

CSE 2600

Intro. To Digital Logic & Computer Design

Bill Siever
&
Michael Hall

This week

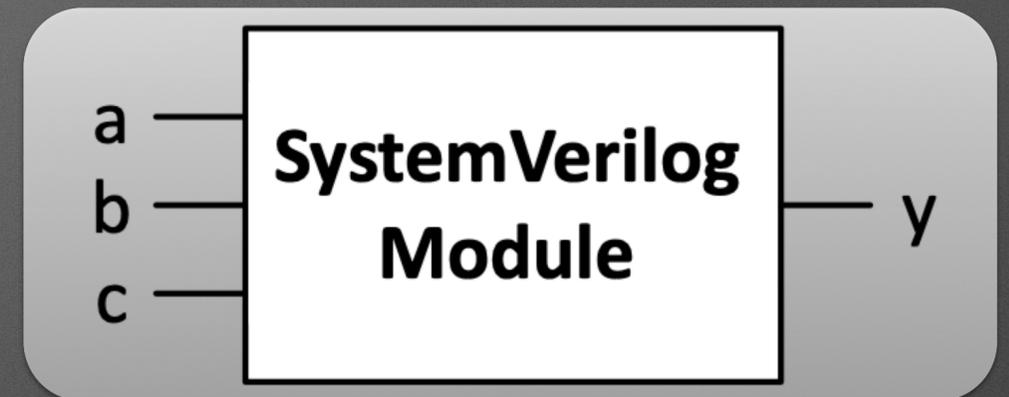
- Hw 4A requires an in-person demo (during office hours) for full credit.
- Demos required on most/all remaining assignments for full credit. Can start on the due date and must be completed with 10 days after.
- Hw 4B posted / due March 17th (Tues after break).
- Assignments must be pushed to GitHub and then submitted to Gradescope. As with JLS assignments, there's usually a Gradescope autograder that should be checked to confirm submission and much of the credit.

Chapter 4

Review: HDLs *Describe* Hardware

- Uses
 - “Synthesis” : Transformation to real hardware
 - Like compilers used for programming languages
 - Simulation: Confirm modules work together
- Use modules for hierarchical design — important part of managing complexity
- Description Styles
 - Structure (connect 2 input AND to ...)
 - Behavior (if x then y)

(System) Verilog Module: Review



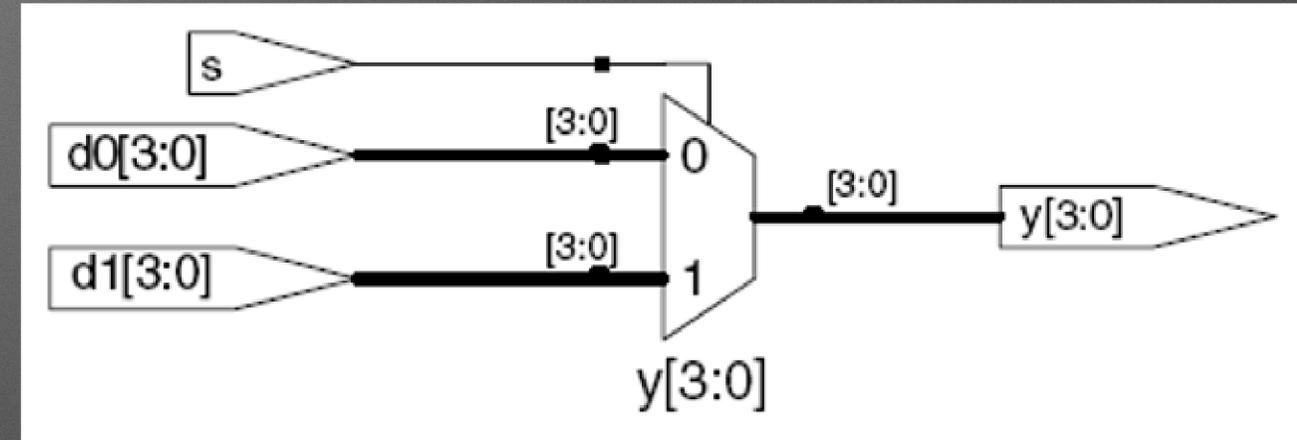
```
module example(input logic a, b, c,  
               output logic y);  
    // module body goes here  
endmodule
```

Input & Output
are like the Pins
On chips or in
JLS



(System) Verilog

- Conditionals via Ternary operator (? :)



```
module mux2 (input logic [3:0] d0, d1,  
            input logic s,  
            output logic [3:0] y);  
    assign y = s ? d1 : d0;  
endmodule
```

Behavioral

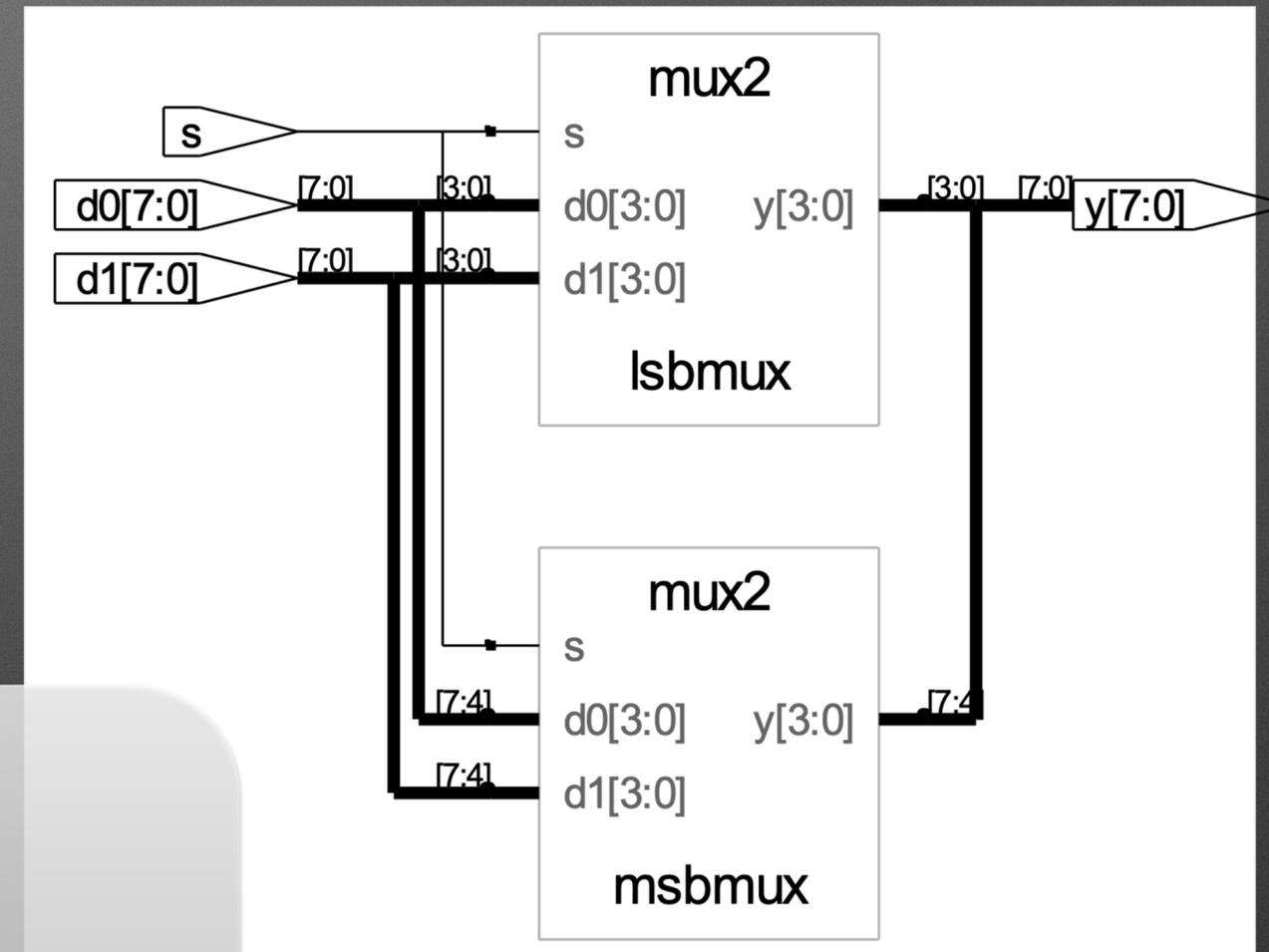
8-bit mux2: Hierarchical

```
module mux2_8 (input logic [7:0] d0, d1,  
              input logic s,  
              output logic [7:0] y);
```

```
    mux2 lsbmux (d0[3:0], d1[3:0], s, y[3:0]);
```

```
    mux2 msbmux (d0[7:4], d1[7:4], s, y[7:4]);
```

```
endmodule
```



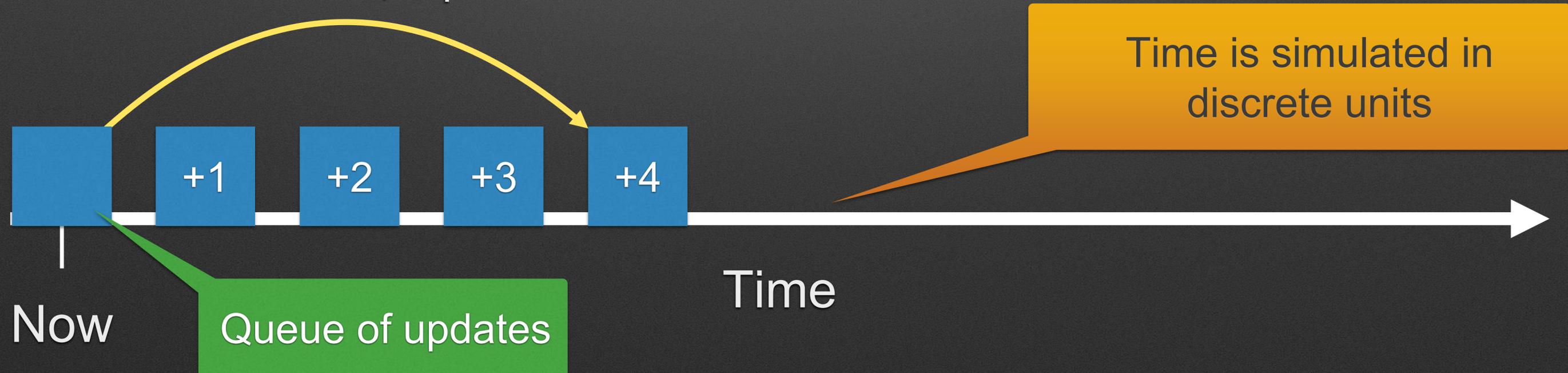
Sequential Logic

always: **Based on *Events***

- Concept of “event” is related to simulation and “event driven programming”
- JLS uses events: An OR gate “reacts” to events and schedules an update
See [here](#)

Discrete Time Event Simulator

- Computes all activities / updates for “now”
 - They cause new activities that need to be handled in the future (at: now + prop delay). Those are put in a queue at for that time.
Ex: Update an or-gate’s output at now+4
- Move on to “now +1”, repeat



Discrete Time Event Simulator

- Updating values in current turn: Incrementally or all at once at end of turn
 - Ex: Assume x is 1 and y is 0
 - Incremental:
 - $x = 0$
 - $y = x$
 - x's final value is 0
 - y's final value is 0 too
- All at once / end of turn
 - $x \leq 0$
 - $y \leq x$
- x's final value is 0
- y's final value is 1

SystemVerilog Standard

- Why all the simulation details?
- Quick intro to SystemVerilog Standard
 - Section 9 / 9.2

always **Statement**

- Form:
always @(sensitivity list)
statement;
- When event in sensitivity list occurs, statement is executed
- Ex: always @(posedge clock)
statement;
- Verilog: Don't use this in here

always **Statement**

- Form:
~~always @(sensitivity list)~~
statement;
- When event in sensitivity list occurs, statement is executed
- Verilog: Don't use this in here

always in 2600

- Form 1: Comb logic
always_comb
statement;

Use blocking assignment (=)

- Statement(s) are (complex) combinational logic. Like if/else or case.
Updates when any (relevant/used) input changes

- Form 2: Registered (synchronous, synthesize able, sequential) logic
always_ff @(sensitivity list)
statement;

- Often @(posedge clock) used

Use non-blocking assignment (<=)

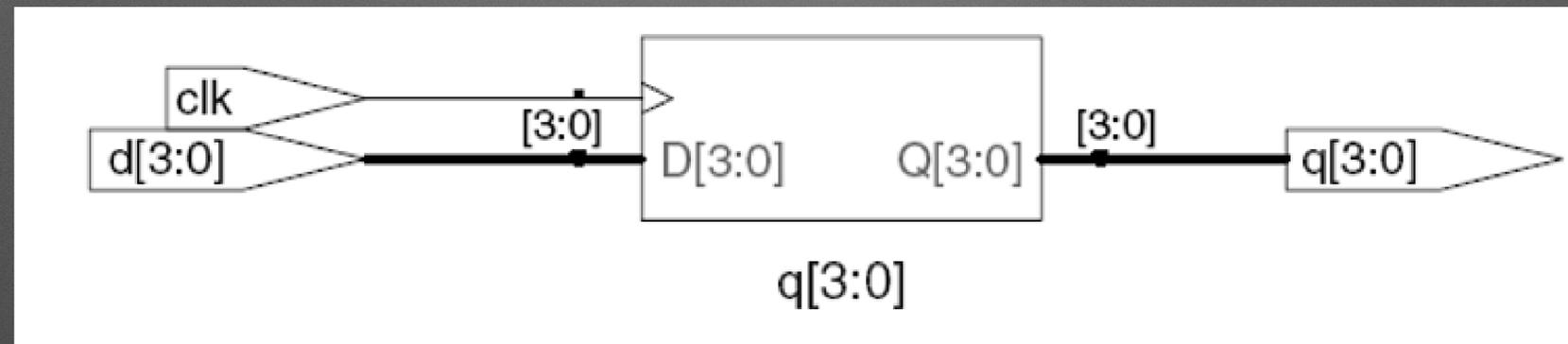
Assignments

- Form 1: Continuous Assignment
assign var = expression;
 - Continuously assigned! Largely a wired connection
- Forms 2 & 3 *in Procedures* (in some form of always*) :
 - Blocking (=): Will be “instant” in terms of simulation
 - always_comb
 - Non-Blocking (<=): Will occur at end of turn all at once
 - always_ff

Rules for Assignments

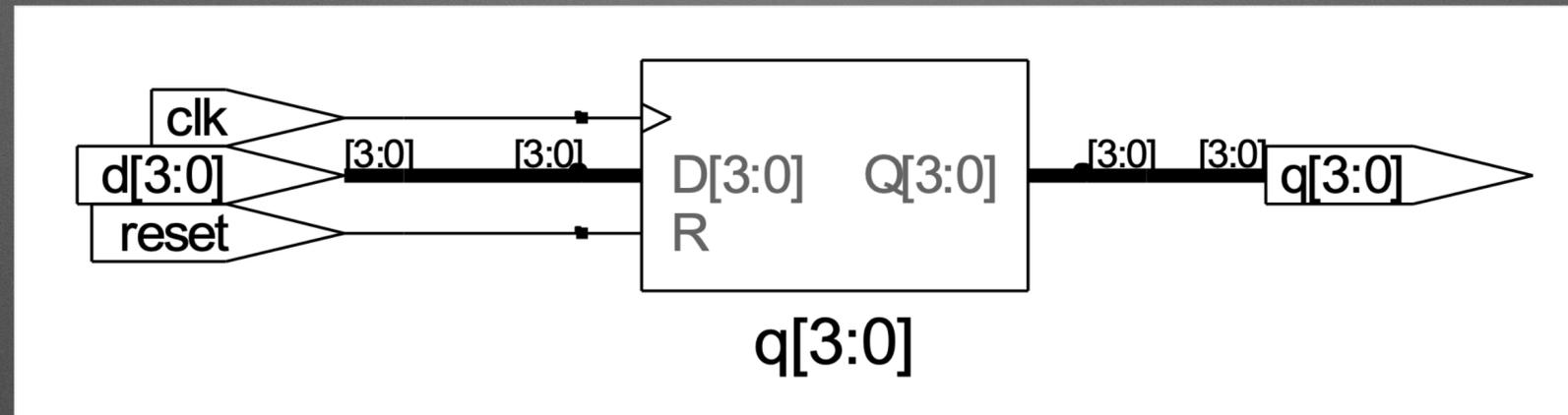
- Synchronous sequential logic
use `always_ff @(posedge clk)` and nonblocking assignments (`<=`)
`always_ff @(posedge clk)`
`q <= d; // nonblocking`
- *Simple* combinational logic
use continuous assignments (`assign`)
`assign y = a & b;`
- Complex Combinational Logic
use `always_comb` and blocking assignments (`=`)
- Assign signals in only one `always` or `assign` statement!

Verilog: D Flip-Flop



```
module flop(input logic clk,  
            input logic [3:0] d,  
            output logic [3:0] q);  
    always_ff @(posedge clk)  
        q <= d; // pronounced "q gets d"  
endmodule
```

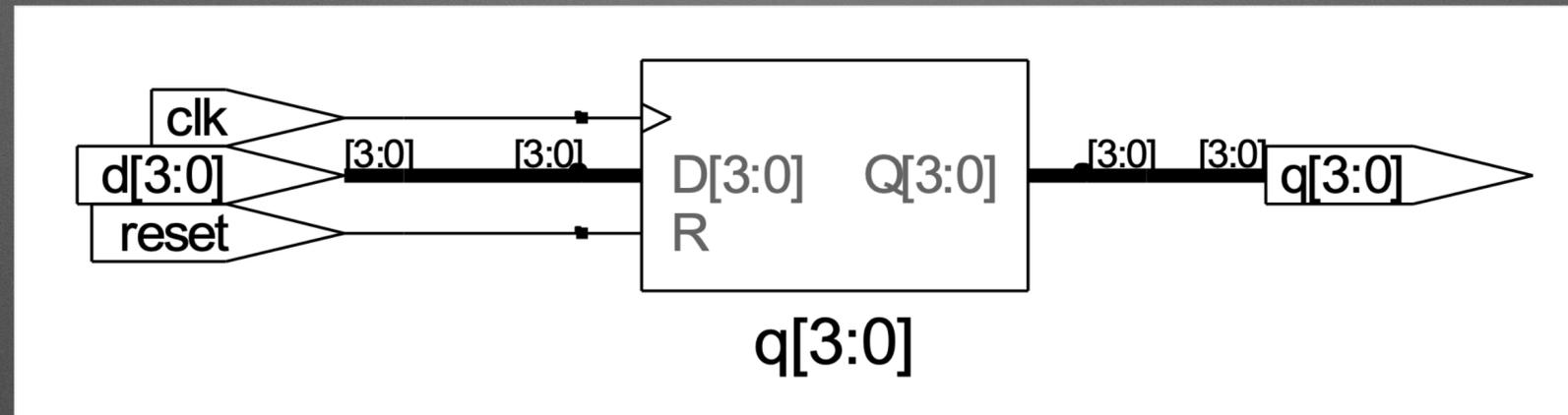
Resettable D-Flip-Flop 1



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

    always_ff @(posedge clk)
        if (reset) q <= 4'b0;
        else      q <= d;
endmodule
```

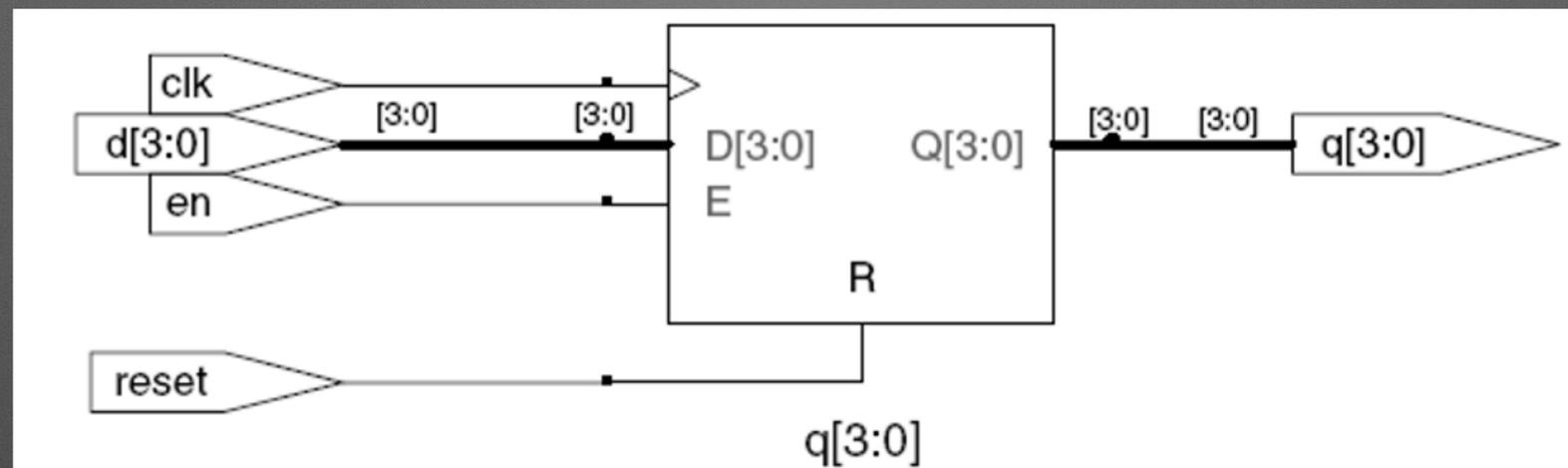
Resettable D-Flip-Flop 2



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

    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else      q <= d;
endmodule
```

Resettable D-Flip-Flop 3



```
module flopr(input logic clk,  
            input logic reset,  
            input logic en,  
            input logic [3:0] d,  
            output logic [3:0] q);  
  
    always_ff @(posedge clk, posedge reset)  
        if (reset) q <= 4'b0;  
        else if (en) q <= d;  
  
endmodule
```

always and Combinational Logic

```
always_comb  
begin  
    y = a & b  
    ...  
end
```

Block of assignments

Could have been done with individual assigns

Notice = ("blocking assignment"), not <= ("non-blocking assignment")

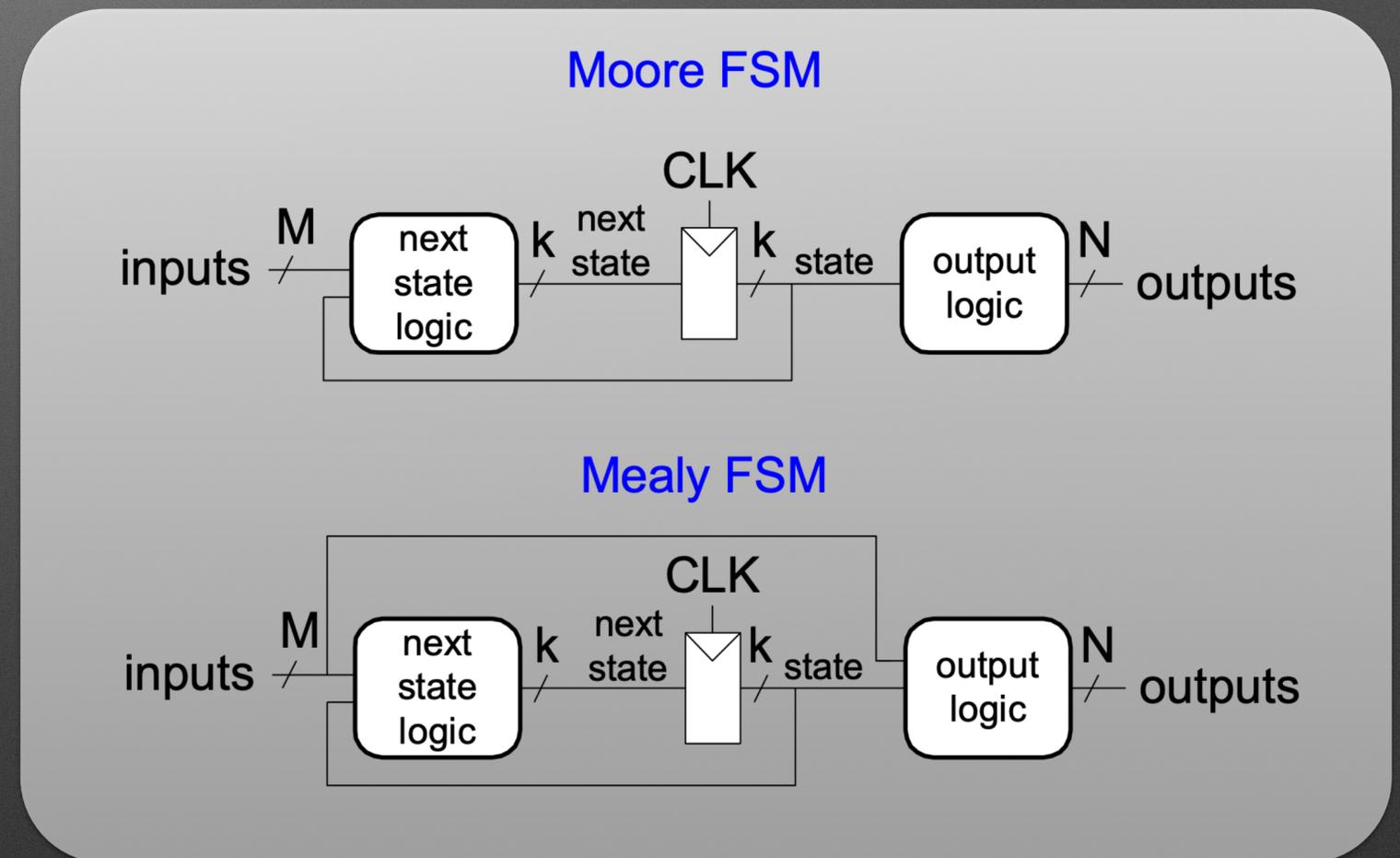
always_comb has nice features

- case : Selection between several options
Great for state machines!
- Must describe all possible combinations to be comb logic. Use default

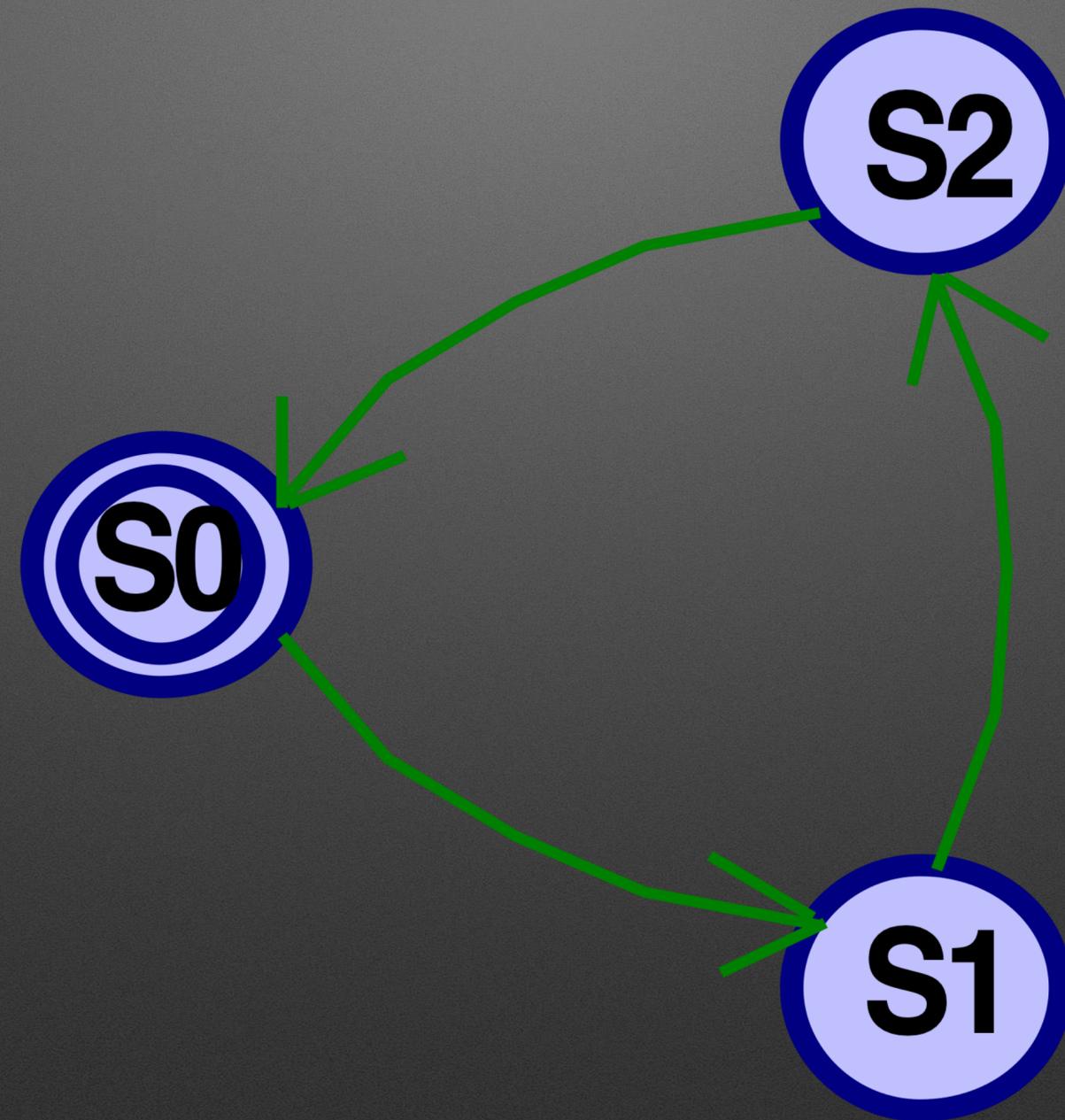
```
case (state)
  soap:          hot = 1;
  highPressureWarm: hot = 1;
  ...
  default: hot = 0;
endcase
```

Verilog FSMs

- Three parts
 - Next state logic (arrows / next state table)
 - State register (active bubble)
 - Output logic (output equations)

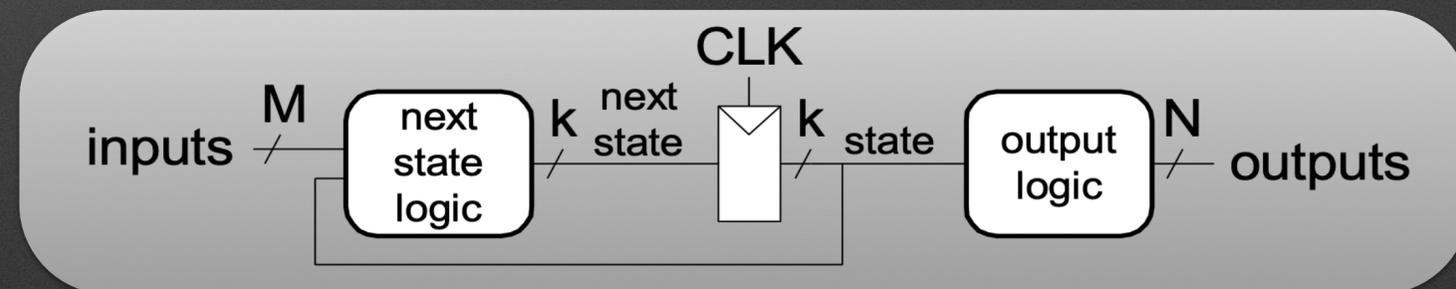
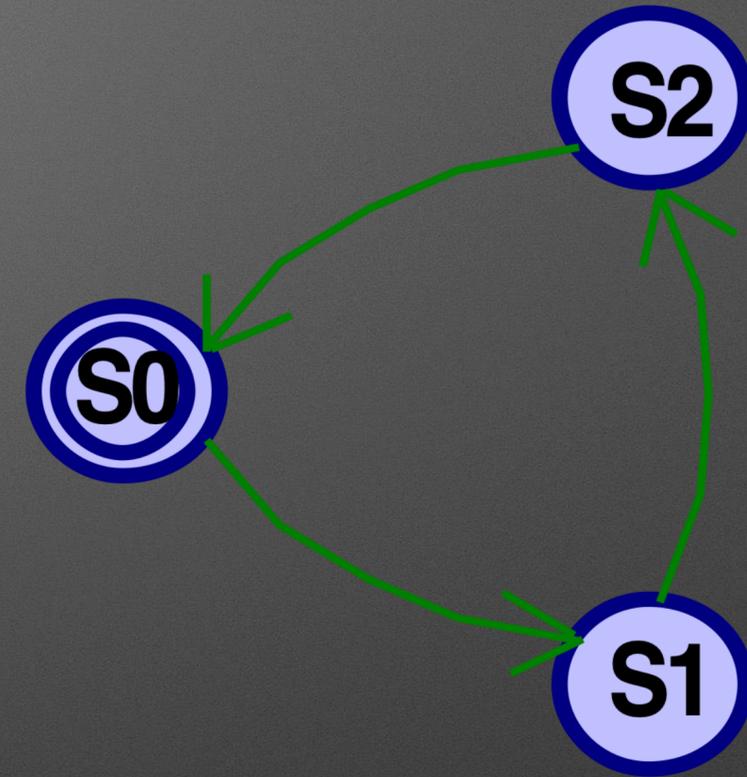


Divide by 3 Counter



Verilog

```
module divideby3FSM(input logic clk,  
                  input logic reset,  
                  output logic q);  
  
    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 q = (state == S0);  
endmodule
```



Parameterized Modules: Declaration

- Way to specify additional details for an *instance* of a generic part
 - Commonly the “width” of the part

```
module mux2
  #(parameter width = 8) // name and default value
  (input logic [width-1:0] d0, d1,
   input logic          s,
   output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Parameterized Modules: Use

- Default or specify parameter for instance:

```
mux2 myMux(d0, d1, s, out);
```

```
mux2 #(12) lowmux(d0, d1, s, out);
```

Ports: Positional vs. Named

- Default or specify parameter for instance:

```
logic a, b, sel, y
mux2 myMux(a, b, sel, y);
```

vs.

```
mux2 myMux(.d0(a), .d1(b),
           .s(sel), .out(y));
```

```
module mux2
    #(parameter width = 8)
    (input logic [width-1:0] d0, d1,
     input logic          s,
     output logic [width-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

Test Bench: Overview & Concept

(Simple w/ Asserts)

Hw4A: simple_comb_tb

FPGA: Field Programmable Gate Array

FPGA

- Field Programmable
- Gate Array
- [Lattice iCE40 UP5k](#): Architecture Overview
 - RAMs, (Dual and Single Port)
 - Look Up Tables (LUTs): 4 inputs
 - D Flip Flops
 - Lots: ~5,000

**Playground: Combinational logic,
hardware, synthesis, and parameters**

**Disclaimer: Don't (typically) do this!
(Think before working...Be more
methodical!)**

Examples

- Leds assignment(s)
- Using keys and assign / logic
- Spinner module
 - Adjusting parameters
 - Multiple spinners

Questions

- Why case statement implies sequential logic if not all possible input combinations are defined? Not so clear when to use wire, reg, tri or what are they.
- How does synthesis decide the exact hardware structure from behavioral code?
- First, when should I use blocking vs. nonblocking assignments, and how do they change the final circuit? Second, how do I pick between Mealy and Moore FSMs for different jobs? Finally, what's the easiest way to write a testbench in SystemVerilog to check my design without having to manually look at every output?
- What is the difference between always and always_comb? Is the choice of using assign, always_ff, always_comb just based on the fact of whether you want to use non-blocking or blocking assignments? Are there any other factors to consider?
- How do synthesis tools choose the optimal state encoding when we use enumeration types instead of binary values?