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:

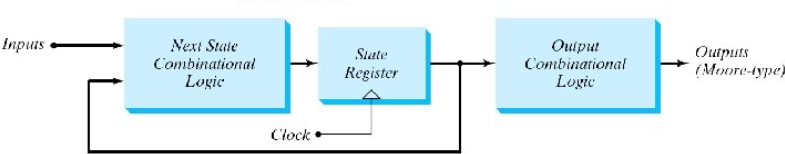
1. **States**: A finite number of conditions or configurations.
2. **Inputs**: Signals or conditions that affect transitions.
3. **Outputs**: Signals or actions based on the current state.
4. **Transitions**: Rules defining how the FSM moves between states

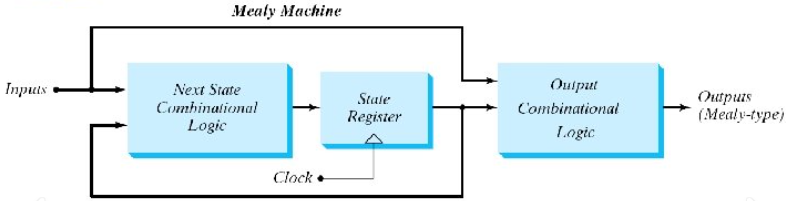
DESIGNING STEPS:

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. |

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

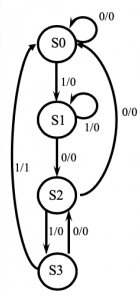
dout = (current\_state == S3 && din == 1'b1);

end

endmodule

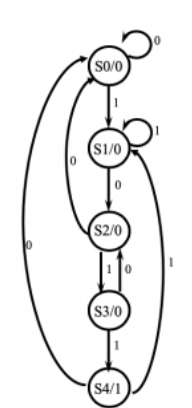
MELAY

NON- OVERLAPPING



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

SEQUENCE DETECOR 1011

MELAY

OVERLAPPING

MOORE

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

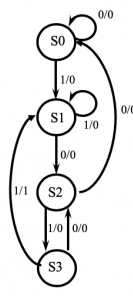
end

always @(current\_state or din) begin

dout =

(current\_state == S4) ? din : 0; end

endmodule



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

