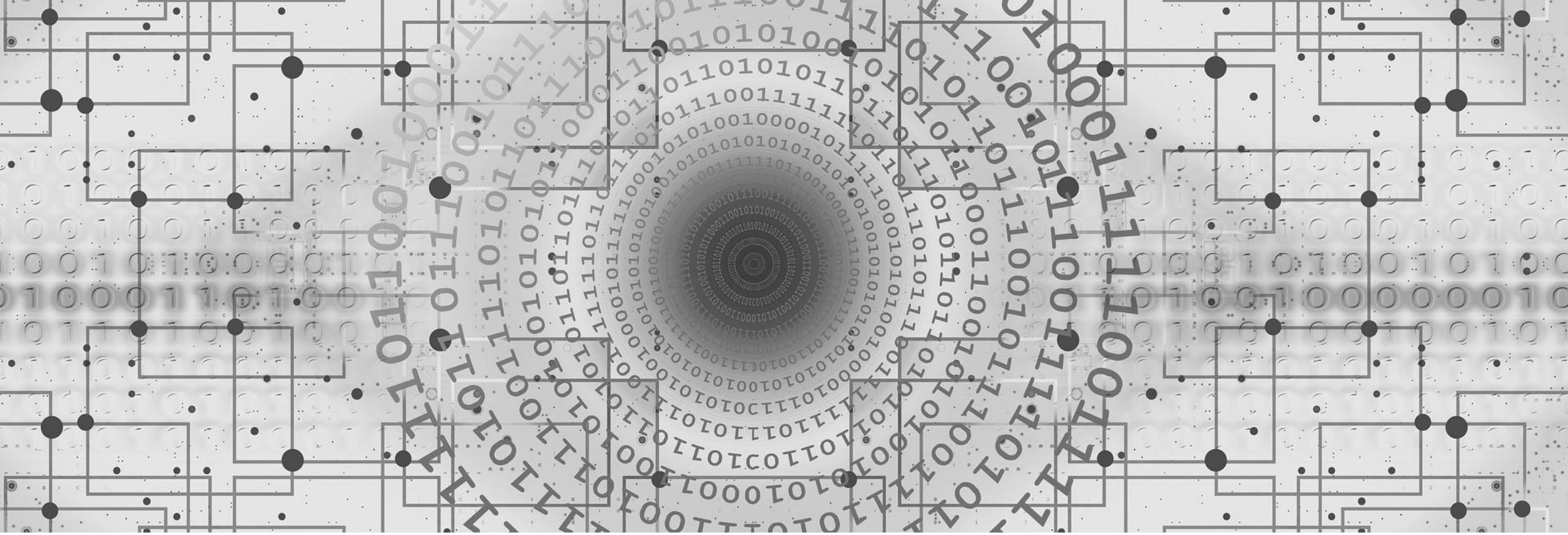


Lecture-12 & 13: Finite State Machine

ECE-111 Advanced Digital Design Project

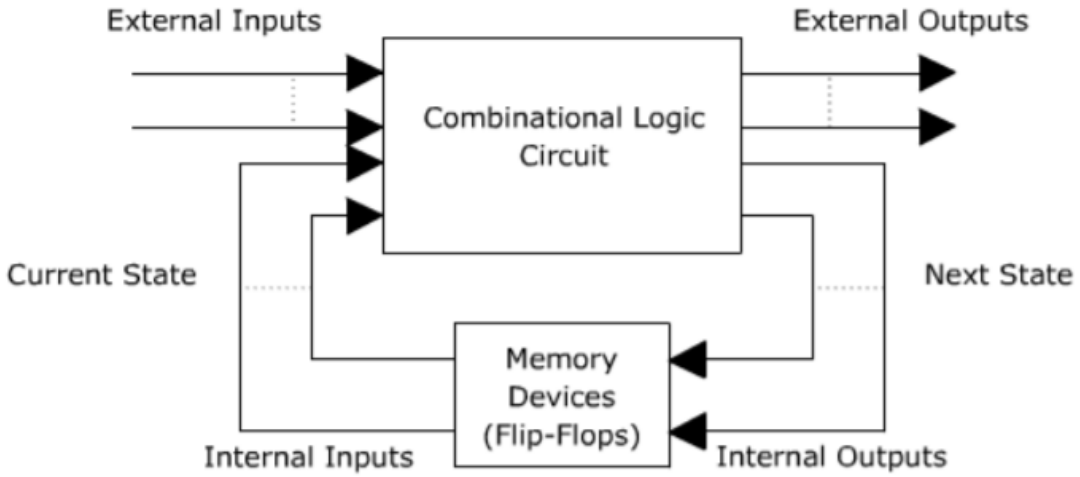
Vishal Karna

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Summer 2025

Finite State Machine in RTL Design

- **Sequential circuits works on a clock cycle which may be synchronous or asynchronous.**
  - Sequential circuits use current inputs and stored past information (typically in form of previous states or outputs) feeding it back to the circuit next clock cycle



- **Finite State Machine (FSM) is a computational model used to design sequential logic**
  - It can be conceived as an abstract machine that stores the current state of a system and transitions between states based on input signals and clock cycles
  - These state transitions produce outputs that reflect the system's behavior in response to changing conditions
  - Ideal for designing control logic, protocol handling, and state-based decision making in hardware
  - FSM based system modeling, enables predictable and verifiable RTL design
  - It can be implemented using models like Mealy and Moore machine.

Finite State Machine in RTL Design

❑ Core Components and Characteristics of an FSM

- **States:** Represents distinct configurations the system can be in
  - An FSM Can only have finite number of pre-defined states
  - An FSM exists only in one specific state at any given moment.
    - Think of a traffic light: it can be in the "Red," "Yellow," or "Green" state
  - An FSM typically has a designated start state (example **idle** or **reset**), and may also have one or more end (or accepting) states.

- **Transitions:** Represents rules for moving between states.
  - An FSM changes state based on inputs (events, conditions) and pre-defined rules.
    - For example, the traffic light transitions from "Red" to "Green" when a timer expires
  - **DFAs (Deterministic Finite Automata) :** always transition to the same state for a given input

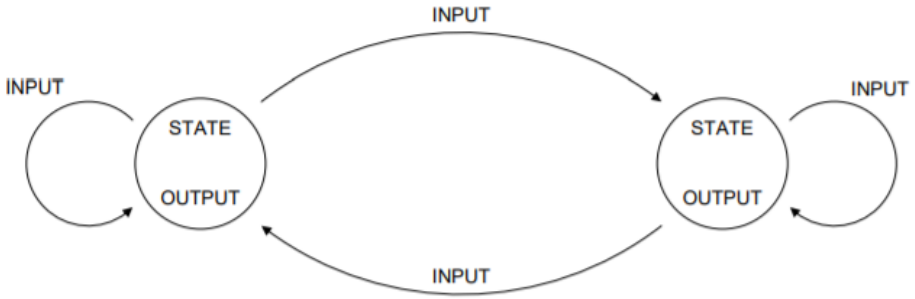
- **Inputs/Outputs:** Signals that drive transitions and reflect system behavior.
  - The input triggers the state transition. It could be a button press, a sensor reading, or a timeout.
  - Some FSMs also have outputs associated with states or transitions.
    - For instance, a vending machine might output a drink after a successful money transaction

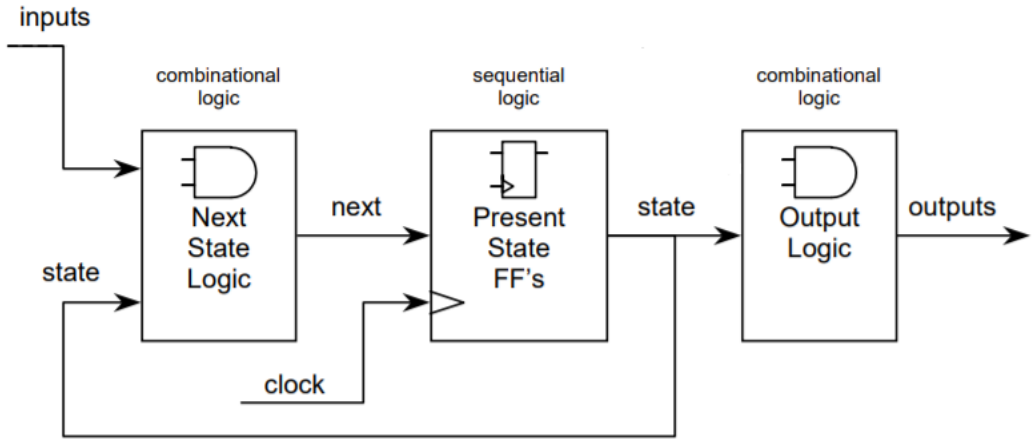
- **Clock:** Synchronizes state updates.

Moore and Mealy Finite State Machine (FSM)

❑ Moore FSM

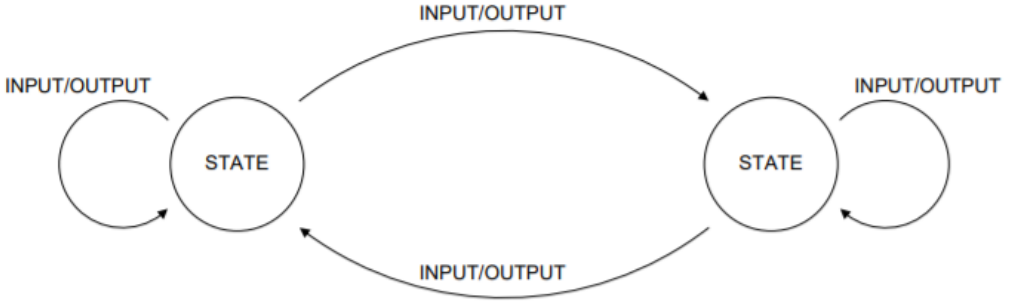
- Output is solely based on present state of FSM
- Output is associated with a state
- Generally, more states than mealy (more hardware)
- More logic required to decode the outputs resulting in more circuit delays.
- Output react slower to input (one clock cycle later)
- Synchronous output and state generation

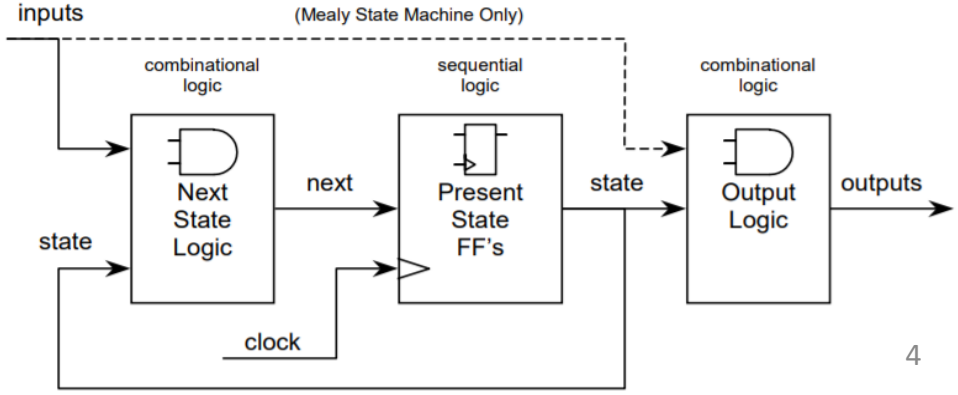




❑ Mealy FSM

- Output based on present state and input(s)
- Output changes during transition of states
- Generally, less states than moore
- Reacts faster to inputs and generally reacts in same cycle
- Asynchronous output generation
- Typically, more complex to design than moore

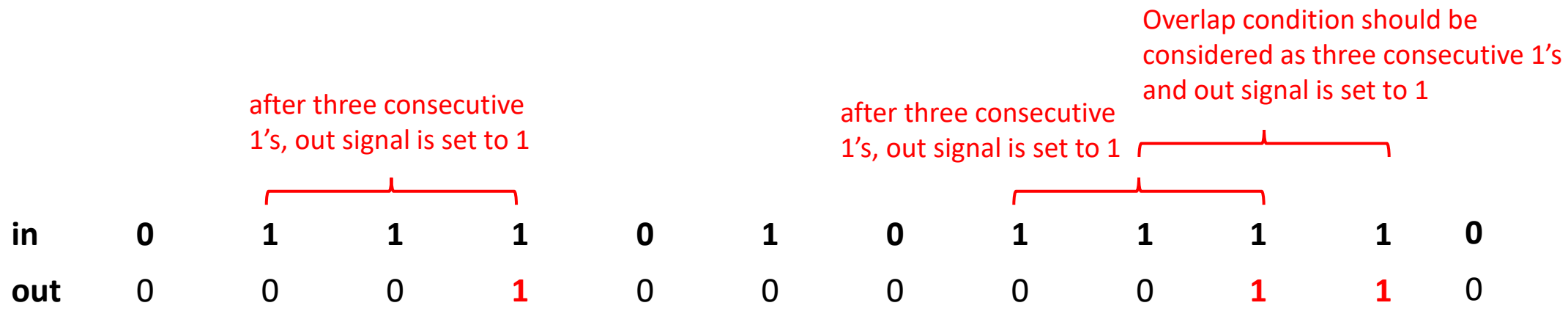




Moore and Mealy Finite State Machine

❑ Example : Sequence Detector can be developed using Finite State Machine

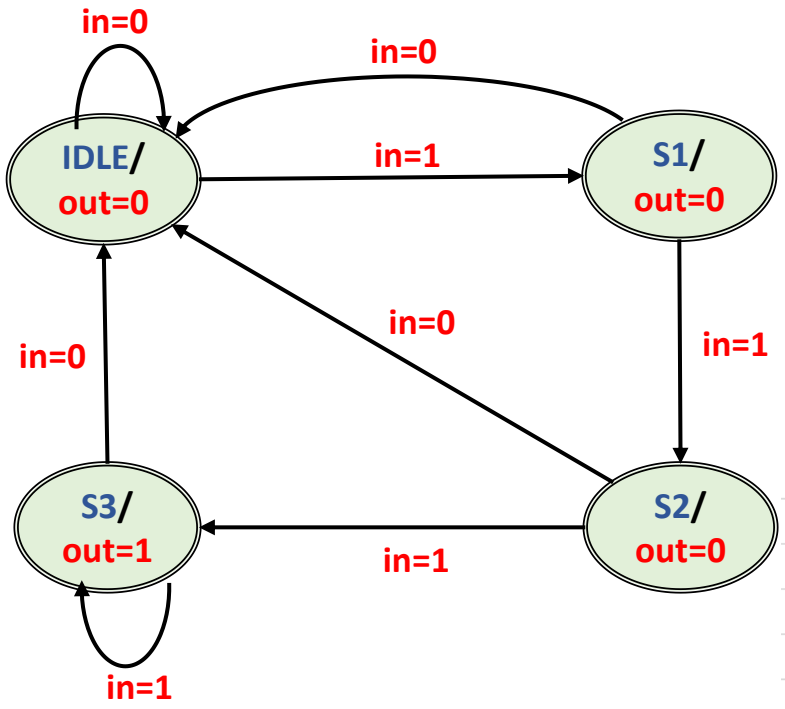
- Design a circuit to detect consecutive series of three or more ‘1’s in serial input bit stream
- Output will become ‘1’ when three or more consecutive ones are detected
- 4 states required to such sequence detector state machine :
  - State IDLE: reset sate (zero 1s detected)
  - State S1: one 1 detected
  - State S2: two 1s detected
  - State S3: three 1s detected
- Let’s consider below mentioned input bit stream and observe output (out) behavior



Moore and Mealy Finite State Machine (FSM)

- FSM can be represented in form of state table or state transition diagram

Moore FSM State Transition Diagram for Sequence Detector

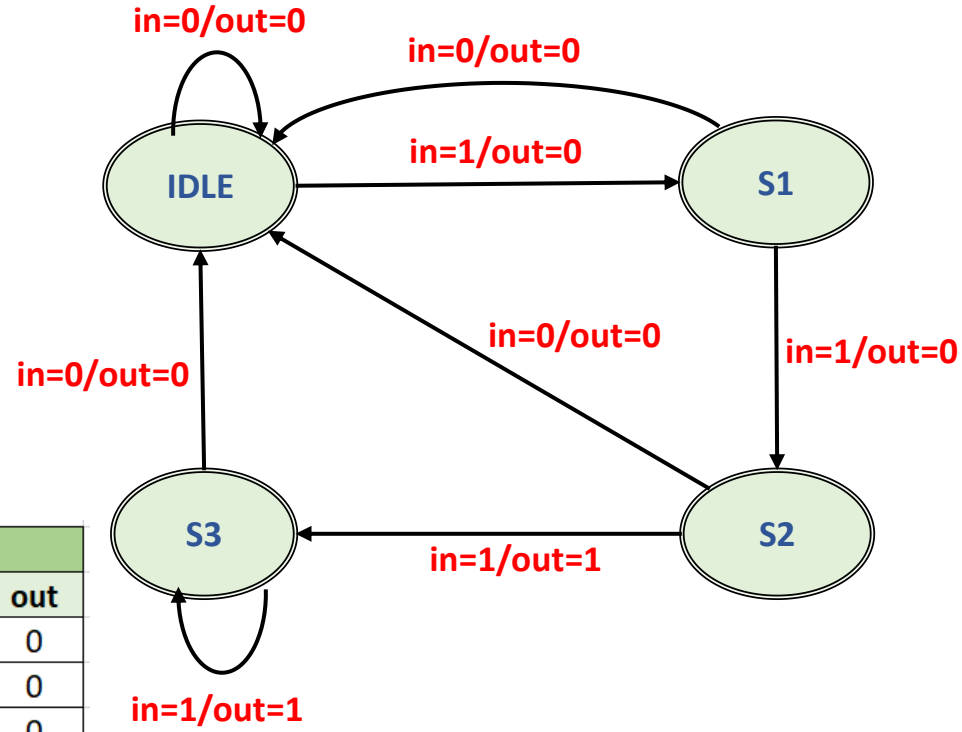


Note : In moore FSM, output is specified as part of present state

State Transition Table for Sequence Detector

inputs			outputs	
rstn	present_state	in	next_state	out
0	-	-	IDLE	0
1	IDLE	0	IDLE	0
1	IDLE	1	S1	0
1	S1	0	IDLE	0
1	S1	1	S2	0
1	S2	0	IDLE	0
1	S2	1	S3	1
1	S3	0	IDLE	0
1	S3	1	S3	1

Mealy FSM State Transition Diagram For Sequence Detector



Note : In mealy FSM, output is specified as part of present state

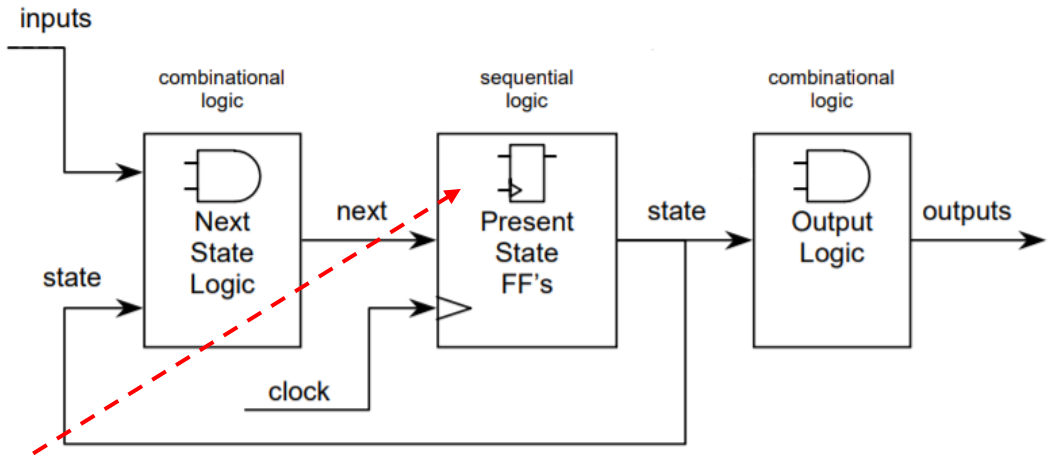
Sequence Detector (Moore FSM – 2 always block approach)

```
module sequence_detector_moore(
 input logic clk, rstn,
 input logic in,
 output logic out);

 // Parameters to define FSM state encodings
 localparam [1:0] IDLE=2'b00,
                  S1=2'b01,
                  S2=2'b10,
                  S3=2'b11;

 // Current state and next state variables
 logic[1:0] present_state, next_state;

 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if(!rstn)
    present_state <= IDLE;
  else
   present_state <= next_state;
 end

 (continued on next page….)
```

State encodings are declared as localparam so that these cannot be modified from outside

Synthesis will generate d-flipflop for present_state

For sequential logic Non-blocking assignments used



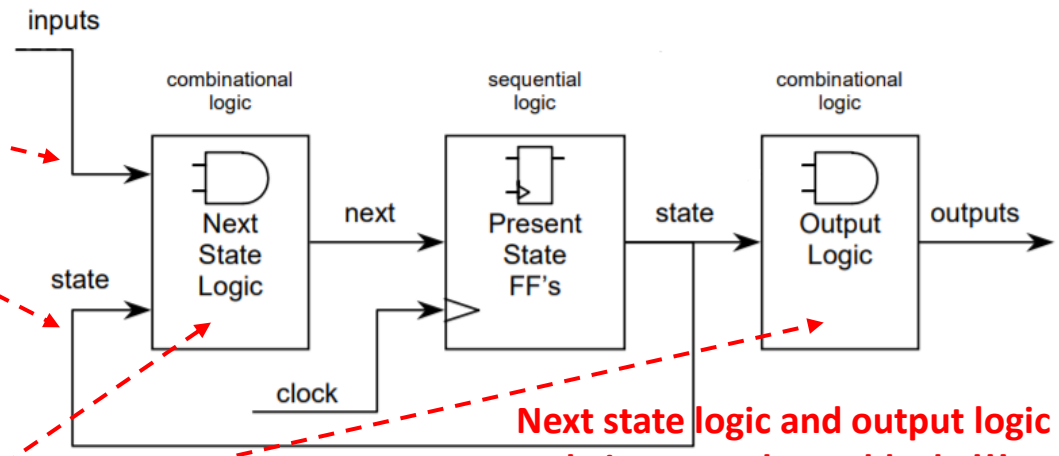
Sequence Detector (Moore FSM – 2 always block approach)

```
// Combination Logic for Next State and Output
always@(present_state,in) begin
 case(present_state)
  IDLE: begin
    out = 0;
    if(in==1) next_state = S1;
    else next_state = IDLE;
  end
  S1: begin
    out = 0;
    if(in==1) next_state = S2;
    else next_state = IDLE;
  end
  S2: begin
    out = 0;
    if(in==1) next_state = S3;
    else next_state = IDLE;
  end
  S3: begin
    out = 1;
    if(in==1) next_state = S3;
    else next_state = IDLE;
  end
  (continued on next page….)
```

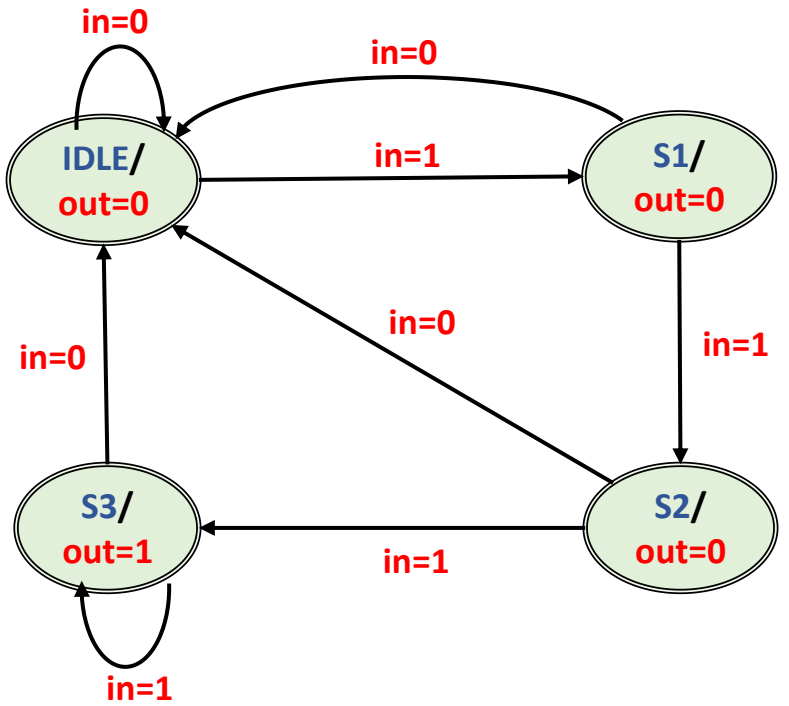
both present_state and input "in" variable should be listed in sensitivity list

For next state and output combination logic, blocking assignments should be used

For Moore FSM, output is set without influence of input "in" signal and purely based on present_state



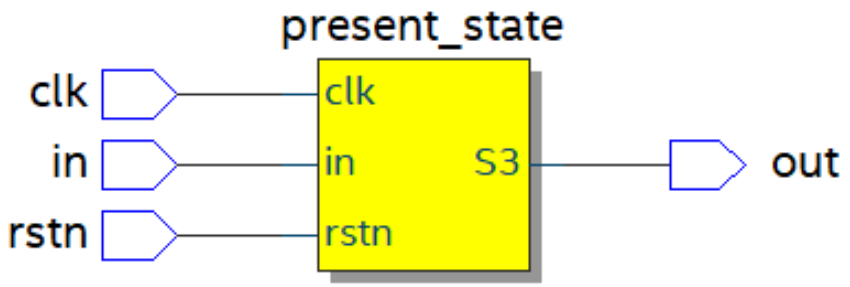
Next state logic and output logic code in same always block !!!



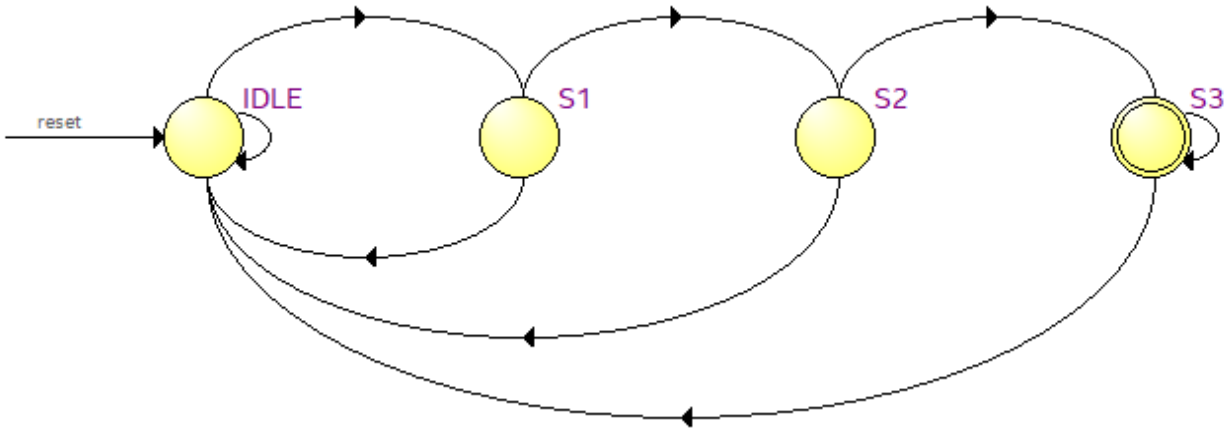
Sequence Detector (Moore FSM – 2 always block approach)

```
default: begin
    out = 0;
    next_state = IDLE;
  end
 endcase
end
endmodule: sequence_detector_moore
```

default is specified in case a bad state is reached



Sequence Detector RTL Netlist View generated from Synthesizer



Sequence Detector State machine diagram generated by Synthesizer

	Source State	Destination State	Condition
1	IDLE	IDLE	(!in) + (in).(!rstn)
2	IDLE	S1	(in).(rstn)
3	S1	IDLE	(!in) + (in).(!rstn)
4	S1	S2	(in).(rstn)
5	S2	IDLE	(!in) + (in).(!rstn)
6	S2	S3	(in).(rstn)
7	S3	IDLE	(!in) + (in).(!rstn)
8	S3	S3	(in).(rstn)

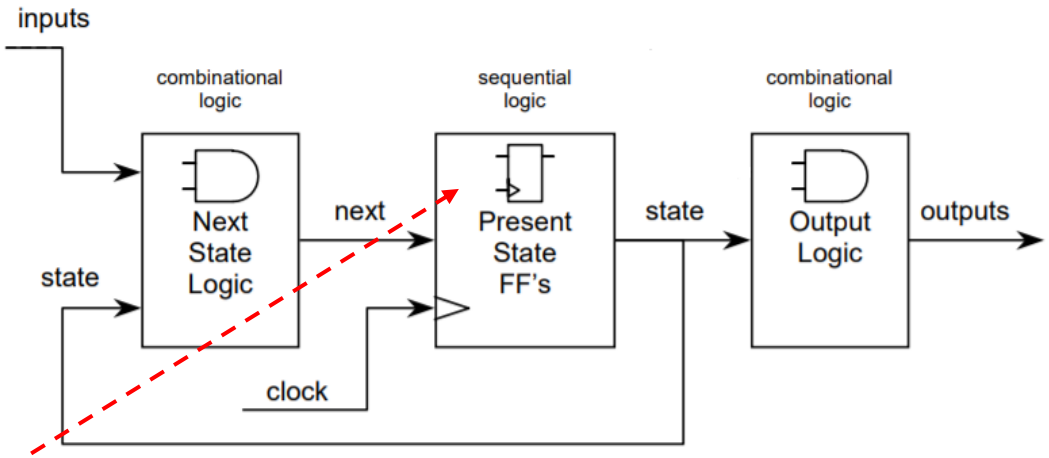
State machine Transition Table

Sequence Detector (Moore FSM – 3 always block approach)

```
module sequence_detector_moore(
 input logic clk, rstn,
 input logic in,
 output logic out);

 // Parameters to define FSM state encodings
 localparam [1:0] IDLE=2'b00,
                  S1=2'b01,
                  S2=2'b10,
                  S3=2'b11;

 // Current state and next state variables
 logic[1:0] present_state, next_state;

 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if(!rstn)
    present_state <= IDLE;
  else
   present_state <= next_state;
 end

 (continued on next page….)
```

State encodings are declared as localparam so that these cannot be modified from outside

1st always block for present state sequential logic



Sequence Detector (Moore FSM – 3 always block approach)

```
// Combination Logic for Next State and Output
always@(present_state,in) begin
 case(present_state)
  IDLE: begin
    if(in==1) next_state = S1;
    else next_state = IDLE;
  end
  S1: begin
    if(in==1) next_state = S2;
    else next_state = IDLE;
  end
  S2: begin
    if(in==1) next_state = S3;
    else next_state = IDLE;
  end
  S3: begin
    if(in==1) next_state = S3;
    else next_state = IDLE;
  end
  default: next_state = IDLE;
endcase
 end
 (continued….)
```

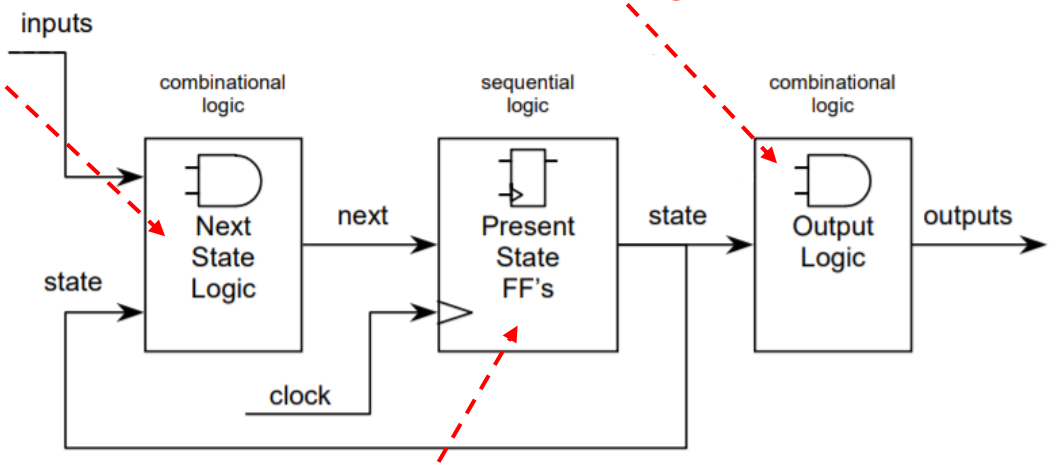
both present_state and input "in" variable should be listed in sensitivity list

```
always@(present_state) begin
 case(present_state)
  S3: out = 1;
  default: out = 0;
 endcase
end
endmodule: sequence_detector_moore
```

Output is dependent Only on present state in moore

2nd always block for separate combinational block for next state logic

3rd always block separate combinational block for output logic



1st always block for present state FF (see previous slide)

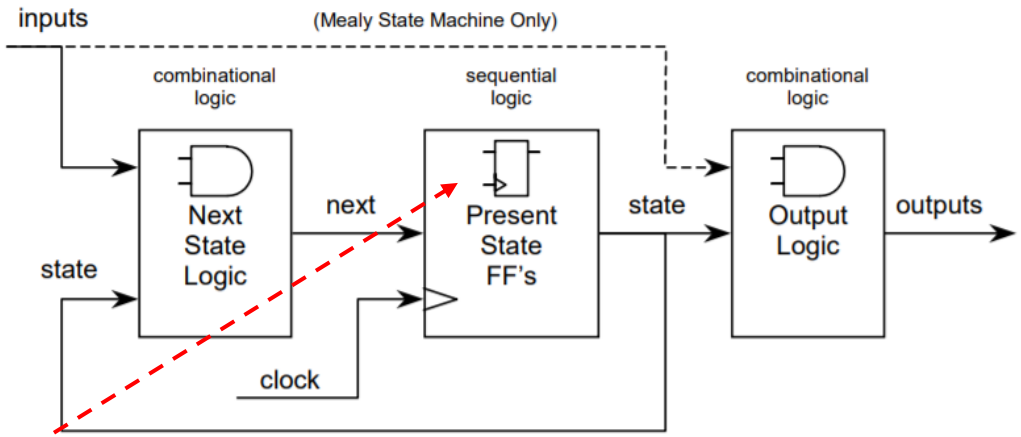
Sequence Detector (Mealy FSM)

```
module sequence_detector_mealy(
 input logic clk, rstn,
 input logic in,
 output logic out);

 // Parameters to define FSM state encodings
 localparam [1:0] IDLE=2'b00,
                  S1=2'b01,
                  S2=2'b10,
                  S3=2'b11;

 // Current state and next state variables
 logic[1:0] present_state, next_state;

 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if(!rstn)
    present_state <= IDLE;
  else
   present_state <= next_state;
 end

 (continued on next page….)
```

State encodings are declared as localparam so that these cannot be modified from outside

Synthesis will generate d-flipflop for present_state

For sequential logic Non-blocking assignments used



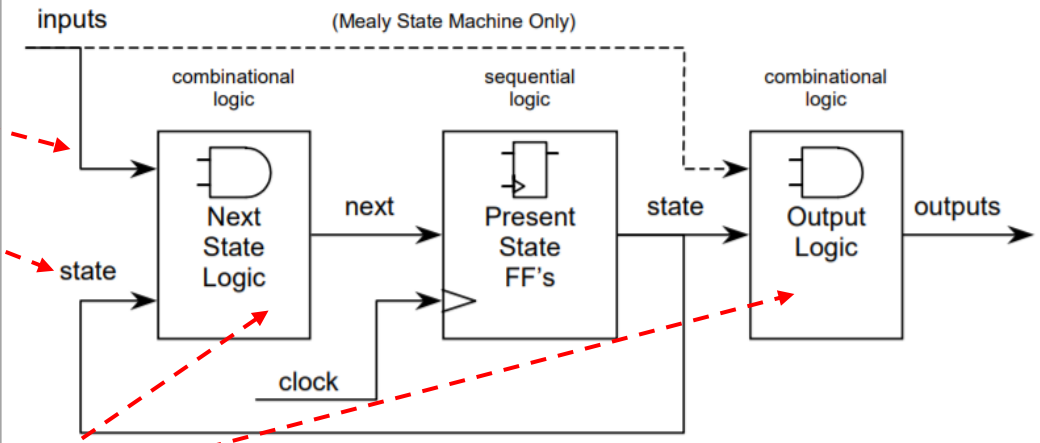
Sequence Detector (Mealy FSM)

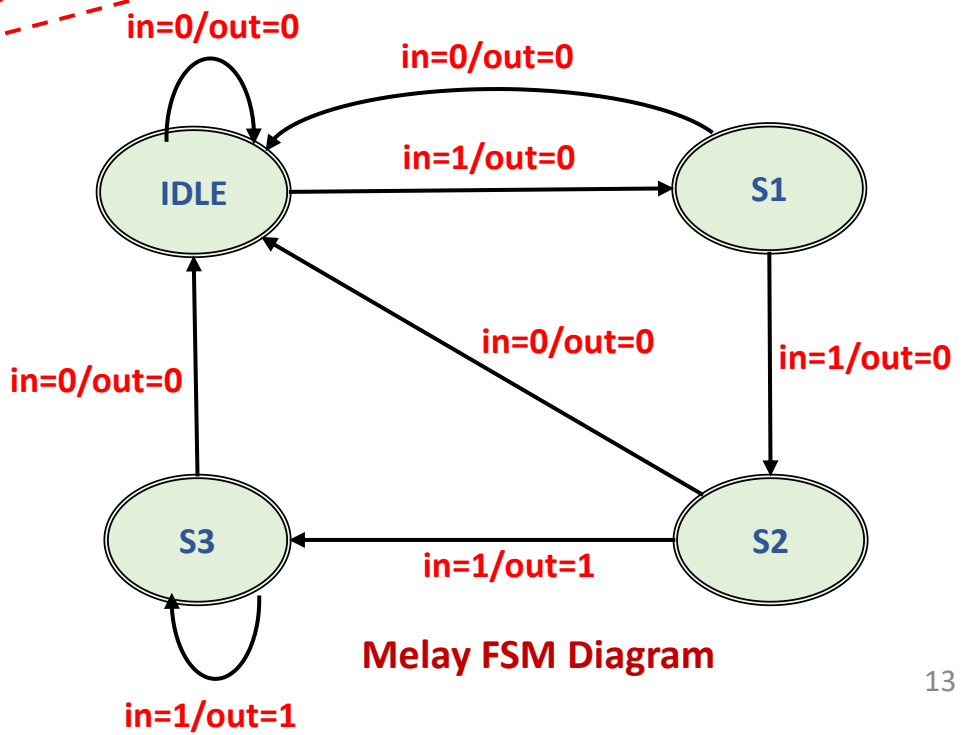
```
// Combination Logic for Next State and Output
always@(present_state, in) begin
 case(present_state)
  IDLE: begin
   if(in==1) begin
    next_state = S1;
    out = 0;
   end
   else begin
    next_state = IDLE;
    out = 0;
   end
  end
  S1: begin
   if(in==1) begin
    next_state = S2;
    out = 0;
   end
   else begin
    next_state = IDLE;
    out = 0;
   end
  end
 (continued on next page….)
```

both present_state and input "in" variable should be listed in sensitivity list

For next state and output combination logic, blocking assignments should be used

For Mealy FSM, output is set based on influence of both input "in" signal and present_state
Note : out = 0; statement is specified within if(in == 1) condition in mealy. And in moore, out = 0; is specified outside if(in == 1) condition





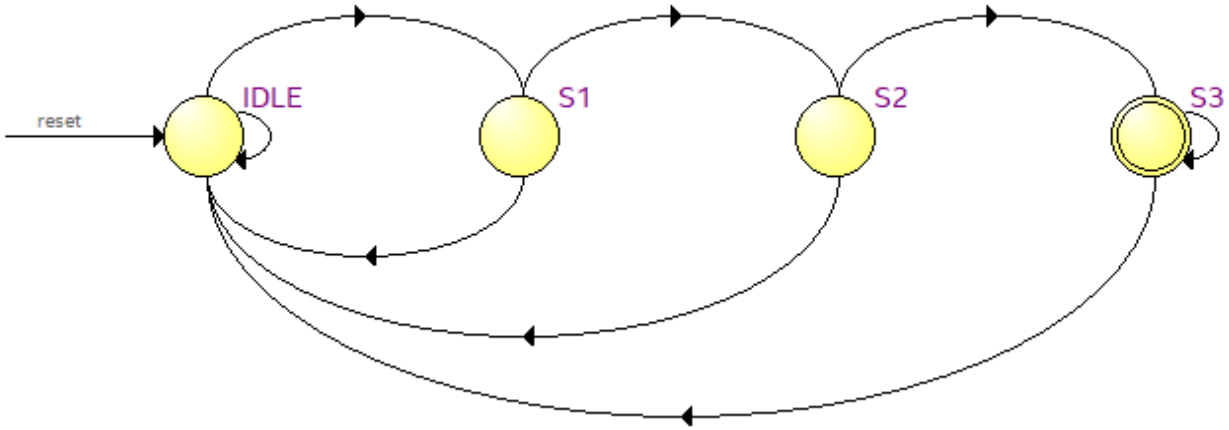
Melay FSM Diagram

Sequence Detector (Mealy FSM)

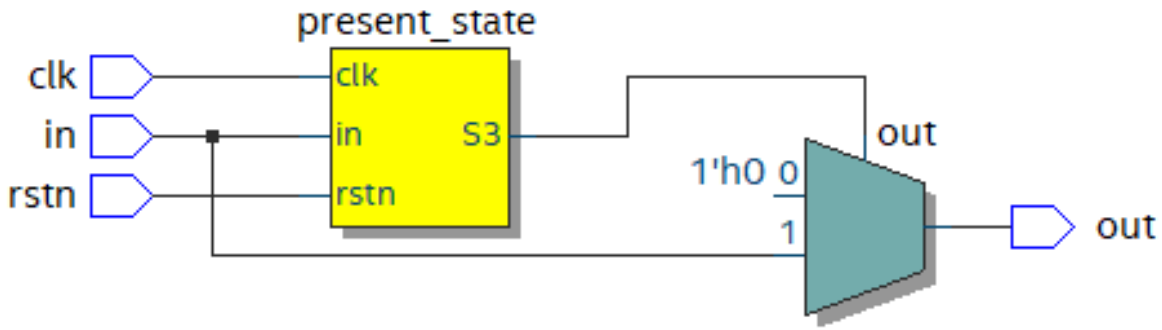
```
S2: begin
    if(in==1) begin
      next_state = S3;
      out = 1;
    end
    else begin
      next_state = IDLE;
      out = 0;
    end
 end
 S3: begin
    if(in==1) begin
      next_state = S3;
      out = 1;
    end
    else begin
      next_state = IDLE;
      out = 0;
    end
  default: begin out = 0; next_state = IDLE; end
 endcase
end
endmodule: sequence_detector_mealy
```

For each state, under else condition if "out=0" is not present then Synthesis compiler will create latch for the output "out" signal

default is specified in case a bad state is reached



Sequence Detector State machine diagram generated by Synthesizer

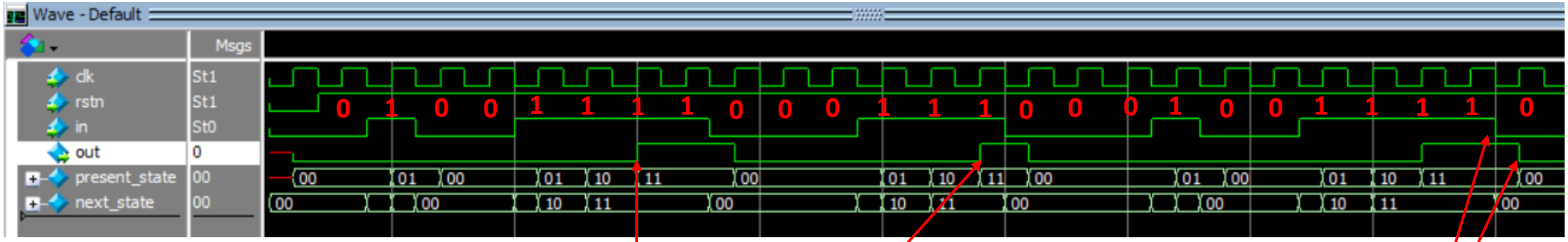


Sequence Detector RTL Netlist view generated by Synthesizer

Sequence Detector Simulation Snapshot(Moore FSM vs Mealy FSM)

Sequence Detector Moore FSM Simulation Snapshot

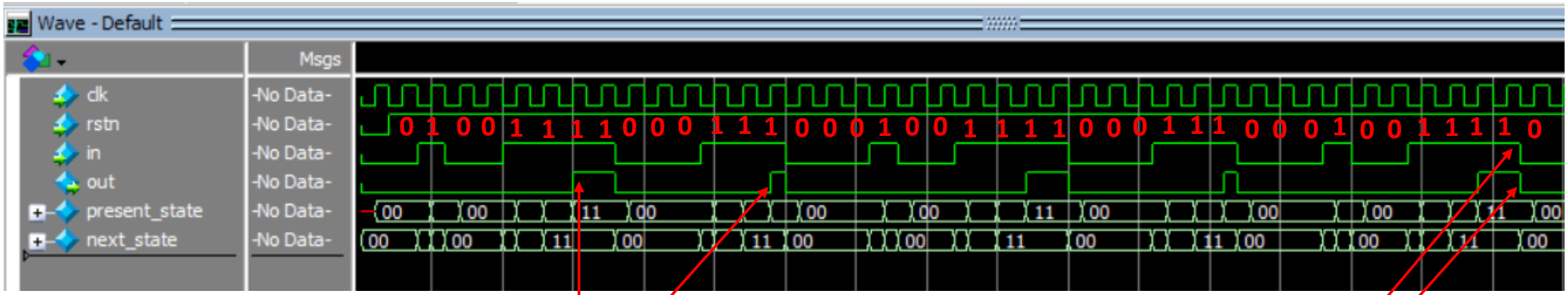
(Note : simulation result for 2 always vs 3 always block approach is same)



After three 1' detected out goes to '1'

Output did not react to change in input immediately in moore

Sequence Detector Mealy FSM Simulation Snapshot



After three 1' detected out goes to '1'

Output reacted to change in input immediately in mealy

Sequence Detector (One always block approach)

```
module sequence_detector_one_always_block(
 input logic clk, rstn,
 input logic in,
 output logic out);

 // Parameters to define FSM state encodings
 localparam [1:0] IDLE=2'b00,
     S1=2'b01, S2=2'b10, S3=2'b11;

 // Current state and next state variables
 logic[1:0] state;  // Does not need two separate state variable

 // Use of same clocked always block
always_ff@(posedge clk) begin
  if(!rstn)
    state <= IDLE;
    out <= 0;
  else
    case(state)
      IDLE: begin
        out <= 0;
        if(in==1) state <= S1;
        else state <= IDLE;
       end
```

Inputs are no longer asynchronously sampled. Change in input is captured with respect to clock edge event

Use of non-blocking assignment statement in case encoding branches

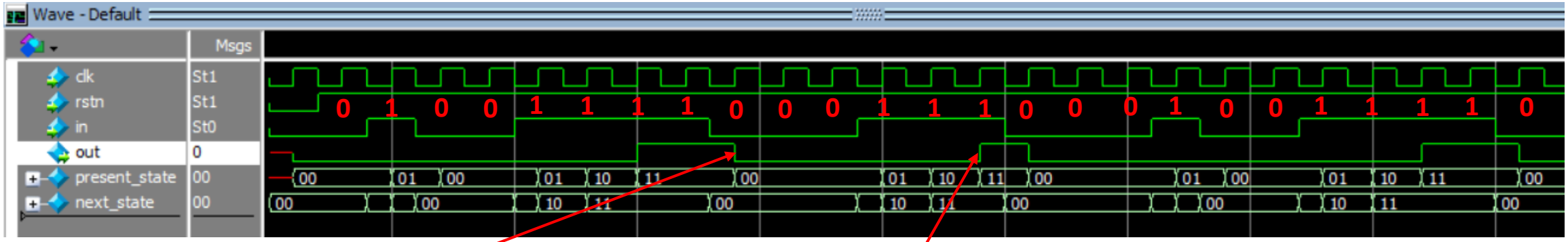
```
// Combination Logic for Next State and Output
 S1: begin
    out <= 0;
    if(in==1) state <= S2;
    else state <= IDLE;
  end
  S2: begin
    out <= 0;
    if(in==1) state <= S3;
    else state <= IDLE;
  end
  S3: begin
    out <= 1;
    if(in==1) state <= S3;
    else state <= IDLE;
  end
  default: begin
    out <= 0;
    state <= IDLE;
  end
 endcase
end
endmodule: sequence_detector_one_always_block
```

Output will stay asserted longer even when state has transitioned to another state where output should have gone back to reset or some other value

Sequence Detector Simulation Snapshot (One block vs Two block approach)

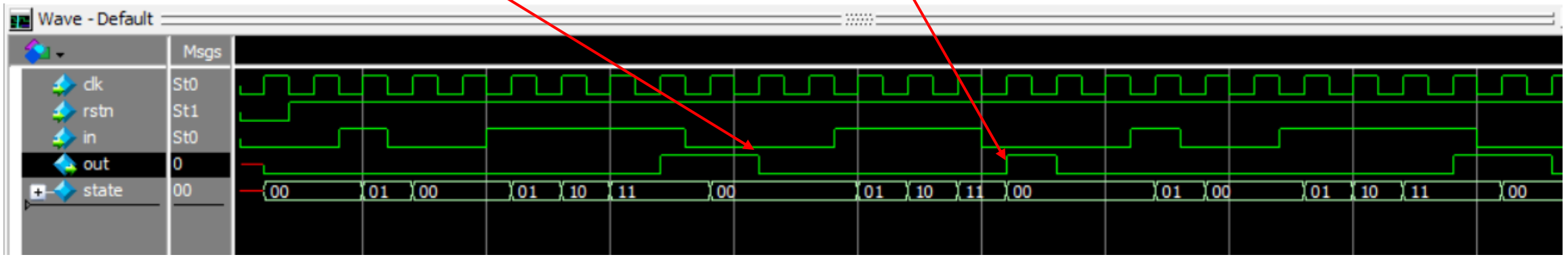
Sequence Detector Moore FSM Simulation Snapshot

(Note : simulation result for 2 always vs 3 always block approach is same)



In case of 1 always block implementation out = 1 stays for additional one additional clock cycle then 2 or 3 always block approach

out = 1 took additional one cycle in case of 1 always block approach compared to 2 or 3 always block implementation



Sequence Detector FSM Simulation Snapshot

(Note : simulation result for 1 always block

One Always Block Approach For FSM Modeling

- **One always block state machine is slightly more simulation time efficient than the two always block state machine :**
  - Since the inputs are only examined on clock changes (less work for simulator)

- **There some dis-advantages of one always block approach :**
  - RTL simulation does not model accurate gate level implementation
    - At the gate level combinational logic output updates when any input changes
    - In one process RTL simulation, combinational logic mixed with sequential logic, it is only evaluated at the clock edge
  - State machine can be more difficult to modify and debug since all sequential and combinational logic is mixed in one always block
  - Placing output assignments inside of the always block will infer output flip-flops.
    - This might lead to late availability of output. Sometimes not desirable in some applications.

- **Output assignments inside of a sequential always block cannot be Mealy outputs**
  - Hence cannot model Mealy FSM using one always block approach !

State Encoding

- **States in FSM are represented by encoded value. There are multiple choices:**
  - Binary encoding, One-hot encoding, Gray encoding, Johnson encoding and more.
  - Binary encoding and one-hot encoding are two commonly choices.

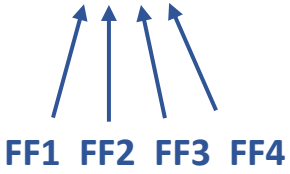
- **Binary encoding of states**
  - binary encoding For N states, use ceil($\log_2 \mathrm{N}$) bits to encode the state with each state represented by a unique combination of the bits.

  - Tradeoffs:
    - Most efficient use of state registers (less number of Flipflops are required)
    - Slower and more complicated combinational logic to detect when in a particular state.

  - Example :
    - In case of 4 states, 2 Flipflops are required to represent each state
    - **enum** logic [1:0] {RESET = 2'b00, WAIT = 2'b01, LOAD = 2'b10, DONE = 2'b11} next_state

State Encoding

- **One-hot encoding of states :**
  - For **N** states, use **N** bits to encode the state
    - bit corresponding to the current state is 1, all the others 0.
  - **Tradeoffs:**
    - Leads to simpler and Faster design. Much less combinational logic for state decoding
      - Good alternative when trying to optimize speed or to reduce power dissipation.
    - Can be costly in terms of FFs for FSMs with large number of states
    - FPGAs have larger number for Flipflops. Therefore one-hot state machine encoding is often a very appropriate with most FPGA targets
  - **Example :** In case of 4 states, 4 Flipflops are required for state encoding (one FF per state)
    - **STATE S1** : 4'b0001
    - **STATE S2** : 4'b0010
    - **STATE S3** : 4'b0100
    - **STATE S4** : 4'b1000



One-Hot FSM Implementation Example

```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state       <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit synthesis case = full, parallel
      state[IDLE] :   if (req)        next[BBUSY] = 1'b1;
                      else            next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)      next[BBUSY] = 1'b1;
                      else if ( dly)  next[BWAIT] = 1'b1;
                      else            next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)       next[BFREE] = 1'b1;
                      else            next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)        next[BBUSY] = 1'b1;
                      else            next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Index into the state register, not state encodings

Onehot requires larger declarations

Reset modification

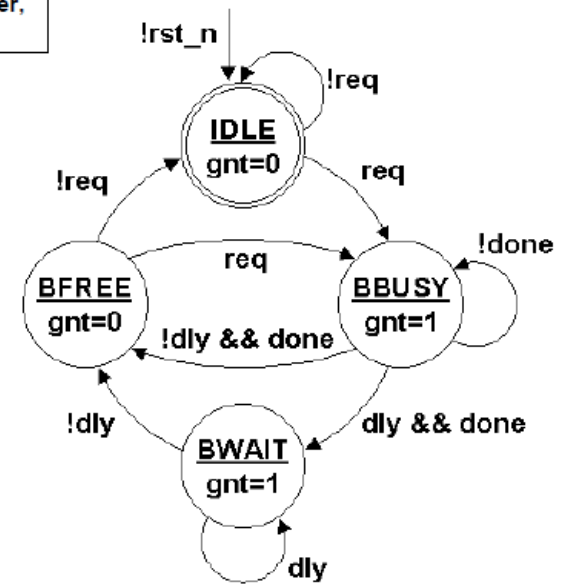
Must make all-0's assignment

case (1'b1)

Add "full" & "parallel" case

state[*current_state*] case items

Only update the next[*next state*] bit



Credit to : Clifford E. Cummings, Sunburst Designs

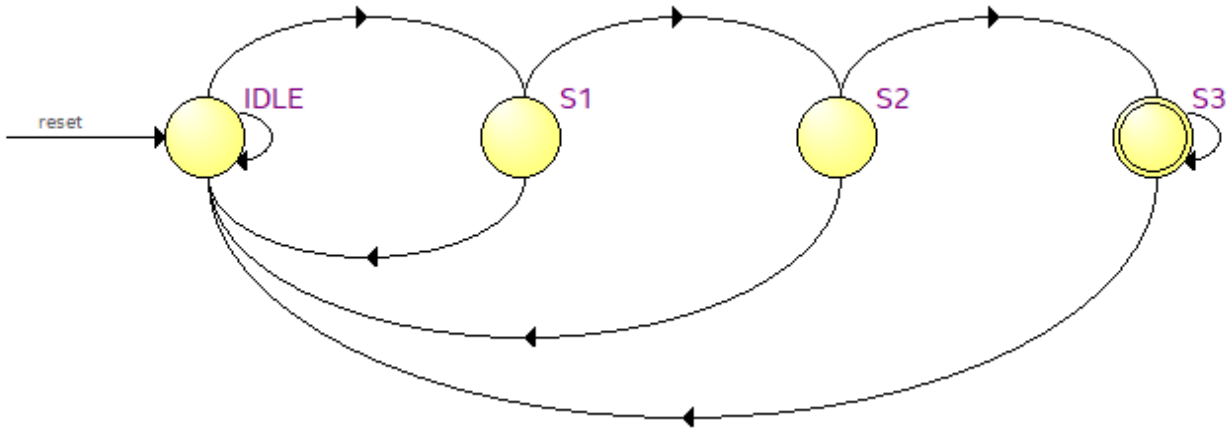
One Hot State Encoding

- **In case of one-hot encoding style FSM, Quartus Prime will not auto-generate FSM state diagrams !**
  - **Example:** if sequence detector state machine state encoding is changed from binary counting to one hot encoding style as described below, no state machine diagram generated however it will still work as an FSM

```
parameter[3:0]  IDLE=4'b0001, S1=4'b0010,  S2=4'b0100, S3=1000;
logic[3:0] present_state, next_state;
```

Sequence Detector State machine diagram will not be generated by Quartus Synthesizer

Functionally logic will still behave as a correct FSM !

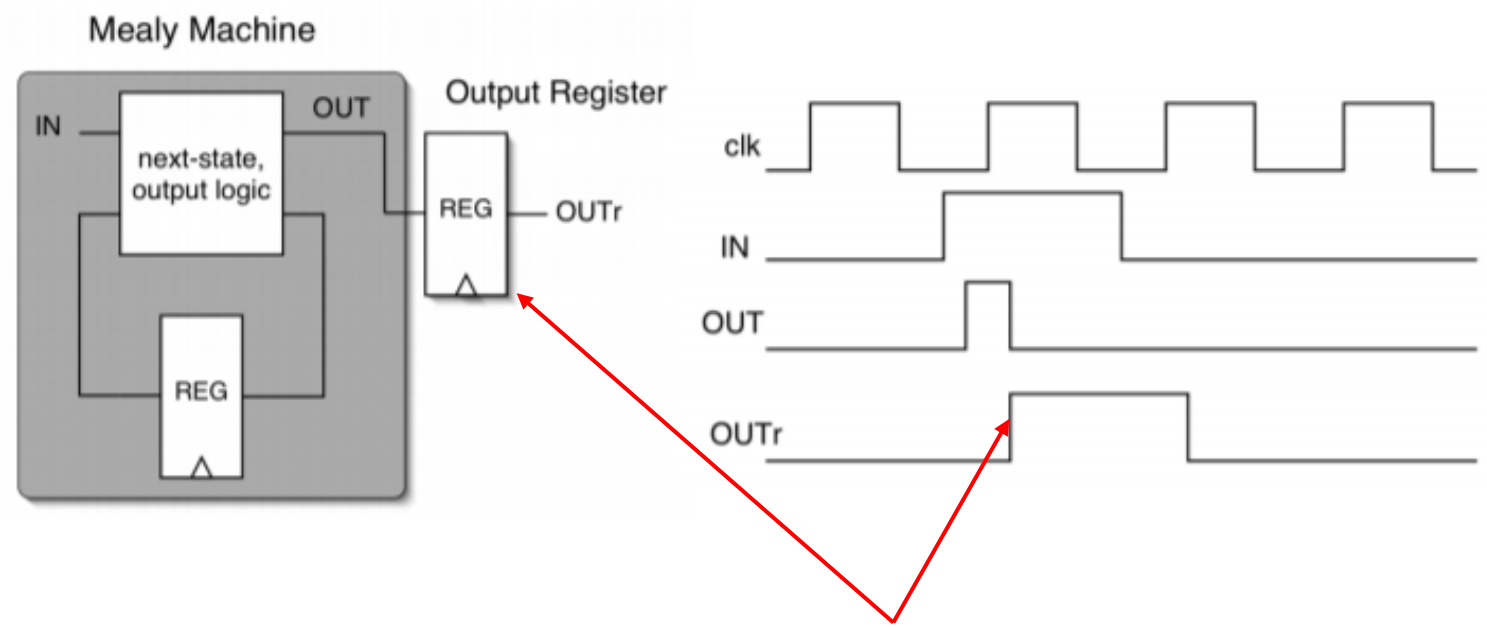


State Encoding Constants and Variables Declaration

- **There are multiple ways to declare FSM state encoding constants and variables :**
  - **Using enumeration :**
    - **enum logic[1:0]** {**WAIT**=2'b00, **EDGE=**2'b01} present_state, next_state;
  - **Using Parameter :**
    - **parameter** logic [1:0] **RESET** = 2'b00, **WAIT** = 2'b01, **LOAD** = 2'b10, **DONE** = 2'b11;
      logic [1:0] present_state, next_state;
  - **Using Local Parameter :**
    - **localparam** logic [1:0] **RESET** = 2'b00, **WAIT** = 2'b01, **LOAD** = 2'b10, **DONE** = 2'b11;
      logic [1:0] present_state, next_state;
  - **Using Using typedef : (Introduced in SystemVerilog. Not supported in Verilog)**
    - **typedef enum** logic [1:0] {**WAIT**=2'b00, **EDGE=**2'b01} **e_states**;
      e_states present_state, next_state;

Final Note on Moore vs Mealy FSM Modeling

- **A given state machine can have both Moore and Mealy style outputs.**
  - Nothing wrong with either implementation, but one has to be aware of the timing differences between the two types.
  - The output timing behavior of the Moore machine can be achieved in a Mealy machine by "**registering**" the Mealy output values
  - **Note : registering output or registered output means having flipflop at the output signal**



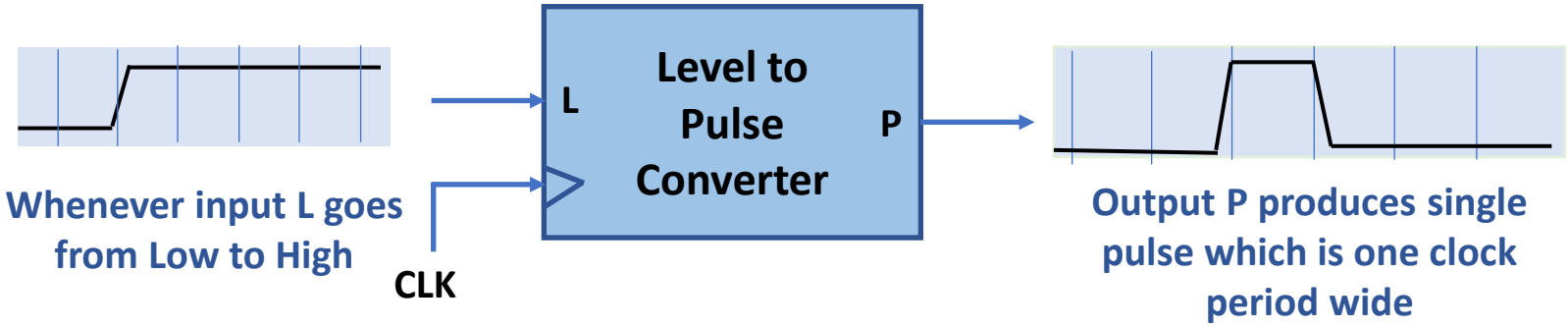
Registered Output of Mealy matches the output timing behavior of Moore machine

FSM Design Steps Using SystemVerilog

- **Specify circuit function**
- **Draw state transition diagram**
- **Minimize number of States using techniques (example : implication chart or row matching)**
- **Derive state transition table**
- **Determine next state and output function**
- **Assign encodings (bit patterns) to symbolic states**
- **Implement State Machine using SystemVerilog :**
  - Use either of the parameters/enum/localparam/typedef to represent encoded states.
  - Use separate always blocks for next state register assignment and combinational logic blocks
  - Use **always_comb** for all combinational block specification
  - Use **case** within combinational block (including default case item expression).
    - Within each **case** section assign all outputs and next state value based on inputs.
    - Ensure **default** case item is specified to avoid latches
- **Note:**
  - For Moore style machine make outputs dependent only on state not dependent on inputs.
  - Try not to mix up combinational logic and sequential logic inside same always block

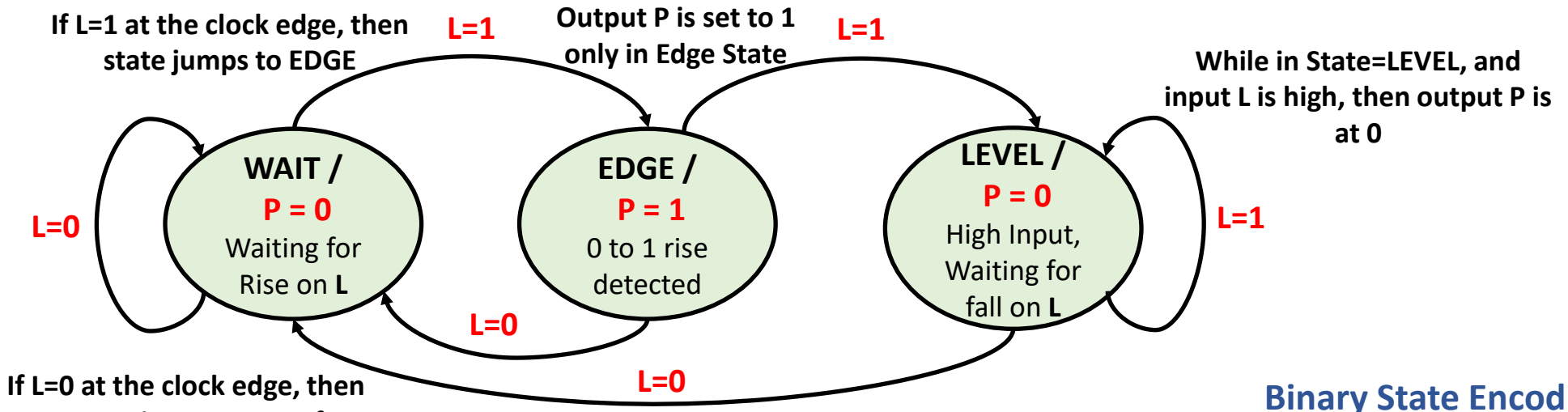
Level to Pulse Converter

- **A level-to-pulse converter produces a single cycle pulse each time its input goes high**
  - It is also known as synchronous rising edge detector
  - **Usage:** Pushing a button of a signal traffic light controller at pedestrian crossing
    - Pressing button for arbitrary period of time by pedestrian should generate single-cycle enable signals for counters in traffic light controller system



Level to Pulse Converter : Moore and Mealy State Diagrams

Moore State Transition Diagram

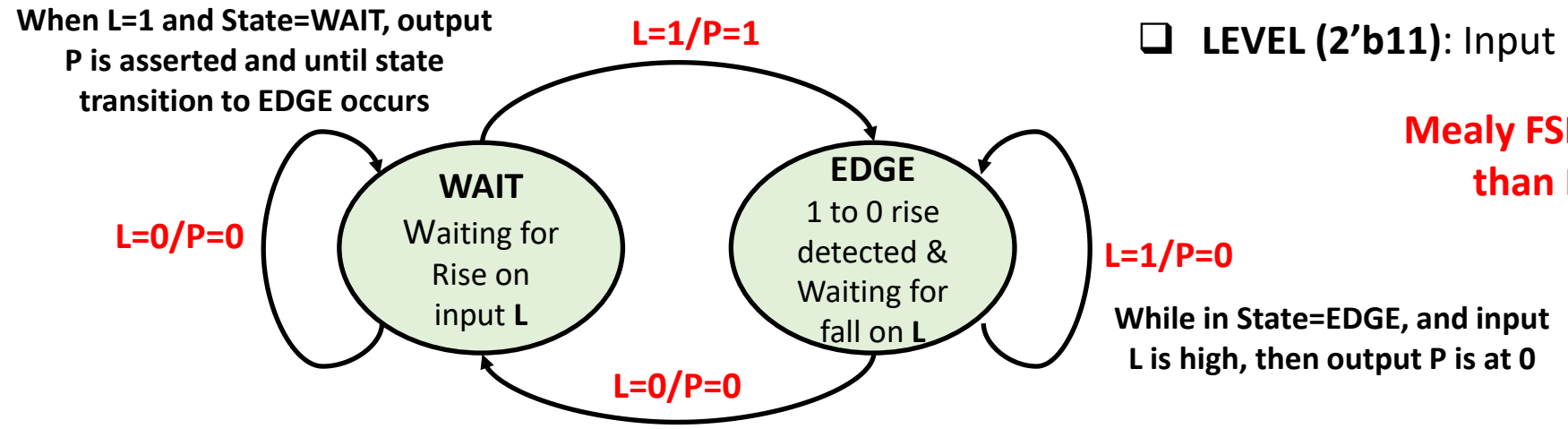


Binary State Encoding

- **WAIT (2'b00)** : Input is '0', wait for '1'
- **EDGE (2'b01)** : '0' to '1' edge detected on Input
- **LEVEL (2'b11)**: Input is stable at '1'

Mealy FSM has 1 less state than Moore FSM !!

Mealy State Transition Diagram



Level to Pulse Converter (Moore FSM)

```
module level_to_pulse_converter_moore(
 input logic clk, rstn,
 input logic L,
 output logic P);

// FSM state encodings and state registers declaration
enum logic[1:0] {WAIT=2'b00,
                 EDGE=2'b01,
                 LEVEL=2'b11} present_state, next_state;
```

State variables are declared as enumerated encoded logic

```
// Sequential Logic for present state
always_ff@(posedge clk) begin
 if(!rstn)
   present_state <= WAIT;
 else
  present_state <= next_state;
end
```

Synthesis will generate d-flipflop for present_state

For sequential logic Non-blocking assignments used

(continued….)

```
// Combination Logic for Next State and Output
always_comb begin
 case(present_state)
   WAIT: begin
     P = 0;
     if(L==1) next_state = EDGE;
     else next_state = WAIT;
   end
   EDGE: begin
     P = 1;
     if(L==1) next_state = LEVEL;
     else next_state = WAIT;
   end
   LEVEL: begin
     P = 0;
     if(L==1) next_state = LEVEL;
     else next_state = WAIT;
   end
   default: begin
     P = 0;  next_state = WAIT;
   end
 endcase
end
endmodule: level_to_pulse_converter_moore
```

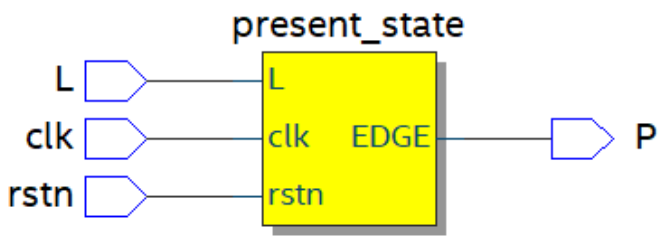
for next state combinational logic **always_comb** specified to automatically Infer sensitivity list

Output **P** is set to '1' as soon as rising edge on **P** is detected

Since output **P** requirement is single cycle pulse, **P** is set to '0' if input stays at level '1' after rising edge detection

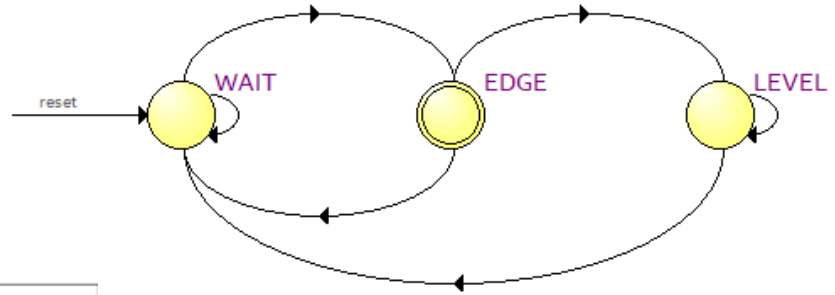
Level to Pulse Converter Simulation and Synthesis Results (Moore FSM)

Post Synthesis RTL Netlist Schematic



Resource	Usage
Estimated ALUTs Used	2
-- Combinational ALUTs	2
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	2

Moore State Diagram

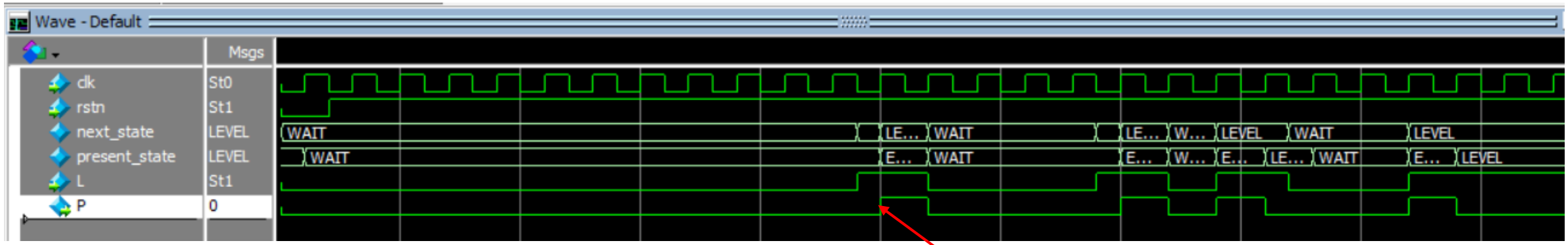


Mealy State Transition Table

	Source State	Destination State	Condition
1	EDGE	LEVEL	(L).(rstn)
2	EDGE	WAIT	(!L) + (L).(!rstn)
3	LEVEL	LEVEL	(L).(rstn)
4	LEVEL	WAIT	(!L) + (L).(!rstn)
5	WAIT	WAIT	(!L) + (L).(!rstn)
6	WAIT	EDGE	(L).(rstn)

2 Flipflops to represent three states WAIT EDGE, LEVEL

Simulation Waveform



Output **P** is single cycle pulse, available after somtime there is rising edge on input **L**

Level to Pulse Converter (Mealy FSM with Reduced States)

```
module level_to_pulse_converter_mealy(
 input logic clk, rstn,
 input logic L,
 output logic P);

// FSM state encodings and state registers declaration
enum logic[1:0] WAIT=2'b00,
                EDGE=2'b01} present_state, next_state;
```

Only two states required for Mealy compared to Moore FSM

```
// Sequential Logic for present state
always_ff@(posedge clk) begin
 if(!rstn)
   present_state <= WAIT;
 else
  present_state <= next_state;
end
```

(continued….)

```
// Combination Logic for Next State and Output
 always_comb begin
 case(present_state)
   WAIT: begin
     if(L==1) begin
       next_state = EDGE; P = 1;
     end
     else begin
       next_state = WAIT;  P = 0;
     end
   end
   EDGE: begin
     if(L==1) begin
       next_state = EDGE; P = 0;
     end
     else begin
       next_state = WAIT; P = 0;
     end
   end
   default: begin P = 0; next_state = WAIT; end
 endcase
end
endmodule: level_to_pulse_converter_mealy
```

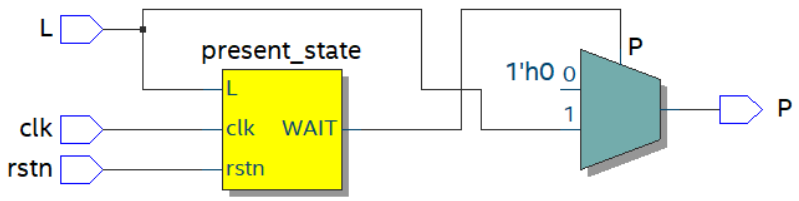
for next state combinational logic **always_comb** specified to automatically Infer sensitivity list

Output **P** is set to '1' as soon as input L is detected as '1' in **WAIT** state

Since output **P** requirement is single cycle pulse, **P** is set to '0' if input stays at level '1' after rising edge detection

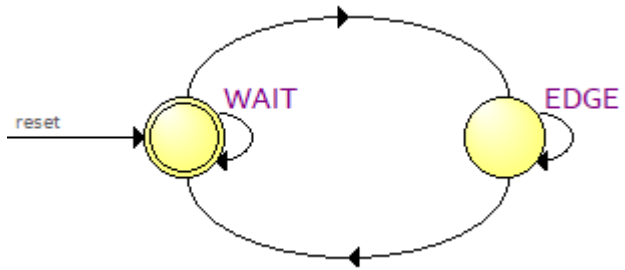
Level to Pulse Converter Simulation and Synthesis Results (Mealy FSM)

Post Synthesis RTL Netlist Schematic



Resource	Usage
Estimated ALUTs Used	2
-- Combinational ALUTs	2
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	1

Moore State Diagram

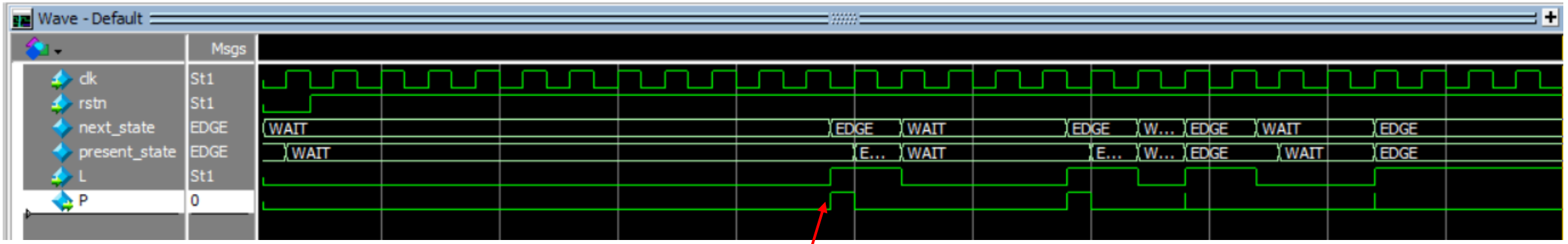


Mealy State Transition Table

	Source State	Destination State	Condition
1	EDGE	EDGE	(L).(rstn)
2	EDGE	WAIT	(!L) + (L).(!rstn)
3	WAIT	EDGE	(L).(rstn)
4	WAIT	WAIT	(!L) + (L).(!rstn)

1 Flipflop to represent two states WAIT and EDGE.
1 less D-flipflop required in Mealy compared to Moore FSM

Simulation Waveform



Output **P** is available as soon as input **L** changed from '0' to '1'. However output **P** pulse is not stable for 1 cycle. Which does not meet design requirement for **P** to be 1 cycle pulse !!

Level to Pulse Converter (Mealy FSM with Reduced States + Registered output)

```
module level_to_pulse_converter_mealy(
 input logic clk, rstn,
 input logic L,
 output logic P);

 // FSM state encodings and state registers declaration
 enum logic[1:0] {WAIT=2'b00,
                  EDGE=2'b01} present_state, next_state;
```

Only two states required for Mealy compared to Moore FSM

```
 logic r_P; // local variable declaration

 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if(!rstn) begin
    present_state <= WAIT;
    P <= 0;
  end
  else begin
   present_state <= next_state;
   P <= r_P;
  end
 end
```

(continued….)

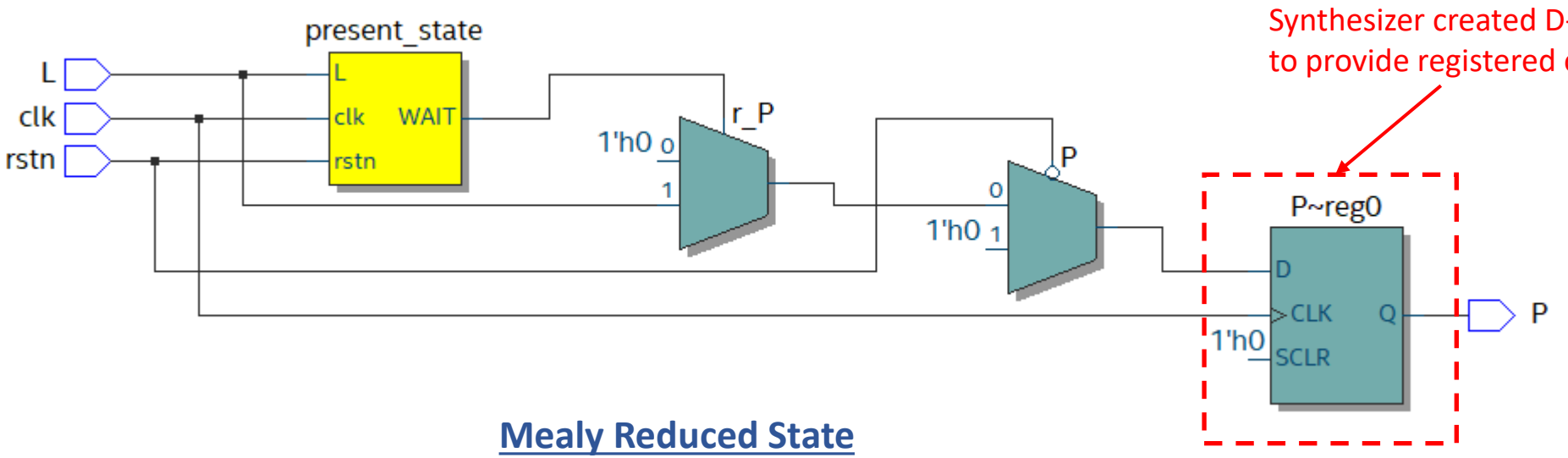
Synthesizer will create D-flipflop to provide registered output **P.**
Output P is registered to ensure P is at least 1 cycle pulse

```
 // Combination Logic for Next State and Output
  always_comb begin
  case(present_state)
    WAIT: begin
      if(L==1) begin
        next_state = EDGE; r_P = 1;
      end
      else begin
        next_state = WAIT;  r_P = 0;
      end
    end
    EDGE: begin
      if(L==1) begin
        next_state = EDGE; r_P = 0;
      end
      else begin
        next_state = WAIT; r_P = 0;
      end
    end
    default: begin r_P = 0; next_state = WAIT; end
  endcase
 end
endmodule: level_to_pulse_converter_mealy
```

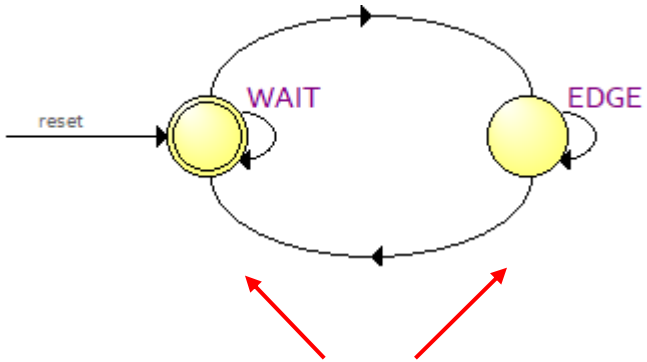
Output is assigned to local variable **r_P**

Level to Pulse Converter Synthesis Results (Mealy FSM) : With Registered Output

Post Synthesis RTL Netlist Schematic



Mealy Reduced State Diagram



Only 2 states in Mealy FSM compared to Moore which as 3 States

Mealy Reduced State Transition Table

	Source State	Destination State	Condition
1	EDGE	WAIT	(!L) + (L).(!rstn)
2	EDGE	EDGE	(L).(rstn)
3	WAIT	WAIT	(!L) + (L).(!rstn)
4	WAIT	EDGE	(L).(rstn)

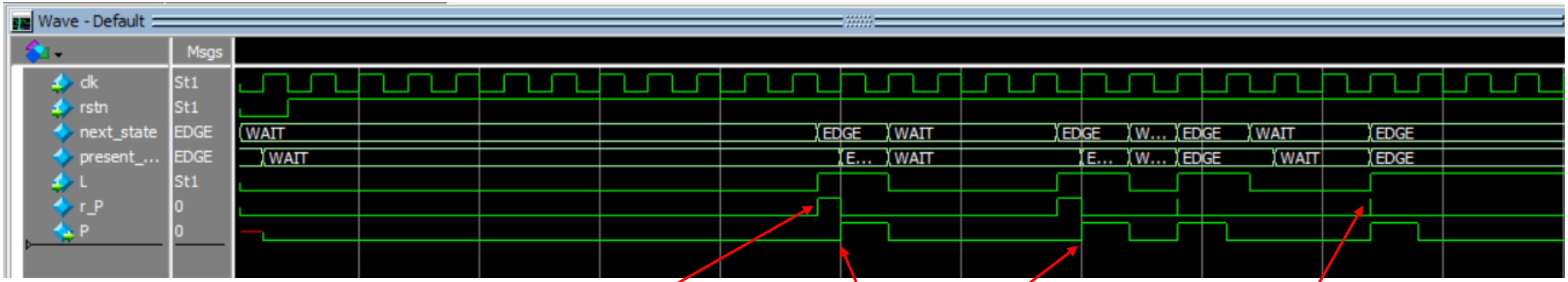
Post Synthesis Resource Utilization

Resource	Usage
Estimated ALUTs Used	2
-- Combinational ALUTs	2
-- Memory ALUTs	0
-- LUT_REGs	0
Dedicated logic registers	2

1 Flipflop for registering output **P** and 1 Flipflop for representing two states WAIT and EDGE

Level to Pulse Converter Simulation Result(Mealy FSM) : With Registered Output

Post Synthesis RTL Netlist Schematic



Unregistered output **r_P** is available as soon as input **L** changed from '0' to '1'

Registered Output **P** which is single cycle pulse when there is rising edge on **L** detected.

Glitches in Simulation on unregistered output **r_P**

References

- **For FSM design and synthesis coding guidelines refer to below mentioned white paper :**
  - http://www.sunburst-design.com/papers/CummingsSNUG1998SJ_FSM.pdf