Verilog Tutorial
Sequential Logic & FSM

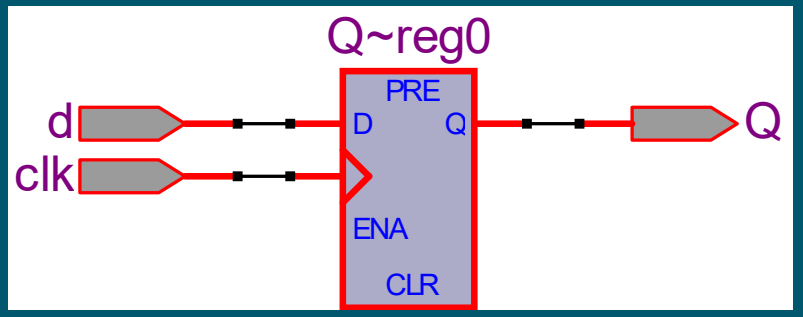
Shutha Pulendran

October 24, 2016

Verilog Recap

- Verilog is a hardware description language
- Module - describes logic circuit
- Wire - an internal connection within a module
- Top-level module - the largest module that contains all of the circuits
- Module instantiation - used for multiple submodules
- Verilog is case sensitive
- Defining constants and bus structures
- Behavioural Verilog using `always` block
  - Using `if … else` statements
  - Using `case` statements
  - Using `reg` for always block output

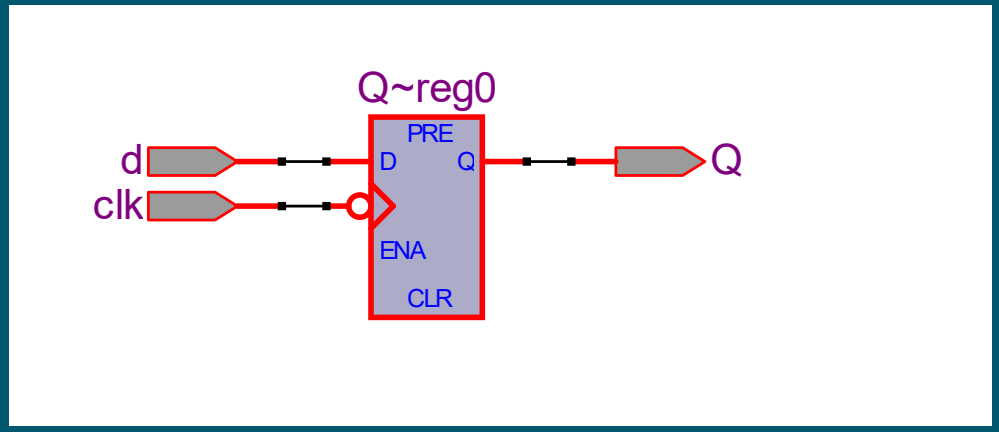
Positive-Edge Trigged D Flip-Flop




```
module d_flipflop_pos(input clk, input d, output reg Q);
        always @(posedge clk) //Notice the keyword posedge
                begin
                        Q <= d; // Notice the operator <=
                end
endmodule
```

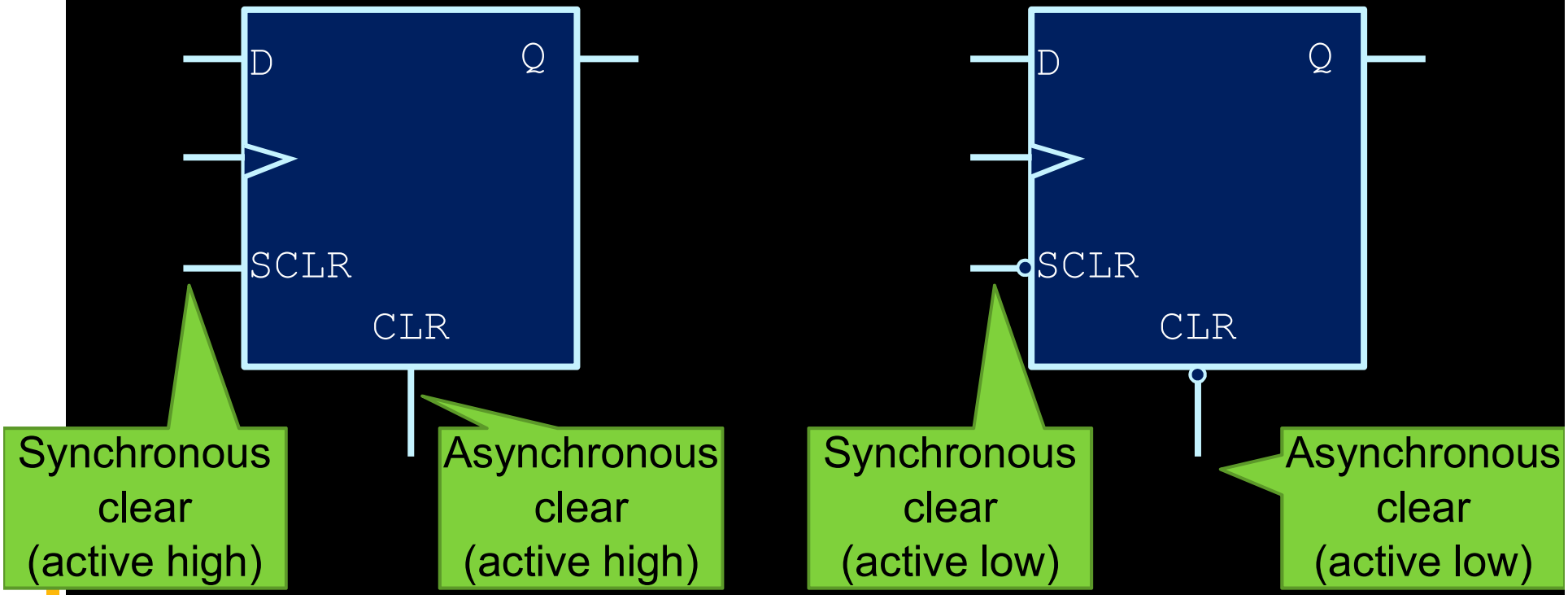
- Use **<=** for edge-triggered `always` block
- Use **=** for `always(*)` block

Negative-Edge Triggered D Flip-Flop



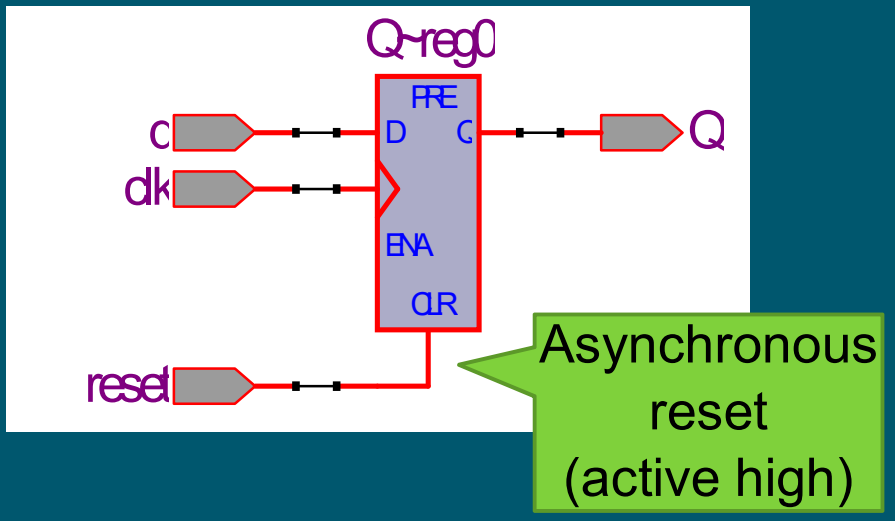

```
module d_flipflop_neg(input clk, input d, output reg Q);
        always @(negedge clk) //notice the keyword negedge
                begin
                        Q <= d;
                end
endmodule
```

Review - Reset Notation



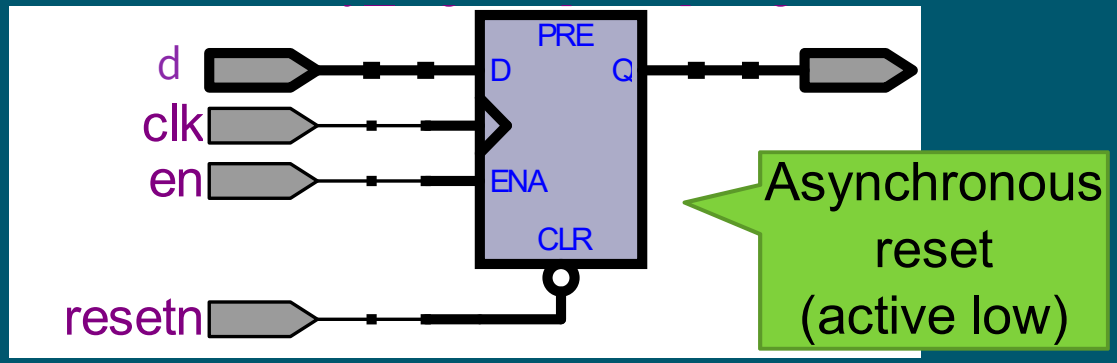
from course notes

D Flip-Flop with Asynchronous Reset



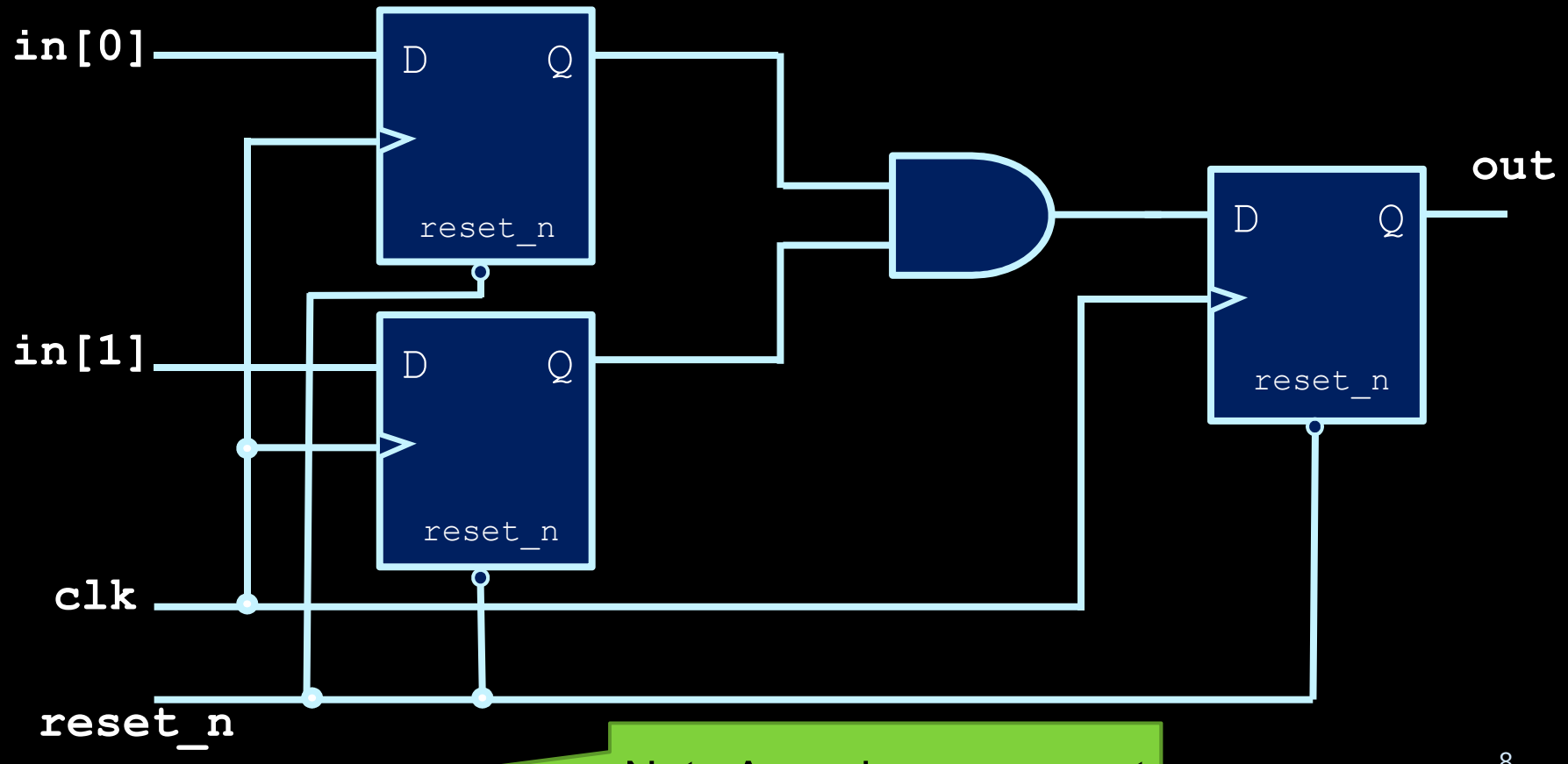

```
module d_flipflop_reset(input clk, input d, input reset,
output reg Q);
        always @(posedge clk or posedge reset)
                begin
                        if (reset == 1'b1)
                                Q <= 1'b0;
                        else
                                Q <= d;
                end
endmodule
```

D Flip-Flop with Enable and Asynchronous reset




```
module d_flipflop_enable(input clk, input d, input reset, input
en, output reg Q);
      always @(posedge clk, negedge reset)
            begin
                  if (reset == 1'b0)
                        Q <= 1'b0;
                  else if (en = 1'b1)
                        Q <= d;
                  else
                      Q <= Q; // These two lines are not needed
            end
endmodule
```

Verilog Code ?



Note Asynchronous reset (active low)

Solution 1 - Using Module Instantiation

```
module main_circuit1 (input clk, input reset_n, input [1:0] in, output out);

  wire q1, q2, and_out;

  d_ff ff1(.clk(clk), .d(in[0]),   .reset_n(reset_n), .Q(q1));
  d_ff ff2(.clk(clk), .d(in[1]),   .reset_n(reset_n), .Q(q2));
  d_ff ff3(.clk(clk), .d(and_out),.reset_n(reset_n), .Q(out));

  assign and_out = q1 & q2;

endmodule
```

```
module d_ff(input clk, input d, input reset_n, output reg Q);

always @(posedge clk, negedge reset_n)
        begin
          if (reset_n == 1'b0)
                Q <= 1'b0;
          else
                Q <= d;
          end

endmodule
```

Solution 2 – Using two always block

```
module main_circuit2 (input clk, input [1:0] in, input reset_n, output reg out);

reg [1:0] Q;  // For the output of the first two flipflops
wire and_out; // for the and gate output

  always @ (posedge clk, negedge reset_n)  // the first two flipflops
      begin
         if(reset_n==1'b0)
                  Q <= 2'b00;
         else
                  Q <= in;

      end

  always @ (posedge clk, negedge reset_n) // the last flipflop
      begin
         if(reset_n==1'b0)
                  out <= 1'b0;
         else
                  out <= and_out;

      end

  assign and_out = Q[0] & Q[1];

endmodule
```

Solution 3 – Using one always block

```
module main_circuit3 (input clk, input [1:0] in, input reset_n, output reg out);

reg [1:0] Q;  // For the output of the first two flipflops
wire and_out; // for the and gate output

  always @ (posedge clk or negedge reset_n)  // Implement all flipflops in one
always block. Note all flipflops have the same clock and reset_n signals
      begin
         if(reset_n==1'b0)
                  begin
                      Q <= 2'b00;
                      out <= 1'b0;
                  end
         else
                  begin
                      Q <= in;
                      out <= and_out;
                  end
      end

  assign and_out = Q[0] & Q[1];

endmodule
```

Solution 3 – Another Variation

```
module main_circuit4 (input clk, input [1:0] in, input reset_n, output out);

reg [2:0] Q;  // For the output of all flipflops
wire and_out; // for the and gate output

  always @ (posedge clk, negedge reset_n)  // Implement all flipflops in one
always block
      begin
         if(reset_n==1'b0)
                  Q <= 3'b000;
         else
                  Q <= {and_out, in};
      end

  assign out = Q[2];
  assign and_out = Q[0] & Q[1];

endmodule
```

Combinational Vs. Sequential

Combinational Logic

- Use `always@(*)` block
- Use blocking assignments using = operator
- All cases must be specified inside the `always` block, e.g., `case` statement must contain a `default` statement

Sequential Logic

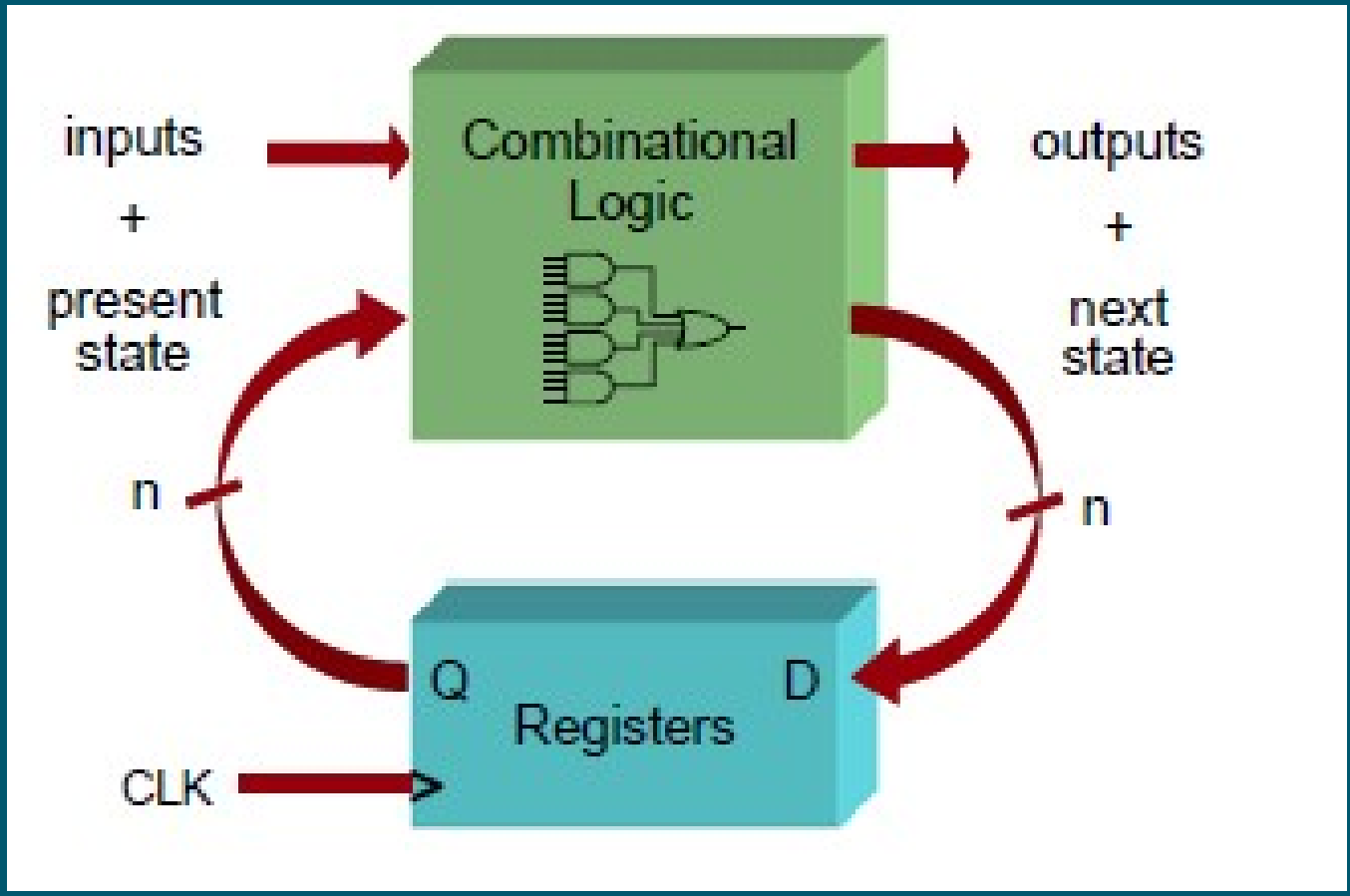
- Use edge-triggered `always` block, i.e., always@(posedge ..) , always@(negedge ...)
- Use non-blocking assignments using <= operator

ORDERING OF *always* blocks DOES NOT MATTER!!!

Important Verilog Tips

- Don't use `posedge` and `negedge` for the same signal inside an `always` block, e.g., `always@(posedge` clk, negedge clk)
- Don't define multiple `always` blocks assigning values to the same `reg`
- Don't mix blocking (=) and nonblocking (<=) assignments within the same `always` block
- You can use spacers for constants for clarity, e.g., 8'b1001_0001

Review - Finite State Machines



Source: MIT Course Notes, 2012

FSM Design Procedure

1. Draw state diagram and state transition table
2. State Assignment
3. Minimize logic and circuit design (not required for some type of Verilog implementation)
4. Implement the design in Verilog

Example: Vending machine FSM

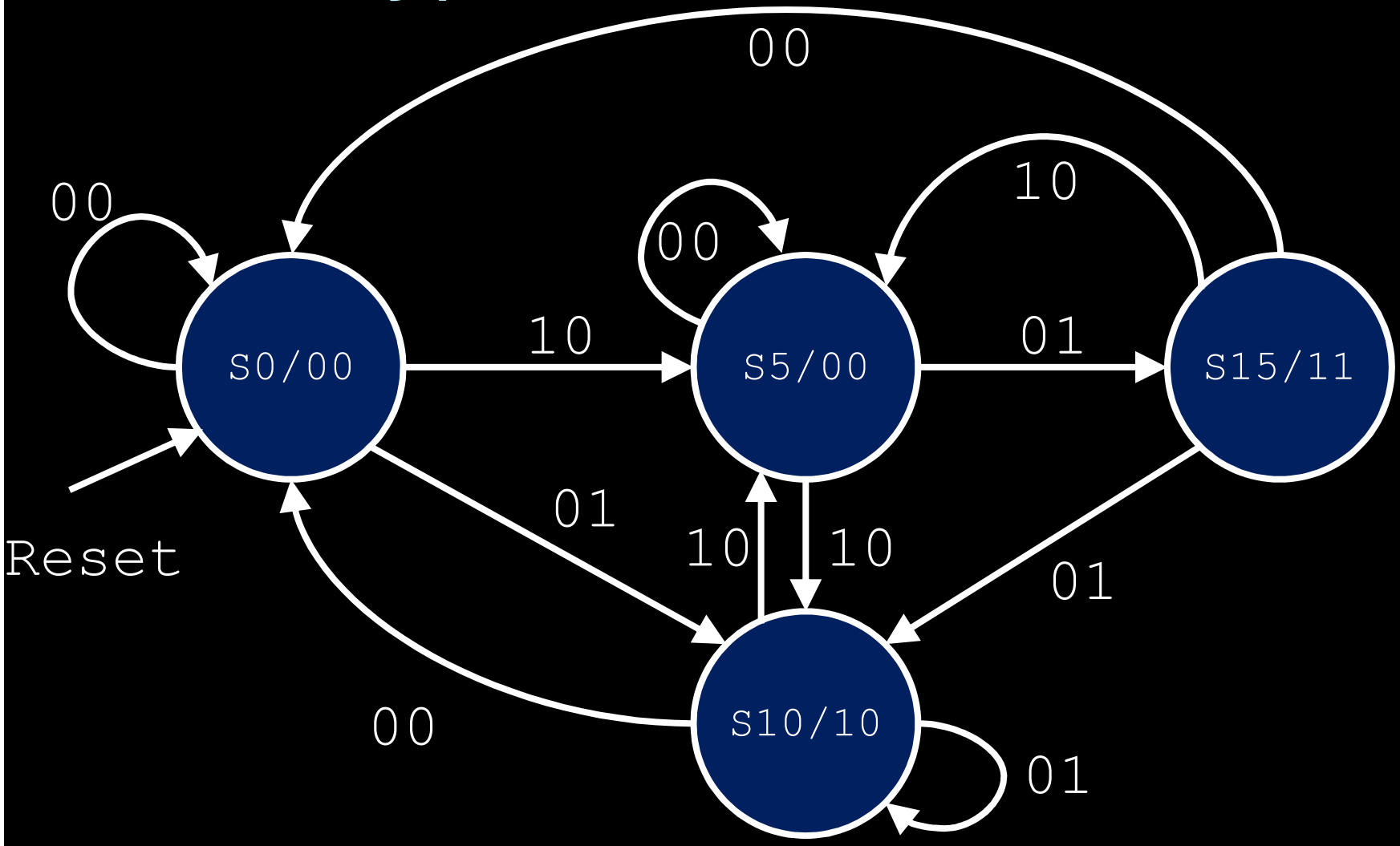
- Let's build a simple vending machine
  - User can put in a nickel (5¢) or a dime (10¢)
  - The cost of the pop is 10¢
  - Dispense the pop when the amount entered is sufficient (10¢ or more), and produce appropriate change if necessary.

Inputs and Outputs

- Inputs: {Nickel (N), Dime (D)}
  - N=1 for Nickel, input -> 10
  - D=1 for Dime, input -> 01
  - Both N and D will never be 1 at the same time (Don't care)
- Outputs: {Dispensing the pop (P), change (C)}
  - P=1 will trigger mechanism for dispensing pop
  - C=1 will trigger mechanism for dispensing change
  - Dispense pop only , output -> 10
  - Dispense change only, output -> 01
  - Dispense both , output -> 11

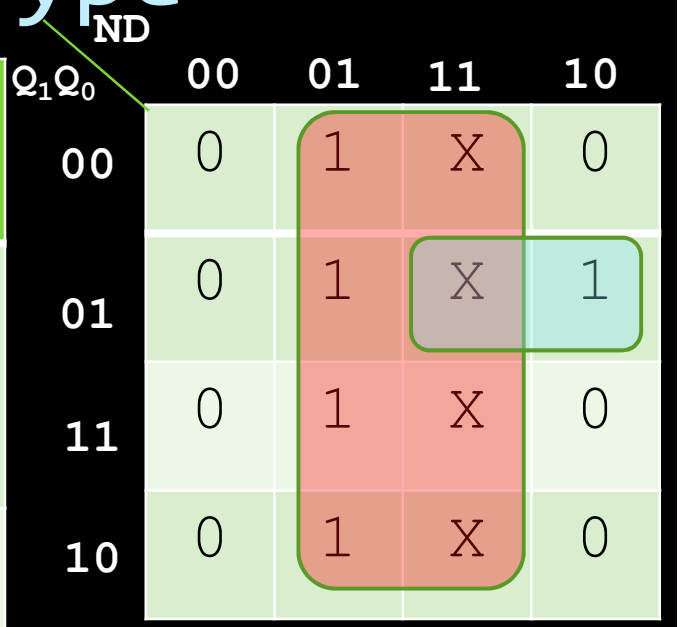
State Diagram
Moore Type

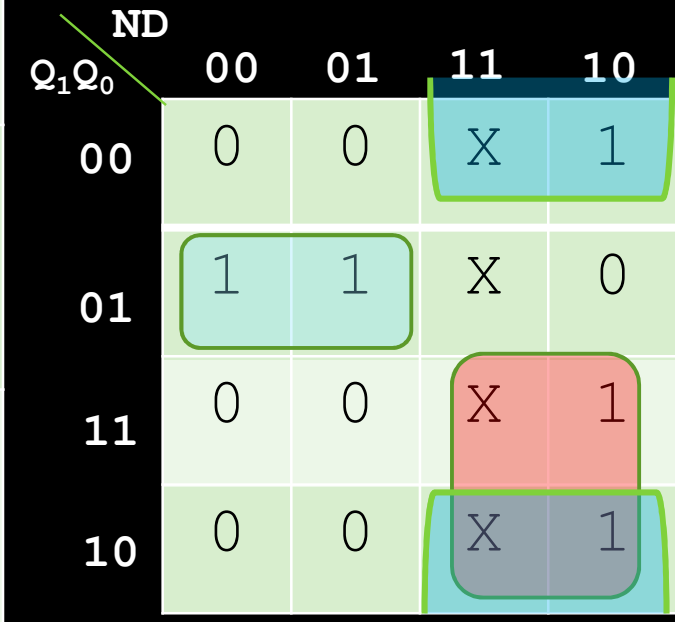
Inputs: {N, D}
Nickel =5¢, Dime = 10¢
Outputs: {P, C}



State Table – Moore Type

Present State (Q_1 Q_0)	Input {N,D}	Next State (D_1 D_0)	Output {P,C}
S0 (00)	00	S0 (00)	00
	01	S10 (10)	
	10	S5 (01)	
S5 (01)	00	S5 (01)	00
	01	S15 (11)	
	10	S10 (10)	
S10 (10)	00	S0 (00)	10
	01	S10 (10)	
	10	S5 (01)	
S15 (11)	00	S0 (00)	11
	01	S10 (10)	
	10	S5 (01)	





Logic Expressions

$$D_1 = D + NQ_0\overline{Q_1}$$

$$D_0 = NQ_1 + N\overline{Q_0} + \overline{N}\,\overline{Q_1}Q_0$$

Next States

$$P = Q_1$$

$$C = Q_1Q_0$$

Outputs

Verilog - Solution 1

```
module vending_machine(input N, input D, input reset_n, input clk,
output P, output C);
wire [1:0] next_state;
reg [1:0] present_state;

always @ (posedge clk, negedge reset_n)  // Implement flipflops in one
always block.
      begin
         if(reset_n==1'b0)
                  present_state <= 2'b00;
         else
                  present_state <= next_state;
      end

// Implement combinational logic (logic expressions)
assign next_state[1] = D | (N & present_state[0] & ~ present_state[1]);
assign next_state[0] = (N & present_state[1]) | (N & ~present_state[0])
                     | (~N & present_state[1] & present_state[0]);

assign P = present_state[1];
assign C = present_state[1] & present_state[0];

endmodule
```

Solution 2

```
module vending_machine2(input N, input
D, input clk, input reset_n, output P,
output C);
reg [1:0] next_state, present_state;
//Define states and state names
localparam [1:0] S0 = 2'b00, S5 =
2'b01 , S10 = 2'b10, S15 = 2'b11;

//Implement next states (combinational)
always @ (*)
begin
case (present_state)
  S0: begin
      if ({N,D}== 2'b00)
          next_state = S0;
      else if ({N,D}== 2'b01)
          next_state = S10;
      else
          next_state = S5;
      end
  S5: begin
       if ({N,D}== 2'b00)
          next_state = S5;
       else if ({N,D}== 2'b01)
          next_state = S15;
       else
          next_state = S10;
       end
```

```
S10: begin
       if ({N,D}== 2'b00)
          next_state = S0;
       else if ({N,D}== 2'b01)
          next_state = S10;
       else
          next_state = S5;
       end
S15: begin
        if ({N,D}== 2'b00)
          next_state = S0;
        else if ({N,D}== 2'b01)
          next_state = S10;
        else
          next_state = S5;
       end
endcase //end of case
end  //end of always@(*)

// Implement flip-flops (sequential)
always @(posedge clk, negedge reset_n)
       begin
          if(reset_n==1'b0)
            present_state <= S0;
          else
           present_state <= next_state;
       end
//Note ordering of combinational and
sequential always blocks does not
matter
```

Solution 2 – Cont'd

```
//specify the outputs
assign P = (present_state == S15) | (present_state == S10) ;
assign C = (present_state == S15);

endmodule
```

```
//specify the outputs (alternative method)
always @ (*)
 begin
      case(present_state)
           S0: {P,C}= 2'b00;
           S5: {P,C}= 2'b00;
          S10: {P,C}= 2'b10;
          S15: {P,C}= 2'b11;
      endcase
 end
endmodule
```

How did we handle the don't cares when input ({N,D}) is 2'b11 ?

Stretch Your Thinking

- Handling don't cares when {N,D} = 2'b11

```
S15: begin
       if ({N,D}== 2'b00)
         next_state = S0;
       else if ({N,D}== 2'b01)
         next_state = S10;
       else
         next_state = S5;
      end
```

```
S15: begin
       if ({N,D}== 2'b00)
         next_state = S0;
       else if ({N,D}== 2'b01)
         next_state = S10;
       else if ({N,D}== 2'b10)
         next_state = S5;
       else
         next_state = S0; // assign
to initial state (reset state)
      end
```

```
localparam [1:0] SX = 2'bxx; //
define don't care state
S15: begin
       if ({N,D}== 2'b00)
         next_state = S0;
       else if ({N,D}== 2'b01)
         next_state = S10;
       else if ({N,D}== 2'b10)
         next_state = S5;
       else
         next_state = SX; // assign
to don't care state
      end
```

Is SX really an additional state?

Additional Coding Styles

```
//Example shown before
S15: begin
       if ({N,D}== 2'b00)
         next_state = S0;
       else if ({N,D}== 2'b01)
         next_state = S10;
       else if ({N,D}== 2'b10)
         next_state = S5;
       else
         next_state = S0;
      end
```

```
// Using case statements
S15: begin
          case ({N,D})
           2'b00 : next_state = S0;
           2'b01 : next_state = S10;
           2'b10 : next_state = S5;
          default: next_state = S0;
         endcase
      end
```

```
// Using Conditional operator
S15: begin
       next_state =  ({N,D}== 2'b00)? S0:
                     (({N,D}== 2'b01)? S10:
                     (({N,D}== 2'b10)? S5: S0));
     end
```

Summary

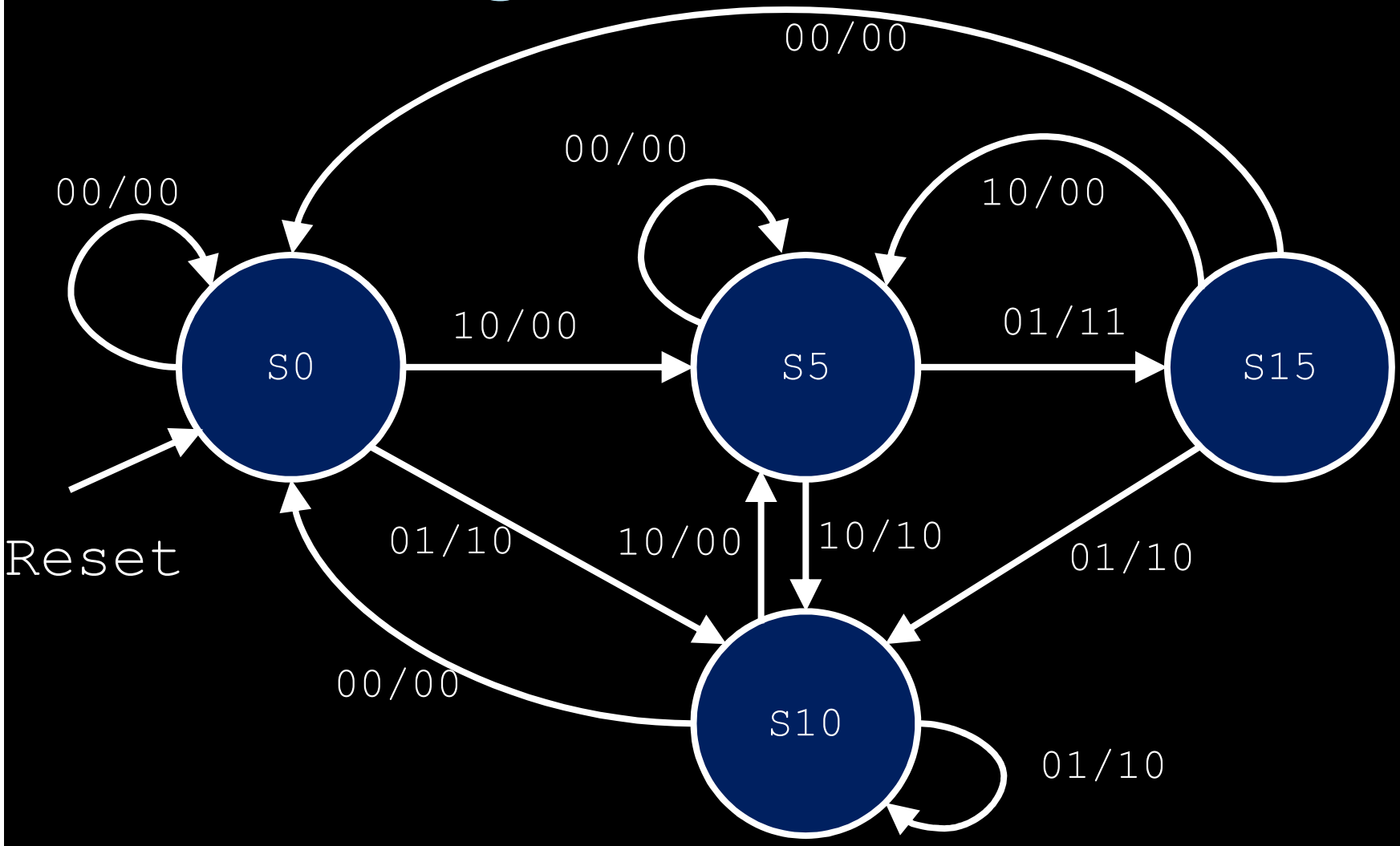
- Combinational `always` block Vs. Sequential `always` block
- FSM design example
  - Assigning and using state labels
  - Handling don't care states
  - Different coding styles

Appendix

Mealy Type FSM for the Vending Machine

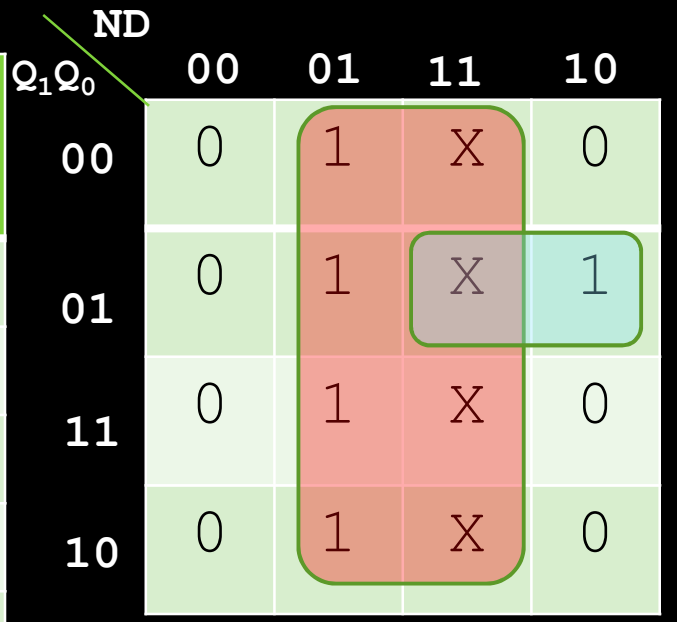
Mealy Type State Diagram

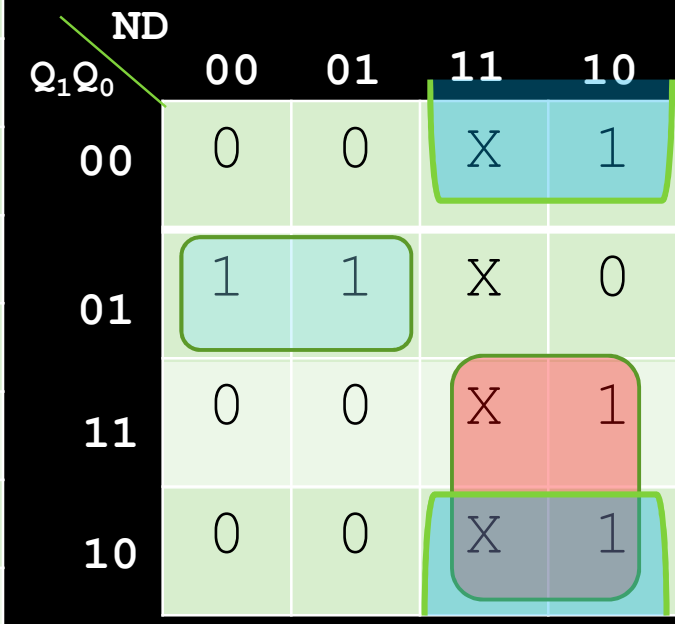
Inputs: {N, D}
Nickel =5¢, Dime = 10¢
Outputs: {P, C}



State Table (Mealy)

Present State (Q_1 Q_0)	Input {N,D}	Next State (D_1 D_0)	Output {P,C}
S0 (00)	00	S0 (00)	00
	01	S10 (10)	10
	10	S5 (01)	00
S5 (01)	00	S5 (01)	00
	01	S15 (11)	11
	10	S10 (10)	10
S10 (10)	00	S0 (00)	00
	01	S10 (10)	10
	10	S5 (01)	00
S15 (11)	00	S0 (00)	00
	01	S10 (10)	10
	10	S5 (01)	00





Logic Expressions

$$D_1 = D + NQ_0\overline{Q_1}$$

$$D_0 = NQ_1 + N\overline{Q_0} + \overline{N}\,\overline{Q_1}Q_0$$

Next States

$$P = D + NQ_0\overline{Q_1} = D_1$$

$$C = D\overline{Q_1}Q_0$$

Outputs

Verilog - Solution 1 (Mealy)

```
module vending_machine(input N, input D, input reset_n, input clk,
output P, output C);
wire [1:0] next_state;
reg [1:0] present_state;

always @ (posedge clk, negedge reset_n)  // Implement flipflops in one
always block.
      begin
         if(reset_n==1'b0)
                  present_state <= 2'b00;
         else
                  present_state <= next_state;
      end

// Implement combinational logic (logic expressions)
assign next_state[1] = D | (N & present_state[0] & ~ present_state[1]);
assign next_state[0] = (N & present_state[1]) | (N & ~present_state[0])
                     | (~N & present_state[1] & present_state[0]);

assign P =  D | (N & present_state[0] & ~ present_state[1]);
assign C =  D & ~present_state[1] & present_state[0];

endmodule
```

Verilog Solution 2 (Mealy)

```
module vending_machine2(input N, input D, input clk, input reset_n,
output reg P, output reg C);
reg [1:0] next_state, present_state;
//Define states and state names
localparam [1:0] S0 = 2'b00, S5 = 2'b01 , S10 = 2'b10, S15 = 2'b11;

//Implement next states and outputs(combinational)
always @ (*)
begin
case (present_state)
 S0: begin
        case ({N,D})
          2'b00 : begin next_state = S0;  {P,C} = 2'b00; end
          2'b01 : begin next_state = S10; {P,C} = 2'b10; end
          2'b10 : begin next_state = S5;  {P,C} = 2'b00; end
         default: begin next_state = S0;  {P,C} = 2'b00; end
        endcase
     end
S5: begin
        case ({N,D})
          2'b00 : begin next_state = S0; {P,C} = 2'b00; end
          2'b01 : begin next_state = S15;{P,C} = 2'b11; end
          2'b10 : begin next_state = S10;{P,C} = 2'b10; end
         default: begin next_state = S0; {P,C} = 2'b00; end
        endcase
     end
```

Solution 2 – Cont'd (Mealy)

```
S10: begin
        case ({N,D})
          2'b00 : begin next_state = S0; {P,C} = 2'b00; end
          2'b01 : begin next_state = S10;{P,C} = 2'b10; end
          2'b10 : begin next_state = S5; {P,C} = 2'b00; end
         default: begin next_state = S0; {P,C} = 2'b00; end
        endcase
     end
S15: begin
        case ({N,D})
          2'b00 : begin next_state = S0; {P,C} = 2'b00; end
          2'b01 : begin next_state = S10;{P,C} = 2'b10; end
          2'b10 : begin next_state = S5; {P,C} = 2'b00; end
         default: begin next_state = S0; {P,C} = 2'b00; end
        endcase
     end
endcase //end of case
end  //end of always@(*)

// Implement flip-flops (sequential)
always @(posedge clk, negedge reset_n)
      begin
         if(reset_n==1'b0)
           present_state <= S0;
         else
          present_state <= next_state;
       end

endmodule
```