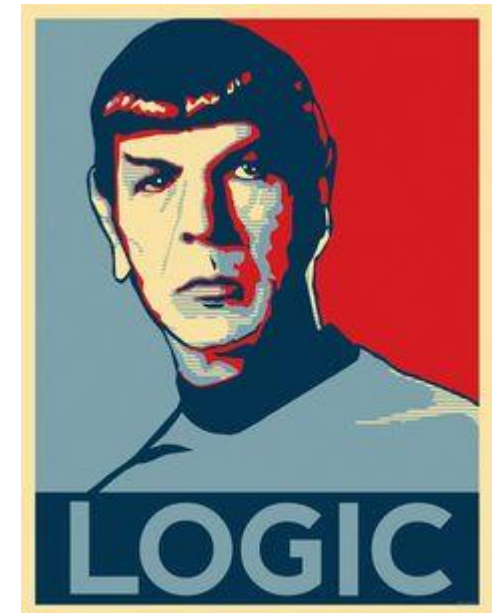
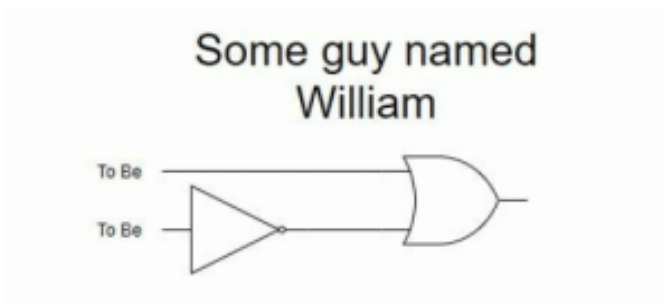


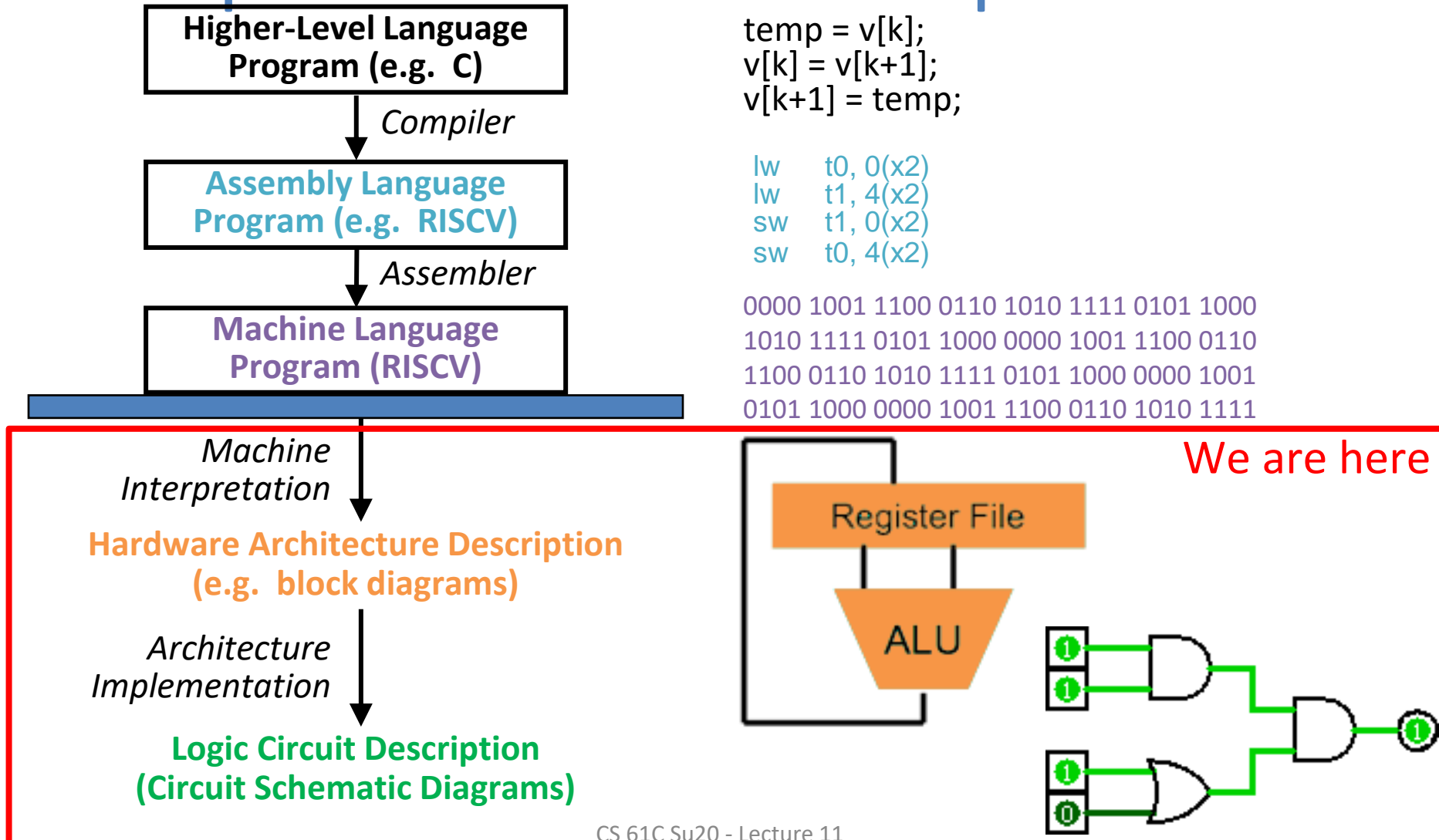
Great Ideas in Computer Architecture

Sequential Digital Logic

Instructor: Sean Farhat

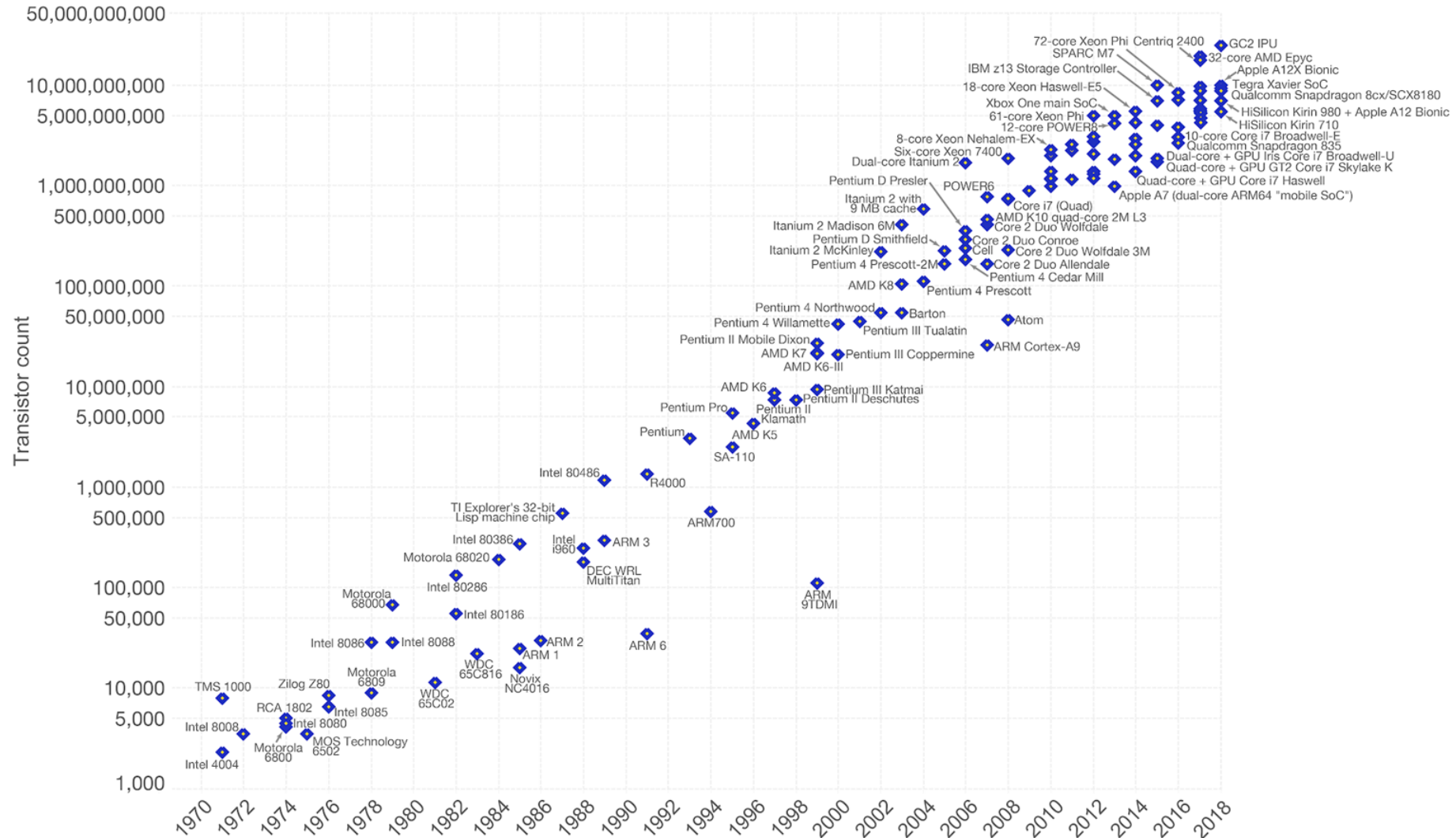


Great Idea #1: Levels of Representation & Interpretation



Our World
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



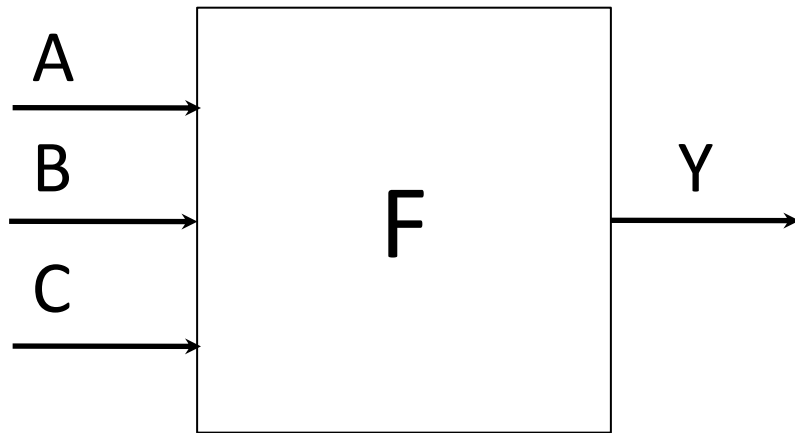
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at [OurWorldinData.org](https://ourworldindata.org). There you find more visualizations and research on this topic.

Licensed under [CC-BY-SA](#) by the author Max Roser.

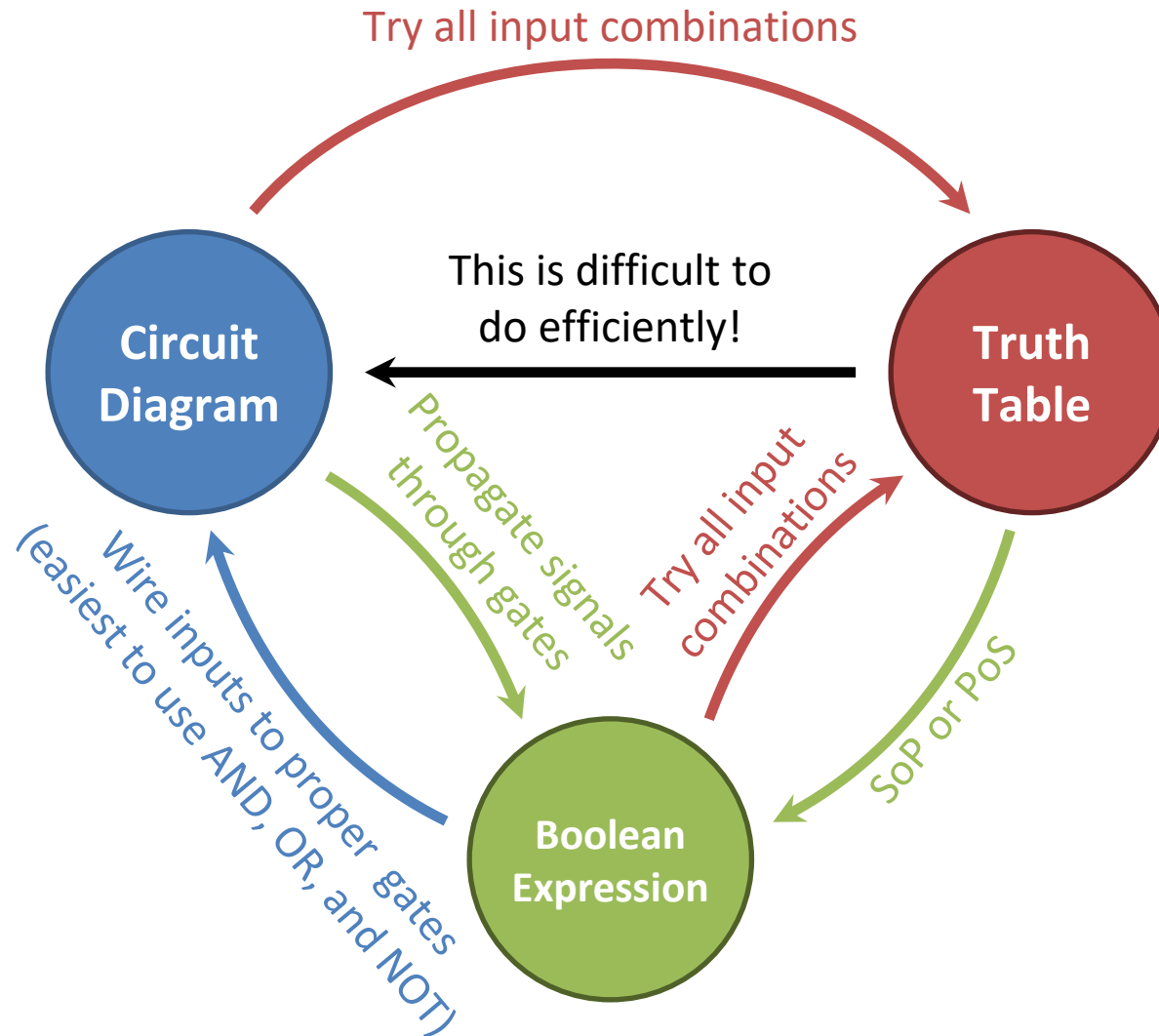
Review

- Combinational Logic

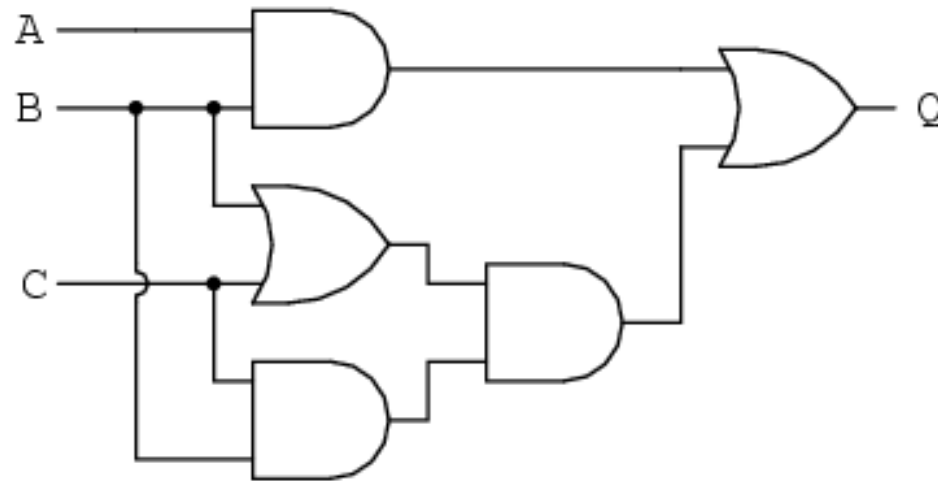


A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Converting Combinational Logic

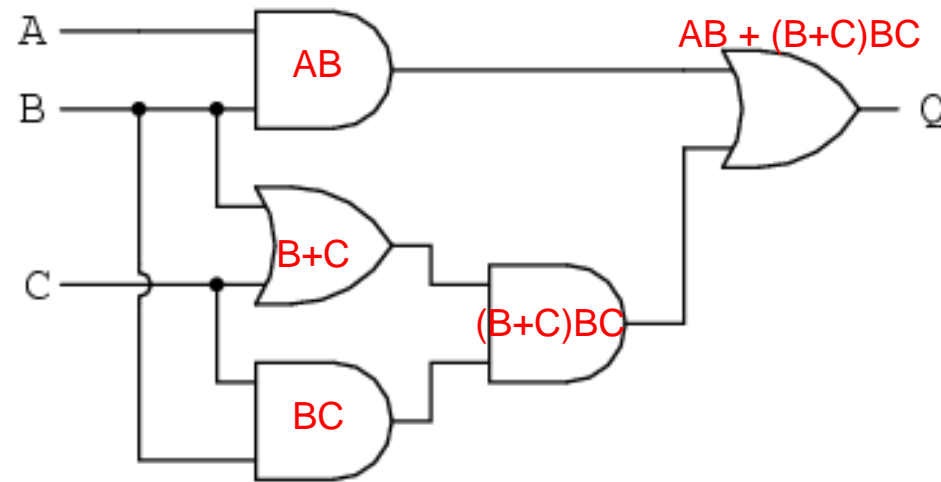


Question: Which boolean equation represents the following circuit?



- (A) $AB+BC+C$
- (B) $AB+(B+C)BC$
- (C) $(A+B)((BC)+(B+C))$
- (D) Q

Question: Which boolean equation represents the following circuit?



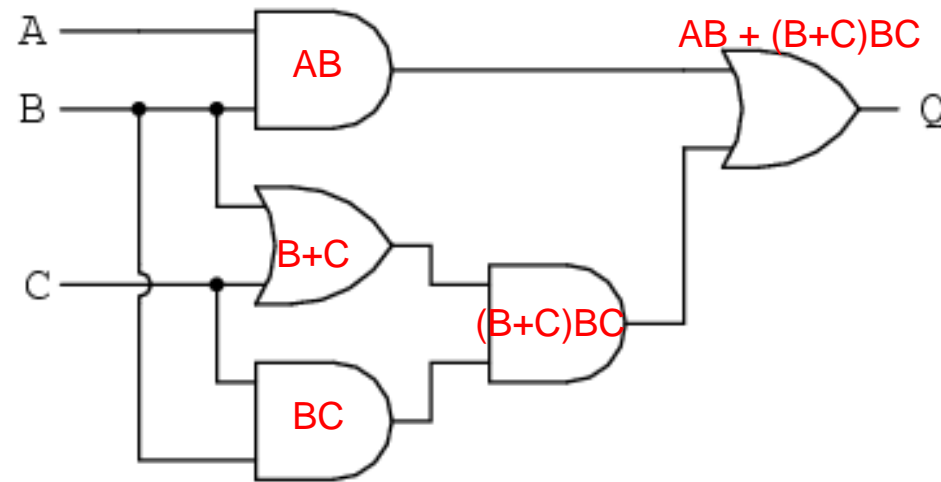
(A) $AB+BC+C$

(B) $AB+(B+C)BC$

(C) $(A+B)((BC)+(B+C))$

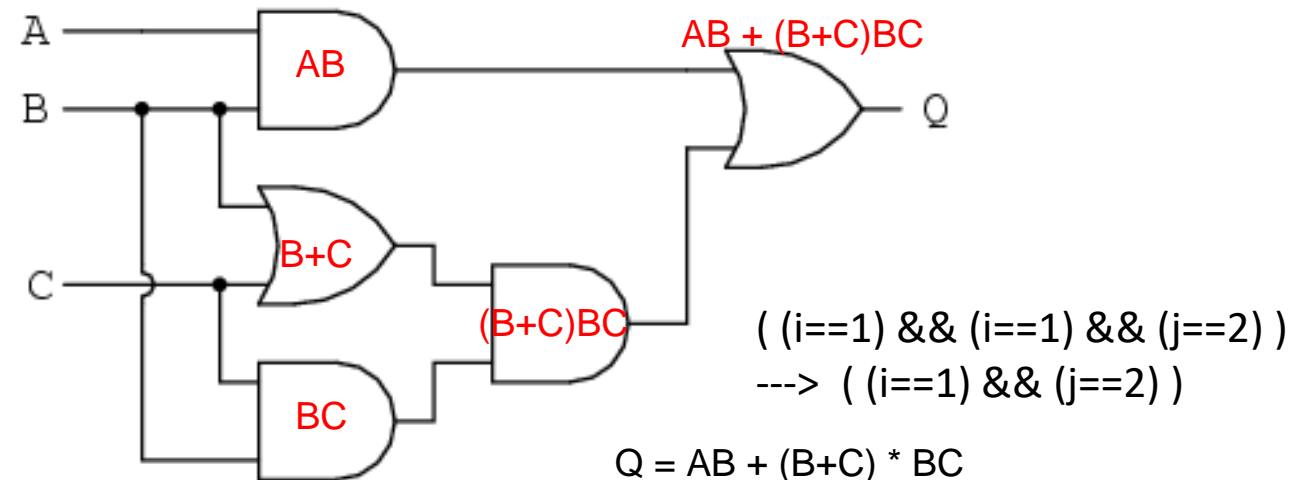
(D) Q

Question: What is the MOST simplified Boolean Algebra expression for the following circuit?



- (A) $B(A + C)$
- (B) $B + AC$
- (C) $AB + B + C$
- (D) $A + C$

Question: What is the MOST simplified Boolean Algebra expression for the following circuit?



(A) $B(A + C)$

(B) $B + AC$

(C) $AB + B + C$

(D) $A + C$

$$Q = AB + (B+C) * BC$$

By distributing the BC into (B+C) we get:

$$Q = AB + (BBC + CBC)$$

Using the multiplicative idempotent law again we know that $B*B = B$ and that $C*C=C$ so we get:

$$Q = AB + (BC + BC)$$

Using the additive idempotent law ($B + B = B$) we get:

$$Q = AB + BC$$

By factoring out the B, we get the final answer of

$$Q = B(A+C)$$

Agenda

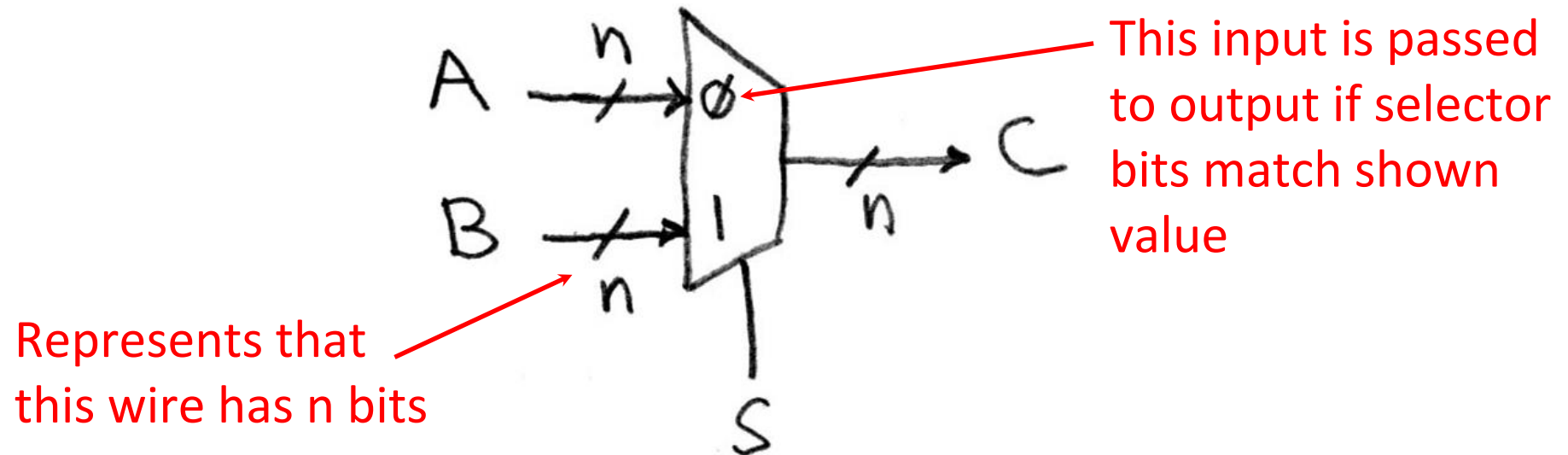
- **Muxes**
- Sequential Logic Timing
- Maximum Clock Frequency
- Finite State Machines
- Functional Units
- Summary

Bonus Slides

- Logisim Intro

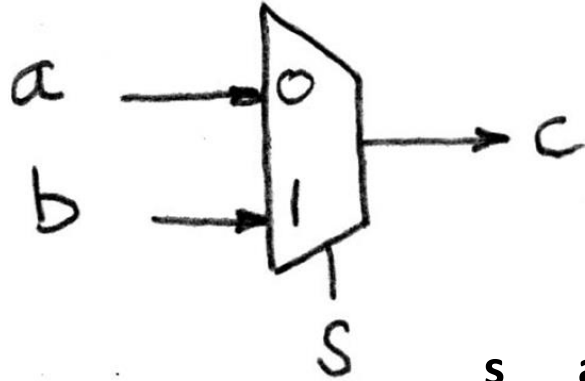
Data Multiplexor

- Multiplexor (“MUX”) is a *selector*
 - Place one of multiple inputs onto output (N-to-1)
- Shown below is an n-bit 2-to-1 MUX
 - Input S selects between two inputs of n bits each



Implementing a 1-bit 2-to-1 MUX

- **Schematic:**



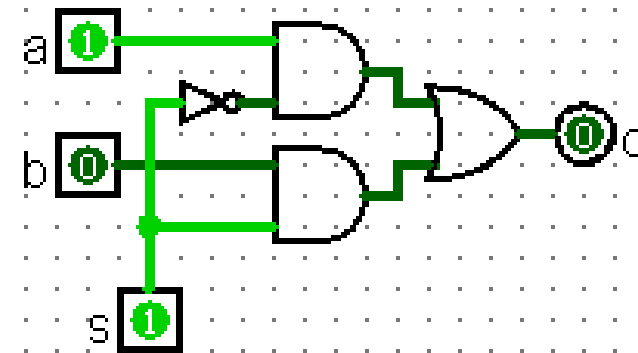
- **Truth Table:**

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

- **Boolean Algebra:**

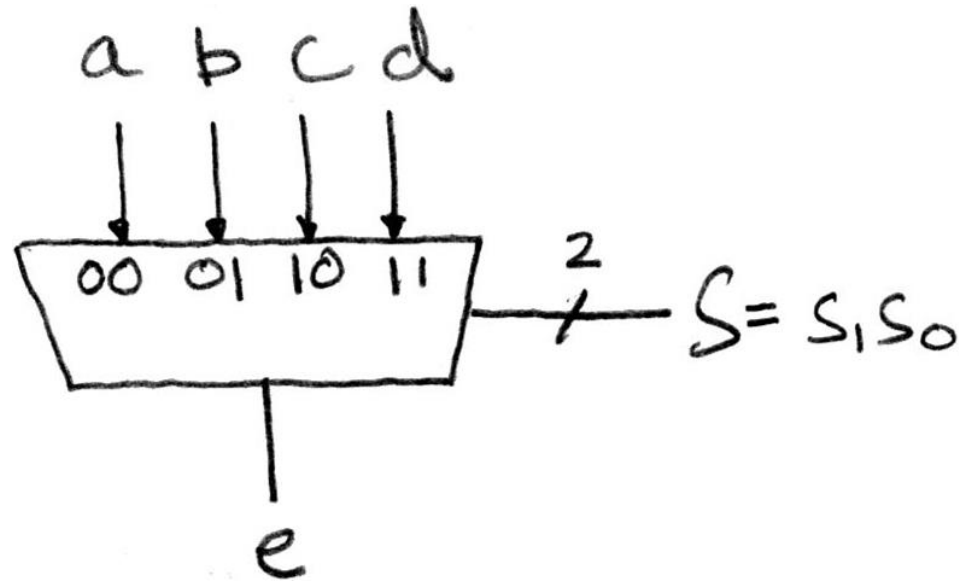
$$\begin{aligned}c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\&= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\&= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\&= \bar{s}(a(1)) + s((1)b) \\&= \bar{s}a + sb\end{aligned}$$

- **Circuit Diagram:**



1-bit 4-to-1 MUX (1/2)

- **Schematic:**



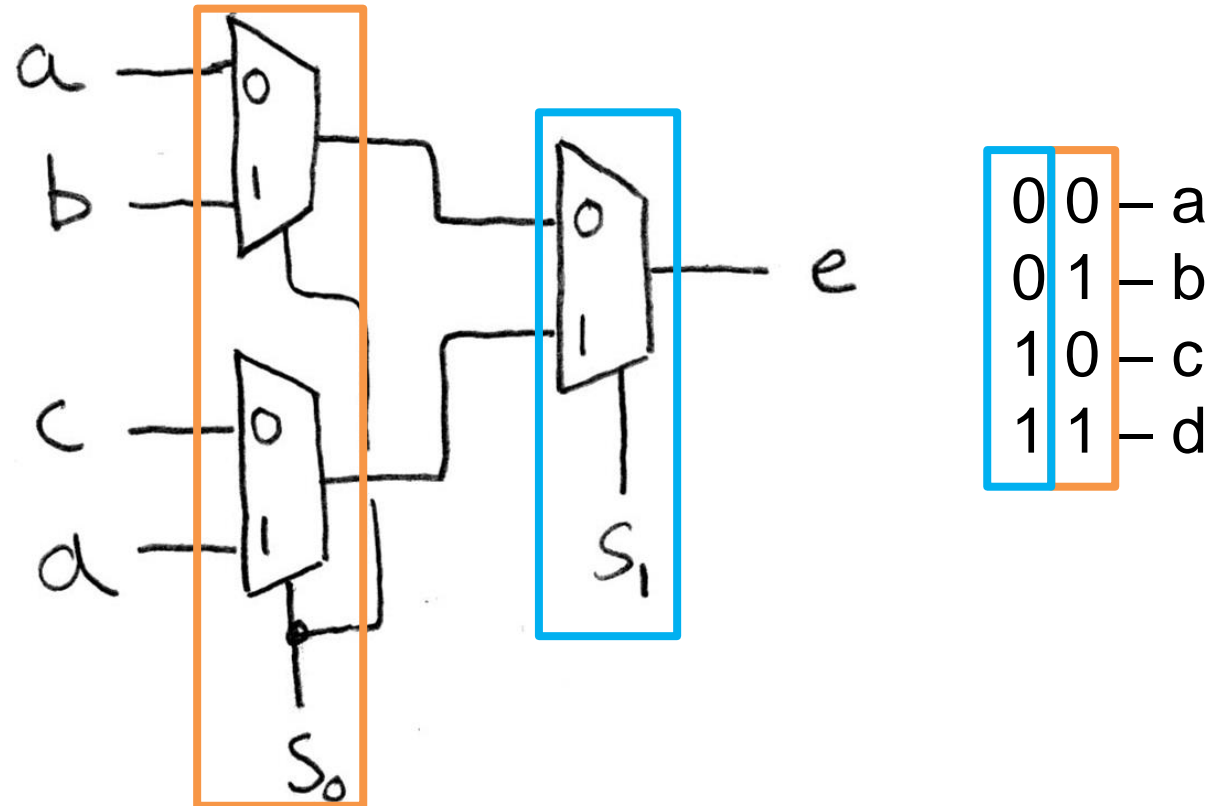
- **Truth Table:** How many rows? 2^6

- **Boolean Expression:**

$$e = \neg s_1 \neg s_0 a + \neg s_1 s_0 b + s_1 \neg s_0 c + s_1 s_0 d$$

1-bit 4-to-1 MUX (2/2)

- Can we leverage what we've previously built?
 - Alternative hierarchical approach:



Agenda

- Muxes
- **Sequential Logic Timing**
- Maximum Clock Frequency
- Finite State Machines
- Functional Units
- Summary

Bonus Slides

- Logisim Intro

Type of Circuits

- *Digital Systems* consist of two basic types of circuits:
 - Combinational Logic (CL)
 - Output is a function of the inputs only, not the history of its execution
 - e.g. circuits to add A, B (ALUs)
 - Sequential Logic (SL)
 - Circuits that “remember” or store information
 - a.k.a. “State Elements”
 - e.g. memory and registers (Registers)

Uses for State Elements

- Place to store values for some amount of time:
 - Register files (like in RISC-V)
 - Memory (caches and main memory)
- *Help control flow of information between combinational logic blocks*
 - State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage

Accumulator Example

An example of why we would need sequential logic

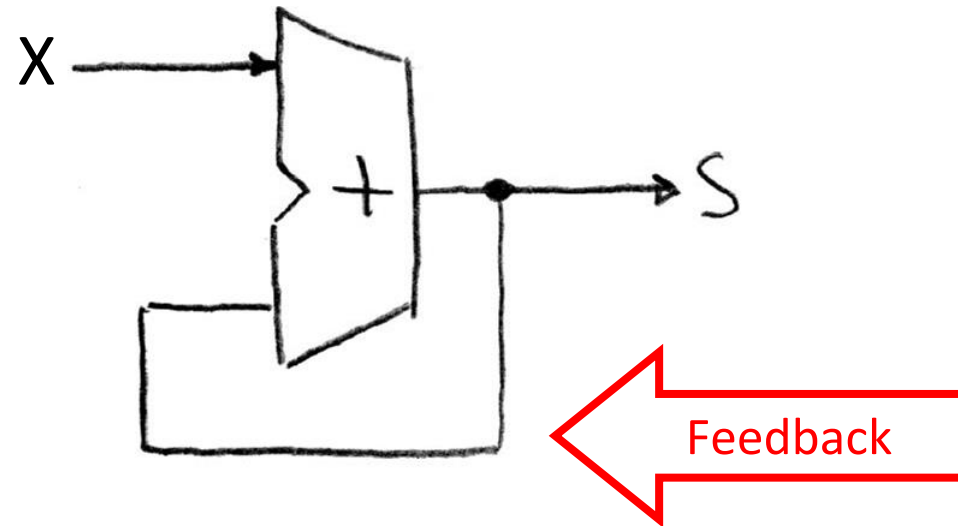


Want: $S=0;$
 for X_1, X_2, X_3 over time...
 $S = S + X_i$

Assume:

- Each X value is applied in succession, one per cycle
- The sum since time 1 (cycle) is present on S

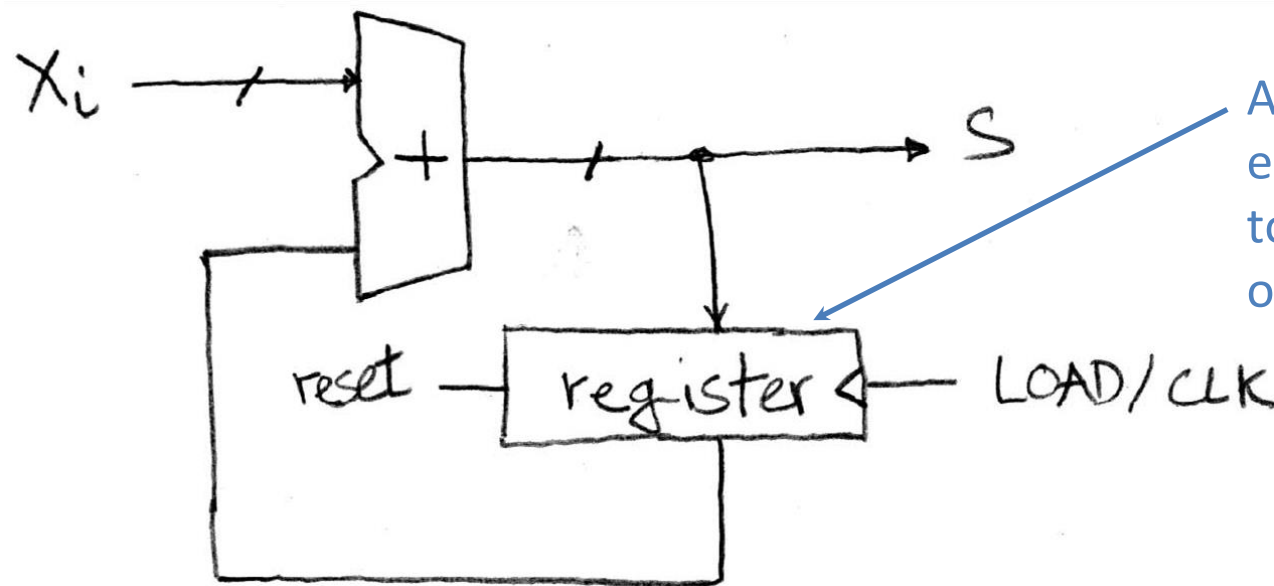
First Try: Does this work?



No!

- 1) How to control the next iteration of the 'for' loop?
- 2) How do we say: 'S=0'?

Second Try: How About This?

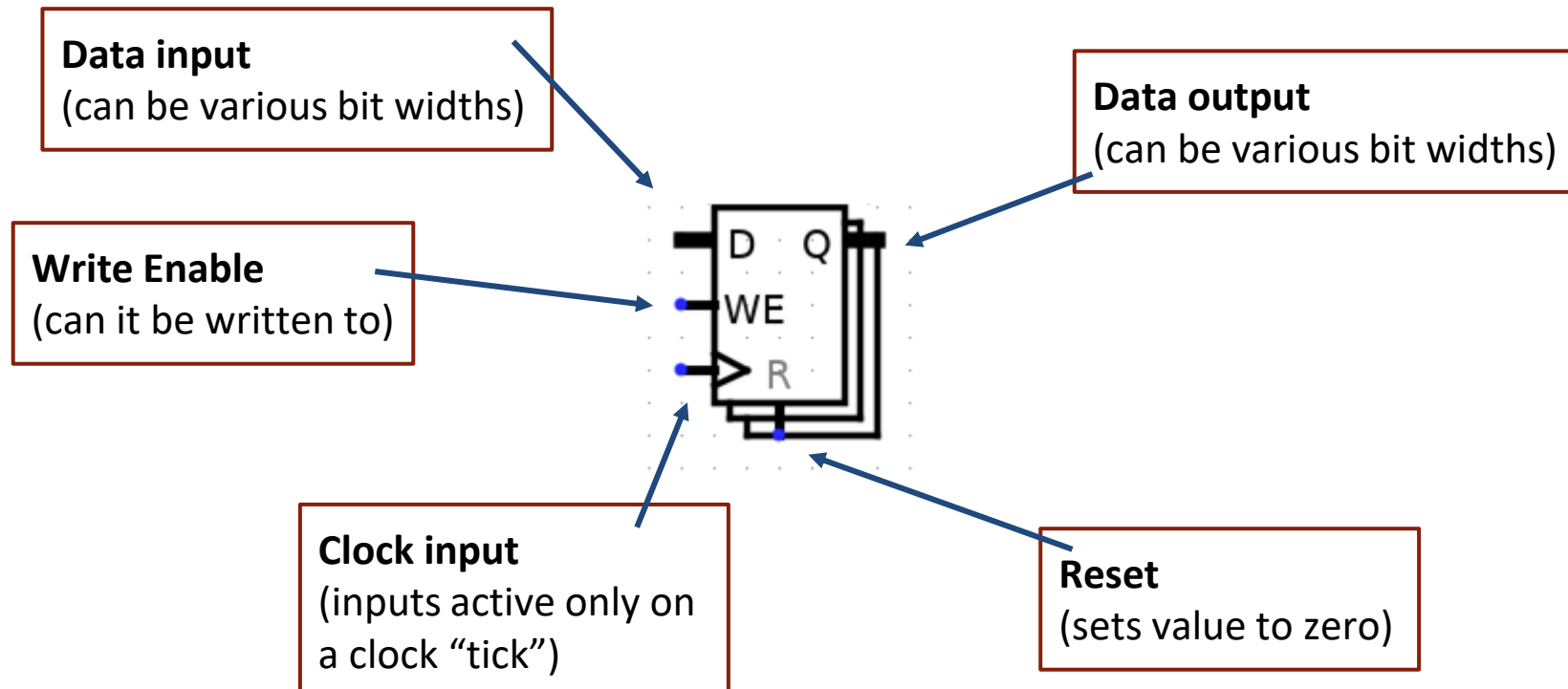


A *Register* is the state element that is used here to hold up the transfer of data to the adder

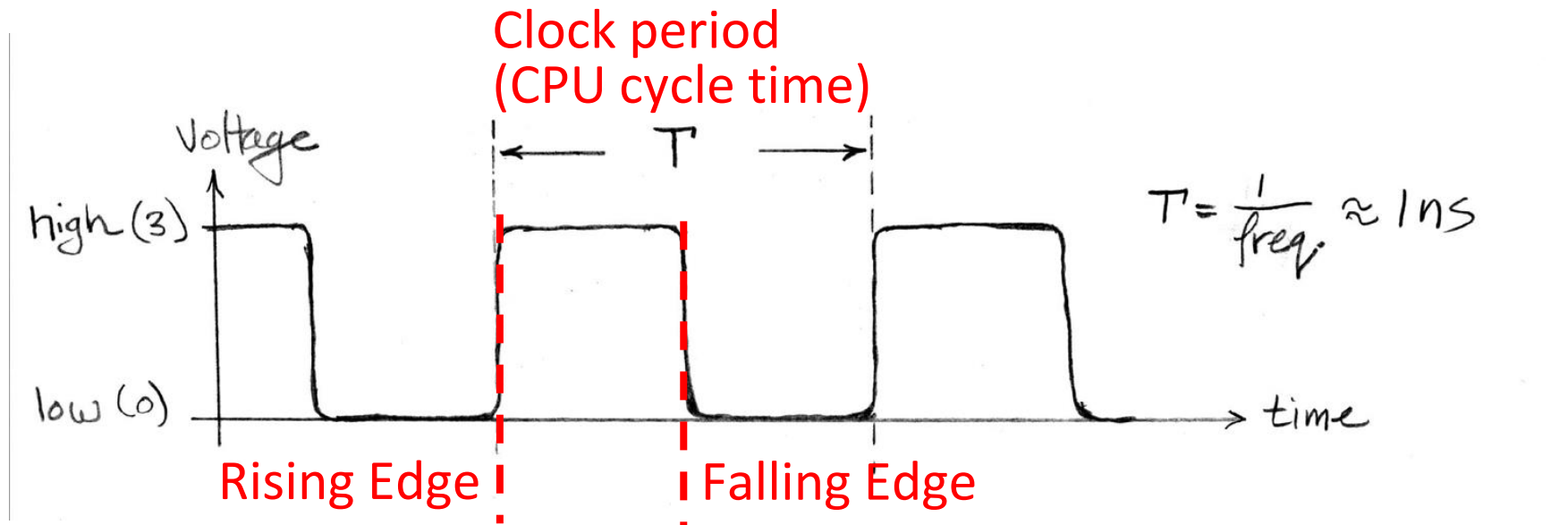
Registers

Same as registers in assembly:

- small memory storage locations

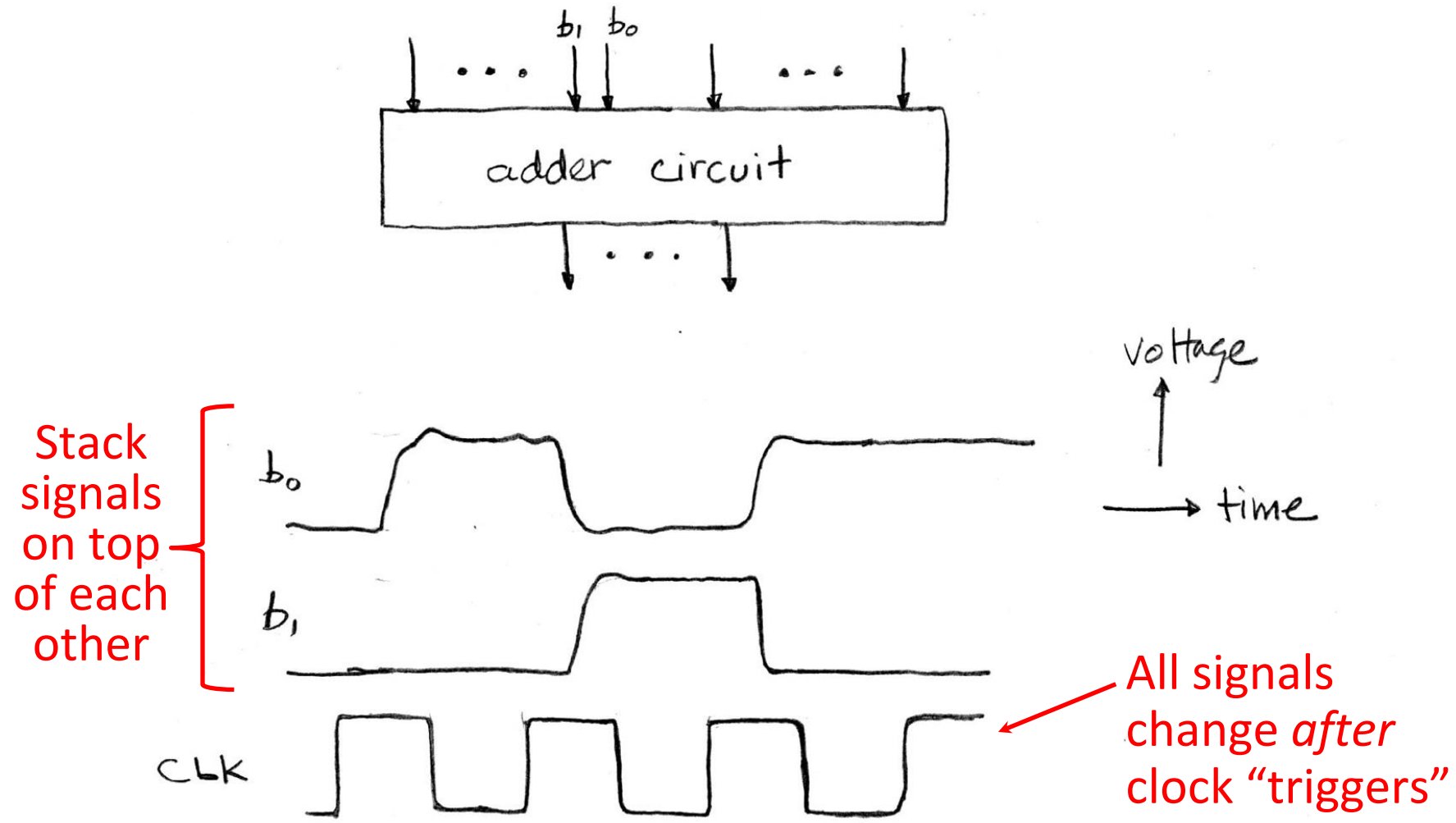


Signals and Waveforms: Clocks



- **Signals** transmitted over wires continuously
- Transmission is effectively instantaneous
 - Implies that any wire only contains one value at any given time

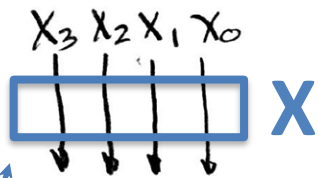
Signals and Waveforms



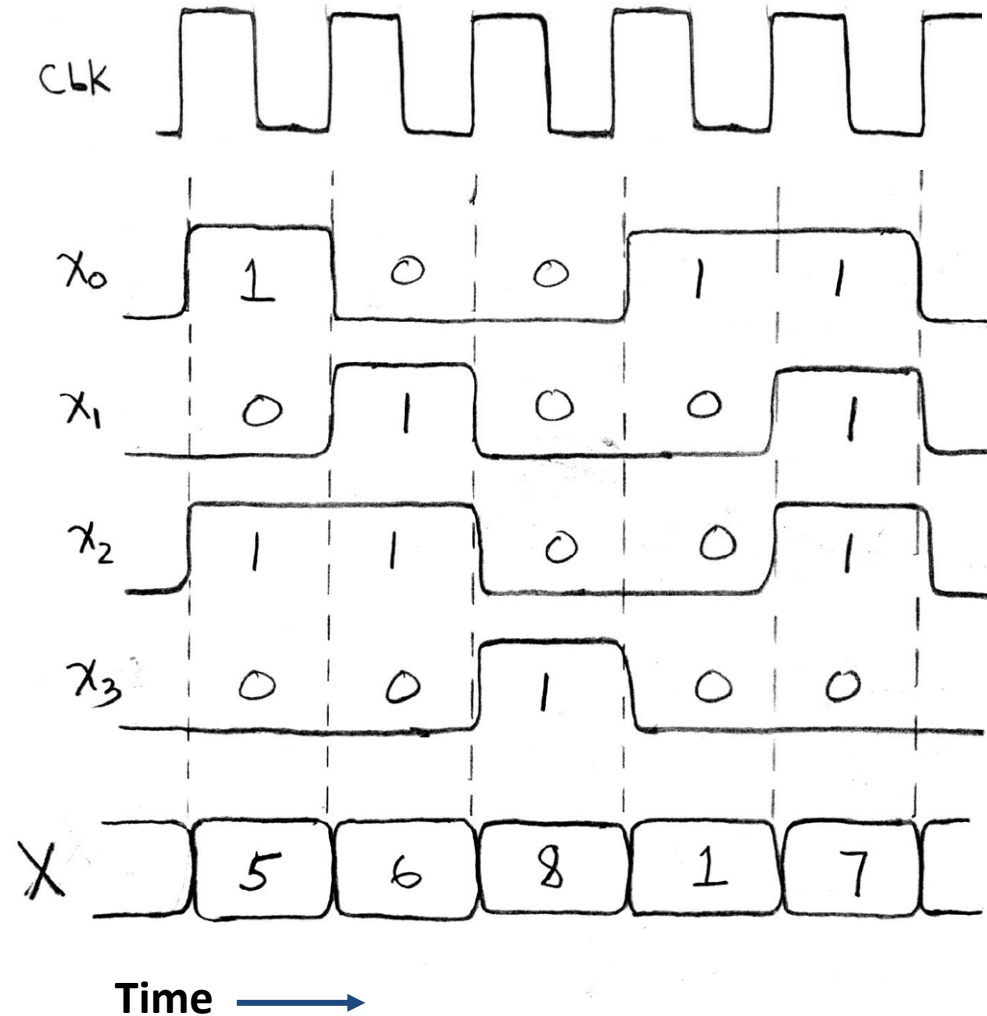
Dealing with Waveform Diagrams

- Easiest to start with CLK on top
 - Solve signal by signal, from inputs to outputs
 - Can only draw the waveform for a signal if *all* of its input waveforms are drawn
- When does a signal update?
 - A *state element* updates based on CLK triggers
 - A *combinational element* updates ANY time ANY of its inputs changes

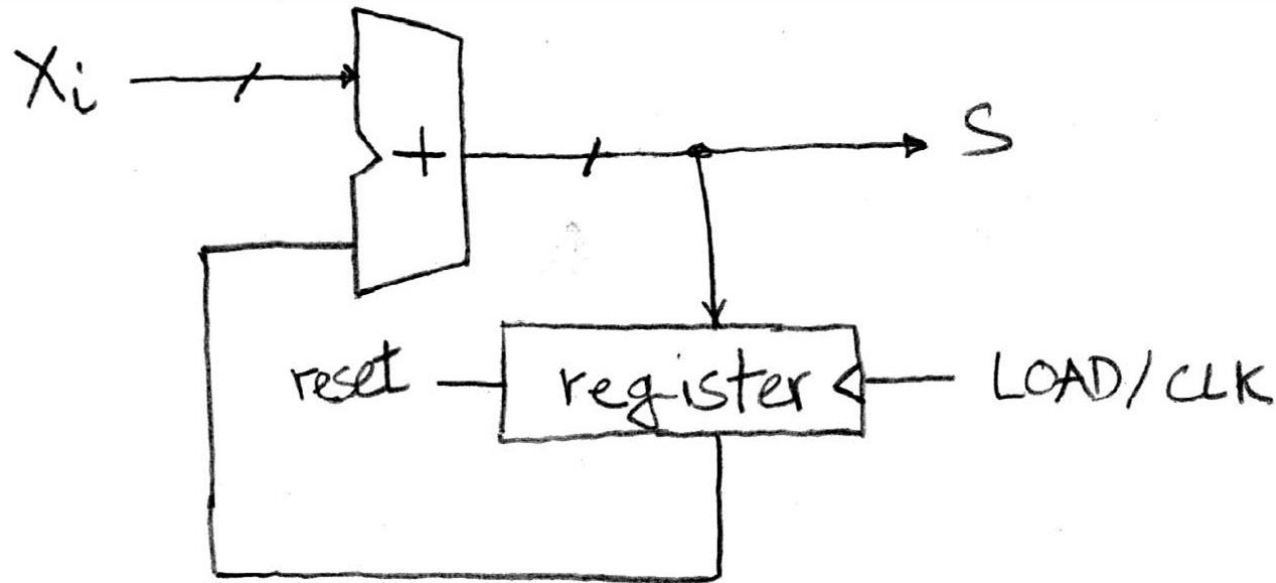
Signals and Waveforms: Grouping



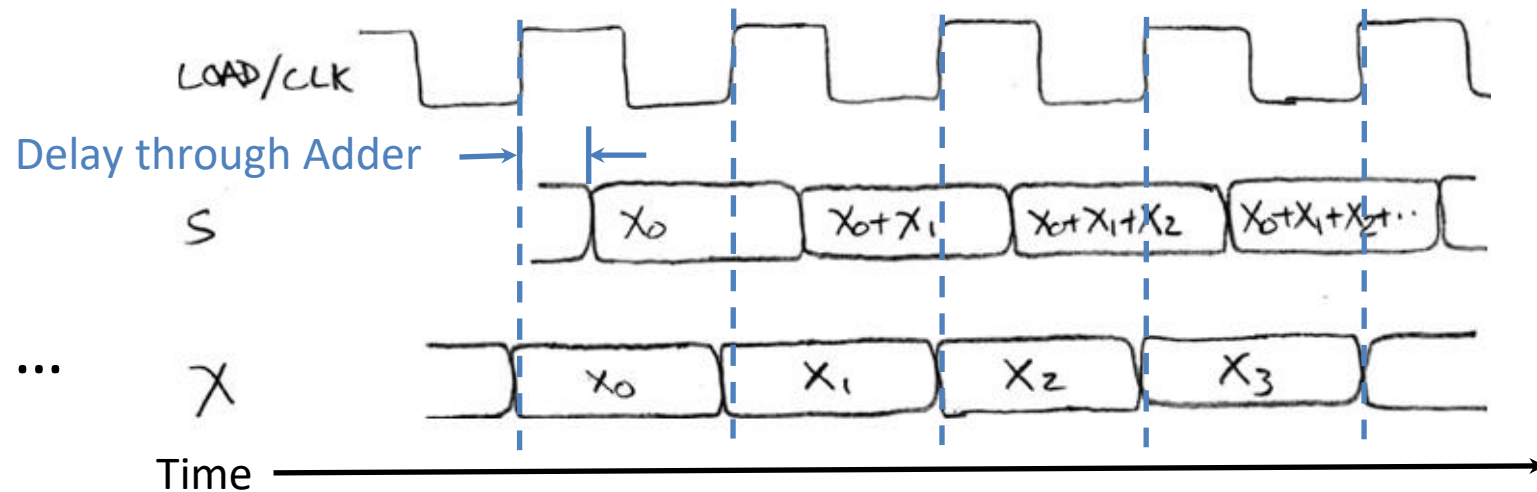
A group of wires when interpreted as a bit field is called a *bus*



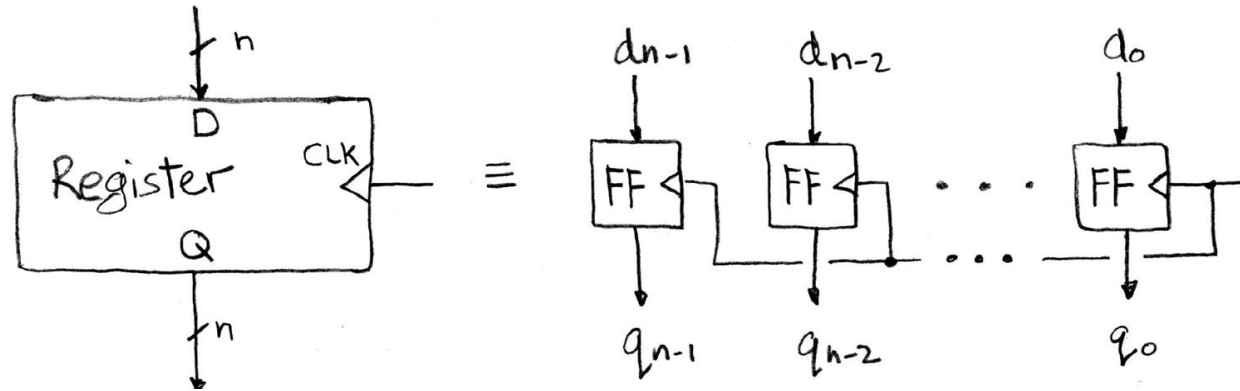
Second Try: How About This?



Rough
timing ...



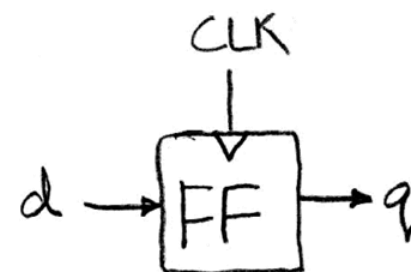
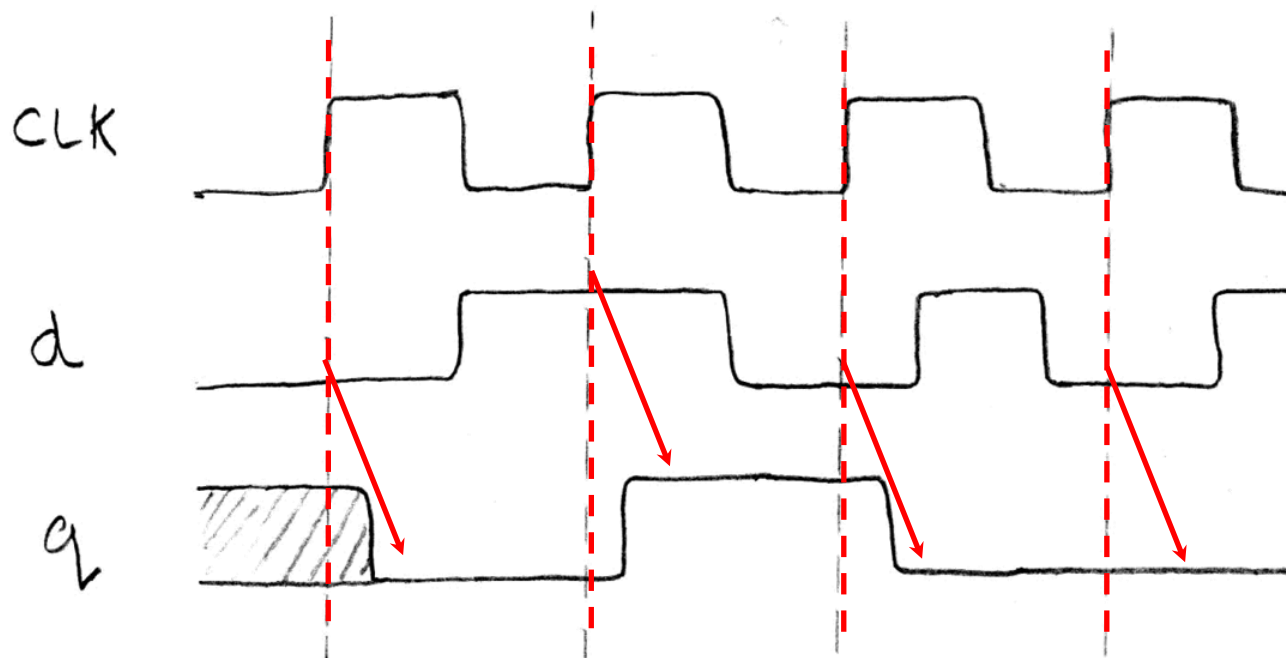
Register Internals



- n instances of a *“Flip-Flop”*
 - Output flips and flops between 0 and 1
- Specifically this is a “D-type Flip-Flop”
 - D is “data input”, Q is “data output”
 - In reality, has 2 outputs (Q and \bar{Q}), but we only care about 1
- http://en.wikibooks.org/wiki/Practical_Electronics/Flip-flops

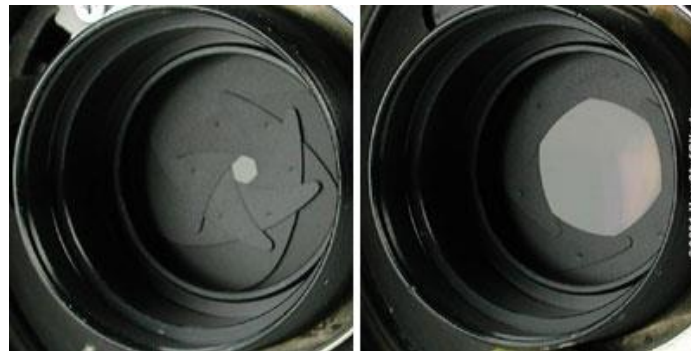
Flip-Flop Timing Behavior (1/2)

- Edge-triggered D-type flip-flop
 - This one is “rising edge-triggered”
- “On the rising edge of the clock, input d is sampled and transferred to the output. At other times, the input d is ignored and the previously sampled value is retained.”
- Example waveforms:



Flip-Flop Timing Terminology (1/3)

- Camera Analogy: non-blurry digital photo
 - *Don't move* while camera shutter is opening
 - *Don't move* while camera shutter is closing
 - *Wait until* image appears on the display

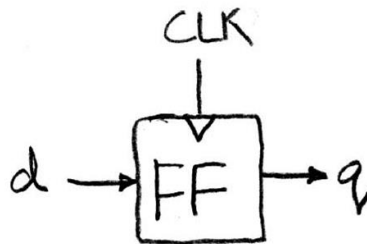


Flip-Flop Timing Terminology (2/3)

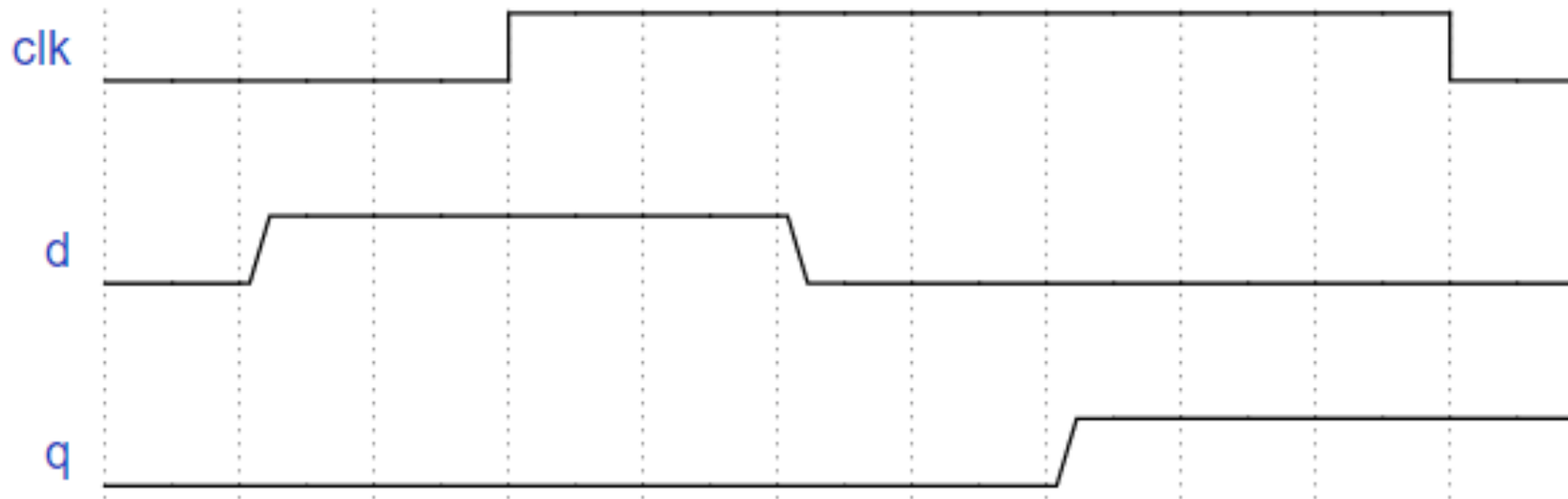
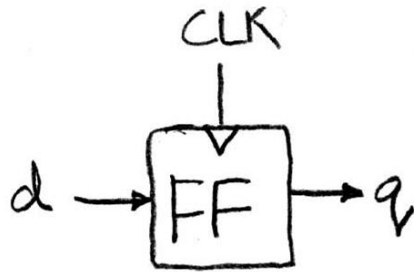
- Camera Analogy: Taking a photo
 - *Setup time*: don't move since about to take picture (open camera shutter)
 - *Hold time*: need to hold still after shutter opens until camera shutter closes
 - *Time to data*: time from open shutter until image appears on the output (viewfinder)

Flip-Flop Timing Terminology (3/3)

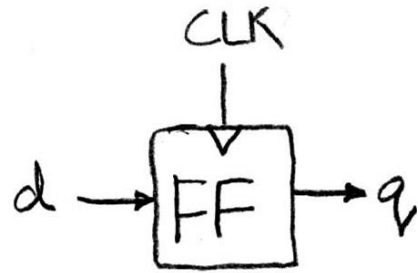
- Now applied to hardware:
 - *Setup Time*: how long the input must be stable *before* the clock trigger for proper input read
 - *Hold Time*: how long the input must be stable *after* the clock trigger for proper input read
 - *“Clock-to-Q” Delay*: how long it takes the output to change, measured from the clock trigger



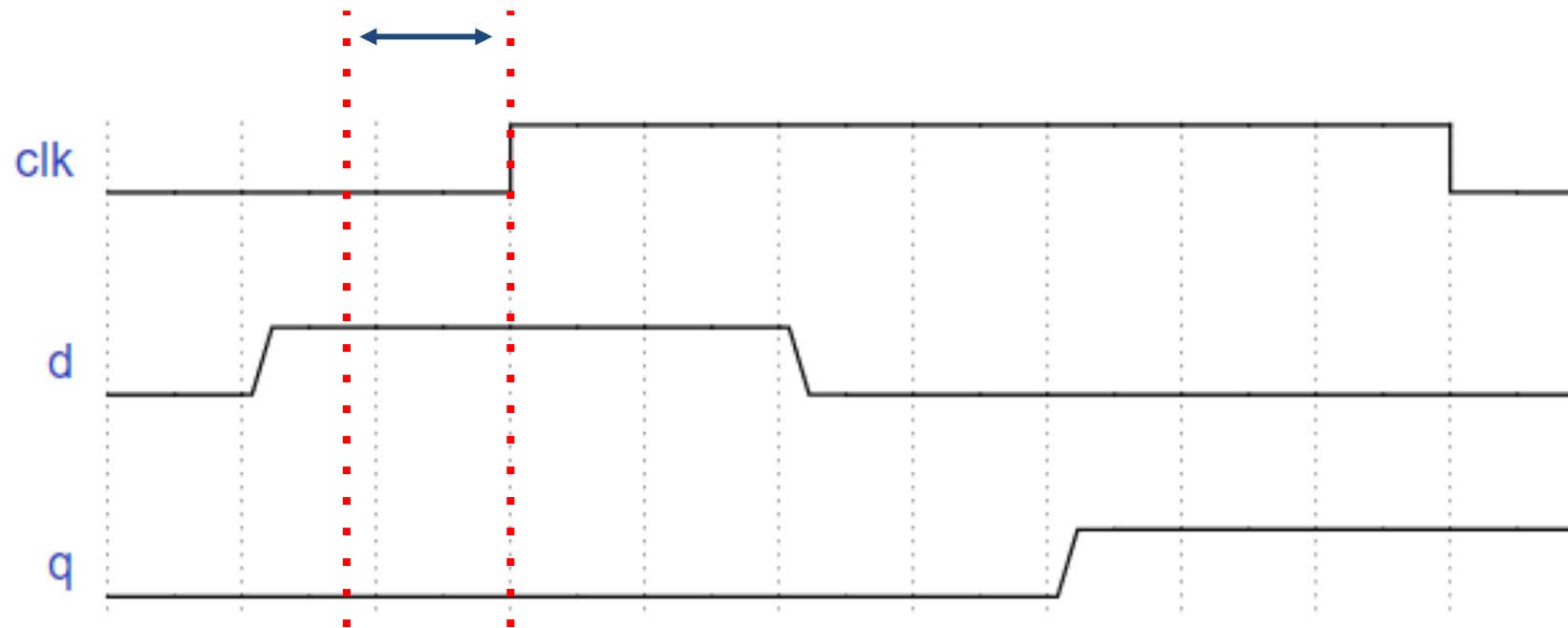
Flip-Flop Timing Behavior



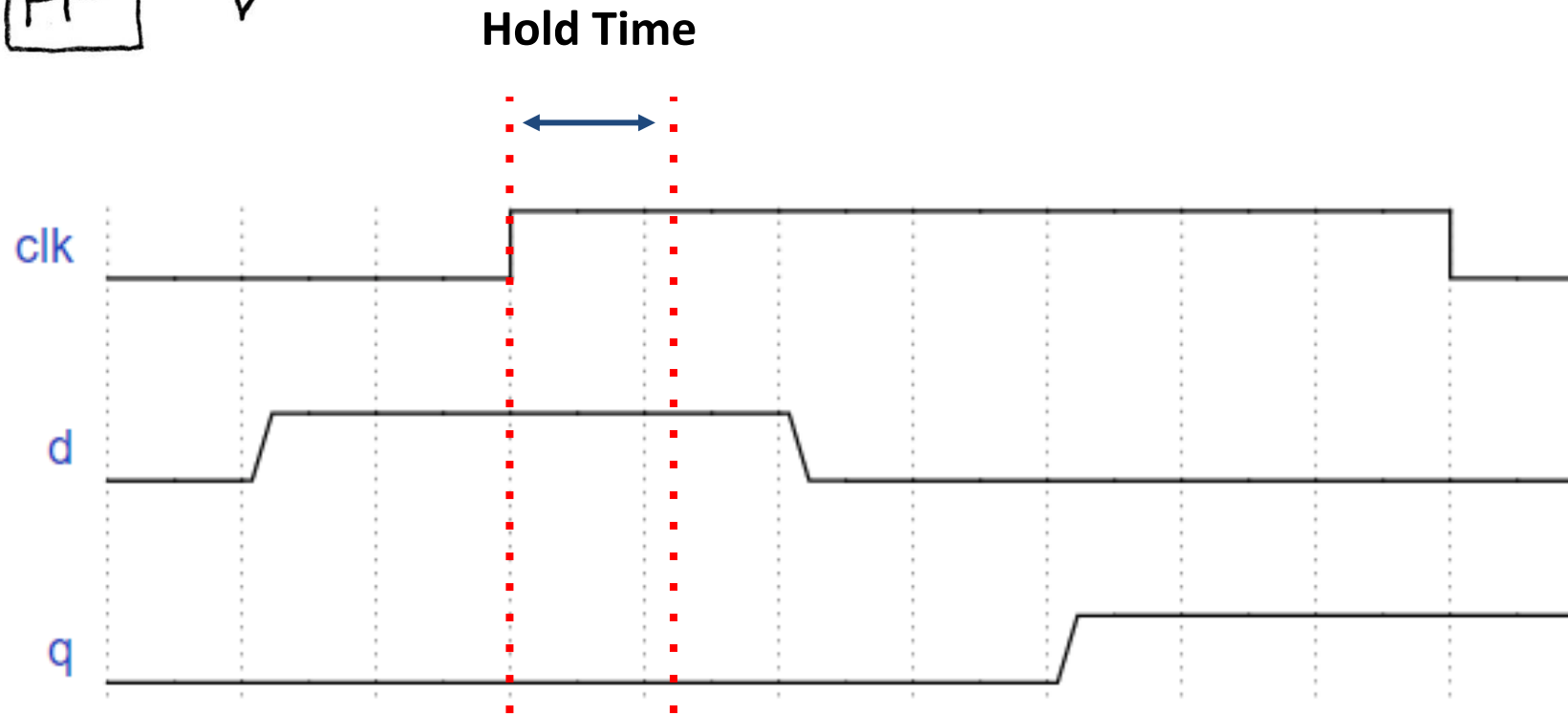
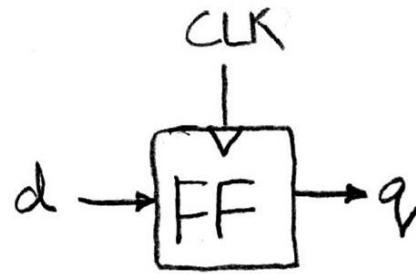
Flip-Flop Timing Behavior



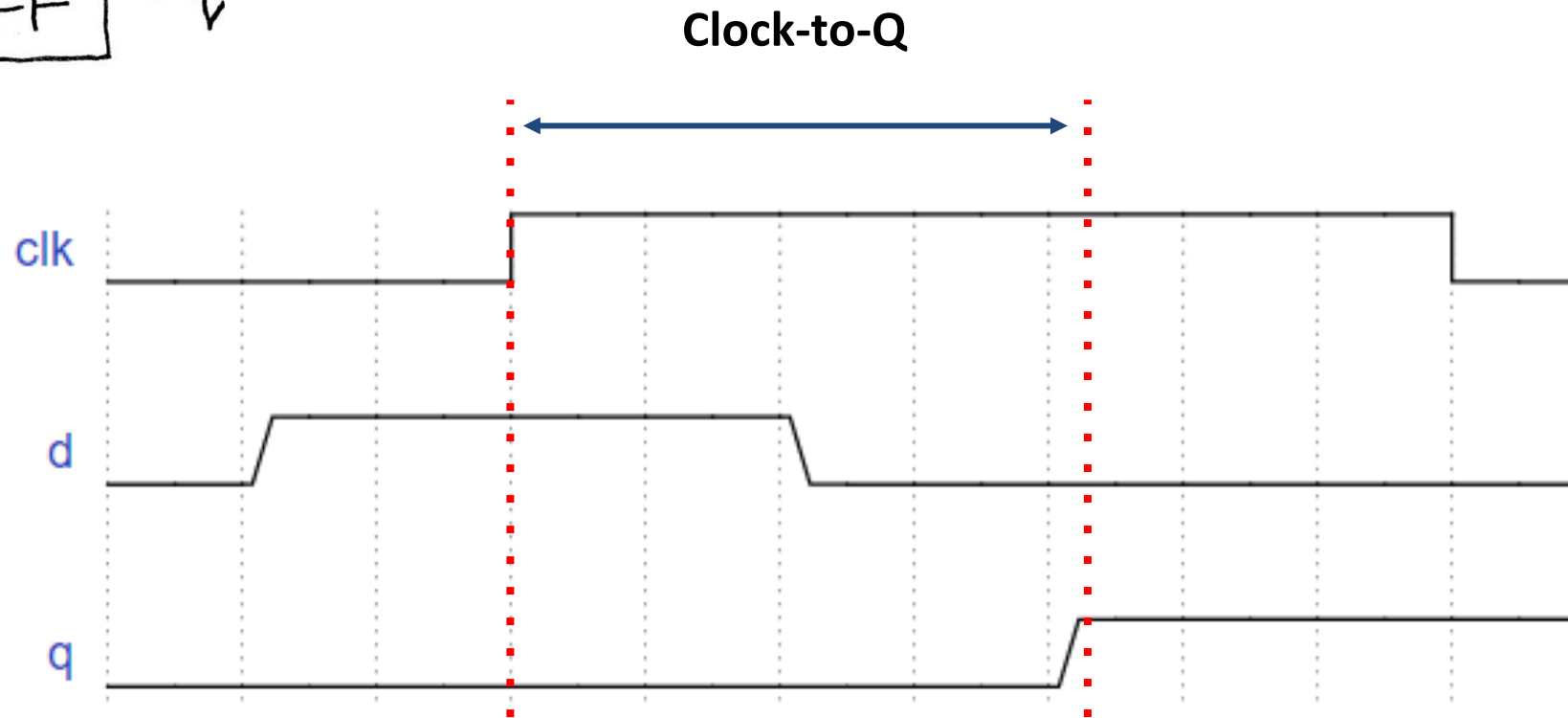
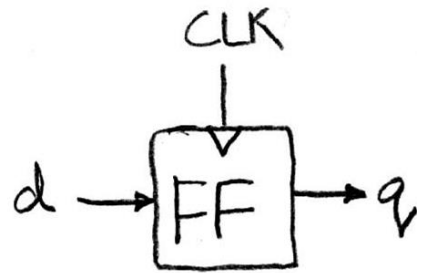
Setup Time



Flip-Flop Timing Behavior

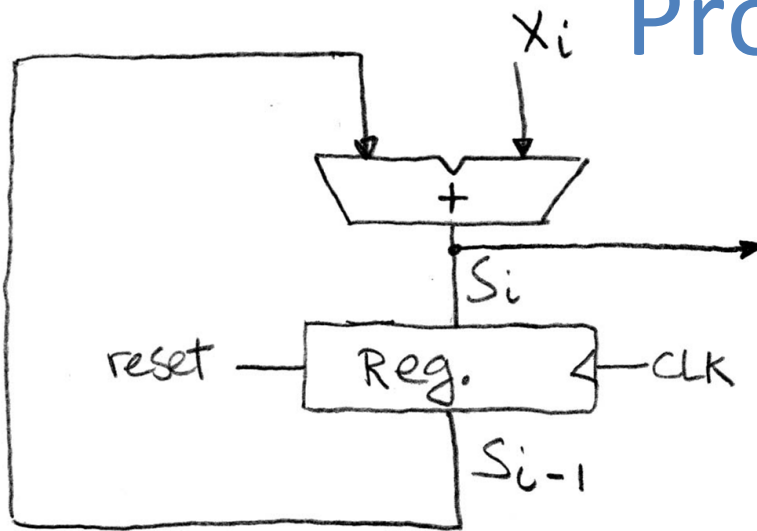


Flip-Flop Timing Behavior

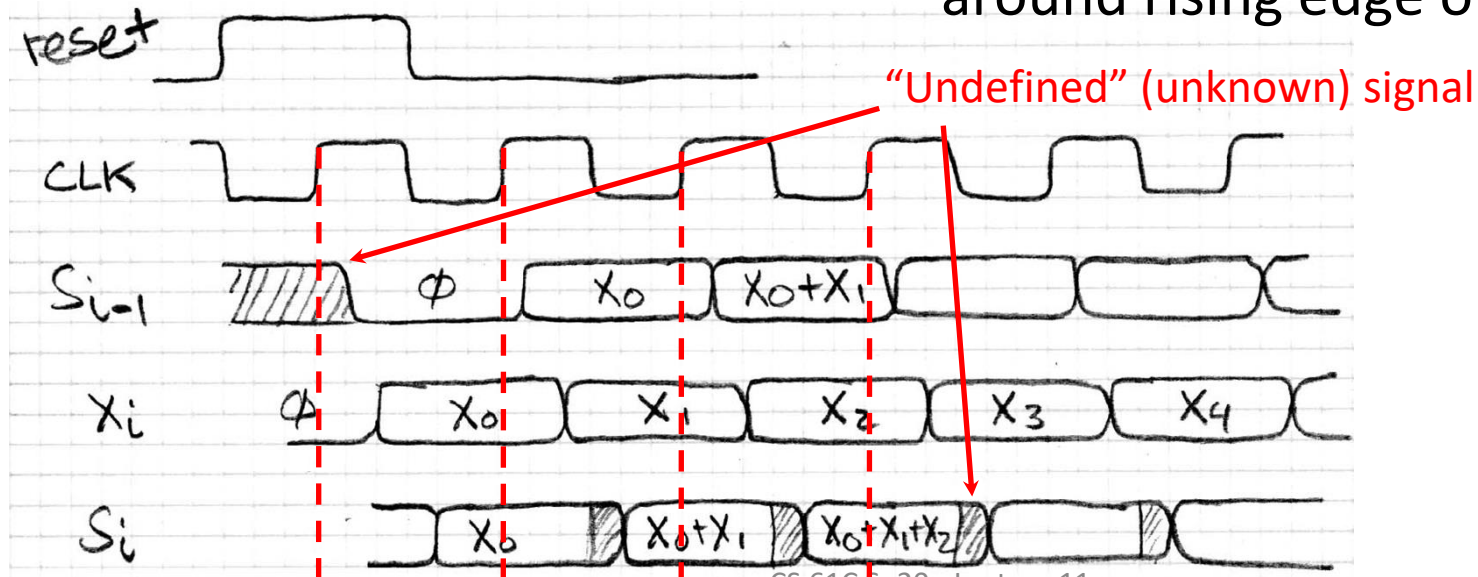


Accumulator Revisited

Proper Timing



- reset signal shown
- In practice X_i might not arrive to the adder at the same time as S_{i-1}
- S_i temporarily is wrong, but register always captures correct value
- In good circuits, instability never happens around rising edge of CLK



Review of Timing Terms

- **Clock:** steady square wave that synchronizes system
- **Flip-flop:** one bit of state that samples every rising edge of Clock (positive edge-triggered)
- **Register:** several bits of state that samples on rising edge of Clock (positive edge-triggered); also has RESET
- **Setup Time:** when input must be stable *before* Clock trigger
- **Hold Time:** when input must be stable *after* Clock trigger
- **Clock-to-Q Delay:** how long it takes output to change from Clock trigger

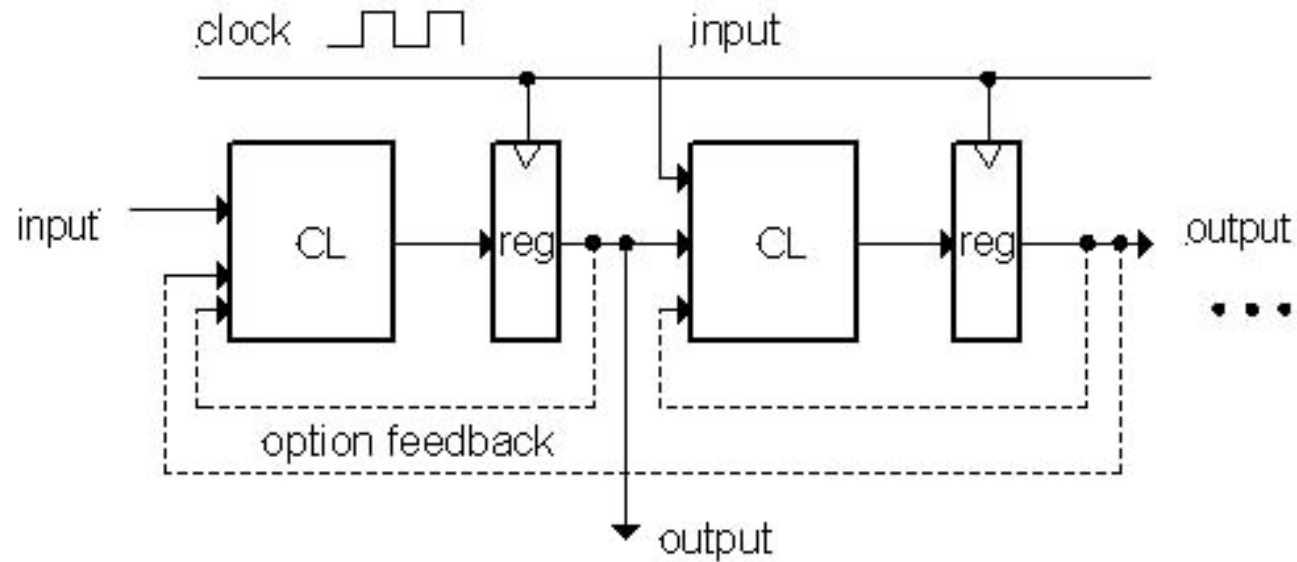
Agenda

- Muxes
- Sequential Logic Timing
- **Maximum Clock Frequency**
- Finite State Machines
- Functional Units
- Summary

Bonus Slides

- Logisim Intro

Model for Synchronous Systems



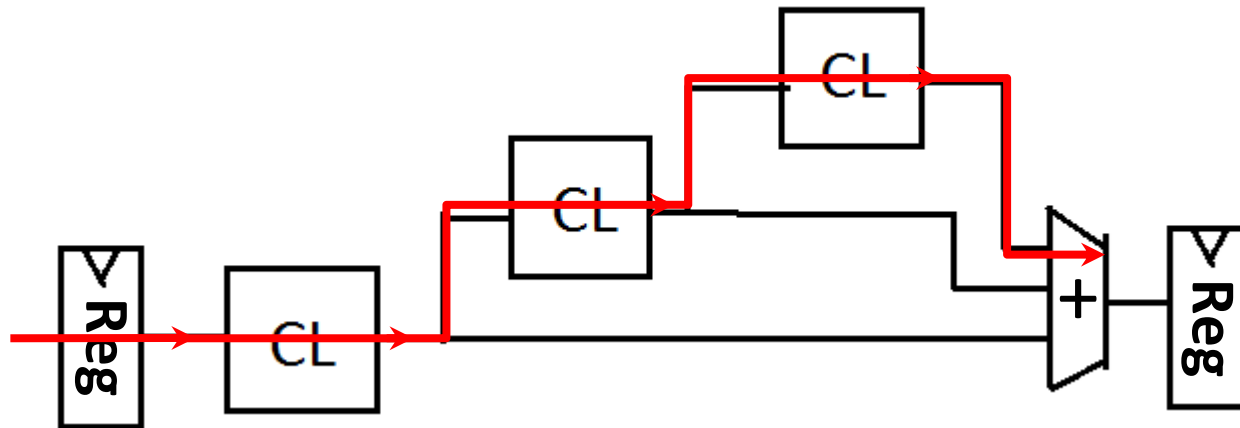
- Combinational logic blocks separated by registers
 - Clock signal connects only to sequential logic elements
 - Feedback is optional depending on application
- How do we ensure proper behavior?
 - How fast can we run our clock?

When can the input change?

- Needs to be stable for duration of setup time + hold time
- Often unstable until at least clock-to-q time has passed
 - Because register output isn't ready yet
- Needs to account for all combinational logic delay too

The Critical Path

- The *critical path* is the longest delay between *any* two registers in a circuit
- The clock period must be *longer* than this critical path, or the signal will not propagate properly to that next register

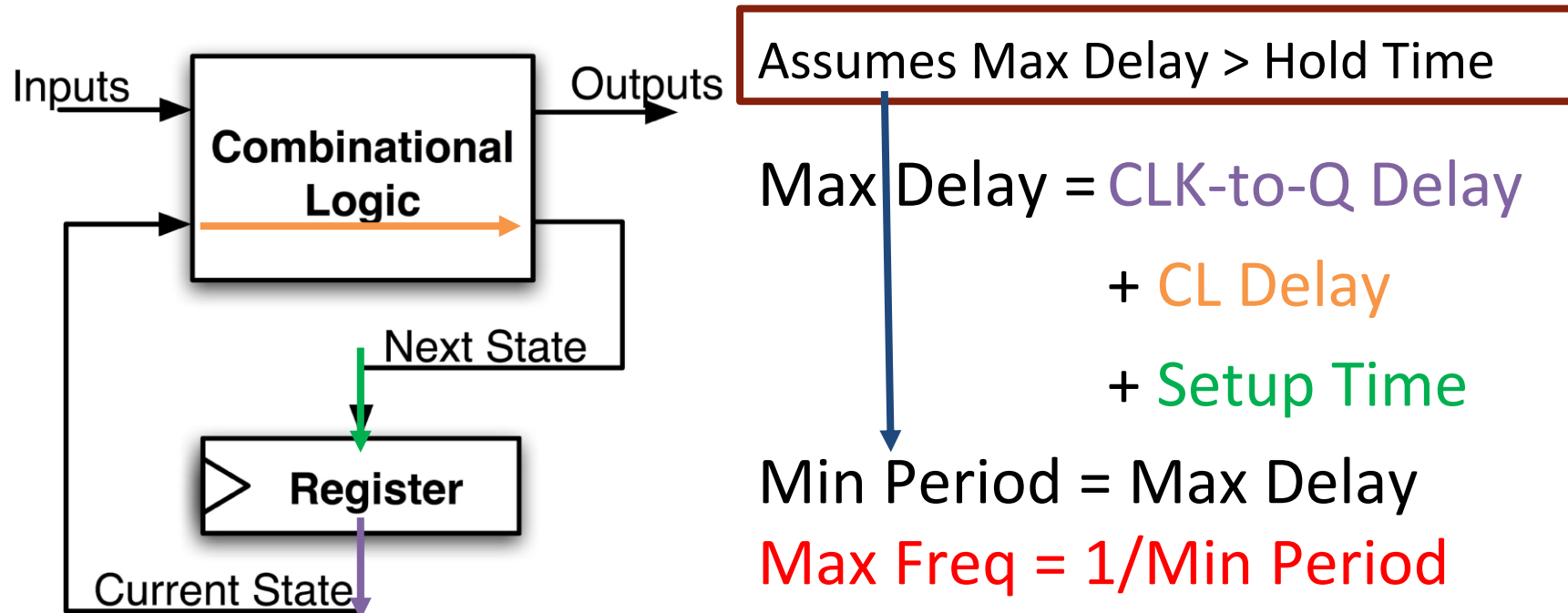


Critical Path =

CLK-to-Q Delay
+ CL Delay 1
+ CL Delay 2
+ CL Delay 3
+ Adder Delay
+ Setup Time

Maximum Clock Frequency

- What is the max frequency of this circuit?
 - Limited by how much time needed to get correct Next State to Register (t_{setup} constraint)



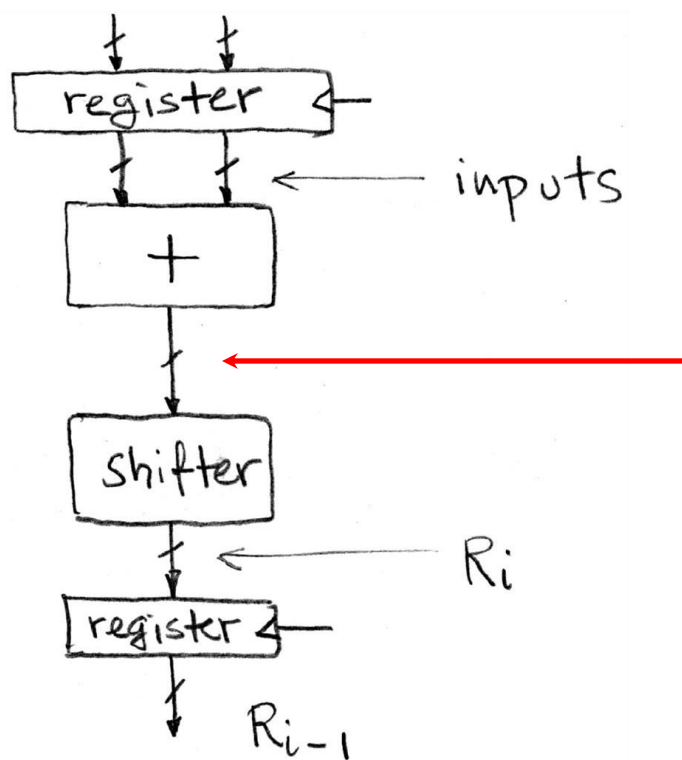
How do we go faster?

Pipelining!

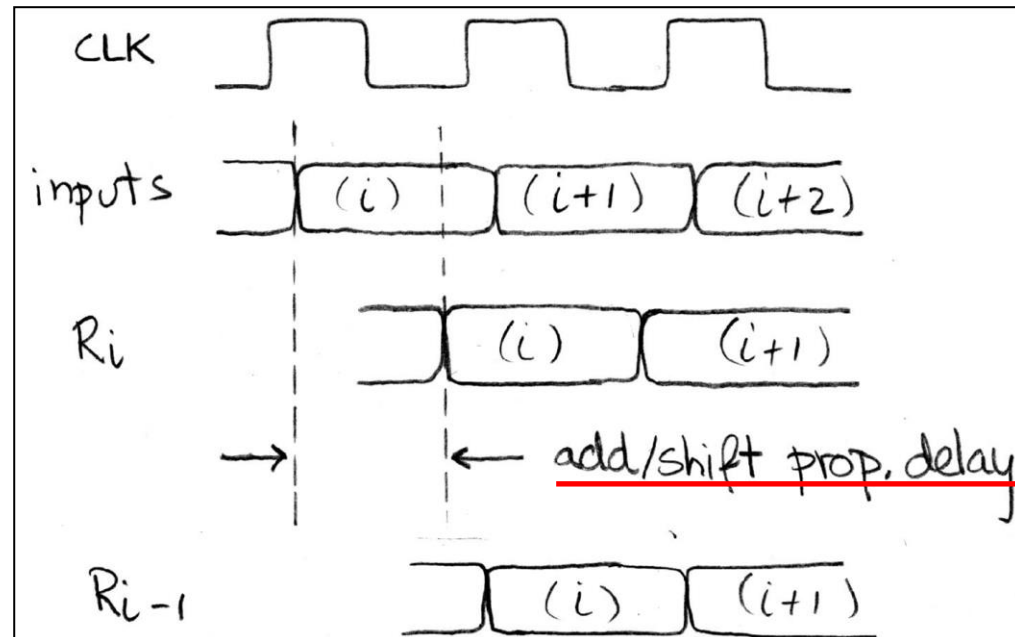
- Split operation into smaller parts and add a register between each one.

Pipelining and Clock Frequency (1/2)

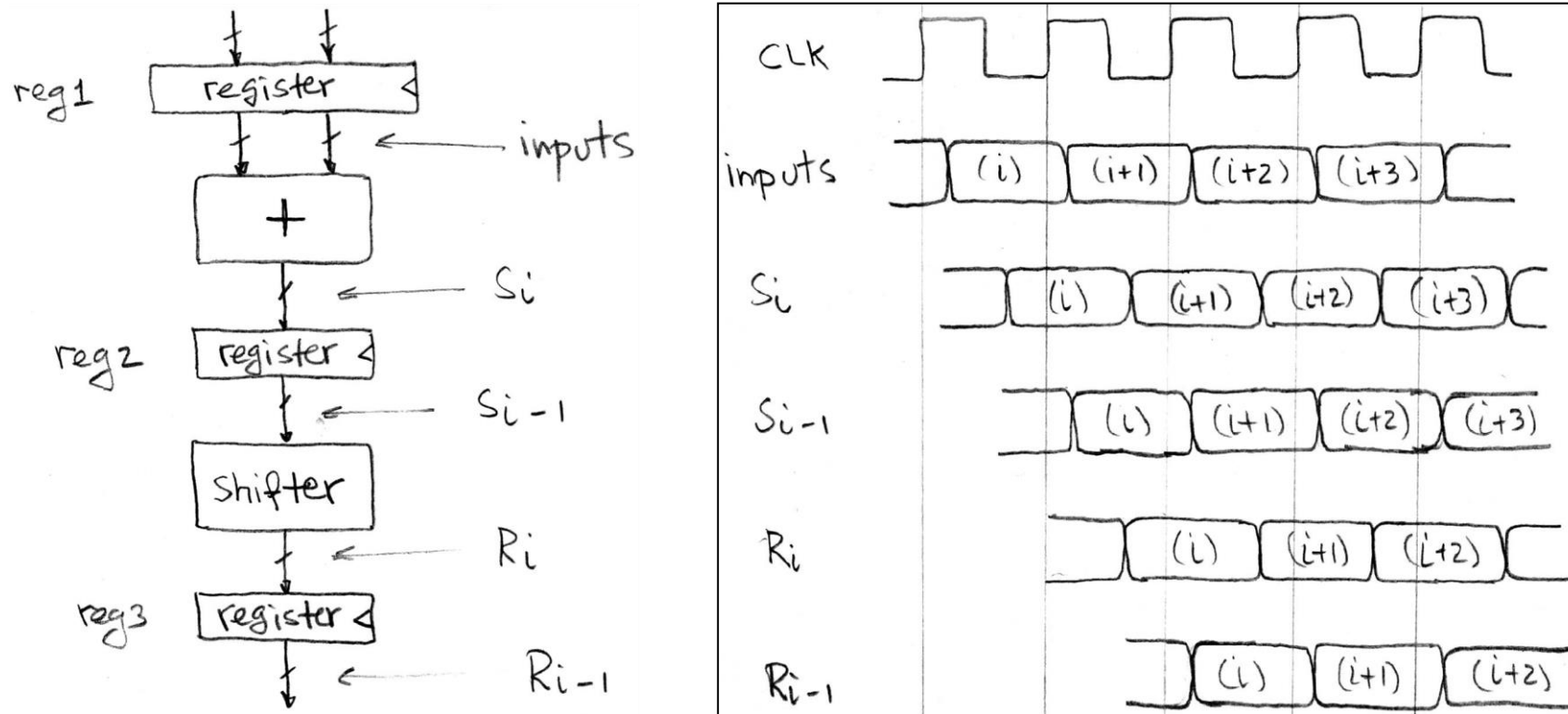
- Clock period limited by propagation delay of adder and shifter
 - Add an extra register to reduce the critical path!



Timing:



Pipelining and Clock Frequency (2/2)



- Reduced critical path \rightarrow allows higher clock freq.
- Extra register \rightarrow extra (shorter) cycle to produce first output

A Pipelining Analogy

-Thanks to student Samm Du

Imagine putting out a fire with buckets

- One person could carry the bucket all the way from a pond to the fire

OR

- A line of people could hand buckets off all the way to the house

The time for the first bucket gets longer with the line of people. But future buckets come WAY faster

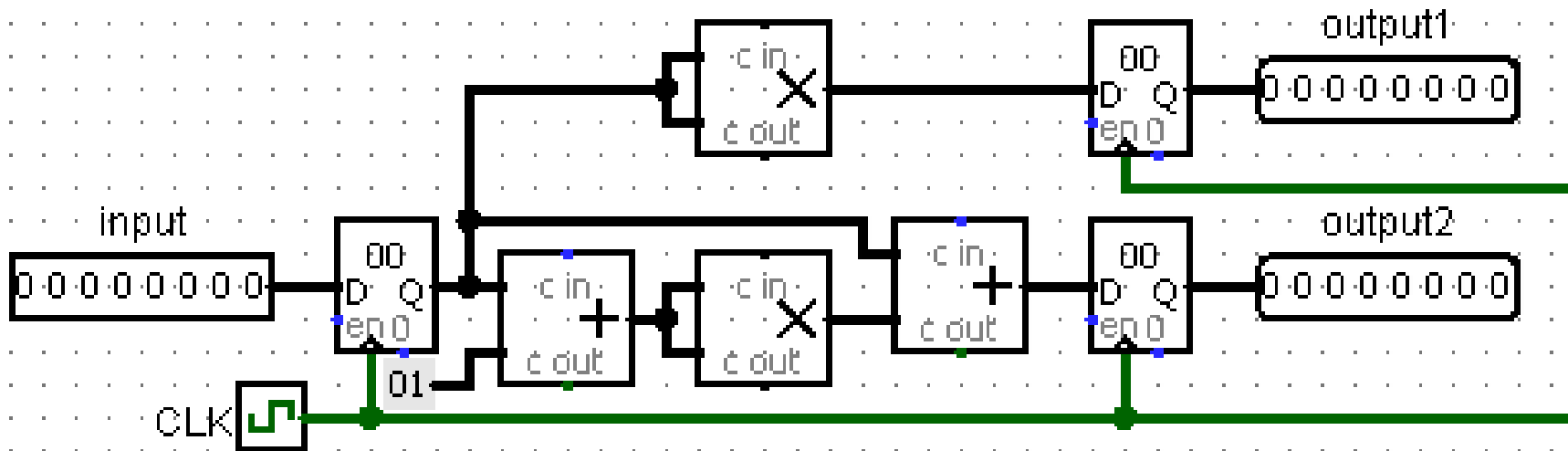
Pipelining Basics

- By adding more registers, break path into shorter “stages”
 - Aim is to reduce critical path
 - Signals take an additional clock cycle to propagate through *each* stage
- New critical path must be calculated
 - Affected by placement of new pipelining registers
 - Faster clock rate → higher throughput (outputs)
 - More stages → higher startup latency
- **Pipelining tends to improve performance**
 - More on this (and application to CPUs) later

Question: Want to run on 1 GHz processor.

$t_{\text{add}} = 100 \text{ ps}$. $t_{\text{mult}} = 200 \text{ ps}$. $t_{\text{setup}} = t_{\text{hold}} = 50 \text{ ps}$.

What is the maximum clock-to-q time?

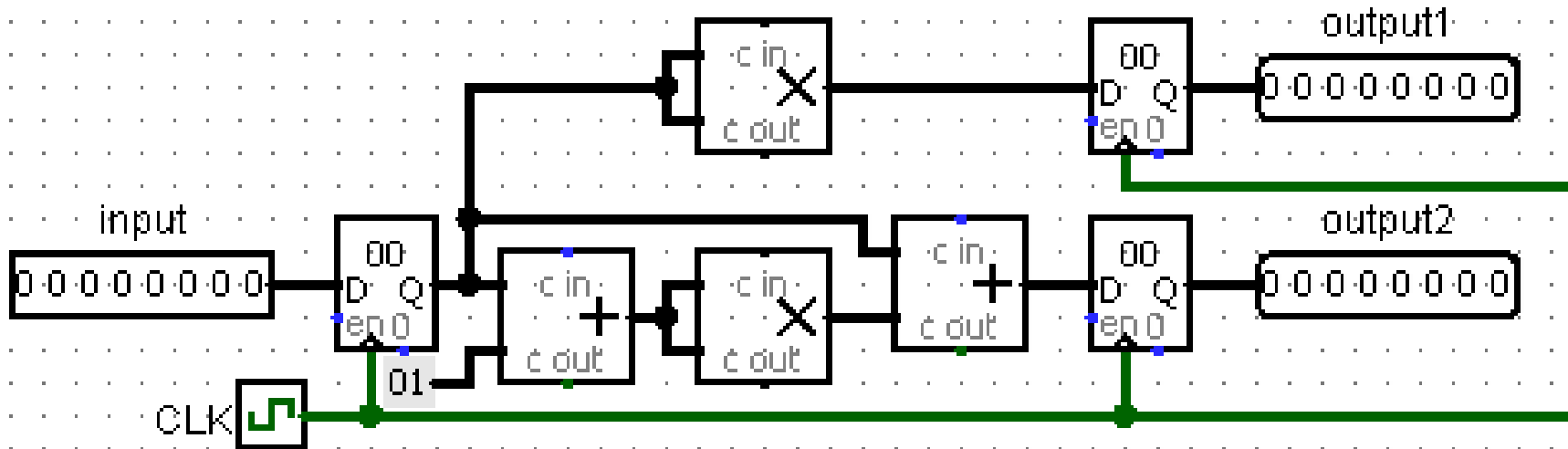


- (A) 550 ps
- (B) 750 ps
- (C) 500 ps
- (D) 700 ps

Question: Want to run on 1 GHz processor.

$t_{\text{add}} = 100 \text{ ps}$. $t_{\text{mult}} = 200 \text{ ps}$. $t_{\text{setup}} = t_{\text{hold}} = 50 \text{ ps}$.

What is the maximum clock-to-q time?



(A) 550 ps

(B) 750 ps

(C) 500 ps

(D) 700 ps

Bottom path is critical path:

clock-to-q + 100 + 200 + 100 + 50 < 1000 ps = 1ns

clock-to-q + 450 < 1000 ps

clock-to-q < 550

Agenda

- Muxes
- Sequential Logic Timing
- Maximum Clock Frequency
- **Finite State Machines**
- Functional Units
- Summary

Bonus Slides

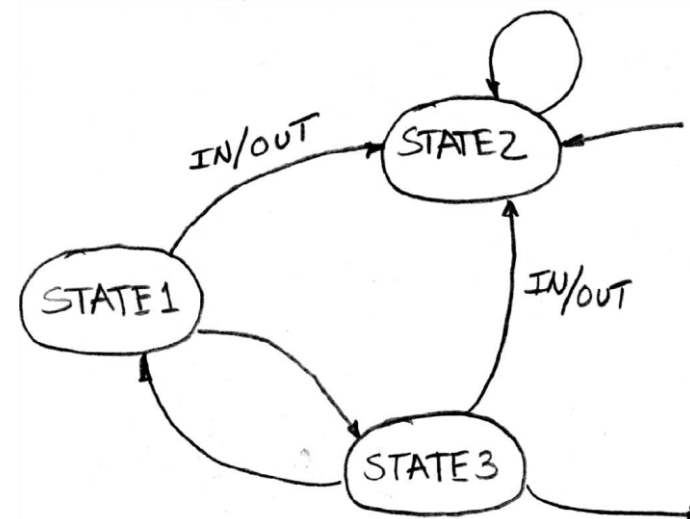
- Logisim Intro

Back to representations

- How do we represent sequential logic?
 - Truth tables could account for history
 - We could do boolean logic with prior state as a variable
- We can also use a new representation:
Finite State Machines

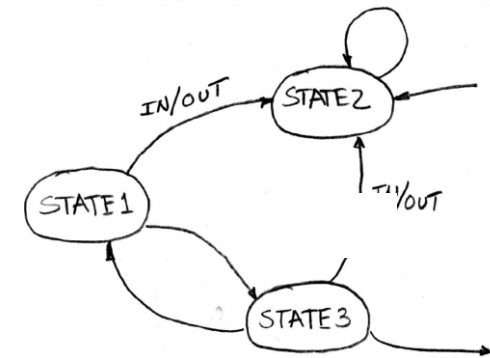
Finite State Machines (FSMs)

- A convenient way to conceptualize computation over time
- Function can be represented with a *state transition diagram*
- With combinational logic and registers, any FSM can be implemented in hardware!



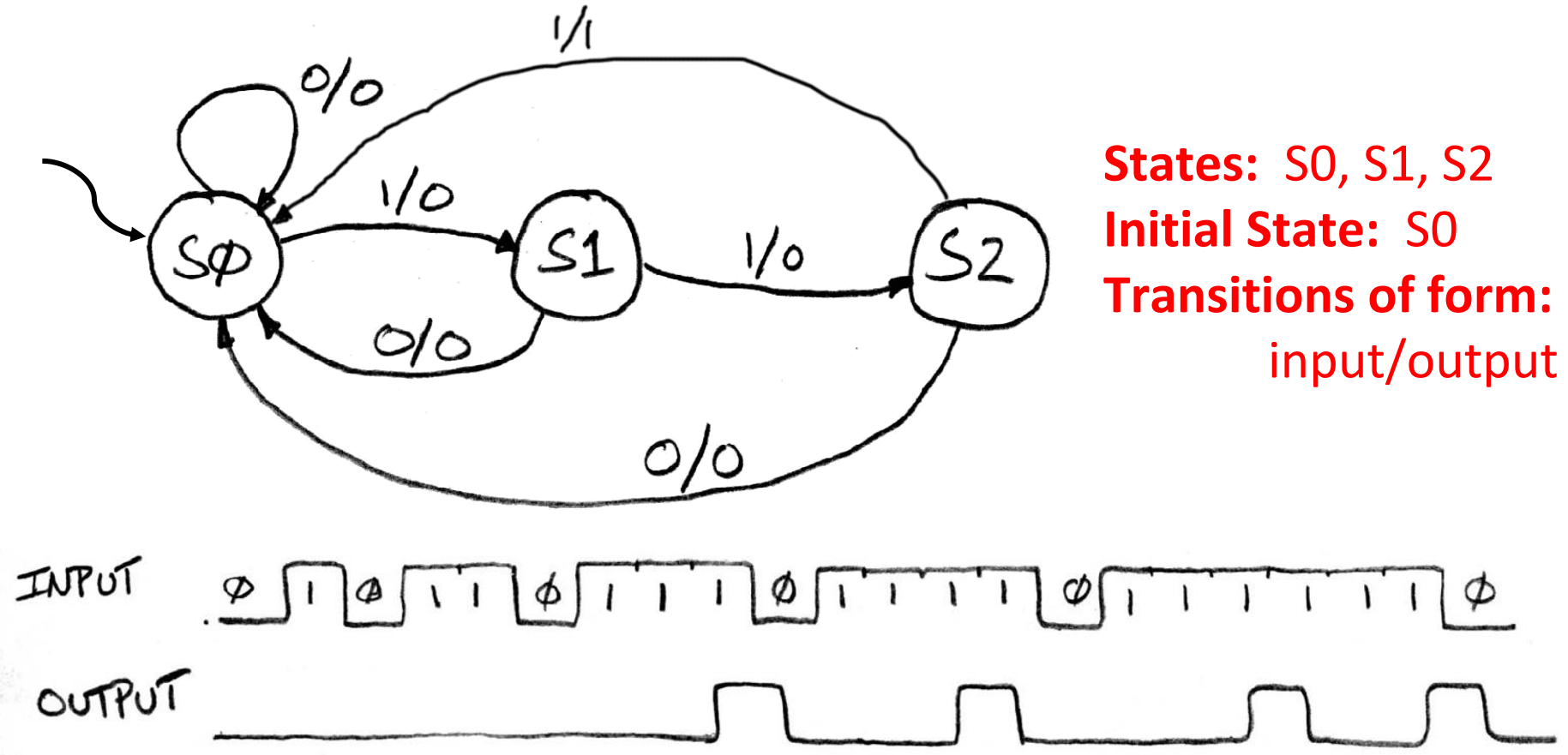
FSM Overview

- An FSM (in this class) is defined by:
 - A set of *states* S (circles)
 - An *initial state* s_0 (only arrow not between states)
 - A *transition function* that maps from the current input and current state to the output and the next state (arrows between states)
- State transitions are controlled by the clock:
 - On each clock cycle the machine checks the inputs and generates a new state (could be same) and new output



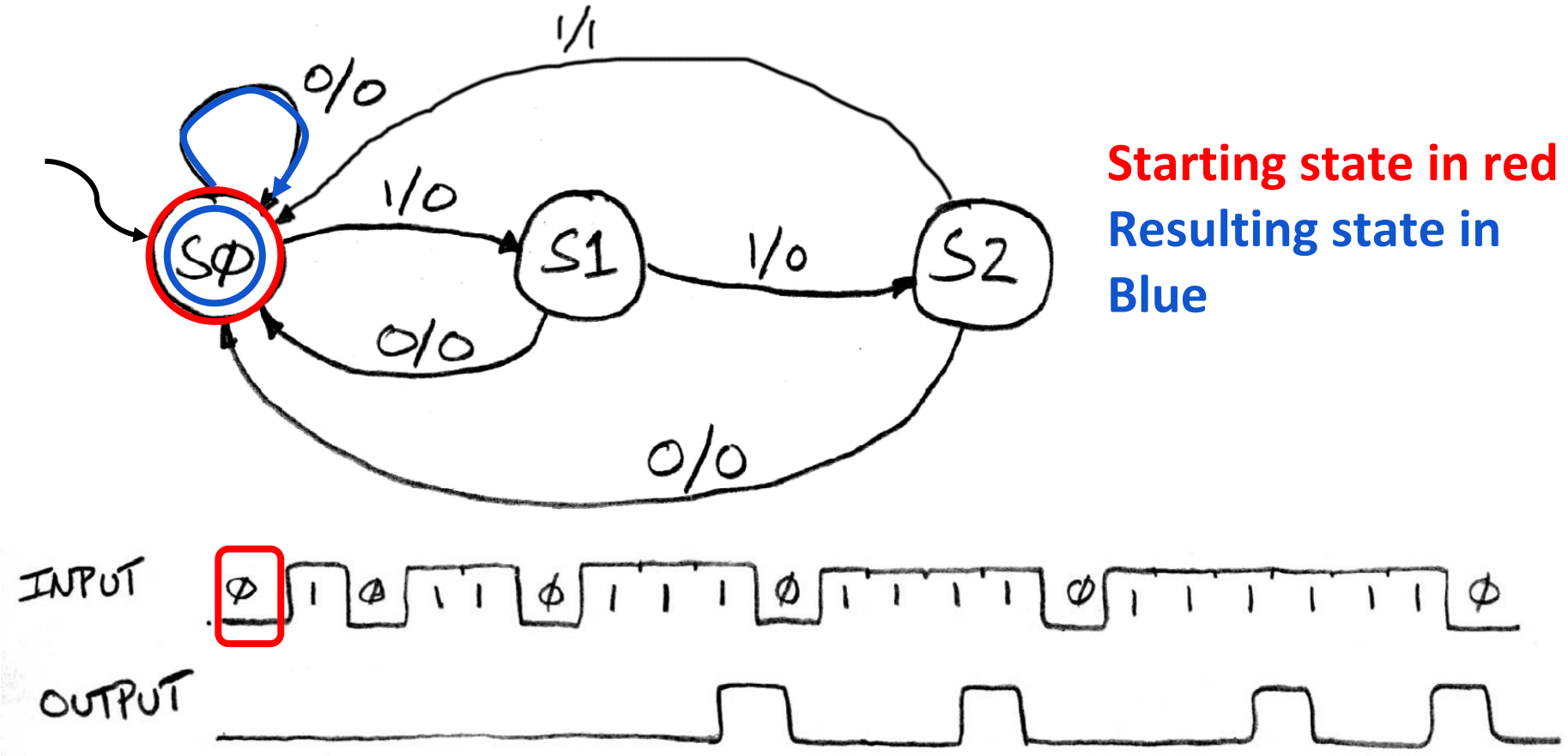
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

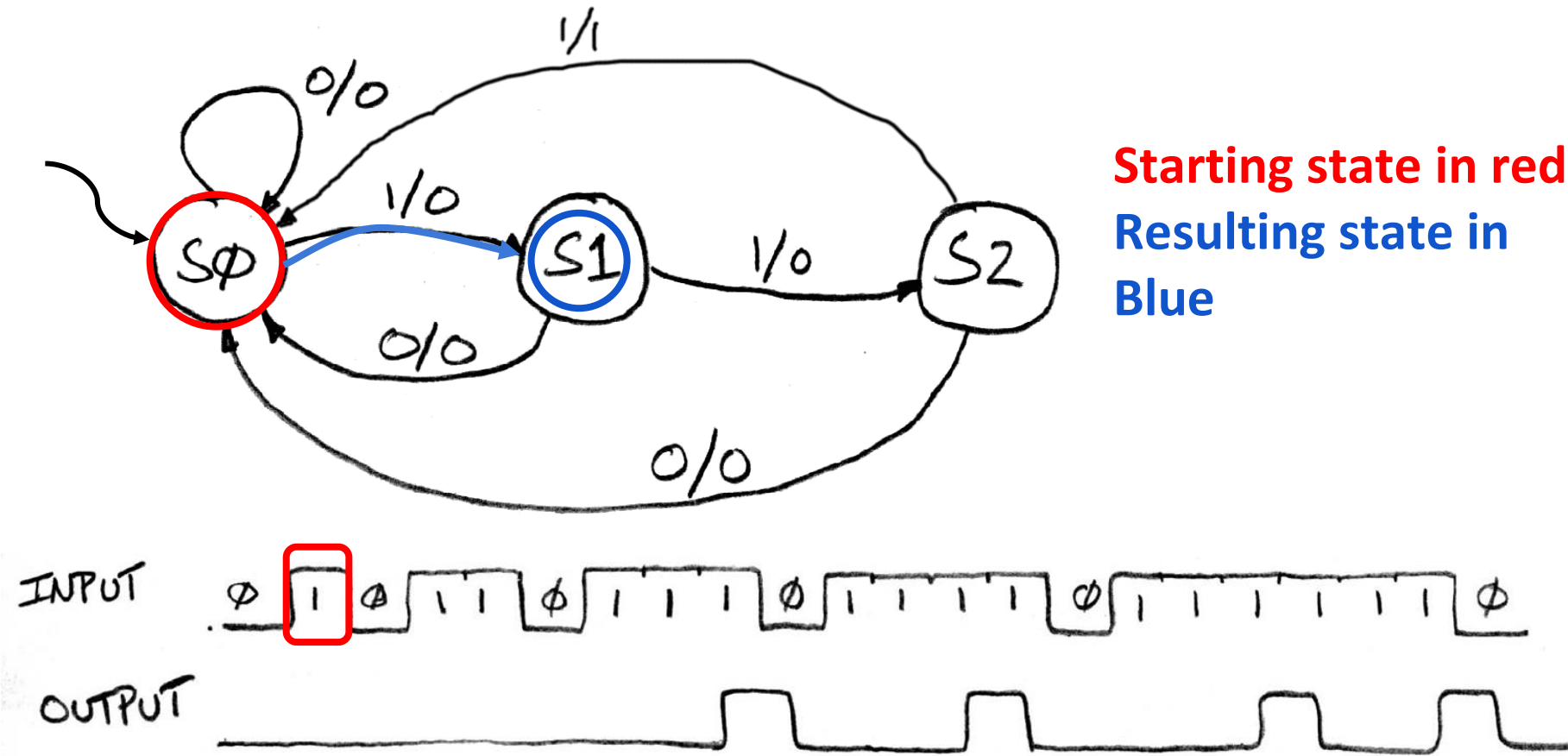


Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input

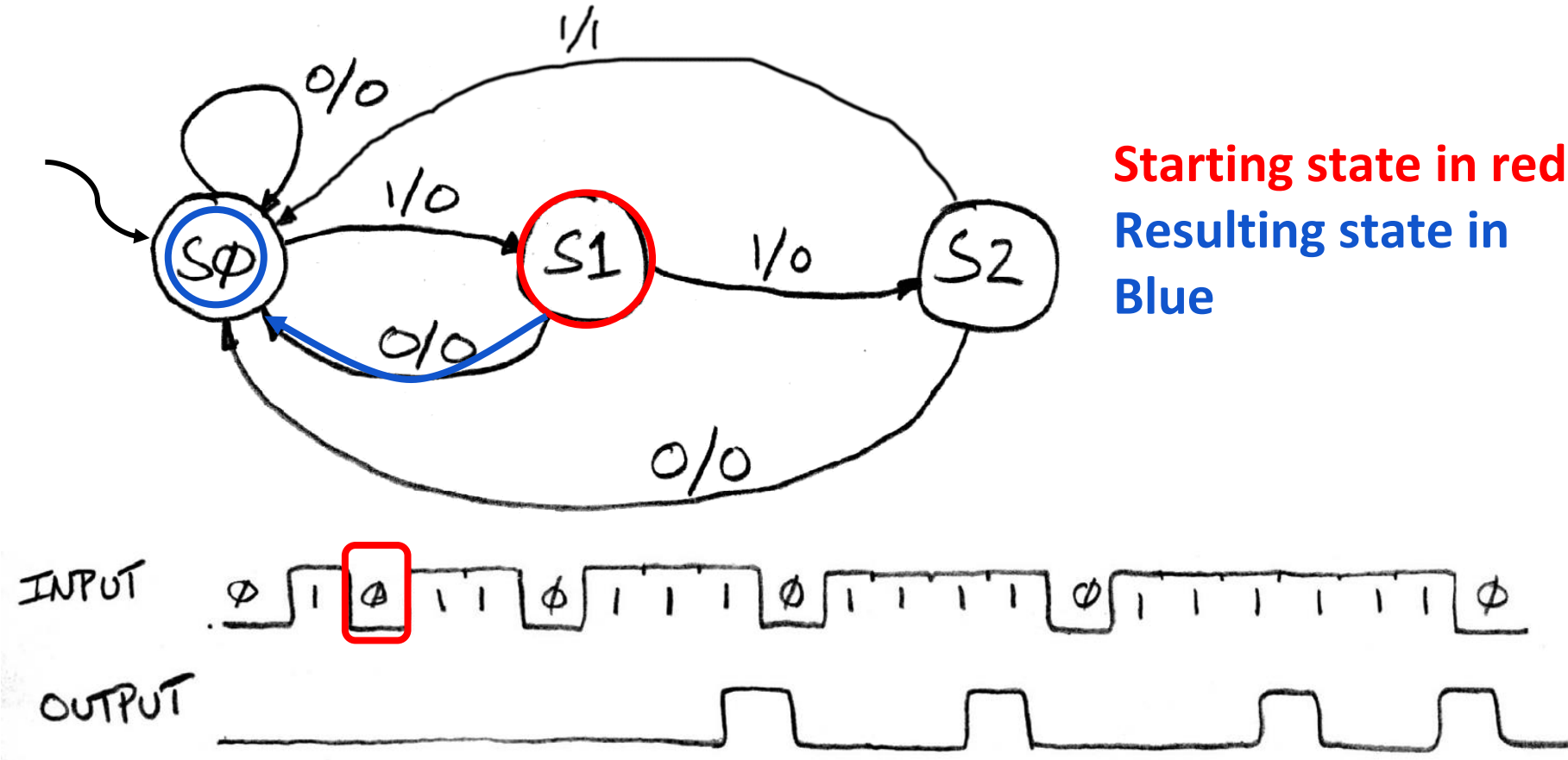


Example: 3 Ones FSM



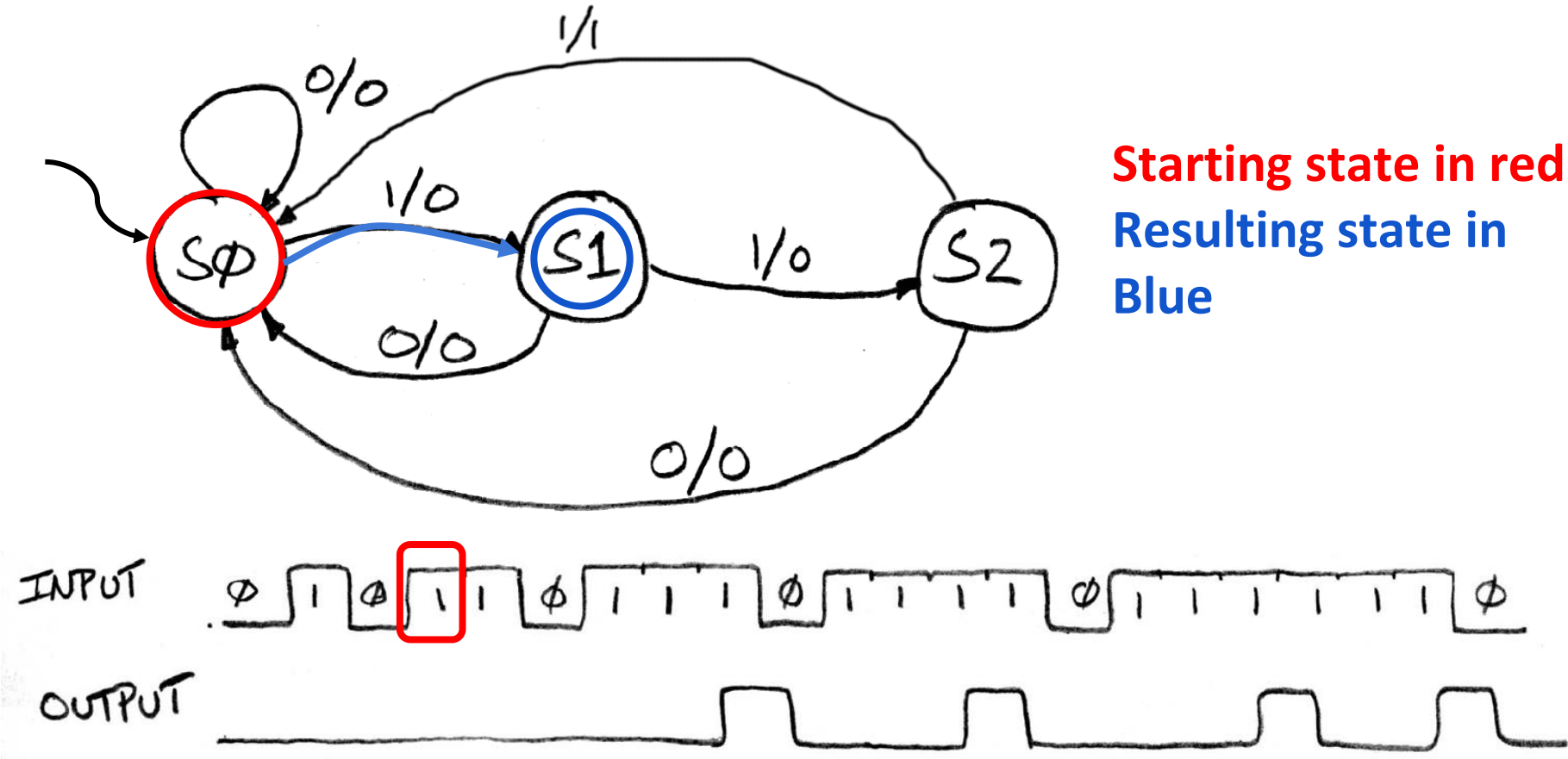
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



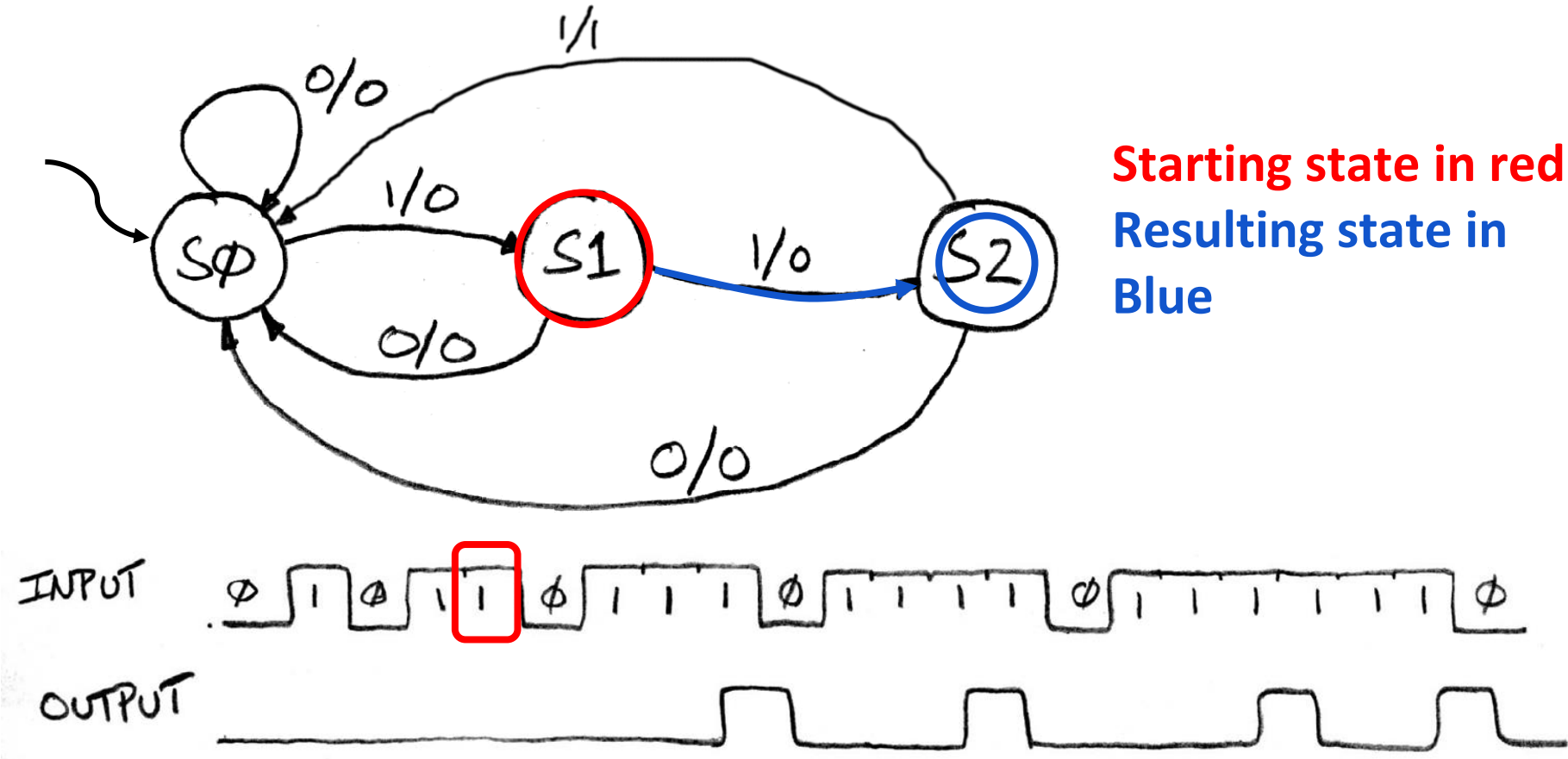
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



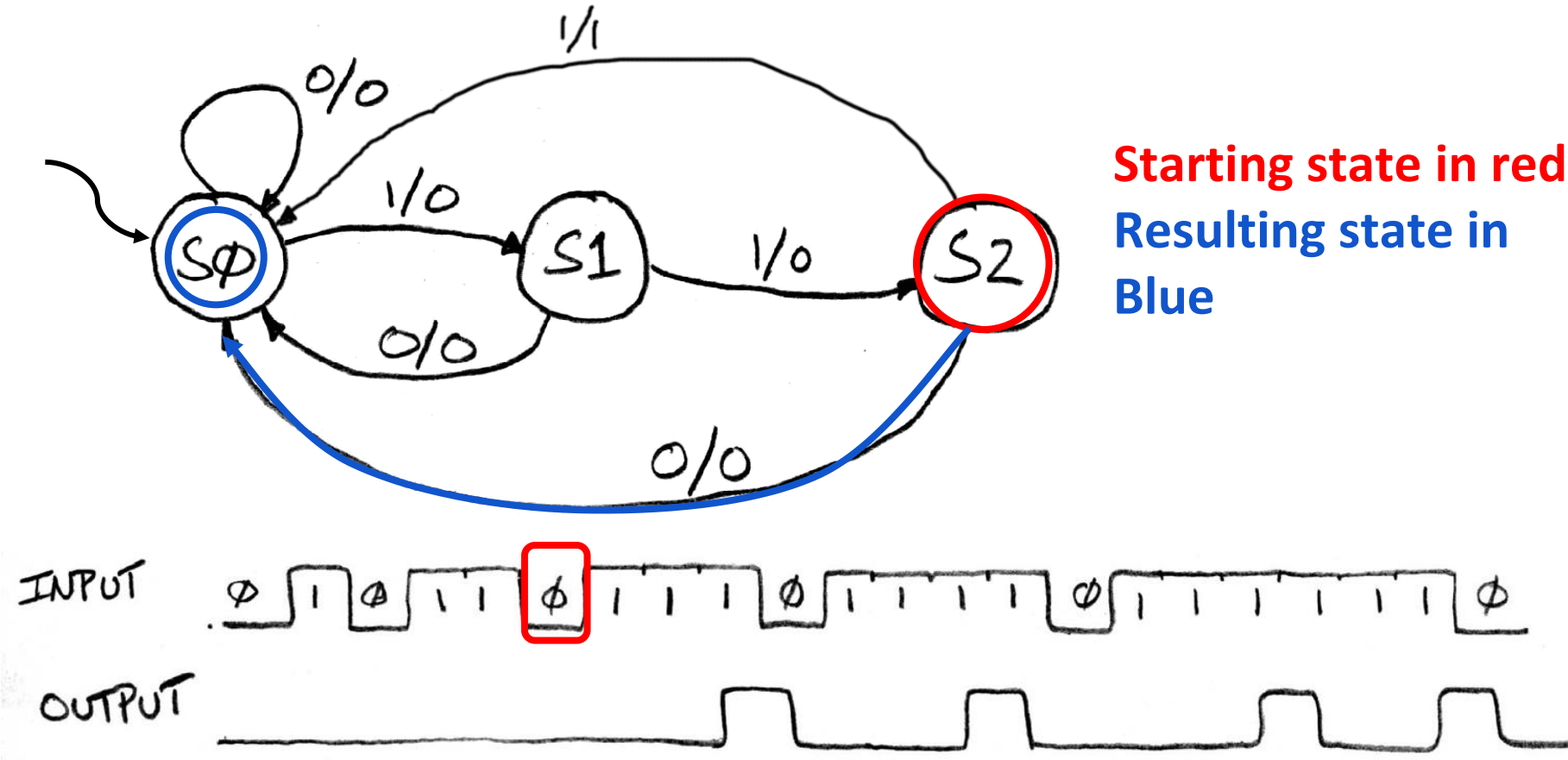
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



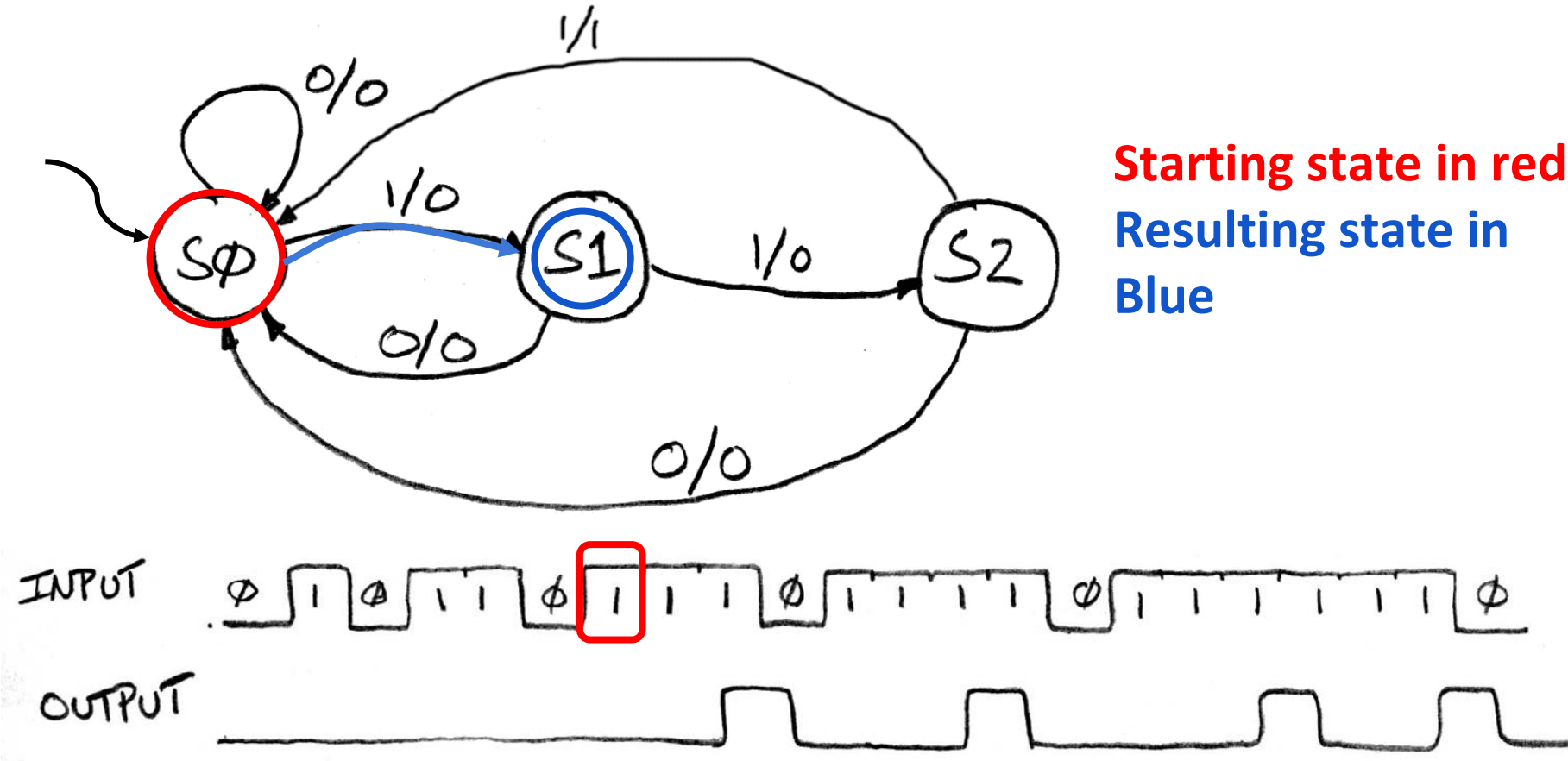
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



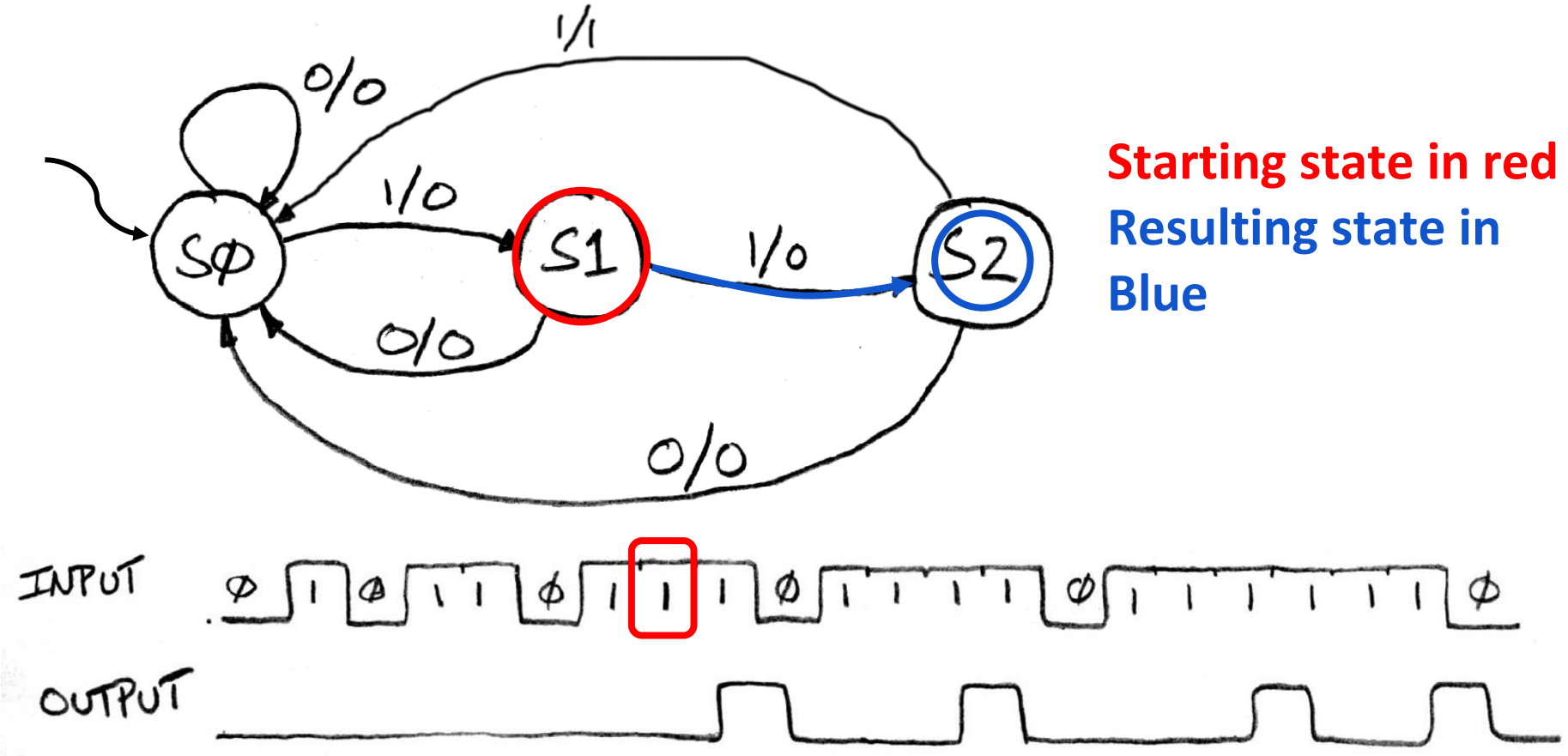
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



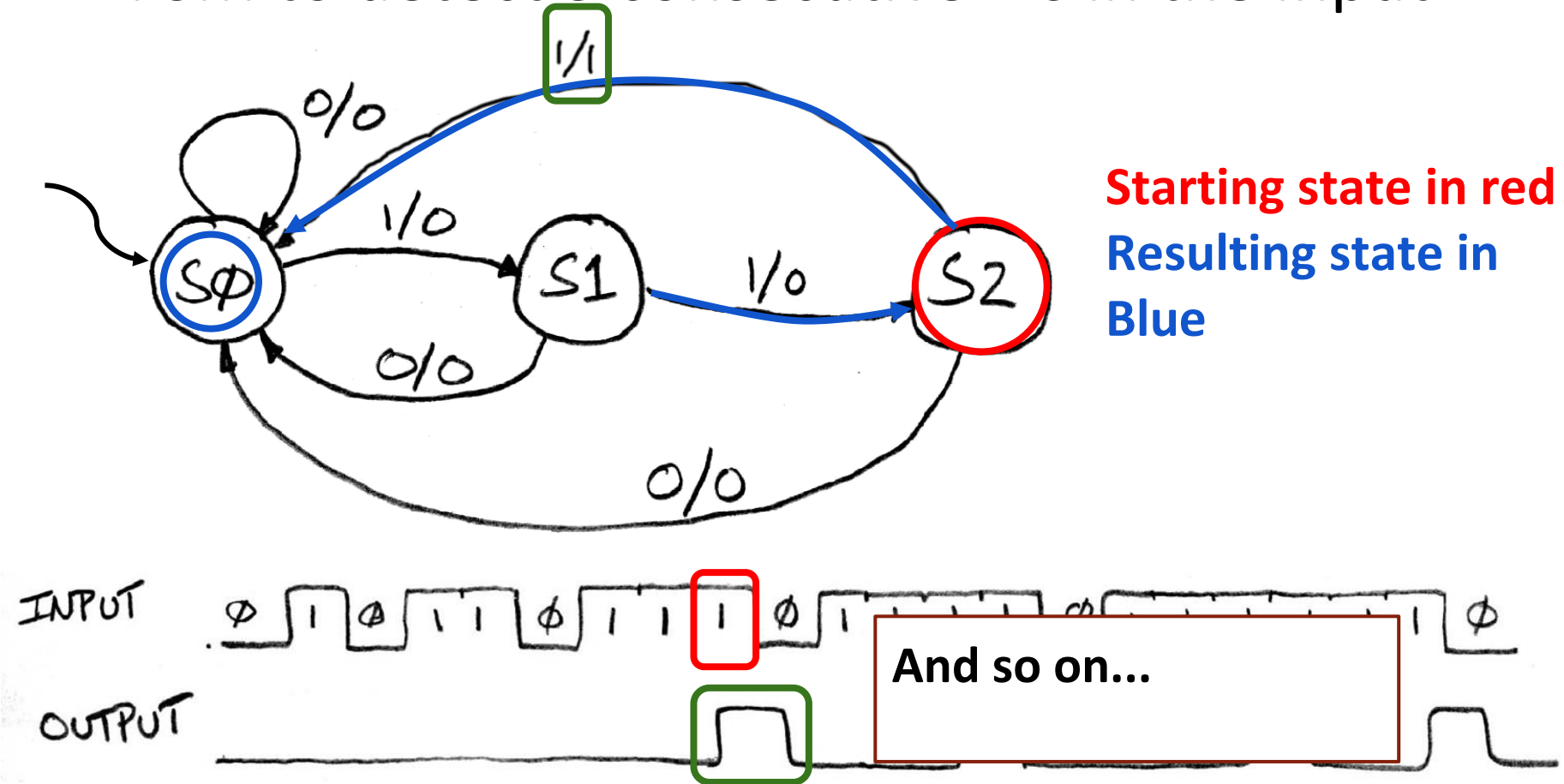
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



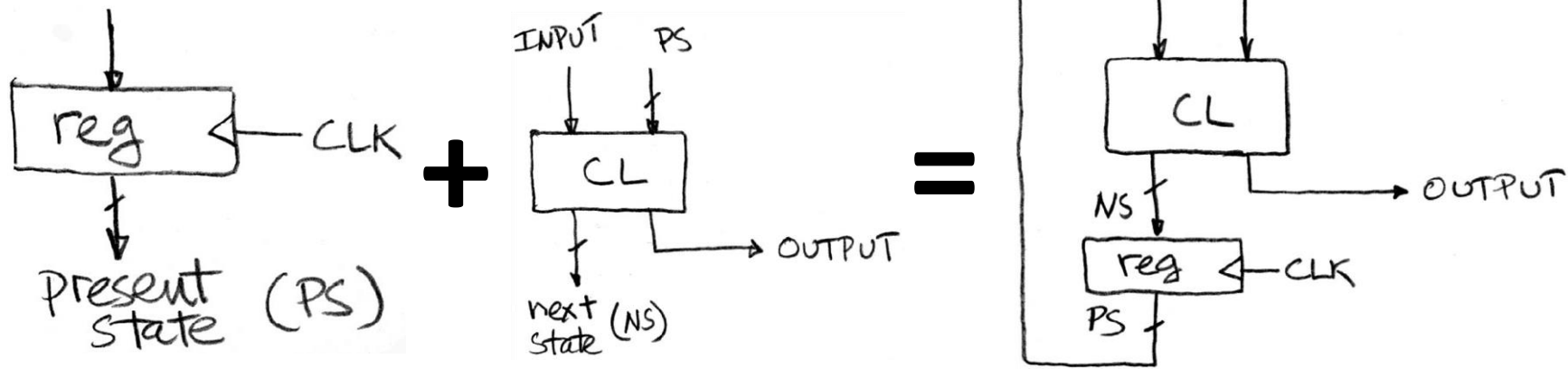
Example: 3 Ones FSM

- FSM to detect 3 consecutive 1's in the Input



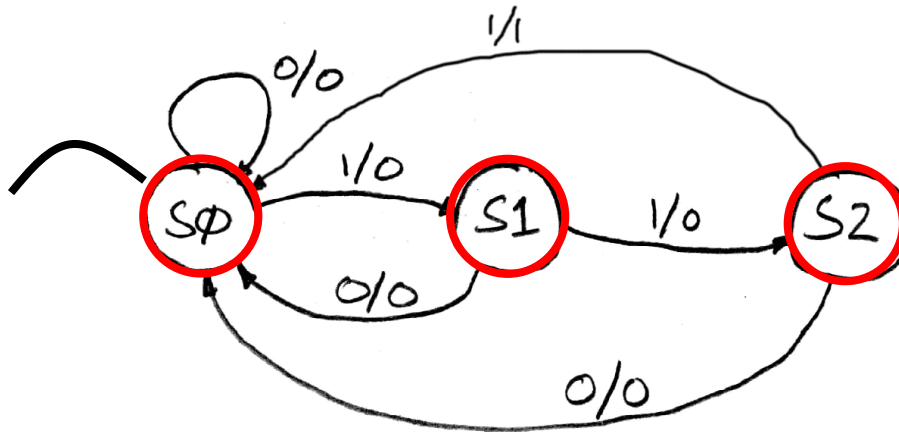
Hardware Implementation of FSM

- Register holds a representation of the FSM's state
 - Must assign a *unique* bit pattern for each state
 - Output is *present/current state* (PS/CS)
 - Input is *next state* (NS)
- Combinational Logic implements transition function (state transitions + output)



FSM: Combinational Logic

- Read off transitions into Truth Table!
 - **Inputs:** Current State (CS) and Input (In)
 - **Outputs:** Next State (NS) and Output (Out)



CS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

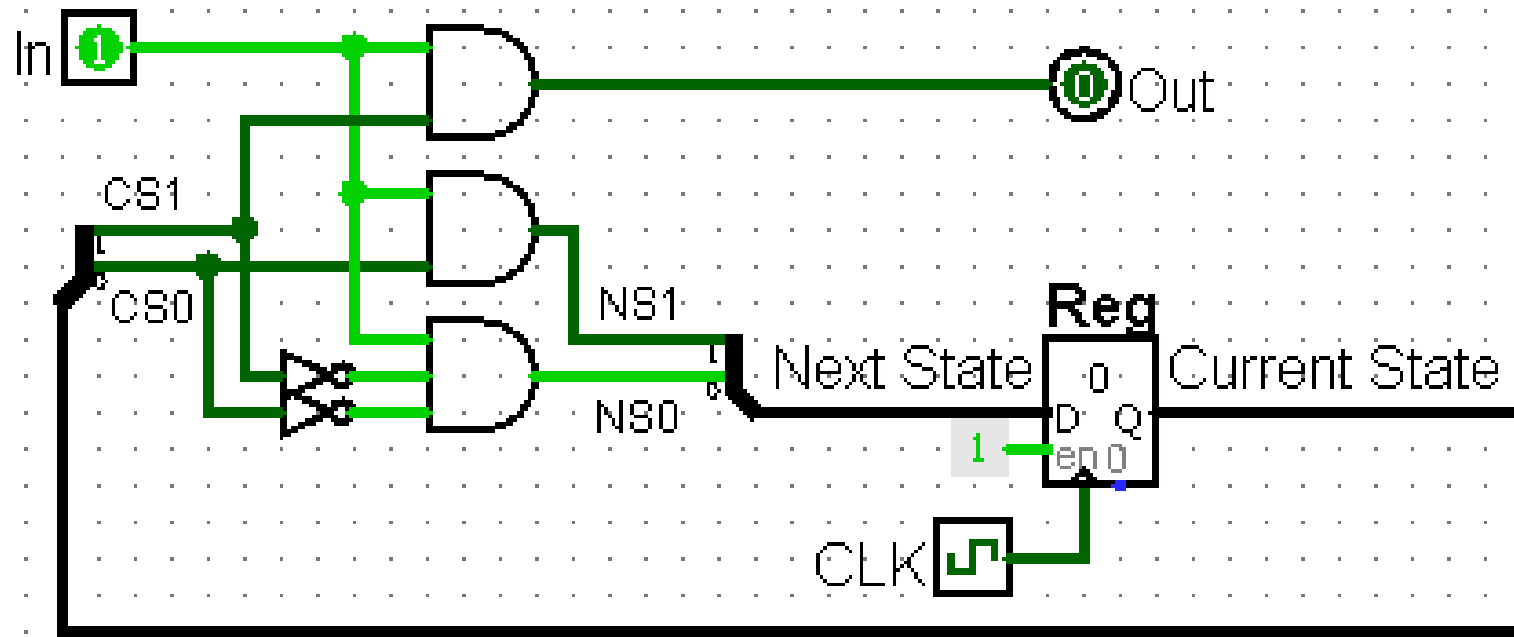
Unspecified Output Values

- Our FSM has only 3 states
 - 2 entries in truth table are undefined/unspecified
- Use symbol 'X' to mean it can be either a 0 or 1
 - Make choice to simplify final expression
 - Any choice is correct

CS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1
11	0	XX	X
11	1	XX	X

3 Ones FSM in Hardware

- 2-bit **Register** needed for state
- **CL:** $NS_1 = CS_0In$, $NS_0 = \neg CS_1 \neg CS_0In$, $Out = CS_1In$



Agenda

- Muxes
- Sequential Logic Timing
- Maximum Clock Frequency
- Finite State Machines
- **Functional Units**
- Summary

Bonus Slides

- Logisim Intro

Functional Units (a.k.a. Execution Unit)

- Now that we know sequential logic, we can explore some pieces of a processor!
- Functional Units are a part of the processor that perform operations and calculations based on the running program
 - Arithmetic Logic Unit
 - Floating Point Unit
 - Load/Store Unit
 - and several more...

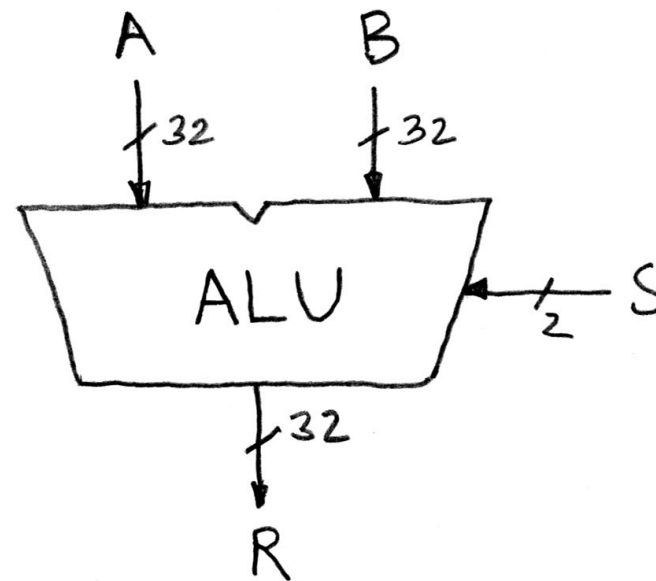
Invented by *John Von Neumann*

He also invented

- Stored Program Concept
- Mergesort
- Mutually Assured Destruction

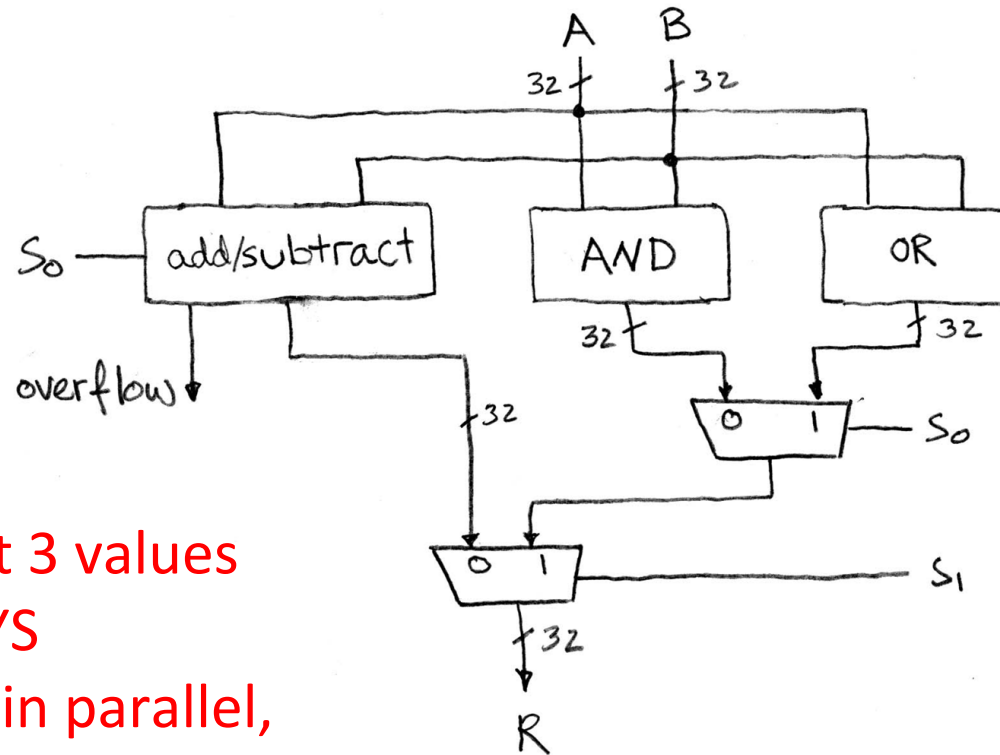
Arithmetic Logic Unit (ALU)

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



when $S=00$, $R = A + B$
when $S=01$, $R = A - B$
when $S=10$, $R = A \text{ AND } B$
when $S=11$, $R = A \text{ OR } B$

Simple ALU Schematic



Notice that 3 values
are ALWAYS
calculated in parallel,
but only 1 makes it to
the Result

Adder/Subtractor Design

- 1) Combinational Logic design we've seen before:
 - write out truth table
 - convert to boolean logic
 - minimize logic
 - then implement


How big might truth table and/or Boolean expression get?

Adder/Subtractor Design

- 2) Break down the problem into smaller pieces that we can cascade or hierarchically layer
 - Let's try this approach instead

Adder/Subtractor: 1-bit LSB Adder

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

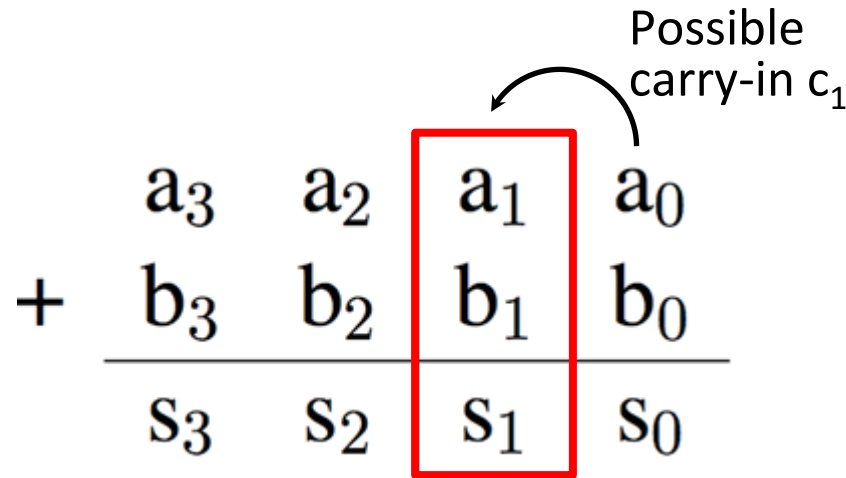
Carry-out bit 

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

$$s_0 = a_0 \text{ XOR } b_0$$

$$c_1 = a_0 \text{ AND } b_0$$

Adder/Subtractor: 1-bit Adder



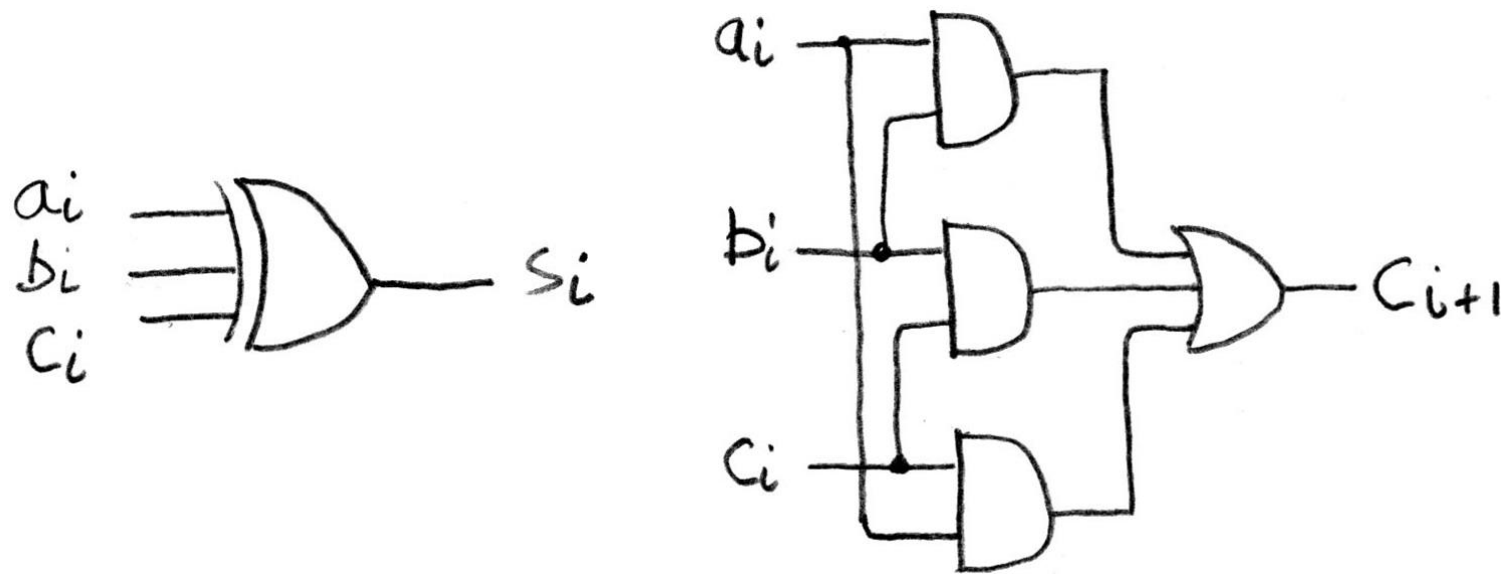
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Here defining XOR of many inputs to be 1 when an *odd* number of inputs are 1

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

Adder/Subtractor: 1-bit Adder

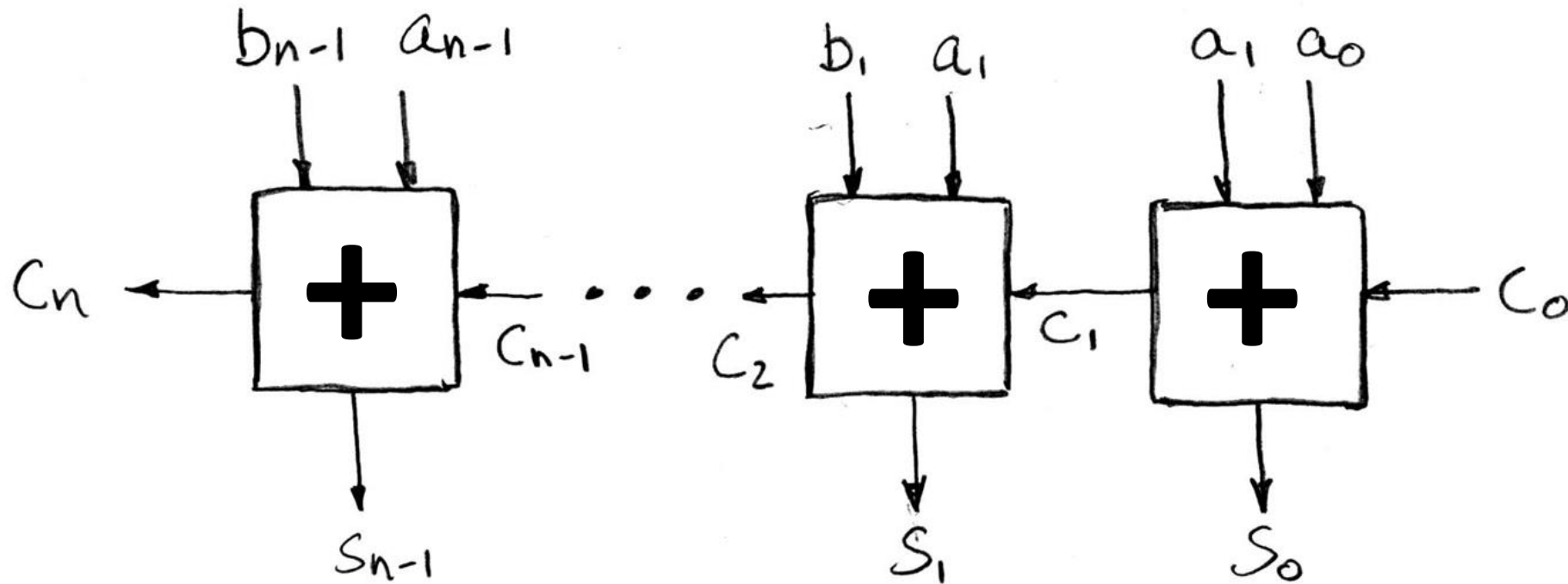
- Circuit Diagrams:**



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

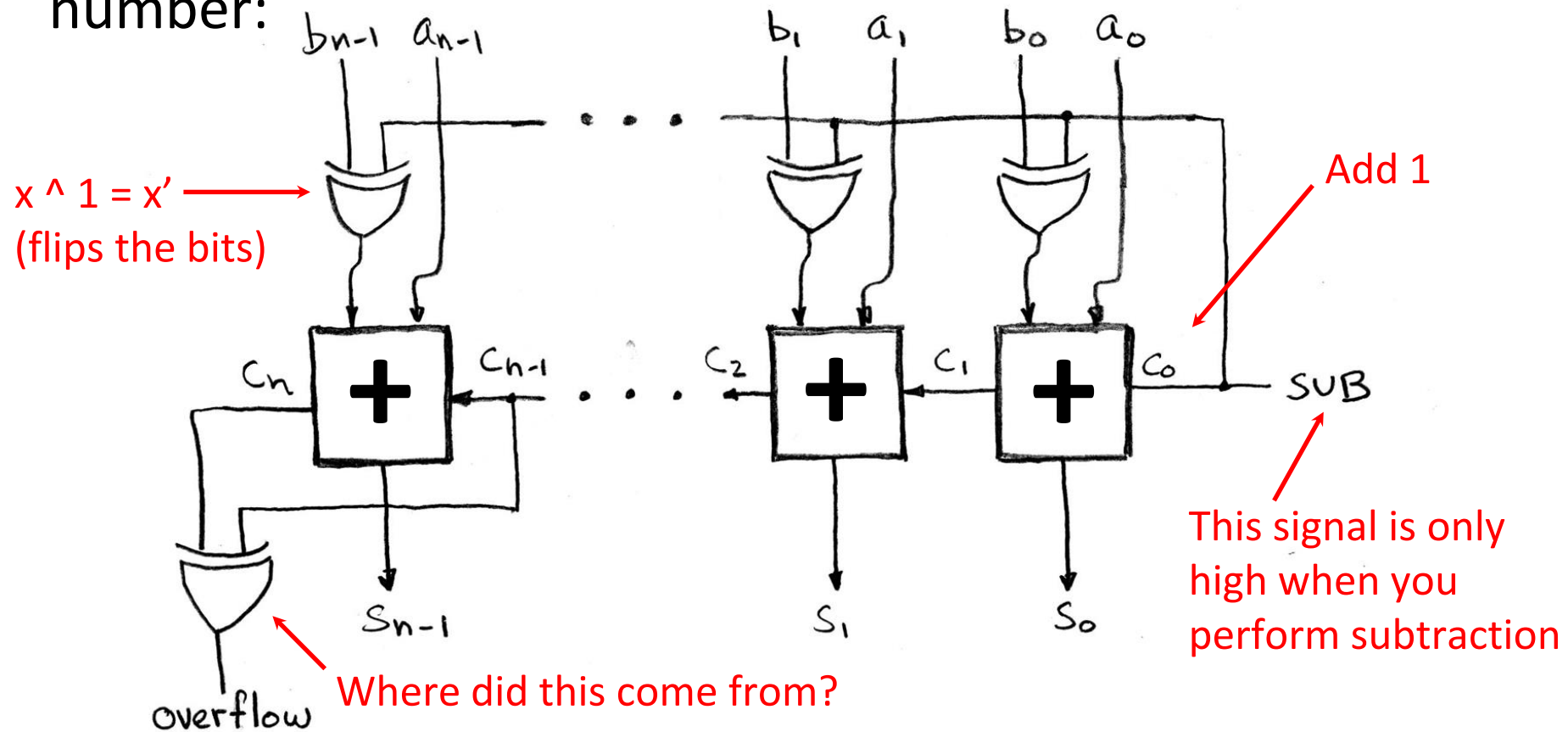
N x 1-bit Adders \rightarrow N-bit Adder

- Connect CarryOut_{i-1} to CarryIn_i to chain adders:



Two's Complement Adder/Subtractor

- Subtraction accomplished by adding negated number:



Detecting Overflow

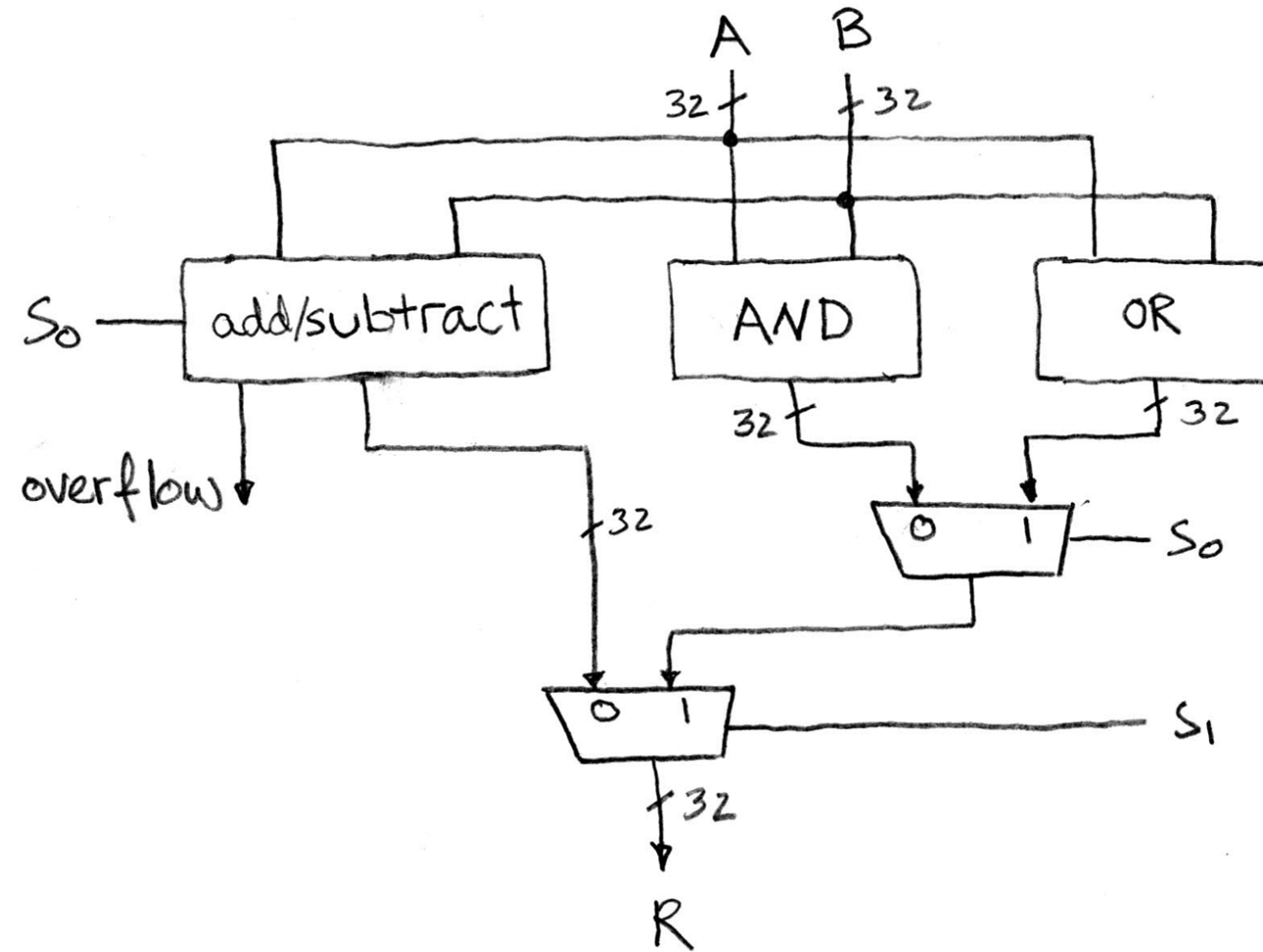
- Unsigned overflow
 - On addition, if carry-out from MSB is 1
 - On subtraction, if carry-out from MSB is 0
 - This case is a lot harder to see than you might think
- Signed overflow
 - 1) Overflow from adding “large” positive numbers
 - 2) Overflow from adding “large” negative numbers

Signed Overflow Examples (4-bit)

- Overflow from two positive numbers:
 - $0111 + 0111$, $0111 + 0001$, $0100 + 0100$.
 - Carry-out from the 2nd MSB (but not MSB)
 - $\text{pos} + \text{pos} \neq \text{neg}$
- Overflow from two negative numbers:
 - $1000 + 1000$, $1000 + 1111$, $1011 + 1011$.
 - Carry-out from the MSB (but not 2nd MSB)
 - $\text{neg} + \text{neg} \neq \text{pos}$
- Expression for signed overflow: $C_n \text{ XOR } C_{n-1}$

$$\begin{array}{r} 1100 \\ 1100 \\ \hline 11000 \end{array}$$

Simple ALU Schematic



Agenda

- Muxes
- Sequential Logic Timing
- Maximum Clock Frequency
- Finite State Machines
- Functional Units
- **Summary**

Bonus Slides

- Logisim Intro

Summary

- Hardware systems are constructed from *Stateless* Combinational Logic and *Stateful* Sequential Logic (includes registers)
- Circuits have a delay to them, and the critical path (longest delay between registers) determines the maximum clock frequency
- Finite State Machines can be used to represent sequential logic states

BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

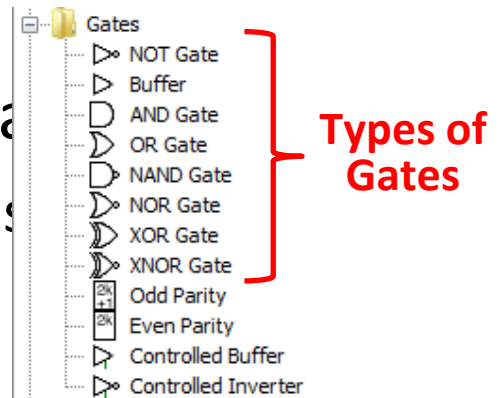
They have been prepared in a way that should be easily readable and the material will be touched upon in following lectures and labs.

Logisim

- Open-source (i.e. free!) “graphical tool for designing and simulating logic circuits”
 - Runs on Java on any computer
 - Download to your home computer via class login or the Logisim website (we are using version 2.14.6)
- No programming involved
 - Unlike Verilog, which is a hardware description language (HDL)
 - Click and drag; still has its share of annoying quirks
- <http://ozark.hendrix.edu/~burch/logisim/>

Gates in Logisim

- Click gate
- Can s



Options

Selection: AND Gate	
Facing	East
Data Bits	1
Gate Size	Medium
Number Of Inputs	5
Output Value	0/1
Label	Medium
Label Font	SansSerif Plain 12
Negate 1 (Top)	No
Negate 2	No
Negate 3	No
Negate 4	No
Negate 5 (Bottom)	No

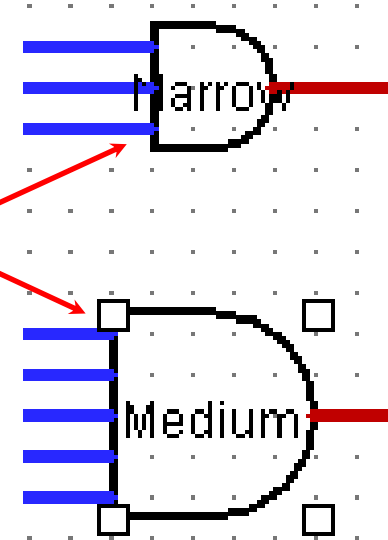
face

cing or select gate *later* to change

← bus width n

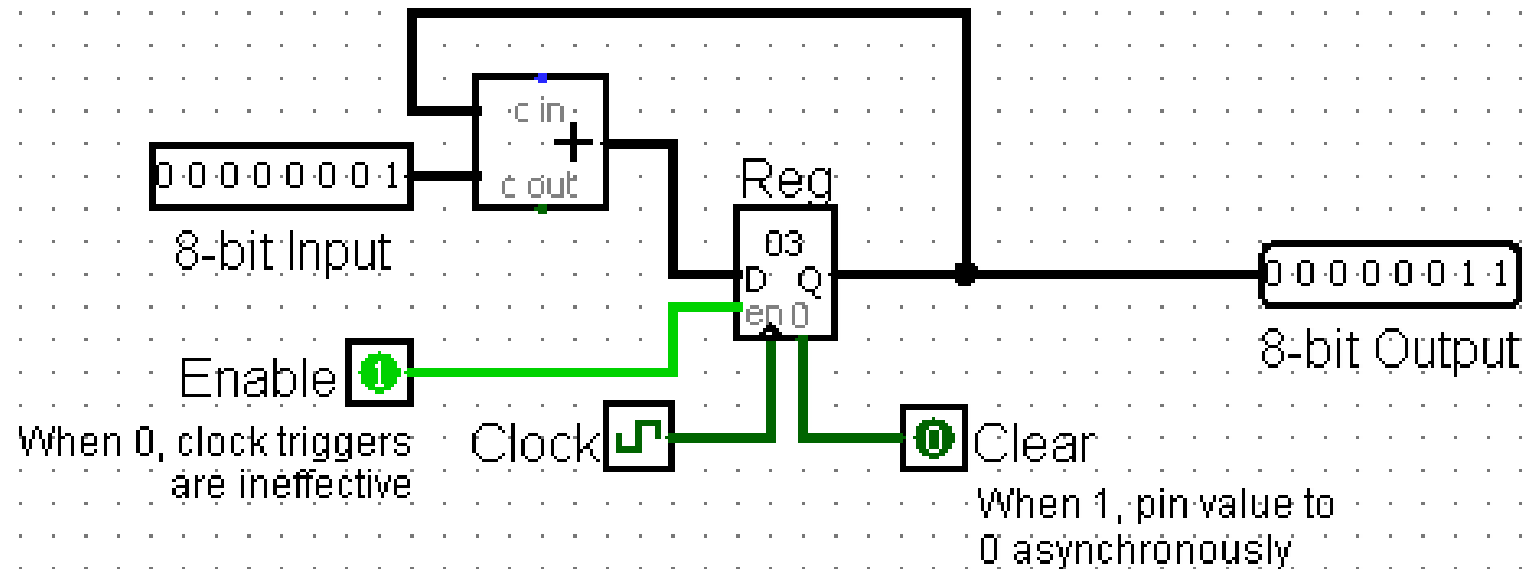
← # inputs

← labeling not necessary,
but can help



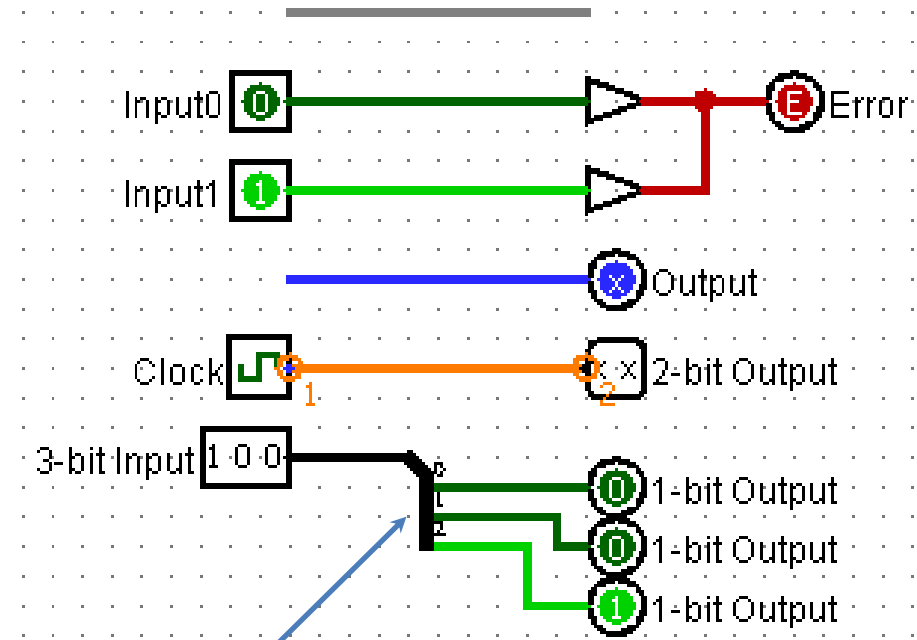
Registers in Logisim

- Flip-flops and Registers in “Memory” folder
- 8-bit accumulator:



Wires in Logisim

- Click and drag on existing port or wire
- **Color schemes:**
 - **Gray:** unconnected
 - **Dark Green:** low signal (0)
 - **Light Green:** high signal (1)
 - **Red:** error
 - **Blue:** undetermined signal
 - **Orange:** incompatible widths
- **Tunnels:** all tunnels with same label are connected

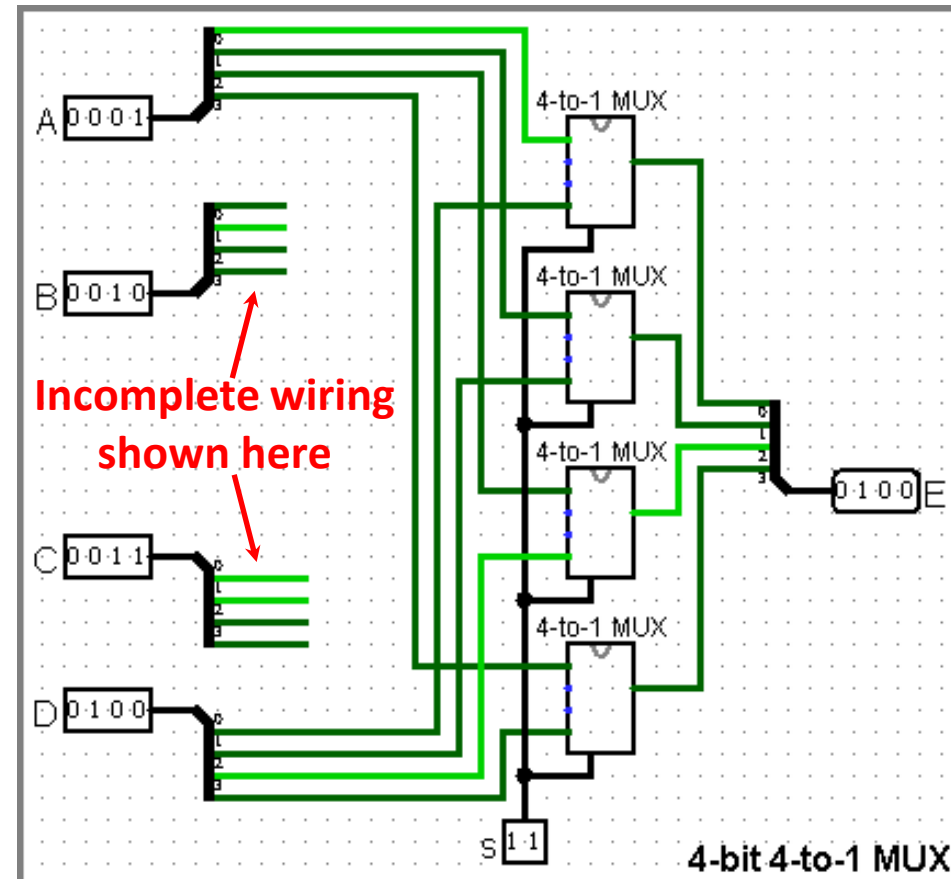
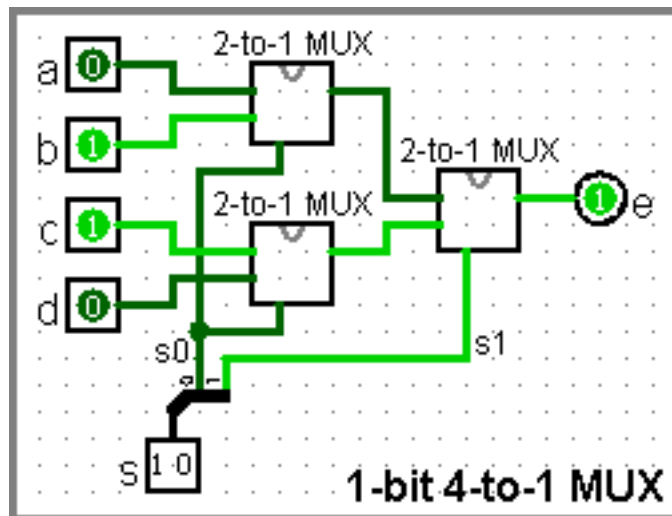
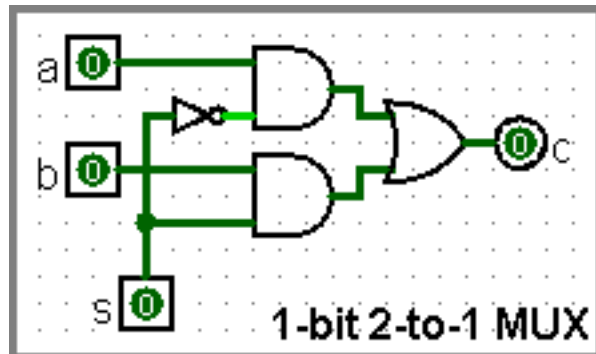


"Splitter" used to adjust bus widths



Subcircuits Example

- Logisim equivalent of procedure or method
 - Every project is a hierarchy of subcircuits



Common Mistakes in Logisim

- Connecting wires together
 - Crossing wires vs. connected wires
- Losing track of which input is which
 - Mis-wiring a block (e.g. CLK to Enable)
 - Grabbing wrong wires off of splitter
- Errors:

