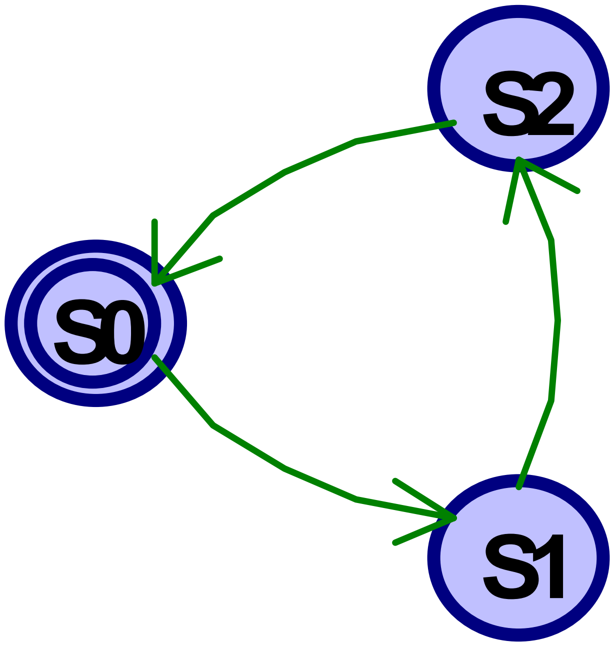
Lecture 4: Continuation of SystemVerilog

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Last Lecture: Divide by 3 FSM

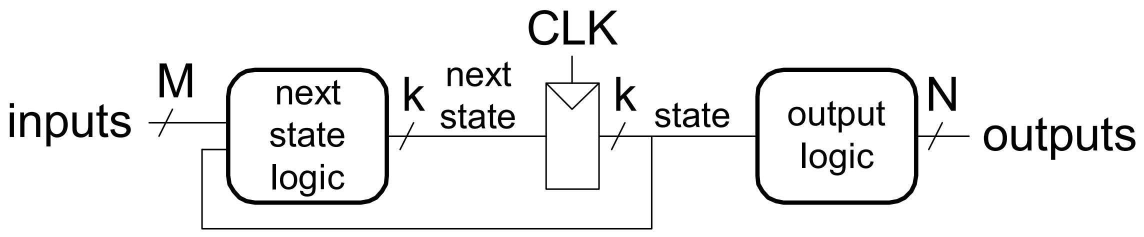
- Output should be "1" every 3 clock cycles



The double circle indicates the reset state

Finite State Machines (FSMs)

- A simple Moore machine looks like the following



FSM Example in SystemVerilog

```
module divideby3FSM (input logic clk, reset_n,
                     output logic q);

  enum logic [1:0] {S0=2'b00, S1=2'b01, S2=2'b10} state; // declare states as enum

  // next state logic and state register
  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n)
      state <= S0;
    else begin
      case (state)
        S0: state <= S1;
        S1: state <= S2;
        S2: state <= S0;
      endcase
    end
  end

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

state transition graph is the same thing as a state transition table, which can be specify as a case statement

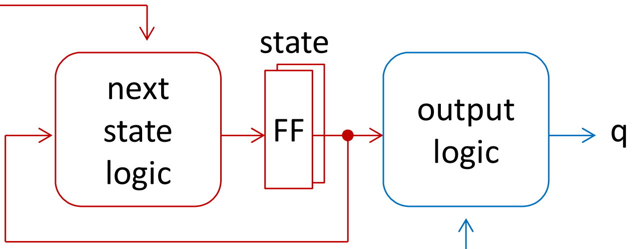
← output is "1" every clock cycles when we are in state S0

FSM Example in SystemVerilog

```
module divideby3FSM (input logic clk, reset_n,
                     output logic q);

  enum logic [1:0] {S0=2'b00, S1=2'b01, S2=2'b10} state; // declare states as enum

  // next state logic and state register
  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n)
      state <= S0;
    else begin
      case (state)
        S0: state <= S1;
        S1: state <= S2;
        S2: state <= S0;
      endcase
    end
  end

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

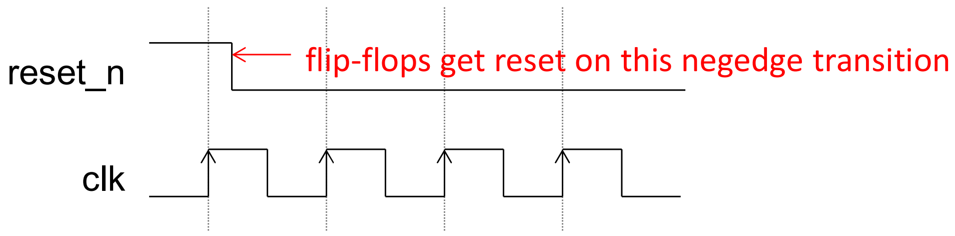
compiler recognizes this "template" should use positive edge-triggered flip-flops w/ negative edge asynchronous reset should be used.

compiler knows this "if" part defines the reset values for flip-flops.



What asynchronous reset means

- "Negative-edge asynchronous reset" means the following:



Continuing with the FSM Example

- What if we want to design the "Divide by 3 FSM" example with just one "always_ff" statement (no separate "assign" statement)?
- Let's assume we still want "q" to be "1" when we are in state "S0".
- Can we put the logic for "q" instead the "always_ff" statement?
- Yes, but a flip-flop will be created for "q"!

FSM Example in SystemVerilog

```
module fsm2 (input logic clk, reset_n, output logic q);
  enum logic [1:0] {S0=2'b00, S1=2'b01, S2=2'b10} state; // declare states as enum

  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n) begin
      state <= S0;
      q <= 1;
    end else begin
      case (state)
        S0: begin
              state <= S1;
              q <= 0;
            end
        S1: begin
              state <= S2;
              q <= 0;
            end
        S2: begin
              state <= S0;
              q <= 1;
            end
      endcase
    end
  end
endmodule
```

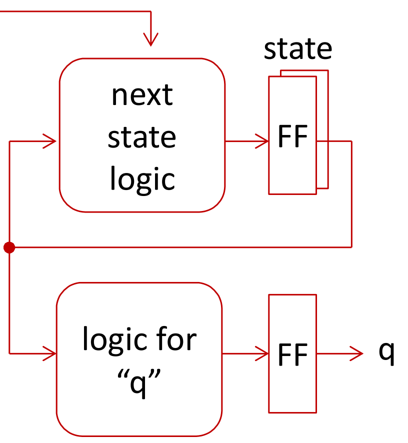
synthesis will generate D-FFs for both "state" and "q"

in order to have the output "q" = 1 when "state" is in S0, have to set the D-FF for "q" in S2 so that the output "q" = 1 when "state" gets to S0.

FSM Example in SystemVerilog

```
module fsm2 (input logic clk, reset_n, output logic q);
  enum logic [1:0] {S0=2'b00, S1=2'b01, S2=2'b10} state; // declare states as enum

  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n) begin
      state <= S0;
      q <= 1;
    end else begin
      case (state)
        S0: begin
                state <= S1;
                q <= 0;
              end
        S1: begin
                state <= S2;
                q <= 0;
              end
        S2: begin
                state <= S0;
                q <= 1;
              end
      endcase
    end
  end
endmodule
```

compiler knows this “if” part defines the reset values for flip-flops.

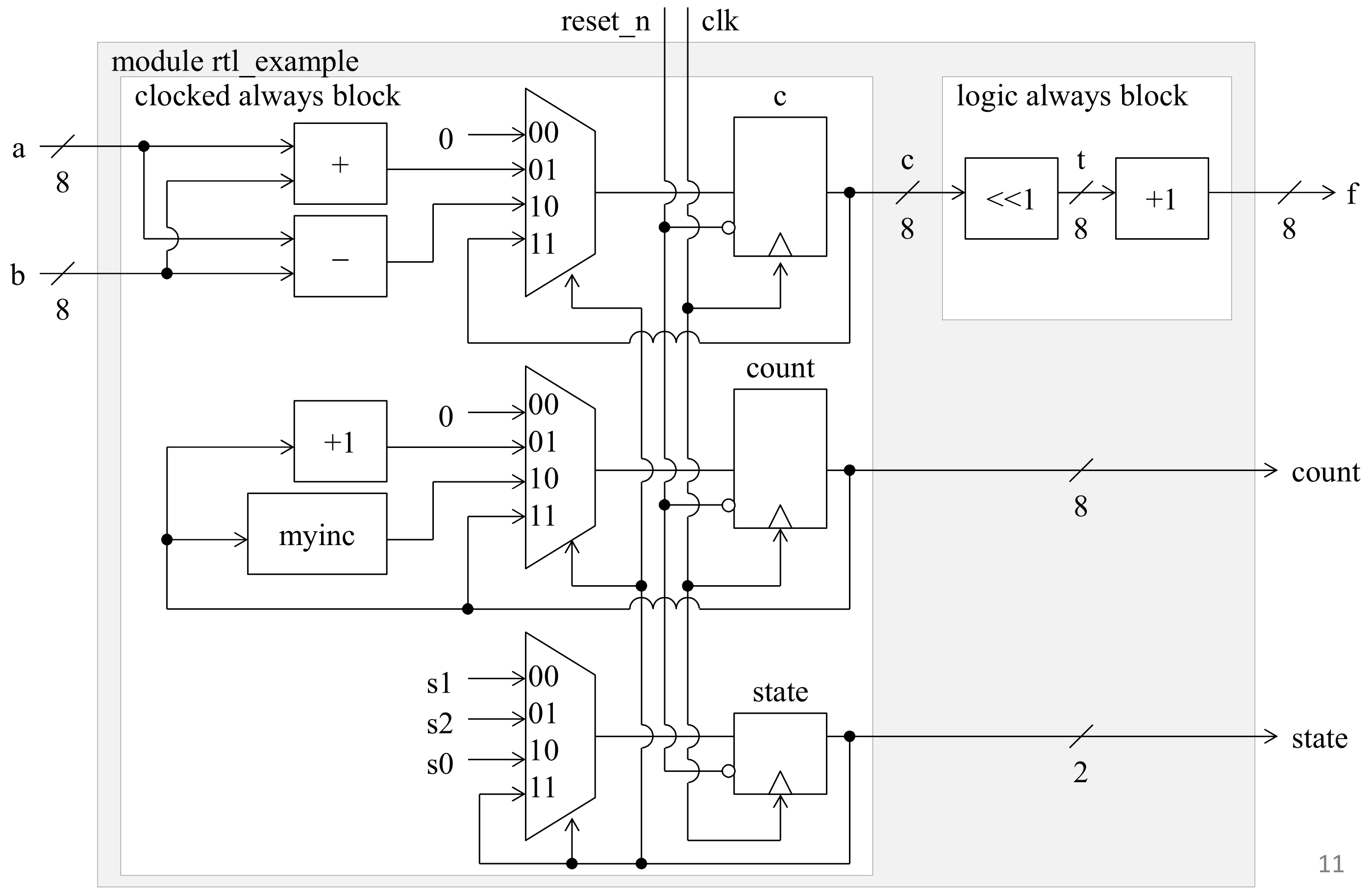


RTL Example

```
module rtl_example (input logic clk, reset_n,
                    input logic [7:0] a, b,
                   output logic [7:0] f, count);

  // default encoding: s0=2’b00, s1=2’b01, s2=2’b10
  enum logic [1:0] {s0, s1, s2} state;
  logic [7:0] c, t;

  function logic [7:0] myinc(input logic [7:0] x);
    logic [7:0] sum;
    logic c; // this is an internal c
  begin
    c = 1;
    for (int i = 0; i < 8; i++) begin
        sum[i] = x[i] ^ c;
        c = c & x[i];
    end
    myinc = sum;
  end
  endfunction

  always_comb
  begin
    t = c << 1;
    f = t + 1;
  end
```

```
  always_ff@(posedge clk, negedge reset_n)
  if (!reset_n) begin
    count <= 0;
    c <= 0; // this c is different than function
    state <= s0;
  end else begin
    case (state)
    s0: begin
          count <= 0;
          c <= 0;
          state <= s1;
        end
    s1: begin
          count <= count + 1;
          c <= a + b;
          state <= s2;
        end
    s2: begin
          count <= myinc(count);
          c <= a - b;
          state <= s0;
        end
    endcase
  end

endmodule
```

RTL Example



Fibonacci Calculator

- F(0) = 0, F(1) = 1
- F(n) = F(n – 1) + F(n – 2), when n > 1

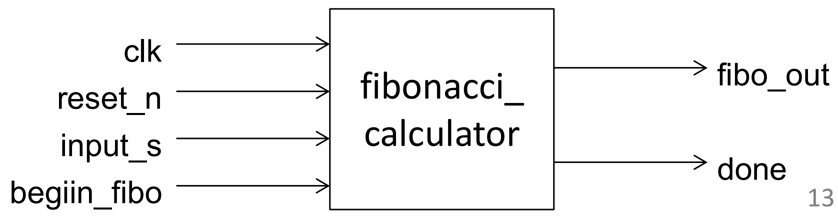
- Examples:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
0	1	1	2	3	5	8	13	21	34

Fibonacci Calculator

- Design a FSM with the interface below.
- `input_s` is “n”, and `fibo_out` is “F(n)”.
- Wait in IDLE state until `begin_fibo`.
- When testbench sees `done==1`, it will check if `fibo_out==` F(`input_s`).

```
module fibonacci_calculator (input logic clk, reset_n,
                             input logic [4:0] input_s,
                             input logic begin_fibo,
                            output logic [15:0] fibo_out,
                            output logic done);
  ...
  always_ff @(posedge clk, negedge reset_n)
  begin
    ...
  end
endmodule
```



Fibonacci Calculator

- Basic idea is to introduce 3 registers:
```
logic [4:0] counter;
logic [15:0] R0, R1;
```
- Set loop counter to “n”
```
counter <= input_s;
```
- Repeat as long as counter is greater than 1 since we already know what F(0) and F(1) are:
```
counter <= counter – 1;
R0 <= R0 + R1;
R1 <= R0;
```
- Finally, set output to “F(n)”
```
done <= 1;
fibo_out <= R0;
```

Fibonacci Calculator

```
module fibonacci_calculator (input logic clk, reset_n,
                             input logic [4:0] input_s,
                             input logic begin_fibo,
                            output logic [15:0] fibo_out,
                            output logic done);
  enum logic [1:0] {IDLE=2'b00, COMPUTE=2'b01, DONE=2'b10} state;

  logic  [4:0] count;
  logic [15:0] R0, R1;

  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n) begin
      state <= IDLE;
      done <= 0;
    end else
      case (state)
        IDLE:
          if (begin_fibo) begin
            count <= input_s;
            R0 <= 1;
            R1 <= 0;
            state <= COMPUTE;
          end
```

in clocked always stmts, D-FFs keep track of previous value, so the missing “else” part will just keep “state” at IDLE.

```
        COMPUTE:
          if (count > 1) begin
            count <= count - 1;
            R0 <= R0 + R1;
            R1 <= R0;
          end else begin
            state <= DONE;
            done <= 1;
            fibo_out <= R0;
          end
        DONE:
          state <= IDLE;
      endcase
  end
endmodule
```

Fibonacci Calculator

- A three state solution is provided.
- Design it using <u>only 2 states</u> (or fewer) w/o creating flip-flops for `fibo_out` or `done`, and should work for n ≥ 1. Hint: use “assign” statements for `fibo_out` and `done`.
- Use provided “tb_fibonacci_calculator.sv” to verify your design using ModelSim.
- Synthesize your 2-state “fibonacci_calculator.sv” design using Quartus.

Fibonacci Calculator: 2 States

```
module fibonacci_calculator (...);

  enum logic {IDLE=1'b0, COMPUTE=1'b1} state;

  logic  [4:0] count;
  logic [15:0] R0, R1;

  assign done = ...; // no FF will be generated
  assign fibo_out = ...; // no FF will be generated

  // just update count, R0, R1, state in always_ff
  always_ff @(posedge clk, negedge reset_n)
  begin
    if (!reset_n)
      ...
    else case (state)
    IDLE:
      ...
    COMPUTE:
      ...
    endcase
  end
endmodule
```