Test 2

ENGR 210 / CSCI B441

Addition / Subtraction II

Andrew Lukefahr

Announcements

- P2 is out! Due next Friday
- c and v bits updates

carry

overflow

Add = 1 + 1 = 2

a = 0000 0001

b = 0000 0001 =

s = 1010

r = 0000 0000

C = 0 for everything but Addition & Subtraction

V = defined for Addition & Sub

→ undefined for all other operations

→ AG will use V=0 for all cases other than Addition & Subtraction

wire vs logic

- **wire**
  - Only used with 'assign' and module outputs
  - Boolean combination of inputs
  - Can never hold state
- **logic**
  - Used with 'always' and module outputs
  - Can be Boolean combination of inputs
  - Can also hold state

```
wire w;
assign w = 'h0;
```

```
// w = 1 if x == 1
//     0 if x == 2
//     1 if x == 3
```

always_comb Blocks

```
wire foo;
assign foo = x & y | z;
```

... is equivalent to ...

```
wire foo = x & y | z;
```

... is equivalent to ...

```
logic foo;
always_comb //comb-inational
    foo = x & y | z;
```

```
logic foo;
always-comb begin
    if(x & y | z)
        foo = 1'h1;
    else
        foo = 1'h0;
end
```

wrong: `logic foo = x & y;`

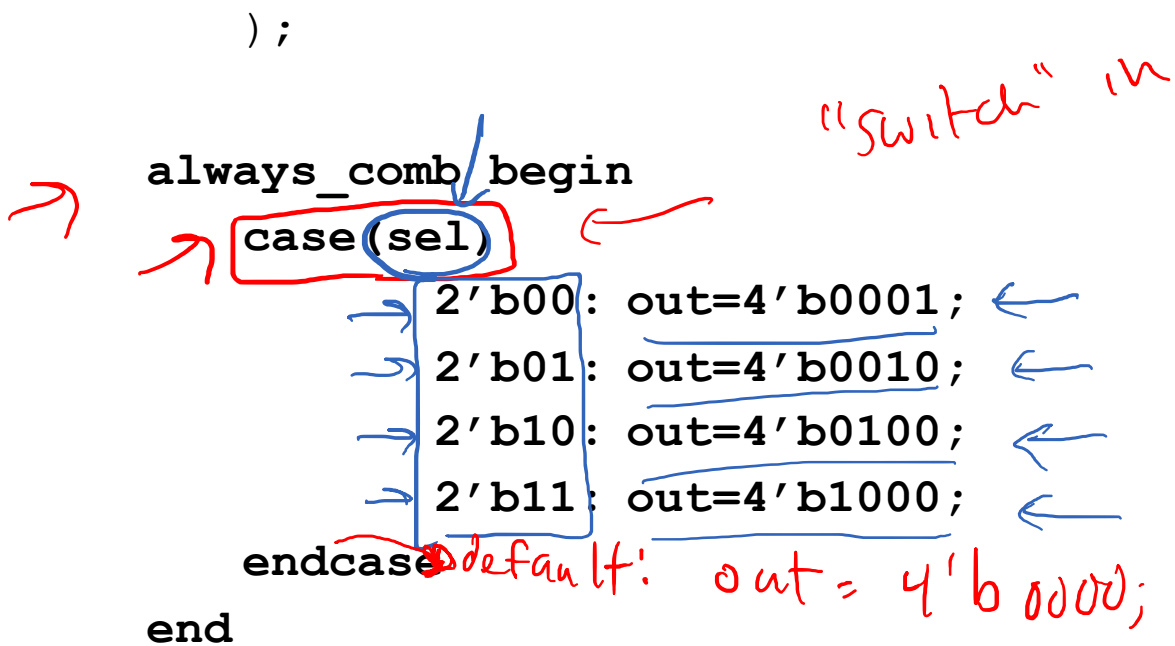
works:
```
logic foo;
assign foo = x & y;
```

always_comb blocks with if

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );
    always_comb begin
        if (sel == 2'b00) begin
                out = 4'b0001;
        end else if (sel == 2'b01) begin
                out = 4'b0010;
        end else if (sel == 2'b10) begin
                out = 4'b0100;
        end else if (sel == 2'b11) begin
                out = 4'b1000;
        end
    end
endmodule
```

always_comb with case

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        case(sel)
            2'b00: out=4'b0001;
            2'b01: out=4'b0010;
            2'b10: out=4'b0100;
            2'b11: out=4'b1000;
        endcase
    end

endmodule
```



always_comb with case

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        case(sel)
                2'b00: out=4'b0001;
                2'b01: out=4'b0010;
                2'b10: out=4'b0100;
                                        // what about sel==2'b11?
        endcase
    end

endmodule
```

out can be whatever!

* add logic after class

always_ff
next time

always_comb with case

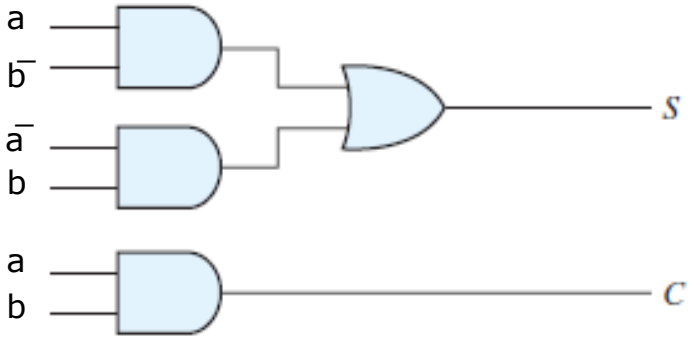
```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        out = 4'b0000; //default
        case(sel)
                2'b00: out=4'b0001;
                2'b01: out=4'b0010;
                2'b10: out=4'b0100;
                default: out=4'b0000; // what about sel==2'b11?
        endcase
    end

endmodule
```

Always specify defaults for **always_comb**!

"Warning: Inferred Latch"

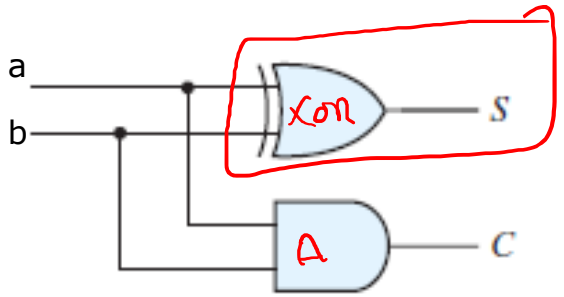
Half Adder

i	*a*	*b*	*C*	*S*
0	0	0	0	0
1	0	1	0	1
2	1	0	0	1
3	1	1	1	0



$S = a\,\bar{b} + \bar{a}\,b = a \oplus b$

$C = a\,b$

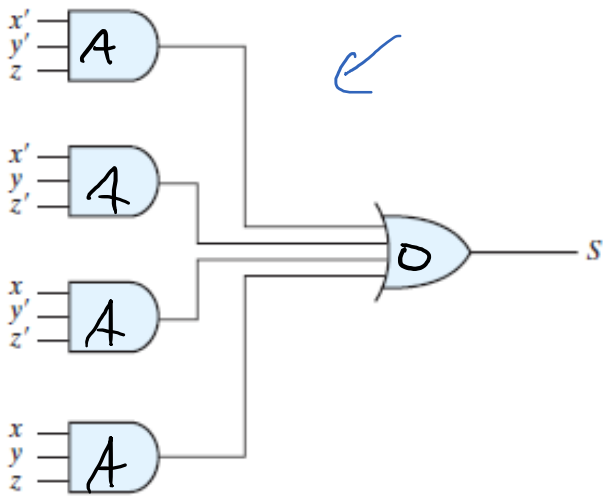


P2: use "+" in Verilog

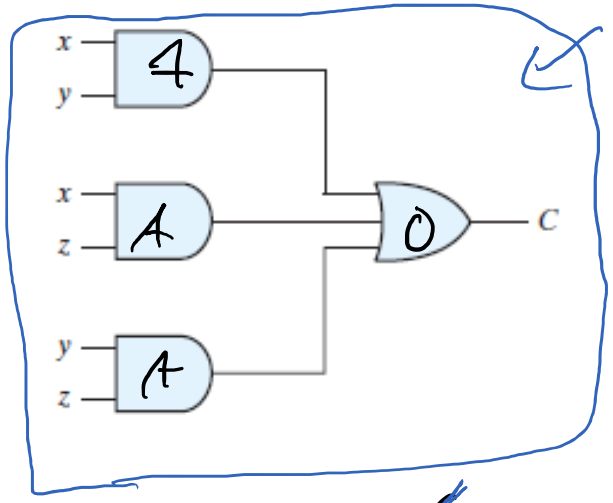
Full Adder

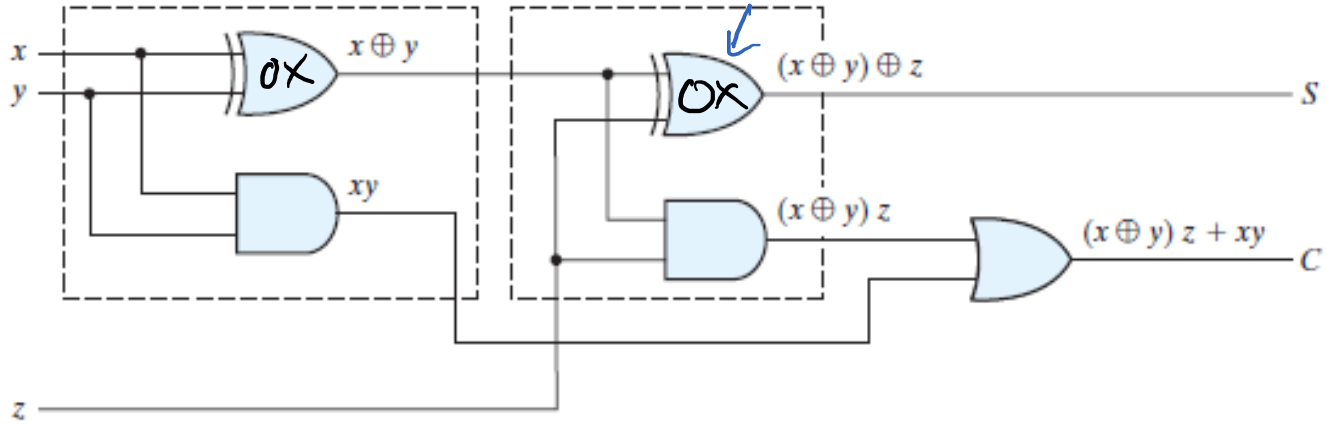
i	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Sum

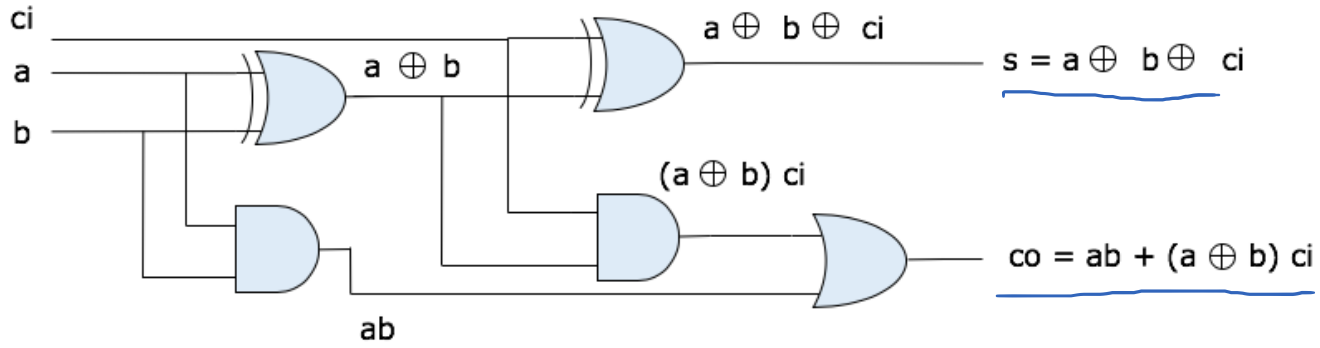


Carry



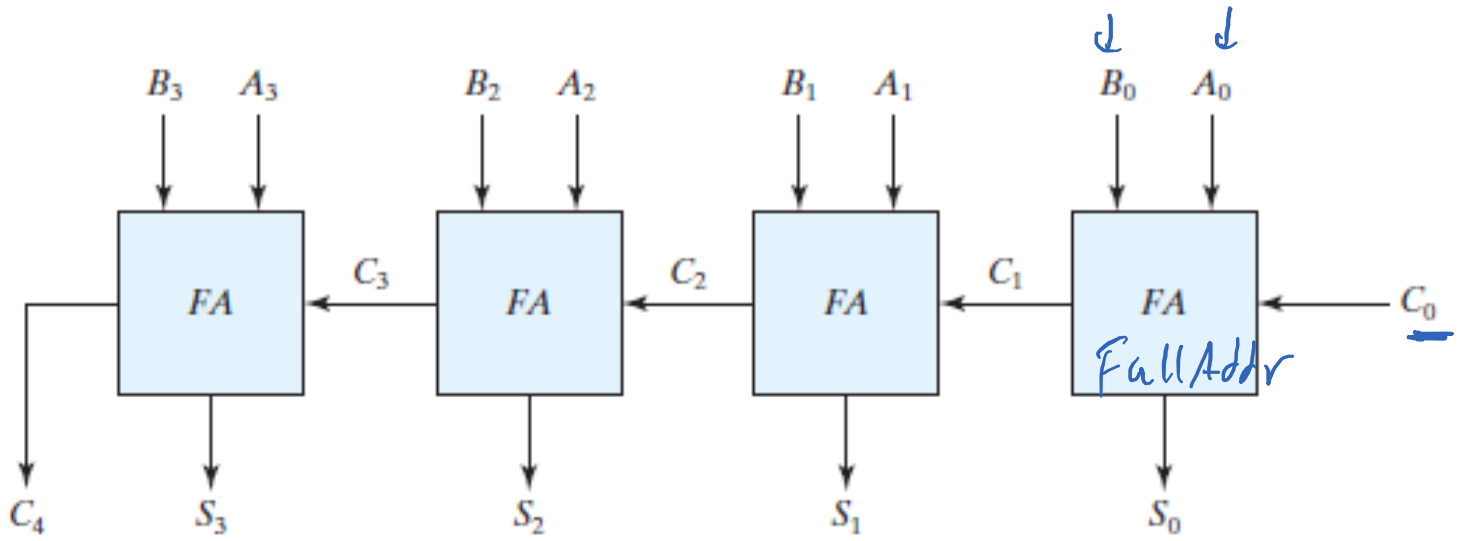


1-Bit “Full” Adder




```
module FullAddr (
   input a,b,ci,
   output s, co
   );

   assign s = a ^ b ^ ci;
   assign co = (a & b) | (( a ^ b) & ci);

endmodule
```

Ripple-Carry Adder

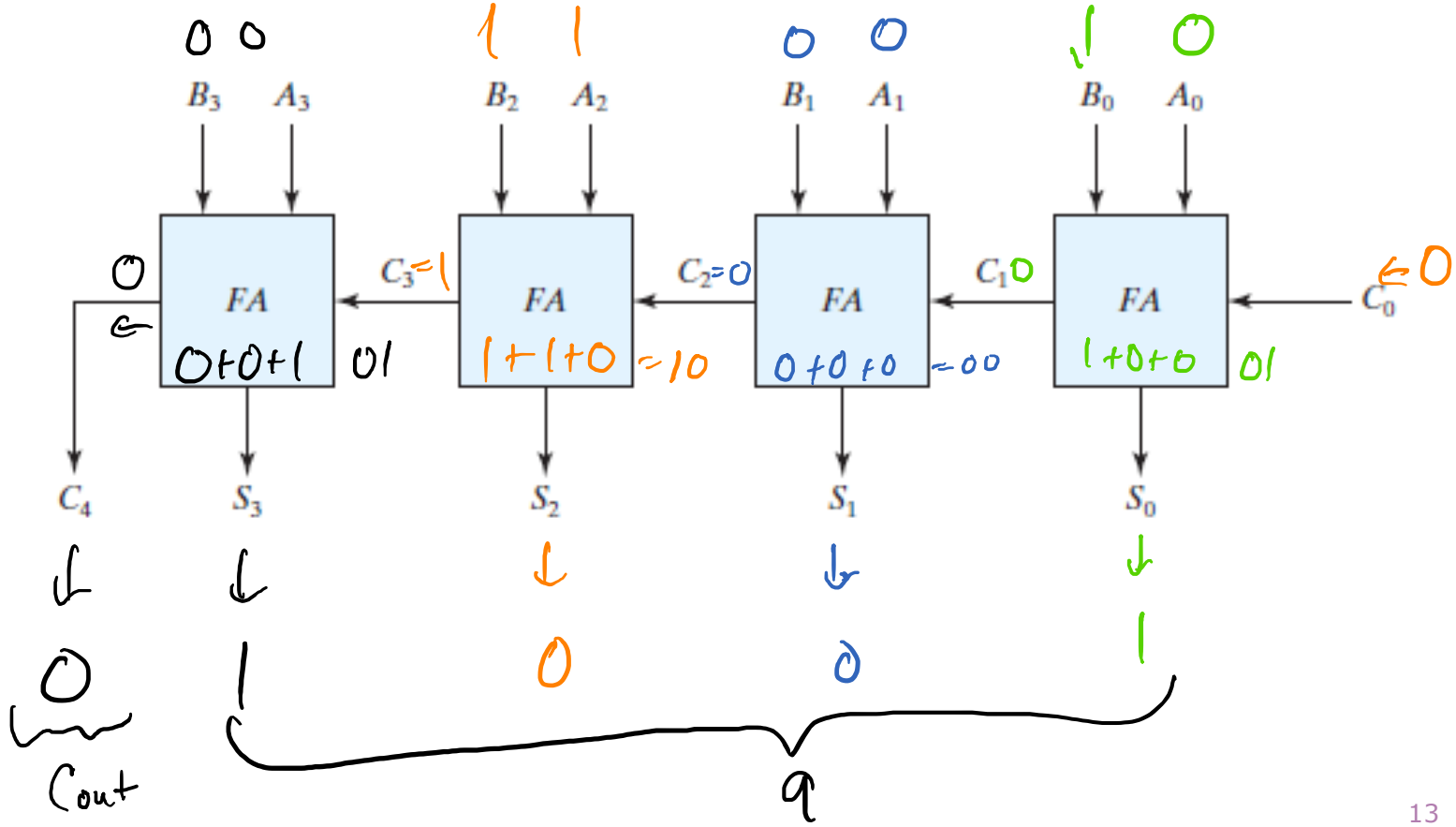


Ripple-Carry Adder

$a = 4 = 0100$

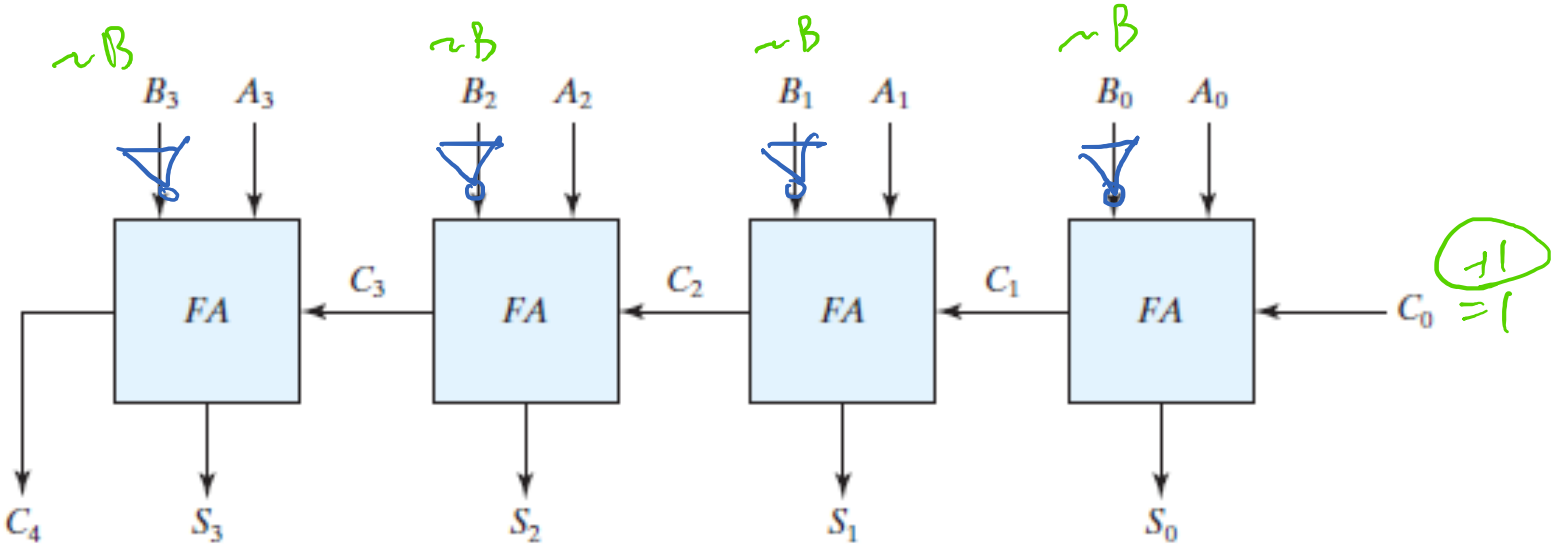
$+\ b = 5 = +0101$

$q = 1001$



$A - B = A + (-B) = A + (\sim B + 1)$

Subtraction with Adders?



Adder/Subtractor

m = 0

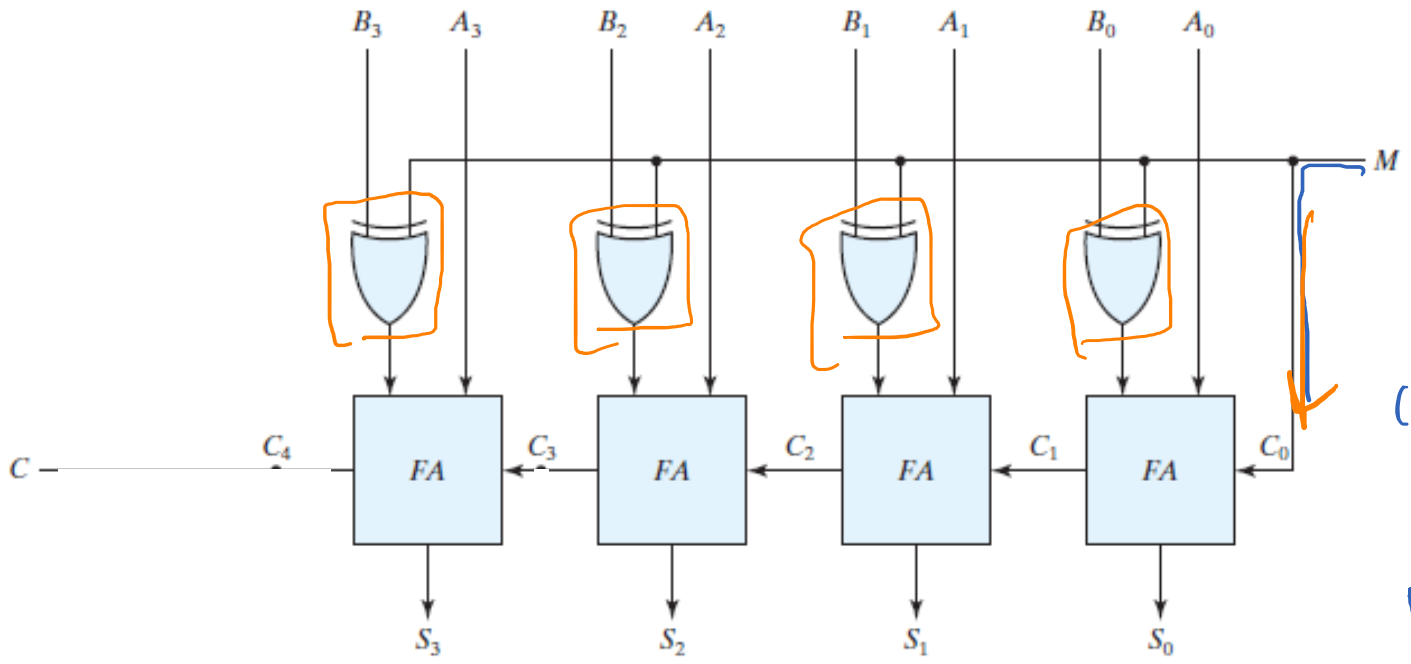
S = A + B + 0

m = 1

S = A + (~B) + 1

- Mode input:
  - If *M = 0,* then *S = A + B,* the circuit performs addition
  - If *M = 1,* then *S = A + $\overline{B}$ + 1,* the circuit performs subtraction

B	m	XOR
0	0	0
0	1	1
1	0	1
1	1	0



if M = 0

XOR = B

if m = 1

XOR = ~B

if M = 0

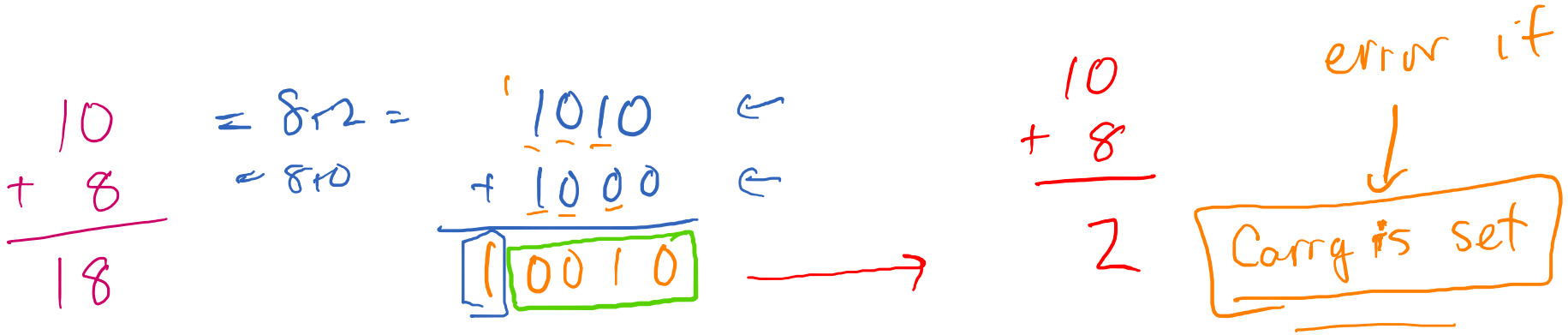
$C_0 = 0$

if m = 1

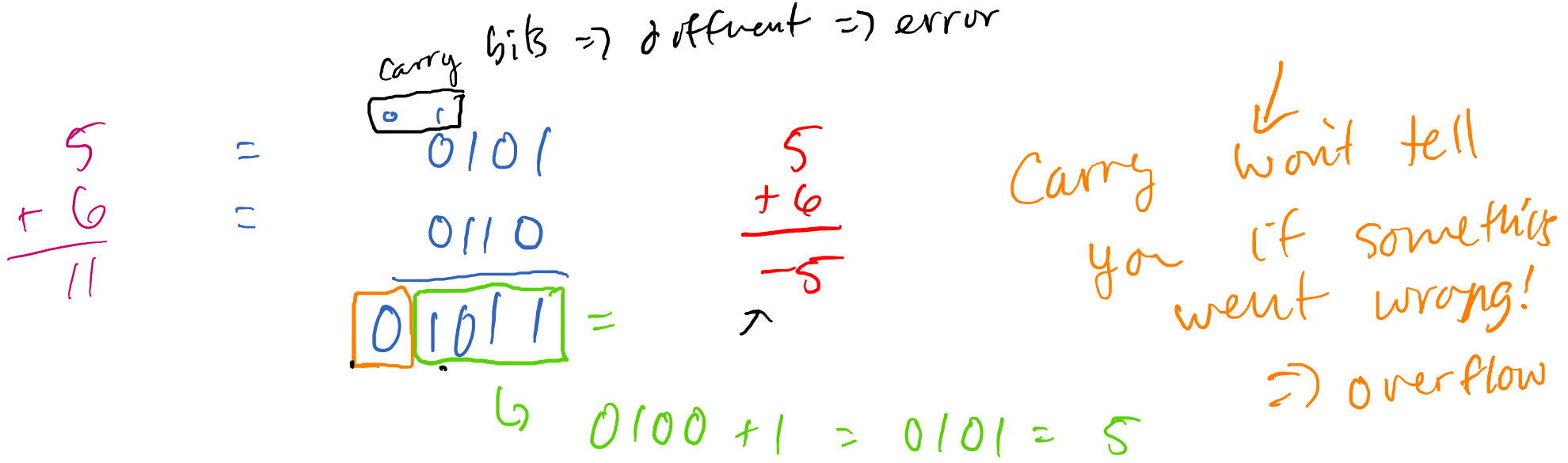
$C_0 = 1$

Overflow for signed numbers?

- Unsigned



- Signed



Overflow for signed numbers?

$$\begin{array}{r} -2 \\ +-1 \\ \hline -3 \end{array}$$

Overflow for signed numbers

$$\begin{array}{cccccc} c_4 & c_3 & c_2 & c_1 & & \\ & x_3 & x_2 & x_1 & x_0 & \\ + & y_3 & y_2 & y_1 & y_0 & \\ \hline z_4 & z_3 & z_2 & z_1 & z_0 & \end{array}$$

Overflow detection

- When two numbers with *n* digits each are added and the sum is a number occupying *n* + 1 digits, we say that an overflow occurred.
- The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.
- When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.
- In case of signed numbers, two details are important:
  the leftmost bit always represents the sign,
  negative numbers are in 2's-complement form.
- When two signed numbers are added:
  the sign bit is treated as part of the number
  the end carry does not indicate an overflow.

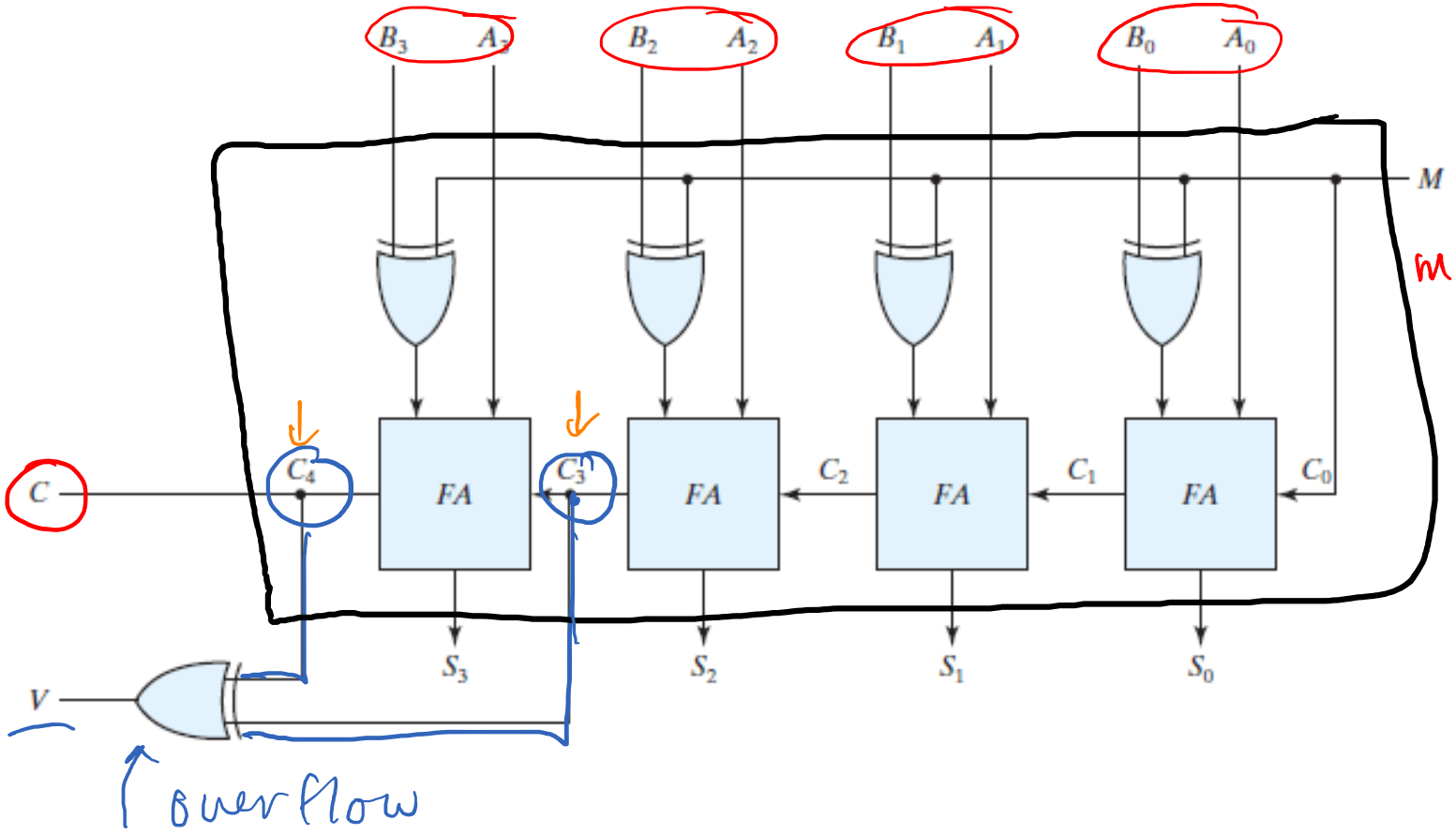
Overflow detection

- An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers.
- An overflow may occur if the two numbers added are both positive or both negative.
- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.

  If these two carries are equal, there was no overflow.

  If these two carries are not equal, an overflow has occurred.
- If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

Adder with overflow detection



M = 0 ⇒ Add

M = 1 ⇒ Subtract

C = 1 ⇒ bad math for unsigned

V = 1 ⇒ bad math for signed

WARNING: MAJOR TOPIC SHIFT

```
logic [7:0] a;
logic [7:0] b;
```

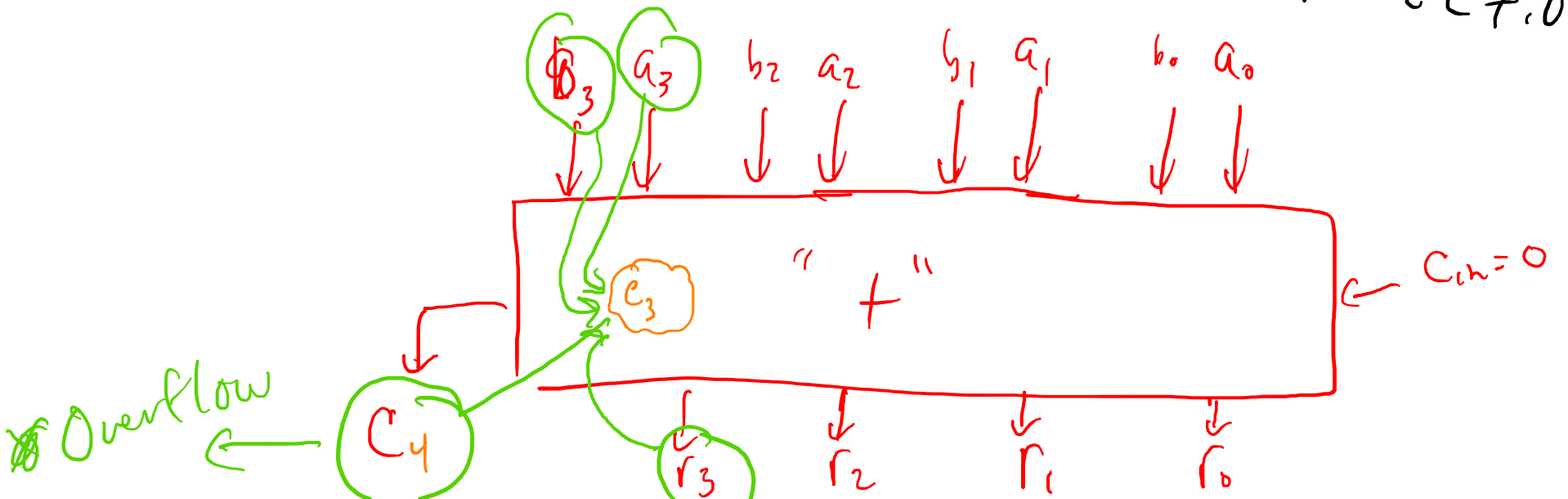
```
 1000  0000      128
+1000  0001     +129
------------    ----
 0000  0001      257
 ^
 carry
```

```
logic [7:0] z;
z = a + b;
```

⇒

```
logic [8:0] z;
z = {1'b0, a} +
    {1'b0, b};
c = z[8];
r = z[7:0];
```

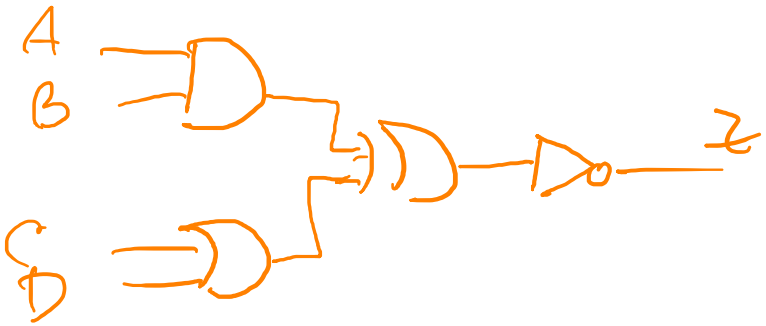
SEQUENTIAL LOGIC



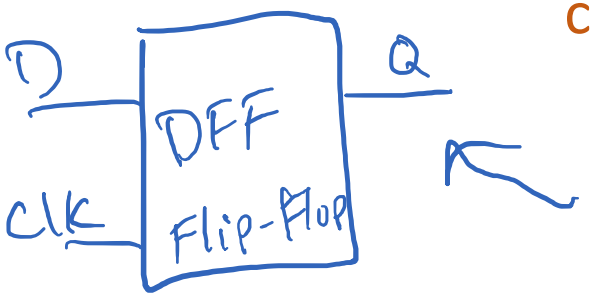
Sequential vs. Combinational

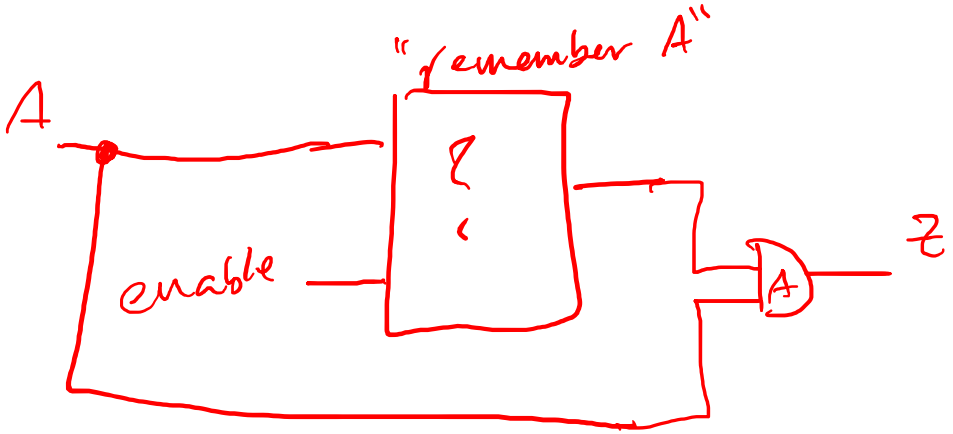
Stopped here!

- Combinational Logic
  - The output is a combination of the current inputs only

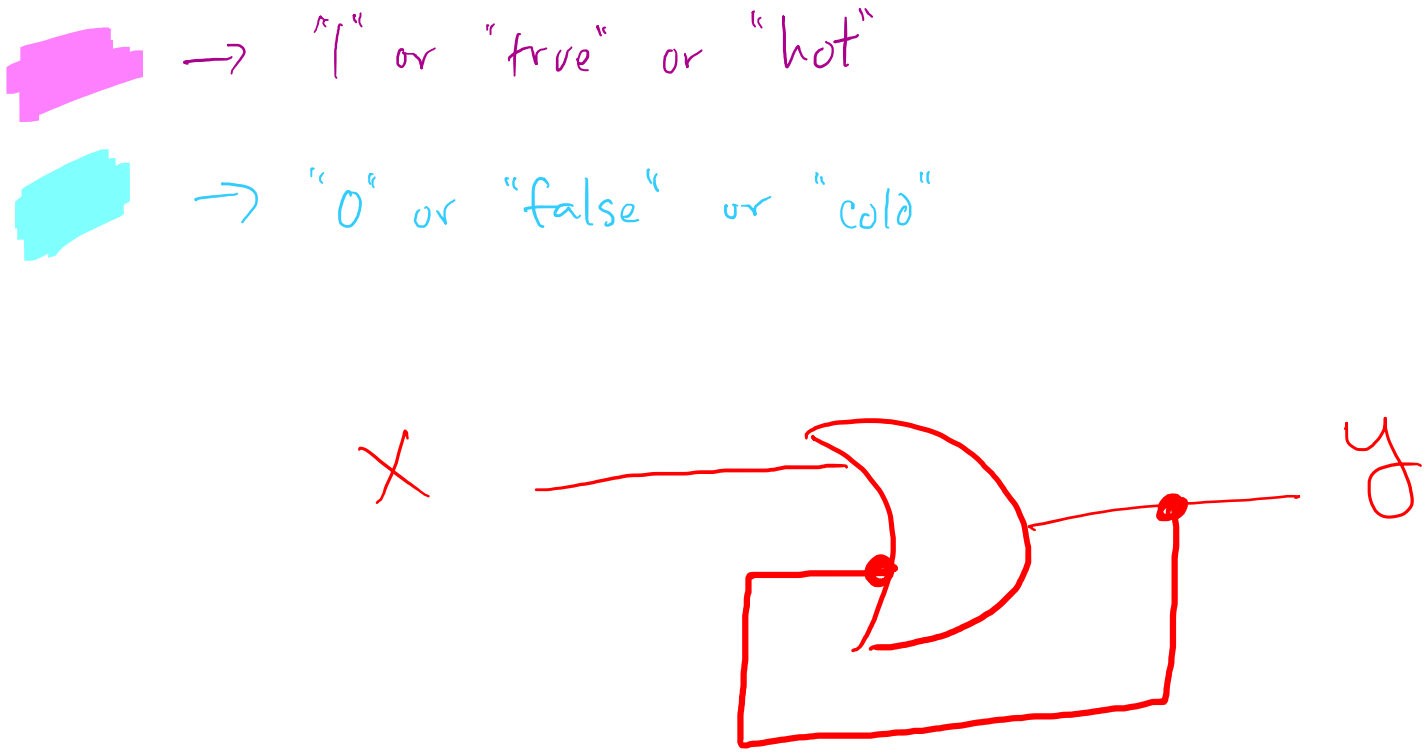


- Sequential Logic
  - The output is a combination of the current **and past** inputs

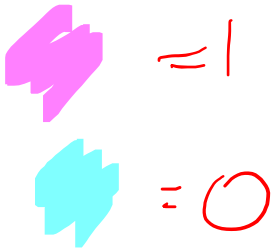


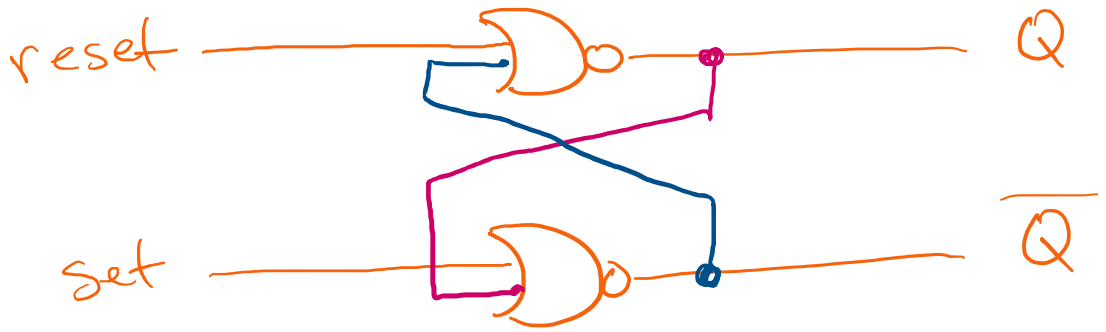


How do I store data?



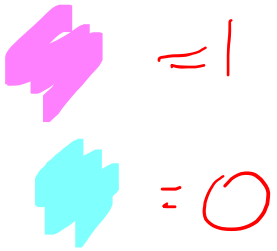
SR (Set-Reset) Latch

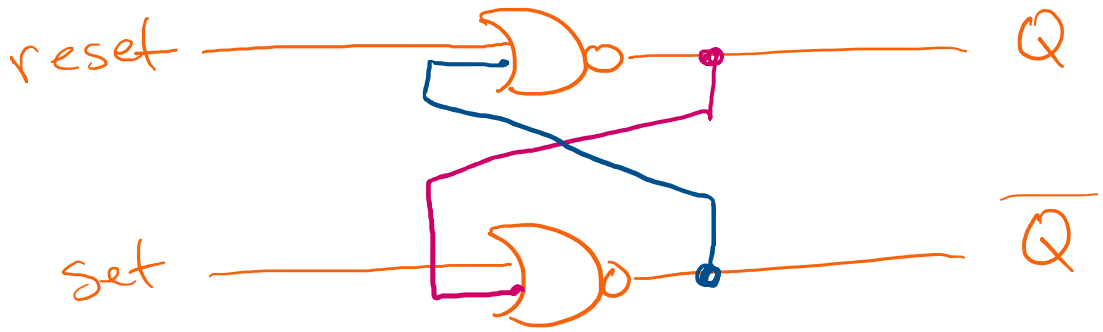




reset	set	Q	$\overline{Q}$

SR (Set-Reset) Latch



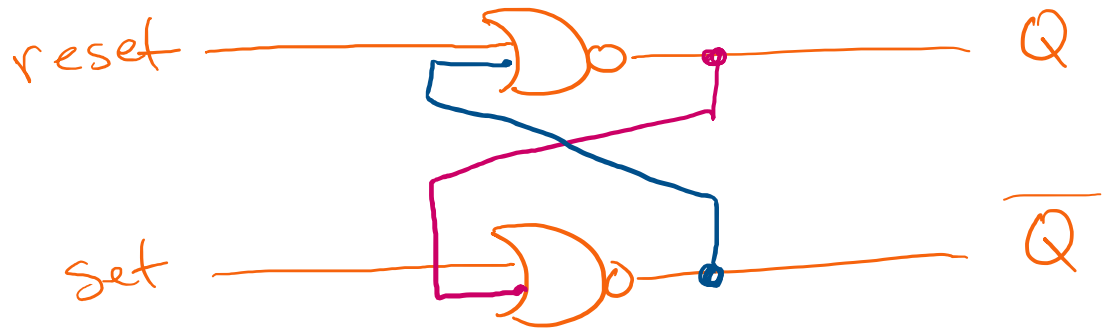


reset	set	Q	$\overline{Q}$
0	1	1	0

SR Latch

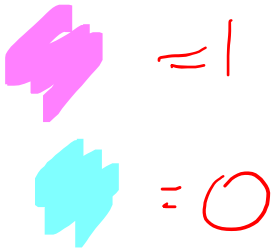
= 1

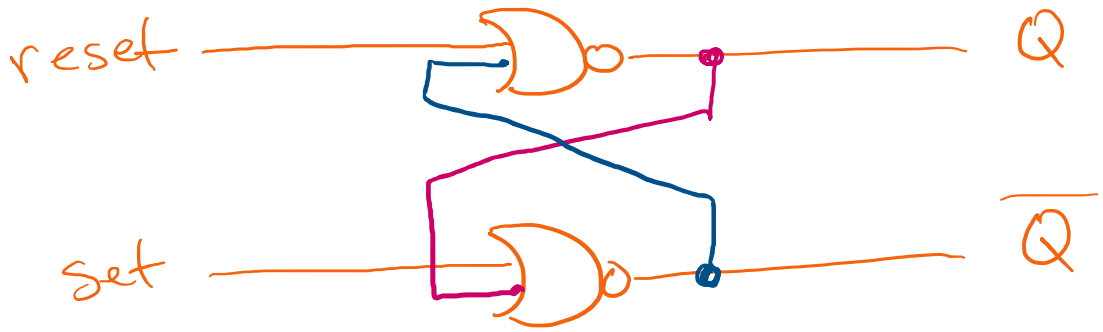
= 0



reset	set	Q	$\overline{Q}$
0	1	1	0
0	0	1	0

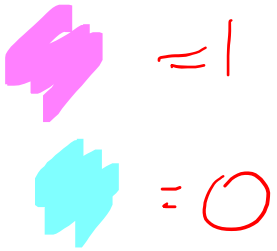
SR Latch

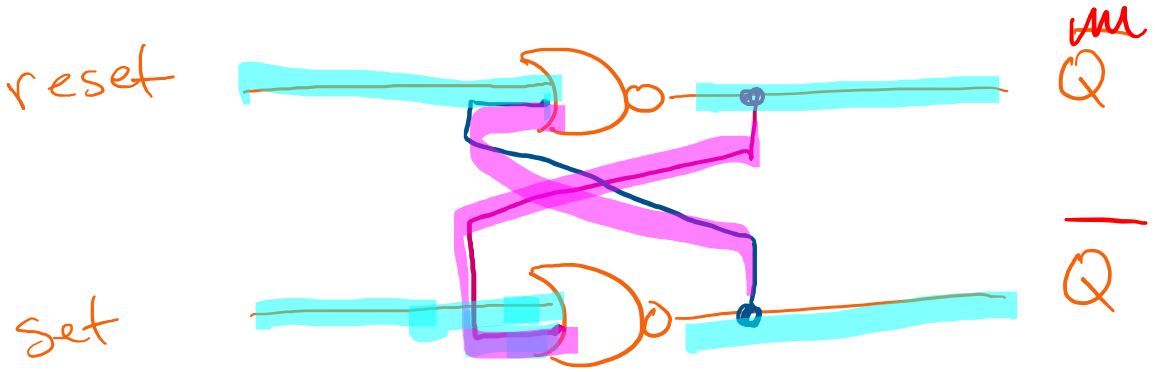




reset	set	Q	$\overline{Q}$
0	1	1	0
0	0	1	0
1	0	0	1

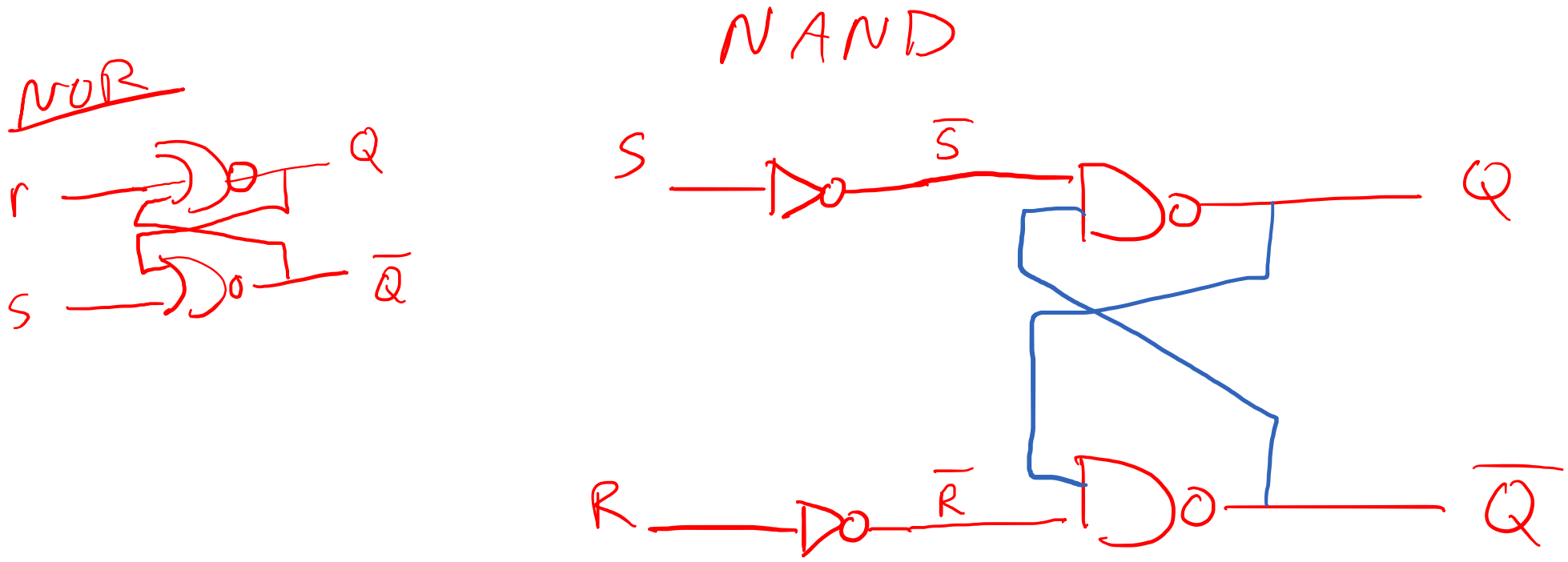
SR Latch w/ S=1 & R=1





set	reset	Q	$\overline{Q}$
1	1	0	0
0	0	0,1,0,1,0,1,0,1,...	

SR Latch w/NAND gates



→ better setup for Flip-Flops

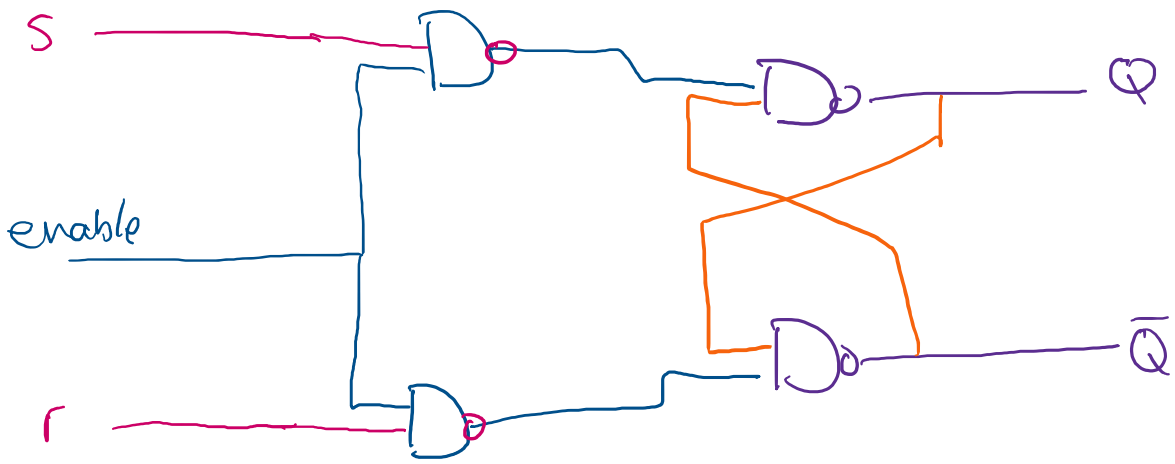
→ easier for me to draw

SR Latch w/NAND gates

SR Latch with Enable

SR Latch with Enable

Prevent changes in S & R from changing circuit output



S	R	E	Q	$\overline{Q}$
x	x	0	Q	$\overline{Q}$
1	0	1	1	0
0	1	1	0	1

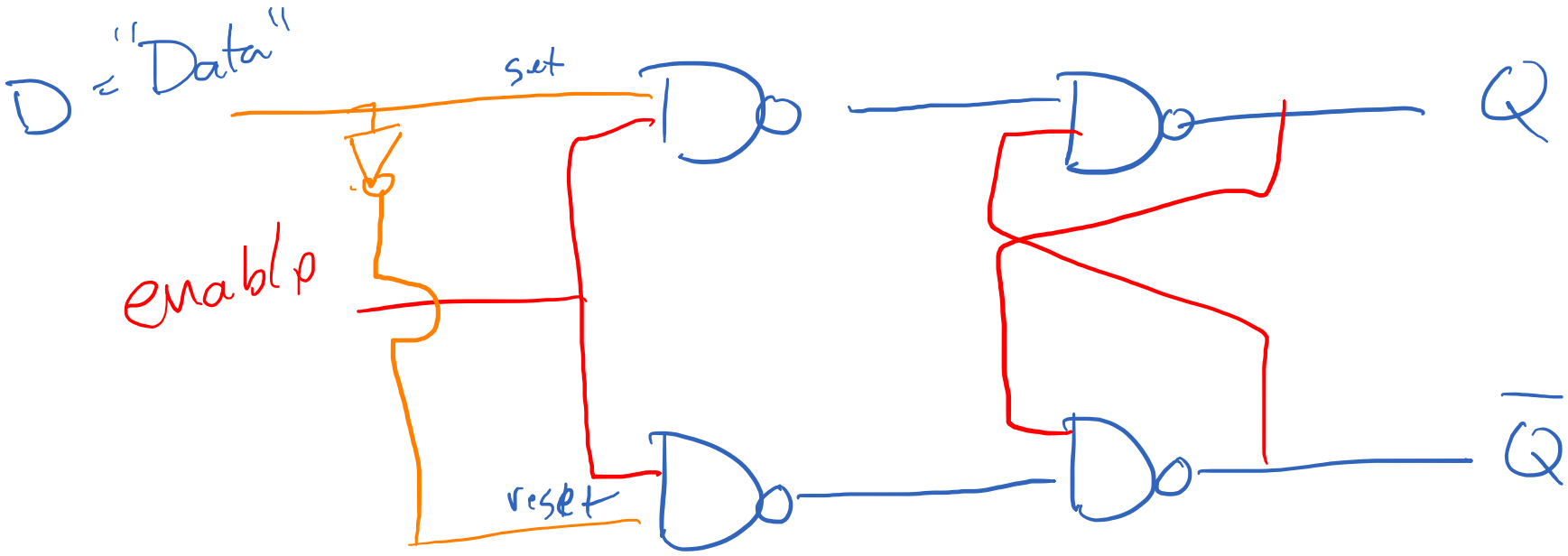
added enable bit

regular $\overline{SR}$ Latch

* assume no S=1 & R=1

D-Latch

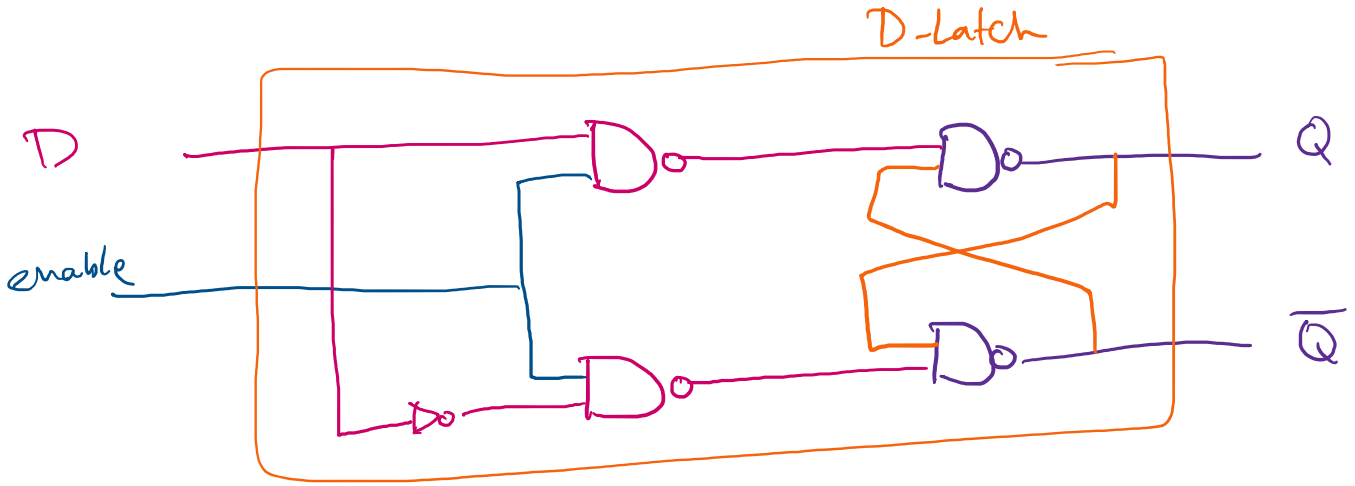
"Data" Latch



D-Latch


= 1

= 0

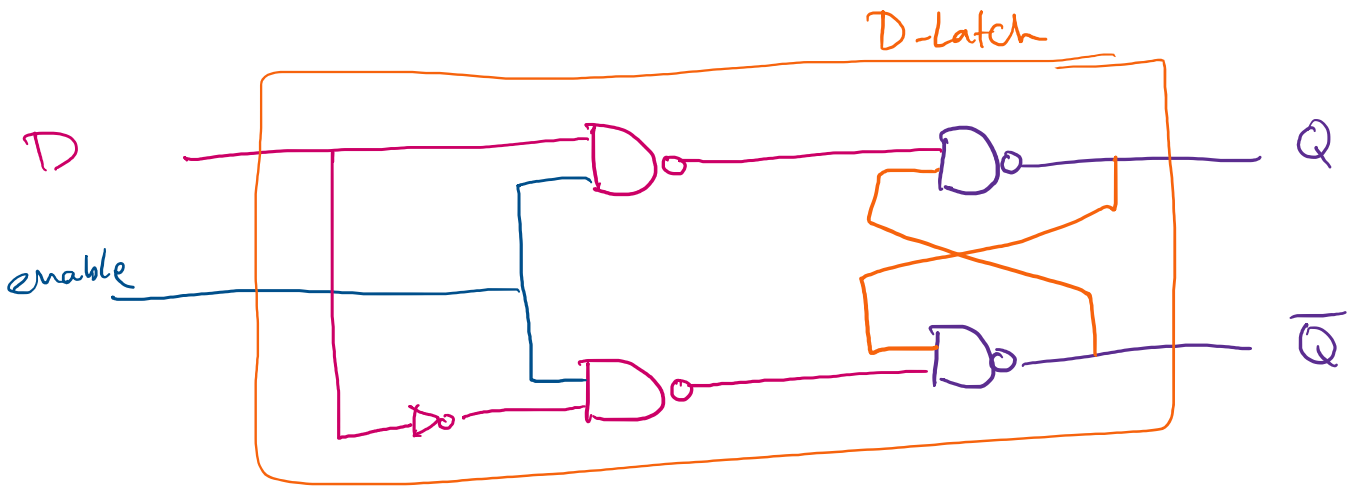


D	en	Q	$\overline{Q}$

D-Latch

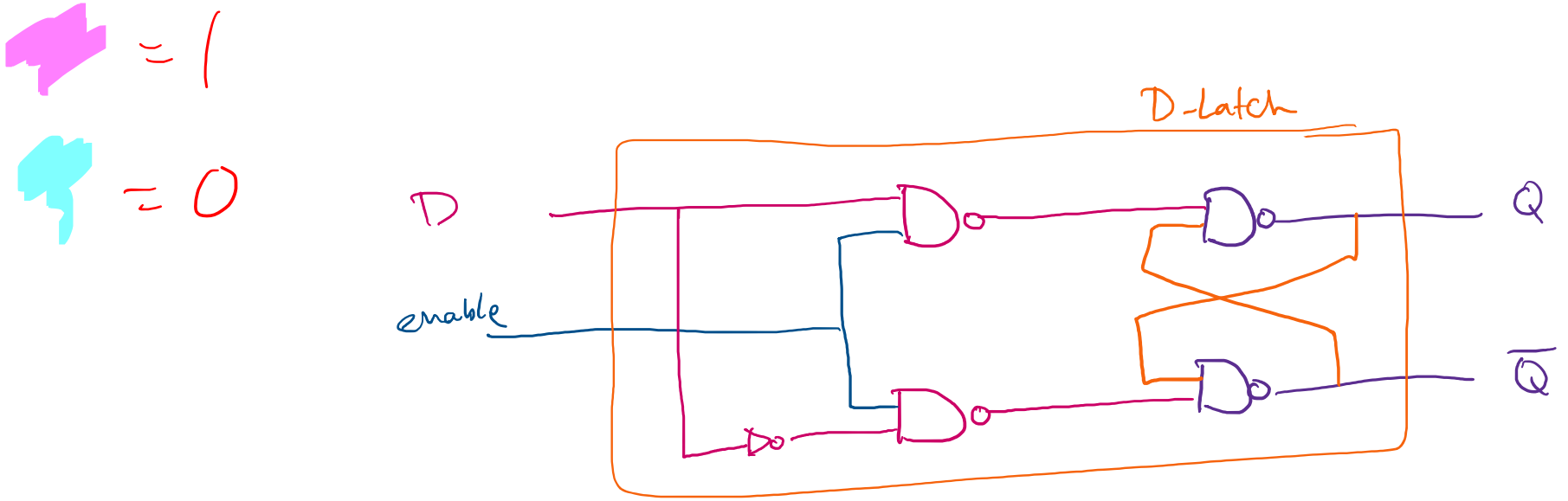
= 1

= 0



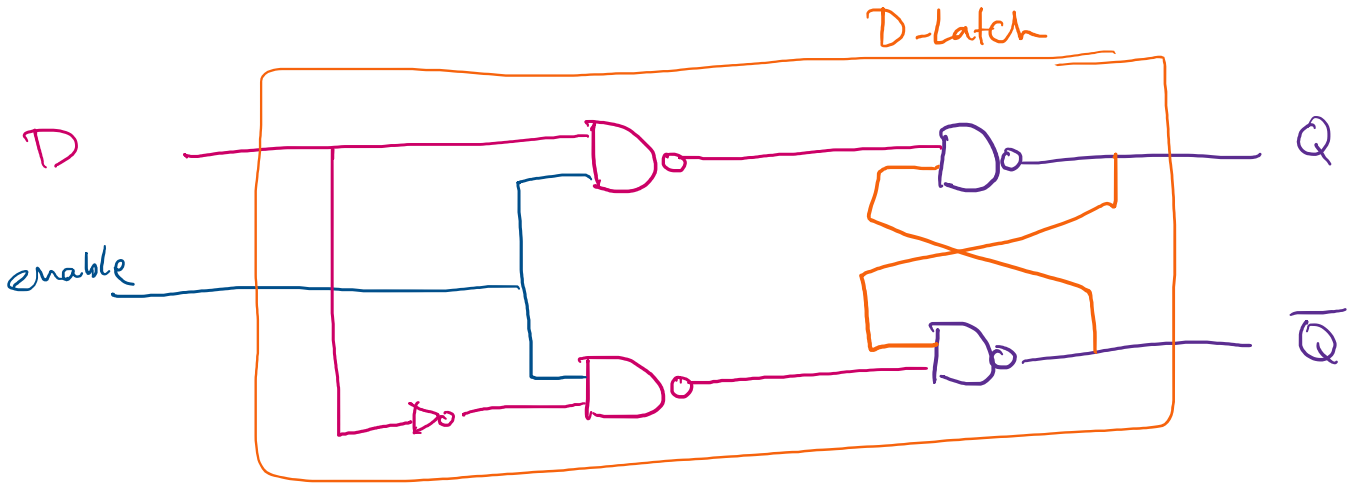
D	en	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$

D-Latch



D	en	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
1	0	Q	$\overline{Q}$

D-Latch



D	en	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
1	0	Q	$\overline{Q}$
0	1	0	1

Next Time

- D Flip-Flops
- Latches / Flops in Verilog
- `always_comb` and `always_ff` blocks