CSE 260M / ESE 260
Intro. To Digital Logic & Computer Design

Bill Siever
&
Michael Hall

This week

- Thursday: Studio — Need 1 Hw kit *and cable* for each group!

- Exam 1: Will be returned later this week

  - Midterm Grades on Canvas, etc. soon after

- Hw 4A probably returned too

- Hw 4B posted tonight:

  - Due Monday, March 18th @ 11:59pm (reading for ch 5 also due then!)

# Chapter 4

Review: HDLs *Describe* Hardware

- Uses

  - Simulation: Confirm modules work together

  - "Synthesis" : Transformation to real hardware

    - Like compilers used for programming languages

- 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
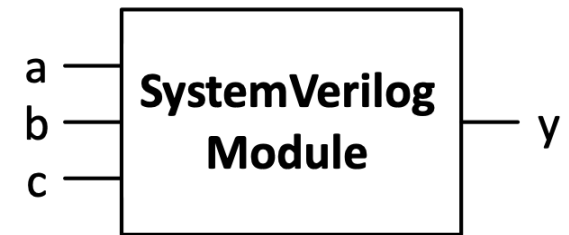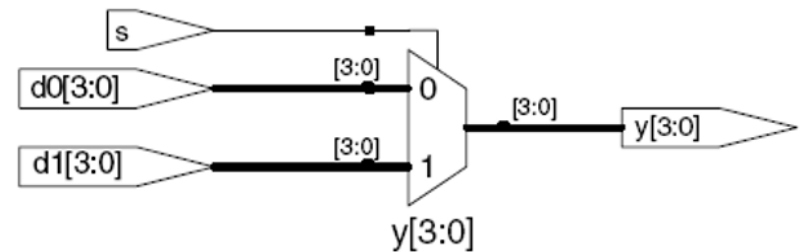


```
module example(input  logic a, b, c,
               output logic y);
   // module body goes here
endmodule
```

Input & Output are like the Pins On chips or in JLS
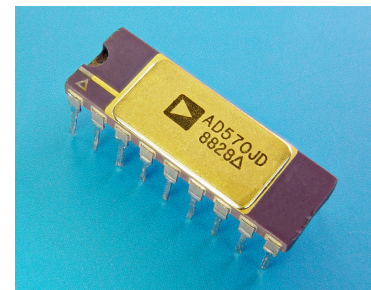
(System) Verilog

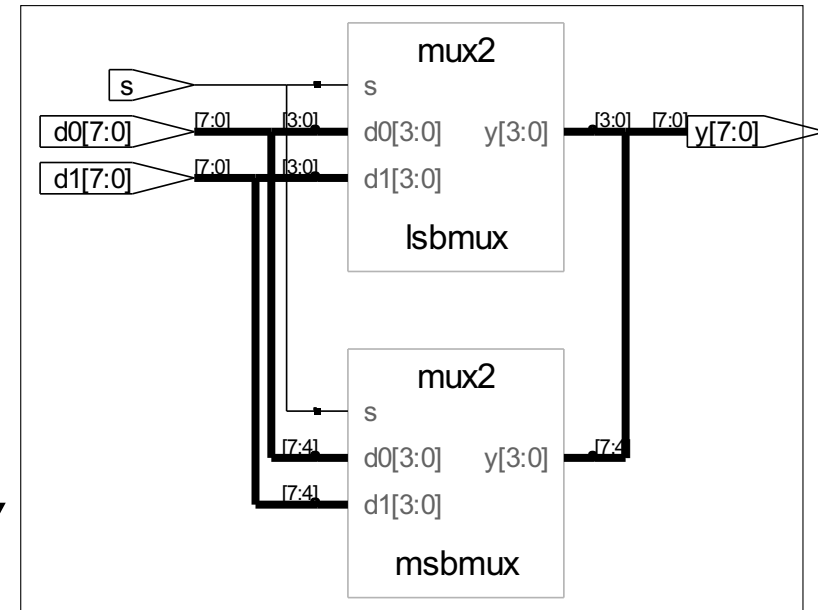- **Conditionals via Ternary operator (? :)**


```
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

Time is simulated in
discrete units

+1   +2   +3   +4

Time

Now   Queue of updates
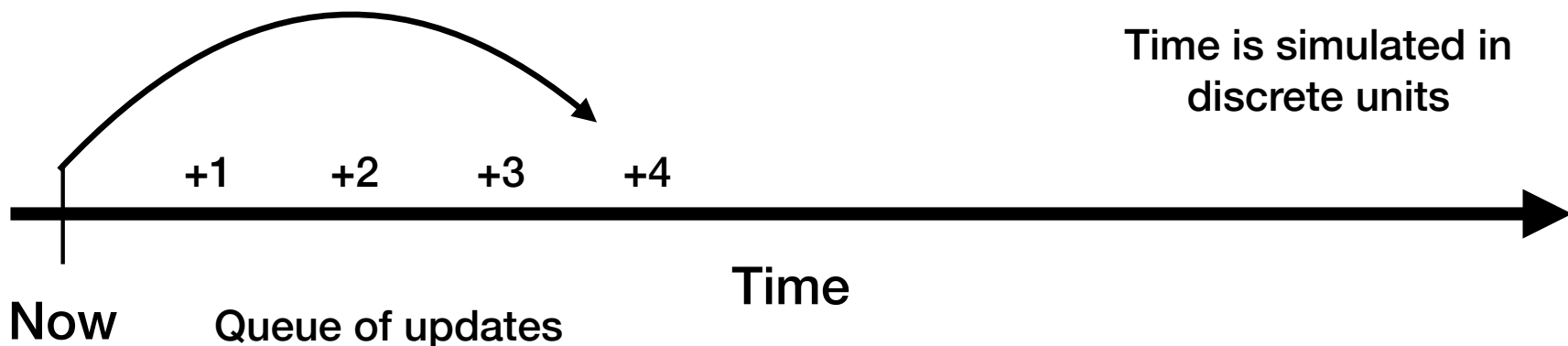
Discrete Time Event Simulator

- 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 <= 0
    y <= 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 260**

- 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

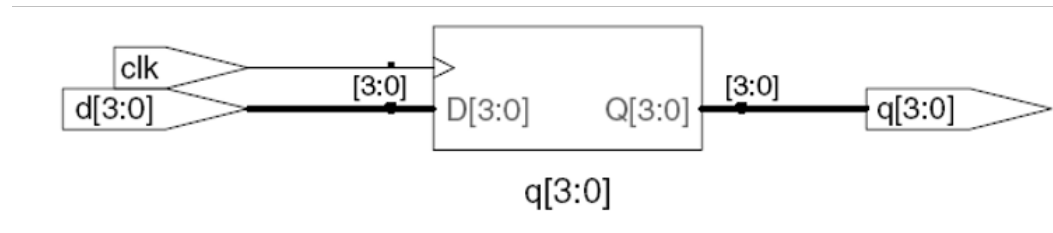   - `always_ff`

Rules for Assignments

- **Synchronous sequential logic**
  use `always_ff @(posedge clk)` **and nonblocking assignments (**`<=`**)**
  ```
  always_ff @(posedge clk)
      q <= d; // nonblocking
  ```

- *Simple* **combinational logic**
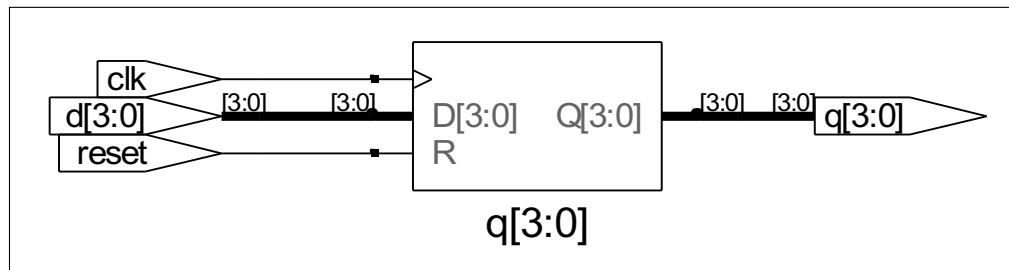 **use continuous assignments (**`assign`**)**
  ```
  assign y = a & b;
  ```

- 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
```

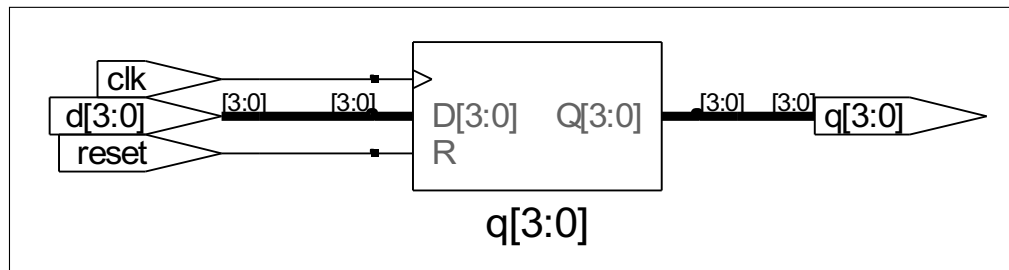# Resettable D-Flip-Flop 1


```
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
```

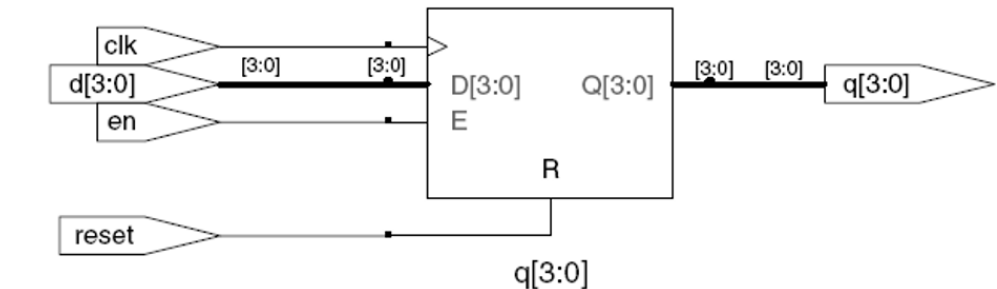# Resettable D-Flip-Flop 2


```
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**

Block of assignments

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

Could have been done with individual `assign`s

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

`always_comb` **has nice features**

- `case` : Selection between several options
  Great for state machines!

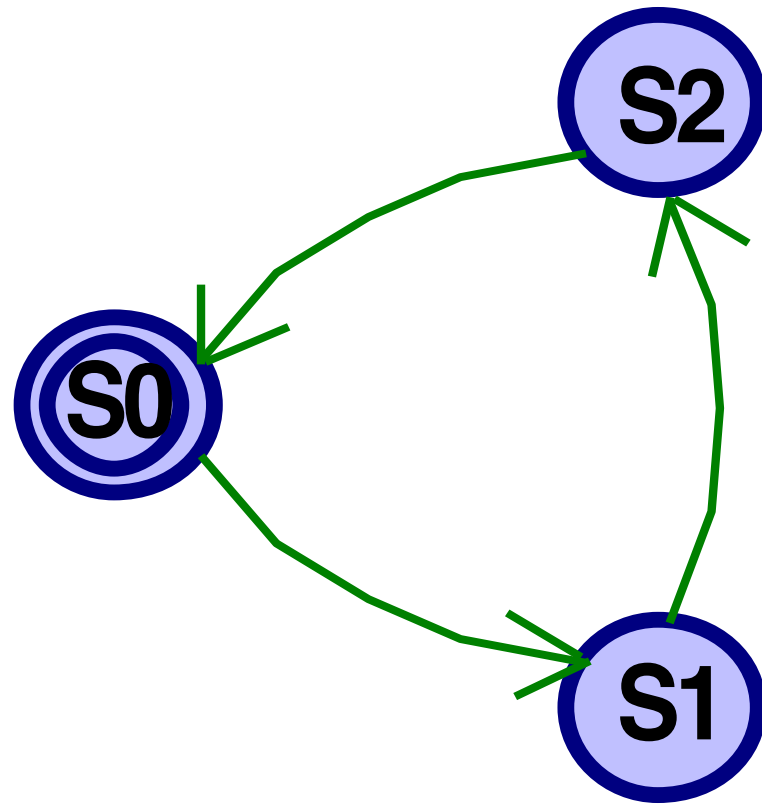 - Must describe all possible combinations to be comb logic. Use `default`

```
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)**

Moore FSM

Mealy FSM
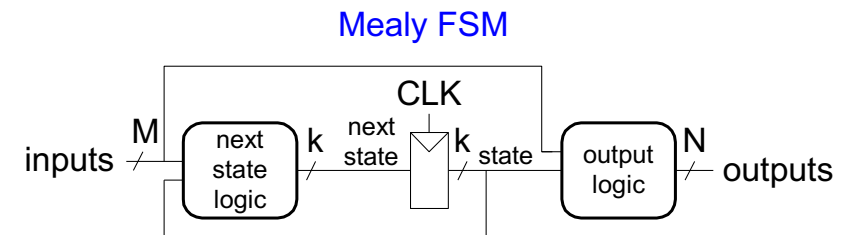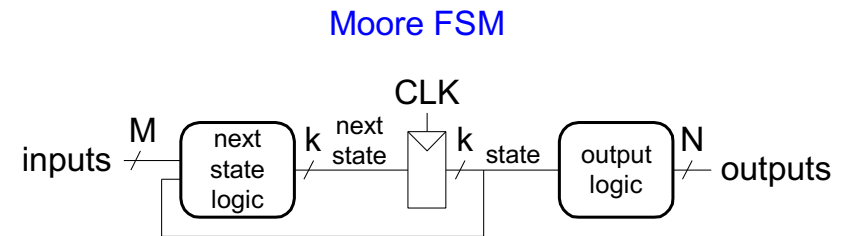
Divide by 3 Counter

Verilog

```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
```
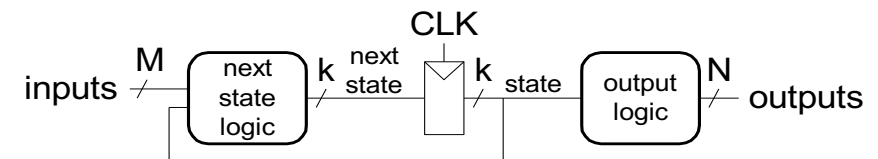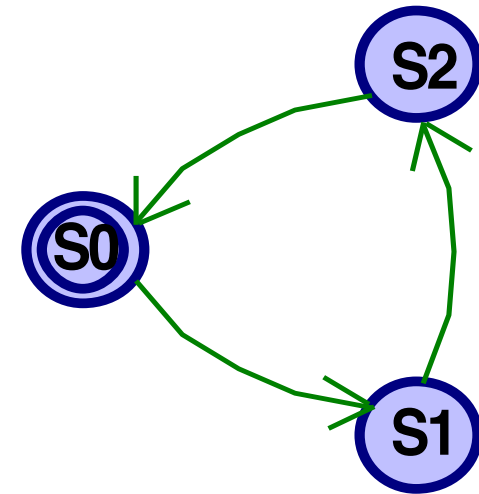
Parameterized Modules: Declaration

- 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, see, 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
    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
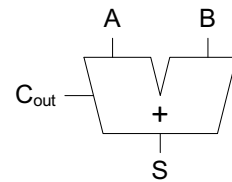
FPGA

- 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
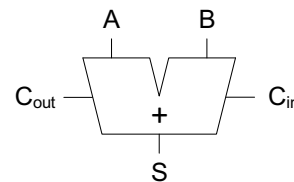
Questions

Adders

Half Adder

A   B

C_{out} —⎛ ⎞—
        ⎝ + ⎠

S

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$

Full Adder

A   B

C_{out} —⎛ ⎞— C_{in}
        ⎝ + ⎠

S

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Ripple Carry Adder: Propagation Delay