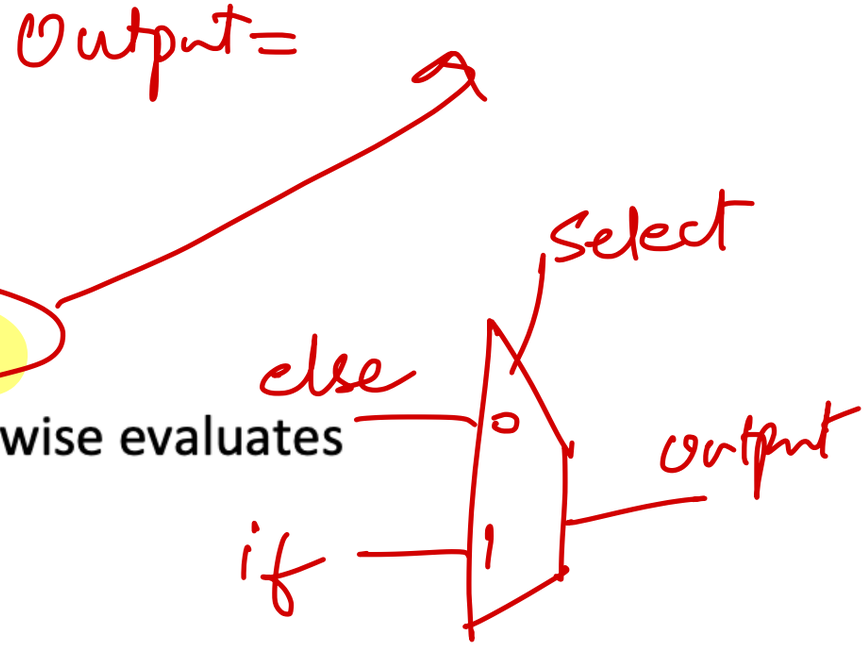
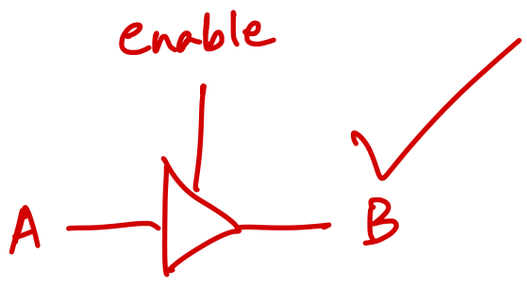
# Ternary Operator

- Conditional assignment
  - `select ? <if_expr> : <else_expr>`
    - If `select` is true, then evaluates to `<if_expr>`, otherwise evaluates to `<else_expr>`
  - What does this look like in hardware?




- **Example:** tristate buffer
  - `enable ? in : 'bZ`
    - When enabled, pass the input to the output, otherwise be high impedance

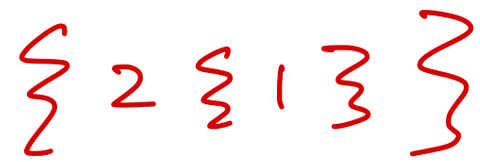



| enable | A | B |
|---|---|---|
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

B = enable ? A : 'bz

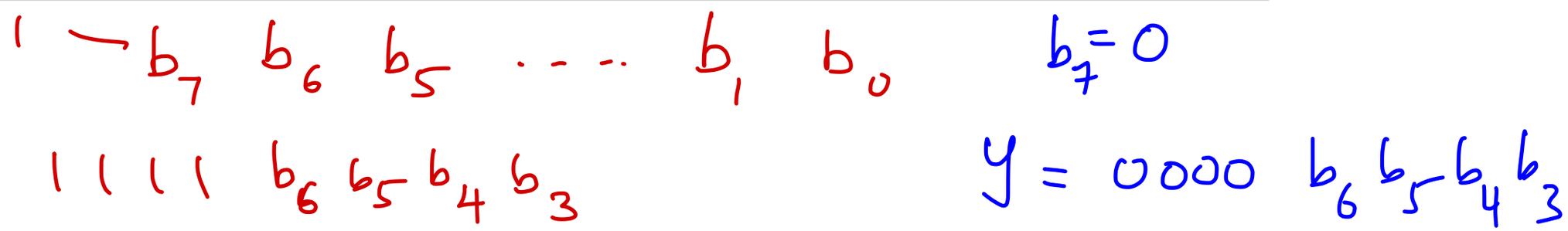
# Bit Manipulation

- Concatenation: `{sig, …, sig}`
  - Ordering matters; result will have combined widths of all signals
- Replication operator: `{n{m}}`
  - repeats value `m`, `n` times



- Exercise: arithmetic right shift preserves the sign bit

```
logic [7:0] x = <some 8-bit constant>;
// replicate the behavior of y = x >>> 3
assign y = { {3{x[7]}}, x[7:3] };
```



# "Looping"

- Code is compiled to hardware, so no execution
  - "Loops" must be *statically unrolled* into multiple statements
  - Loops are just for convenience in code writing
- `repeat (#) <statement(s)>`
  - Makes # copies of statement(s)
- `for (i=0; i<#; i++) <statement(s)>`
  - Makes # copies of statement(s) that vary based on `i`
- `generate` (see reference docs)
  - More "powerful" for-loop typically used for:
    1) Module instantiation
    2) Changing the structure of parameterized modules
    3) Functional and formal verification using assertions

# Modules

- "Black boxes" that we define and instantiate that form the basic building blocks of our design hierarchy
  - **Ports** form the connections between a module and its environment
    - Ports have directionality (`input`, `output`, `inout`), which can be declared within the module or within the port list

```
module tristate(out, in, enable);
   input logic in, enable;
   output tri  out;

   assign out = enable ? in : 'Z;
endmodule
```

C-style.

```
module tristate(output tri out,
                input logic in,
                input logic enable);
   assign out = enable ? in : 'Z;
endmodule
```

# Module Instantiation

- Name an instance and define its port connections
  - `<type> <name> (<port connections>);`
- Assume we have:

```
logic in, enable; tri out;
```

1) Positional connections:

```
// must follow defined port ordering
// signal names can be anything
tristate my_tri(out, in, enable);
```

2) Named/explicit connections:

```
// any ordering & names allowed
tristate my_tri(.out(out), .in(in), .enable(enable));
```

3) *.name* implicit connection:

```
// signal and port names must match exactly
tristate my_tri(.out, .in, .enable);
```

# Parameters

- A **parameter** is a named constant
  - Typically used for widths and timing

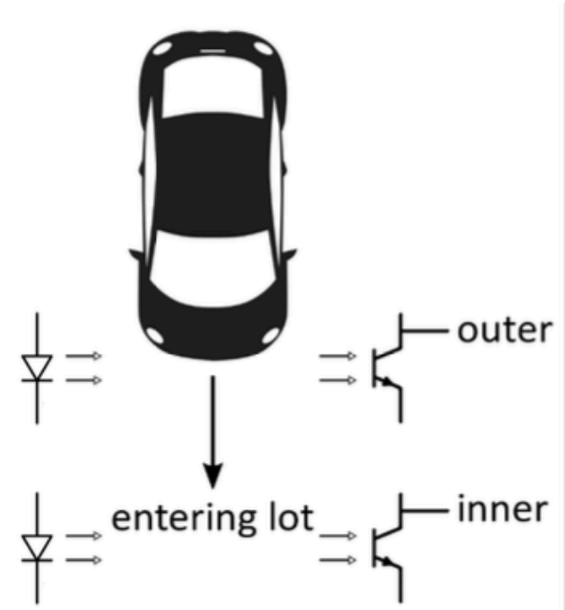
```
parameter N = 8;          // bus width
parameter period = 100;   // timing constant
```

- A parameterized module:
  - `module <name> #(<parameter list>) (<port list>);`
  - Parameters should be given default values
    - *e.g.*, `#(parameter N = 8)`

- Exercises:
  - Define a parameterized `tristate` (tristate buffer)
  - Define a parameterized `multibitAND`

# Lab 1 Notes

- Read the spec carefully!
  - For scenarios that are not described, it's up for you to define; describe and defend your decisions in your report
  - Also read `371_Assignments.pdf`
- Plan and design *before* you start coding!
- Test your code in small pieces *as you go*
  - Lab report due before your demo
  - Short sessions (3 min) on LabsLand



# EE/CSE 371
# Design of Digital Circuits and Systems

## Lecture 2: Finite State Machine Review

# Relevant Course Information

1. HW 1 and Lab 1 due this week

   Homework can be completed in groups of up to 4
   Labs can be completed in groups of up to 2

2. Make sure LabsLand is setup and synthesized beforehand

# Parameters (Review)

- A **parameter** is a named constant
  - Typically used for widths and timing

```
parameter N = 8;          // bus width
parameter period = 100;   // timing constant
```

- A parameterized module:
  - `module <name> #(<parameter list>) (<port list>);`
  - Parameters should be given default values
    - *e.g.*, `#(parameter N = 8)`

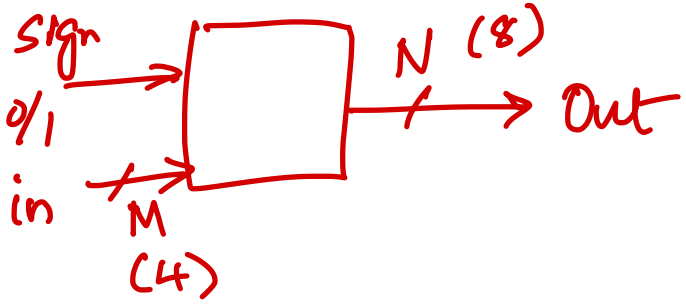
Concatenation: { sig, sig ... }

Replication: { n { m } }

ternary: out = enable ? in : 6'z ;

# Review Question

- There are two forms of bit extensions: zero-extension (add `0`s) and sign-extension (copy MSB)
- Write out SystemVerilog pseudocode for a parameterized *extender* module  M = 4, N = 8
  - Inputs `sign` (1 bit), `in` ($M$ bits); output `out` ($N$ bits > $M$)
  - `out` should either be the sign-extended version of `in` (`sign` = `1`) or the zero-extended version of `in` (`sign` = `0`)



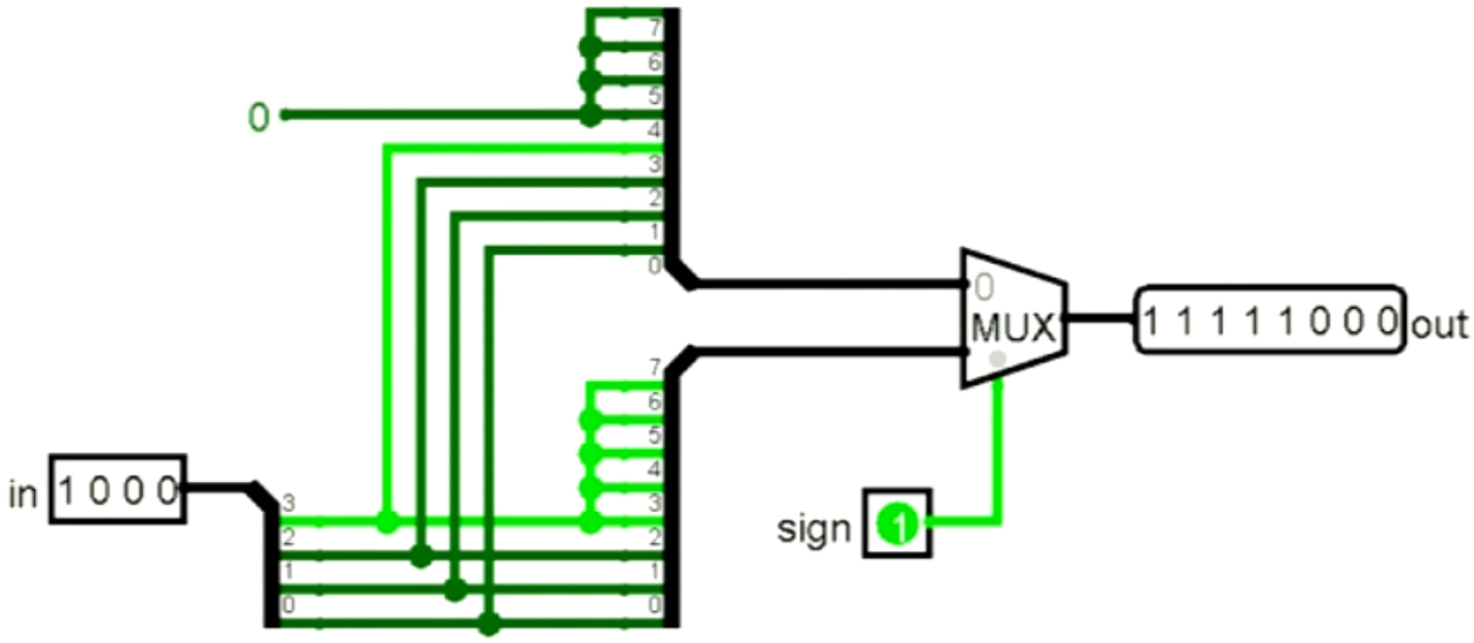
```
module extender #(parameter M = 4, N = 8)
                 (output [N-1:0] out,
                  input [M-1:0] in,
                  input sign);

   assign out = sign ? {{(N-M){in[M-1]}},in} : {{(N-M){1'b0}},in};

endmodule
```

MSB of in

- Hardware if $M = 4$ and $N = 8$:



# Lecture Outline

- **SystemVerilog Review & Tips (Cont.)**
- FSMs
- Testbenches

# Structural vs. Behavioral Revisited

- Not a strict definition of these terms, so exact classification is not that important
- Structural:
  - Instantiating modules (library and user-defined) and defining port connections
  - `assign`: continuous assignment
    - Used with nets

# Verilog Procedural Blocks

- A *procedural block* is made up of behavioral code in the form of procedural statements that are executed sequentially
  - The block itself is awakened/triggered in a non-sequential manner
- `initial`: block triggered once at time zero
  - Non-synthesizable (*i.e.*, for simulation/testbenches only)
- `always`: block triggered by a *sensitivity list*
  - Any object that is assigned a value in an `always` statement must be declared as a variable (*e.g.*, `logic` or `reg`).

# SystemVerilog Procedural Blocks

- SystemVerilog introduced variants on always that are generally more robust and more specialized
- `always_comb`: intended for combinational logic
  - Sensitivity list is automatically built
- `always_latch`: intended for latch-based logic
  - Sensitivity list is automatically built
- `always_ff`: intended for sequential logic (*i.e.*, synchronous/clocked)
  - Sensitivity list must be specified

# Latch vs. Flip-Flop

- Both are bistable multivibrators (2 stable states) that can store information
- A latch is *asynchronous*; a flip-flop is *edge-triggered*

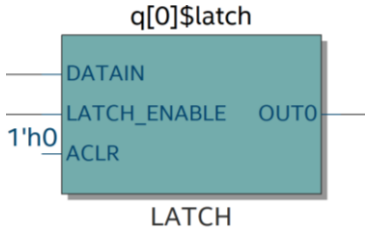
semi-synchronous

q can change when CLK = 1

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

   always_latch
      if (clk) q <= d;
      else q <= q;
endmodule
```

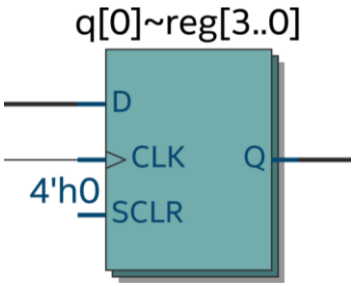
@(clk, d)




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

   always_ff @(posedge clk)
      q <= d;

endmodule
```



# Inferred Latches

- *Warning:* easy to write code with inadvertent latches
  - Check your synthesis output for "Inferred latch"
  - Usually from *incomplete assignments* – unspecified branch infers latch behavior
- **Question:** which of the following will synthesize and, if so, what will the hardware look like?

```
always_latch
   if (clk) q <= d;
```


```
always_comb
   if (clk) q <= d;
```

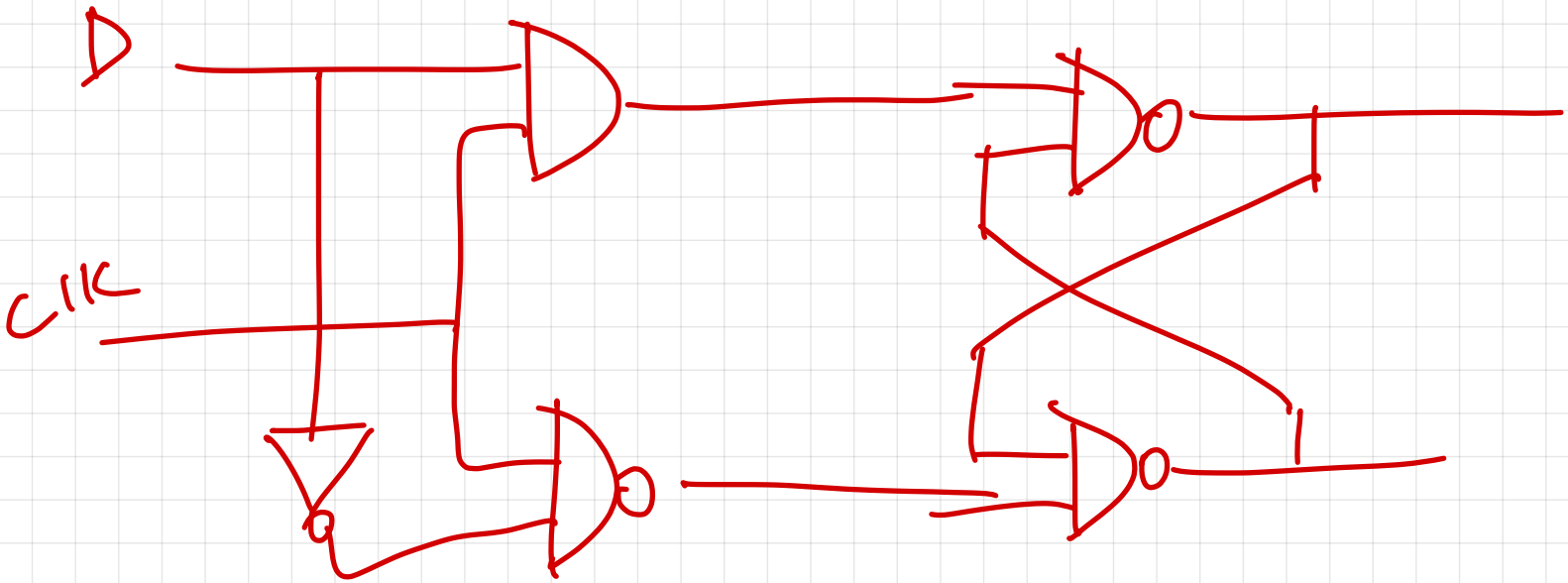
latch.

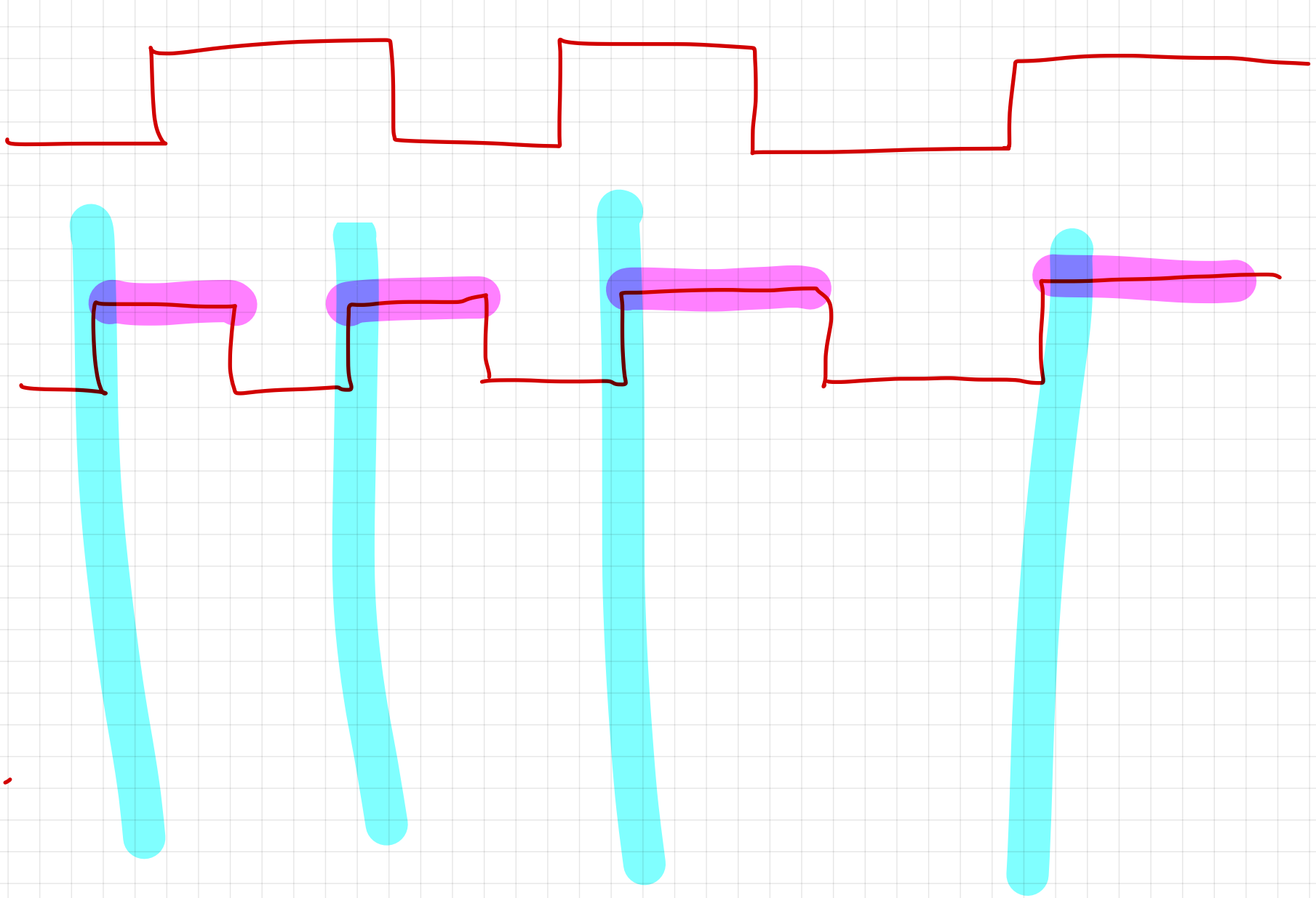
```
always_latch
   if (clk) q <= d;
   else     q <= ~d;
```

else q <= q;


```
always_comb
   if (clk) q <= d;
   else     q <= ~d;
```

- Demo: Tools → "Netlist Viewers" → "RTL Viewer"


D
CLK


d
CLK
q_latch
CLK=1
q_ff.
pos.

# `case` Statement

- Create combinational logic and is easier to read than lots of `if`/`else` statements
  - Must always be inside an `always` block
  - Each case has an implied C-style break

```
module seven_seg(bcd, segs);

   input  logic [3:0] bcd;
   output logic [6:0] segs;

   always_comb
      case (bcd)
         //            abc_defg
         0: segs = 7'b011_1111;
         1: segs = 7'b000_0110;
         2: segs = 7'b101_1011;
         3: segs = 7'b100_1111;
         4: segs = 7'b110_0110;
         5: segs = 7'b110_1101;
         6: segs = 7'b111_1101;
         7: segs = 7'b000_0111;
         8: segs = 7'b111_1111;
         9: segs = 7'b110_1111;

      endcase

endmodule
```

# `case` Statement

- Create combinational logic and is easier to read than lots of `if`/`else` statements
  - Must always be inside an `always` block
  - Each case has an implied C-style break
  - Remember to use `default` to avoid incomplete assignments!

```
module seven_seg(bcd, segs);

   input  logic [3:0] bcd;
   output logic [6:0] segs;

   always_comb
      case (bcd)
         //              abc_defg
         0: segs = 7'b011_1111;
         1: segs = 7'b000_0110;
         2: segs = 7'b101_1011;
         3: segs = 7'b100_1111;
         4: segs = 7'b110_0110;
         5: segs = 7'b110_1101;
         6: segs = 7'b111_1101;
         7: segs = 7'b000_0111;
         8: segs = 7'b111_1111;
         9: segs = 7'b110_1111;
      -> default:  segs = 7'bX;
      endcase

endmodule
```

# Other SystemVerilog Resources

- SystemVerilog Language Reference Manual
  - On ~~Canvas~~ Piazza, Files → "Reference Material"
  - 586 pages…
- SystemVerilog articles
  - https://www.systemverilog.io/
  - http://www.verilogpro.com/
  - https://www.chipverify.com/systemverilog/systemverilog-tutorial
- One style guide for SystemVerilog
  - https://www.systemverilog.io/styleguide
  - We won't enforce, but good guidelines

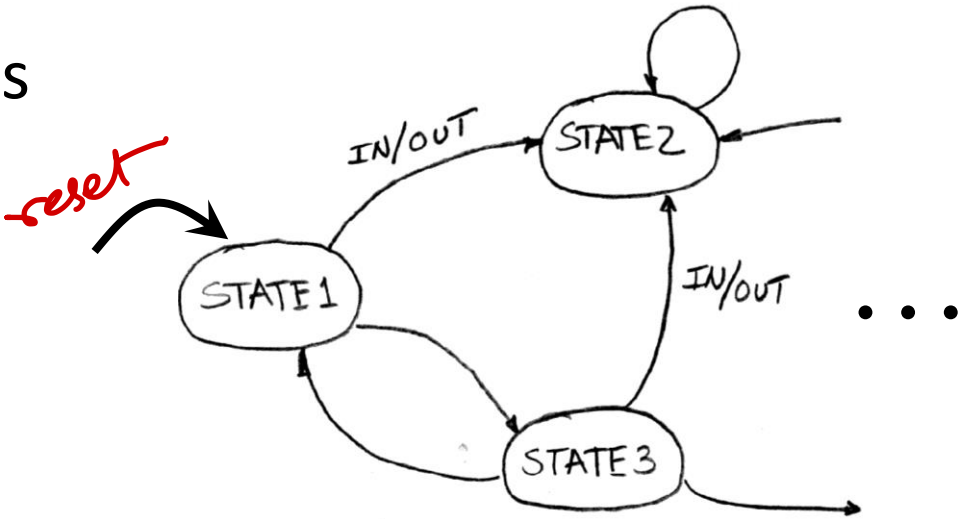
# Lecture Outline

- SystemVerilog Review & Tips (Cont.)
- **Finite State Machine Design**
- Testbenches

# Finite State Machines (FSMs)

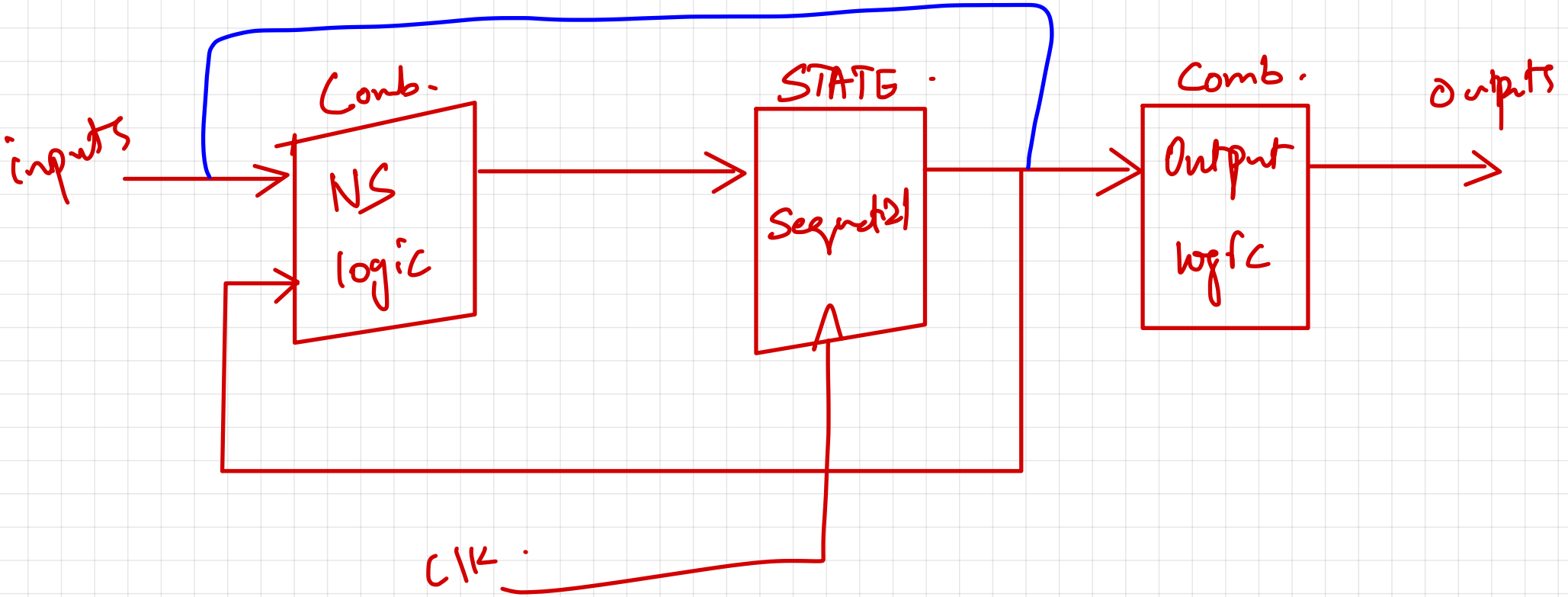
- A convenient way to conceptualize computation over time using a *state transition diagram*
  - Consists of a *set of states*, an *initial state*, and a *transition function*

- FSM implementations come in 3 blocks:
  - State register (SL)
  - Next state logic (CL)
  - Output logic (CL)



# Moore FSM.

# MEALY FSM



# FSM Implementation Notes

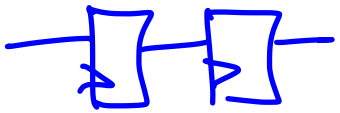
enum {S0,S1,S2} ps,ns;

- States must be assigned a binary encoding
  - More readable by using parameters or an enum
  - Encoding choices can affect logic simplification
- Reset signal can be synchronous (responds to `clk`) or asynchronous (responds to `reset`)
  - Determined by whether or not `reset` is in sensitivity list

@(posedge clk)

@(posedge clk, posedge reset)

- State logic (next state logic + state update) can be written as 1 combined block or 2 separate blocks
- If input is asynchronous, may want to add a two-flip-flop *synchronizer* to deal with metastability



# FSM SystemVerilog Design Pattern

- Which, if any, construct(s) would you expect to use for each of the following basic sections of a module that implements an FSM?

enum {A,B,C} PS, NS;

- *// define states and state variables*

  initial assign always_comb always_ff None

- *// next state logic*

  initial assign always_comb always_ff None

- *// output logic*

  initial assign always_comb always_ff None

- *// state update logic*

  initial assign always_comb always_ff None