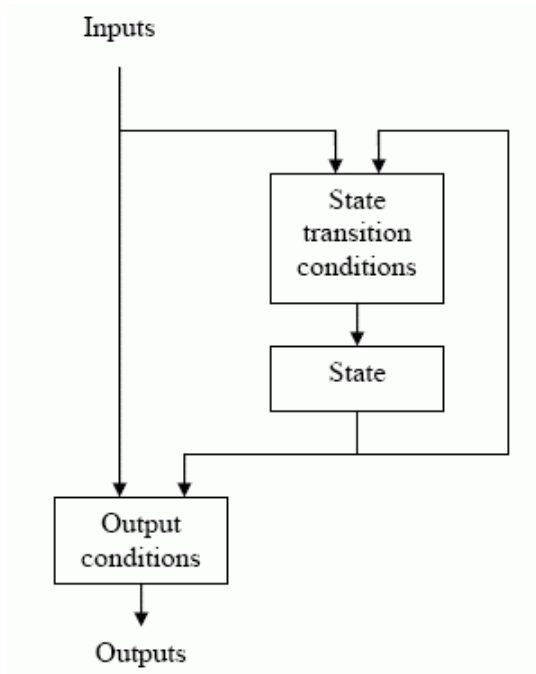


Lecture #15: State 2 and Blocks



2006-07-24

Andy Carle



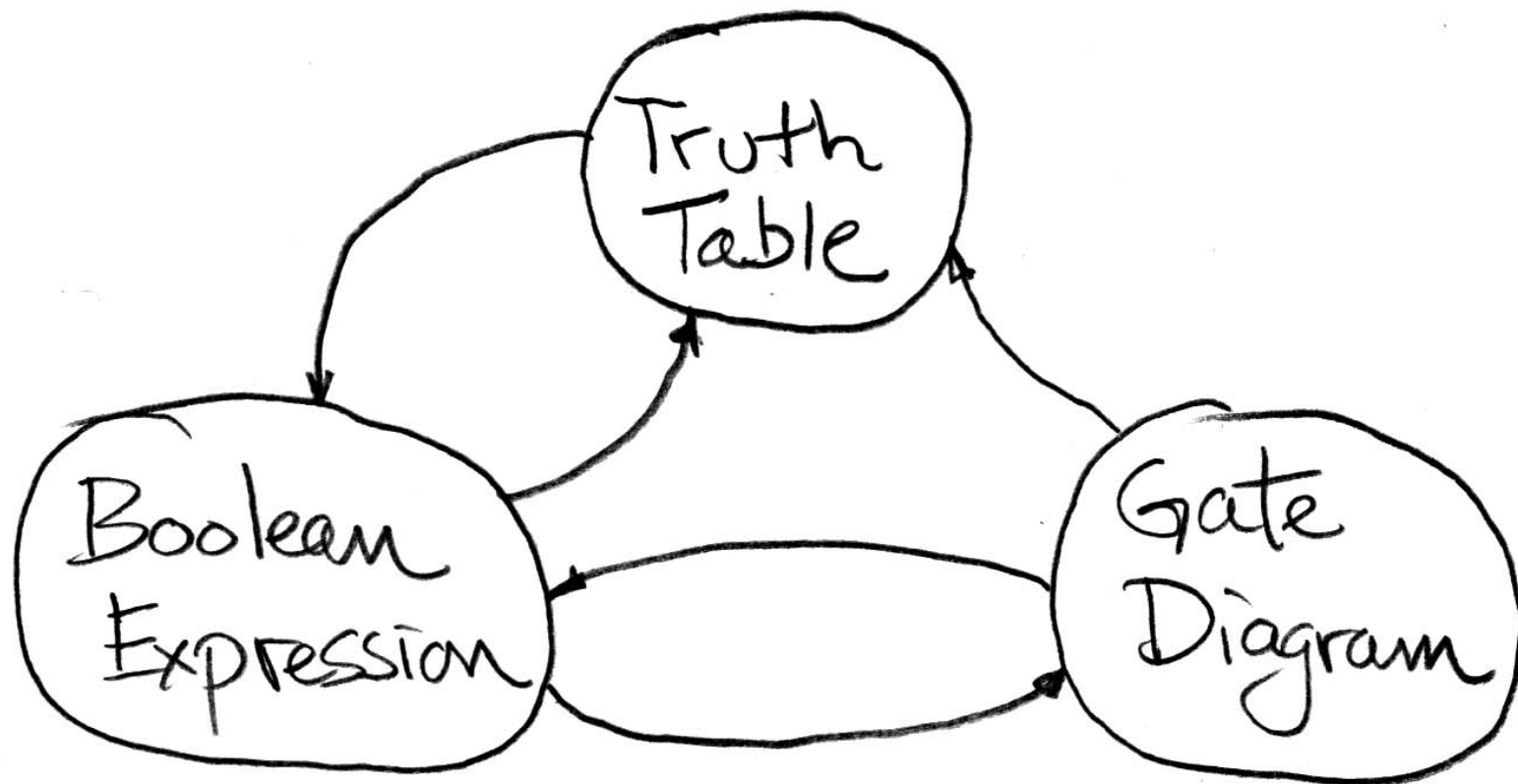
Outline

- **Review**
- **Clocks**
- **FSMs**
- **Combinational Logic Blocks**

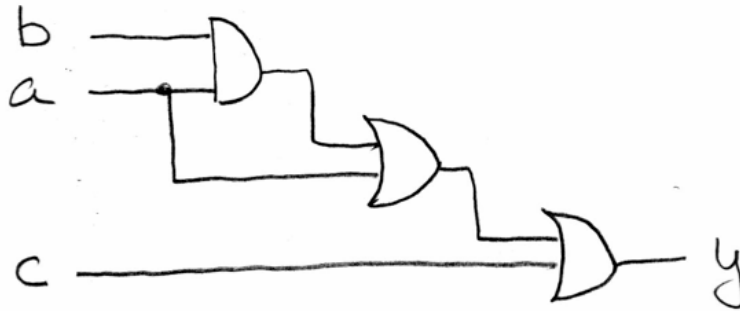


Review (1/3)

- Use this table and techniques we learned to transform from 1 to another



(2/3): Circuit & Algebraic Simplification



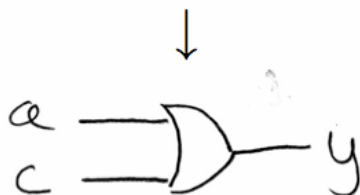
original circuit

$$y = ((ab) + a) + c$$

equation derived from original circuit

$$\begin{aligned} &\downarrow \\ &= ab + a + c \\ &\downarrow \\ &= a(b + 1) + c \\ &= a(1) + c \\ &= a + c \end{aligned}$$

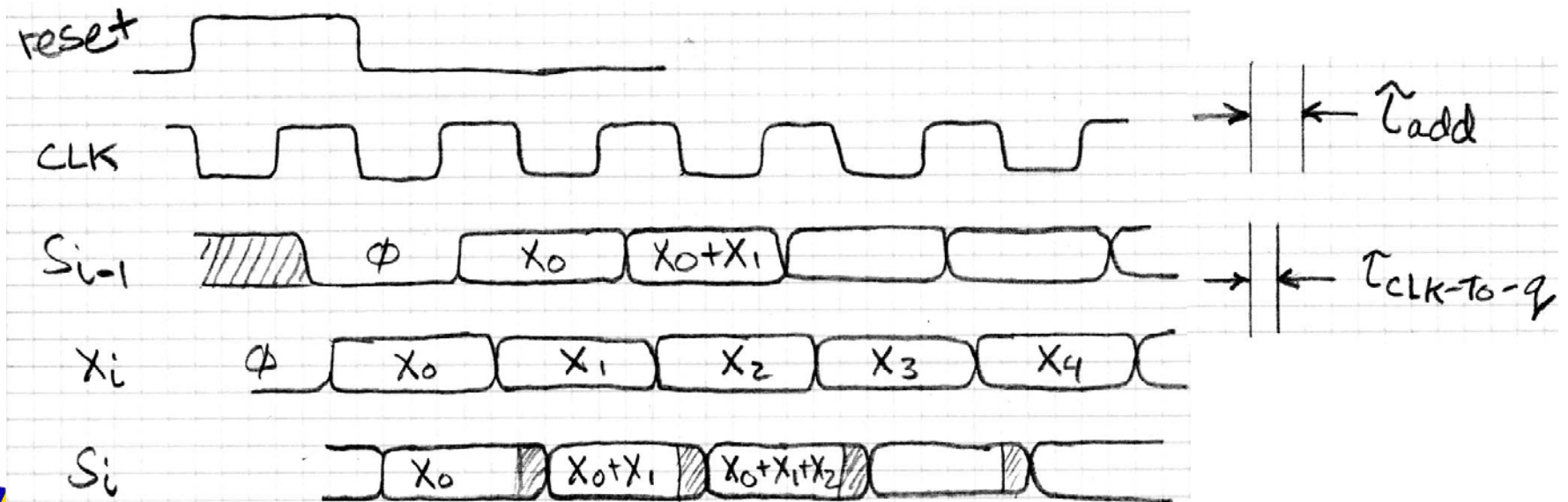
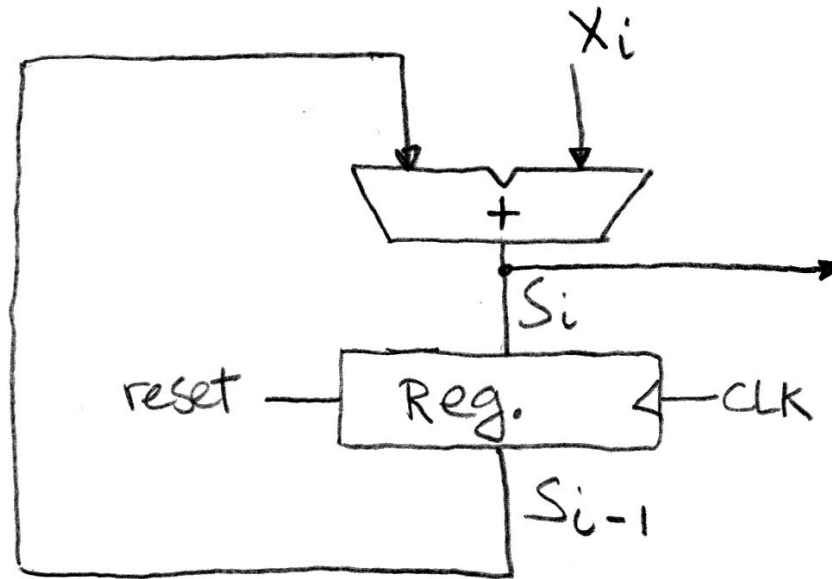
algebraic simplification



simplified circuit



Review (3/3)



Clocks

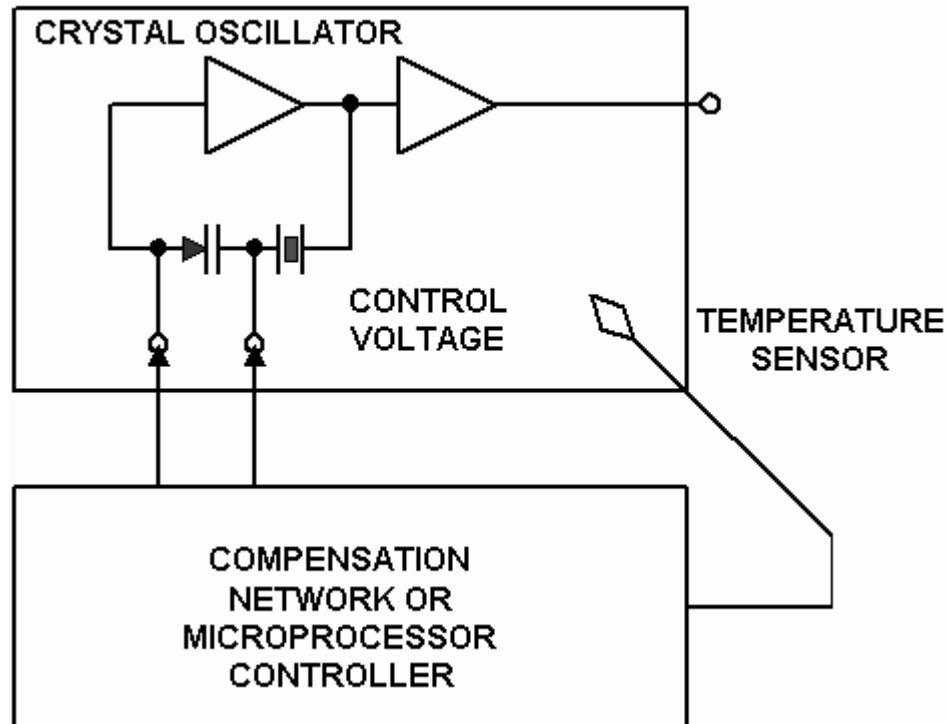
- **Need a regular oscillator:**



- **Wire up three inverters in feedback?...**
 - **Not stable enough**
 - **1->0 and 0->1 transitions not symmetric.**
- **Solution: Base oscillation on a natural resonance. But of what?**

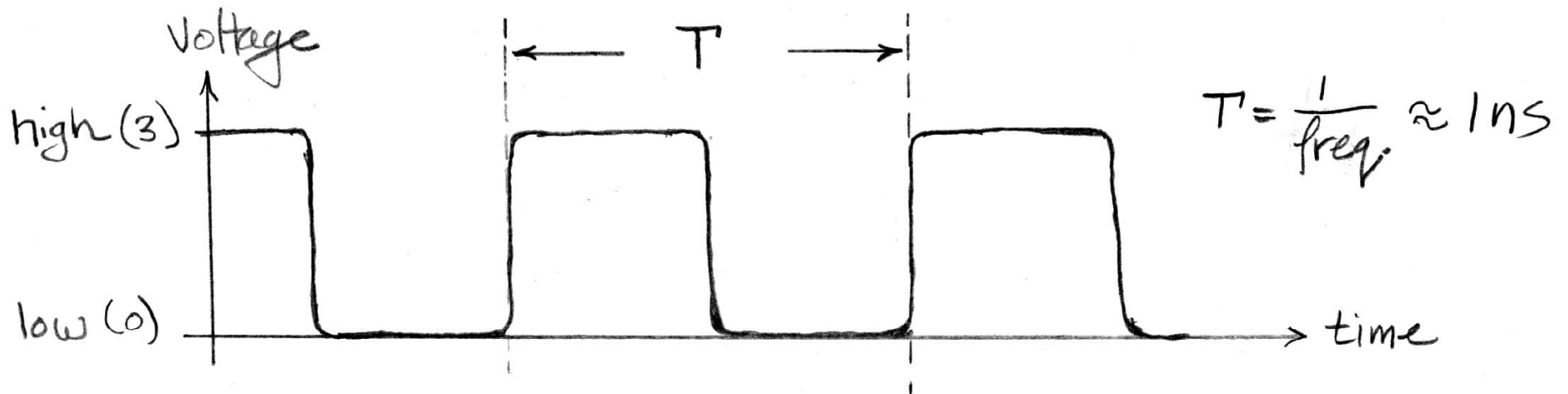


Clocks



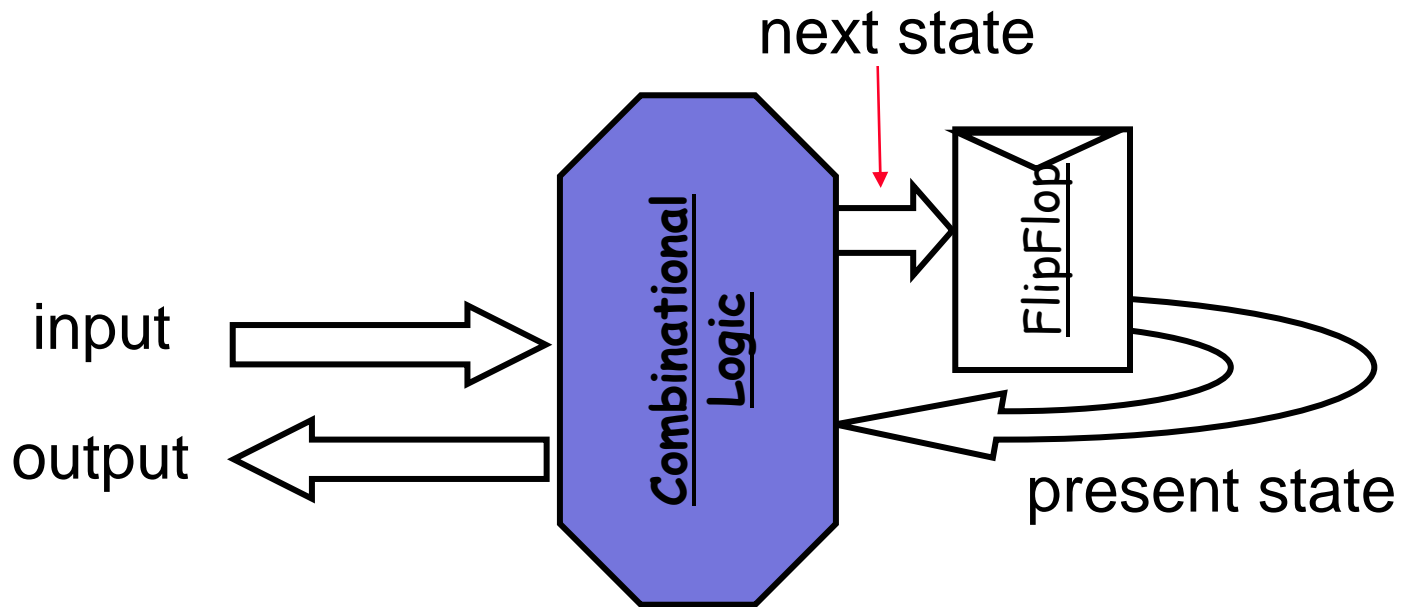
- **Controller puts AC across crystal:**
 - At anything but resonant freqs → destructive interference
 - Resonant freq → **CONSTRUCTIVE!**

Signals and Waveforms: Clocks



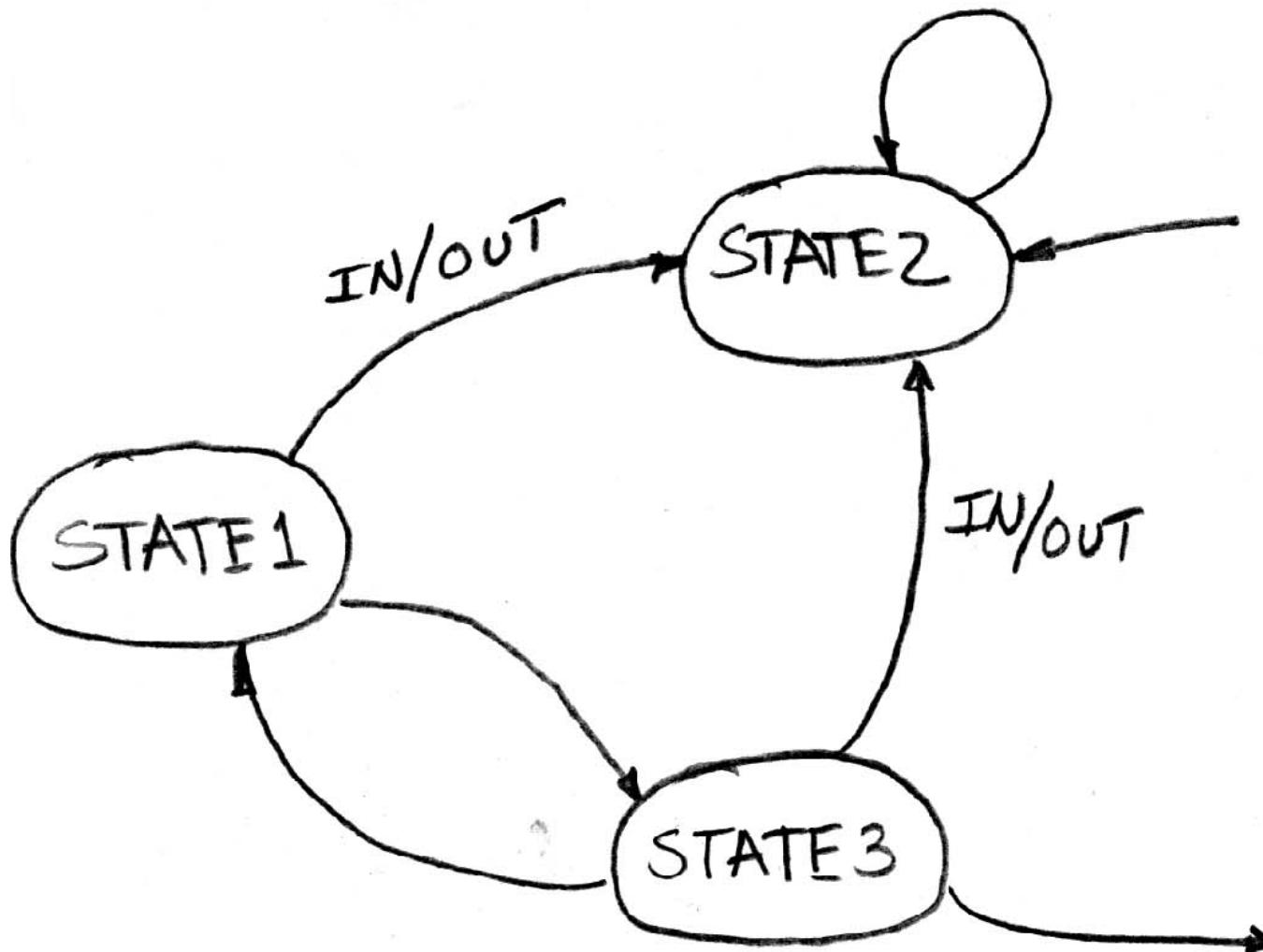
FSMs

- **With state elements, we can build circuits whose output is a function of **inputs and current state**.**

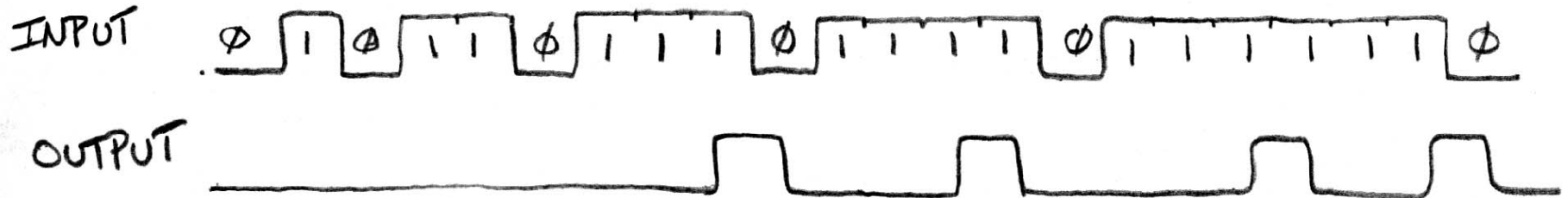


- **State transitions will occur on clock edges.**

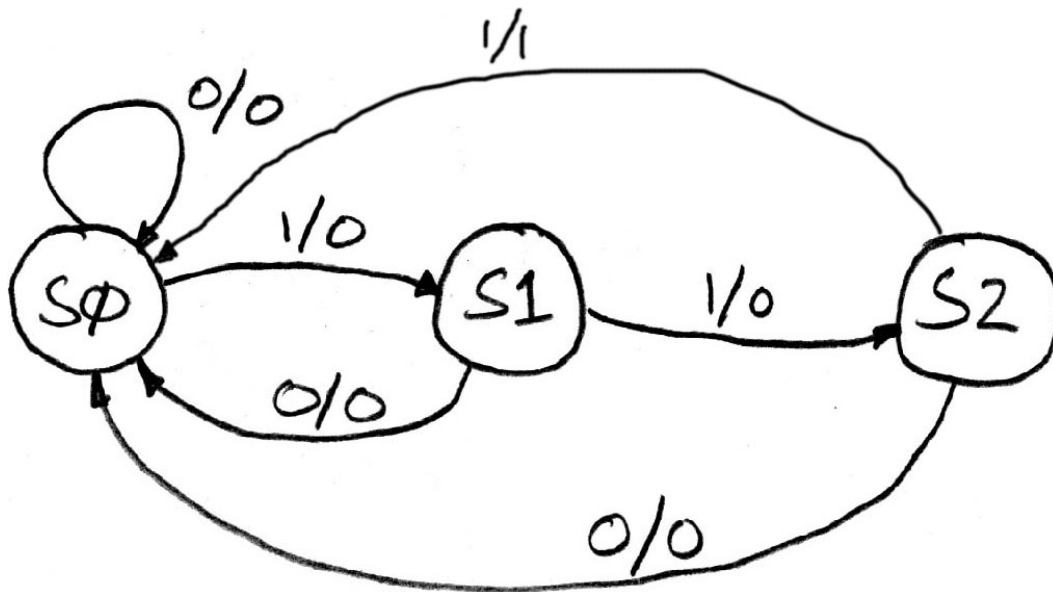
Finite State Machines Introduction



Finite State Machine Example: 3 ones...



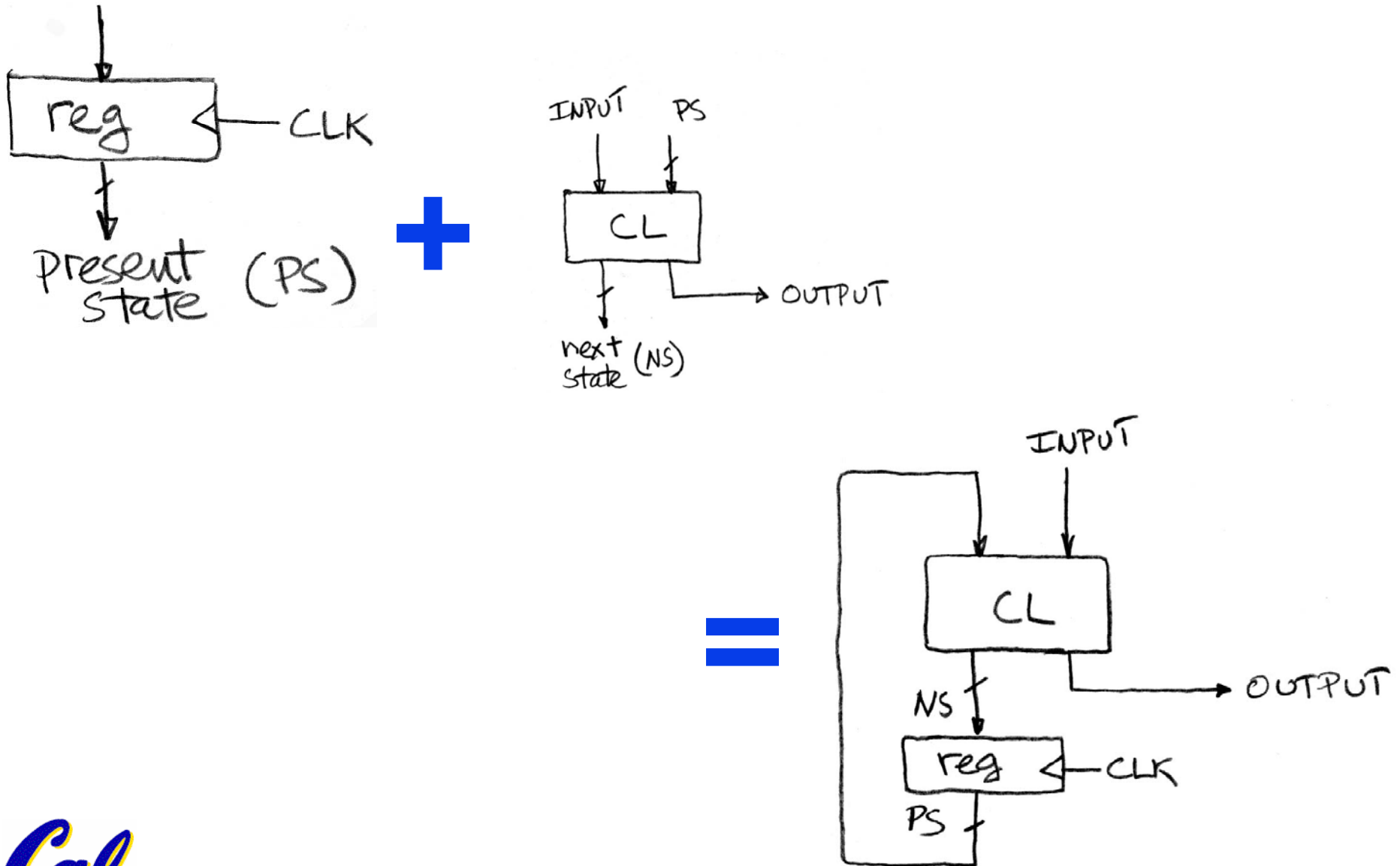
Draw the FSM...



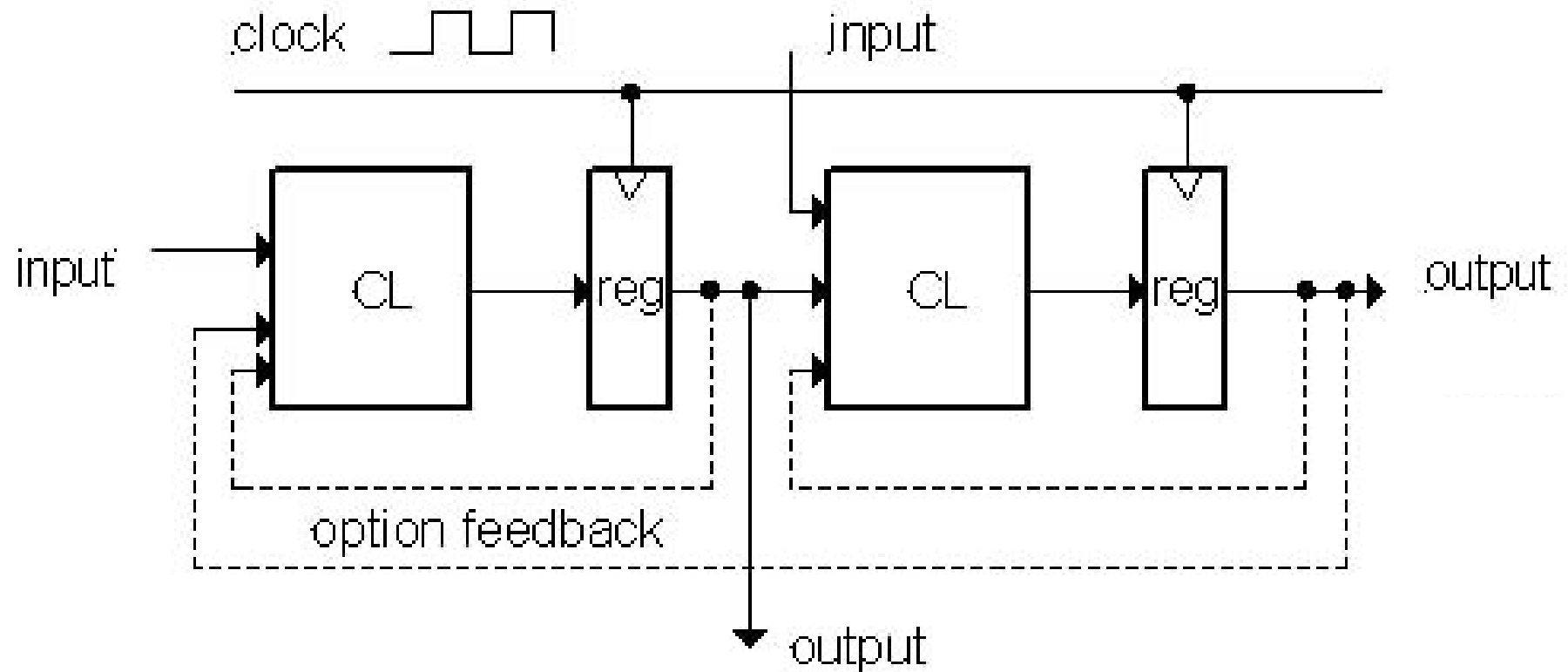
PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1



Hardware Implementation of FSM



General Model for Synchronous Systems



Peer Instruction 1

- **Two bit counter:**
 - **4 States: 0, 1, 2, 3**
 - **When input c is high, go to next state**
 - (3->0)
 - **When input is low, don't change state**
 - **On the transition from state 3 to state 0, output a 1. At all other times, output 0.**

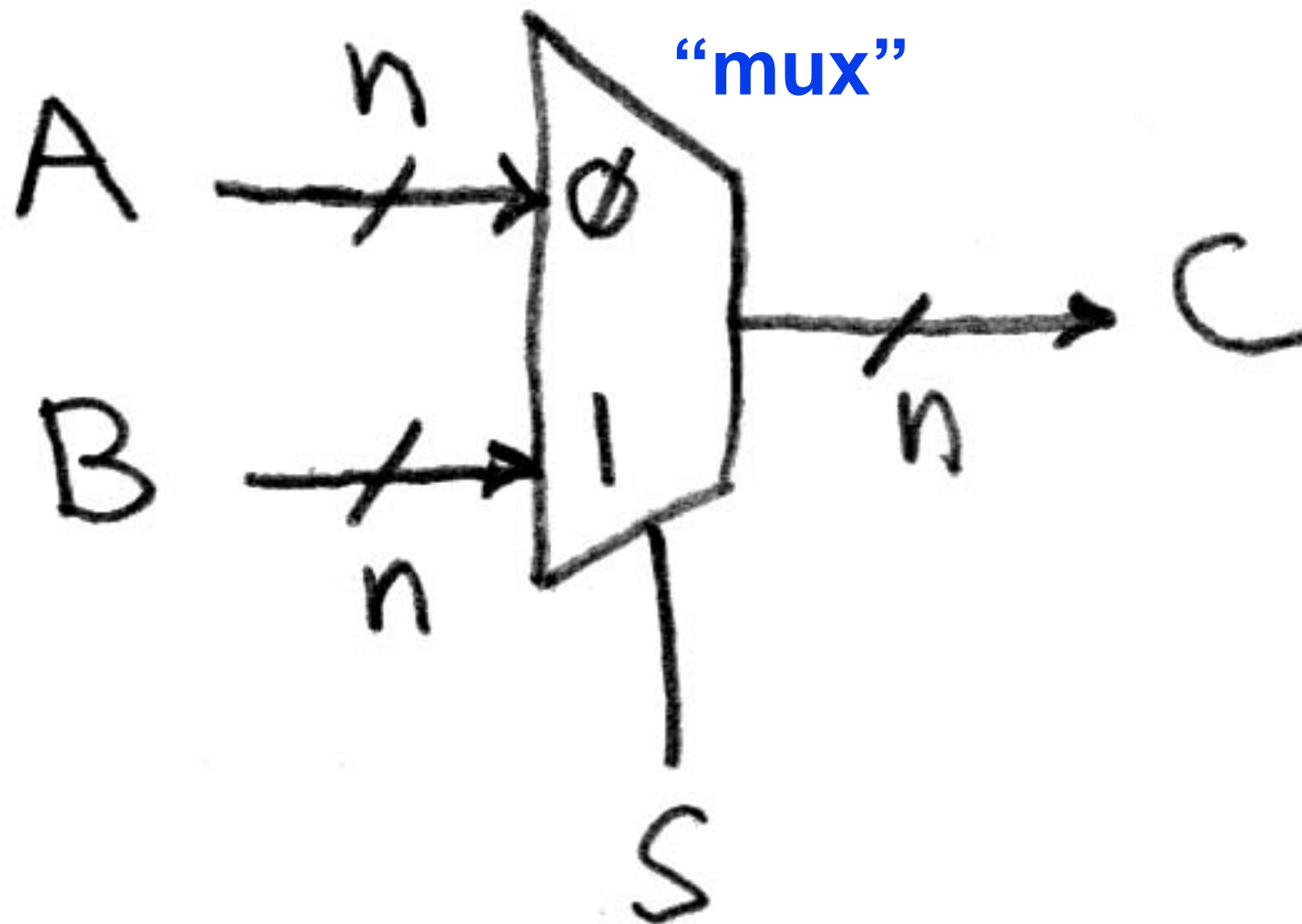


CL Blocks

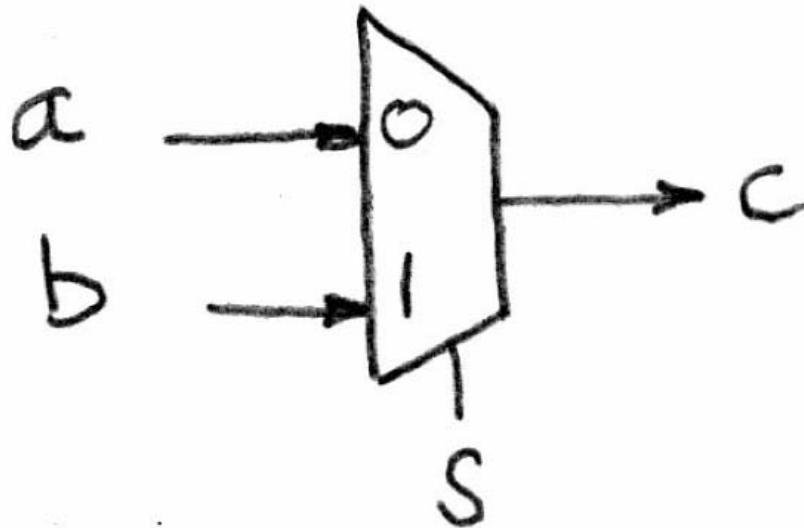
- **Let's use our skills to build some CL blocks:**
 - **Multiplexer (mux)**
 - **Adder**
 - **ALU**



Data Multiplexor (here 2-to-1, n-bit-wide)



N instances of 1-bit-wide mux



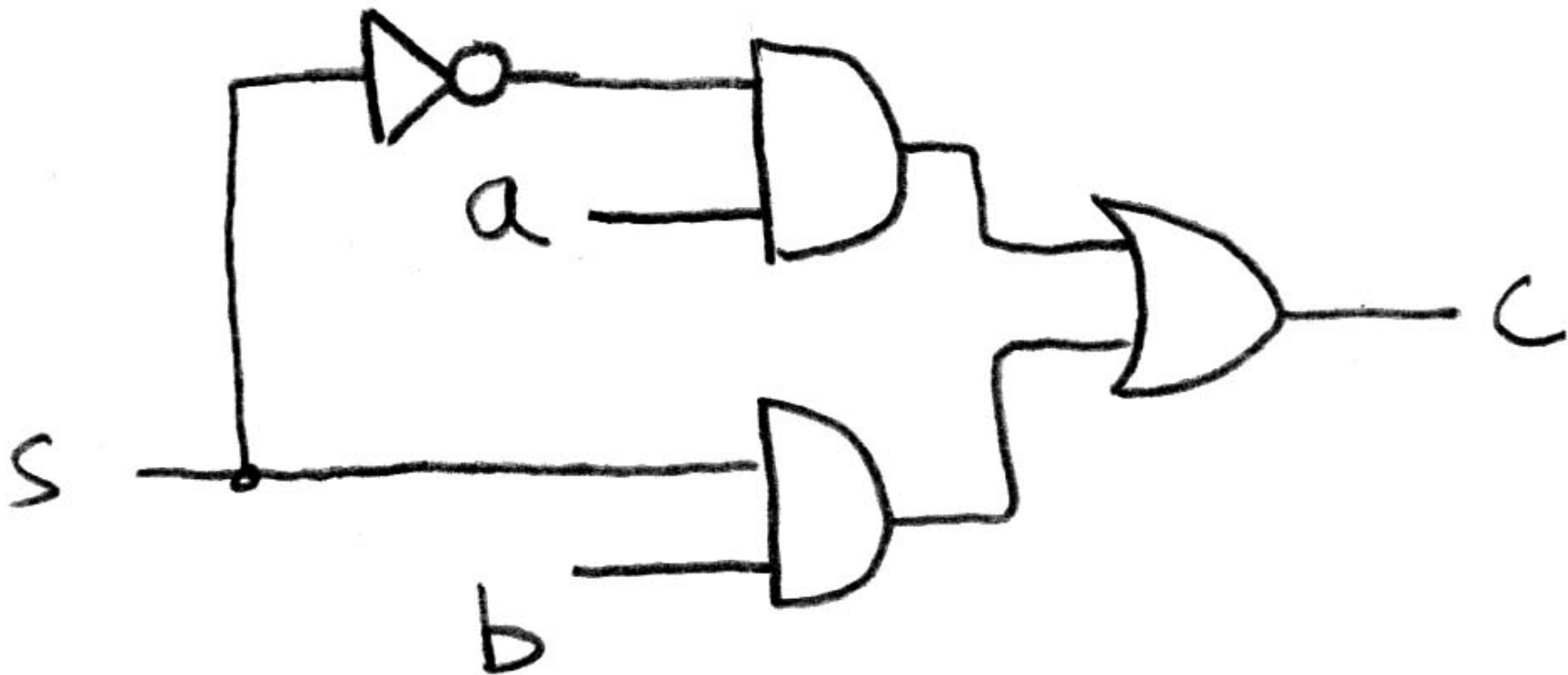
s	ab	c
0	00	0
0	01	0
0	10	1
0	11	1
1	00	0
1	01	1
1	10	0
1	11	1

$$\begin{aligned}c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\ &= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\ &= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\ &= \bar{s}(a(1) + s((1)b) \\ &= \bar{s}a + sb\end{aligned}$$

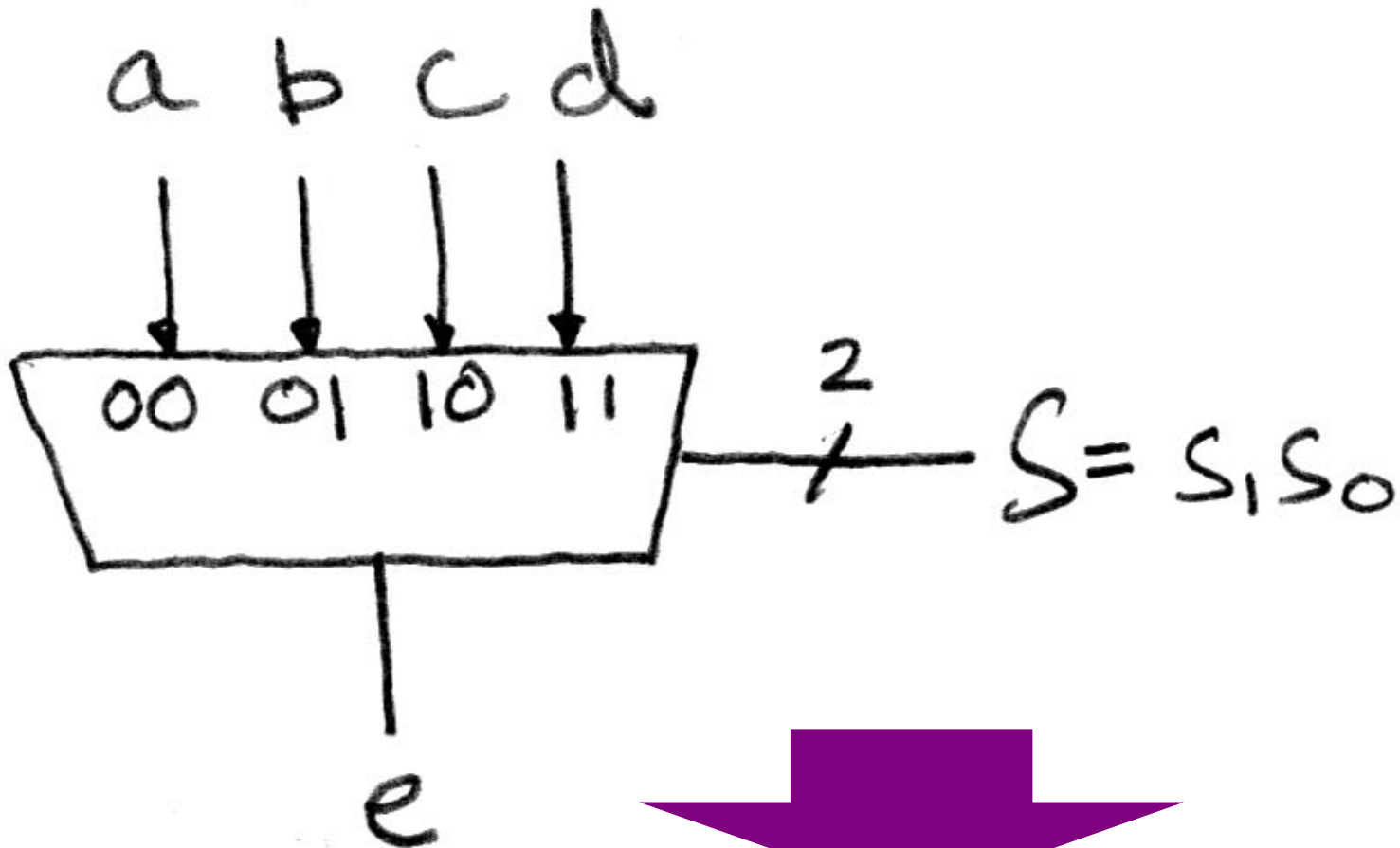


How do we build a 1-bit-wide mux?

$$\bar{s}a + sb$$

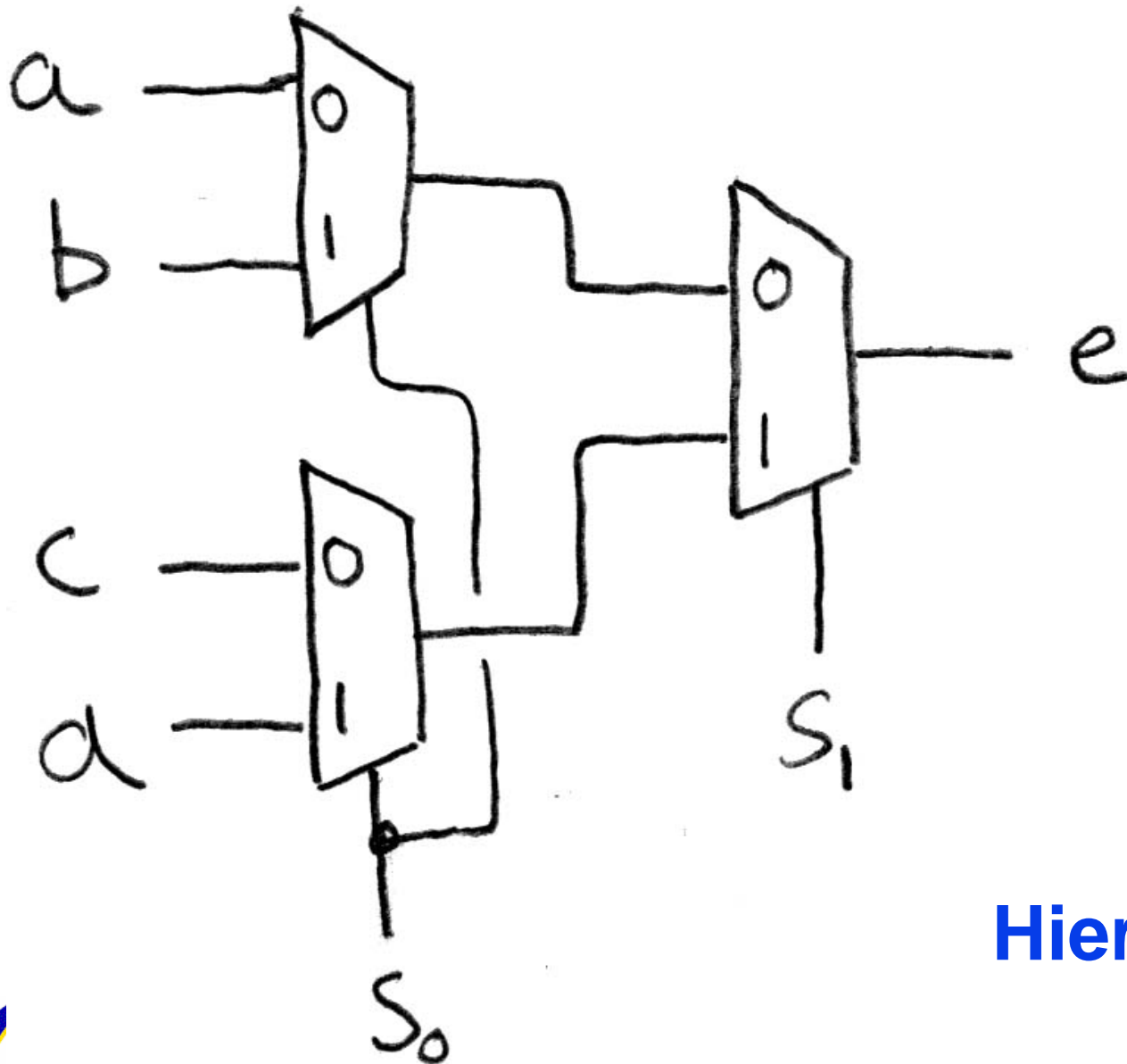


4-to-1 Multiplexor?



$$e = \overline{s_1}\overline{s_0}a + \overline{s_1}s_0b + s_1\overline{s_0}c + s_1s_0d$$

An Alternative Approach

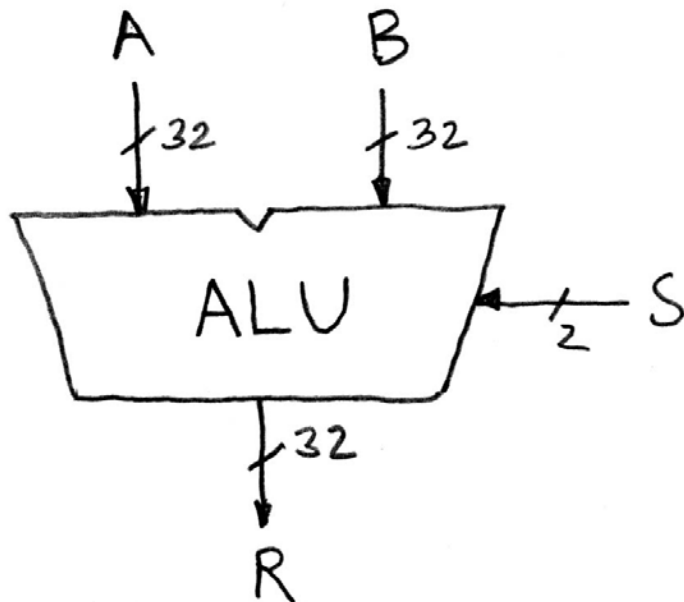


Hierarchically!



Arithmetic and Logic Unit

- Most processors contain a logic block called “Arithmetic/Logic Unit” (ALU)
- We’ll show you an easy one that does ADD, SUB, bitwise AND, bitwise OR



when $S=00$, $R=A+B$

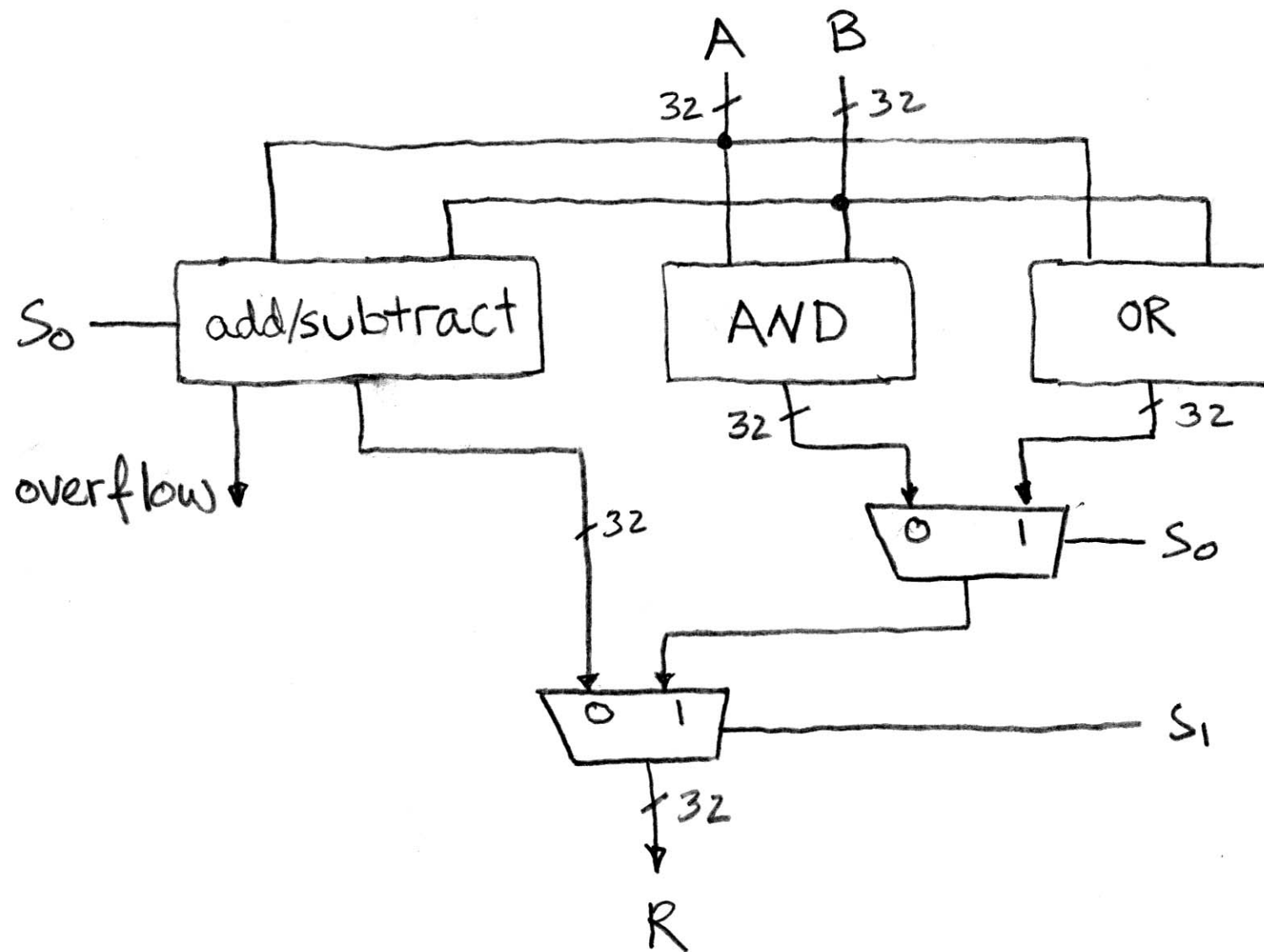
when $S=01$, $R=A-B$

when $S=10$, $R=A \text{ AND } B$

when $S=11$, $R=A \text{ OR } B$



Our simple ALU

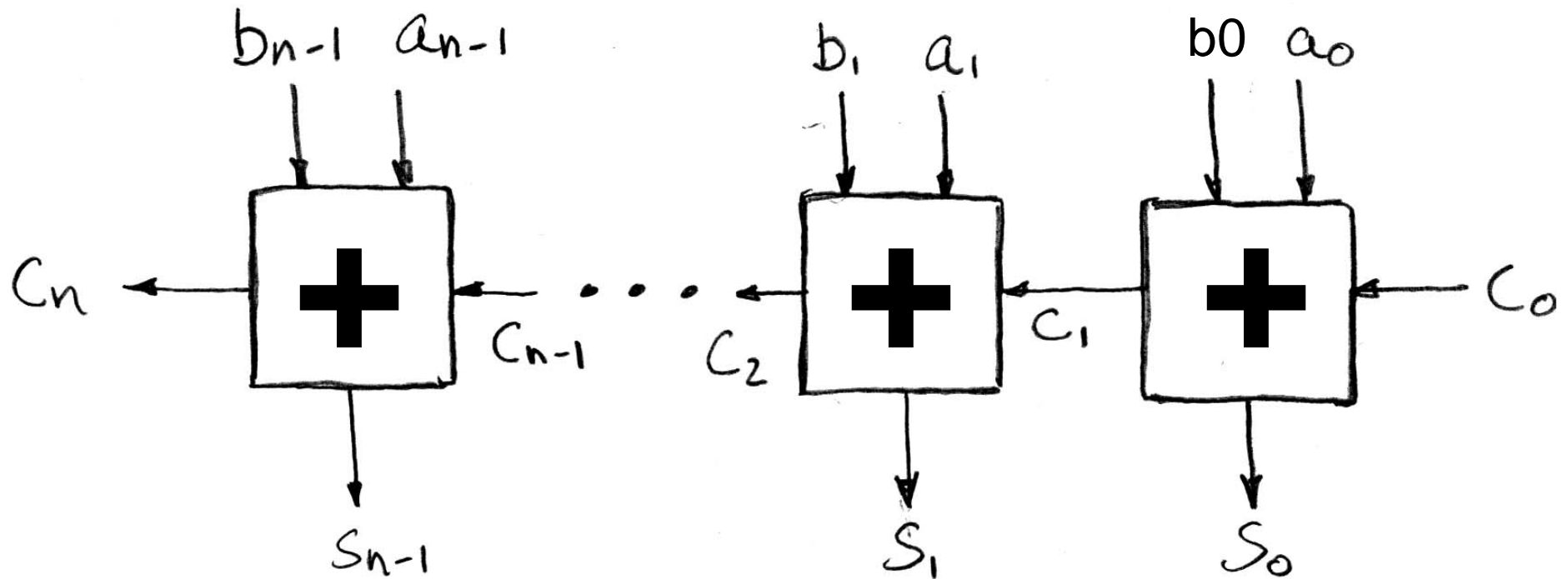


Adder/Subtractor Design -- how?

- **Truth-table, then determine canonical form, then minimize and implement as we've seen before**
- **Look at breaking the problem down into smaller pieces that we can cascade or hierarchically layer**



N 1-bit adders \Rightarrow 1 N-bit adder



Adder/Subtractor – One-bit adder LSB...

$$\begin{array}{rcccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline s_3 & s_2 & s_1 & s_0 & \end{array}$$

a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$s_0 =$$

$$c_1 =$$



Adder/Subtractor – One-bit adder (1/2)...

+	a_3	a_2	a_1	a_0
	b_3	b_2	b_1	b_0
	s_3	s_2	s_1	s_0

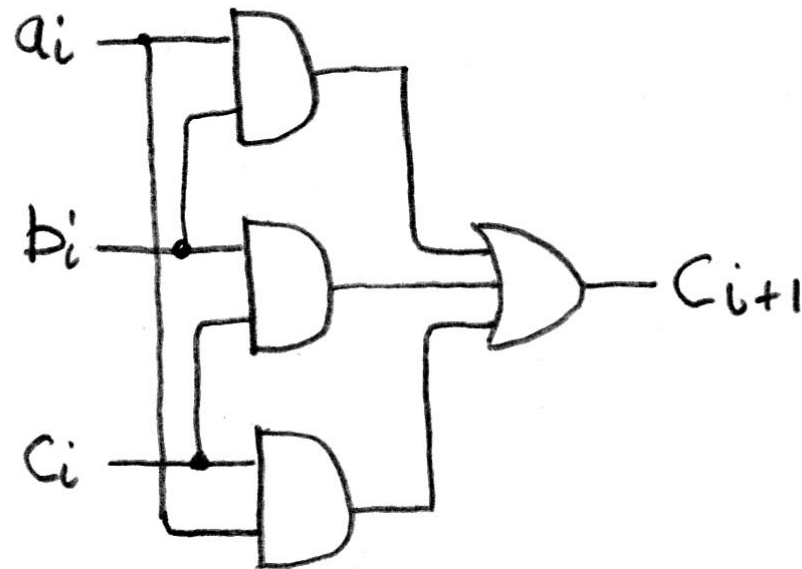
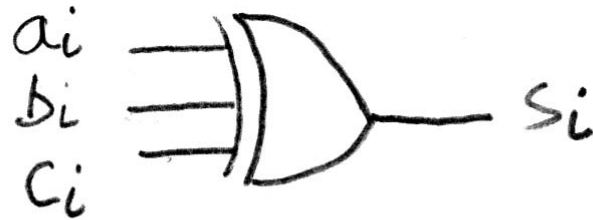
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i =$$

$$c_{i+1} =$$



Adder/Subtractor – One-bit adder (2/2)...

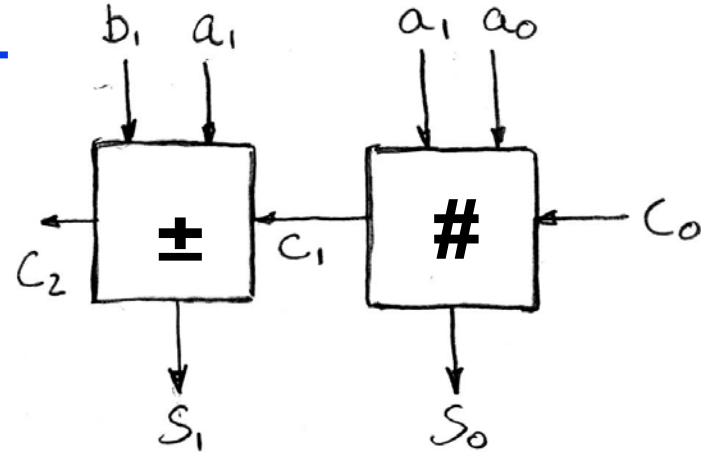


$$s_i = \text{XOR}(a_i, b_i, c_i)$$
$$c_{i+1} = \text{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$

What about overflow?

- Consider a 2-bit signed # & overflow:

- 10 = -2 + -2 or -1
- 11 = -1 + -2 only
- 00 = 0 NOTHING!
- 01 = 1 + 1 only



- Highest adder

- $C_1 = \text{Carry-in} = C_{in}$, $C_2 = \text{Carry-out} = C_{out}$
- No C_{out} or $C_{in} \Rightarrow$ NO overflow!

What op?

- C_{in} , and $C_{out} \Rightarrow$ NO overflow!

• C_{in} , but no $C_{out} \Rightarrow A, B \text{ both } > 0$, overflow!

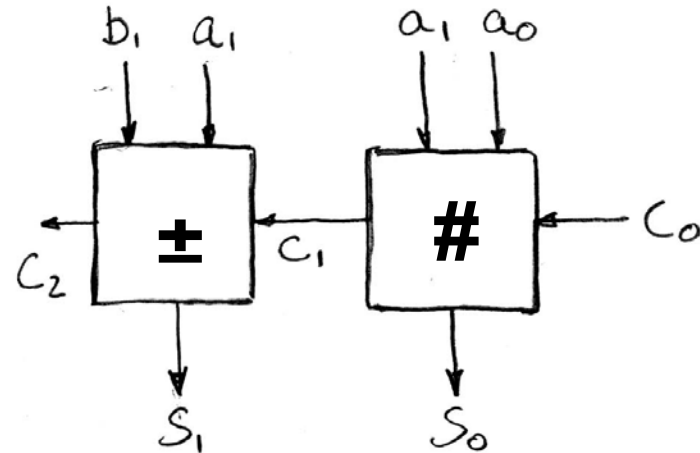
• C_{out} , but no $C_{in} \Rightarrow A, B \text{ both } < 0$, overflow!



What about overflow?

- Consider a 2-bit signed # & overflow:

10 = -2
11 = -1
00 = 0
01 = 1



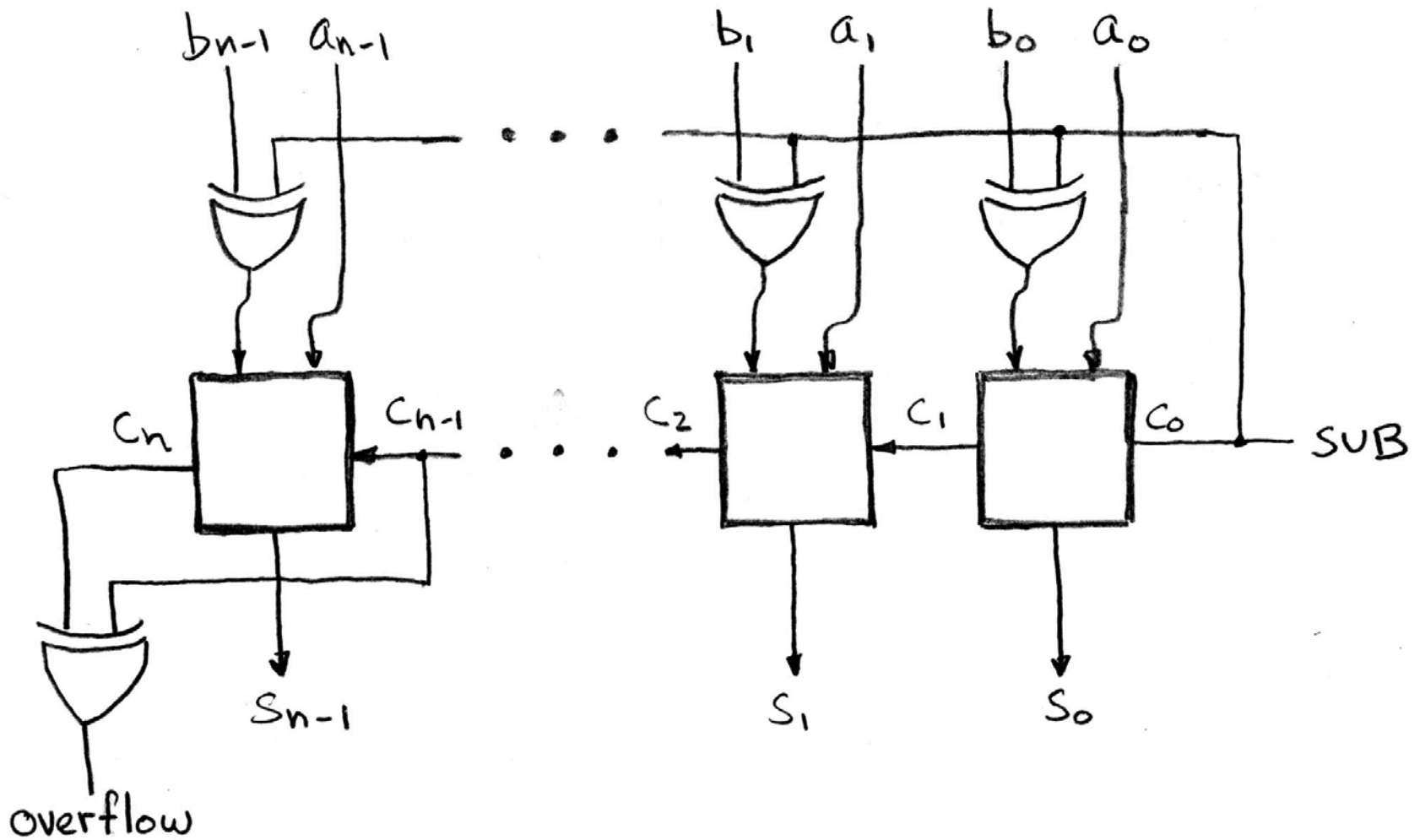
- Overflows when...

- C_{in} , but no $C_{out} \Rightarrow A, B$ both > 0 , overflow!
- C_{out} , but no $C_{in} \Rightarrow A, B$ both < 0 , overflow!

$$\text{overflow} = c_n \text{ XOR } c_{n-1}$$



Extremely Clever Subtractor



Peer Instruction

- A. Truth table for mux with 4 control signals has 2^4 rows**
- B. We could cascade N 1-bit shifters to make 1 N-bit shifter for sll, srl**
- C. If 1-bit adder delay is T, the N-bit adder delay would also be T**



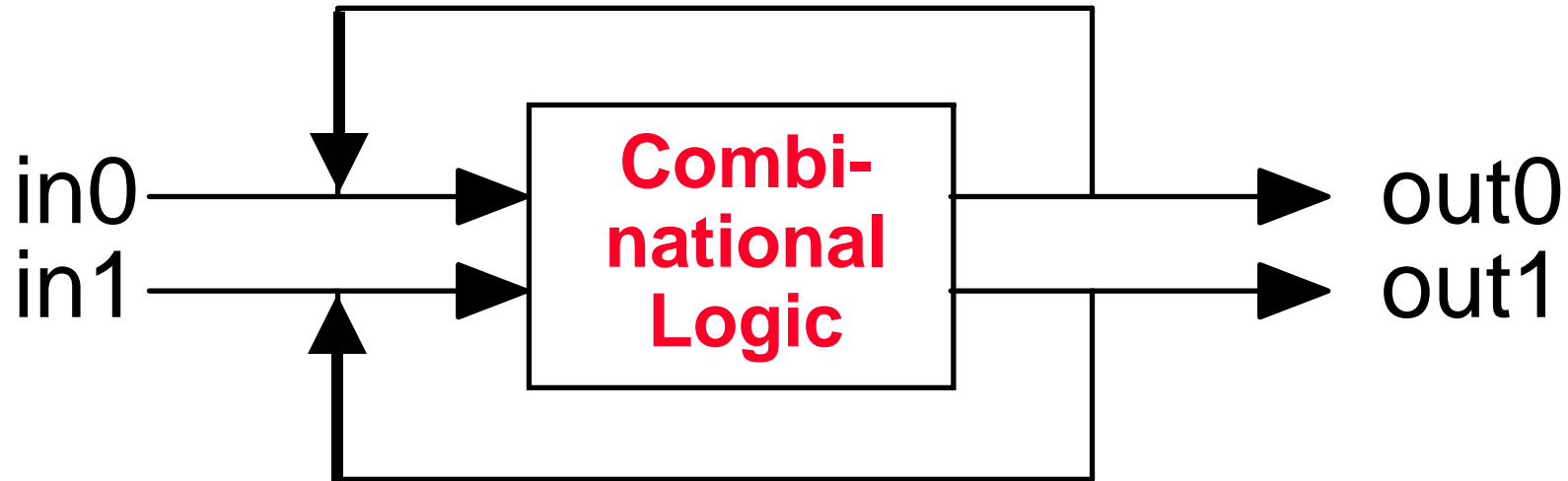
“And In conclusion...”

- **Use muxes to select among input**
 - **S input bits selects 2^S inputs**
 - **Each input can be n-bits wide, indep of S**
- **Implement muxes hierarchically**
- **ALU can be implemented using a mux**
 - **Coupled with basic block elements**
- **N-bit adder-subtractor done using N 1-bit adders with XOR gates on input**
 - **XOR serves as conditional inverter**



State Circuits Overview (Extra Slides)

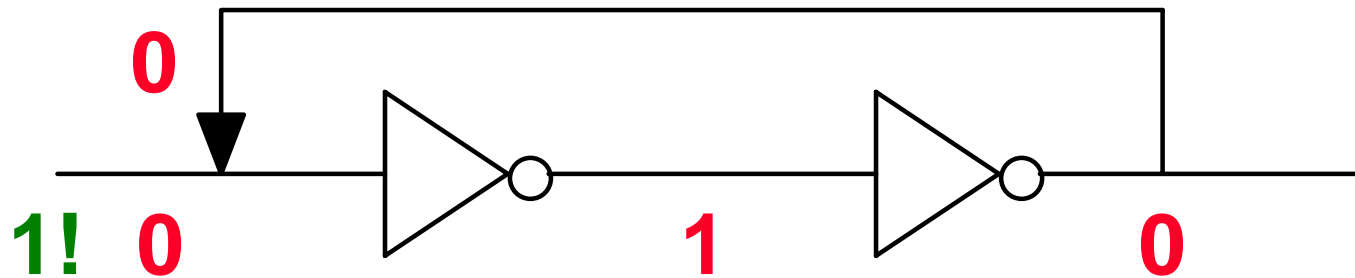
- State circuits have **feedback**, e.g.



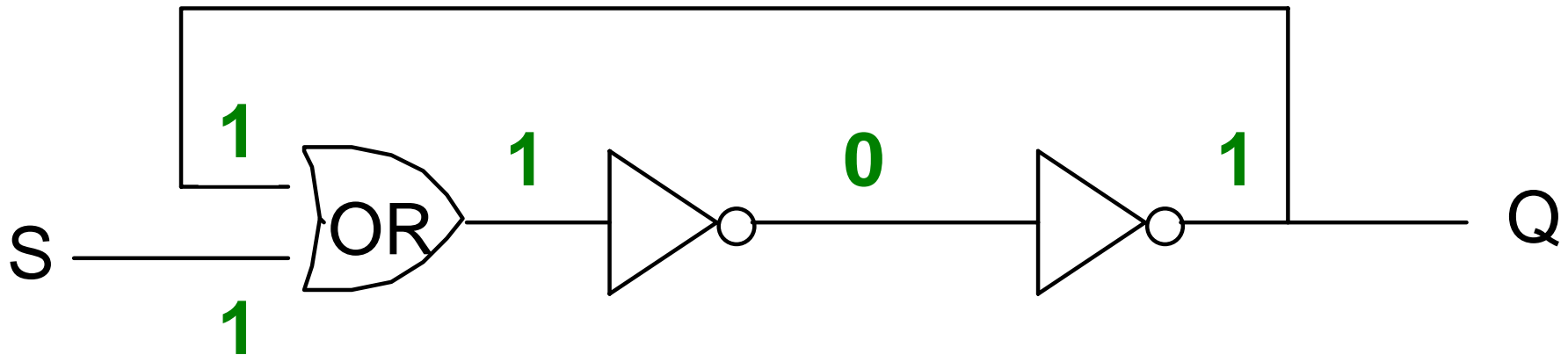
- Output is function of inputs + fed-back signals.
- Feedback signals are the circuit's **state**.
- What aspects of this circuit might cause complications?



A simpler state circuit: two inverters



- When started up, it's **internally stable**.
- Provide an **or** gate for coordination:

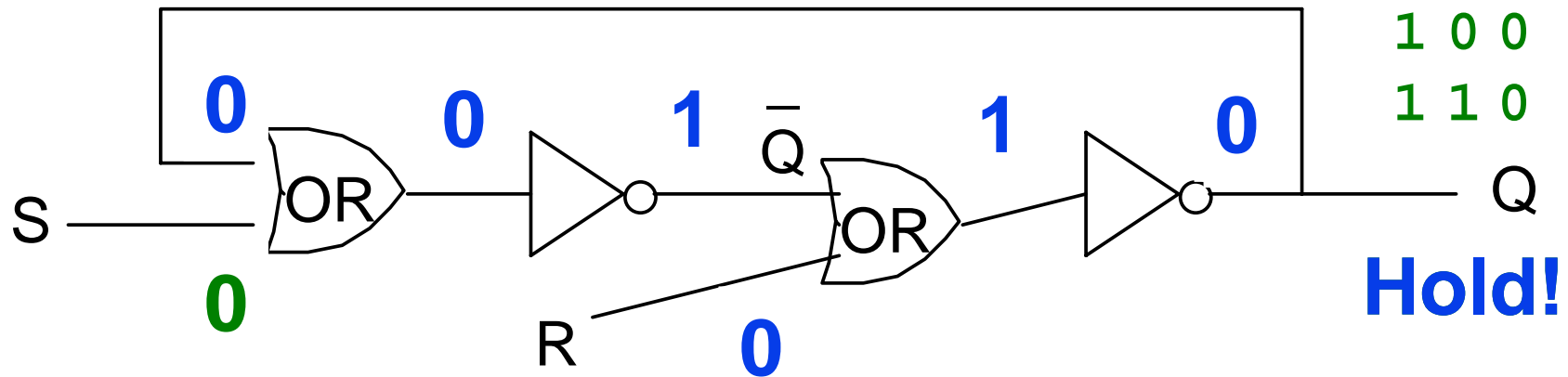


What's the result? **How do we set to 0?**

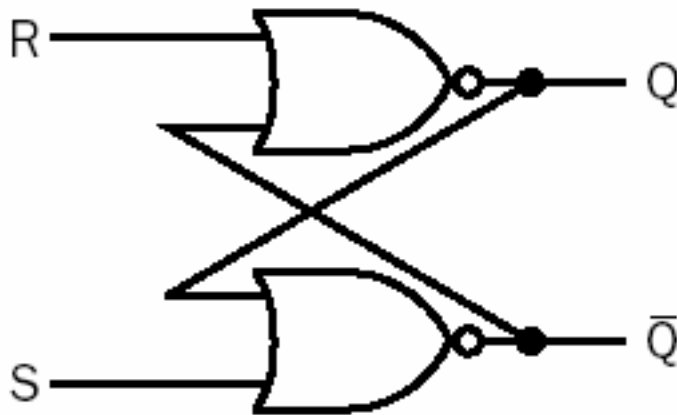
An R-S latch (cross-coupled NOR gates)

- **S** means “set” (to 1),
R means “reset” (to 0).

A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0



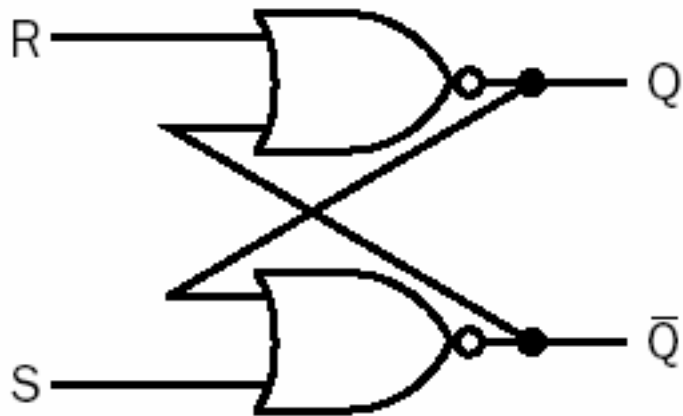
- Adding Q' gives standard RS-latch:



Truth table

S	R	Q
0	0	hold (keep value)
0	1	0
1	0	1
1	1	unstable

An R-S latch (in detail)



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Truth table

S	R	\bar{Q}	Q	Q (t+ Δt)	
0	0	0	1	0	hold
0	0	1	0	1	hold
0	1	0	1	0	reset
0	1	1	0	0	reset
1	0	0	1	1	set
1	0	1	0	1	set
1	1	0	x	x	unstable
1	1	1	x	x	unstable



Controlling R-S latch with a clock

- Can't change R and S while clock is active.

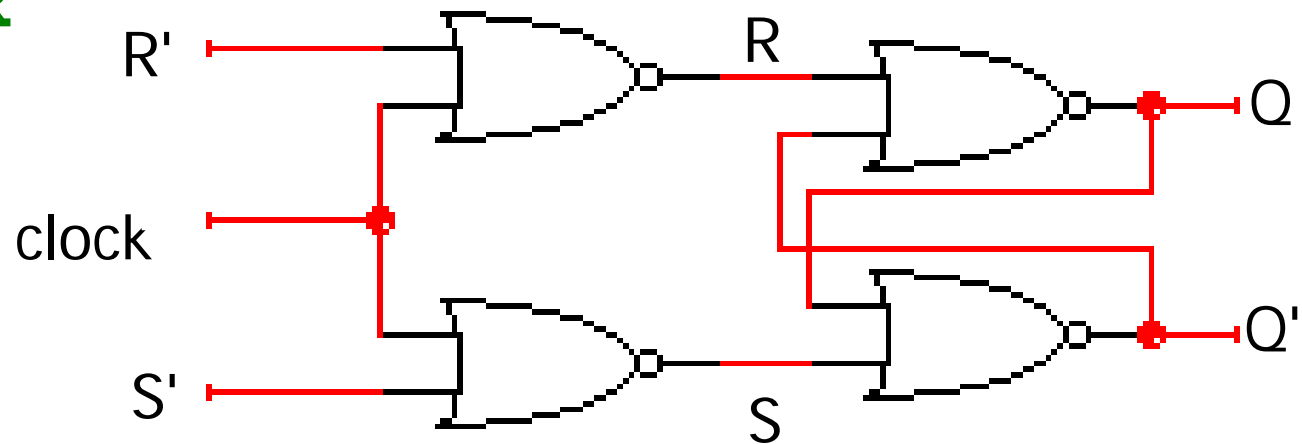
A B NOR

0 0 1

0 1 0

1 0 0

1 1 0



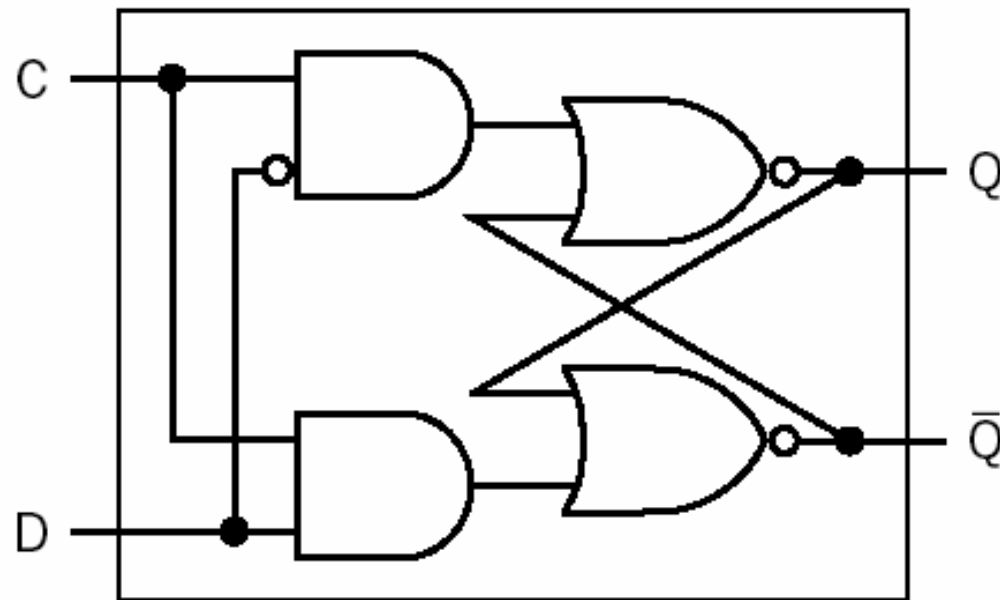
- Clocked latches are called *flip-flops*.



D flip-flop are what we really use

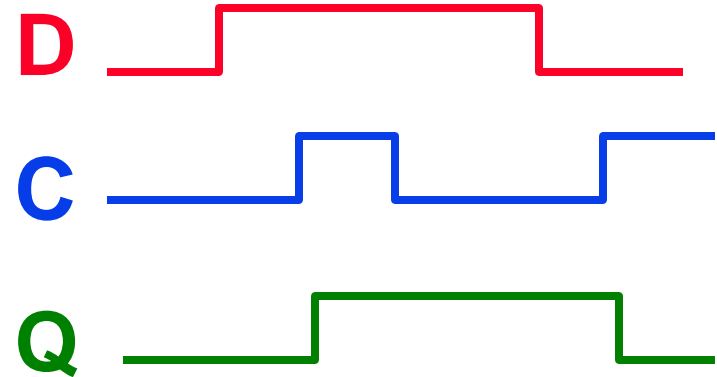
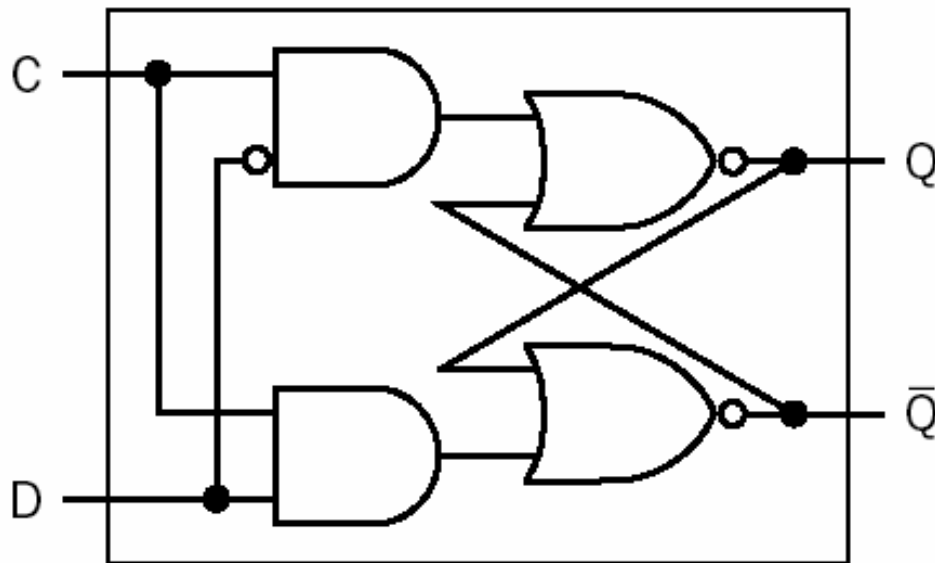
- Inputs C (clock) and D.
- When C is 1, latch open, output = D (even if it changes, “transparent latch”)
- When C is 0, latch closed, output = stored value.

C D AND
0 0 0
0 1 0
1 0 0
1 1 1



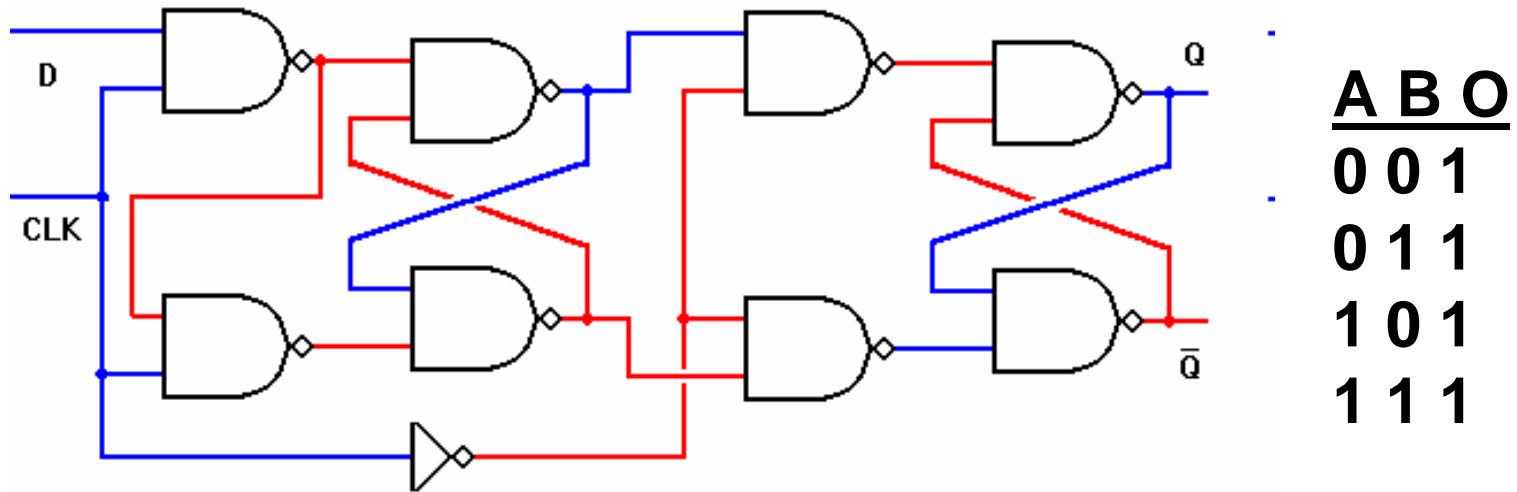
D flip-flop details

- We don't like transparent latches
- We can build them so that the latch is only open for an instant, on the **rising edge of a clock** (as it goes from 0⇒1)



Timing Diagram

Edge Detection



- This is a “rising-edge D Flip-Flop”
 - When the CLK transitions from 0 to 1 (rising edge) ...
 - $Q \leftarrow D$; $Q_{\text{bar}} \leftarrow \text{not } D$
 - All other times: $Q \leftarrow Q$; $Q_{\text{bar}} \leftarrow Q_{\text{bar}}$

