Sequential Logics Design

Outline

- ✓ **Verilog RTL behavior level design**
  - Assignments
  - Behavioral modeling
  - Synthesizable verilog code
- ✓ **Sequential logics design**
  - Flip-flop coding style
  - Flip-flop designs
- ✓ **Finite State Machine**

Verilog RTL Behavior Level Design

Assignments – Continuous Assignment (old)

- ✓ **The assignment is always active, and statements starts with the keyword "assign"**
  - Whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.
    - i.e. wire [3:0] a;
      assign a = b + c; // continuous assignment
- ✓ **Net declaration assignment**
  - An equivalent way of writing net assignment statement.
  - Can be declared once for a specific net.
    - i.e. wire [3:0] a = b + c; // net declaration assignment
      // is equivalent to the above description

Note : It's not allowed which required a concatenation on the LHS

i.e. wire [4:0] {cout , sum} = b+ c ; →*Wrong!*

(wire [3:0] sum ; wire cout ; {cout , sum} = b+ c ; →*OK!*)

Assignments – Continuous Assignment (old)

- ✓ **LHS should be “wire” type**
- ✓ **Logic Assignments**

  Ex:

  assign a = !b ;

  assign a = ~b ;

  assign a = b & c ;

- ✓ **Arithmatic Assignments**

  Ex:

  assign a = - b ;

  assign a = b * c ;

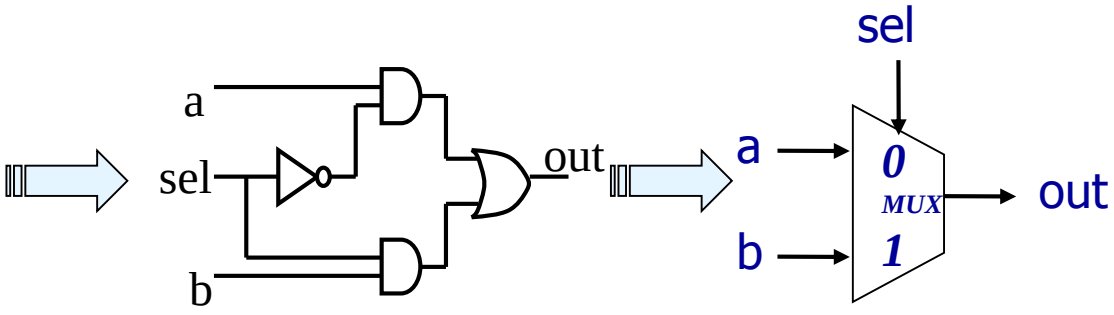
Assignments – Continuous Assignment

✓ Conditional Statements

Syntax:

assign <out_name> = (<expression>) ? true_statement : false_statement

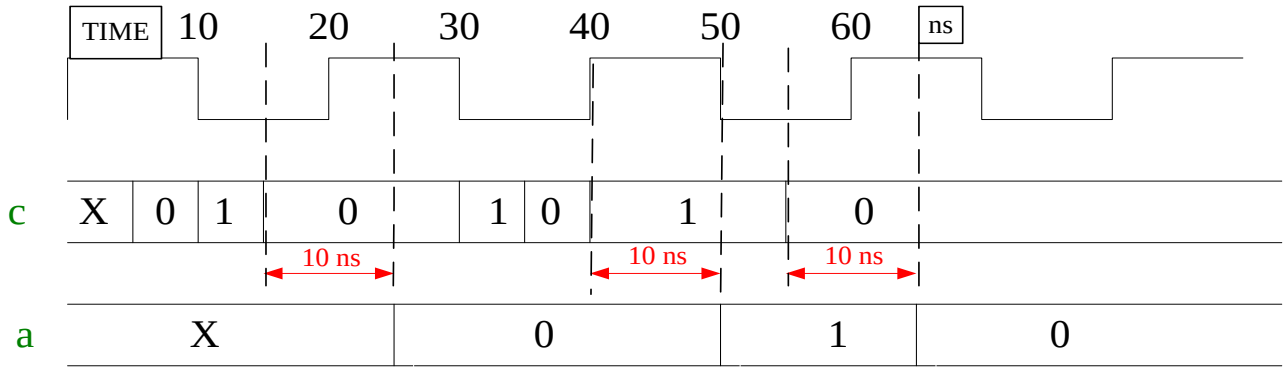
```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
wire        out;

//Continuous assignment
assign  out = (sel==0)?a:b;

endmodule
```



Assignments – Delay (old)

- **Inertial delay**
  - Only one assignment will be occurred

```
`timescale 1ns/10ps
module INERTIAL();

    wire a;
    reg  c;

  assign #10 a = c;

  initial begin
      #5  c = 0;
      #5  c = 1;
      #5  c = 0;
     #15 c = 1;
      #5  c = 0;
      #5  c = 1;
     #15 c = 0;

  #10 $finish;
    end
endmodule
```



Must be stable for 10 ns , so the value of a could update

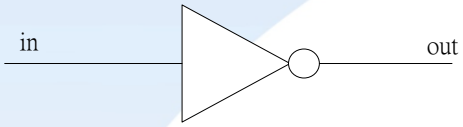
Assignments – Delay (old)

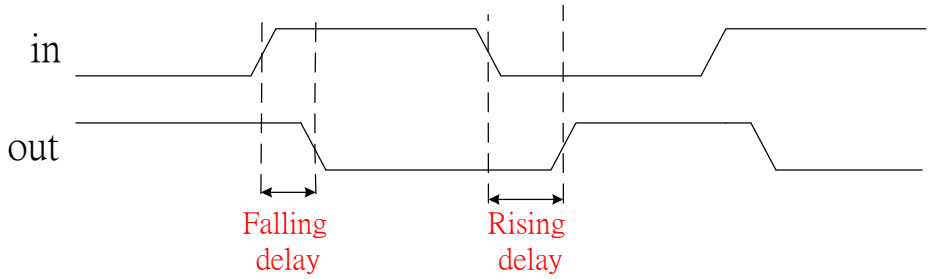
- ✓ **Delay : treated in the same way as for gate delay**
  - The time duration between RHS to LHS.
  - Lexical : #(1st,2nd,3rd) //can have one ,two or three delays
  - Rules : If RHS's lsb is 1 ,then (rising) 1st delay is used.
    If RHS's lsb is 0 ,then (falling) 2nd delay is used.
    If RHS's lsb is z ,then (turn off) 3rd delay is used.
    If RHS's lsb is x ,then the smallest delay is used.

Note : Inertial delay RHS changes before the previous LHS has updated, the lasted value changes only one to be applied.

Assignments – Delay (old)

- Example
  - Example 1





  - Example 2

```
wire [3:0] a, b;
assign #(10,20)  b=a;
initial
begin
   a=4'b0000;
   #100 a=4'b1101;
   #100 a=4'b0111;
   #100 a=4'b1110;
end
```

```
result :
0       a=0000   b=xxxx
20      a=0000   b=0000
100     a=1101   b=0000
110     a=1101   b=1101   //lsb is high
200     a=0111   b=1101
210     a=0111   b=0111   //lsb is high
300     a=1110   b=0111
320     a=1110   b=1110   //lsb is low
```

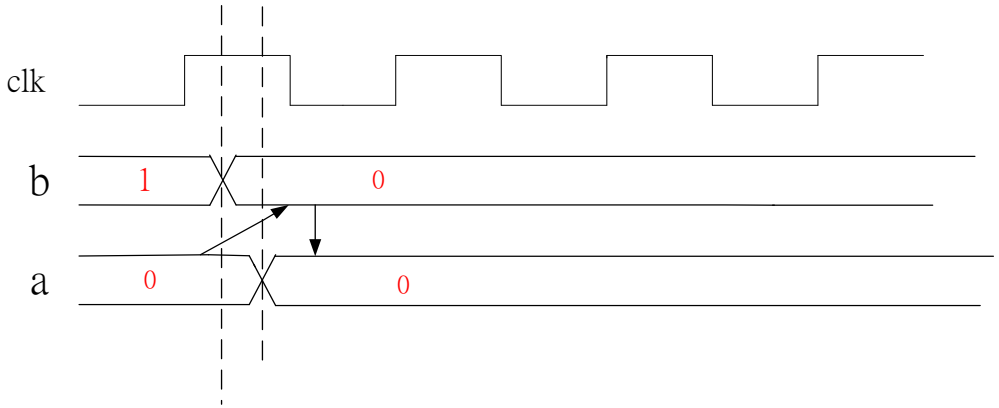
Assignments – Procedural Assignment

- ✓ **Procedural assignments update the value of register variables under the control of the procedural flow.**
- ✓ **Updating reg, integer, time, memory variables.**
  - Occur within always, initial, task and function.
    - i.e. reg a,clk; //*Note "reg", not "wire"*
      always #5 clk = ~clk; // procedural assignment
    - i.e. always @ (b) // procedural assignment with triggers
      a = ~b;
- ✓ **Two types of procedural assignment**
  - Blocking procedural assignments
  - Non-blocking procedural assignments

Assignments – Procedural Assignment (cont.)

- **Blocking procedural assignment**
  - Syntax: <value> = <timing_control><expression>
  - Must be executed before executing the statement that follow it in a sequential block.
    - Example

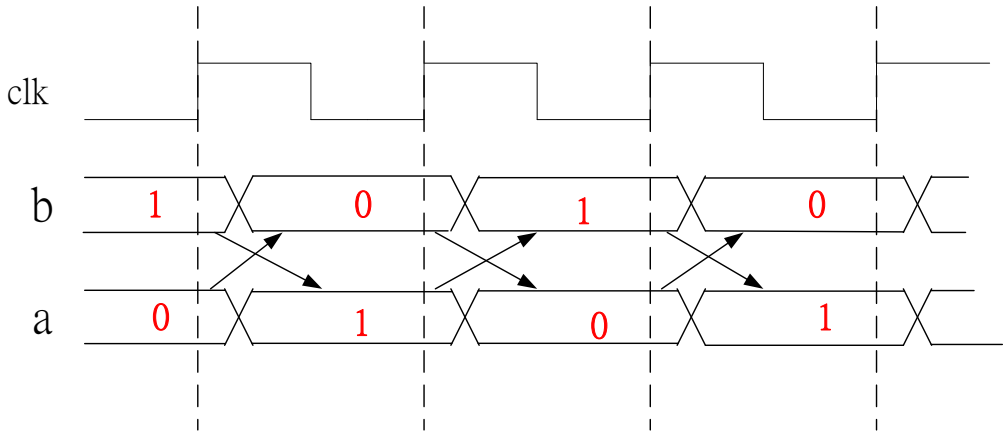
```
initial begin
  a=0;b=1;clk=0;
end
always @(posedge clk)
begin
  b = #1 a;
  a = #1 b;
end
```



Assignments – Procedural Assignment (cont.)

- ✓ **Non-Blocking procedural assignment**
  - Syntax: <value> <= <timing_control><expression>
  - The statements are executed **within the same time step** without regard to order or dependence upon each other.
  - Evaluating Non-blocking divided into two steps:
    - Example

```
initial begin
   a=0;b=1;clk=0;
end
always@(posedge clk)
 begin
     b <= #1 a;
     a <= #1 b;
 end
```



- ◆ Step1 : at posedge clk the simulator evaluates the RHS's new value, and scheduled when to update LHS.
- ◆ Step2 : at 1ns after posedge clk the simulator updates LHS. Because the given delay (#1) is expired.

Behavioral Modeling - Structured procedures

- **Structured procedures (four types)**
  - always statement
    - Each always statement repeats continuously throughout the whole simulation run.
    - Need timing control to prevent Deadlock condition.

      Ex : always areg = ~areg; // Deadlock!!
  - initial statement
    - An initial statement is executed only once.
    - Note : "initial" can't be synthesized!!
  - task, Function (will be introduced in Lec03)

Note : each always statement and initial statement starts a separate activity flow, but all of flow are concurrent.

Behavioral Modeling – Conditional Statements (old)

- **Conditional Statements:** The conditional statement is decide whether to execute a statement.
  - **if…,else if…,else…**: The most commonly used conditional statements. The statement occurs if the expressions controlling the if statement evaluates to be true.

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
reg         out;

//Procedural assignment
always @(a or b or sel)
begin
   if (sel==0)   out  = a;
   else          out  = b;
end
endmodule
```

Anything assigned in an "*always*" block must also be declared as *reg* type

The "*always*" block runs once whenever a signal in the **sensitivity list** changes value

Behavioral Modeling – Conditional Statements (old)

- **case** : (casex, casez) Be used for switching multiple selections.

Syntax:

```
case(expression)
  alternative1 : statement1;
  alternative2 : statement2;
  alternative3 : statement3;
   …
  default : default statement;
endcase
```

Ex:

```
case(MUX)
2'b00: a = b ;
2'b01: a = c;
2'b1x: a = d;
endcase
```

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
reg         out;
//Procedural assignment
always @(a or b or sel) begin
  case(sel)
    1'b0:  out  = a;
    1'b1:  out  = b;
  endcase
end
endmodule
```

Behavioral Modeling – Blocking & Non-blocking

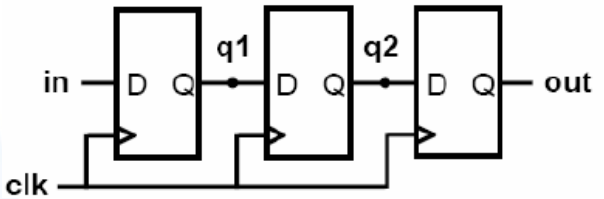
✓ Non-blocking assignment

Non-blocking

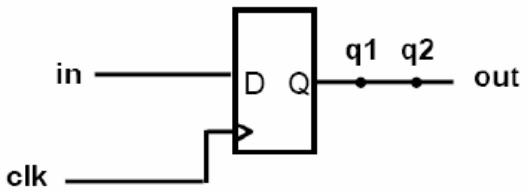
```
always @( posedge  clk)
begin
      q1 <=  in;
      q2 <=  q1;
      out <= q2;
end
```

Blocking

```
always @( posedge  clk)
begin
      q1  =  in;
      q2  =  q1;
      out  = q2;
end
```



Shift register behavior



Single register behavior

Behavioral Modeling – Blocking & Non-blocking (cont.)

✓ Blocking assignment

Blocking

```
always @( a  or  b  or  c)
begin
     x  =  a  &  b;
     y  =  x  |  c;
end
```

Non-blocking

```
always @( a  or  b  or  c)
begin
     x  <=  a  &  b;
     y  <=  x  |  c;
end
```

Blocking behavior	a	b	c	x	y
initial condition	1	1	0	1	1
a changes 1 -> 0	0	1	0	1	1
x = a & b;	0	1	0	0	1
y = x | c;	0	1	0	0	0

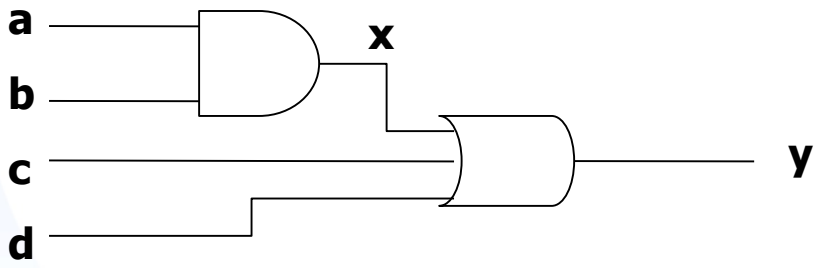
non-blocking behavior	a	b	c	x	y
initial condition	1	1	0	1	1
a changes 1 -> 0	0	1	0	1	1
x <= a & b; y <= x | c;	0	1	0	0	1

- Blocking assignments are in topological order

Behavioral Modeling – Combinational Blocks (old)

- **Combinational block**
  - specify complete but no redundant sensitivity lists
  - assignment should be applied in topological order
- **Using continuous assignments**

```
assign x = a & b ;
assign y = x | c | d ;
```



Behavioral Modeling – Combinational Blocks (old)

- ✓ **Using blocking assignments**

```
always@(a or b or c)
begin
    x = a & b;
    y = x | c | d;
end
// simulation-synthesis mismatch
```

```
always@(a or b or c or d or e)
begin
    x = a & b;
    y = x | c | d;
end
// performance loss
```

```
always@(a or b or c or d or x)
begin
    x = a & b;
    y = x | c | d;
end
// also correct
```

```
always@(a or b or c or d)
begin
    y = x | c | d;
    x = a & b;
end  // not in topological order
// simulation-synthesis mismatch
```

```
always@(a or b or c or d)
begin
    x = a & b;
    y = x | c | d;
end
// best final
```

```
always@*
begin
    x = a & b;
    y = x | c | d;
end
// also correct
```

Behavioral Modeling – Combinational Blocks

- **Using nonblocking assignments**
  - Not need to be in topological order

```
always@(a or b or c or d or x)
begin
    x <= a & b;
    y <= x | c | d;
end
```

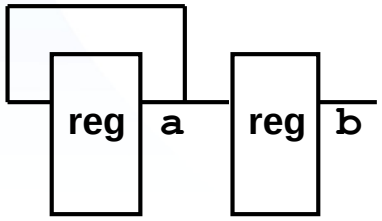
```
always@*
begin
    x <= a & b;
    y <= x | c | d;
end
```

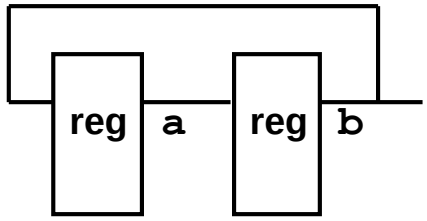
Behavioral Modeling – Sequential Blocks

- ✓ **Sequential block**
  - use non-blocking assignments
  - avoid **race** problems in simulation
- ✓ **Comb./Seq. logic should be separated**

```
always@(posedge clk)
begin
    b = a;
    a = b;
end  // wrong style
```

```
always@(posedge clk)
begin
    b <= a;
    a <= b;
end  // right style
```

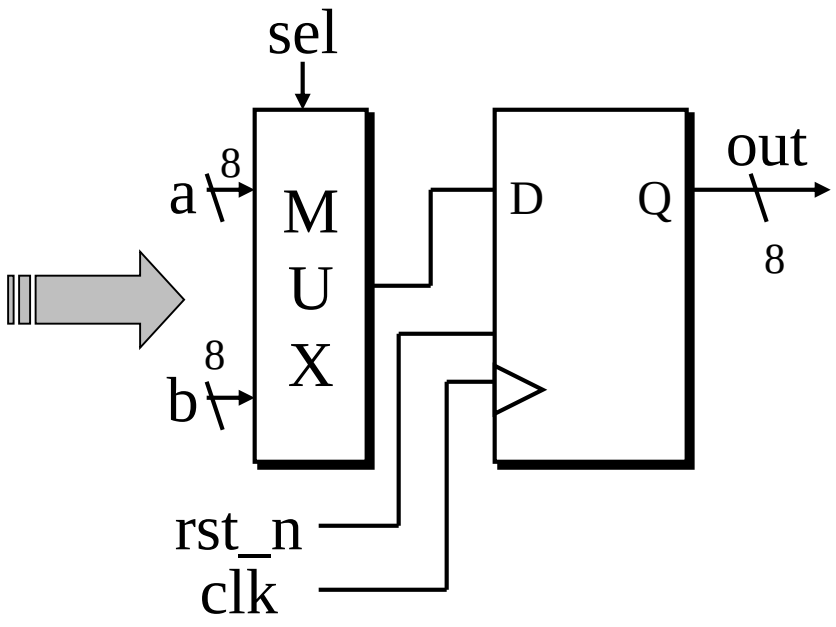




Behavioral Modeling – Register Description

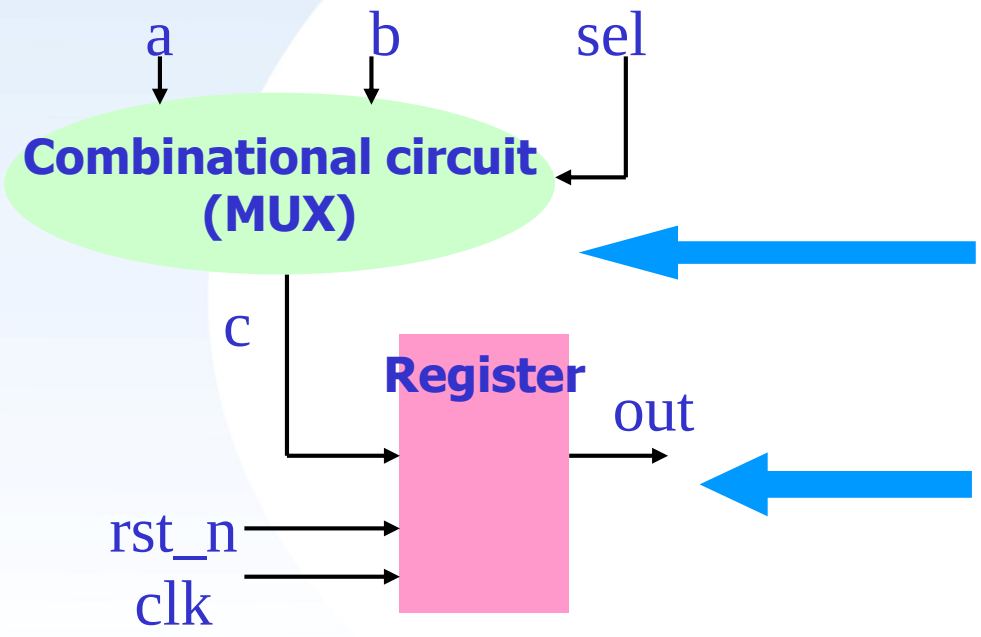
- **Informal description:**
  - Combinational and memory element is combined in one always procedure.

```
module MUX2_1(out,a,b,sel,clk,rst_n);
input       sel,clk,rst_n;
input       [7:0]       a,b;
output      [7:0]       out;
reg         [7:0]       out;

always@(posedge clk or negedge rst_n)
begin
            if(!rst_n) out <= 0;
            else if (sel==0) out <= a;
            else out <= b;
end

endmodule
```



Behavioral Modeling – Register Description (cont.)

- **Normative description (it is a better way):**
  - The combinational description is isolated.



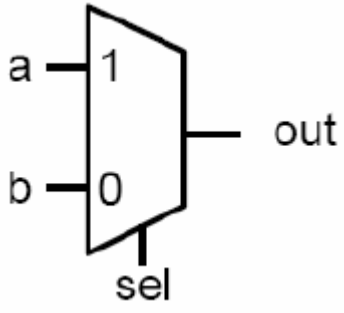

```
module MUX2_1(out,a,b,sel,clk,rst_n);
input       sel,clk,rst_n;
input       [7:0]       a,b;
output      [7:0]       out;
reg         [7:0]       c;
reg         [7:0]       out;

//Continuous assignment
always @(sel or a or b)
begin
    if (sel==b0) c = a;
    else c = b;
end

//Procedural assignment
always @(posedge clk or negedge rst_n)
begin
                if(!rst_n) out <= 0;
                else out <= c;
end
endmodule
```

Behavioral Modeling – Register Description (cont.)

✓ Comparison

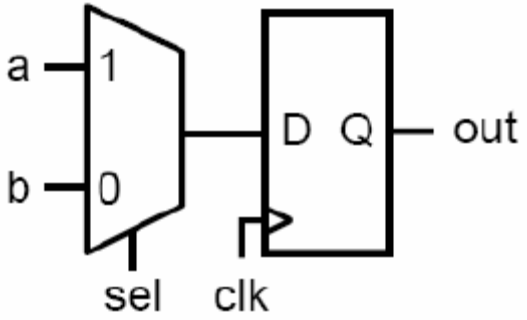
Blocking

```
module  Combinational(out,a,b,sel);
input          sel,a,b;
output         out;
reg            out;

always @(a  or  b  or  sel )
begin
      if( sel )        out  = a;
      else             out  = b;
end
endmodule
```



Non-blocking

```
module  Sequential(out,a,b,sel,clk);
input          sel,a,b,clk;
output         out;
reg            out;

always @(posedge  clk )
begin
      if( sel )        out  <= a;
      else             out  <= b;
end
endmodule
```



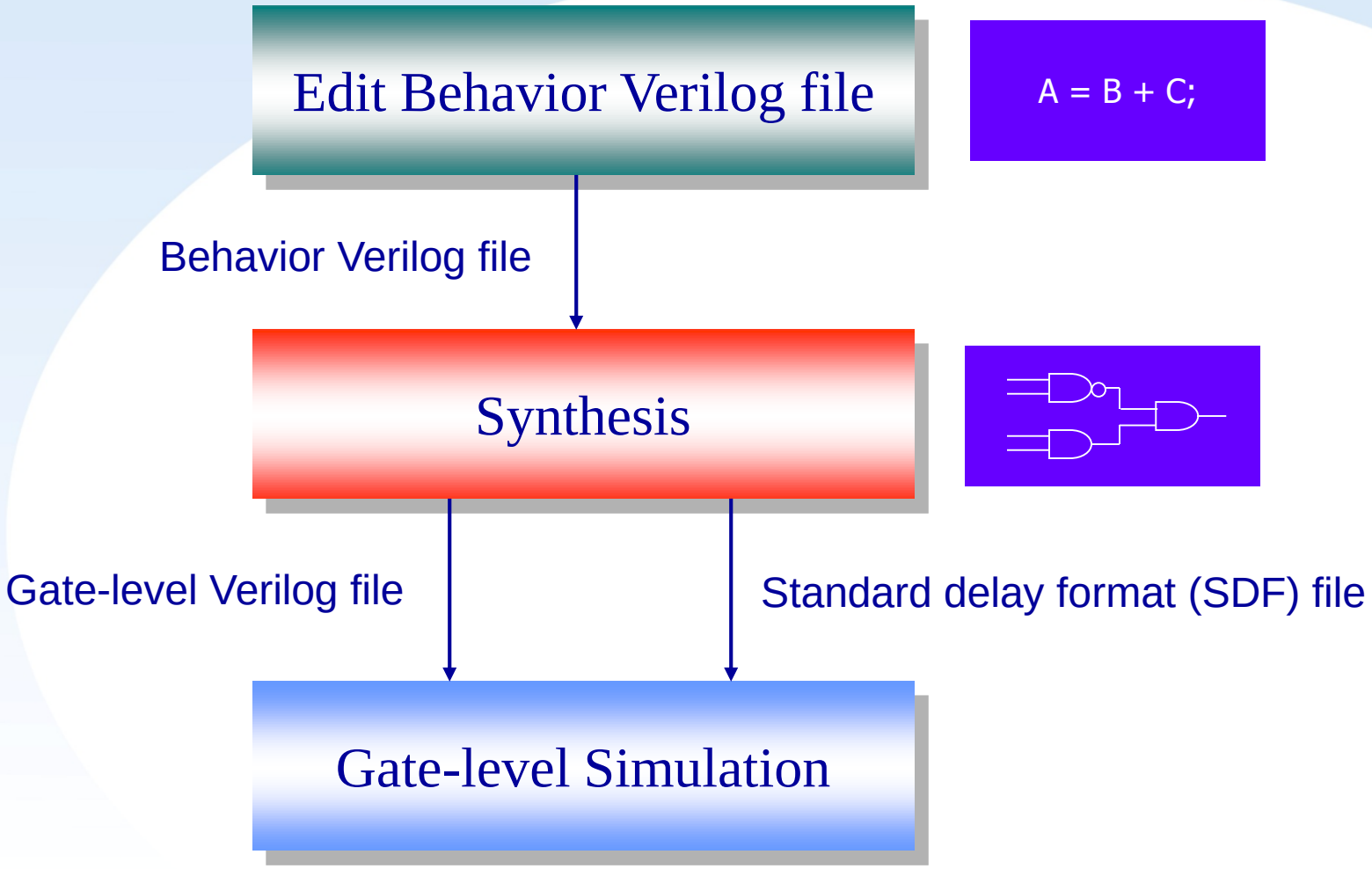
Behavioral Modeling – *Avoid Unintentional* Latch Design

- ✓ **In a sequential circuit -- with clk control**
  - It is a flip-flop so there is not a latch problem.
- ✓ **In a combinational circuit -- without clk control**
  - If some net needs to keep its data, DA will synthesize a latch.
- ✓ **How to avoid?**
  - Conditional statement : must be full cases
  - Otherwise it will produce latches.
    - if – else work together or add default value

    Ex: if (a== b) c = 1 ;
    - Case statement : remember default value

    Ex: case (a) 1'b0: c = b; endcase
- ✓ **Notice**
  - In a combinational circuit, no information will be stored, so latches are not allowed.
- ✓ **Latch is a memory storage device**
  - It will cause the problems of timing analysis .
    - That's why we recommend to avoid latches here!!

Synthesizable Verilog Code



Synthesizable Verilog Code - keyword

module endmodule	begin end	case endcase	if else
? :	default	input output	reg wire
parameter	always	posedge negedge	or
assign	+ - * / %	^ ~ & | ~& == != && ||	for (※) repeat (※)

Synthesizable Verilog Code (cont.)

- ✓ **Four data type can be synthesized**
  - input, output, reg, wire
- ✓ **1-D data type is convenient for synthesis**
  - EX. reg [7:0] a;
    reg [7:0] a[3:0]; →It isn't well for the backend verifications
- ✓ **Examples of synthesizable register description**
  - always@(posedge clk)
  - always@(negedge clk)
  - always@(posedge clk or posedge rst) if(rst).. else..
  - always@(negedge clk or posedge rst) if(rst).. else..
  - always@(posedge clk or negedge rst_n) if(!rst_n).. else..
  - always@(negedge clk or negedge rst_n or negedge set) if(!set).. else if (!rst_n).. else..
- ✓ **Example of not synthesizable register description**
  - always@(posedge clk or negedge clk)

Synthesizable Verilog Code (cont.)

- ✓ **Data has to be described in one always block**

```
always@(posedge clk)
        out <= out + 1;
always@(posedge clk)
        out <= a;
```

Wrong!!

- ✓ **Data has to be described by either blocking assignment or non-blocking assignment.**

```
always@(posedge clk or negedge rst_n)
        if(!rst_n) out = 0;
        else       out <= out + in;
```

Wrong!!

Non-synthesizable Verilog Code – for testbench

- ✓ **Looping Statements except “for”**
- ✓ **Initial Statements**
- ✓ **Task & Function**
- ✓ **Delay Timing Control & Event Timing Control**

Behavioral Modeling – Looping Statements

- **Four types**
  - forever (can't synthesize)
    - Continuously executes a statement until to meet $finish or disable.
  - repeat
    - Executes a statement a fixed number of times.
  - while
    - Executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.
  - for
    - Controls execution of its associated statements by a three step process.

Note : The above is used to run simulation !!

Behavioral Modeling – Looping Statements (cont.)

- **for-loop syntax**
  - for (initial_assignment; condition; step_assignment)
    begin
      statement;
    end
  - Can use to initialize a memory
    EX.

```
integer  i;
always@(posedge clk or posedge reset)
begin
  if(reset)
  begin
     for(i=0; i<1024; i=i+1)
     begin
        memory[i]=0;
     end
  end
  else
   .
   .
   .
```

Behavioral Modeling – Looping Statements (cont.)

- **Example**
  - forever @(posedge clk) rega =~rega;
    - Need timing control to avoid dead-lock
      EX. forever rega = ~ rega // dead-lock !!
  - repeat(size) // if size = 5 then do loop five times
    begin
      statement
    end
  - while(temp) // do loop until temp=0 (false)
    begin
      statement
      temp >> 1;
    end

Note (for repeat) :

1. Number of iteration is a constant :It can synthesize, but it is inefficient. So not recommend to use.
2. If number of iterations is a variable : It can't be synthesized!!

Behavioral Modeling – Timing Controls

- ✓ **Two types**
  - Delay timing control
    - Introduced by symbol (#)
  - Event timing control
    - Introduced by symbol (@)

Behavioral Modeling – Timing Controls (cont.)

- ✓ **Delay timing control**
- ✓ **Delay control can't be synthesized !!**
  - Syntax: =#<number > or #<identifier>
    - #10 rega = regb; // delay the execution by 10 time units
    - #d rega = regb; // d can be a parameter
    - #((d+e)/2) rega= regb; // delay by (d+e)/2 time units
  - Zero delay control – a special case of delay control
    - #0 a=0;
  - Intra-assignment delay control
    - Just as regular delay but the RHS expression is evaluated before, updated LHS after delay.
    - rega <= #1 regb + regc;

      // when time t execution the RHS regb + regc and hold the data to update LHS after 1 time units;

Behavioral Modeling – Timing Controls (cont.)

- **Event timing control**
  - You can use changes on nets and register as event to trigger the execution of a statement
  - Syntax
    - @<event_expression><statement_or_null>
      - Statement will wait for the data of event_expression changed, then execute statement.
    - negative is detected from 1 to x or from x to 0
    - posedge is detected from 0 to x or from x to 1

    Ex : @r rega = regb;
    // controlled by any value changes in the register r;
    @(posedge clk) rega=regb;
    // controlled by posedge on clk;

Behavioral Modeling – Timing Controls (cont.)

- ✓ **Event OR construct**
  - Can be expressed such that occurrence of any one event will trigger the statement
    - @(trig or enable) rega=regb; //controlled by trig or enable
    - @(posedge clk_a or posedge clk_b or trig) rega = regb; //controlled by clk_a or clk_b or trig

Sequential Logics Design

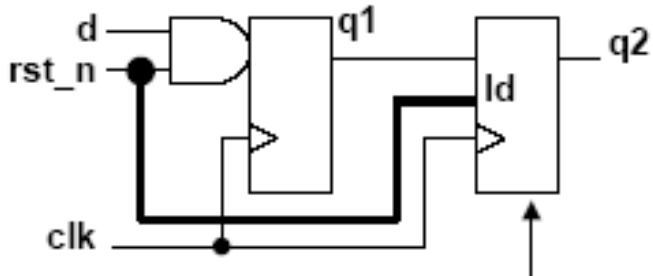
Flip-Flop Coding Style

- ✓ **Bad flip-flop design**
  - Put dissimilar flip-flops in the same always block.
    EX.

```
module badFFstyle (q2, d, clk, rst_n);
  output q2;
  input  d, clk, rst_n;
  reg    q2, q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

q2 is only loaded if rst_n is high

BAD PARTITIONING Style creates EXTRA LOGIC



rst_n becomes a "load-data" signal

Flip-Flop Coding Style (cont.)

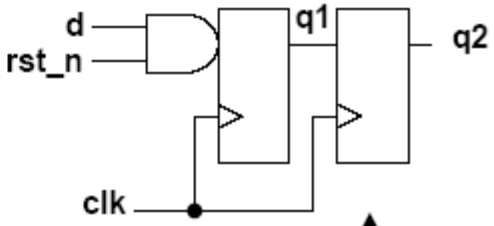
- ✓ **Good flip-flop design**
  - Put dissimilar flip-flops in the separate always blocks.
    EX.

```
module goodFFstyle (q2, d, clk, rst_n);
  output q2;
  input  d, clk, rst_n;
  reg    q2, q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else        q1 <= d;

  always @(posedge clk)
    q2 <= q1;
endmodule
```

q2 is loaded on every posedge clk

Note: To model sequential logic use nonblocking assignments

Good partitioning - no extra logic

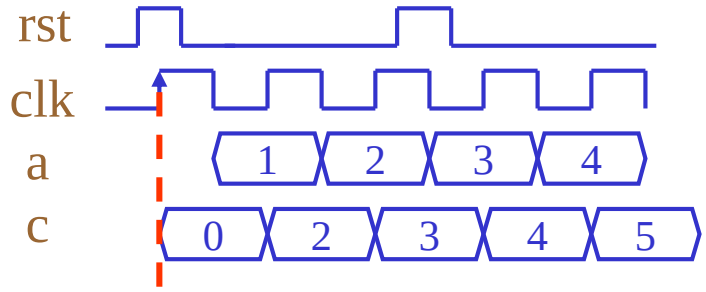


No reset on the follower flip-flop

Flip-Flop Designs

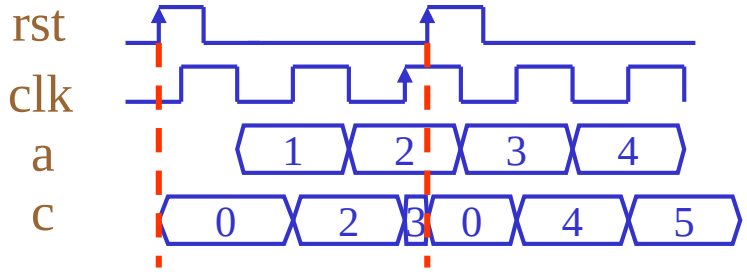
- **Register with synchronous reset**
  - Syntax: @(posedge clk) : for synchronous reset

```
EX.
always @(posedge clk)
begin
  if(rst)     c <= 0;
  else        c <= a + 1;
end
```



- **Register with asynchronous reset**
  - Syntax: @(posedge clk or posedge rst)

```
EX.
always @(posedge clk or posedge rst)
begin
  if(rst)     c <= 0;
  else        c <= a + 1;
end
```



Flip-Flop Designs – Synchronous Resets (cont.)

- **Advantages**
  - Easier work with cycle based simulator
  - Typical recommended for DFT design
    - DFT (Design for test)
  - Easier for ECO (Engineering Change Order)
  - Glitch filtering from reset combinational logic
    - Filter the small glitch of reset between clock
  - Glitch filtering if reset is in a mission-critical application
    - When reset is generated by a set of internal condition, it can filter the glitch of logic equations between clock.
- **Disadvantages**
  - May not be able come out unknown X during simulation
  - Adds delay to data path
  - Can't be reset without clock signal

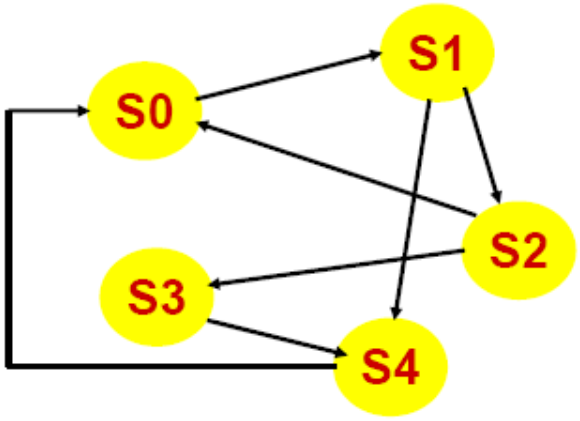
Flip-Flop Designs – Asynchronous Resets (cont.)

- ✓ **Advantages**
  - Reset is immediate
  - No problem related to unknown X propagation in simulation
  - Does not interfere or add extra delay to data path
  - Very easy to implement
  - Reset is independent of clock signal
- ✓ **Disadvantages**
  - Noisy reset line could cause unwanted reset
    - This can be filtered
  - Difficult for ECO
  - DFT prefer synchronous designs

Finite State Machine

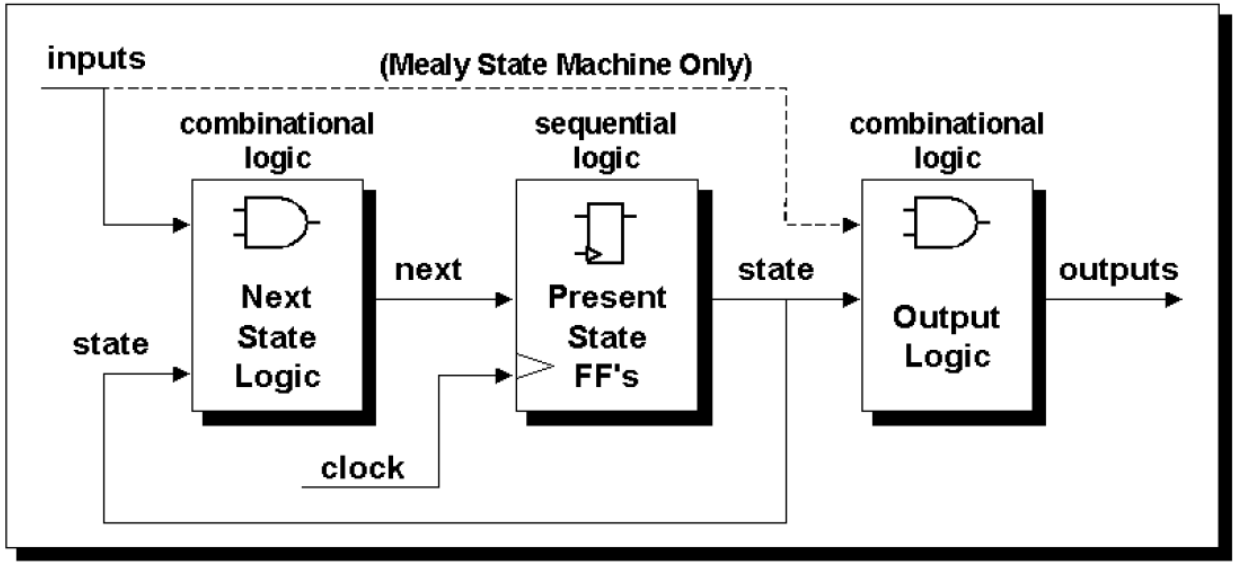
Finite State Machine

- ✓ Finite state machine
  - One of candidates for describing a sequential circuit.
  - Divide a sequential circuit operation into finite number of states.
  - A state machine controller can output results depending on the input signal, control signal and states.
  - As different input or control signal changes, the state machine will take a proper state transition.
- ✓ State diagram



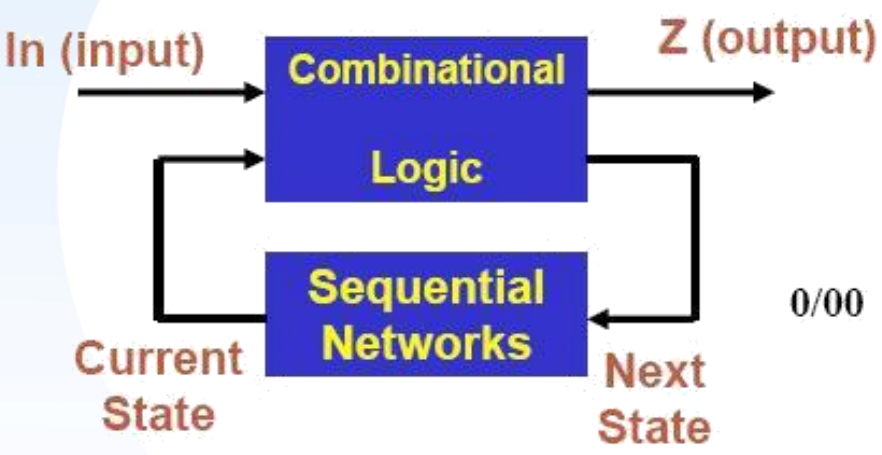
FSM Categories

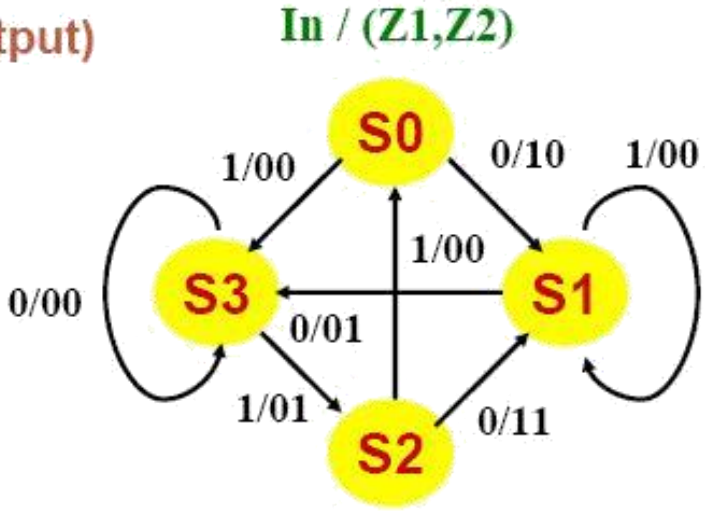
- ✓ Mealy machine
  - The outputs depends on the present state and the incoming input.
- ✓ Moore machine
  - The outputs of the machine depends on the present state only.
  - The output are synchronized with the clock.



Mealy Machine

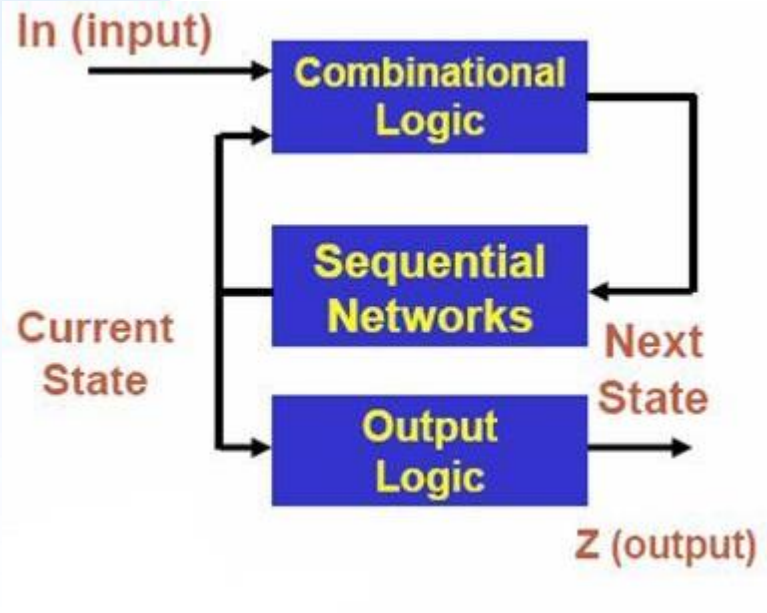
- ✓ Mealy machine
  - Contains sequential networks and combinational logic.
  - The sequential network is a set of filp/flops.
  - The output follows the input and current state.

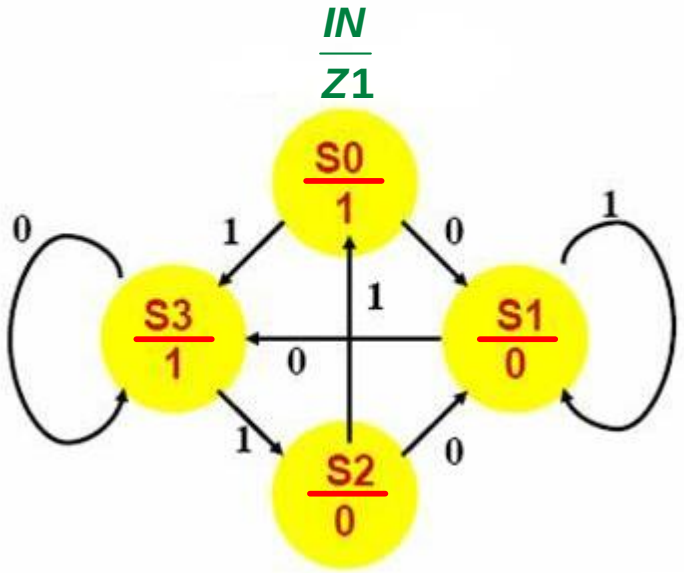




Moore Machine

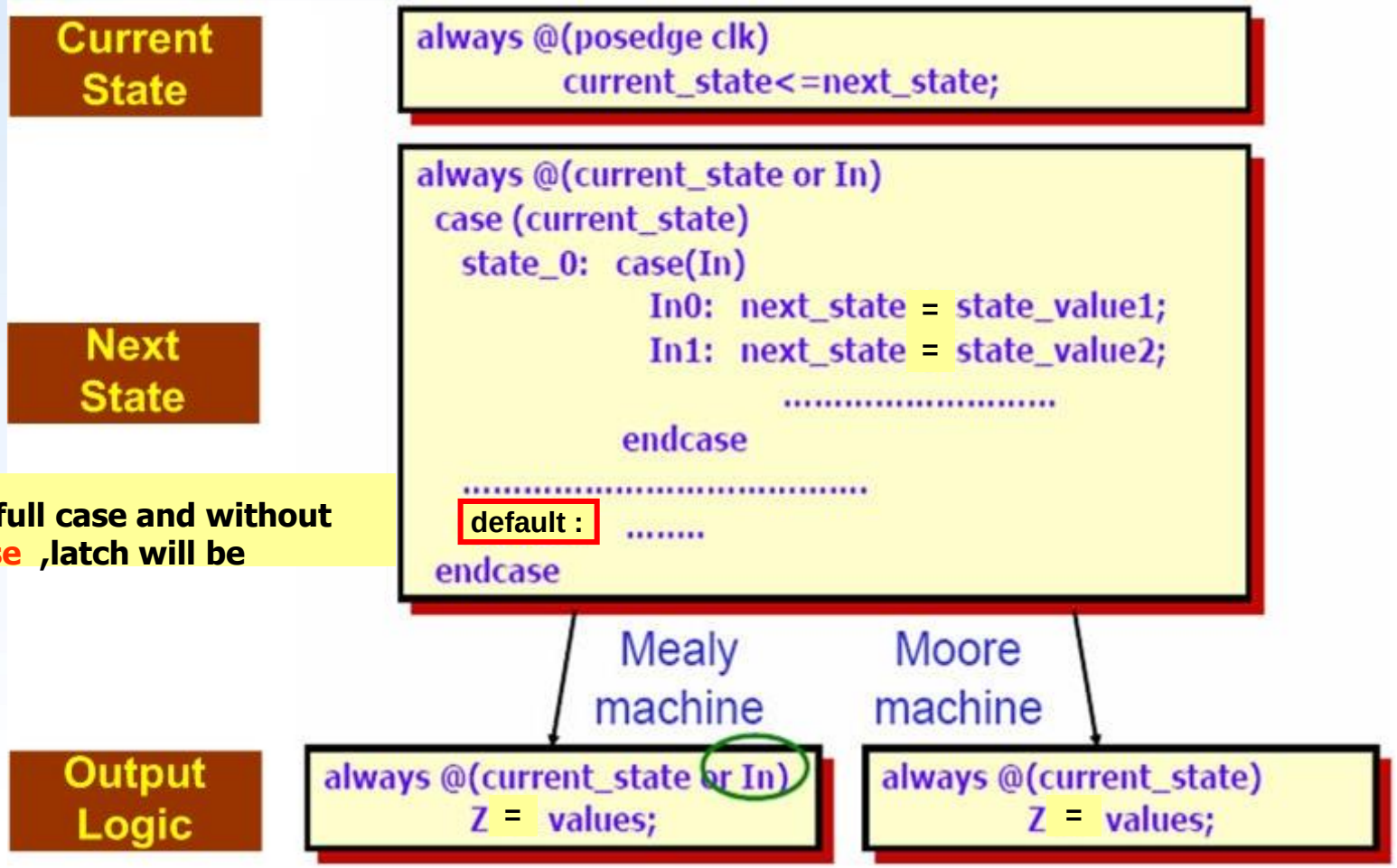
- ✓ Moore machine
  - Contains sequential networks and combinational logic.
  - Sequential network is a set of filp/flops.
  - The output depends on the current state only.





FSM Coding Styles (1/3)

- ✓ FSM coding style 1
  - Separate CS, NS and OL



FSM Coding Styles (2/3)

- ✓ FSM coding style 2
  - Combined CS and NS, Separate OL

Current State & Next State

```
always @(posedge clk) begin
    case (current_state)
       state_0:  case(In)
                    In0:  state<=state_value1;
                    In1:  state<=state_value2;
                          ..........................
                 endcase
       ....................................
       state_n:  ........
    endcase
end
```

Output Logic

Mealy machine

```
always @(current_state or In)
    Z = values;
```

Moore machine

```
always @(current_state)
    Z = values;
```

FSM Coding Styles (3/3)

- ✓ FSM coding style 3
  - Combined NS and OL, Separate CS

Current State

```
always @(posedge clk)
        current_state<=next_state;
```

Next State & Output Logic

```
always @(current_state or In) begin
 case (current_state)
   state_0:  case(In)
               In0:  next_state = state_value1;
               In1:  next_state = state_value2;
                     ..........................
              Z = values;//Mealy machine
             endcase
   ....................................
   state_n:  ........
   Z = values;//Moore machine
 endcase
end
```

FSM States Encoding

- ✓ **States assignment**

No.	Sequential	Gray	Johnson	One-Hot
0	0000	0000	00000000	0000000000000001
1	0001	0001	00000001	0000000000000010
2	0010	0011	00000011	0000000000000100
3	0011	0010	00000111	0000000000001000
4	0100	0110	00001111	0000000000010000
5	0101	0111	00011111	0000000000100000
6	0110	0101	00111111	0000000001000000
7	0111	0100	01111111	0000000010000000
8	1000	1100	11111111	0000000100000000
9	1001	1101	11111110	0000001000000000
10	1010	1111	11111100	0000010000000000
11	1011	1110	11111000	0000100000000000
12	1100	1010	11110000	0001000000000000
13	1101	1011	11100000	0010000000000000
14	1110	1001	11000000	0100000000000000
15	1111	1000	10000000	1000000000000000