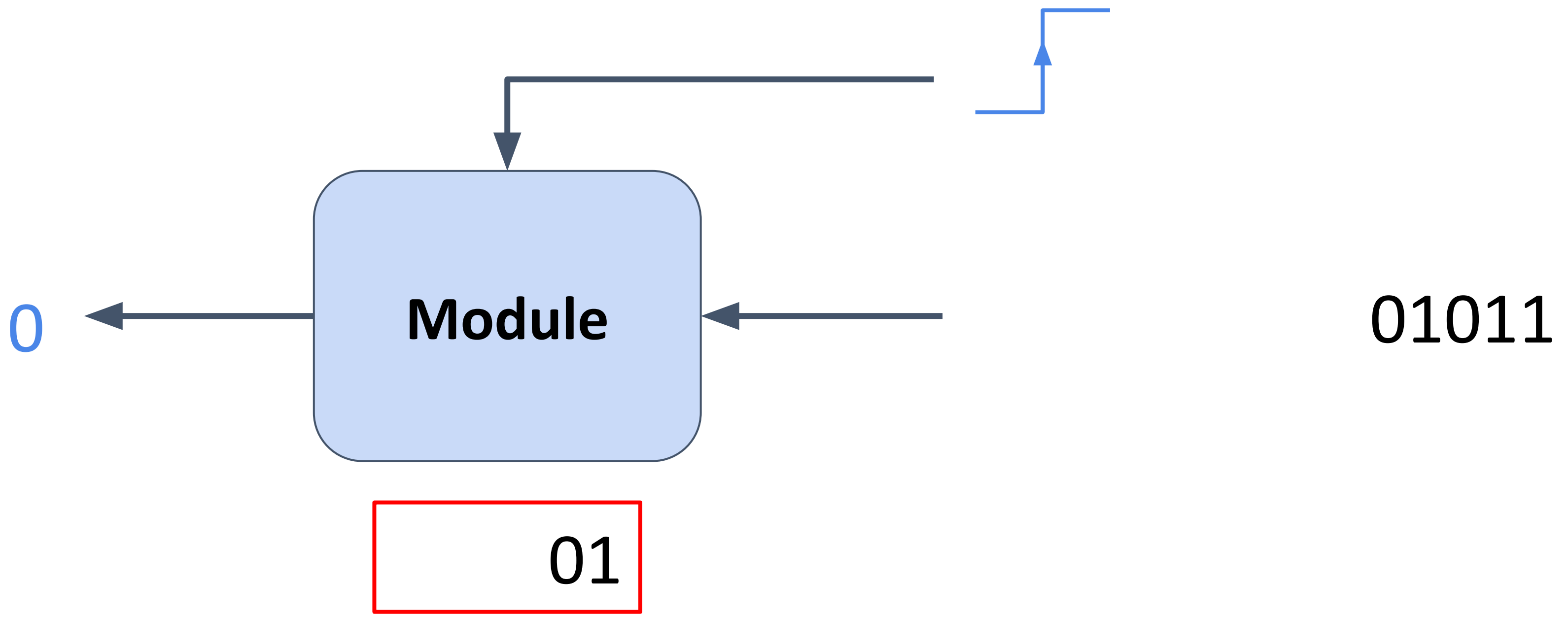
Finite State Machine

- Module demonstration with 0101011 input



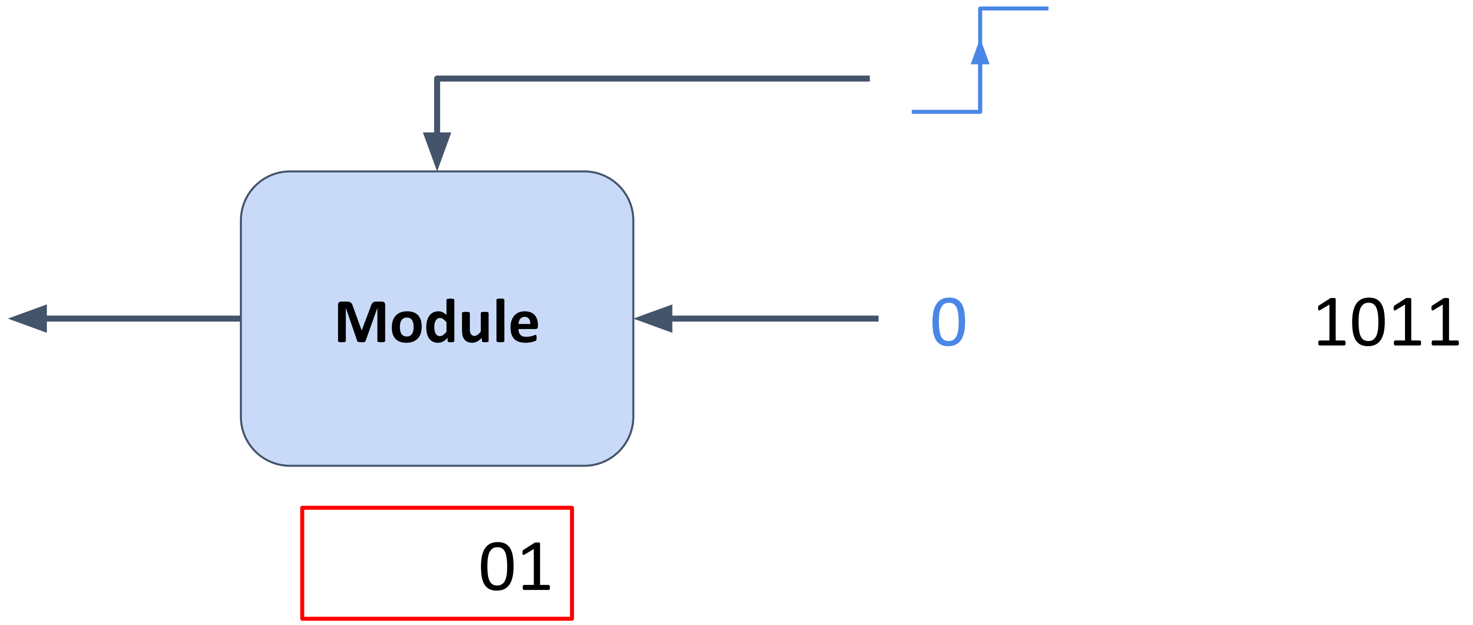
Finite State Machine

- Module demonstration with 0101011 input



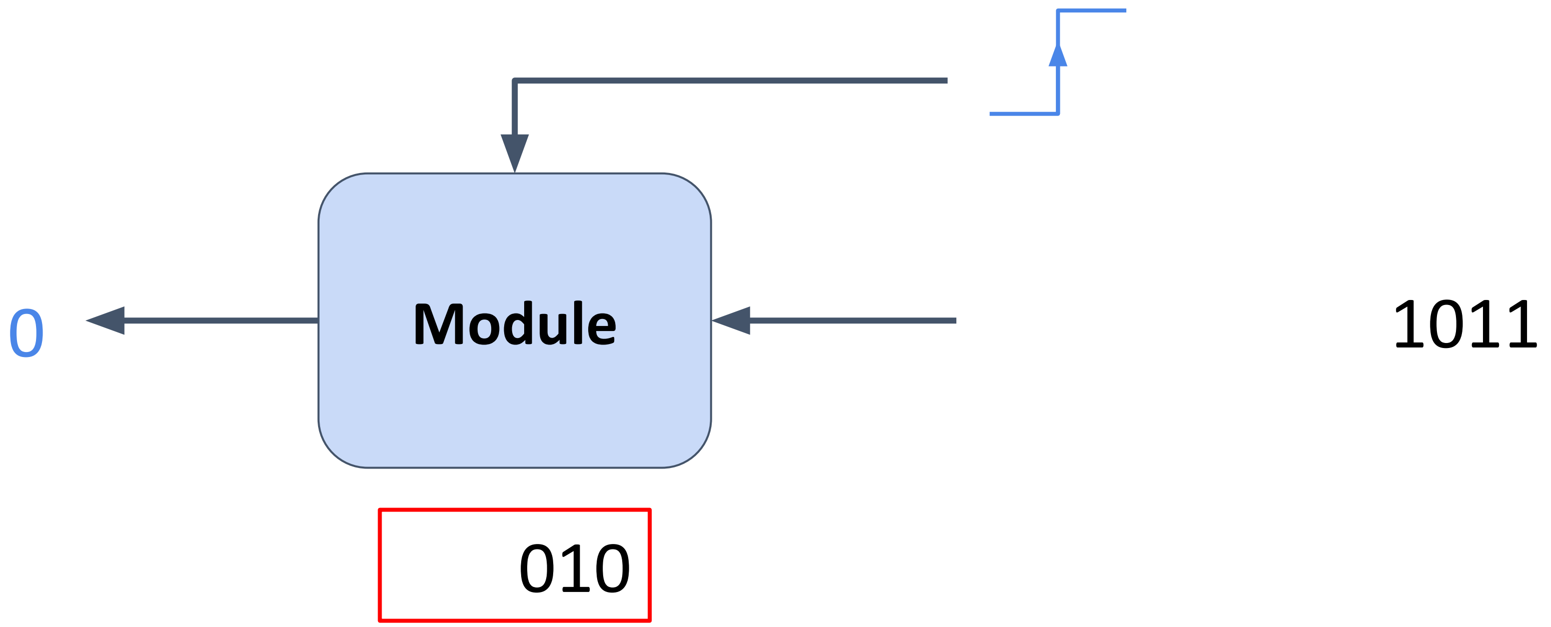
Finite State Machine

- Module demonstration with 0101011 input



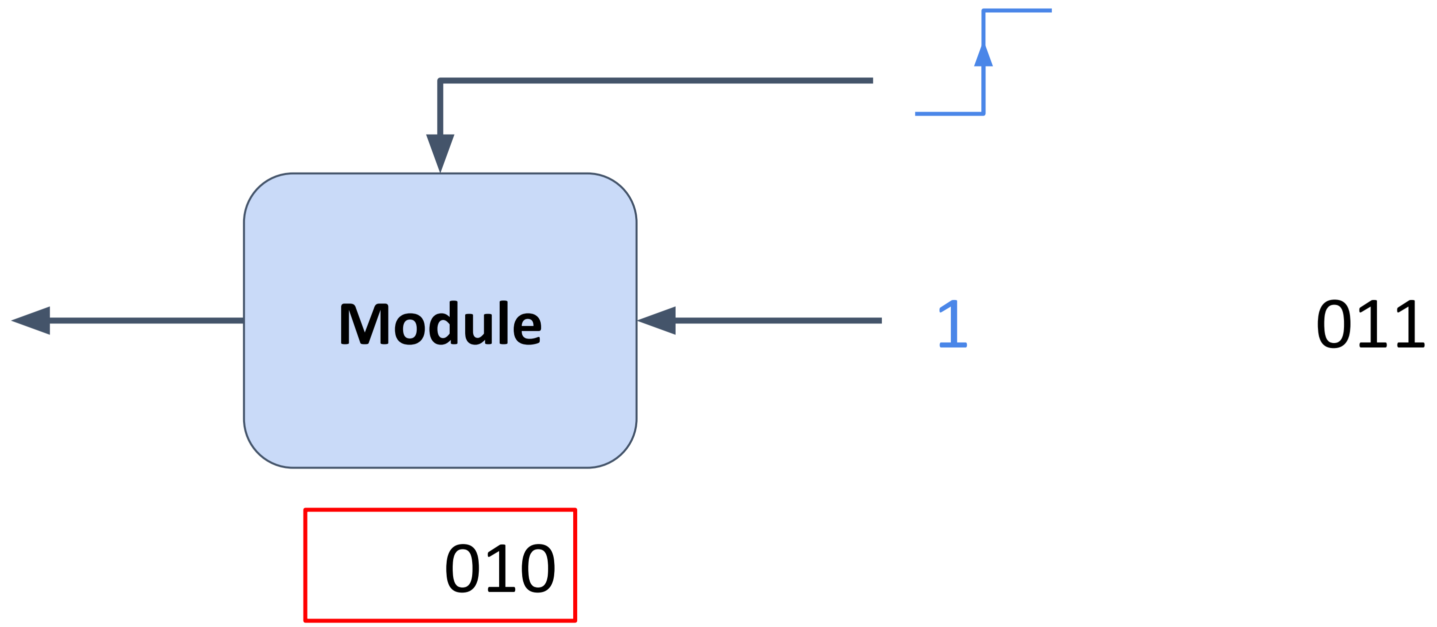
Finite State Machine

- Module demonstration with 0101011 input



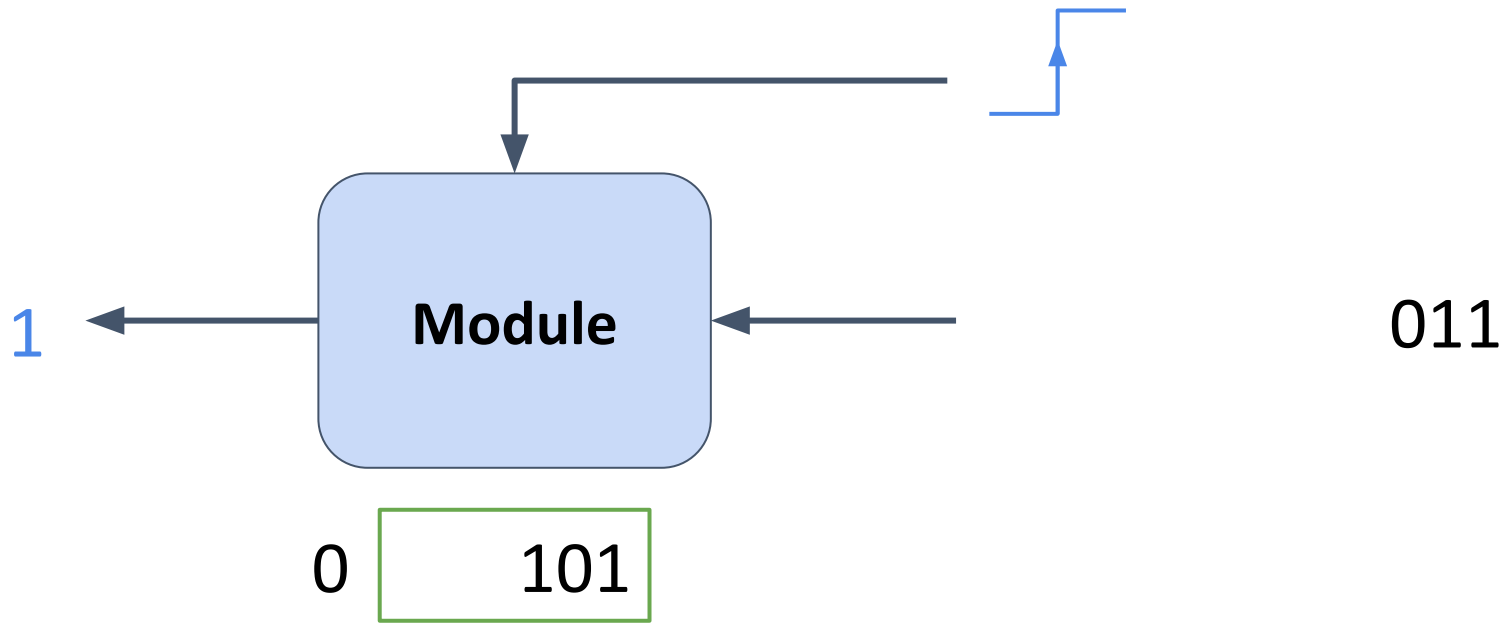
Finite State Machine

- Module demonstration with 0101011 input



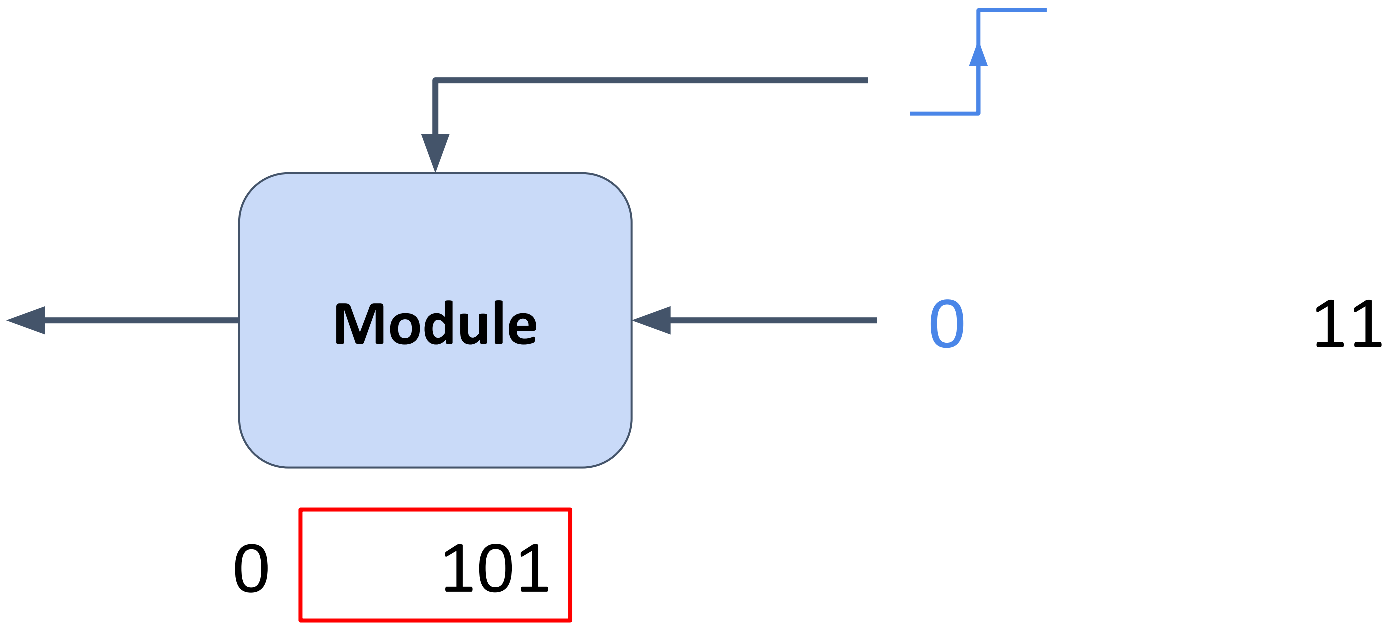
Finite State Machine

- Module demonstration with 0101011 input



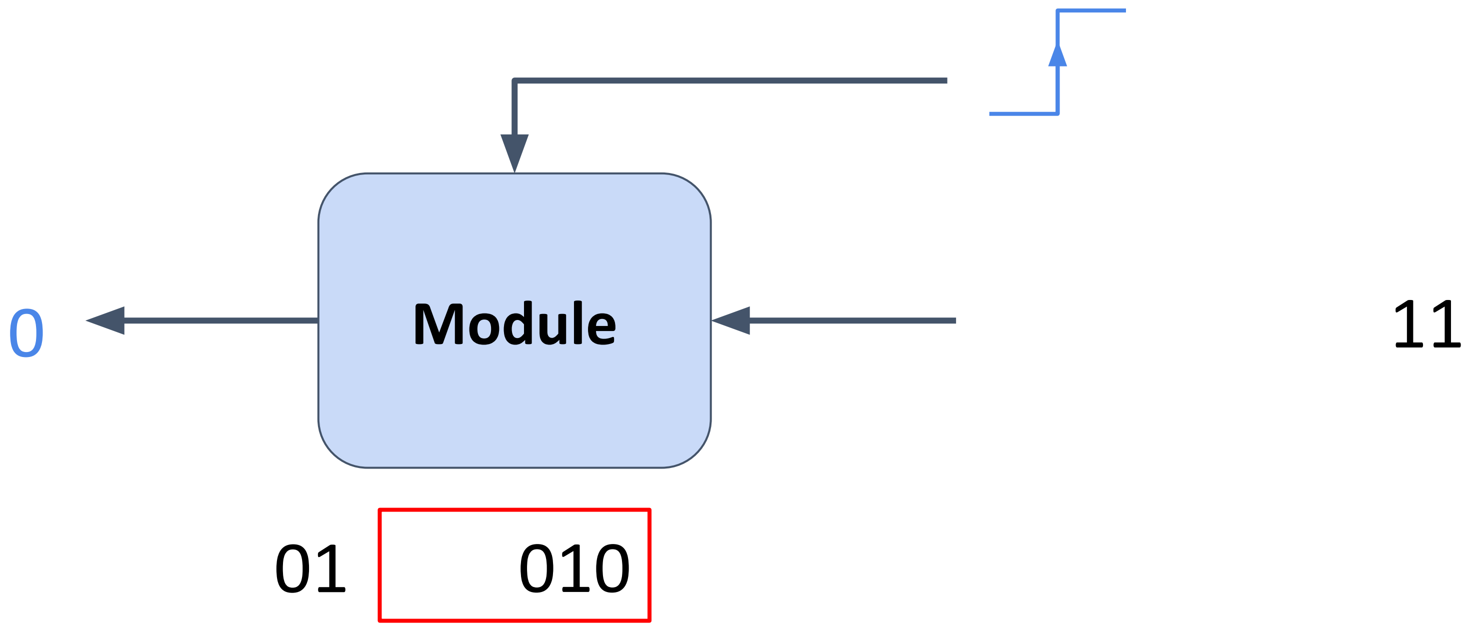
Finite State Machine

- Module demonstration with 0101011 input



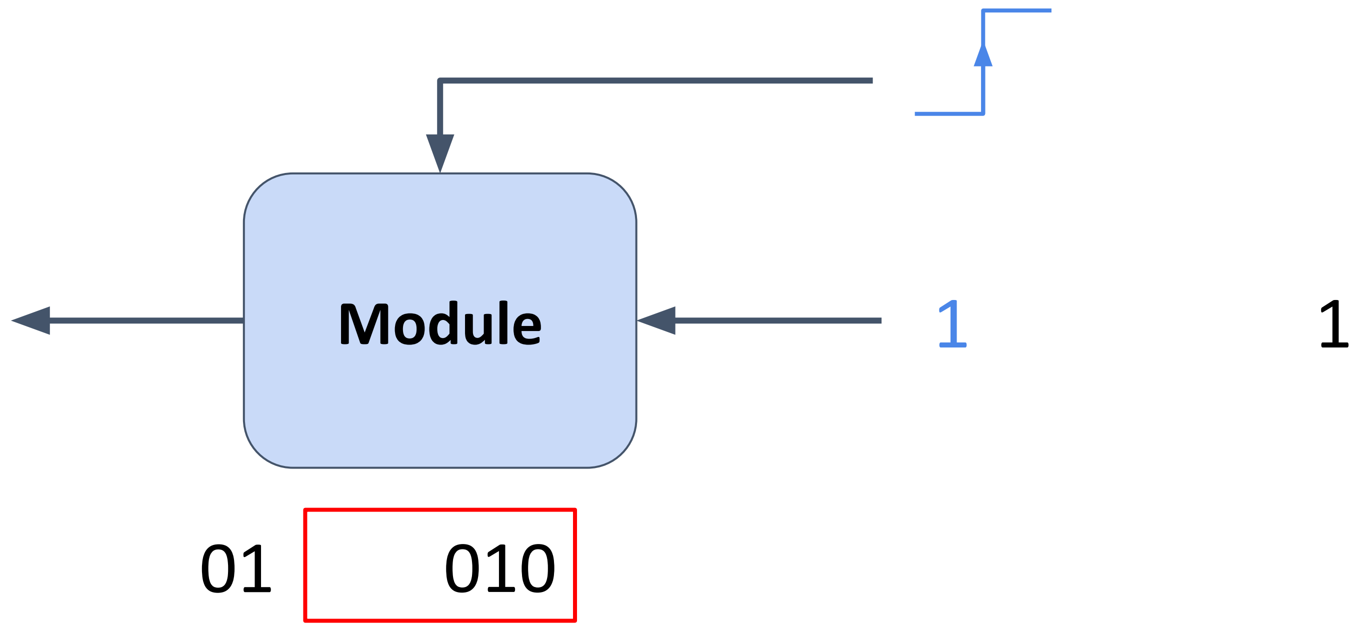
Finite State Machine

- Module demonstration with 0101011 input



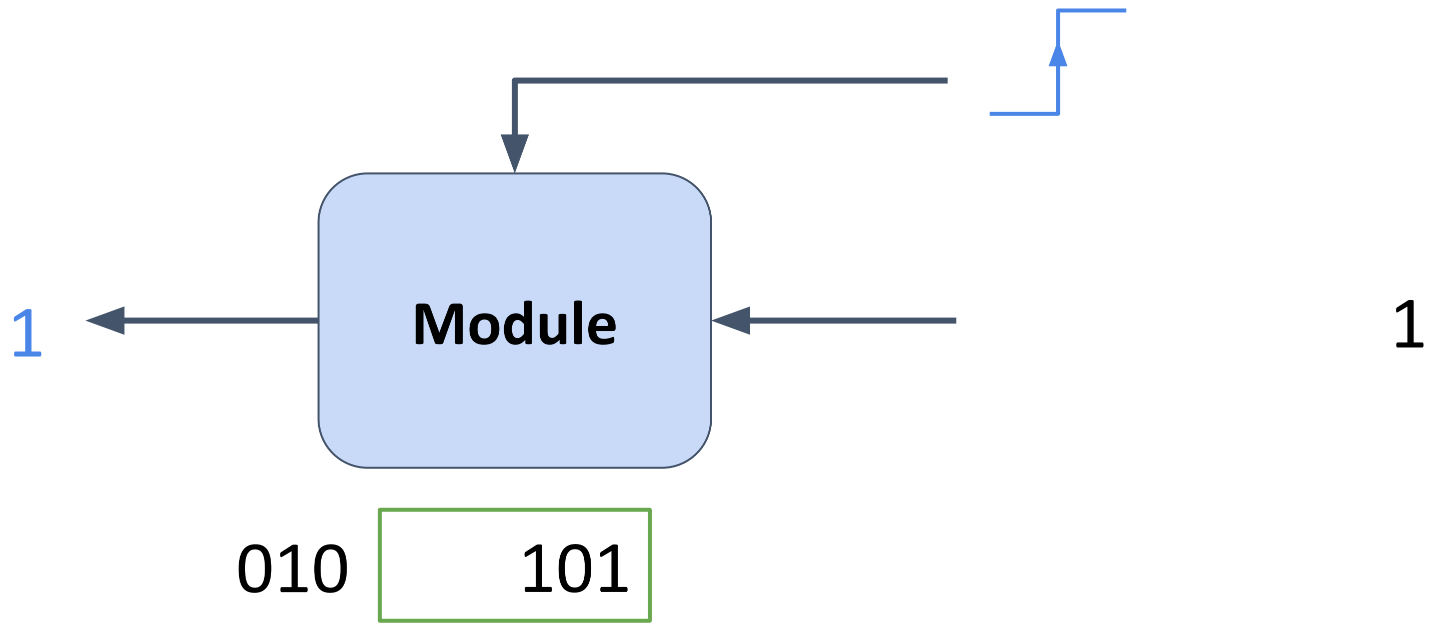
Finite State Machine

- Module demonstration with 0101011 input



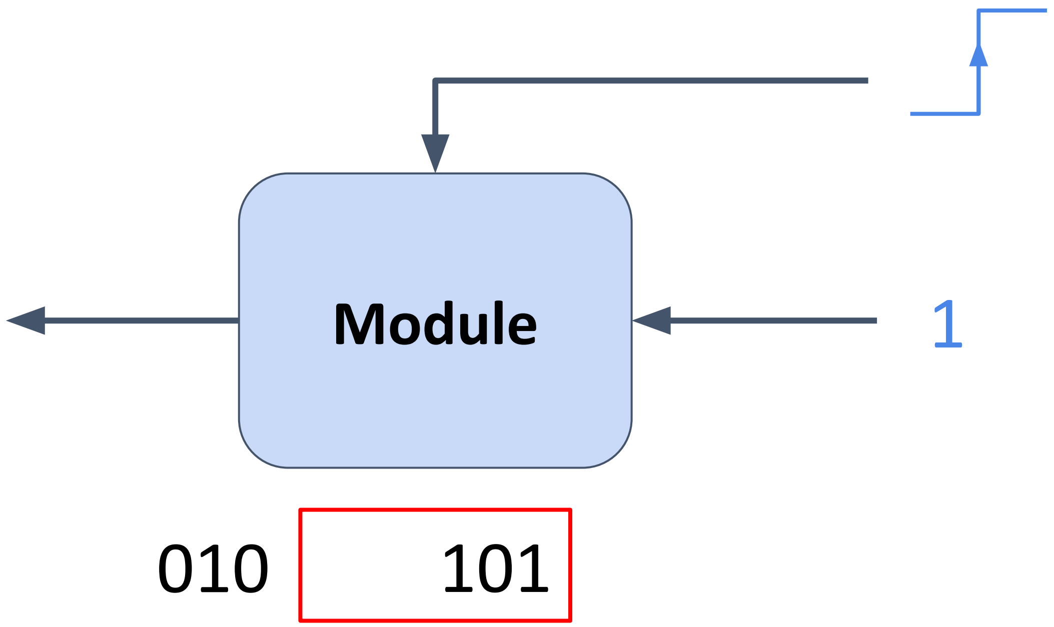
Finite State Machine

- Module demonstration with 0101011 input



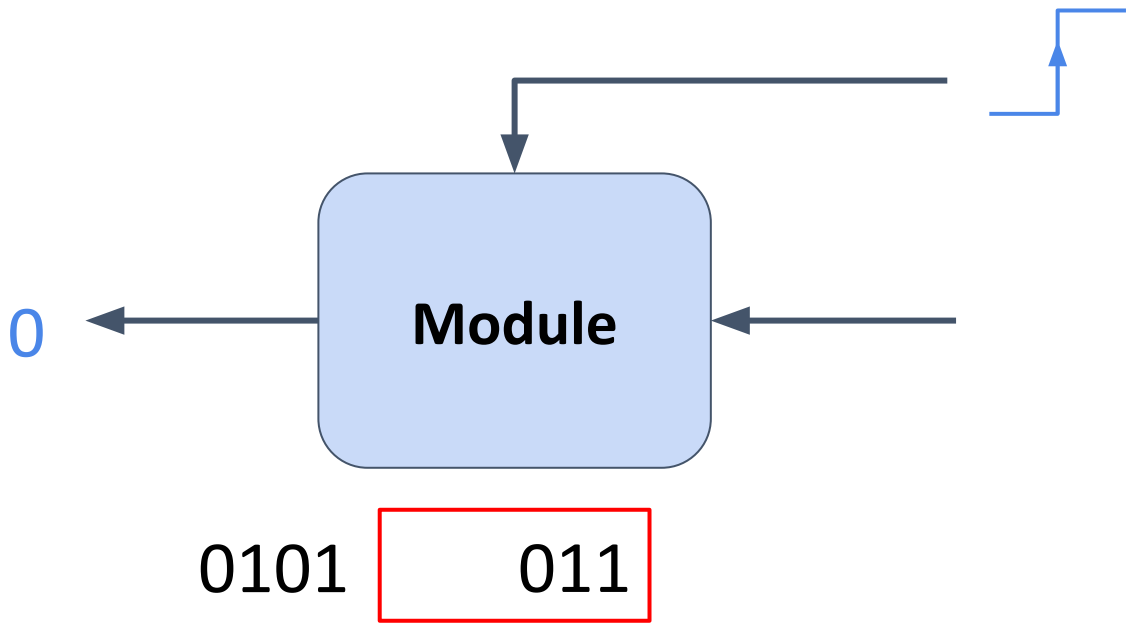
Finite State Machine

- Module demonstration with 0101011 input

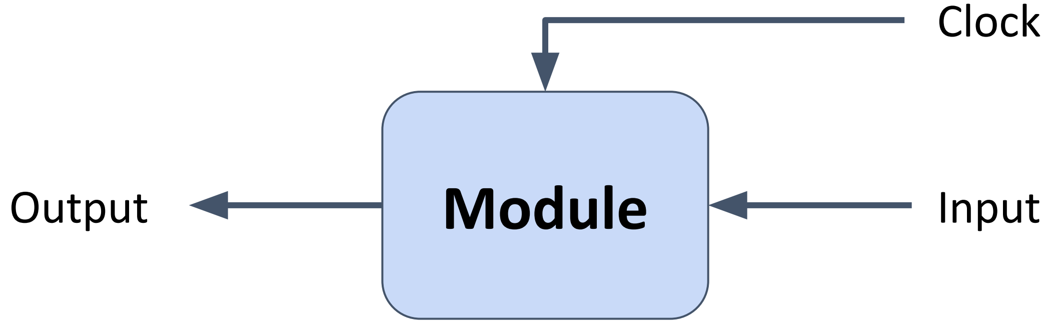


Finite State Machine

- Module demonstration with 0101011 input



Finite State Machine



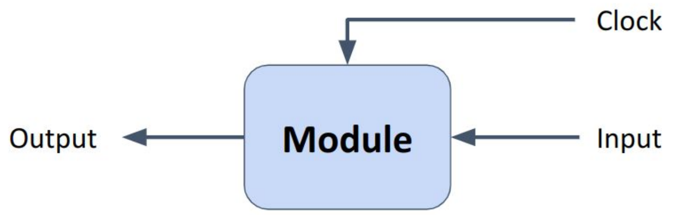
- Finite State Machines can be represented in two ways
  - State Diagrams
  - State Transition Tables

State	Description	Notation
Nothing found	All the next incoming three bits need to be “101” for the “101” pattern	F0
“1” found	Only the next incoming two bits need to be “01” for the “101” pattern	F1
“10” found	Only the next incoming bit needs to be “1” for the “101” pattern	F2
“101” found	The “101” pattern is present	F3

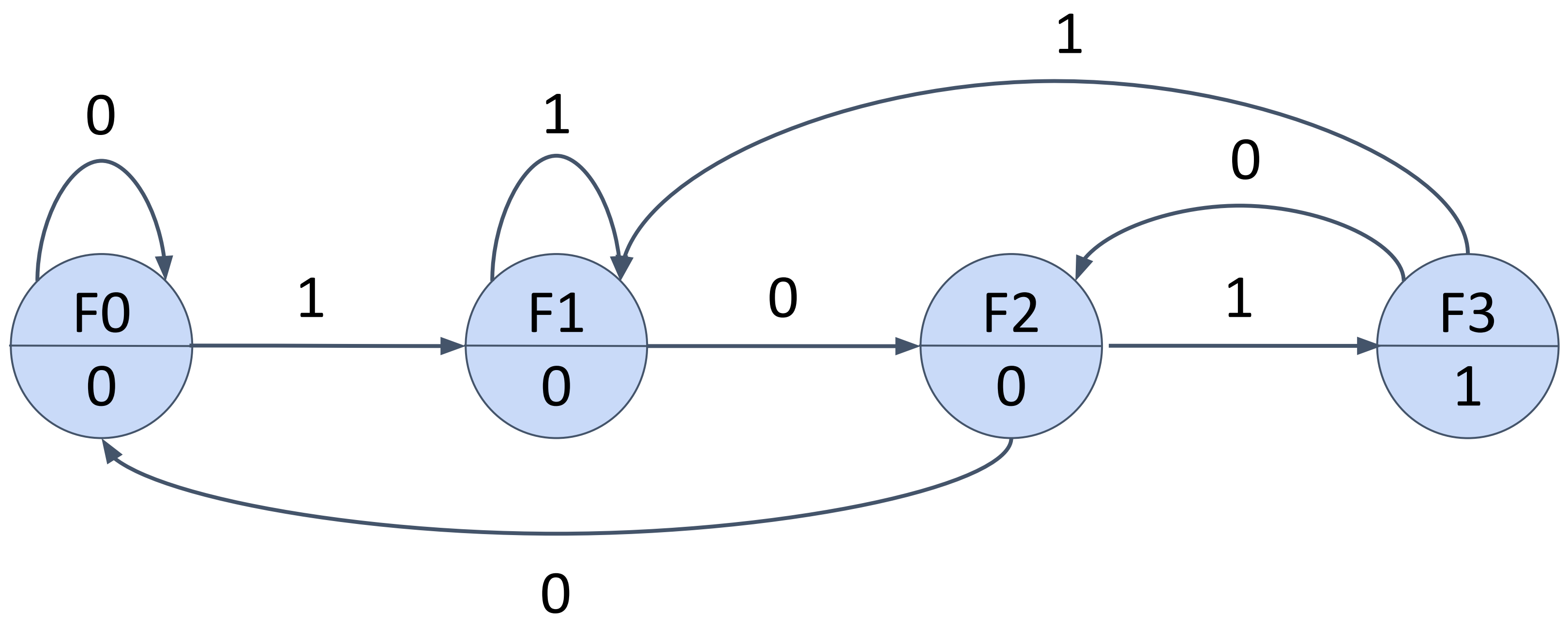
State Diagram

Task:

Identify the “101” bit patterns in a series of incoming bits



State	Description	Notation
Nothing found	All the next incoming three bits need to be “101” for the “101” pattern	F0
“1” found	Only the next incoming two bits need to be “01” for the “101” pattern	F1
“10” found	Only the next incoming bit needs to be “1” for the “101” pattern	F2
“101” found	The “101” pattern is present	F3

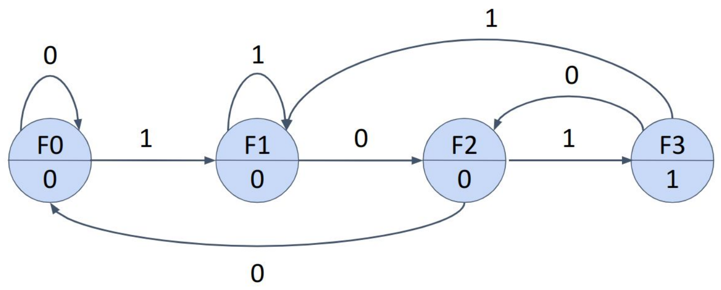


State Transition Table

- We need to represent the states as a bit pattern
- Since there are 4 states, we need 2 bits for the representation

State	Bit Pattern
F0	00
F1	01
F2	10
F3	11

State Transition Table



State	Bit Pattern
F0	00
F1	01

F2	10
F3	11

Current State	Next State		Output
	(Input) 0	(Input) 1	
00	00	01	0
01	10	01	0
10	00	11	0
11	10	01	1

Moore State Machine

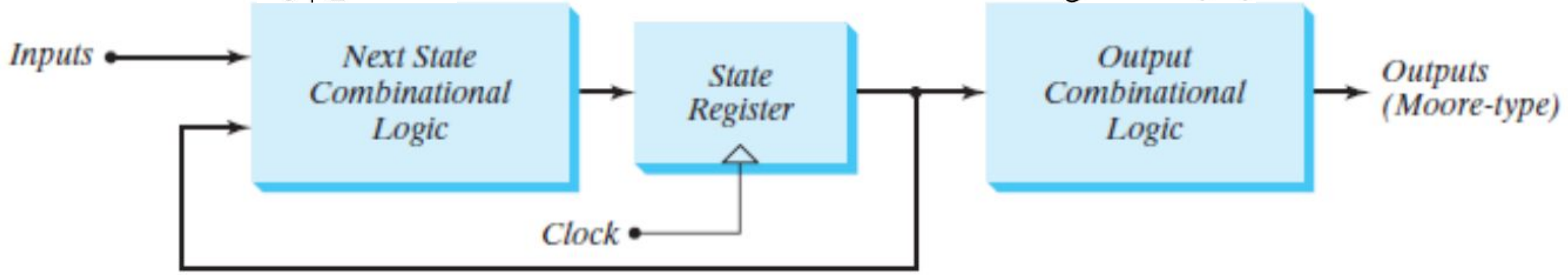
- Output depends only on the current state
- Output changes synchronously

This is a Moore State Machine

Current State	Next State		Output
	(Input) 0	**(Input) 1**	
00	00	01	0
01	10	01	0
10	00	11	0
11	10	01	1

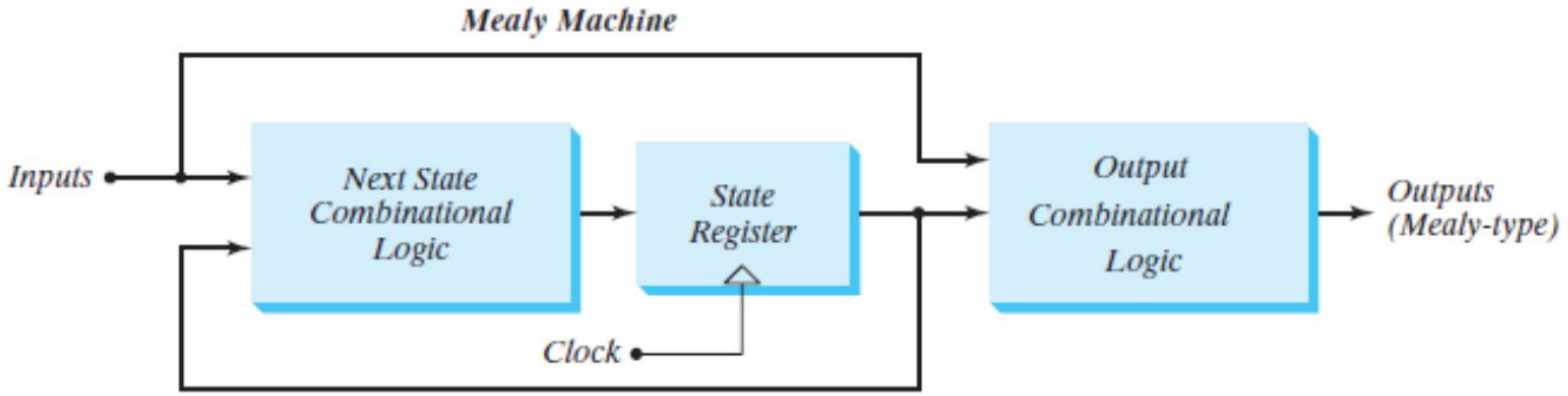
- Input: *x*
- Output: *y*
- Current state: a_t, b_t
- Next state: a_{t+1}, b_{t+1}

$$a_{t+1} = \bar{x}b_t + xa_t\bar{b}_t$$
$$b_{t+1} = x$$
$$y = a_tb_t$$



Mealy State Machine

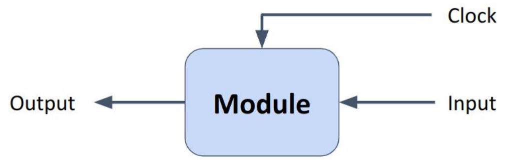
- In Mealy machines, output depends on both the input and the current state
- Most systems involving sequential logic can be implemented as both a Moore State Machine and a Mealy State Machine
  - The choice is with the designer



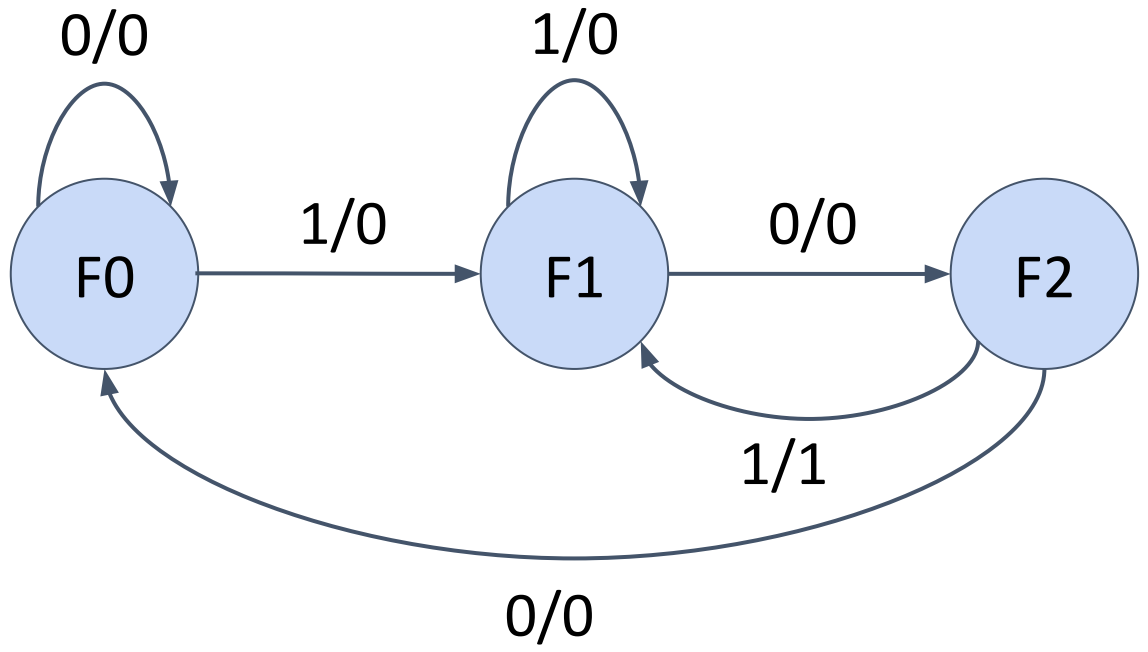
Mealy State Machine

Task:

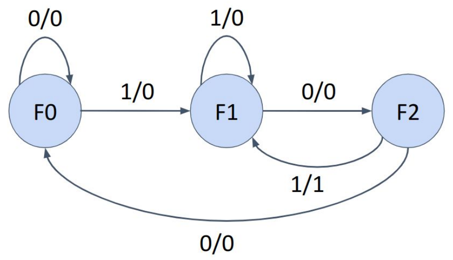
Identify the “101” bit patterns in a series of incoming bits



State	Description	Notation
Nothing found	All the next incoming three bits need to be “101” for the “101” pattern	F0
“1” found	Only the next incoming two bits need to be “01” for the “101” pattern	F1
“10” found	Only the next incoming bit needs to be “1” for the “101” pattern	F2


$$\begin{tikzcd}
F0 \arrow[loop above, "0/0"] \arrow[r, "1/0"] & F1 \arrow[loop above, "1/0"] \arrow[r, "0/0"] & F2 \arrow[l, bend left, "1/1"] \arrow[ll, bend left=60, "0/0"]
\end{tikzcd}$$

State Transition Table



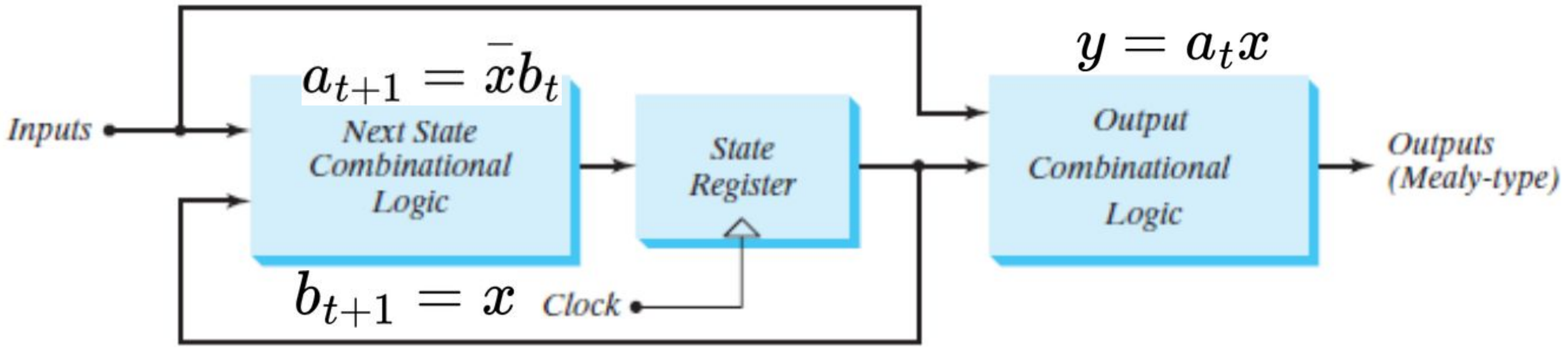
State	Bit Pattern
F0	00
F1	01
F2	10

Current State	Input	Next State	Output
00	0	00	0
00	1	01	0
01	0	10	0
01	1	01	0
10	0	00	0
10	1	01	1

Mealy State Machine

Current State	Input	Next State	Output
00	0	00	0
00	1	01	0
01	0	10	0
01	1	01	0
10	0	00	0
10	1	01	1

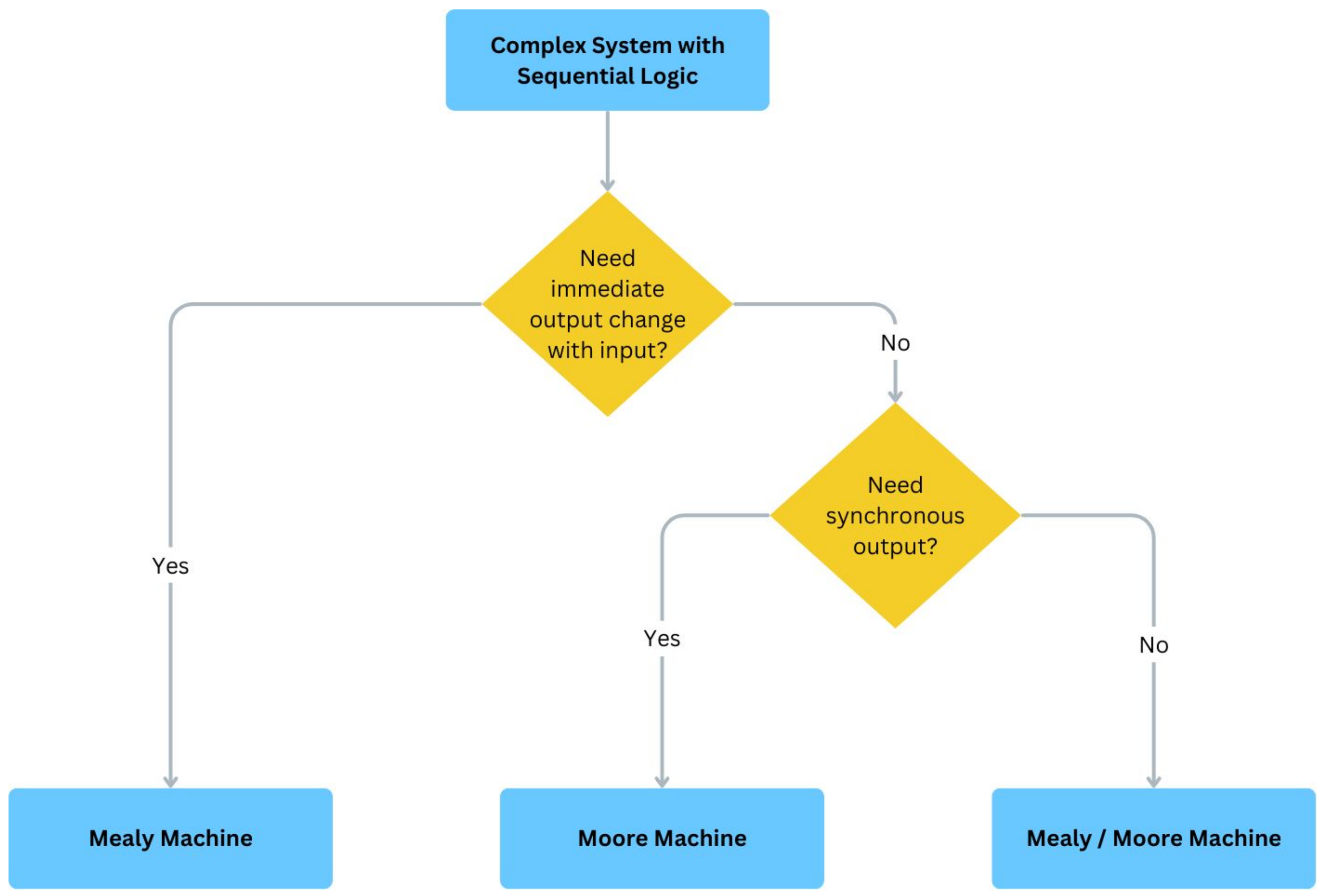
- Input: *x*
- Output: *y*
- Current state: a_t, b_t
- Next state: a_{t+1}, b_{t+1}



Moore Machine Vs Mealy Machine

Moore Machine	Mealy Machine
Outputs are synchronized with the clock	Outputs may change if the inputs change during a clock cycle
Number of states is comparatively higher	Number of states is comparatively lower
Comparatively easier to implement and debug	Comparatively more difficult to implement and debug

Use Moore or Mealy?

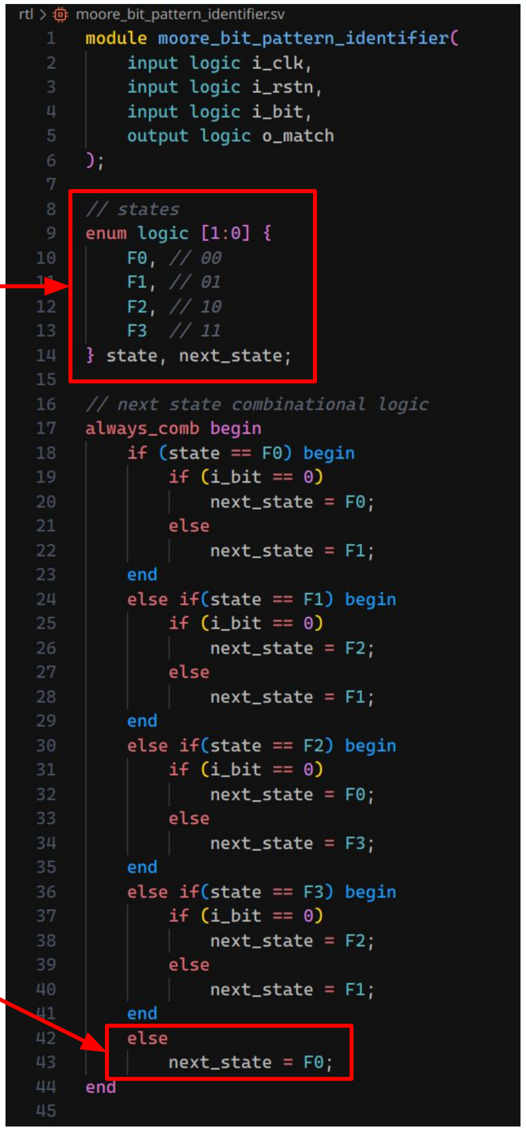


- Mealy Machine:
  - Missile defence system activation alarm
- Moore Machine:
  - Pedestrian crossing signal
- Light activation system that is triggered by a person walking in

Design: Moore Machine

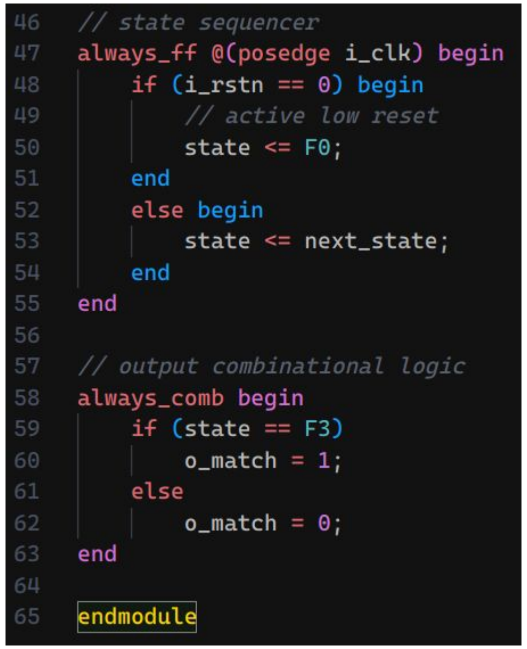
- enum is used to name bit patterns.
- It is useful for clarity when designing and debugging.

- This is needed to stop the synthesis tool from inferring a latch.



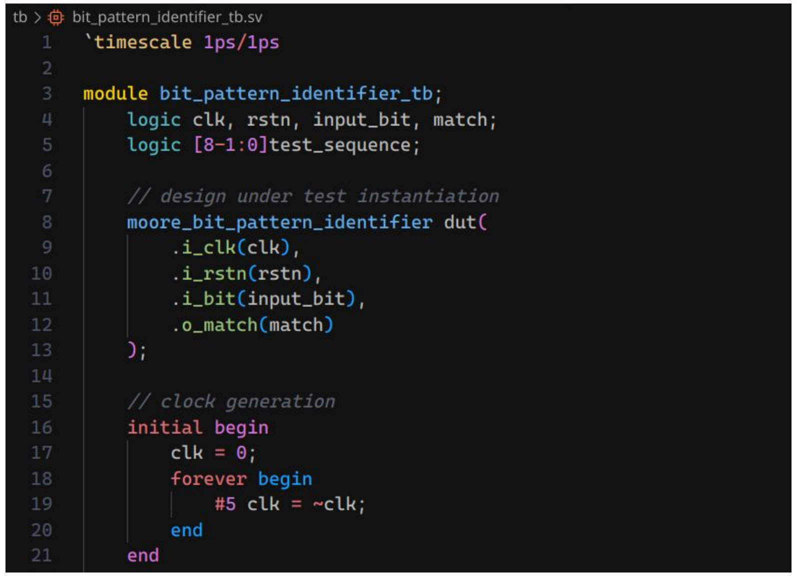
rtl > moore_bit_pattern_identifier.sv

```systemverilog
module moore_bit_pattern_identifier(
    input logic i_clk,
    input logic i_rstn,
    input logic i_bit,
    output logic o_match
);

// states
enum logic [1:0] {
    F0, // 00
    F1, // 01
    F2, // 10
    F3  // 11
} state, next_state;

// next state combinational logic
always_comb begin
    if (state == F0) begin
        if (i_bit == 0)
            next_state = F0;
        else
            next_state = F1;
    end
    else if(state == F1) begin
        if (i_bit == 0)
            next_state = F2;
        else
            next_state = F1;
    end
    else if(state == F2) begin
        if (i_bit == 0)
            next_state = F0;
        else
            next_state = F3;
    end
    else if(state == F3) begin
        if (i_bit == 0)
            next_state = F2;
        else
            next_state = F1;
    end
    else
        next_state = F0;
end

```

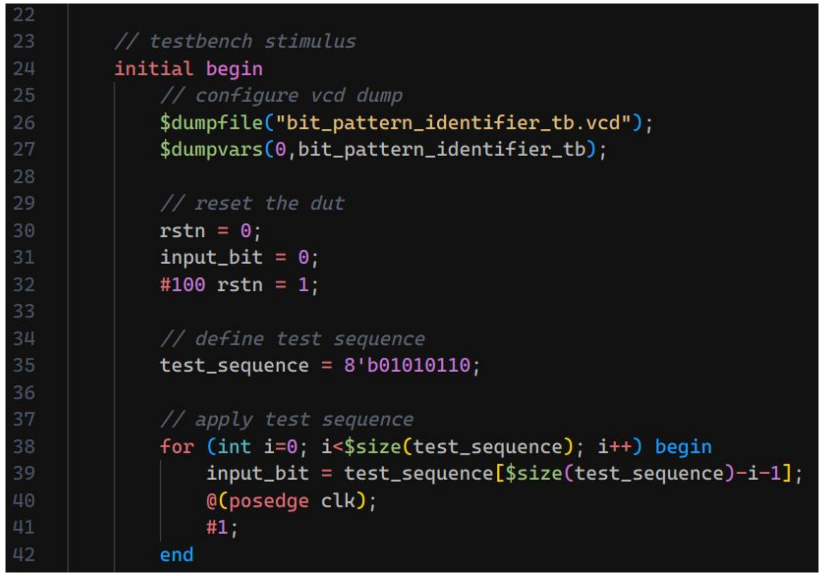


```systemverilog
// state sequencer
always_ff @(posedge i_clk) begin
    if (i_rstn == 0) begin
        // active low reset
        state <= F0;
    end
    else begin
        state <= next_state;
    end
end

// output combinational logic
always_comb begin
    if (state == F3)
        o_match = 1;
    else
        o_match = 0;
end

endmodule
```

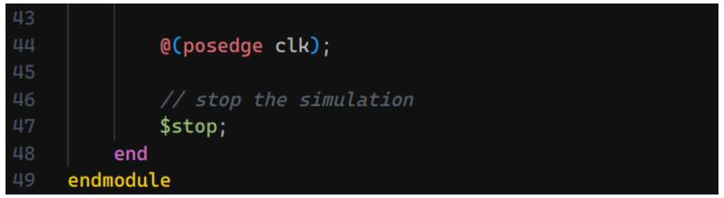
Verification: Moore Machine


tb > bit_pattern_identifier_tb.sv

```systemverilog
`timescale 1ps/1ps

module bit_pattern_identifier_tb;
    logic clk, rstn, input_bit, match;
    logic [8-1:0]test_sequence;

    // design under test instantiation
    moore_bit_pattern_identifier dut(
        .i_clk(clk),
        .i_rstn(rstn),
        .i_bit(input_bit),
        .o_match(match)
    );

    // clock generation
    initial begin
        clk = 0;
        forever begin
            #5 clk = ~clk;
        end
    end
```

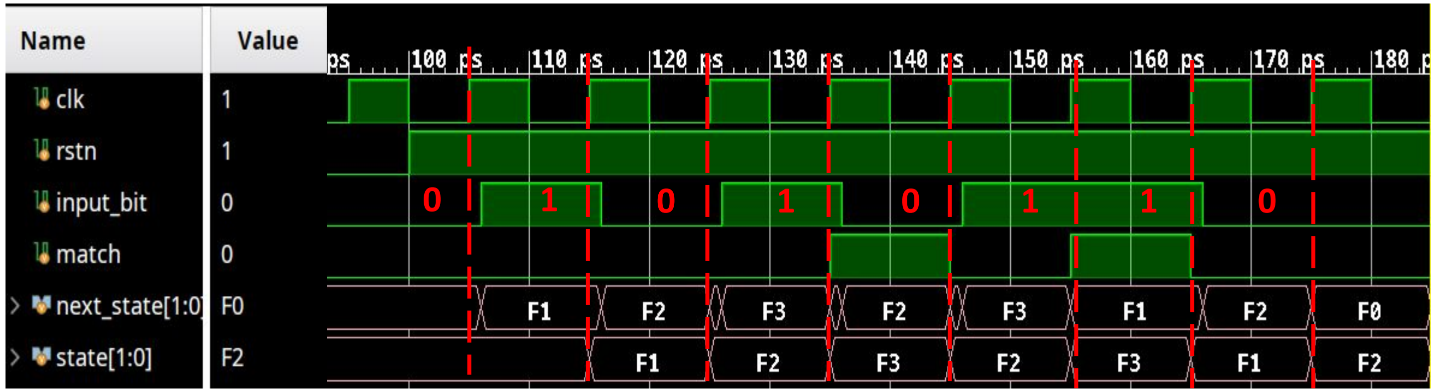


```systemverilog

    // testbench stimulus
    initial begin
        // configure vcd dump
        $dumpfile("bit_pattern_identifier_tb.vcd");
        $dumpvars(0,bit_pattern_identifier_tb);

        // reset the dut
        rstn = 0;
        input_bit = 0;
        #100 rstn = 1;

        // define test sequence
        test_sequence = 8'b01010110;

        // apply test sequence
        for (int i=0; i<$size(test_sequence); i++) begin
            input_bit = test_sequence[$size(test_sequence)-i-1];
            @(posedge clk);
            #1;
        end
```



```systemverilog

            @(posedge clk);

            // stop the simulation
            $stop;
        end
endmodule
```

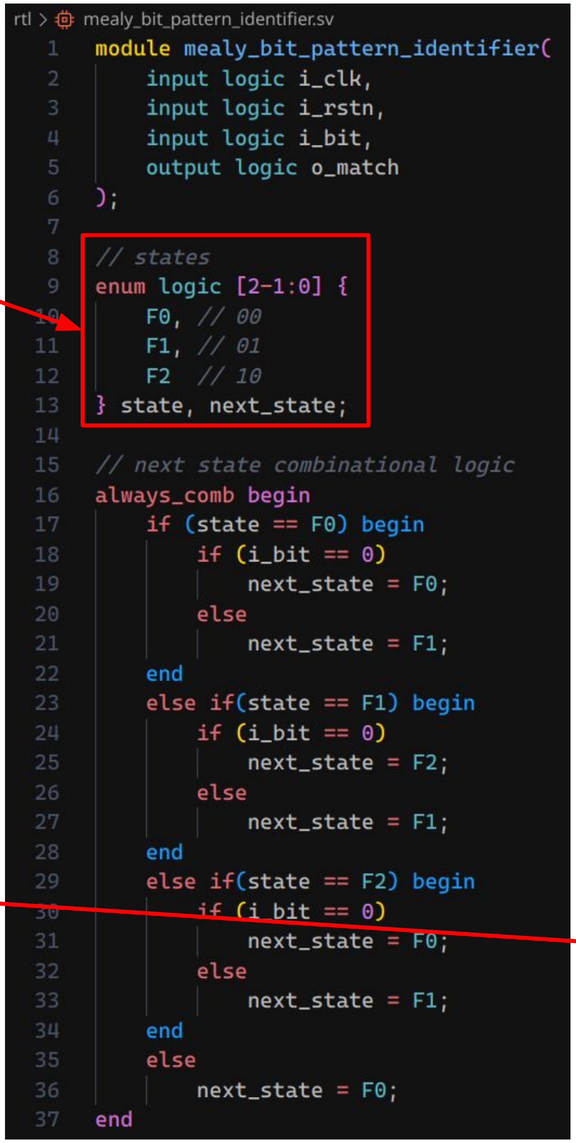
Test Pattern:
01010110



Design: Mealy Machine

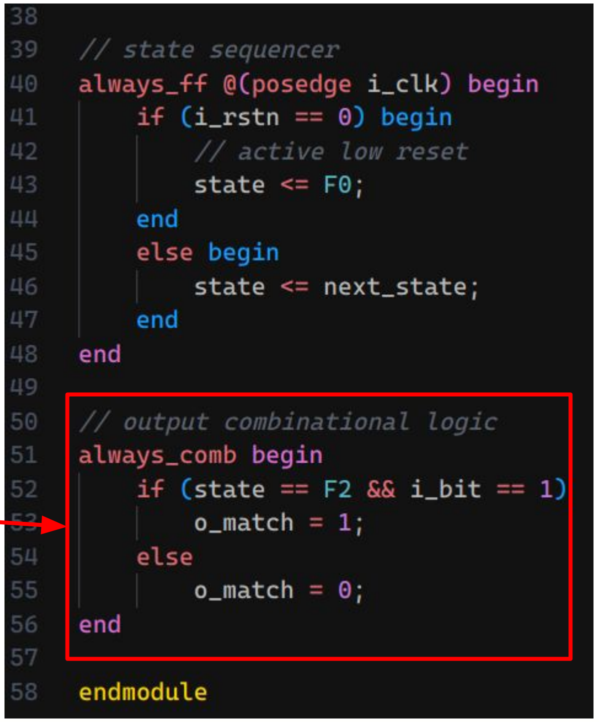
- Less number of states

- Output depends on the input



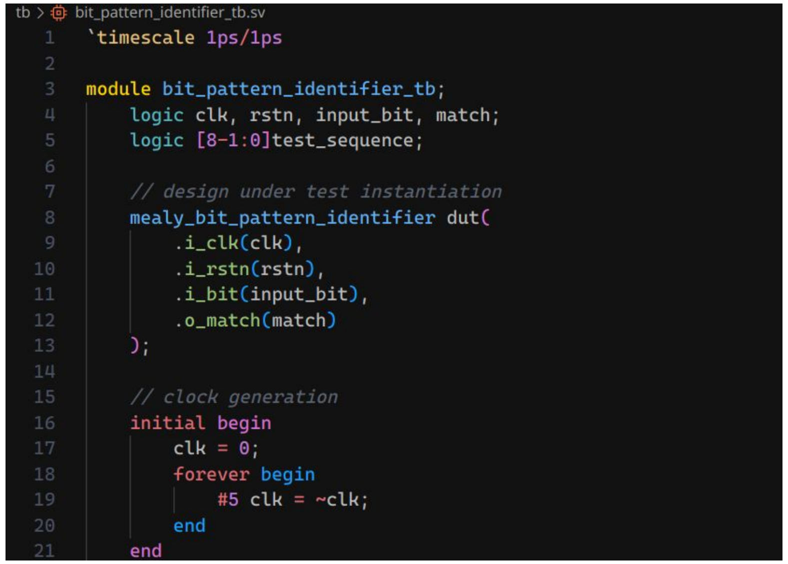
rtl > mealy_bit_pattern_identifier.sv

```systemverilog
module mealy_bit_pattern_identifier(
    input logic i_clk,
    input logic i_rstn,
    input logic i_bit,
    output logic o_match
);

// states
enum logic [2-1:0] {
    F0, // 00
    F1, // 01
    F2  // 10
} state, next_state;

// next state combinational logic
always_comb begin
    if (state == F0) begin
        if (i_bit == 0)
            next_state = F0;
        else
            next_state = F1;
    end
    else if(state == F1) begin
        if (i_bit == 0)
            next_state = F2;
        else
            next_state = F1;
    end
    else if(state == F2) begin
        if (i_bit == 0)
            next_state = F0;
        else
            next_state = F1;
    end
    else
        next_state = F0;
end
```



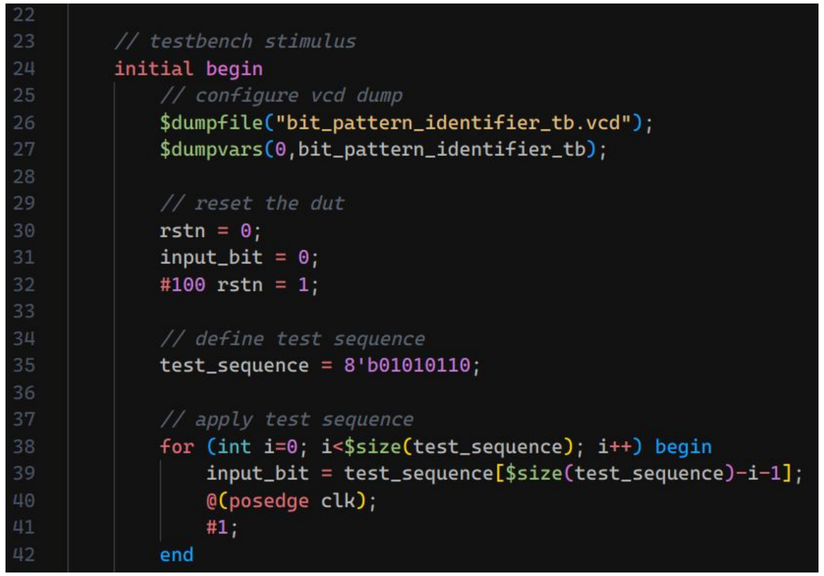
```systemverilog

// state sequencer
always_ff @(posedge i_clk) begin
    if (i_rstn == 0) begin
        // active low reset
        state <= F0;
    end
    else begin
        state <= next_state;
    end
end

// output combinational logic
always_comb begin
    if (state == F2 && i_bit == 1)
        o_match = 1;
    else
        o_match = 0;
end

endmodule
```

Verification: Mealy Machine

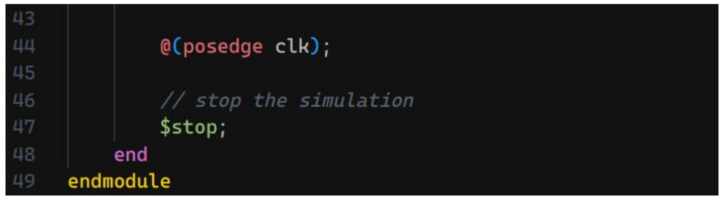


tb > bit_pattern_identifier_tb.sv

```
`timescale 1ps/1ps

module bit_pattern_identifier_tb;
    logic clk, rstn, input_bit, match;
    logic [8-1:0]test_sequence;

    // design under test instantiation
    mealy_bit_pattern_identifier dut(
        .i_clk(clk),
        .i_rstn(rstn),
        .i_bit(input_bit),
        .o_match(match)
    );

    // clock generation
    initial begin
        clk = 0;
        forever begin
            #5 clk = ~clk;
        end
    end
```

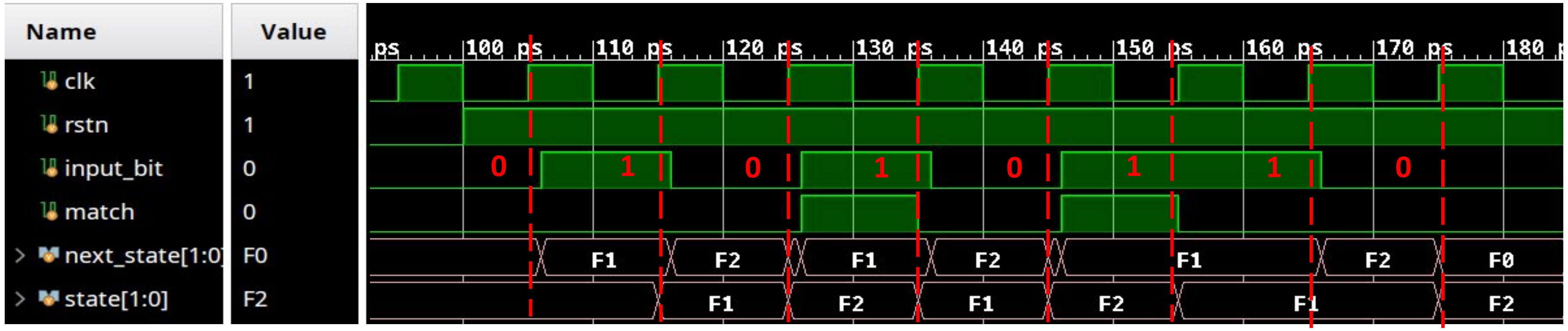


```

    // testbench stimulus
    initial begin
        // configure vcd dump
        $dumpfile("bit_pattern_identifier_tb.vcd");
        $dumpvars(0,bit_pattern_identifier_tb);

        // reset the dut
        rstn = 0;
        input_bit = 0;
        #100 rstn = 1;

        // define test sequence
        test_sequence = 8'b01010110;

        // apply test sequence
        for (int i=0; i<$size(test_sequence); i++) begin
            input_bit = test_sequence[$size(test_sequence)-i-1];
            @(posedge clk);
            #1;
        end
```



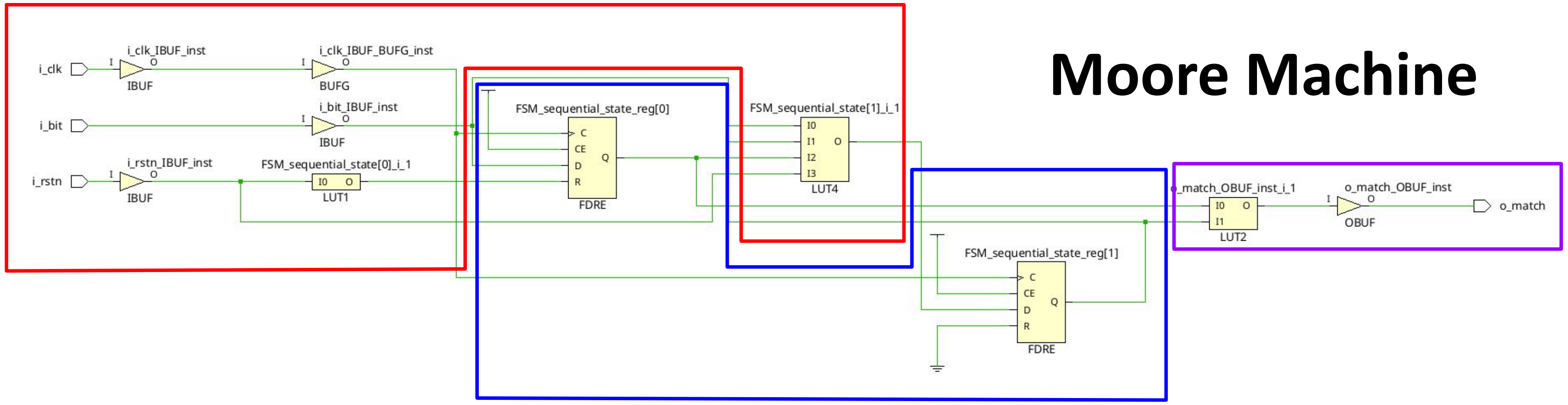
```

            @(posedge clk);

            // stop the simulation
            $stop;
        end
endmodule
```

Test Pattern:
01010110



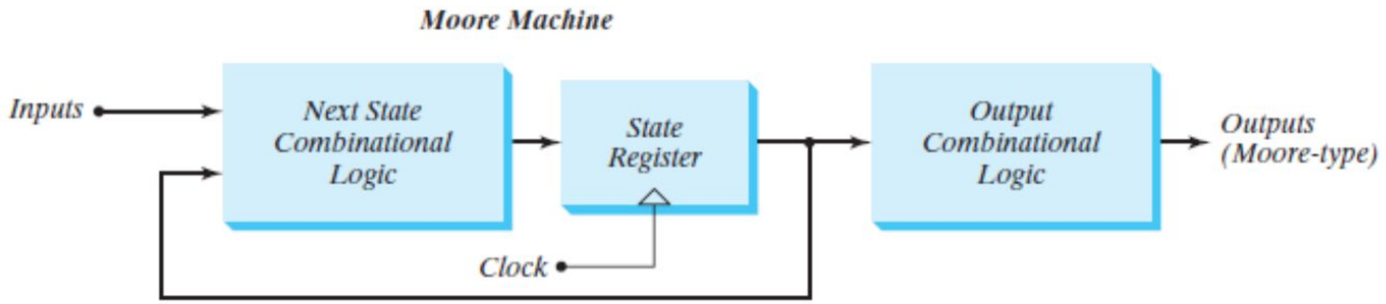
Implementation: Moore Machine

- 🔴 Next state combinational logic
- 🔵 State registers
- 🟣 Output combinational logic



Moore Machine

States in a digital circuit are the all the possible value combinations of its memory elements

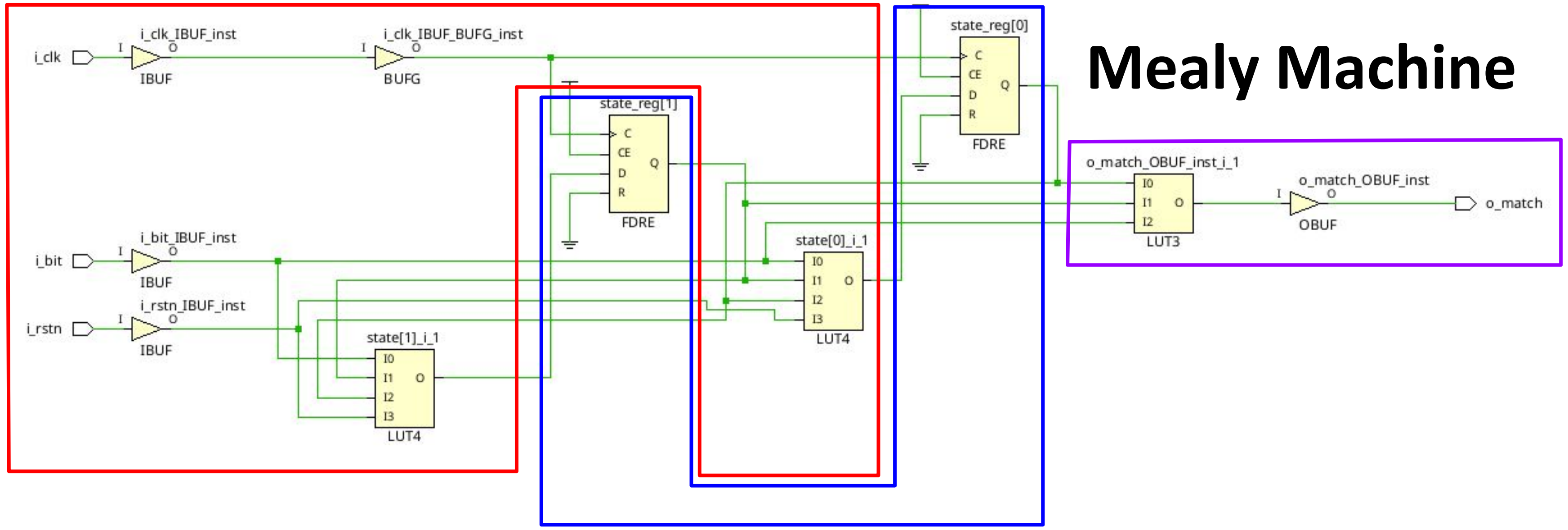


Implementation: Mealy Machine

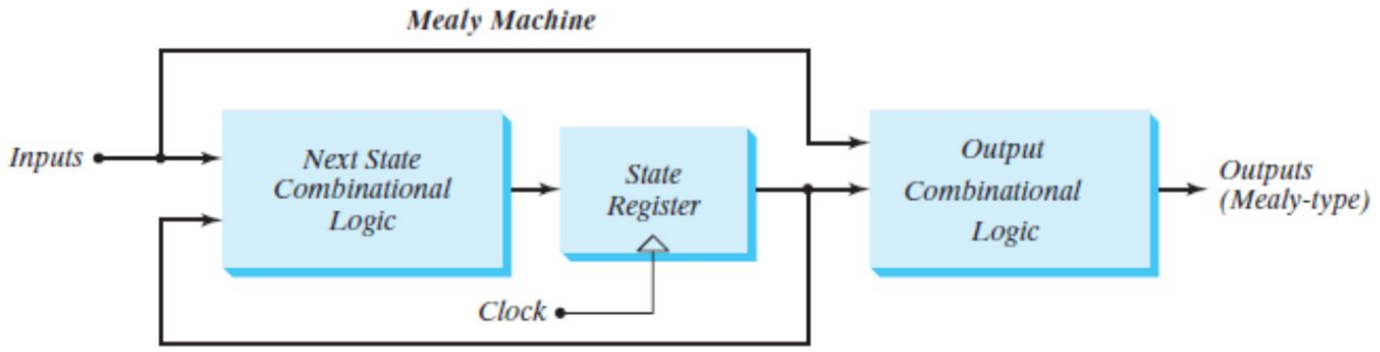
🔴 Next state combinational logic

🔵 State registers

🟣 Output combinational logic



States in a digital circuit are the all the possible value combinations of its memory elements



Q & A