FINITE STATE MACHINES

A Finite State Machine is a computational model consisting of a finite number of states. It transitions between these states based on input signals and current state information. FSMs are defined by:

1. **States**: A finite number of conditions or configurations.
2. **Inputs**: Signals or conditions that affect transitions.
3. **Outputs**: Signals or actions based on the current state.
4. **Transitions**: Rules defining how the FSM moves between states

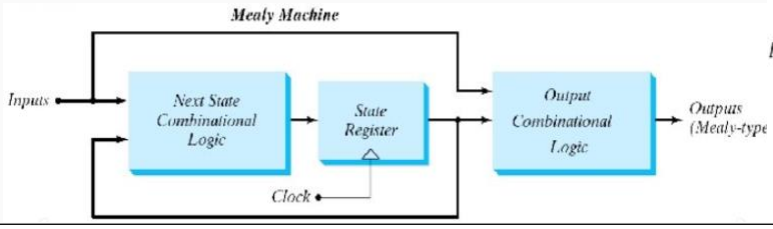
---

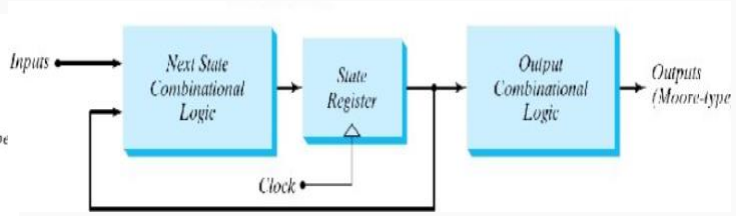
DESIGNING STEPS:

1. **Define the Problem**: Understand the control logic and identify inputs, outputs, and states.
2. **State Diagram**: Draw a state transition diagram illustrating states, transitions, and outputs.
3. **State Encoding**: Assign binary values to states.
4. **Write Verilog Code**: Implement the FSM in Verilog using appropriate constructs.
5. **Simulate and Test**: Verify functionality through simulation.

FSM MACHINE TYPE:

Feature	Mealy Machine	Moore Machine
Output Dependency	Depends on current state and input signals.	Depends only on the current state.
Number of States	May require fewer states for implementation.	May require more states due to state-driven outputs.
Output Timing	Outputs can change in the middle of a clock cycle as inputs change.	Outputs change only at clock edges when state changes.
Complexity	Generally, more complex to design and debug.	Easier to design and debug due to simpler behaviour.
GLITCHES	More prone to glitches as depend on input.	Less prone glitches.
Example Use Cases	Applications requiring immediate response to inputs.	Applications where output stability is more critical.





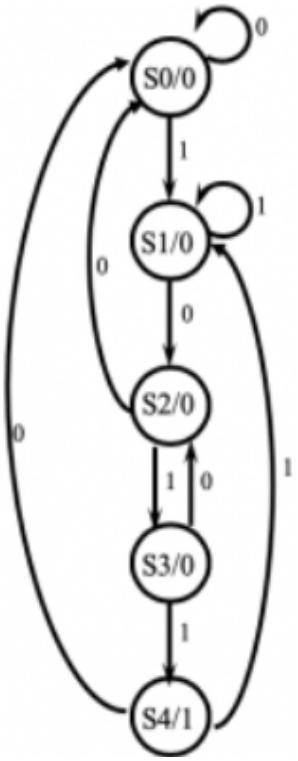
Overlapping Sequence Detection: Allows the detection of a sequence even if it overlaps with a previous sequence.
Non-Overlapping Sequence Detection: Ensures that once a sequence is detected, the FSM resets to the initial state before detecting the next sequence.

State Encoding Techniques: When implementing FSMs, states must be assigned binary code.

Binary encoding	Gray code encoding	One-hot Encoding	Johnson Encoding
Each state is represented by a binary number. Efficient in terms of the number of bits used. May result in complex next-state and output logic.	Consecutive states differ by only one bit. Reduces the risk of glitches during state transitions.	Each state is represented by a single bit set to 1, while all others are 0. Simplifies state transition logic at the cost of more flip-flops.	A circular shift register encoding where one bit shifts in each clock cycle. Useful for specific cyclic or rotating FSMs.

SEQUENCE DETECOR 1011

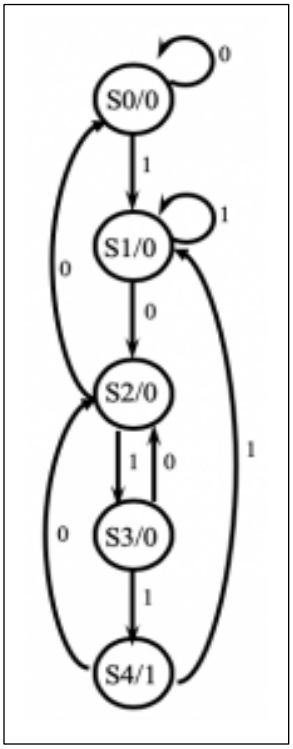
MOORE
NON- OVERLAPPING



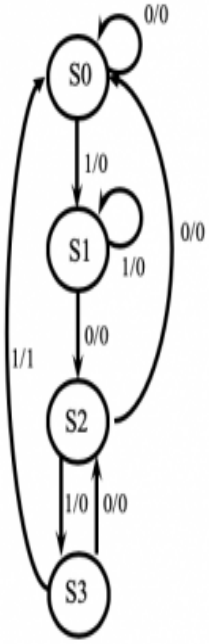

```
module sequence_detector_1011 (
  input clk,
  input reset,
  input din,
  output reg dout
);
 typedef enum reg [2:0]
{S0,S1,S2,S3,S4} state_t;
state_t current_state, next_state;
 always @(posedge clk ) begin
    if (reset)
      current_state <= S0;
    else
      current_state <= next_state;
  end
 always @(*) begin
    case (current_state)
      S0: next_state = din ? S1 : S0;
      S1: next_state = din ? S1 : S2;
      S2: next_state = din ? S3 : S0;
      S3: next_state = din ? S4 : S2;
      S4: next_state = din ? S1 : S2;
      default: next_state = S0;
    endcase
  end
  always @(posedge clk ) begin
    if (reset)
      dout <= 1'b0;
    else
      dout <= (current_state == S4);
  end
endmodule
```

MOORE
OVERLAPPING

```
typedef enum reg [2:0]
{S0,S1,S2,S3,S4} state_t;
state_t current_state, next_state;
// State transition logic
always @(posedge clk ) begin
       if (reset) current_state <= S0;
     else current_state <= next_state;
end
 always @(*) begin
case (current_state)
S0: next_state = din ? S1 : S0;
S1: next_state = din ? S1 : S2;
S2: next_state = din ? S3 : S0;
S3: next_state = din ? S4 : S2;
S4: next_state = din ? S1 : S0;
default: next_state = S0;
endcase
end
 always @(posedge clk )
 begin
   if (reset) dout <= 1'b0;
else dout <= (current_state == S4);
endmodule
```

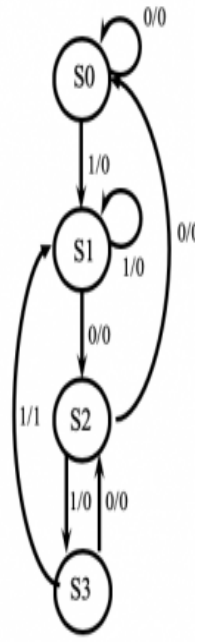


MELAY
NON-
OVERLAPPING




```
module sequence_detector_1011
(
  input clk,
  input reset,
  input din,
  output reg dout
);
 typedef enum reg [2:0]
{S0,S1,S2,S3,S4} state_t;
 state_t current_state, next_state;
always @(posedge clk ) begin
    if (reset)
      current_state <= S0;
    else
      current_state <= next_state;
  end
 always @(*) begin
     case (current_state)
     S0: next_state = din ? S1 : S0;
     S1: next_state = din ? S1 : S2;
     S2: next_state = din ? S3 : S0;
     S3: next_state = din ? S4 : S2;
     S4: next_state = din ? S1 : S2;
  default: next_state = S0;
endcase
  end
always @(*) begin
  dout = (current_state == S3 &&
din == 1'b1);
  end
endmodule
```

MELAY
OVERLAPPING




```
always @(posedge clk ) begin
if (reset) current_state <= S0;
else
current_state <= next_state;
 end
always @(*) begin
case (current_state)
S0: next_state = din ? S1 : S0;
S1: next_state = din ? S1 : S2;
S2: next_state = din ? S3 : S0;
S3: next_state = din ? S4 : S2;
S4: next_state = din ? S1 : S2;
default: next_state = S0;
endcase
end
 always @(current_state or din)
begin
 dout =
 (current_state == S4) ? din : 0;
end
 endmodule
```

Applications of FSM

Finite State Machines (FSMs) are widely used in digital systems due to their ability to model sequential logic efficiently. Below are various applications across different domains:

Digital System Controllers

FSMs are essential in controlling the sequential operations of digital systems, ensuring tasks are performed in a defined order.

- **Traffic Light Controller:** Manages traffic flow by transitioning between light states based on timing and sensor inputs.
- **Vending Machines:** Controls product dispensing and payment processing, handling multiple states like selection, payment, and dispensing.
- **Elevator Control Systems:** Manages floor requests, door operations, and emergency protocols.

2. Communication Protocols

FSMs are integral in defining protocol behaviors, ensuring data integrity and synchronization.

- **UART (Universal Asynchronous Receiver/Transmitter):** Handles data transmission and reception.
- **I2C & SPI Protocols:** Manage communication between microcontrollers and peripherals.
- **Ethernet & USB Controllers:** Implement protocol stacks for data transmission.

3. Processor Control Units

FSMs coordinate operations within processors, enabling efficient execution of instructions.

- **Instruction Decoders:** Determines the operation to execute based on opcode.
- **Pipeline Control:** Manages instruction flow in pipeline stages, handling hazards and dependencies.
- **ALU Control Units:** Directs arithmetic and logical operations based on control signals.

4. Memory Management

FSMs manage read/write operations in memory systems.

- **Cache Controllers:** Handle cache hits, misses, and replacements.
- **DRAM Controllers:** Manage refresh cycles, read/write operations, and bank activations.
- **Flash Memory Controllers:** Oversee block management and wear leveling.

5. Industrial Automation

FSMs control complex machinery and processes in automated systems.

- **Robotic Arm Controllers:** Define movements based on task requirements and sensor inputs.
- **Conveyor Belt Systems:** Control item movement, sorting, and packaging.
- **PLC (Programmable Logic Controllers):** Implement industrial process controls.

6. Consumer Electronics

FSMs enhance user experience and device functionality.

- **Microwave Ovens:** Manage cooking cycles, power levels, and safety checks.
- **Washing Machines:** Control washing cycles, water levels, and spin speed.
- **Remote Controls:** Implement button scanning and command transmission.

7. Automotive Systems

FSMs contribute to vehicle safety and efficiency.

- **Engine Control Units (ECUs):** Regulate fuel injection, ignition timing, and emission control.
- **Anti-lock Braking Systems (ABS):** Monitor wheel speed and modulate brake pressure.
- **Airbag Deployment Systems:** Trigger airbags based on impact sensors.

8. Security Systems

FSMs provide robust security by managing access and monitoring.

- **Biometric Authentication Systems:** Handle fingerprint and facial recognition states.
- **Alarm Systems:** Transition between armed, disarmed, and triggered states.
- **Access Control Systems:** Manage entry based on user credentials