EE1D1: Digital Systems A

BSc. EE, year 1, 2025-2026, lecture 8

SystemVerilog Sequential Circuits

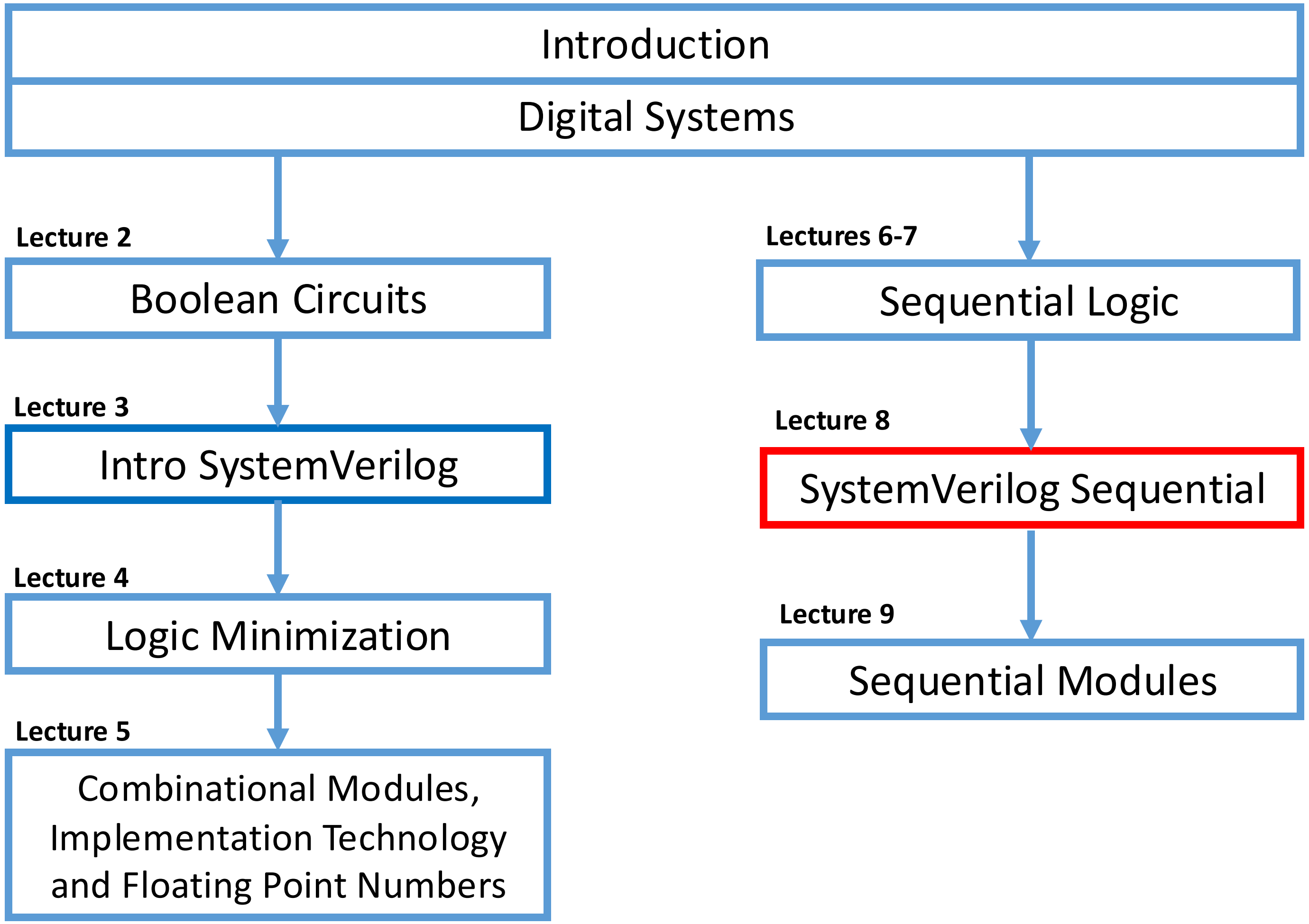
Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

TU Delft

Delft University of Technology

Recap


Introduction
Digital Systems
Lecture 2
Boolean Circuits
Lecture 3
Intro SystemVerilog
Lecture 4
Logic Minimization
Lecture 5
Combinational Modules, Implementation Technology and Floating Point Numbers
Lectures 6-7
Sequential Logic
Lecture 8
SystemVerilog Sequential
Lecture 9
Sequential Modules

Learning Objectives

As student you should be able to:

- Describe latches and flip-flops in SystemVerilog.
- Use the always statement in SystemVerilog to describe sequential circuits and combinational circuits.
- Describe Finite-State Machines in SystemVerilog.
- Describe counters in SystemVerilog.
- Simulate a sequential circuit in SystemVerilog.

Overview

- Recap
- D Flip-flop and Latch
- The Always Statement
- Finite State Machines
- Counters

Sections in book DDCA: 4.4 – 4.6

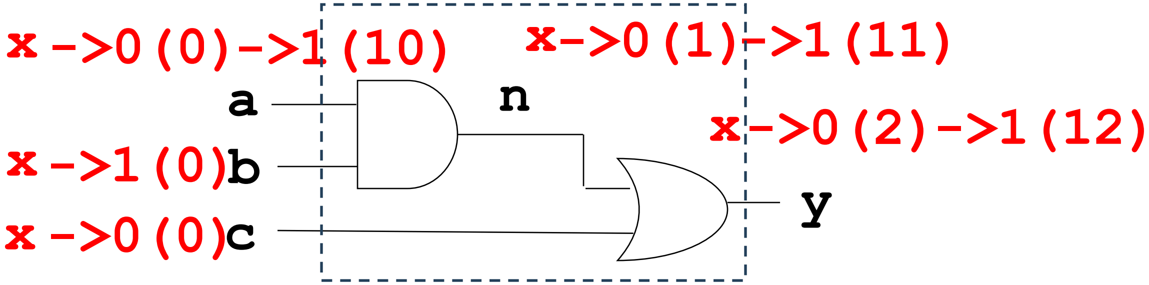
Learned earlier

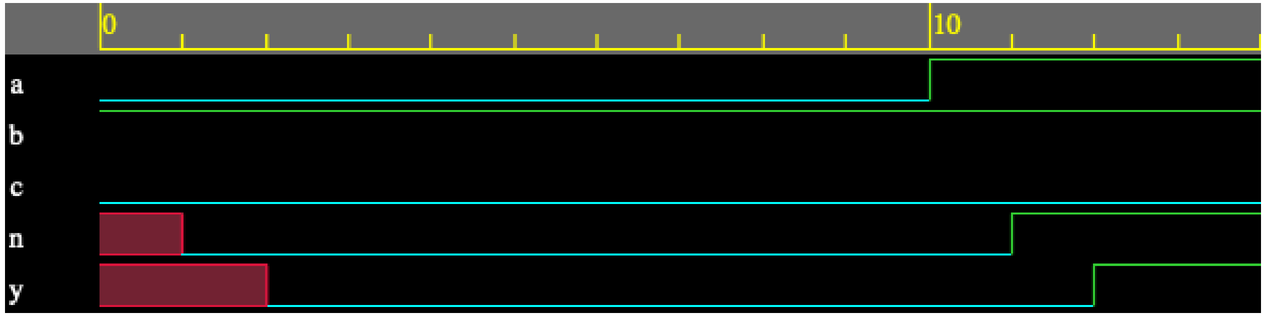
```
module func1(input  logic a, b, c,
                output logic y);
   logic n;

   assign #1 y = n | c;
   assign #1 n = a & b;
endmodule
```

Order is not important

```
module testfunc1();
   logic a, b, c, y;

   func1 dut(a, b, c, y);

   initial begin
           a = 0; b = 1; c = 0;
      #10; a = 1;
   end
endmodule
```

Simulation with testbench





Within this initial block, order is important.
Starting from begin, statements are executed one after another (with specified delays)

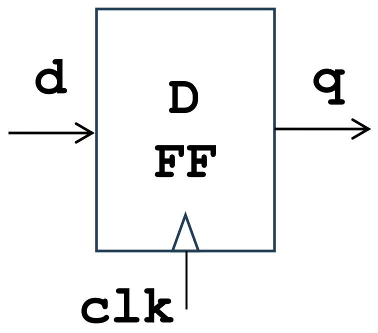
Sequential Logic

- SystemVerilog uses **idioms** to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware after synthesis
- So, you should follow the recommended methods to obtain reliable simulation results as well as good synthesis results.
- Always keep in mind that – although you may be using a high-level description - you are describing hardware: registers (flip-flops) and combinational logic, orchestrated by the clock.

D flip-flop and latch

Describing a D flip-flop

```
module dff(input  logic clk,
           input  logic d,
           output logic q);

   always_ff @(posedge clk)   // on a rising clock edge
      q <= d;                 // q gets the value of d

endmodule
```



@(....) is the sensitivity list. It describes what triggers the execution of the always statement.

Describing a D flip-flop with reset

It is good practice to use resettable registers so that on powerup you can put your system in a known state.

synchronous reset => reset on rising clock edge

```
always_ff @(posedge clk)
   if (reset) q <= 0;
   else q <= d;
```

asynchronous reset => reset at any moment reset becomes 1

```
always_ff @(posedge clk, posedge reset)
   if (reset) q <= 0;
   else q <= d;
```

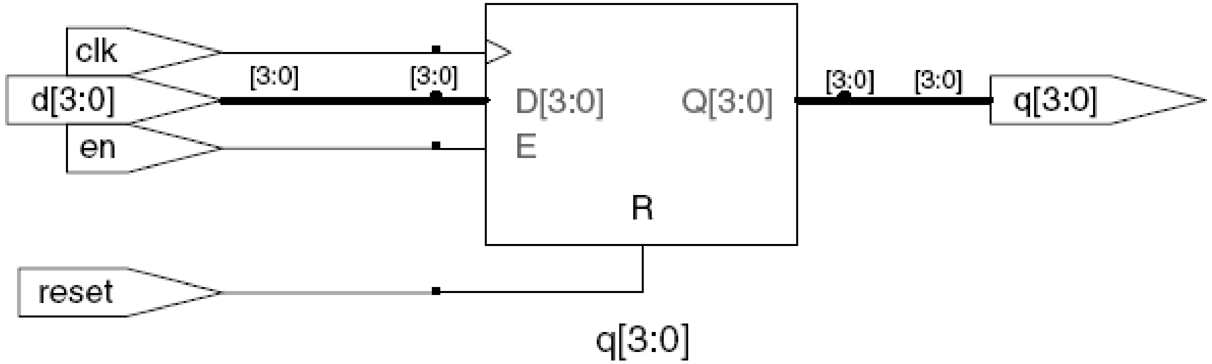
Registers

The following example shows a 4-bit register with asynchronous reset and enable.

It retains its old value if both reset and en are FALSE.

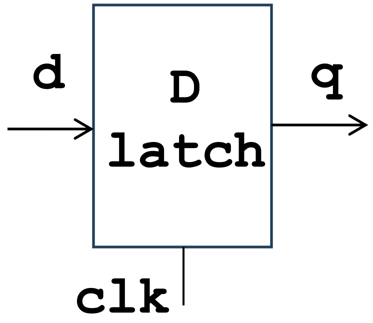
```
module register4bit(input  logic       clk,
                    input  logic       reset,
                    input  logic       en,
                    input  logic [3:0] d,
                    output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if       (reset) q <= 4'b0;
    else if (en)     q <= d;

endmodule
```

Synthesis:



Describing a latch

The **always_latch** statement is used to describe a latch




```
always_latch
   if (clk) q <= d;   // when clk = 1, q gets value of d
                      // when clk = 0, q remembers its value
```

A sensitivity list is not allowed; the **always_latch** statement is evaluated whenever one of the inputs (**clk** or **d**) changes value.

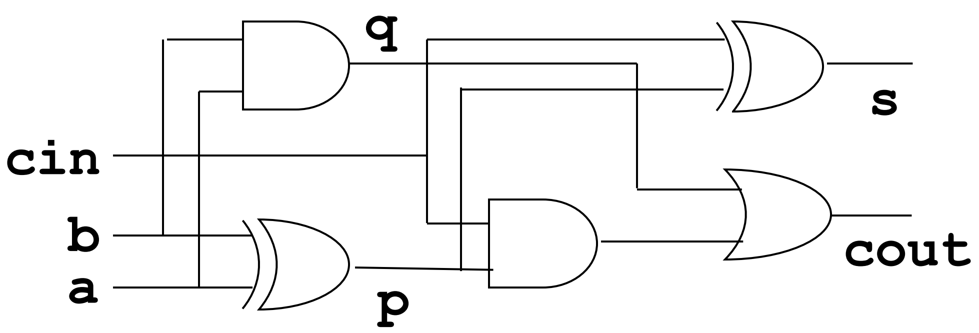
Normally, it is not a good idea to use latches in your circuit. They are transparent as long as clk = 1, so problematic combinational feedback loops may occur. Also, timing of input signals is more difficult to control.

The Always Statement

Using always statement to describe a combinational circuit

An **always_comb** statement can be used to describe a combinational circuit.

```
always_comb
begin
   p = a ^ b;
   q = a & b;
   s = p ^ cin;
   cout = q | (p & cin);
end
```



A sensitivity list is not allowed with **always_comb**; the block will be evaluated whenever one of the inputs (**a**, **b** and **cin**) change.

It is important that you specify *the value of all outputs in all cases*. Otherwise, unwanted latches may be created during synthesis.

```
always_comb
   if (s) y = x1;
   else y = x0;
```

ok

```
always_comb
   if (s) y = x1;
```

not ok

Using the always statement

A general **`always`** statement can also be used.

When it has no sensitivity list, the statement is evaluated whenever one of the inputs change:

```
always
```

When it has a sensitivity list, the statement is evaluated whenever one of the signals in the sensitivity list changes value:

```
always @(a, b, cin)
```

where signal names in the sensitivity list may be preceded by **`posedge`** or **`negedge`** to indicate that evaluation is triggered by only a rising edge or falling edge event .

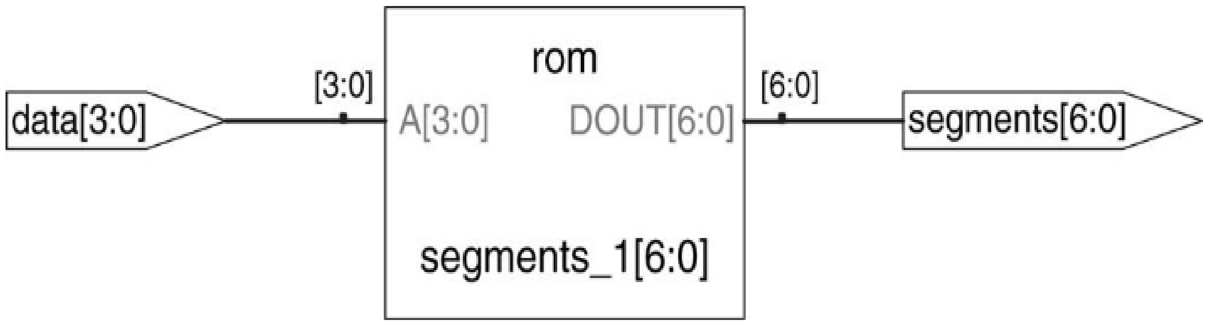
For flip-flops (registers), latches and combinational circuits, the usage of respectively **`always_ff`**, **`always_latch`** and **`always_comb`** is preferred.
The use of **`always`** should be restricted to testbenches.

Case Statement

A combinational circuit for a seven-segment display decoder that uses a case statement

```
module sevenseg(input logic [3:0] data,
                output logic [6:0] segments);
always_comb
   case(data)
       //                    abc_defg
       0:        segments = 7'b111_1110;
       1:        segments = 7'b011_0000;
       2:        segments = 7'b110_1101;
       3:        segments = 7'b111_1001;
       4:        segments = 7'b011_0011;
       5:        segments = 7'b101_1011;
       6:        segments = 7'b101_1111;
       7:        segments = 7'b111_0000;
       8:        segments = 7'b111_1111;
       9:        segments = 7'b111_0011;
       default:  segments = 7'b000_0000;
   endcase
endmodule
```

Synthesis:



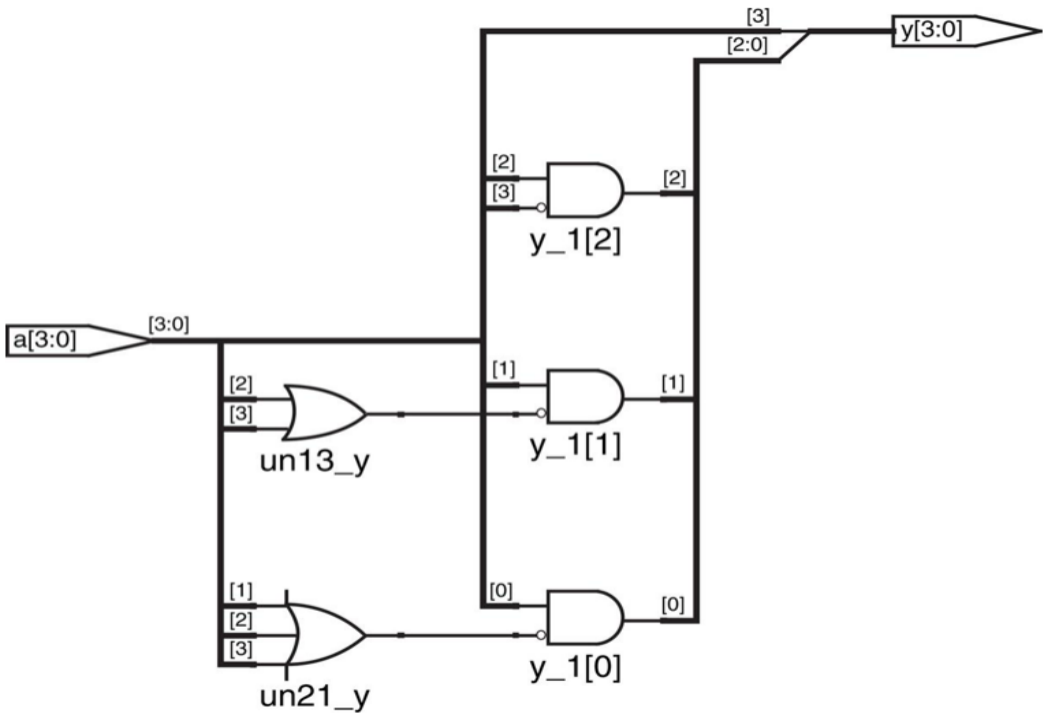
Without default, outputs are not specified for all cases and latches will be inferred during synthesis!

If Statement

A priority circuit that uses a nested if-else statement

```
module priorityckt(input logic [3:0] a,
                   output logic [3:0] y);

   always_comb
      if      (a[3]) y = 4'b1000;
      else if (a[2]) y = 4'b0100;
      else if (a[1]) y = 4'b0010;
      else if (a[0]) y = 4'b0001;
      else           y = 4'b0000;
endmodule
```

Synthesis:

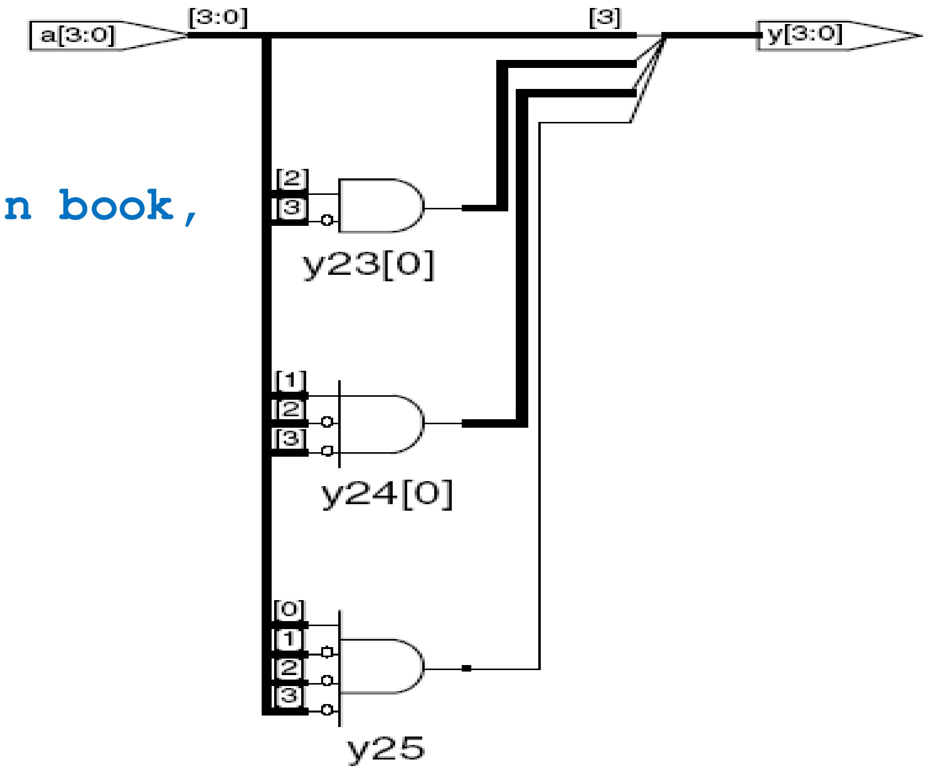


Truth Tables with Don't Cares

- Truth tables may include don't cares to allow more logic simplification.
- The following shows how to describe the previous priority circuit with the inside keyword, which allows don't cares to be used.

```
module priority_case_dc(input  logic [3:0] a,
                        output logic [3:0] y);
  always_comb
    case(a) inside            // ? = don't care
      4'b1???: y = 4'b1000; // casez(a), as used in book,
      4'b01??: y = 4'b0100; // is obsolete
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```

Synthesis:



Blocking vs. Nonblocking Assignment

In an always statement, there is an important difference between using <= or = in an assignment.

- <= is a **nonblocking** assignment: The signal on the left receives the value only after the block has been evaluated. So, the assignments are deferred.
- = is a **blocking** assignment. The signal on the left receives the value immediately after the assignment. The execution of a subsequent statement is “blocked” until this has been done. So, the assignments are immediate, and the order is important.

```
always_ff @(posedge clk)
begin
   n1 <= d;  // nonblocking (deferred)
   q  <= n1; // nonblocking (deferred)
end
```

```
always_ff @(posedge clk)
begin
   n1 = d;  // blocking (immediate)
   q  = n1; // blocking (immediate)
end
```

Suppose that initially d = 1, n1 = 0 and q = 0

Then, after 1 rising clock edge:

n1 = 1 and q = 0

n1 = 1 and q = 1

not different from using q = d

And after a second rising clock edge:

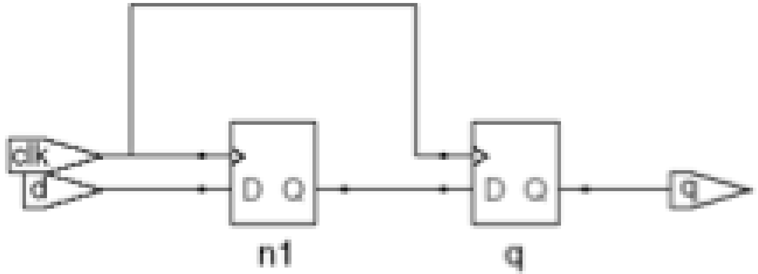
n1 = 1 and q = 1

n1 = 1 and q = 1

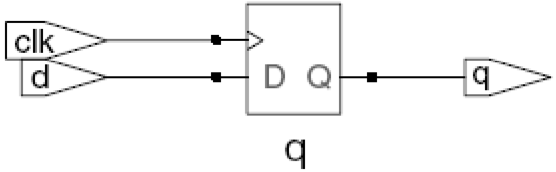
Blocking vs. Nonblocking Assignment

The different behavior also results in a different synthesis results.
(BTW: A synchronizer uses 2 D flip-flops in series to synchronize external inputs with clock.)

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff @(posedge clk)
  begin
    n1 <= d;  // nonblocking
    q  <= n1; // nonblocking
  end
endmodule
```



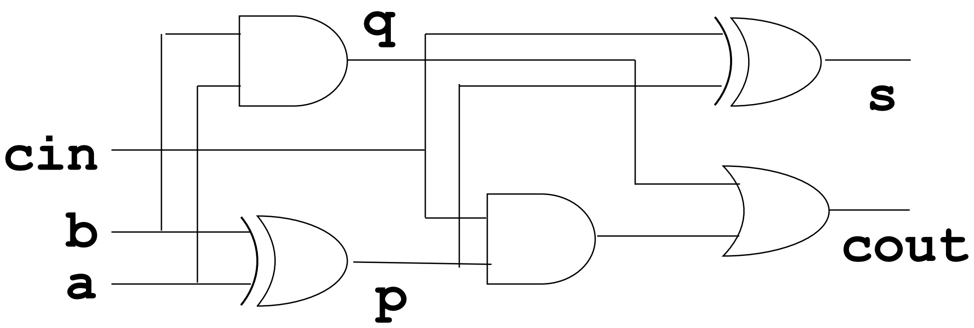

```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
  begin
    n1 = d;  // blocking
    q  = n1; // blocking
  end
endmodule
```



Blocking vs. Nonblocking Assignment

For **always_comb** blocks it is ok to use blocking assignments:

```
always_comb
begin
   p = a ^ b;
   q = a & b;
   s = p ^ cin;
   cout = q | (p & cin);
end
```



Summary of rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @(posedge clk)` and nonblocking assignments (`<=`)

```
always_ff @(posedge clk)
  q <= d; // nonblocking
```

- **Simple combinational logic:** use continuous assignments (`assign…`)

```
assign y = a & b;
```

- **More complex combinational logic:** use `always_comb` and blocking assignments (`=`)
- In an `always_comb` block, assign a value to *each* output for *all* input combinations.
- Assign a signal in *only one* **`always`** statement or continuous assignment statement.

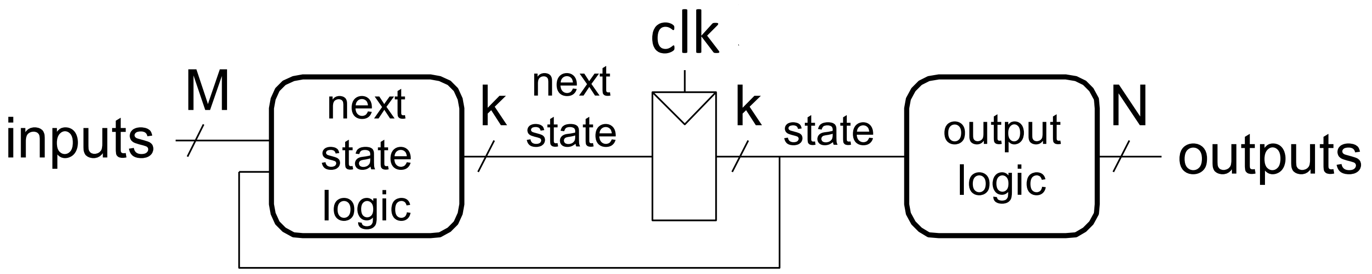
very important

Finite State Machines

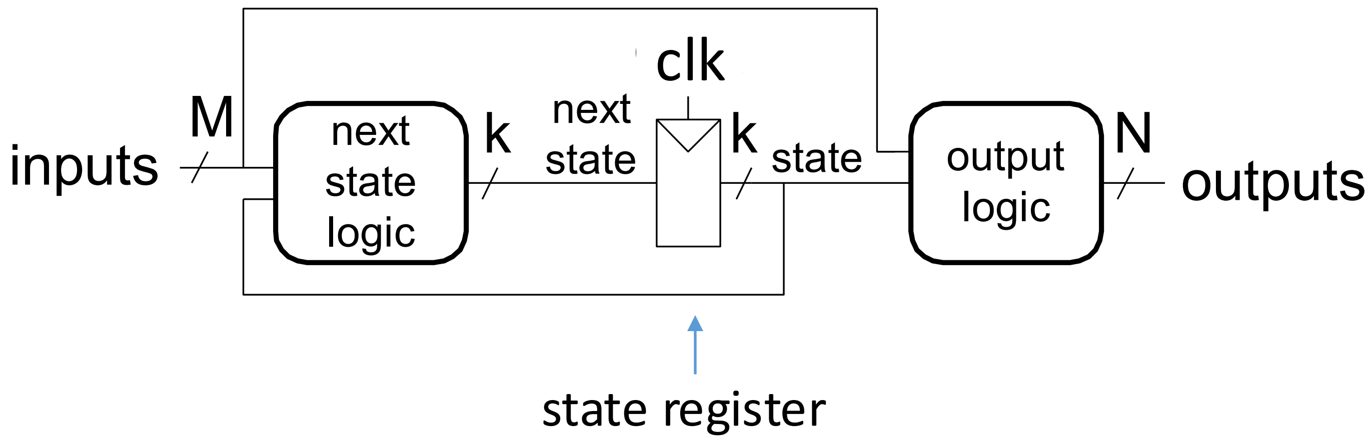
Finite State Machines

- **Three blocks:**
  - next state logic
  - state register
  - output logic

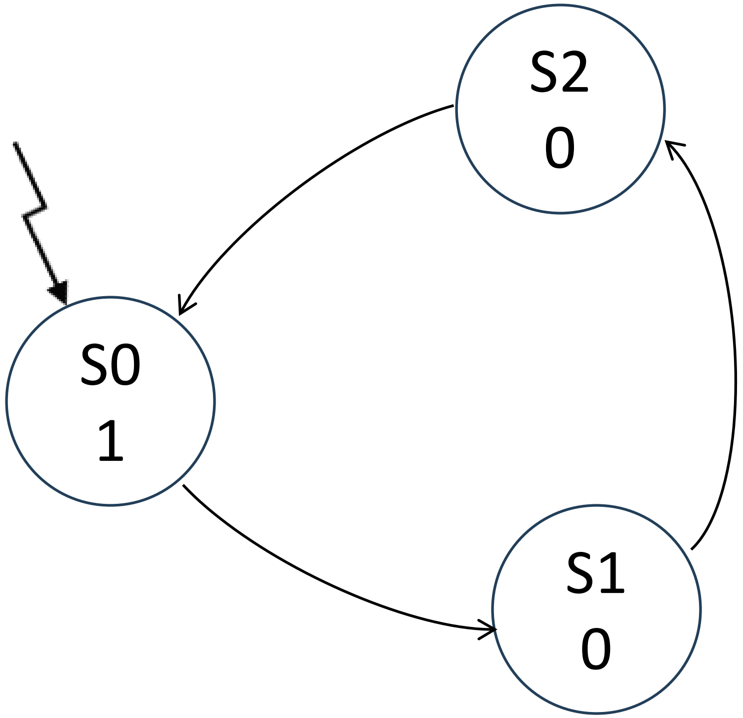
Moore FSM



Mealy FSM

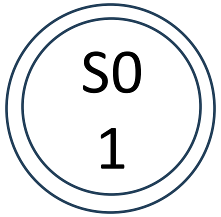


FSM Example 1: Divide by 3



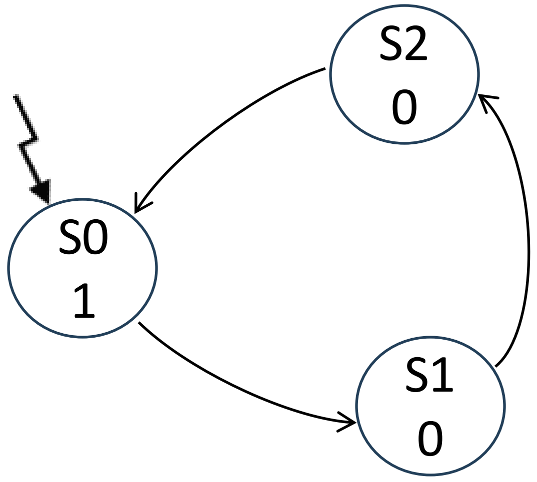
The arrow indicates the reset state

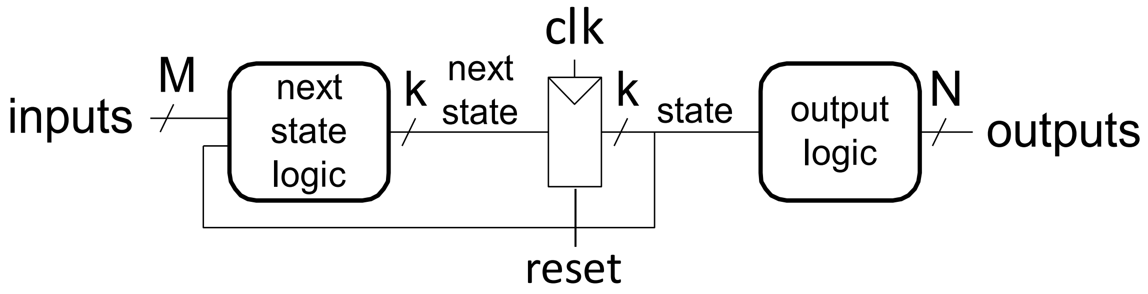
Sometimes also a double circle is used:



FSM Example 1: Divide by 3

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
     case (state)
        S0:      nextstate = S1;
        S1:      nextstate = S2;
        S2:      nextstate = S0;
        default: nextstate = S0;
     endcase

   // output logic
   assign y = (state == S0);
endmodule
```



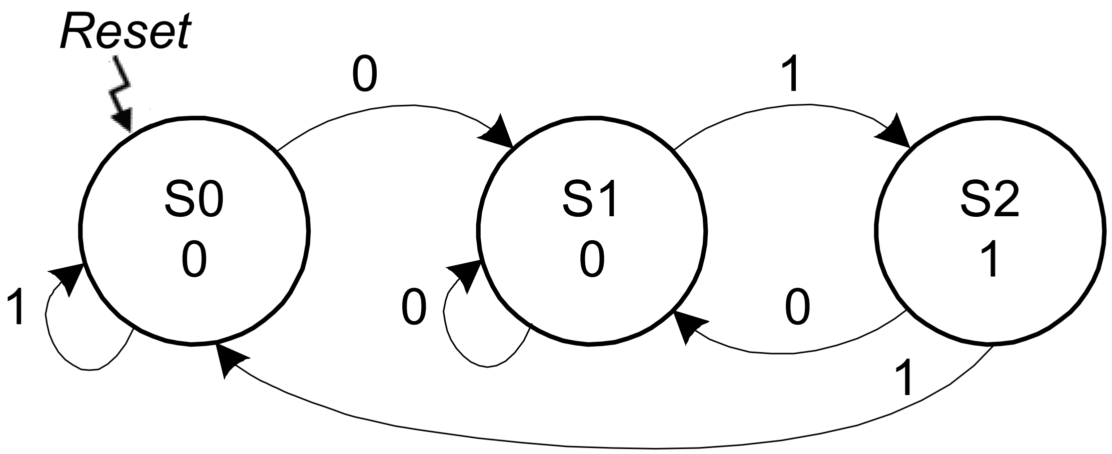


Note that a default case is used in the combinational block, to be sure that output **nextstate** *receives a value in all cases*.

When synthesized, this code will create a circuit with a register that has 2 flip-flops and an asynchronous reset.

FSM Example 2: Sequence Detector

Moore FSM

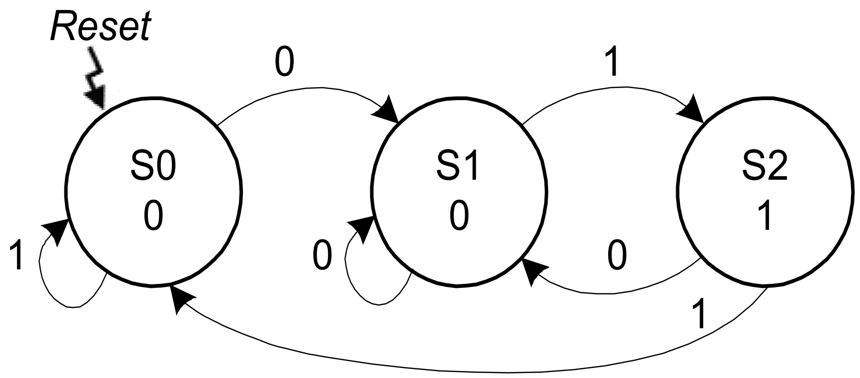


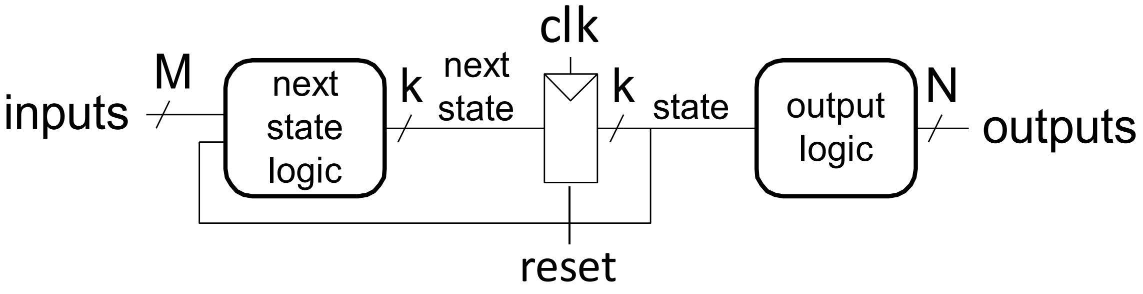
Which sequence will be detected?

Sequence Detector FSM: Moore

```
module patternMoore(input  logic clk, reset, a,
                     output logic y);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
        S0:      if (a) nextstate = S0;
                 else   nextstate = S1;
        S1:      if (a) nextstate = S2;
                 else   nextstate = S1;
        S2:      if (a) nextstate = S0;
                 else   nextstate = S1;
        default: nextstate = S0;
      endcase

   // output logic
   assign y = (state == S2);
endmodule
```

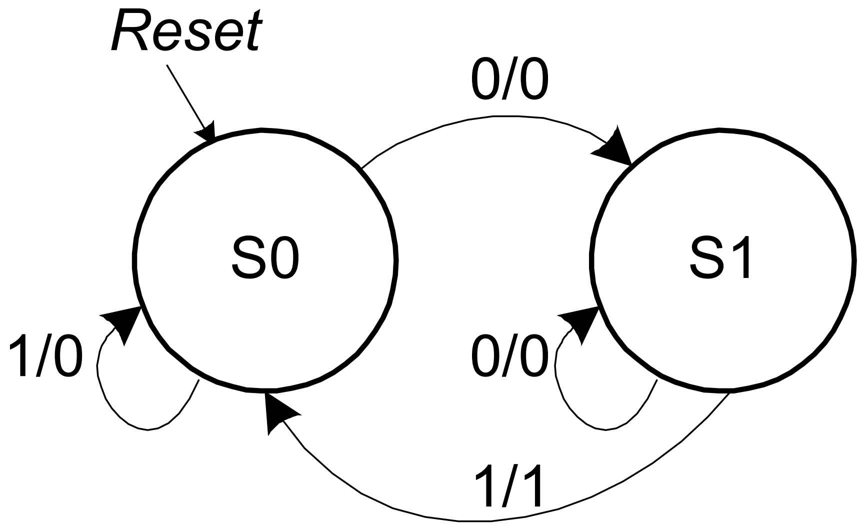
Moore FSM





FSM Example 3: Sequence Detector

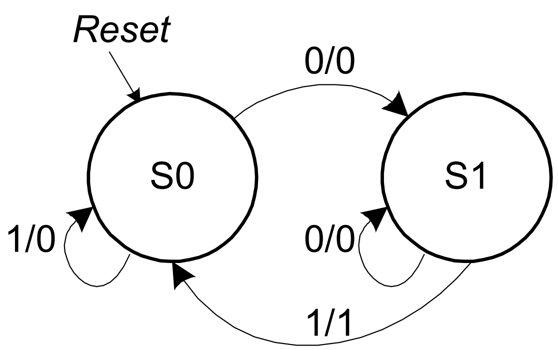
Mealy FSM



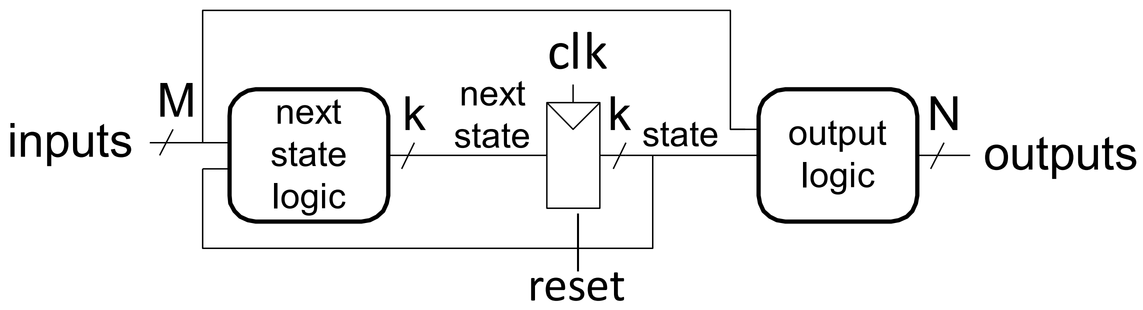
Sequence Detector FSM: Mealy

```
module patternMealy(input  logic clk, reset, a,
                     output logic y);

   typedef enum logic {S0, S1} statetype;
   statetype state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state and output logic combined
   always_comb begin
      // make sure y receives value in all ceses.
      y = 0;
      case (state)
         S0:      if (a) nextstate = S0;
                  else   nextstate = S1;
         S1:      if (a) begin
                           nextstate = S0;
                           y = 1;
                         end
                  else   nextstate = S1;
         default: nextstate = S0;
     endcase
   end
endmodule
```

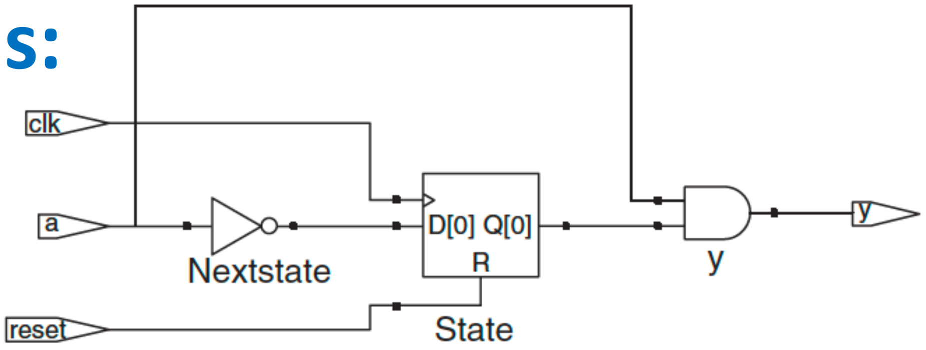
Mealy FSM



Mealy FSM



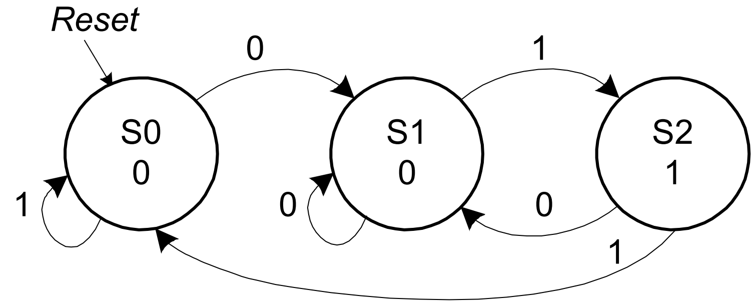
Synthesis:

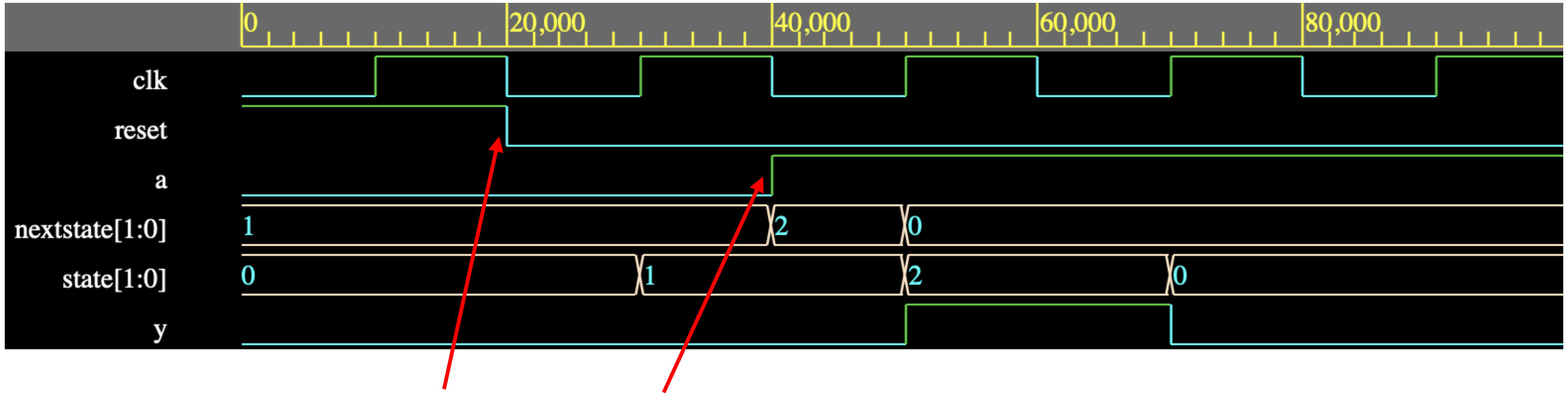


FSM Testbench

```
`timescale 1ns/1ps

module patternMoore_tb();

   logic clk;
   logic reset;
   logic a, y;

   patternMoore dut (clk, reset, a, y);

   initial
      clk = 0;
   always
      #10 clk = ~clk;

   initial begin
          reset = 1; a = 0;
      #20; reset = 0;
      #20;             a = 1;
   end

endmodule
```

Moore FSM





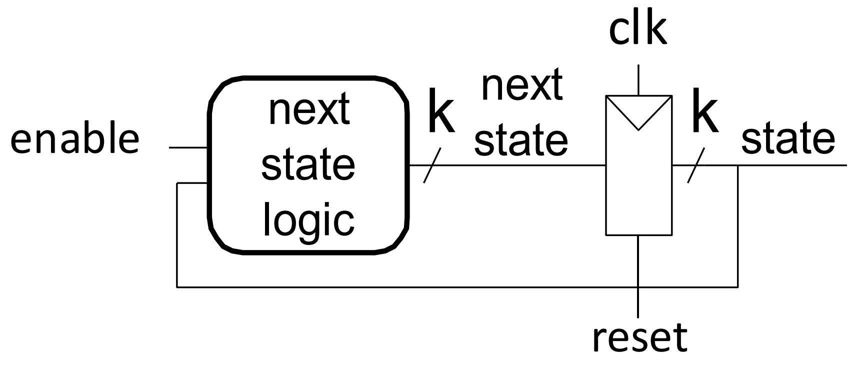
Important: to avoid setup/hold time violations (next lecture), do not change input signals on the rising clock edge. Instead, do it on the negative clock edge.

Counters

Counter

A counter is an FSM that adds 1 to its state in its next state logic.
Usually there is no output logic.

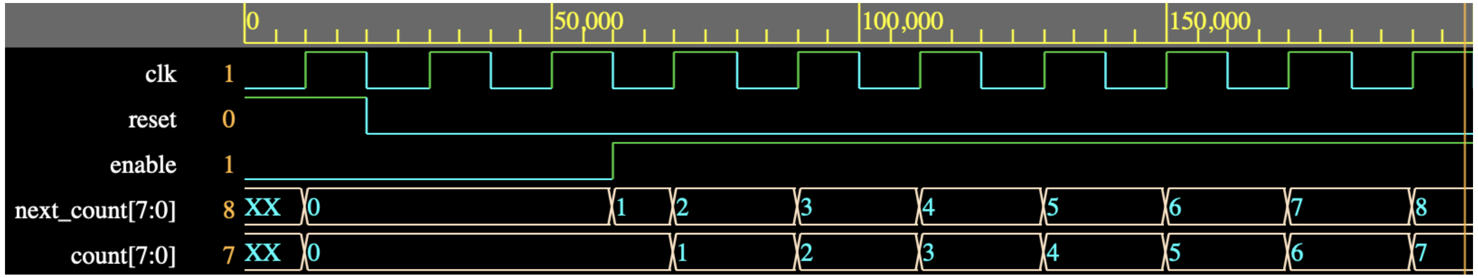
```
module counter(input  logic clk, reset, enable,
               output logic [7:0] count);

   logic [7:0] next_count;

   // register
   always_ff @(posedge clk)
   begin
      if (reset) count <= 0;
      else count <= next_count;
   end

   // next state logic
   always_comb
   begin
      if (enable) next_count = count + 1;
      else next_count = count;
   end
endmodule
```



state ≡ count

Counter simulation

```
module counter_tb();

   logic clk;
   logic reset;
   logic enable;
   logic [7:0] count;

   counter dut (clk, reset, enable, count);

   initial
      clk = 0;
   always
      #10 clk = ~clk;

   initial begin
         reset = 1; enable = 0;
      #20; reset = 0;
      #40;        enable = 1;
   end

endmodule
```



Summary

We have shown how to describe in SystemVerilog:

- Latches and flip-flops.
- More complex combinational circuits like decoders and encoders.
- Finite-State Machines.
- Counters.
- Testbenches for these circuits.