

CSE369: Intro to Digital Design

L2: Logic Minimization, Verilog Deep-dive

Instructor: Naomi Alterman

Teaching Assistants:

Isabel Froelich
[Head TA]

Packard Stephenson

Mark Garcia

Ewan Kae

Violet Monserate

Avesta Rahimi

Administrivia

- ❖ Lab demo time slots are out on Ed
 - Use comments on the Ed thread to request lab time swaps

- ❖ Lab 1 this week – Basic Logic
 - **Report + code due on Wednesday @ 2:30p**
 - Check the lab report requirements closely
 - Show up to your demo time with your design synthesized and your simulations ready to view 🙌

Today

- ❖ **Logic Minimization**
- ❖ Verilog Details

Translating between Truth Tables and Boolean Equations

- ❖ Terms of equation come from rows of table
 - For “1” inputs, write variable name
 - For “0” inputs, write complement of variable
- ❖ Sum of Products (SoP)
 - From CSE311, “DNF” (disjunctive normal form)
 - Take truth table rows that output 1:
 - AND the inputs together, OR the rows together
- ❖ Product of Sums (PoS)
 - From CSE311, “CNF” (conjunctive normal form)
 - Take truth table rows that output 0:
 - OR the complemented inputs together, AND the rows together

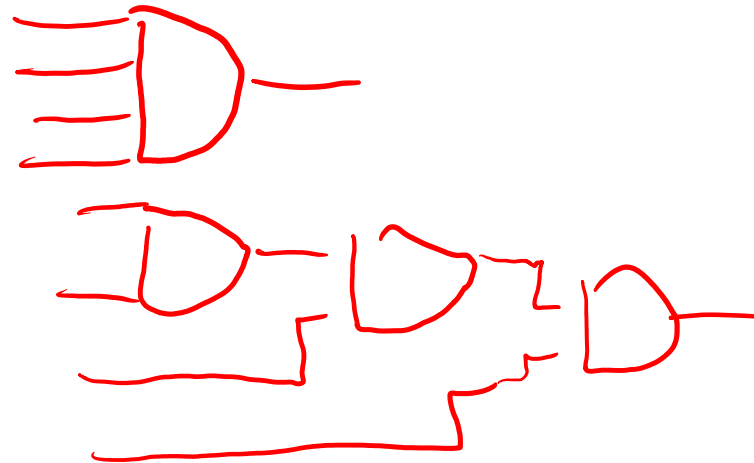
SoP: $C = \bar{A}B + \bar{B}A$

PoS: $\bar{C} = (A + B) \cdot (\bar{A} + \bar{B})$

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Logic minimization

- ❖ Goal: reduce complexity at the “gate” abstraction level
 - Allows us to build smaller and faster hardware
 - Care about both # of gates, # of literals (gate inputs), # of gate levels, and types of logic gates



Logic minimization

❖ Goal: reduce complexity at the “gate” abstraction level

- Allows us to build smaller and faster hardware
- Care about both # of gates, # of literals (gate inputs), # of gate levels, and types of logic gates

❖ Faster hardware?

- Fewer inputs implies faster gates in some technologies
- Fan-ins (# of gate inputs) are limited in some technologies
- Fewer levels of gates implies reduced signal propagation delays
- # of gates (or gate packages) influences manufacturing costs
- Simpler Boolean expressions → smaller transistor networks → smaller circuit delays → faster hardware

electrical implementation
of an abstract
gate

Are Logic Gates Created Equal?

❖ No!

2-Input Gate Type	# of CMOS transistors
NOT	2
AND	6
OR	6
NAND	4
NOR	4
XOR	8
XNOR	8

❖ Can recreate all other gates using only NAND or only NOR gates

■ Called “universal” gates

■ ...How?

■ For NAND:

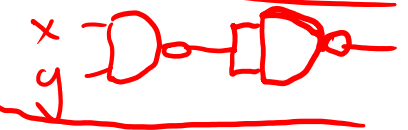
■ $\bar{X} = X \text{ NAND } X$

■ $X \cdot Y = \overline{(X \text{ NAND } Y)}$

■ $X + Y = \bar{X} \text{ NAND } \bar{Y}$

De Morgan's Law

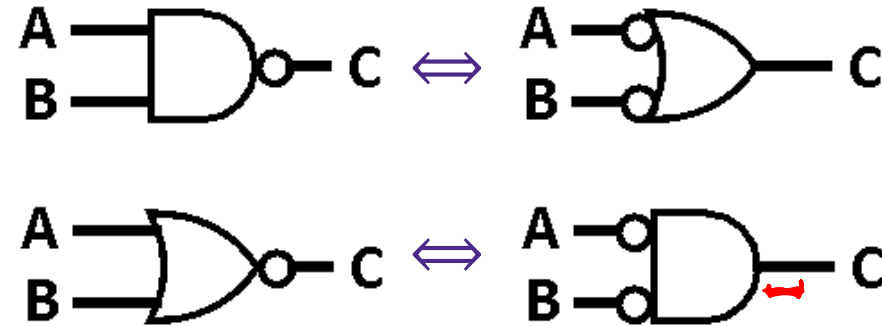
x	x	nand
0	0	1
0	1	1
1	0	1
1	1	0



DeMorgan's Law

$$\diamond \overline{X + Y} = \bar{X} \cdot \bar{Y}$$

$$\diamond \overline{X \cdot Y} = \bar{X} + \bar{Y}$$



- ❖ In Boolean Algebra, converts between AND-OR and OR-AND expressions

- $Z = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C$

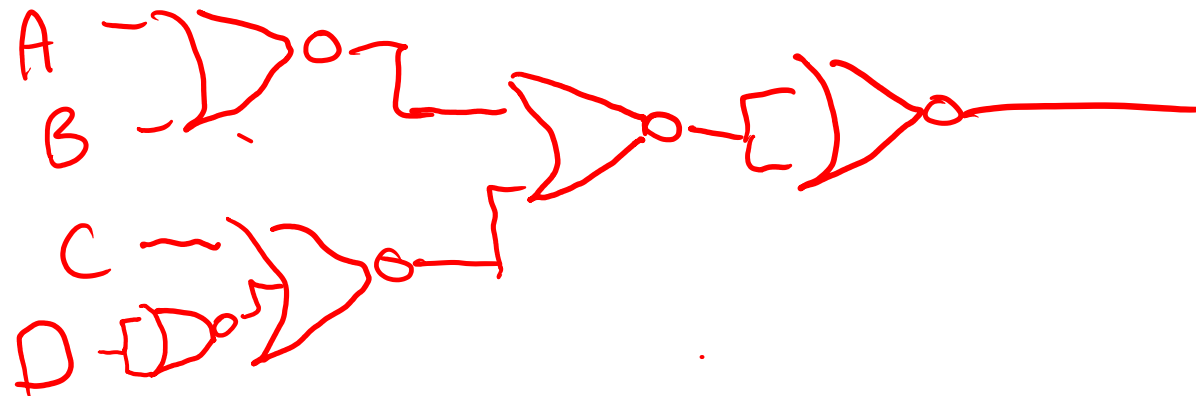
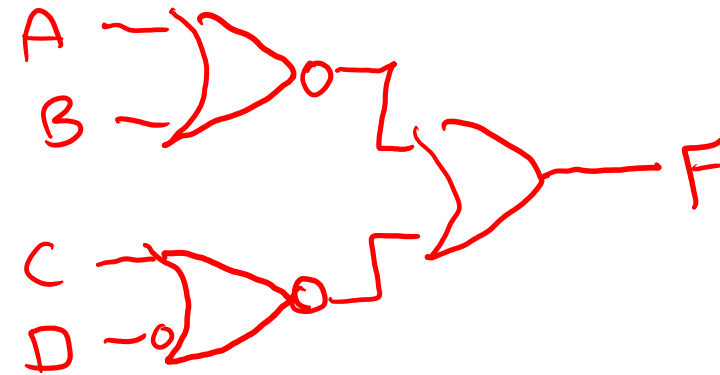
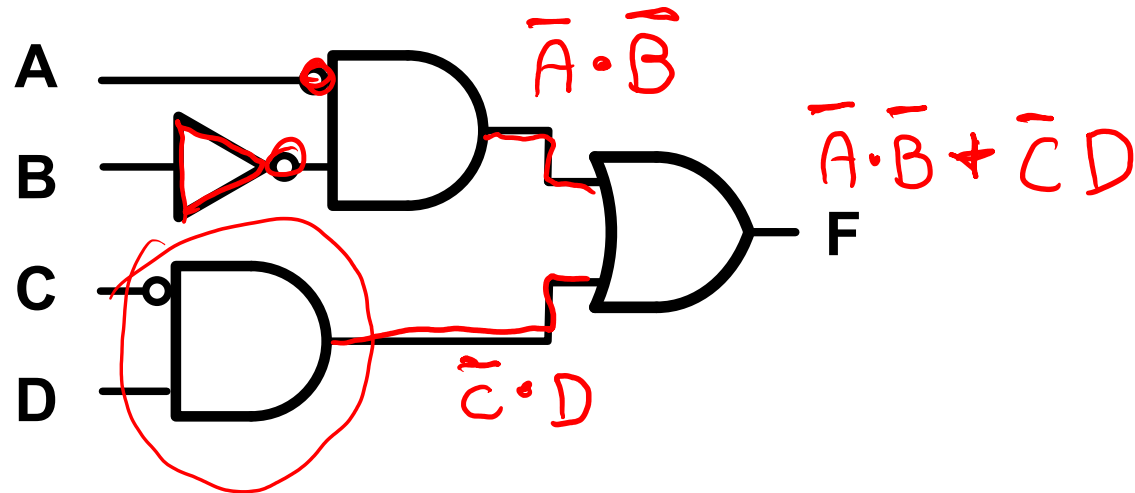
- $\bar{Z} = (A + B + \bar{C}) \cdot (A + \bar{B} + \bar{C}) \cdot (\bar{A} + B + \bar{C})$

- ❖ At gate level, can convert from AND/OR to NAND/NOR gates

 **"Bubble pushing":** "Flip" all input/output bubbles and "switch" gate

DeMorgan's Law Practice Problem

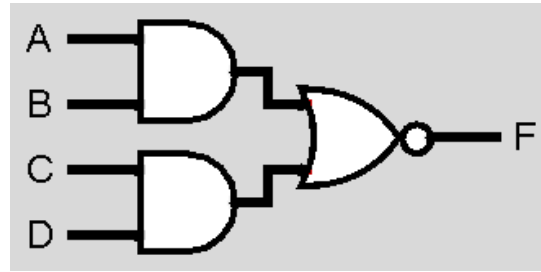
- ❖ Implement this circuit with only NOR gates:



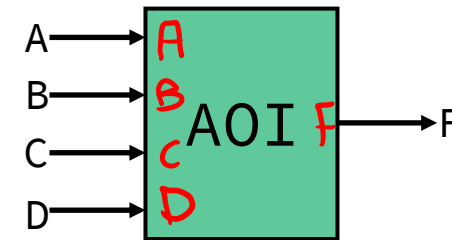
Today

- ❖ Logic Minimization
- ❖ **Verilog Details**

Basic Verilog example



Block Diagram:



// Verilog code for AND-OR-INVERT gate

```
→ module AOI (F, A, B, C, D);  
    output logic F;  
    input  logic A, B, C, D;  
  
    assign F = ~(A & B | C & D);  
endmodule
```

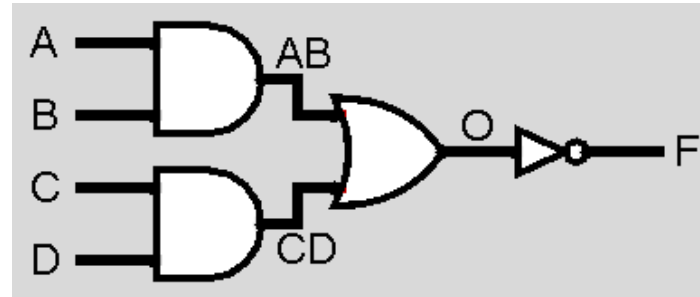
Verilog Data

- ❖ **Nets:** carry bits from one gate to another
 - SystemVerilog type: “**wire**”
 - Think of it *like* an immutable reference to mutable data in C++
- ❖ **Variables:** like a net, but the circuit setting the voltage can change
 - SystemVerilog type: “**reg**” or “**logic**”
 - NB: nothing to do with “registers” from Assembly (☹) !
 - “Variable” refers to the *source* of the data, not the data itself (“**driving** the voltage”)
 - Think of it *like* a pointer whose location is determined by a switch case in C++
- ❖ ...In this class, we’ll just use “**logic**” for everything

Verilog “Execution”

- ❖ Physical wires transmit voltages (electrons) near-instantaneously
 - Wires by themselves have no notion of sequential execution
- ❖ Gates and modules are **constantly** performing computations
- ❖ In pure hardware, there is **no** notion of **initialization**
 - A wire that is not driven by a voltage will naturally pick up a voltage from the environment
- ❖ In pure hardware, there is **no** notion of **reassignment**
 - Verilog variables represent physical wires. The **value** carried by the wire can change, but the wire's **endpoints** do not

Equivalent implementation



// Verilog code for AND-OR-INVERT gate

```
module AOI (F, A, B, C, D);  
    output logic F;  
    input  logic A, B, C, D;  
    logic  AB, CD, O; // intermediate signals  
  
    assign AB = A & B;  
    assign CD = C & D;  
    assign O  = AB | CD;  
    assign F  = ~O;  
endmodule
```

Verilog Primitives

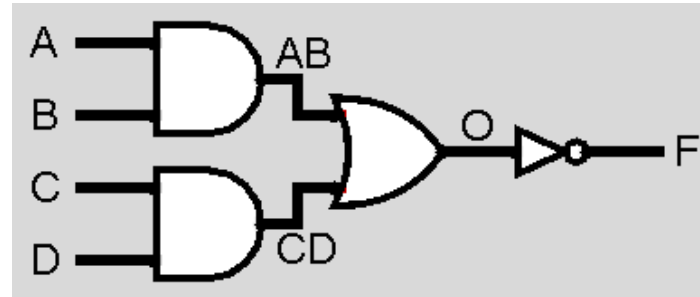
❖ Gates:

Gate	Verilog Syntax
NOT a	$\sim a$
a AND b	$a \ \& \ b$
a OR b	$a \ \ b$
a NAND b	$\sim(a \ \& \ b)$
a NOR b	$\sim(a \ \ b)$
a XOR b	$a \ ^ \ b$
a XNOR b	$\sim(a \ ^ \ b)$

❖ Modules: “classes” in Verilog that define *blocks* of logic

- **Input:** Signals passed from outside to inside of block
- **Output:** Signals passed from inside to outside of block

Another equivalent implementation



// Verilog code for AND-OR-INVERT gate

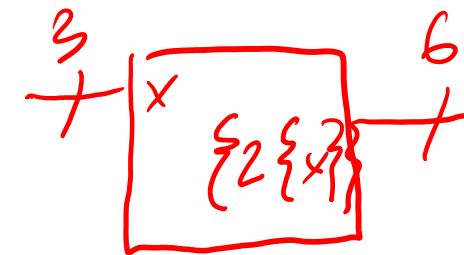
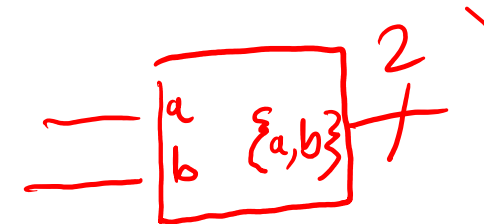
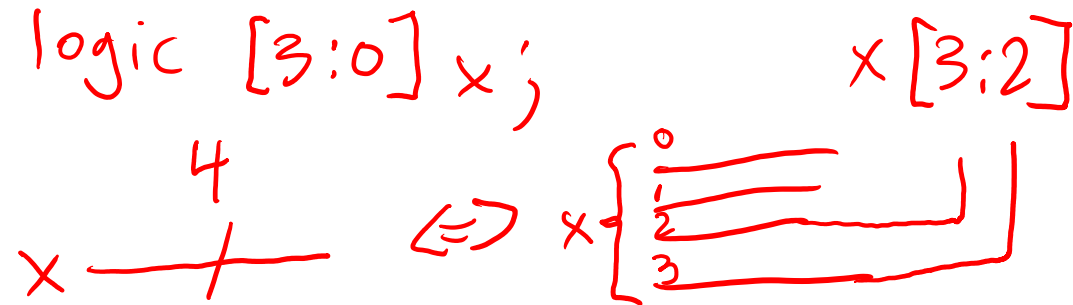
```
module AOI (F, A, B, C, D);  
    output logic F;  
    input  logic A, B, C, D;  
    logic  AB, CD, O; // intermediate signals  
  
    and a1(AB, A, B);  
    and a2(CD, C, D);  
    or  o1(O, AB, CD);  
    not n1(F, O);  
endmodule
```

} was:

```
assign AB = A & B;  
assign CD = C & D;  
assign O  = AB | CD;  
assign F  = ~O;
```

Verilog multi-bit signals

- ❖ Defining an N-bit signal: **logic** [N-1:0] varname;
 - Called an “**n-bit bus**”
 - Actually just N wires physically next to each other
 - Access bits and slices with “array” syntax (e.g., bus[1], bus[4:2])
 - Drawn in block diagrams as little labelled slashes (no slash means “1 bit”)
- ❖ **Signal concatenation:** {var1, ..., varN}
 - Ordering matters; left side is MSB, right side is LSB
 - Result's width is the combined widths of all signals
 - No standard block diagram notation, just draw a labelled block
- ❖ **Replication operator:** {n{m}}
 - repeats value m, n times
 - No standard block diagram notation, just draw a labelled block



Verilog data literals

❖ Logic Values

- **0** = zero, low, FALSE
- **1** = one, high, TRUE
- ➔ ■ **X** = unknown, uninitialized, or contention (“multiple drivers”)
- ➔ ■ **Z** = floating/disconnected/high impedance/“no drivers”

➔ Multi-bit constants: $n'r###...$

- n is bit width, b is radix/base, $\#$ s are the actual digits
- e.g., $4'd12$, $4'b1100$, $4'hC$

1
0xC

Let's check our understanding

```

logic [4:0] apple;
logic [3:0] pear;
logic [9:0] orange;
assign apple = 5'd20;
assign pear = {1'b0, apple[2:1], apple[4]};

```

Handwritten annotations:

- For `apple = 5'd20`, the value 20 is written in binary as $4'b10100$ with bit positions 4, 3, 2, 1, 0 written above it.
- For `pear = {1'b0, apple[2:1], apple[4]}`, the components are annotated: `1'b0` is 0, `apple[2:1]` is 10, and `apple[4]` is 1.

- ❖ What's the value of pear?

Handwritten answer:

$$pear = 4'b0101 = 4'd5$$

- ❖ If we want orange to be the *sign-extended* version of apple, what is the appropriate Verilog statement?

Handwritten answer:

```

assign orange = { {5{apple[4]}}, apple }

```

Miso Moment

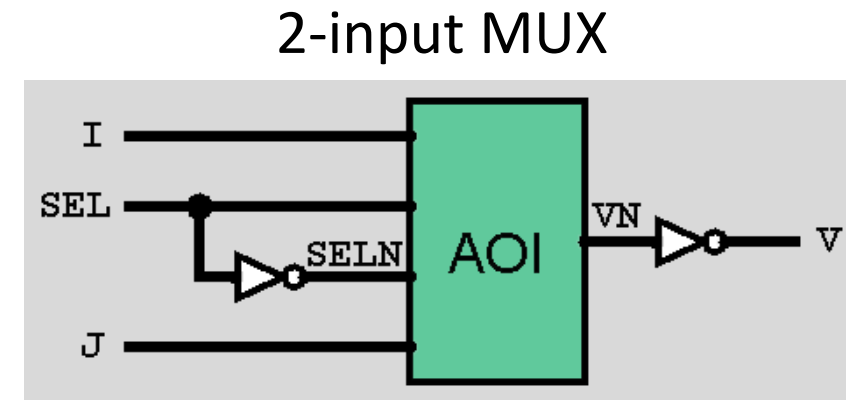


Module Hierarchy

// Verilog code for 2-input multiplexer

```
module AOI (F, A, B, C, D);  
    output logic F;  
    input  logic A, B, C, D;  
  
    assign F = ~((A & B) | (C & D));  
endmodule
```

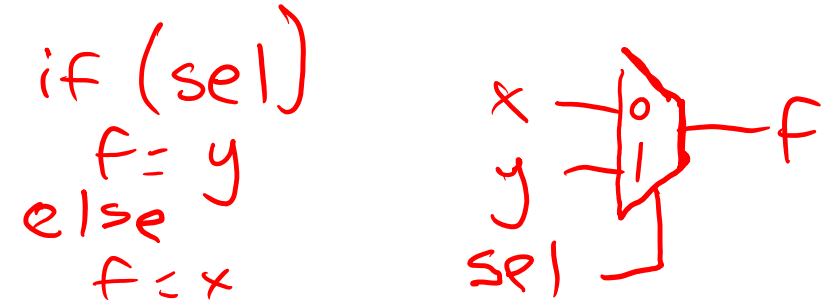
```
module MUX2 (V, SEL, I, J);    // 2:1 multiplexer  
    output logic V;  
    input  logic SEL, I, J;  
    logic  SELN, VN;  
  
    not G1 (SELN, SEL);  
    AOI G2 (.F(VN), .A(I), .B(SEL), .C(SELN), .D(J));  
    not G3 (V, VN);  
endmodule
```



Verilog “control flow”

❖ Verilog has synthesizable **if** and **case** statements

- In hardware, each “clause” exists in parallel and is always computing its output whether the condition is true or not
- A **multiplexer (“mux”)** is placed physically after these blocks to electrically connect one of them to the output signal



❖ Verilog requires us to wrap these statements in an **always_comb** block

- Block defines the full set of circuits that *may* drive the value on a **logic** variable
- Idea: the last assignment in an always block to a given variable is the result that gets used

❖ There are several other kinds of “**always**” block.

- We’ll learn about them in a few weeks!

Getting muxy

3-Input Majority

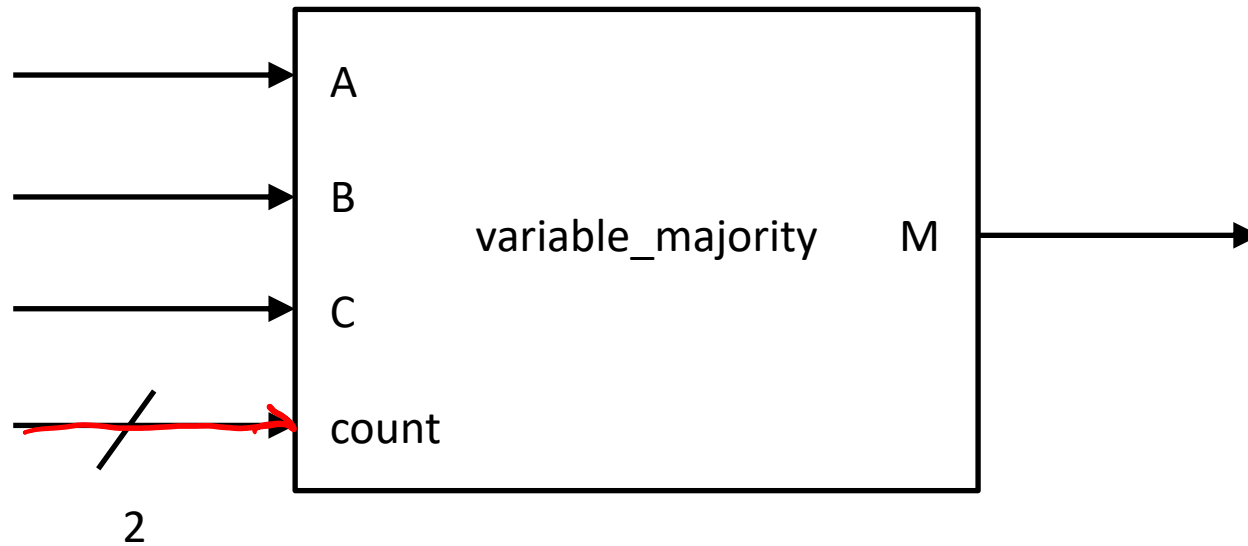
A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

```
module majority_three(  
    input logic A, input logic B, input logic C,  
    output logic M  
);  
    always_comb begin  
        case ({A,B,C})  
            3'b000: M = 0;  
            3'b001: M = 0;  
            3'b010: M = 0;  
            3'b011: M = 1;  
            3'b100: M = 0;  
            3'b101: M = 1;  
            3'b110: M = 1;  
            3'b111: M = 1;  
        endcase  
    end  
endmodule
```

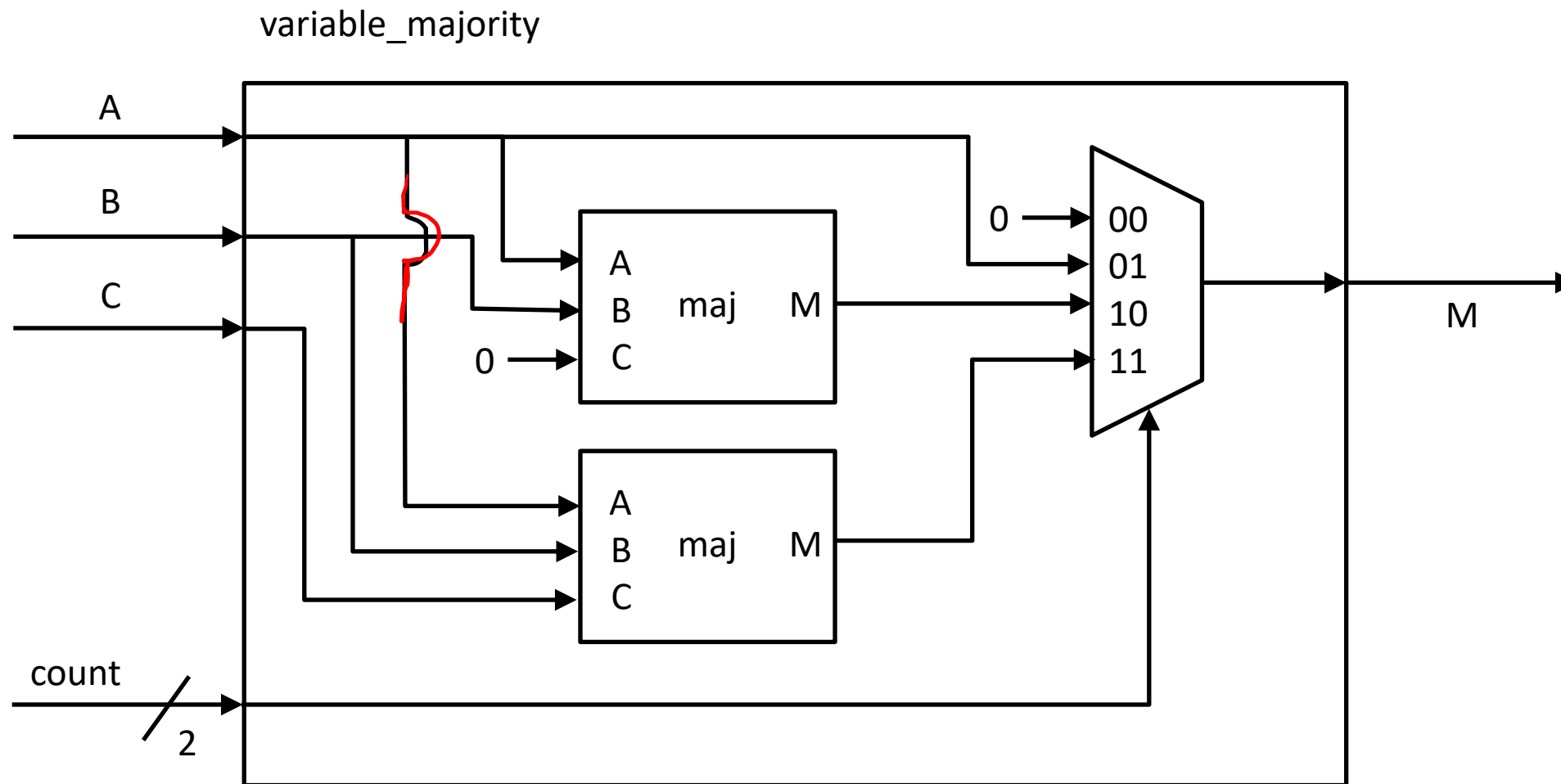
concat syntax

Getting muxier

- ❖ Let's make a majority circuit for *up to three* inputs



Muxes all the way down



Muxes all the way down (Verilog)

```
module variable_majority (  
    input logic A, input logic B, input logic C,  
    input logic [1:0] count,  
    output logic M,  
);  
    logic maj2, maj3;  
    majority_three two_input (.A(A), .B(B), .C(1'b0), .M(maj2));  
    majority_three three_input (.A(A), .B(B), .C(C), .M(maj3));  
    always_comb begin  
        if (count == 3'd1) begin  
            M = A;  
        end else if (count == 3'd2) begin  
            M = maj2;  
        end else if (count == 3'd3) begin  
            M = maj3;  
        end else begin  
            M = 0;  
        end  
    end  
end  
endmodule
```

one block
on yep
diagram

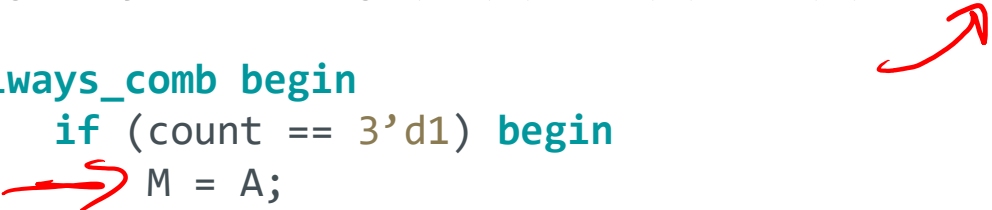


Bad Verilog

```
module variable_majority (  
    input logic A, input logic B, input logic C,  
    input logic [1:0] count,  
    output logic M,  
);  
    logic maj2, maj3;  
    always_comb begin  
        if (count == 3'd1) begin  
            M = A;  
        end else if (count == 3'd2) begin  
            majority_three two_input (.A(A), .B(B), .C(1'b0), .M(maj2));  
            M = maj2;  
        end else if (count == 3'd3) begin  
            majority_three three_input (.A(A), .B(B), .C(C), .M(maj3));  
            M = maj3;  
        end else begin  
            M = 0;  
        end  
    end  
end  
endmodule
```

Bad Verilog

```
module variable_majority (  
    input logic A, input logic B, input logic C,  
    input logic [1:0] count,  
    output logic M,  
);  
    majority_three maj (.A(A), .B(B), .C(C), .M(M));  
  
    always_comb begin  
        if (count == 3'd1) begin  
            M = A;  
        end else if (count == 3'd2) begin  
            C = 1'b0;  
        end else if (count == 3'd3) begin  
            // nothing to do...  
        end else begin  
            M = 0;  
        end  
    end  
end  
  
endmodule
```



Muxes all the way down

