ECE 351
Verilog and FPGA Design

Week 7_2: **Modeling sequential logic (wrap-up)**
Avoiding unintended latches

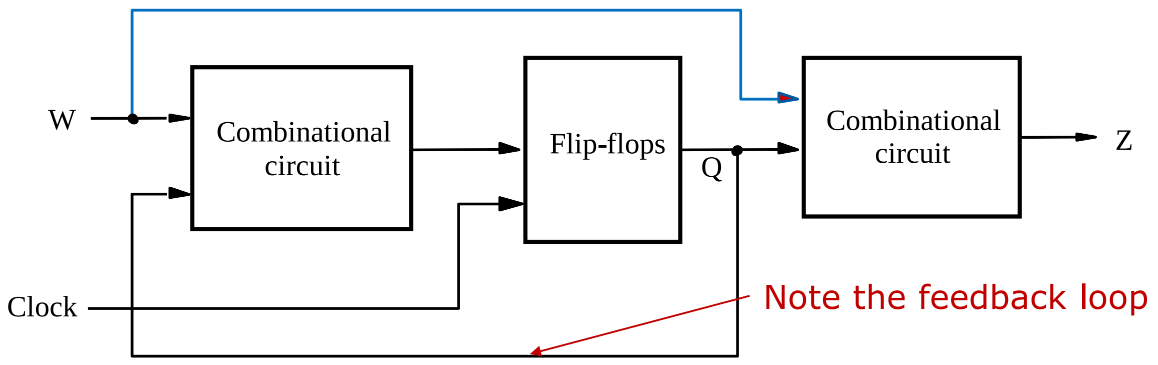
Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Finite State Machines

Sources:

- Roy's lecture notes from days long past (2003 – 2004)
- *Fundamentals of Digital Logic with Verilog 1st Edition*, Stephen Brown and Zvonko Vranesic, McGraw-Hill, 2003

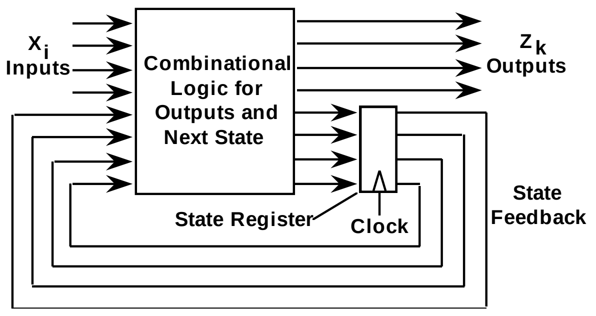
Sequential circuits - general form

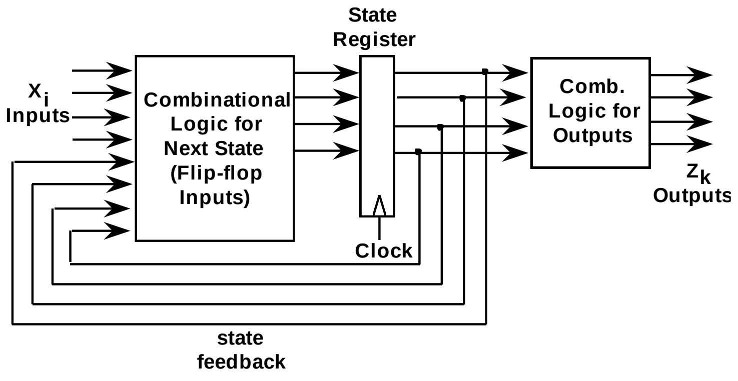


- A Sequential circuit is a circuit where the outputs depend on past behavior and the present value of the inputs
- A Synchronous sequential circuit is a sequential circuit where a clock signal is used to control the sequencing
- A Sequential circuit of this form is also called a Finite State Machine (FSM)

Portland State UNIVERSITY

Mealy and Moore machines





Mealy Machine

- ❑ Outputs depend on state AND inputs
- ❑ Input change causes an immediate output change
- ❑ Asynchronous signals

Moore Machine

- ❑ Outputs are function solely of the current state
- ❑ Outputs change synchronously with state changes

Portland State UNIVERSITY

Our first FSM design (FSM)

Step 1: Understand the specification

- Specification:
  - The circuit has one input w and one output z
  - All changes occur on the positive edge of a clock signal
  - The output z is equal to 1 if during two immediately preceding clock cycles the input w was equal to 1. Otherwise, z = 0;

Sample Sequence:

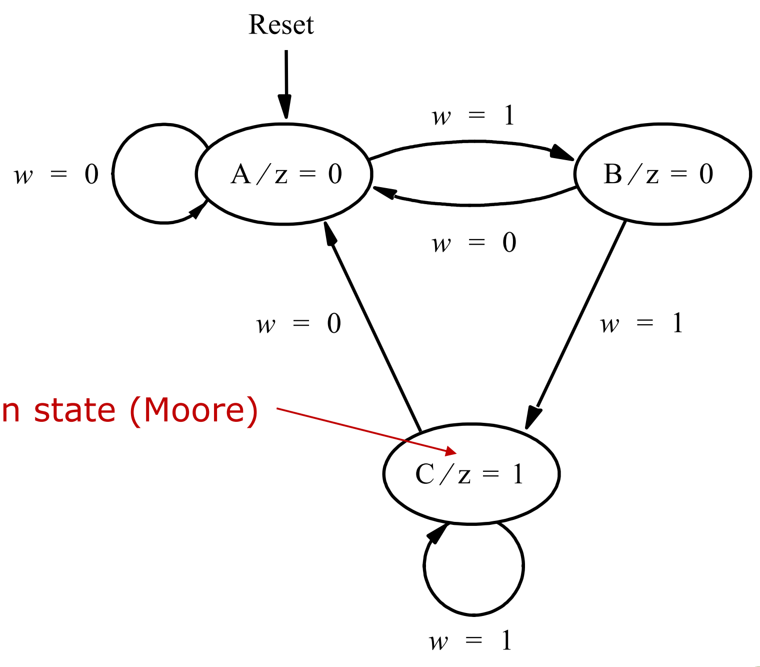
Clockcycle:	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Source: B & V Section 8-1

Our first FSM design example (cont'd)

Step 2: Draw a State transition diagram

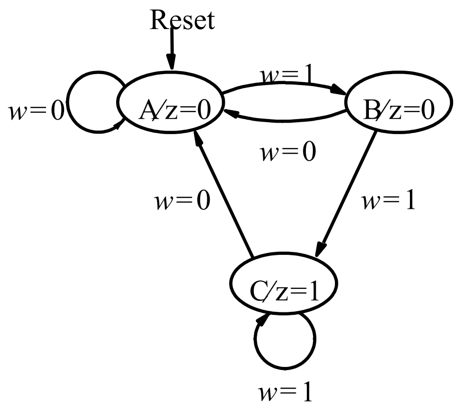
output is in state (Moore)



Source: B & V Section 8-1

Our first FSM design example (cont'd)

Step 3: Map the State transition diagram to a state table



Present state	Next state $w=0$	Next state $w=1$	Output z
A	A	B	0
B	A	C	0
C	A	C	1

Source: B & V Section 8-1

Our first FSM design example (cont'd)

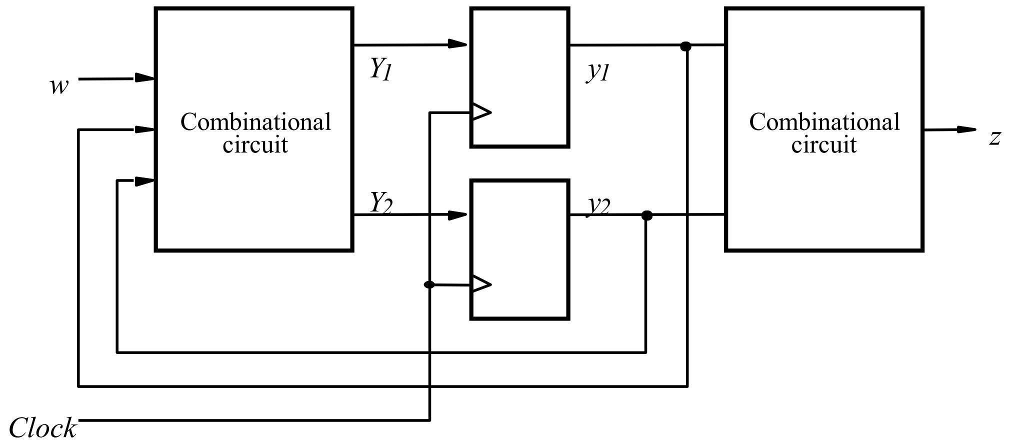
Step 4b: Create the State Assigned table

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Our First FSM Design Example (cont’d)

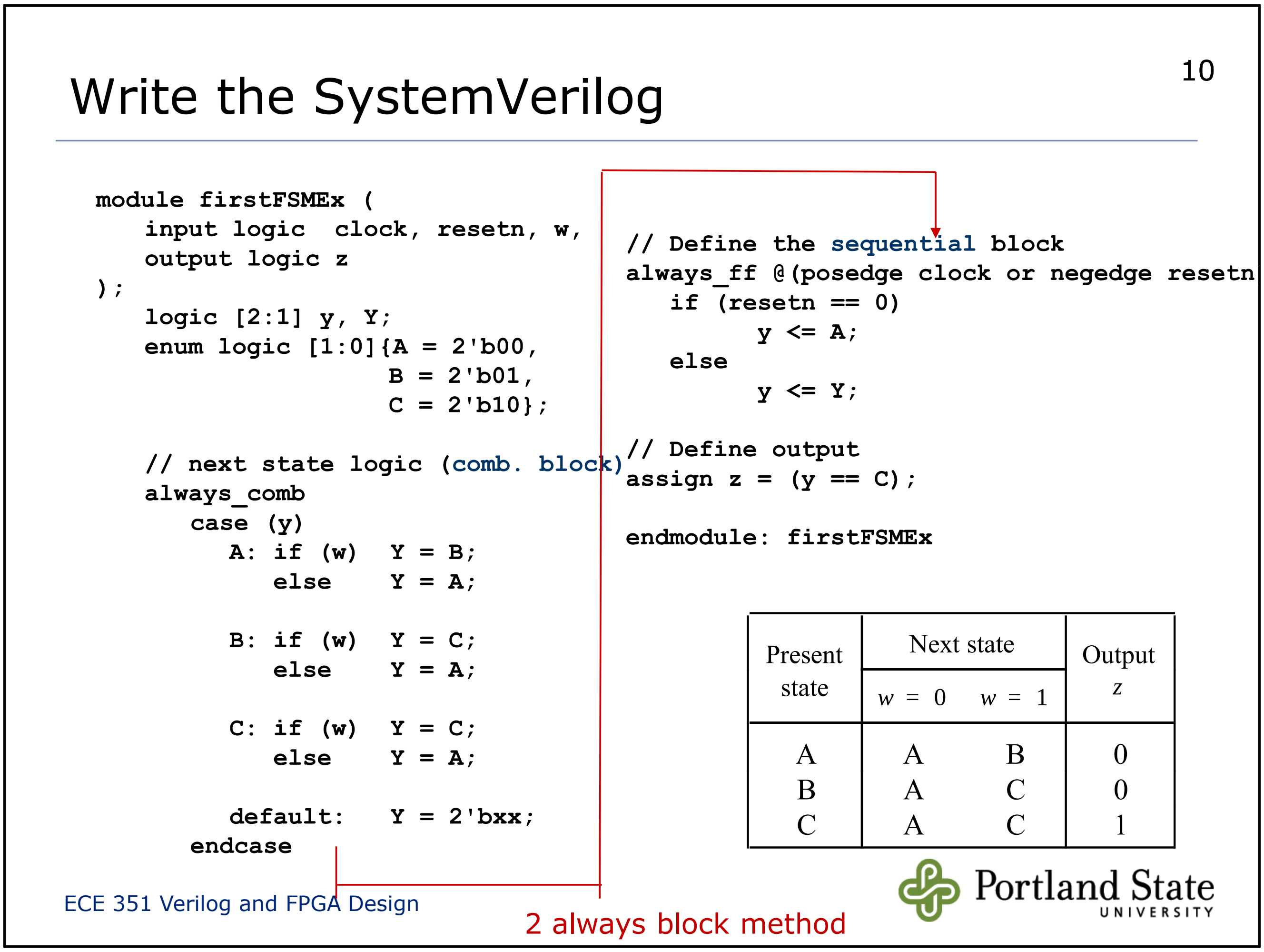
Step 4c: Map the State Table to hardware



Step 4d: Use K-maps, alternate state mappings, etc. to minimize logic

Step 5: Draw schematics

Source: B & V Section 8-1



Write the SystemVerilog

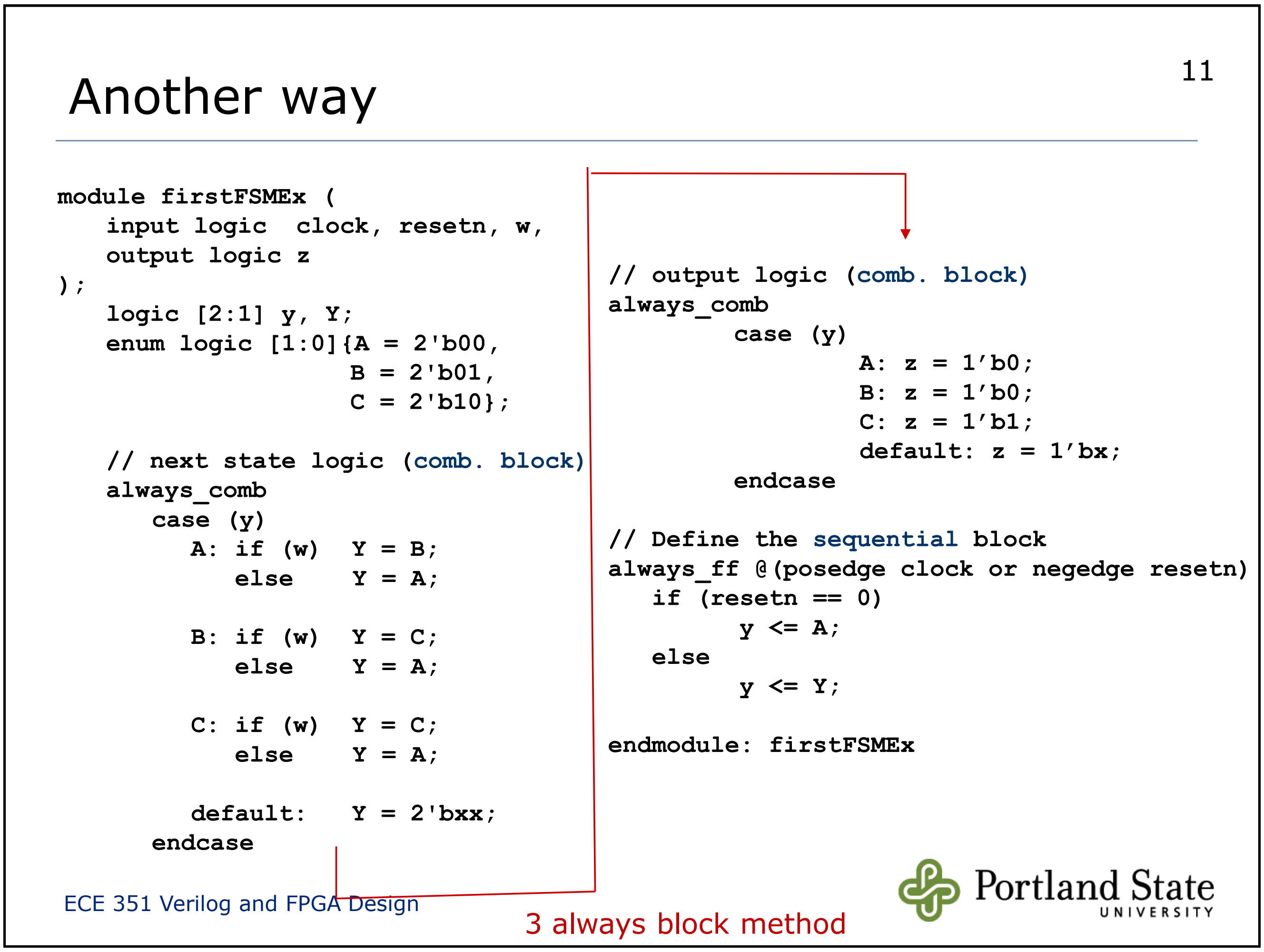
```
module firstFSMEx (
    input logic  clock, resetn, w,
    output logic z
);
    logic [2:1] y, Y;
    enum logic [1:0]{A = 2'b00,
                     B = 2'b01,
                     C = 2'b10};

    // next state logic (comb. block)
    always_comb
       case (y)
         A: if (w)  Y = B;
            else    Y = A;

         B: if (w)  Y = C;
            else    Y = A;

         C: if (w)  Y = C;
            else    Y = A;

         default:   Y = 2'bxx;
       endcase
```

```
// Define the sequential block
always_ff @(posedge clock or negedge resetn
   if (resetn == 0)
         y <= A;
   else
         y <= Y;

// Define output
assign z = (y == C);

endmodule: firstFSMEx
```

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	A	C	0
C	A	C	1

2 always block method

Portland State UNIVERSITY



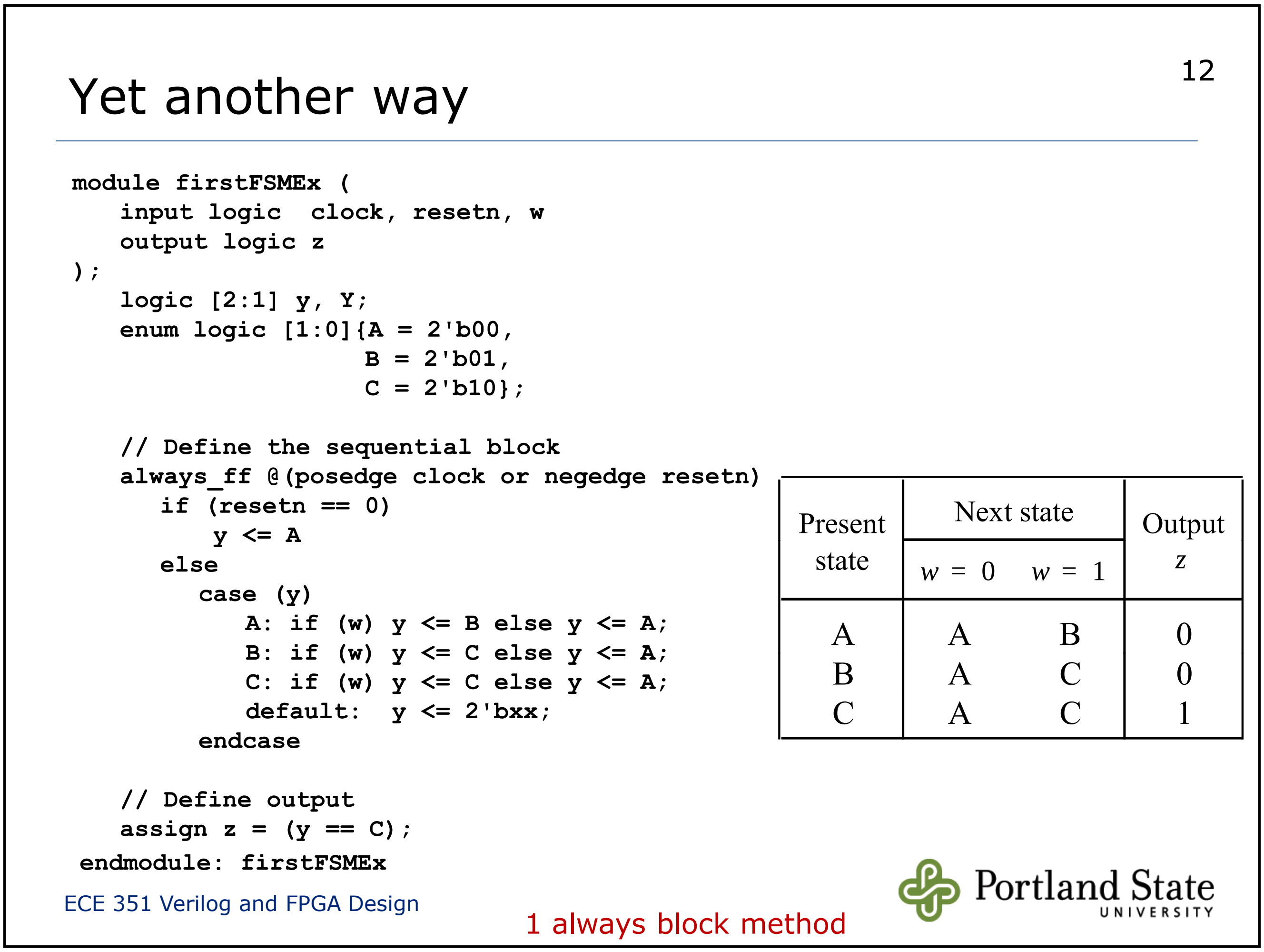
Another way

```
module firstFSMEx (
   input logic  clock, resetn, w,
   output logic z
);
   logic [2:1] y, Y;
   enum logic [1:0]{A = 2'b00,
                    B = 2'b01,
                    C = 2'b10};

   // next state logic (comb. block)
   always_comb
      case (y)
         A: if (w)   Y = B;
            else     Y = A;

         B: if (w)   Y = C;
            else     Y = A;

         C: if (w)   Y = C;
            else     Y = A;

         default:    Y = 2'bxx;
      endcase
```

```
// output logic (comb. block)
always_comb
         case (y)
                  A: z = 1'b0;
                  B: z = 1'b0;
                  C: z = 1'b1;
                  default: z = 1'bx;
         endcase

// Define the sequential block
always_ff @(posedge clock or negedge resetn)
   if (resetn == 0)
         y <= A;
   else
         y <= Y;

endmodule: firstFSMEx
```

3 always block method

Portland State UNIVERSITY



Yet another way

```
module firstFSMEx (
   input logic  clock, resetn, w
   output logic z
);
   logic [2:1] y, Y;
   enum logic [1:0]{A = 2'b00,
                    B = 2'b01,
                    C = 2'b10};

   // Define the sequential block
   always_ff @(posedge clock or negedge resetn)
      if (resetn == 0)
         y <= A
      else
         case (y)
            A: if (w) y <= B else y <= A;
            B: if (w) y <= C else y <= A;
            C: if (w) y <= C else y <= A;
            default:  y <= 2'bxx;
         endcase

   // Define output
   assign z = (y == C);
 endmodule: firstFSMEx
```

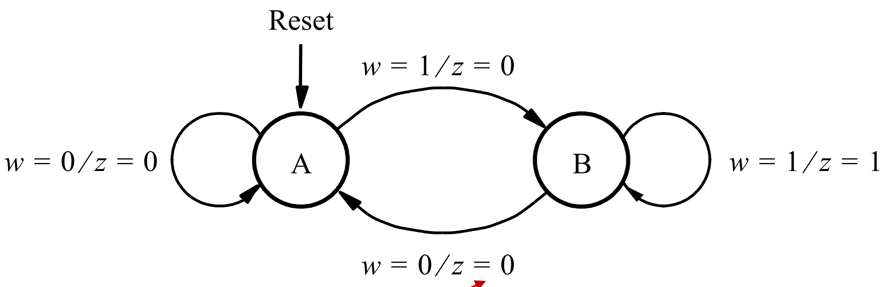
Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1

1 always block method

Portland State UNIVERSITY

Mealy state model

Clockcycle:	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0



output is in transition (Mealy)

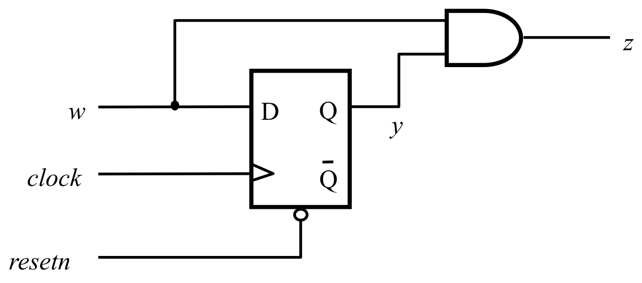
Mealy state model (cont'd)

Present state	Next state		Output *z*	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

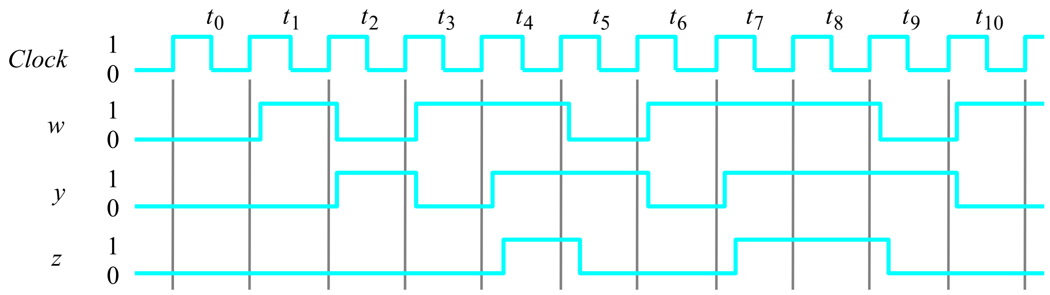
	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

Portland State UNIVERSITY

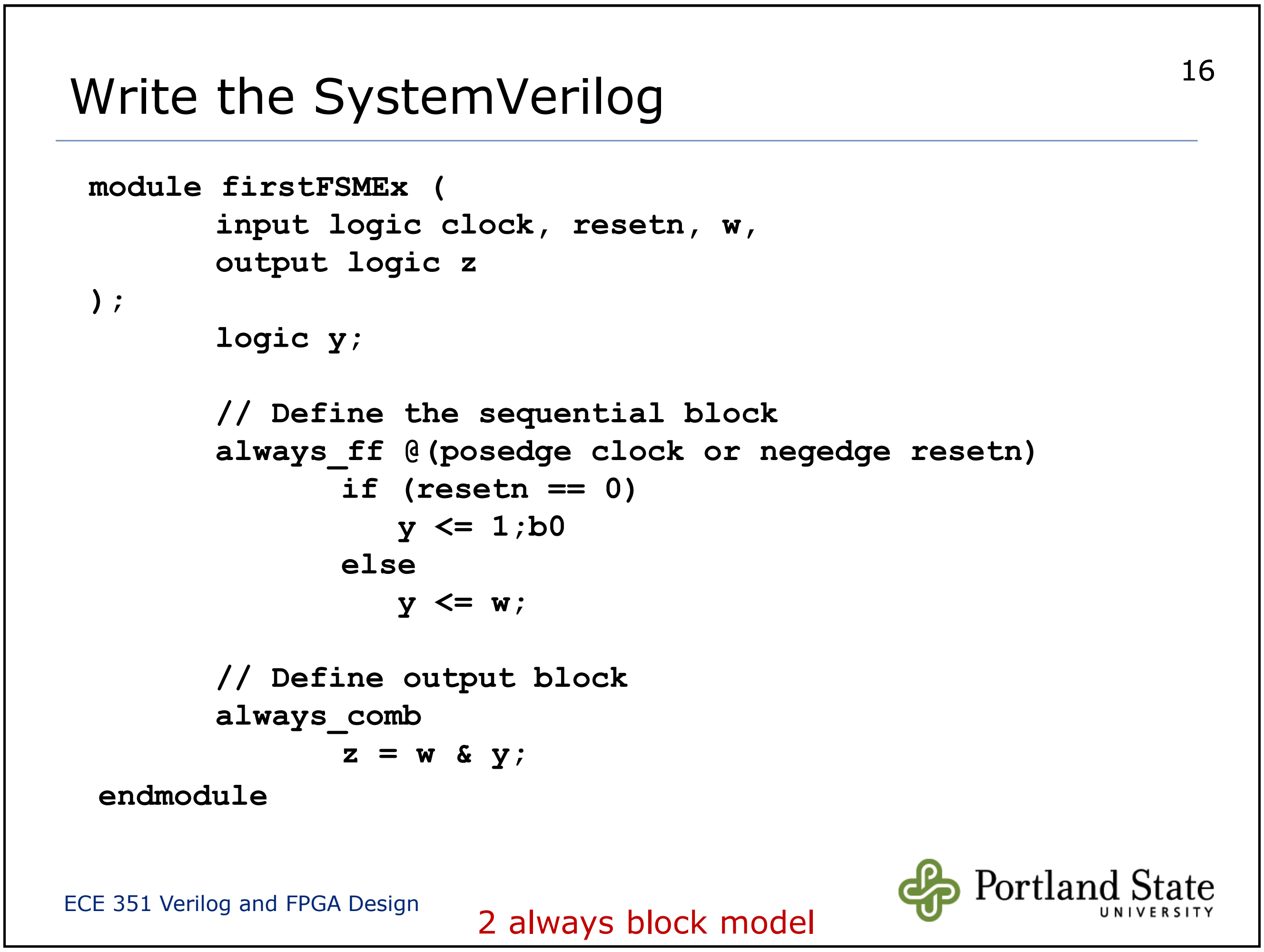
Mealy implementation



(a) Circuit



(b) Timing diagram



Write the SystemVerilog

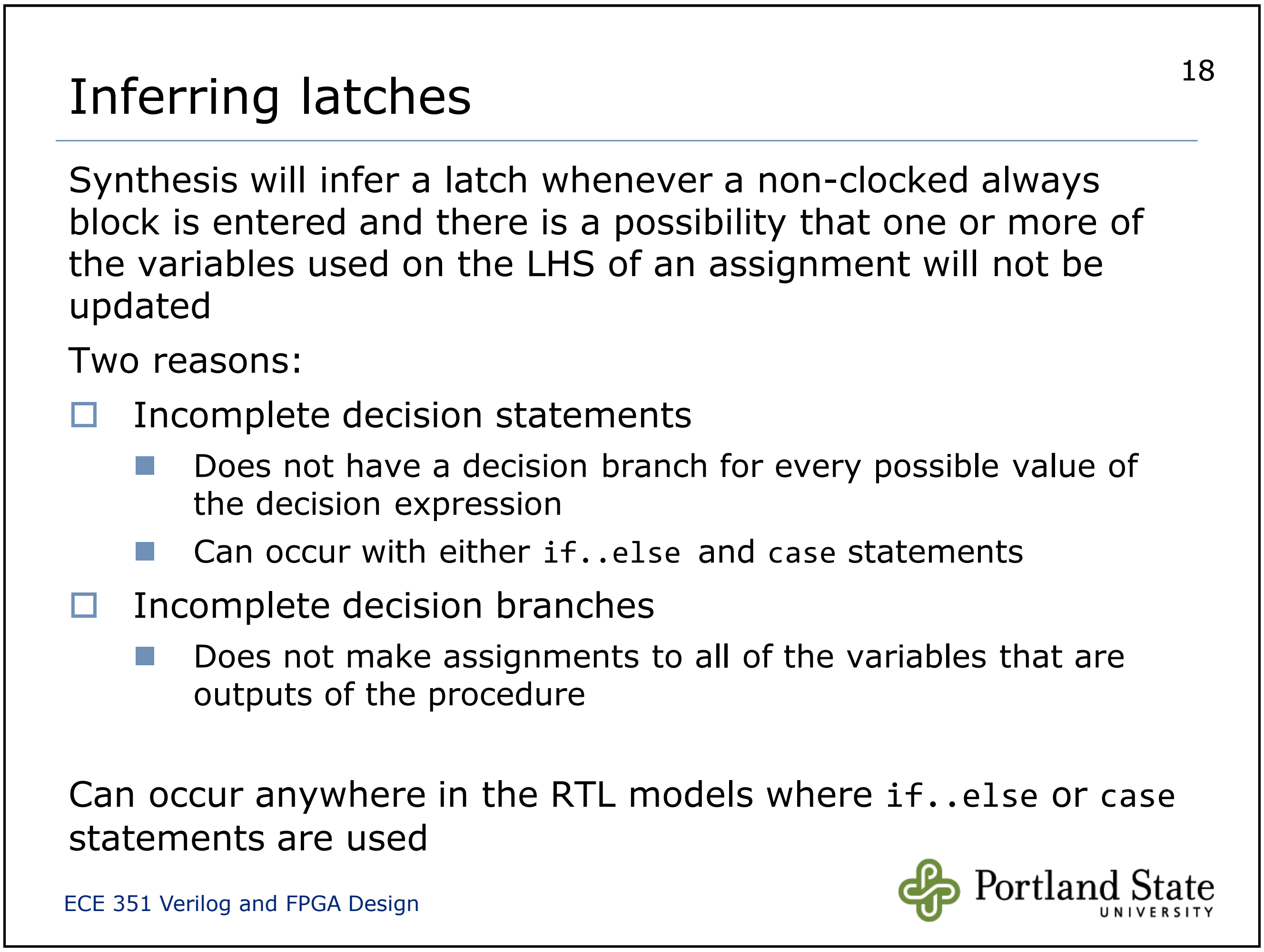
```
module firstFSMEx (
        input logic clock, resetn, w,
        output logic z
);
        logic y;

        // Define the sequential block
        always_ff @(posedge clock or negedge resetn)
                if (resetn == 0)
                   y <= 1;b0
                else
                   y <= w;

        // Define output block
        always_comb
                z = w & y;
 endmodule
```

2 always block model

Coding to avoid unintentional latches

Sources:

- RTL Modeling with System Verilog for Simulation and Synthesis, Stuart Sutherland



Inferring latches

Synthesis will infer a latch whenever a non-clocked always block is entered and there is a possibility that one or more of the variables used on the LHS of an assignment will not be updated

Two reasons:

- Incomplete decision statements
  - Does not have a decision branch for every possible value of the decision expression
  - Can occur with either `if..else` and `case` statements
- Incomplete decision branches
  - Does not make assignments to all of the variables that are outputs of the procedure

Can occur anywhere in the RTL models where `if..else` or `case` statements are used

Portland State UNIVERSITY

Examples of inferring latches (cont'd)

Incomplete decision statements

```
always_comb begin // 3-to-1 multiplexor
  case (select)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
  endcase
end
```

Incomplete decision branches

```
always_comb begin // add or subtract
  case (mode)
    1'b0: add_result      = a + b;
    1'b1: subtract_result = a - b;
  endcase
end
```

Inadvertent latches in state machine models

```
always_comb begin
  case (current_state)    // 5 states, binary count encoding
    3'b000: control_bus = 4'b0000;
    3'b001: control_bus = 4'b1010;
    3'b011: control_bus = 4'b1110;
    3'b100: control_bus = 4'b0110;
    3'b101: control_bus = 4'b0101;
  endcase
end
```

```
always_comb begin
  case (1'b1)           // 5 states, one-hot encoding
    current_state[0]: control_bus = 4'b0110;
    current_state[1]: control_bus = 4'b1010;
    current_state[2]: control_bus = 4'b1110;
    current_state[3]: control_bus = 4'b0110;
    current_state[4]: control_bus = 4'b0101;
  endcase
end
```

Portland State UNIVERSITY

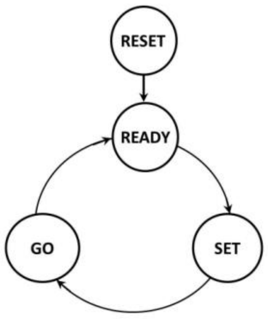
Avoiding inferred latches

- Best practice is to avoid them by fully specifying all of the output values of decision statements for all the possible decision branches

But if you can’t…or decide not to:

- Five common coding styles:
  - Use a `default` case item to assign known output values
  - Use a pre-case assignment before the `case` statement that assigns known known output values
  - Use the `unique` and `priority` decision modifiers
  - ~~Use the obsolete…and dangerous…`full_case synthesis`~~
  - ~~Use an `X` assignment value to indicate don’t care conditio~~ns

Example 9.2



```
module simple_fsm
(input  logic clk, rstN,
 output logic get_ready, get_set, get_going
);

  typedef enum logic [2:0] {RESET = 3'b000, // Johnson Count
                            READY = 3'b001,
                            SET   = 3'b011,
                            GO    = 3'b111} states_t;

  states_t current_state, next_state;

  // state sequencer
  always_ff @(posedge clk or negedge rstN)  // async reset
    if (!rstN) current_state <= RESET;      // active-low reset
    else       current_state <= next_state;

  // next state decoder
  always_comb begin
    case (current_state)
      RESET  : next_state = READY;
      READY  : next_state = SET;
      SET    : next_state = GO;
      GO     : next_state = READY;
    endcase
  end

  // output decoder [illegible]
  always_comb begin
    [illegible]
    case (current_state)
      READY  : get_ready = '1;
      SET    : get_set   = '1;
      GO     : get_going = '1;
    endcase
  end
endmodule: simple_fsm
```

Style 1:Default case item w/ known values

```
// next state decoder
always_comb begin
  case (current_state)
    RESET  : next_state = READY;
    READY  : next_state = SET;
    SET    : next_state = GO;
    GO     : next_state = READY;
  endcase
end
```

```
always_comb begin
  case (current_state)
    RESET   : next_state = READY;
    READY   : next_state = SET;
    SET     : next_state = GO;
    GO      : next_state = READY;
    default: next_state = RESET;   // reset if error
  endcase
end
```

Portland State UNIVERSITY

Style 2: Pre-case assignment, known values

```
// next state decoder
always_comb begin
  case (current_state)
    RESET  : next_state = READY;
    READY  : next_state = SET;
    SET    : next_state = GO;
    GO     : next_state = READY;
  endcase
end
```

```
// output decoder (using pre-case assignment)
always_comb begin
  {get_ready, get_set, get_going} = 3'b000; // clear bits
  case (current_state)
    READY  : get_ready = '1;
    SET    : get_set   = '1;
    GO     : get_going = '1;
  endcase
end
```

```
always_comb begin
  next_state = RESET; // default to reset if invalid state
  case (current_state)
    RESET  : next_state = READY;
    READY  : next_state = SET;
    SET    : next_state = GO;
    GO     : next_state = READY;
  endcase
end
```

Style 3: unique and priority modifiers

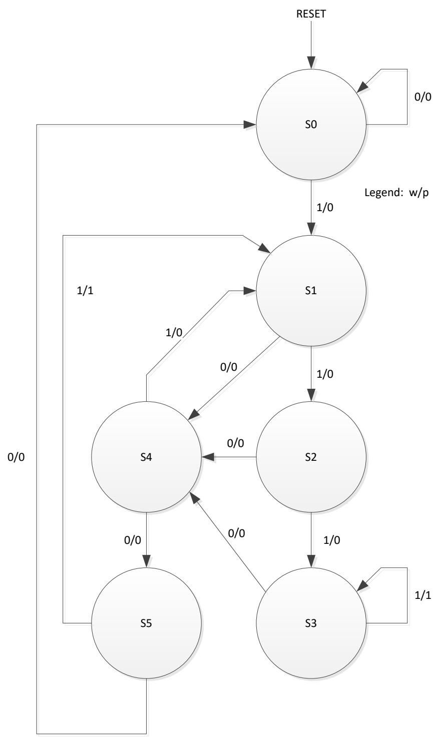
```
// next state decoder
always_comb begin
  case (current_state)
    RESET  : next_state = READY;
    READY  : next_state = SET;
    SET    : next_state = GO;
    GO     : next_state = READY;
  endcase
end
```

```
always_comb begin
  unique case (current_state)
    RESET  : next_state = READY;
    READY  : next_state = SET;
    SET    : next_state = GO;
    GO     : next_state = READY;
  endcase
end
```

Portland State UNIVERSITY

Let’s write some code for a FSM

A more complex pattern recognizer



A pattern recognizer state machine has a single input *w* and an output *p*. The pattern recognizer should generate *p = 1* when the previous 4 values of *w* were 1001 or 1111; otherwise, *p = 0*. Overlapping input patterns are allowed and should generate the correct output.

An example of the desired behavior for the state machine is:

w: 0101111001100111111

***p*: 0000001001000100111**

The inputs to the module are *w*, *clock*, and *reset*. There is one output, *p*.

Next Time

- Topics:
  - Functions and tasks
  - Review midterm exam solutions
- You should:
  - Review Sutherland Ch 6 (Functions and Tasks)
  - Read Sutherland Ch 9
- Homework, projects and quizzes
  - Homework #3 will be released Thu, 13-May. Due to D2L by 10:00 PM on Mon, 24-May

Portland State UNIVERSITY