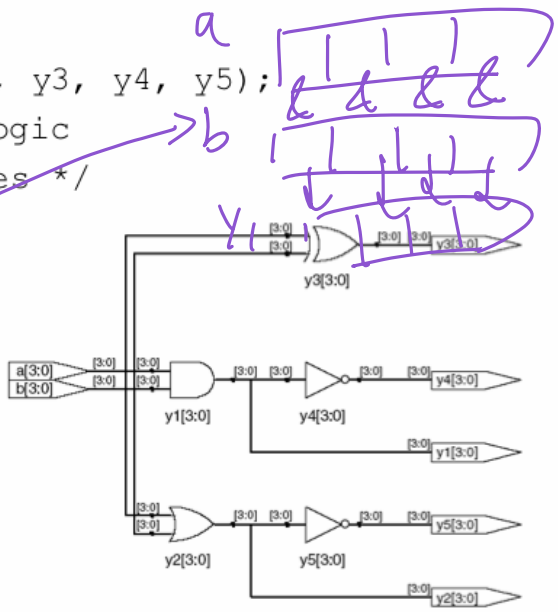
Structural Modeling - Hierarchy

```
module and3(input  a, b, c,
            output y);
  assign y = a & b & c;
endmodule


module inv(input  a,
           output y);
  assign y = ~a;
endmodule


module nand3(input  a, b, c
             output y);
  wire n1;                      // internal signal

  and3 andgate(a, b, c, n1);  // instance of and3
  inv  inverter(n1, y);       // instance of inverter
endmodule
```


ELSEVIER

Bitwise Operators

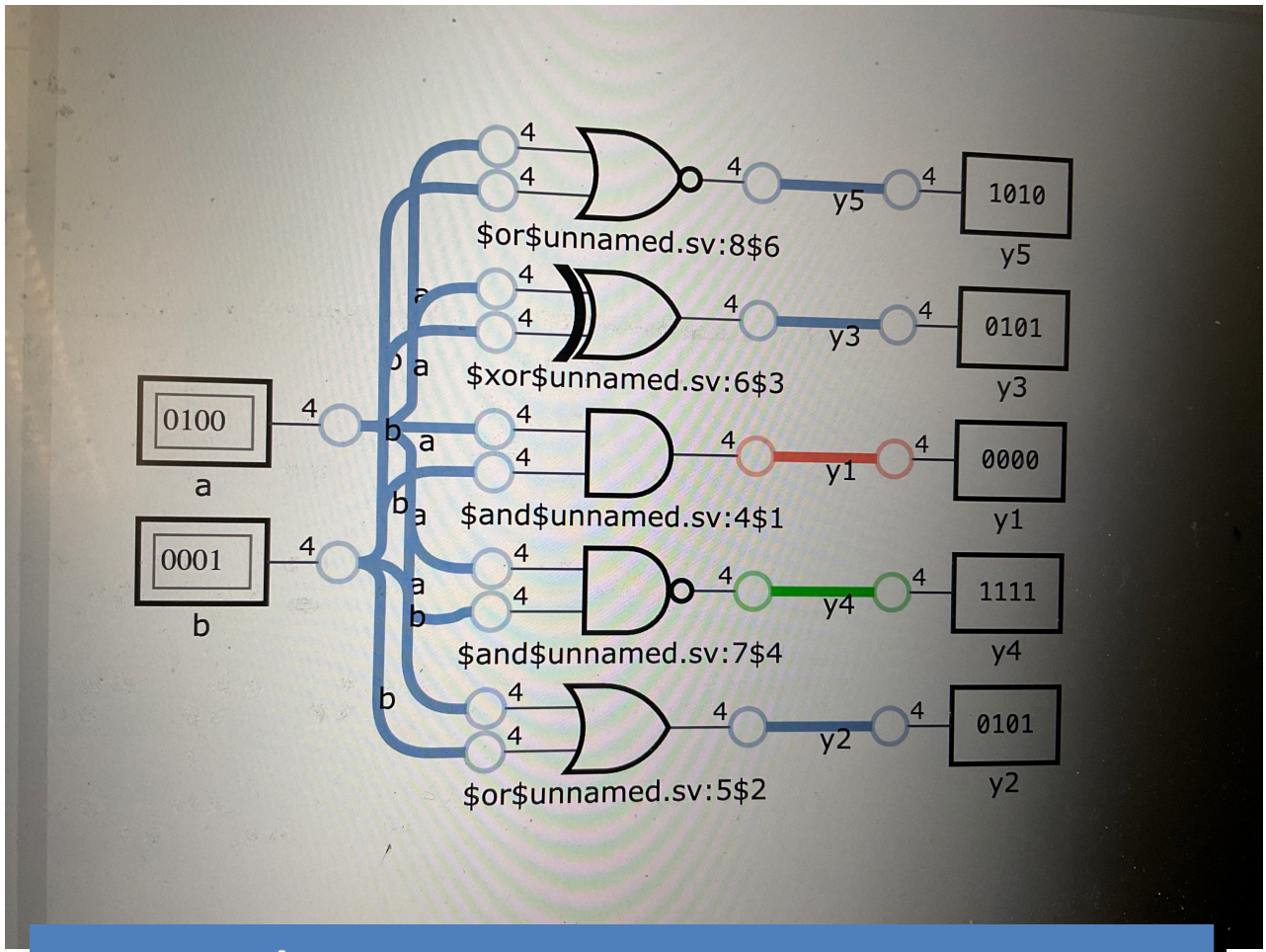
```
module gates(input  [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```



// single line comment

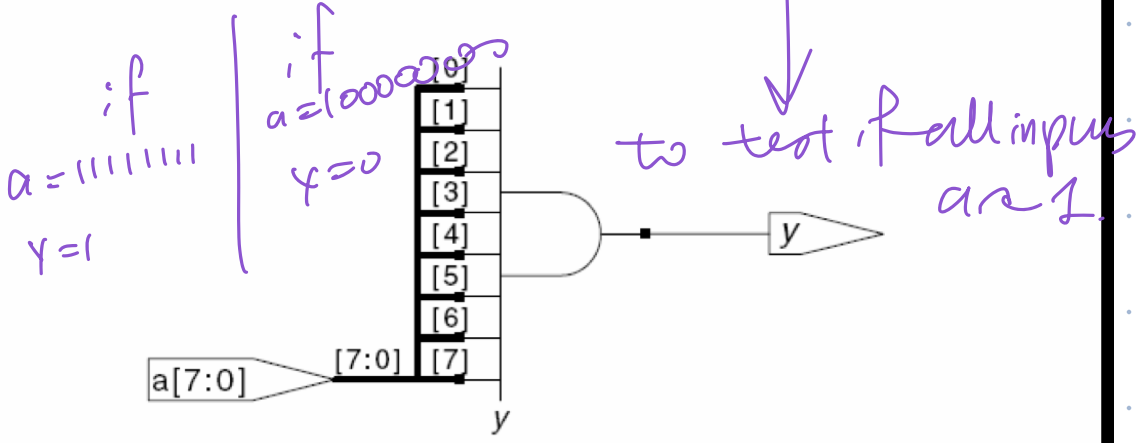
/*...*/ multiline comment

ELSEVIER



Reduction Operators

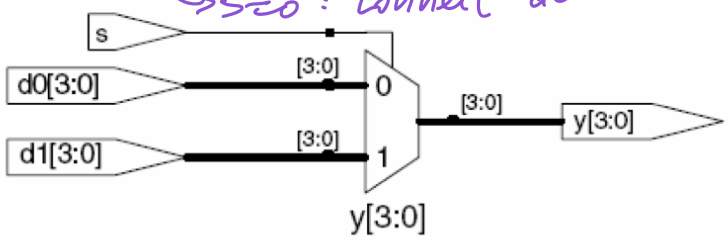
```
module and8(input  [7:0] a,
            output       y);
   assign y = &a;
   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```



Conditional Assignment

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

s=1 : connect d1
s=0 : connect d0

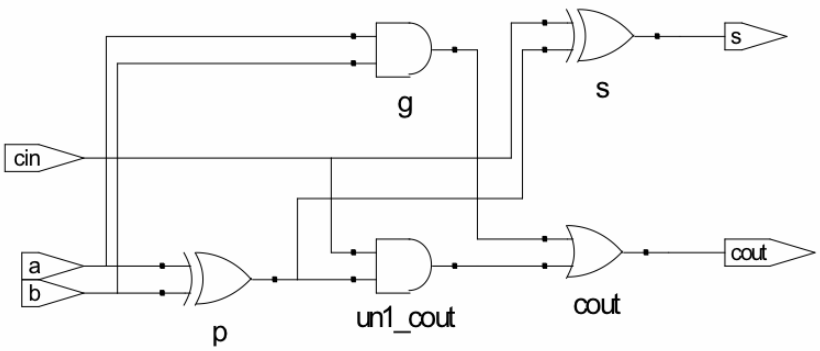


`? :` is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

ELSEVIER

Internal Variables

```
module fulladder(input  a, b, cin, output s, cout);
  wire p, g;        // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



ELSEVIER

Precedence

Defines the order of operations

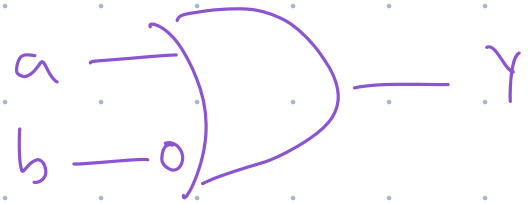
Highest

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

ELSEVIER

for example: $y = a | \sim b$



Numbers

Format: N'Bvalue

N = number of bits, B = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

Bit Manipulations: Example 1

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// if y is a 12-bit signal, the above statement produces:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

// underscores (_) are used for formatting only to make
   it easier to read. Verilog ignores them.
```

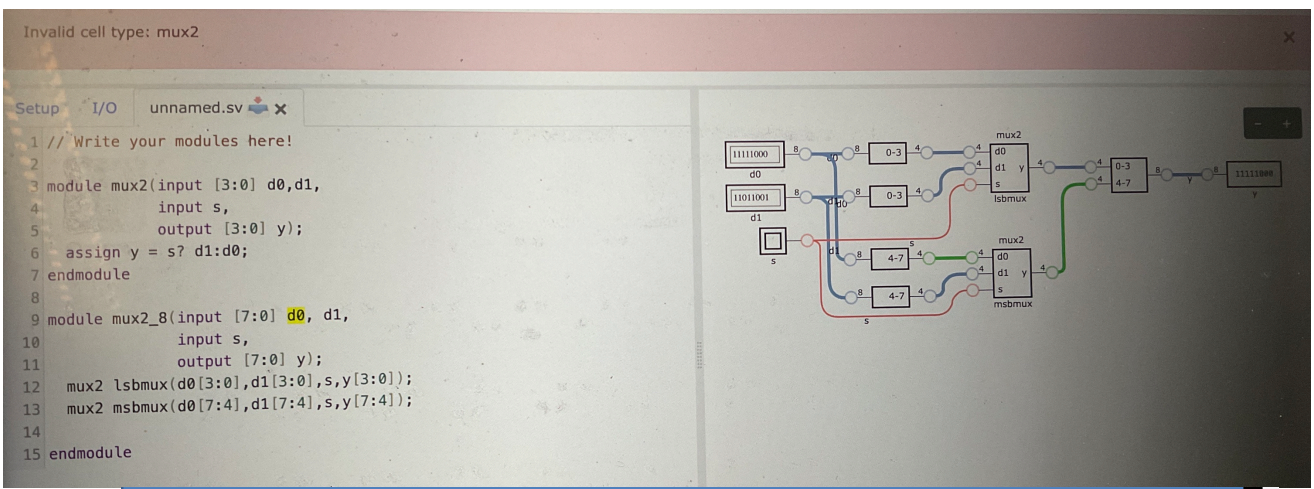
set a = 100 ⟹ a[2] = 1 a[0] = 0.

Bit Manipulations: Example 2

Verilog:

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [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
```

Synthesis: (didn't get what this function does.

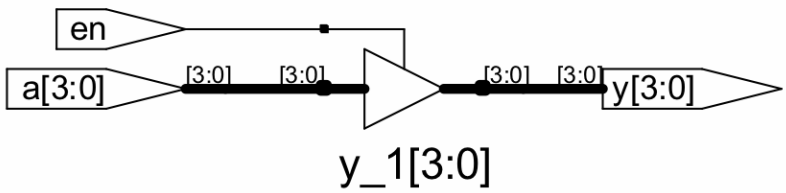


Z: Floating Output

Verilog:

```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

Synthesis:



Introduction: Sequential Logic.

↳ Depends on current & prior input values → it has memory.

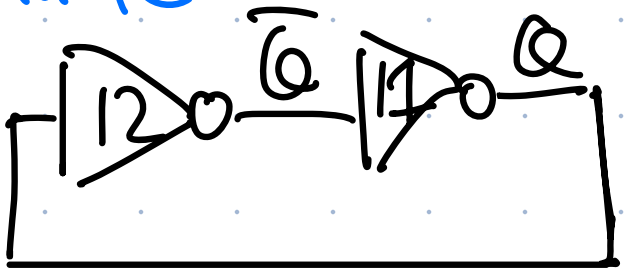
- State: all info needed to det. future state.
- Latches & flip-flops: state elements that store one bit of state.
- synchronous sequential circuits: combinational logic followed by a bank of flip flops

State elements that store state

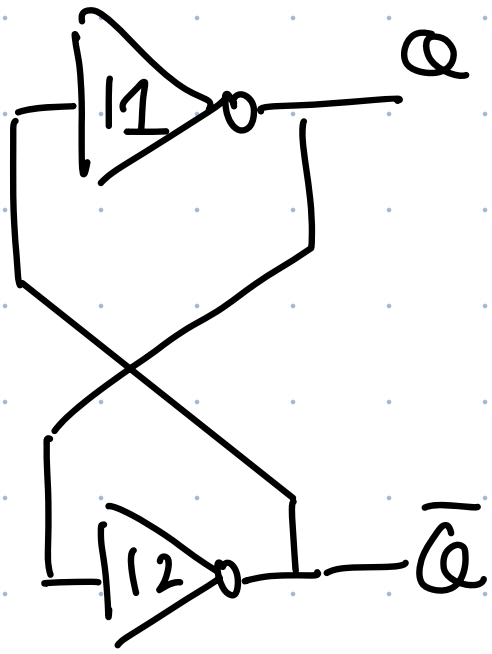
- Bistable Circuit
- SR Latch
- D Latch
- D Flip-Flop

Problem: No inputs where's the input.

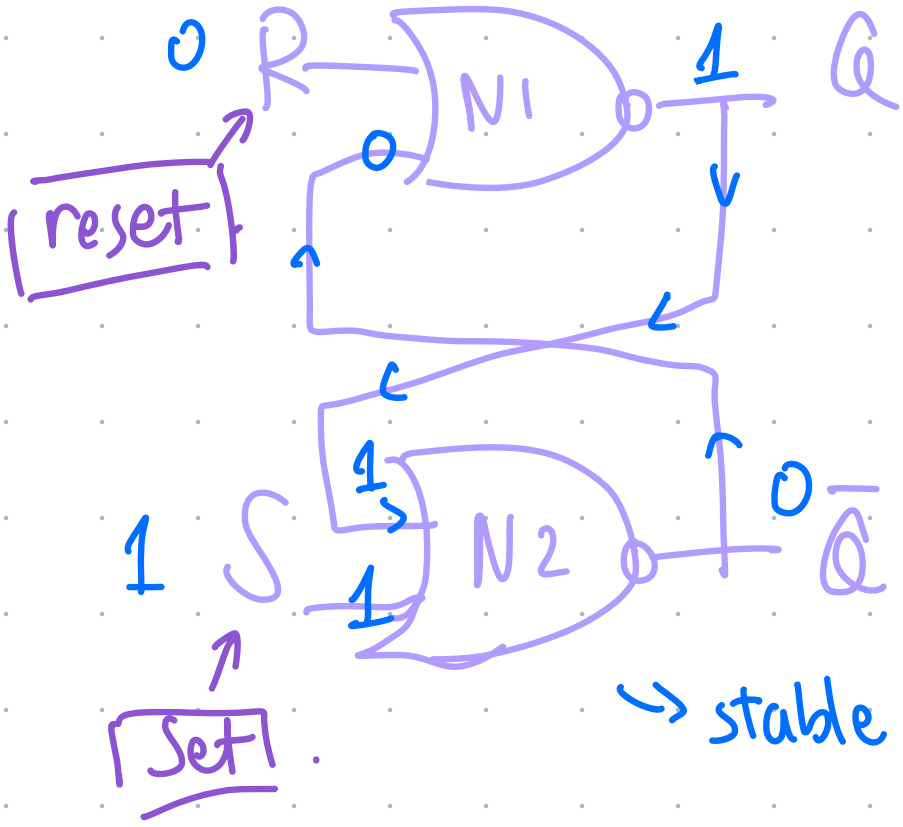
Bistable Circuit



stable state $\begin{cases} Q=0 \rightarrow \bar{Q}=1 \\ Q=1 \rightarrow \bar{Q}=0 \end{cases}$



SR Latch.

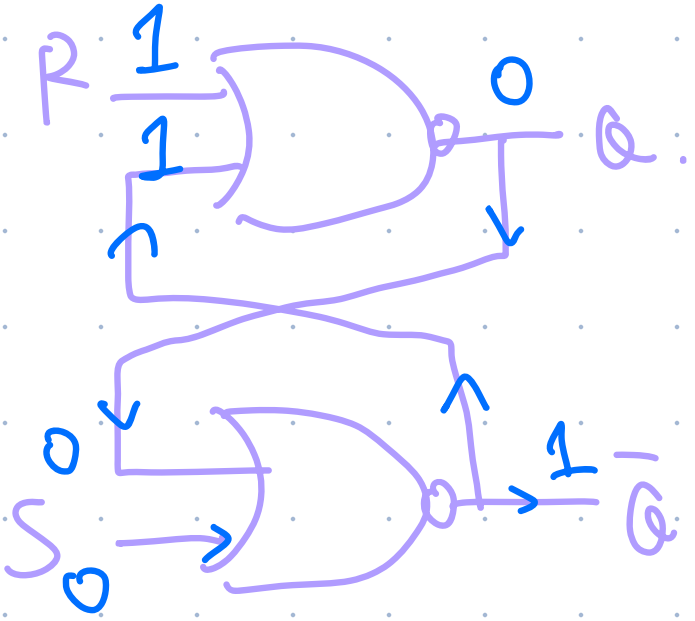


→ stable.

then

$Q = 1$ &

$\bar{Q} = 0$.



then

$Q = 0$

$\bar{Q} = 1$

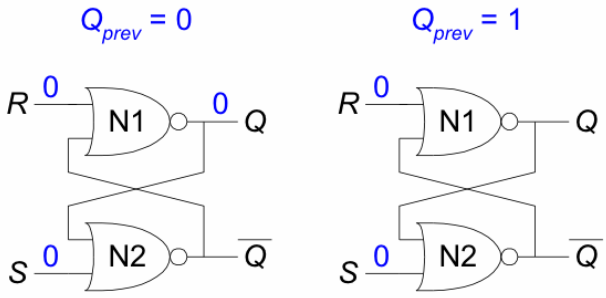
SR Latch Analysis

– $S = 0, R = 0$:

then $Q = Q_{prev}$

Memory!

that's cool!

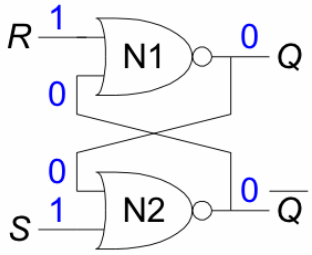


– $S = 1, R = 1$:

then $Q = 0, \overline{Q} = 0$

Invalid State

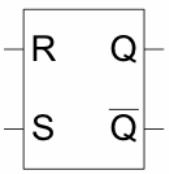
$\overline{Q} \neq$ NOT Q



ELSEVIER

- SR stands for Set/Reset Latch
  - Stores one bit of state (Q)
- Control what value is being stored with S, R inputs
  - **Set:** Make the output 1 ($S = 1, R = 0, Q = \mathbf{1}$)
  - **Reset:** Make the output 0 ($S = 0, R = 1, Q = \mathbf{0}$)
- **Must do something to avoid invalid state (when $S = R = 1$)**

SR Latch Symbol



ELSEVIER

D Latch

- Two inputs: *CLK*, *D*
  - ***CLK***: controls *when* the output changes
  - ***D*** (the data input): controls *what* the output changes to
- Function
  - When ***CLK* = 1**,

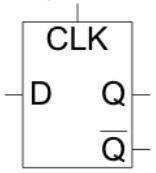
    D passes through to *Q* (*transparent*)
  - When ***CLK* = 0**,

    Q holds its previous value (*opaque*)
- Avoids invalid case when

  $Q \neq \text{NOT } \overline{Q}$

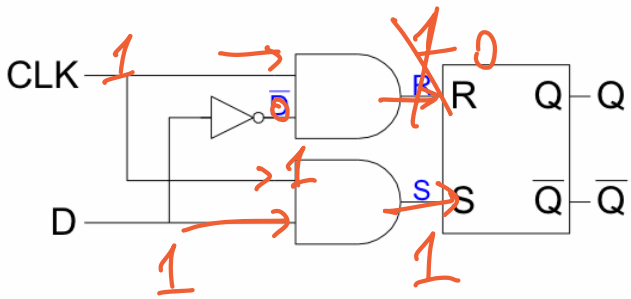
that's cool!

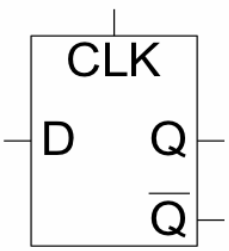
D Latch Symbol



ELSEVIER

D Latch Internal Circuit





CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X					
1	0					
1	1					

ELSEVIER