

# Great Ideas in Computer Architecture

*Sequential Logic, Finite State Machines*

Instructor: Nick Riasanovsky

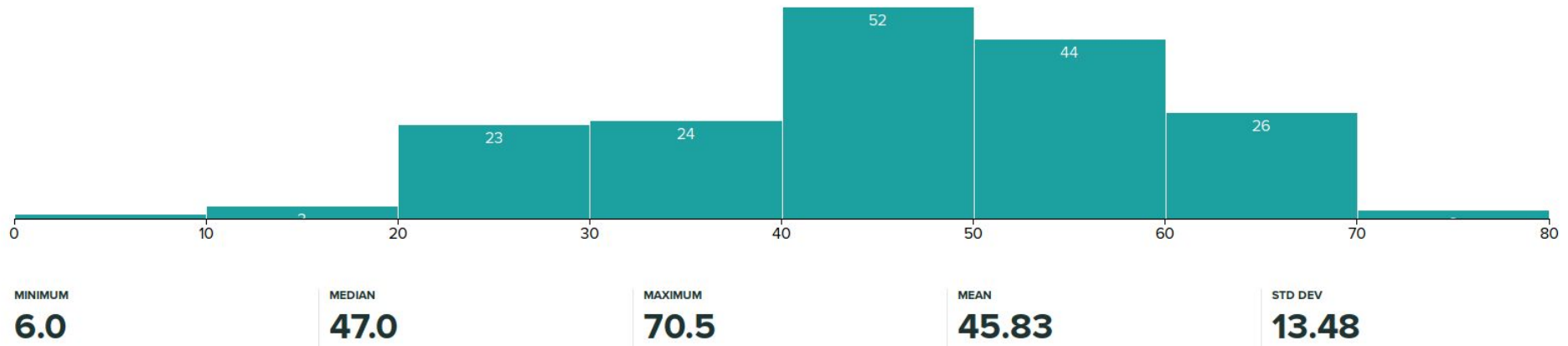
# Nick's Lecture Feedback

- 2 Big Takeaways:
  - My lectures are too fast
    - Understood and I will go slower from now on
  - Cool it with the analogies
    - Understood! But with a couple topics you will still see some analogies
    - Only ones with a clear and easy to understand relationship
    - No more Harry Potter explanations

# Questions

- Pretty seem overall to be satisfied with the number of questions we answer
- We LOVE to answer your questions but this is some of what make lectures have a time crunch
  - Going too fast is still a me problem I'll fix
- If you have a question that expands beyond the material in the course please consider using the piazza lecture thread

# Midterm Grades



- Midterm Grades are released on gradescope
- Regrades available after lecture, due by Tues
- You did great, this exam was hard

# Great Idea #1: Levels of Representation & Interpretation

Higher-Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g. RISC-V)

Assembler

Machine Language Program (RISC-V)

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw    t0, 0(x2)  
lw    t1, 4(x2)  
sw    t1, 0(x2)  
sw    t0, 4(x2)
```

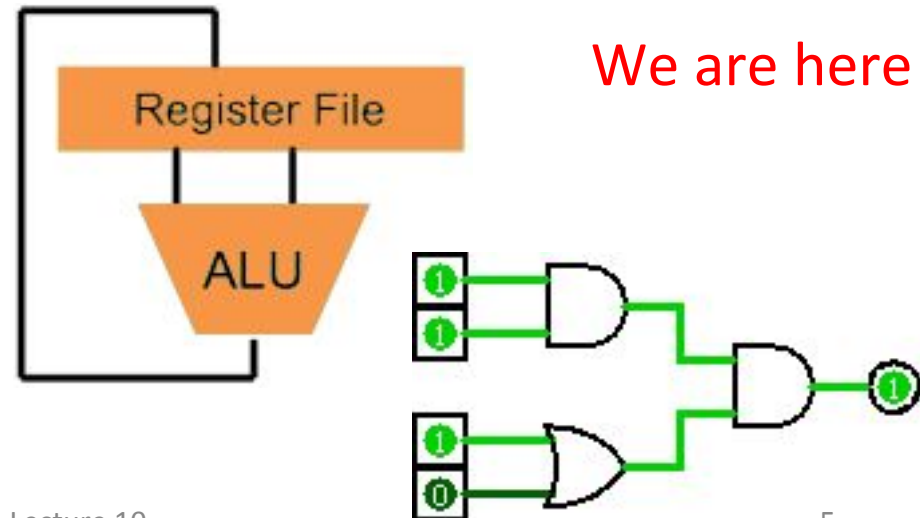
```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

Architecture Implementation

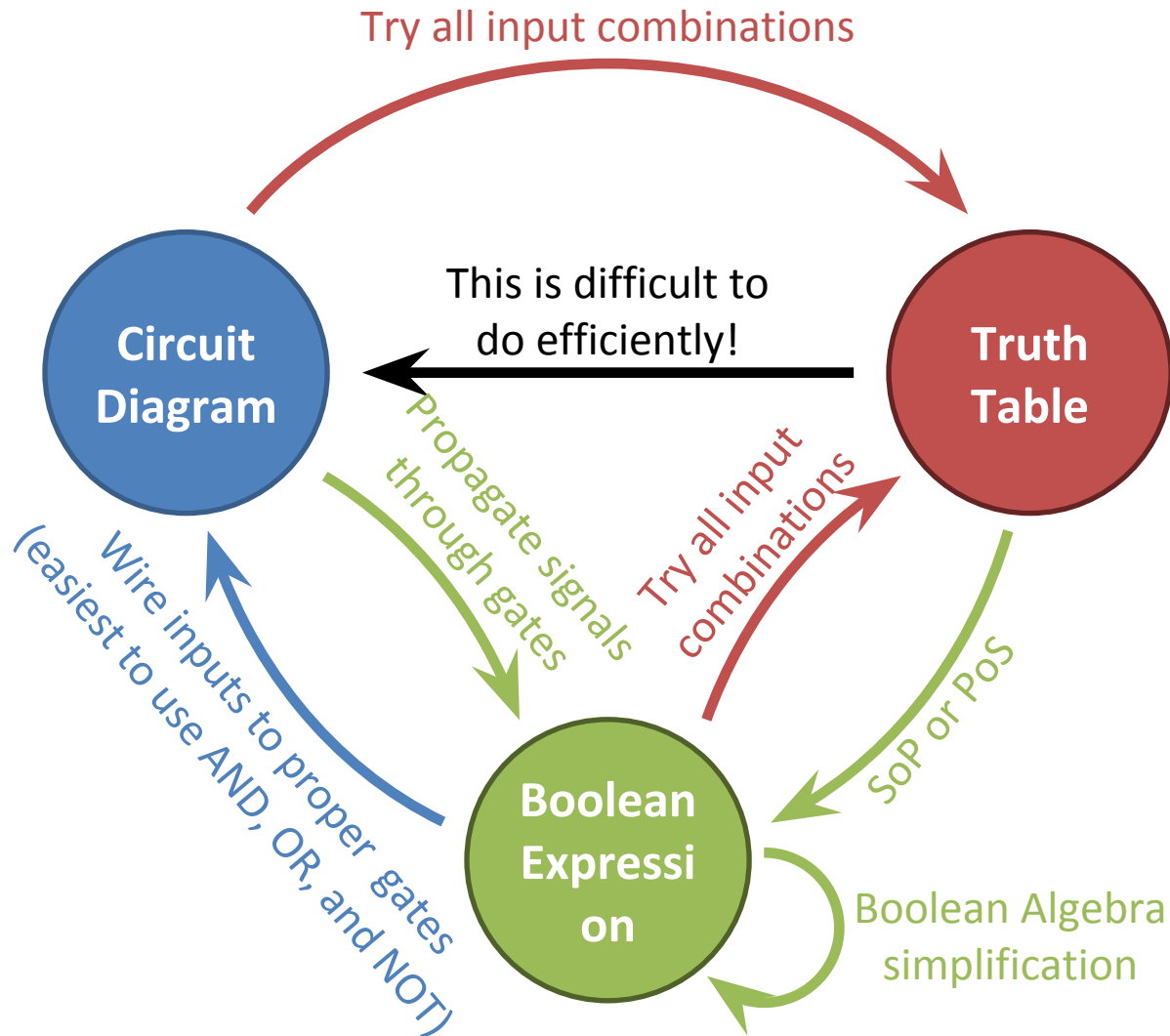
Logic Circuit Description (Circuit Schematic Diagrams)



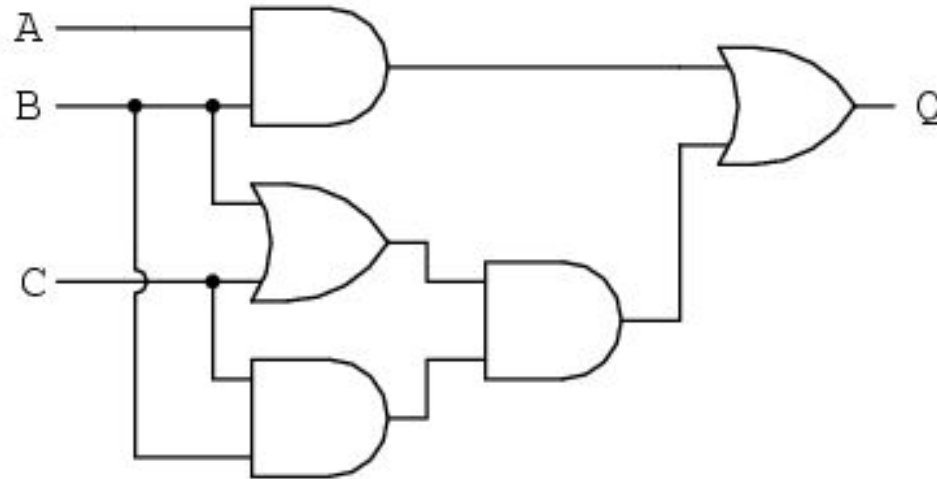
# Review

- Hardware is made up of transistors and wires
  - *Building blocks of all higher-level blocks*
- Synchronous Digital Systems
  - All signals are seen as either 0 or 1
  - Consist of two basic types of circuits:
    - Combinational Logic (CL)
      - AND, OR, XOR
      - Use Truth Tables -> Boolean Algebra
    - Sequential Logic (SL)

# Converting Combinational Logic



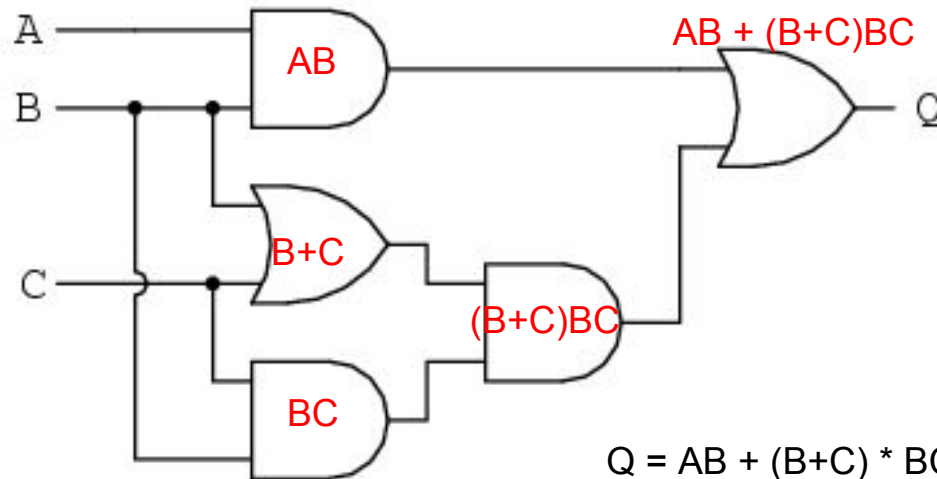
**Question:** What is the MOST simplified Boolean Algebra expression for the following circuit?



- (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

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

# Review

- Hardware is made up of transistors and wires
  - *Building blocks of all higher-level blocks*
- Synchronous Digital Systems
  - All signals are seen as either 0 or 1
  - Consist of two basic types of circuits:
    - Combinational Logic (CL)
      - AND, OR, XOR
      - Use Truth Tables -> Boolean Algebra
    - **Sequential Logic (SL)**

# Today's Menu

- Sequential Logic (SL)
  - Pulse of a Clock controls flow of information
  - Allows us to keep state (the basis of memory)
  - Understand how to efficiently use our hardware
- Finite State Machines
  - Abstract away combinational and sequential logic into **functions**
    - These functions take in a stream of bits, do something with them, and output something deterministic

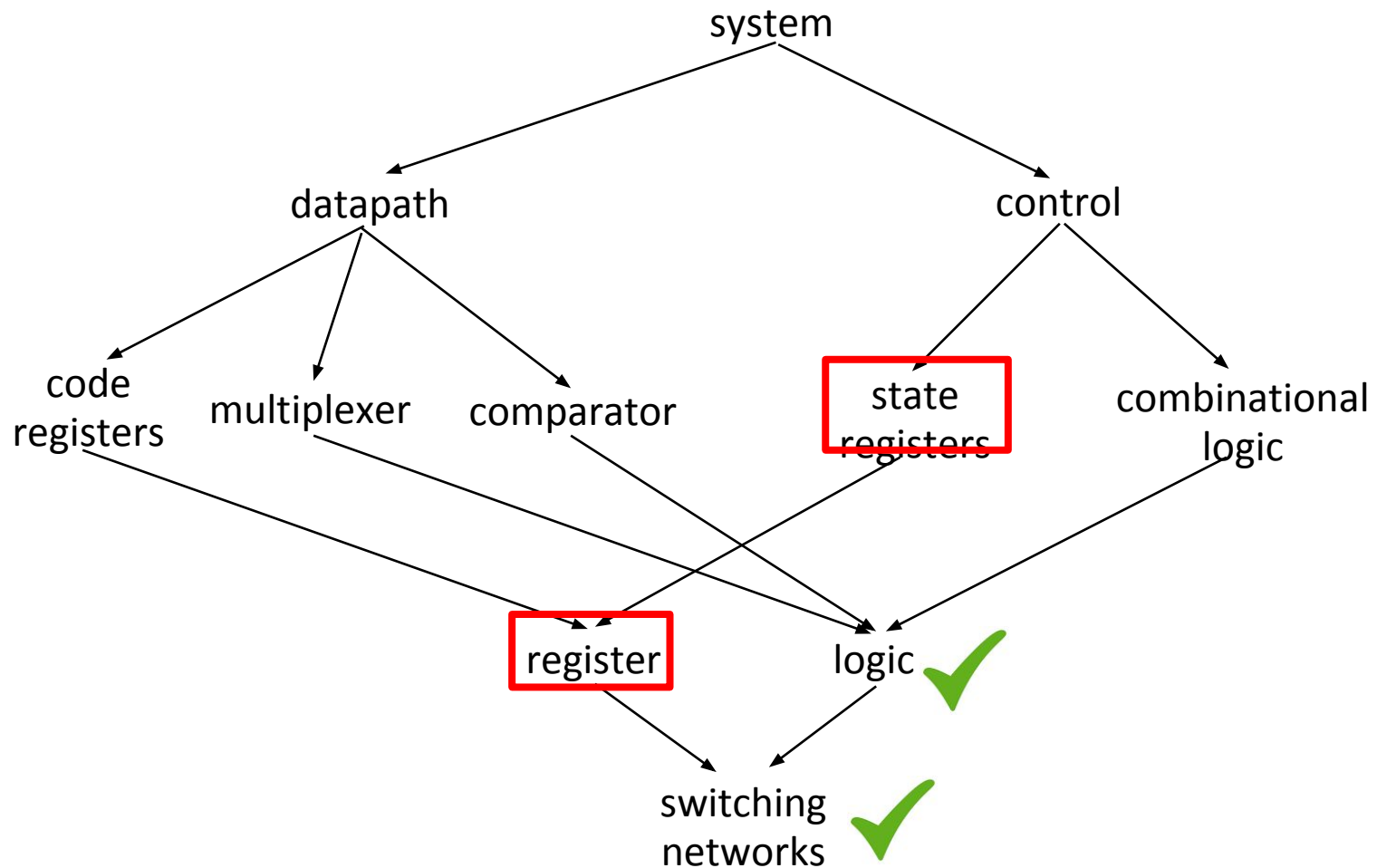
# Agenda

- Transistors, Switching Networks
- Combinational Logic Representations
  - Truth Tables
  - Boolean Algebra
- Meet the Staff
- **Sequential Logic**
  - **Timing Terminology**
  - **State Elements**

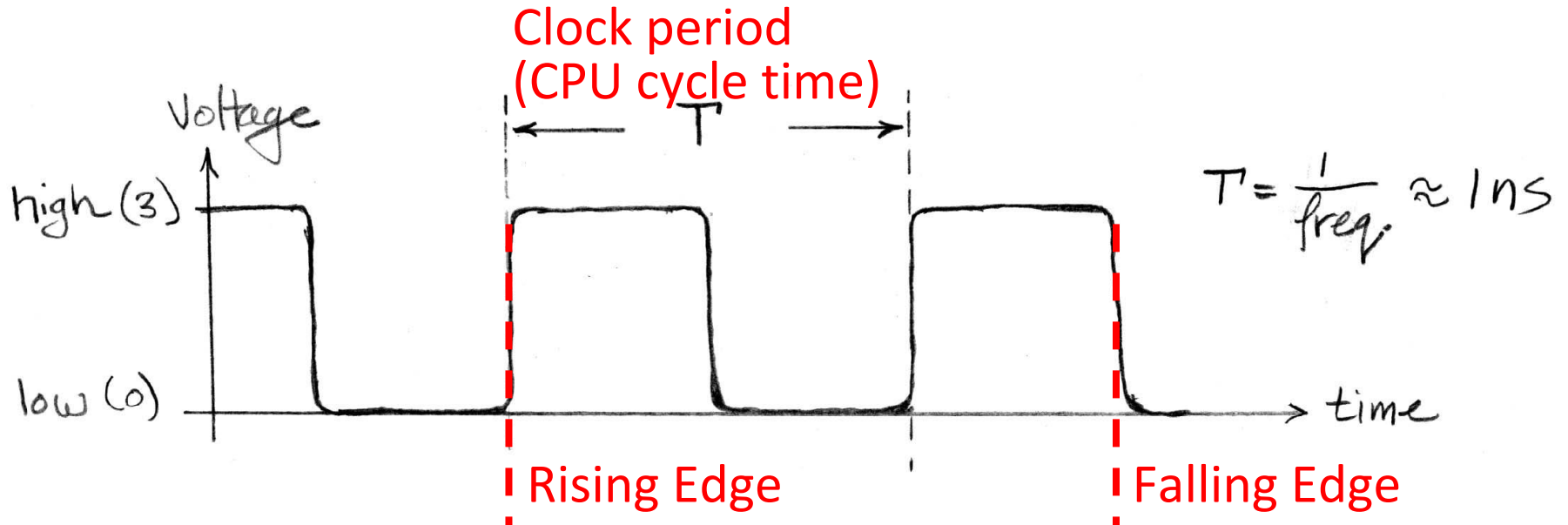
# Type of Circuits

- *Synchronous 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)

# Hardware Design Hierarchy

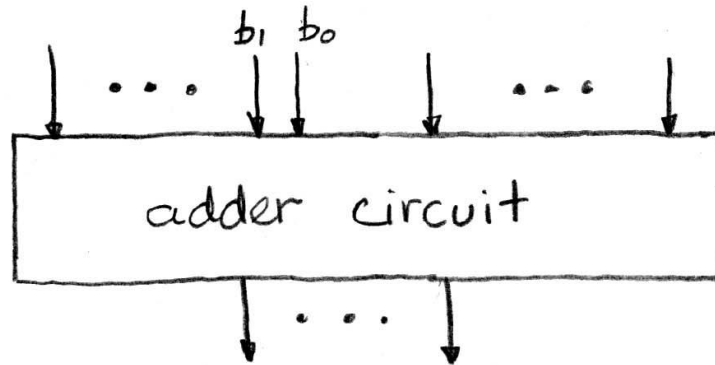


# Signals and Waveforms: Clocks

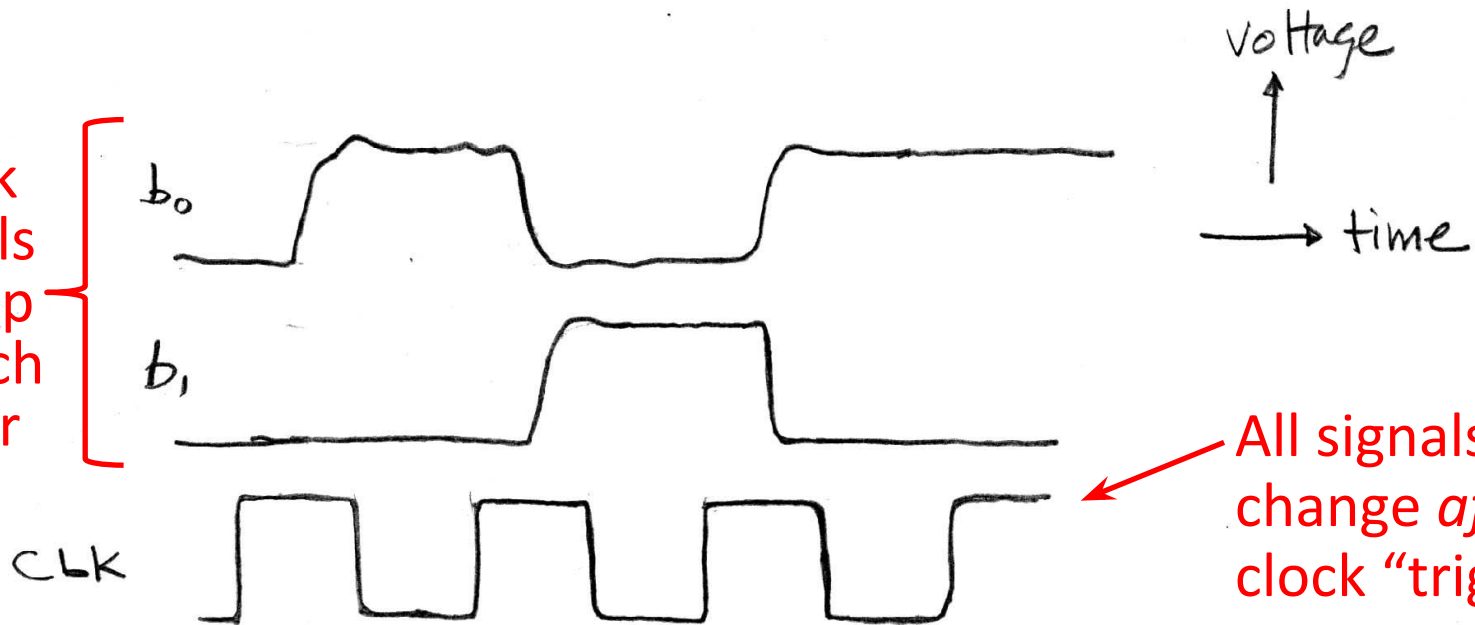


- **Signals** transmitted over wires continuously
- Transmission is effectively instantaneous
  - Implies that any wire only contains one value at any given time

# Signals and Waveforms

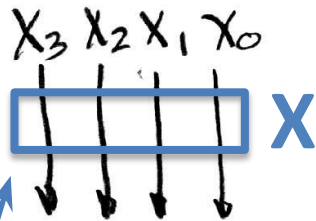
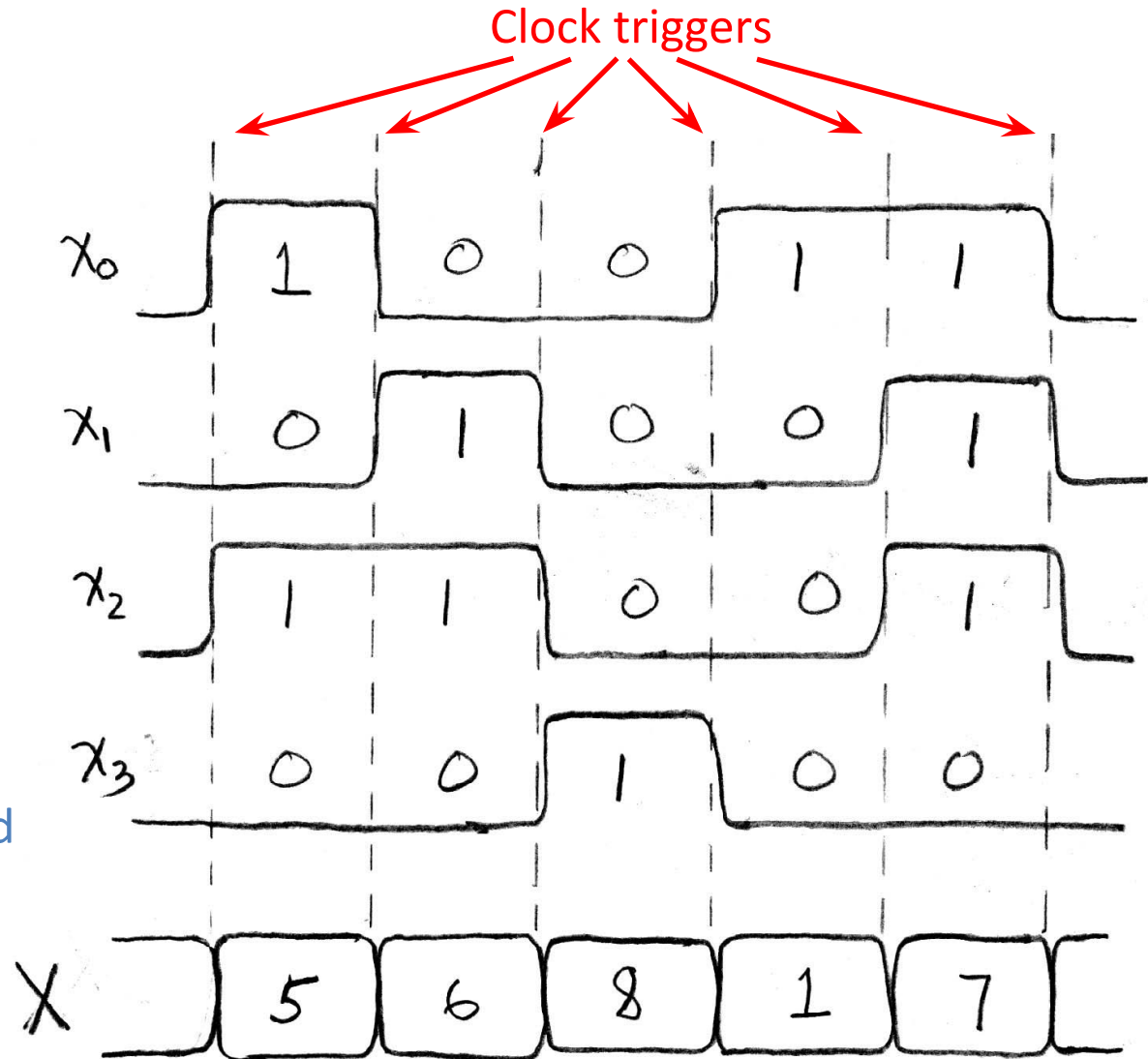


Stack signals on top of each other





# Signals and Waveforms: Grouping



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

# 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 to control the flow of information.



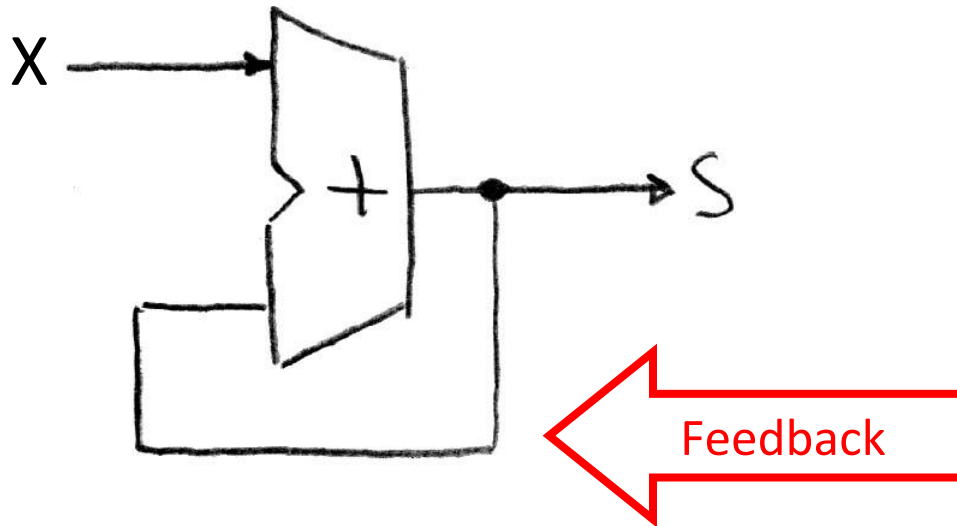
Want:  $S=0;$   
for  $X_1, X_2, X_3$  over time...

$$S = S + X_i$$

Assume:

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

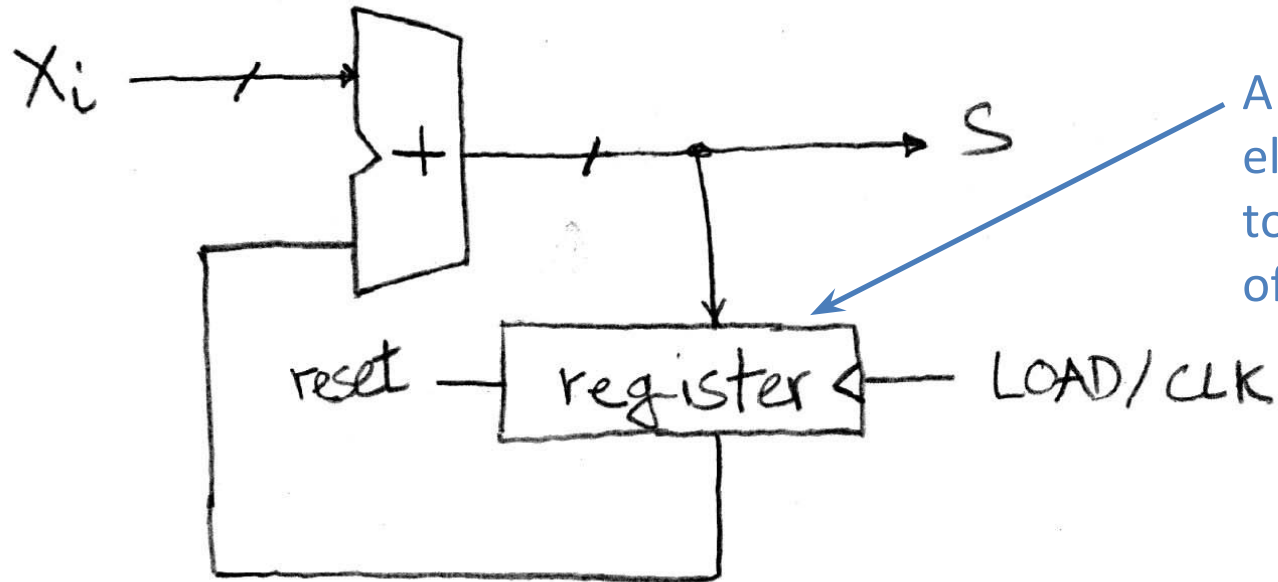
# First Try: Does this work?



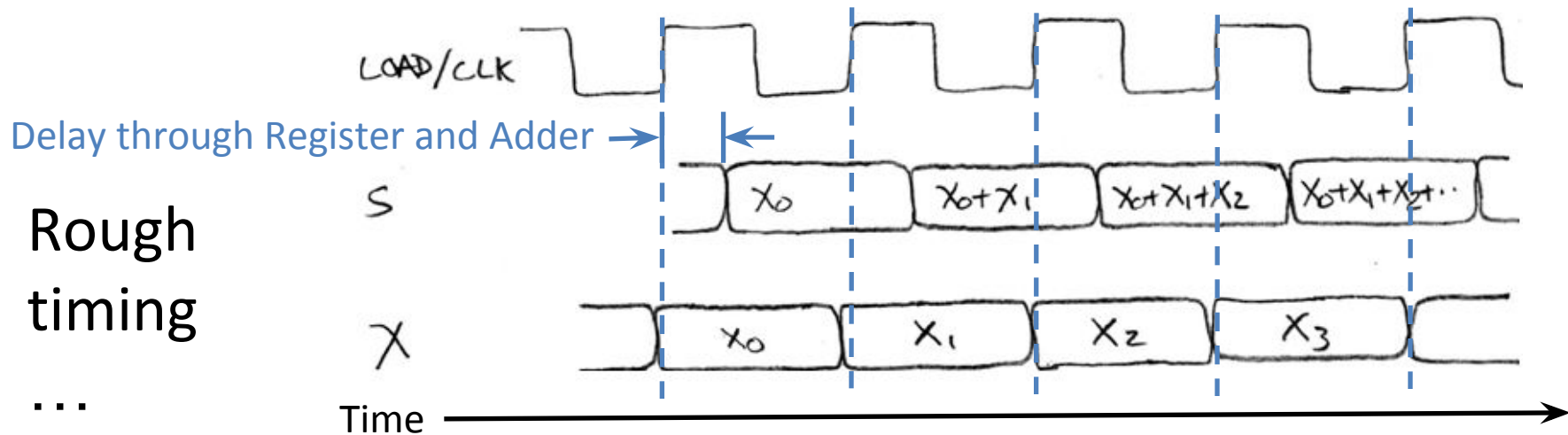
No!

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

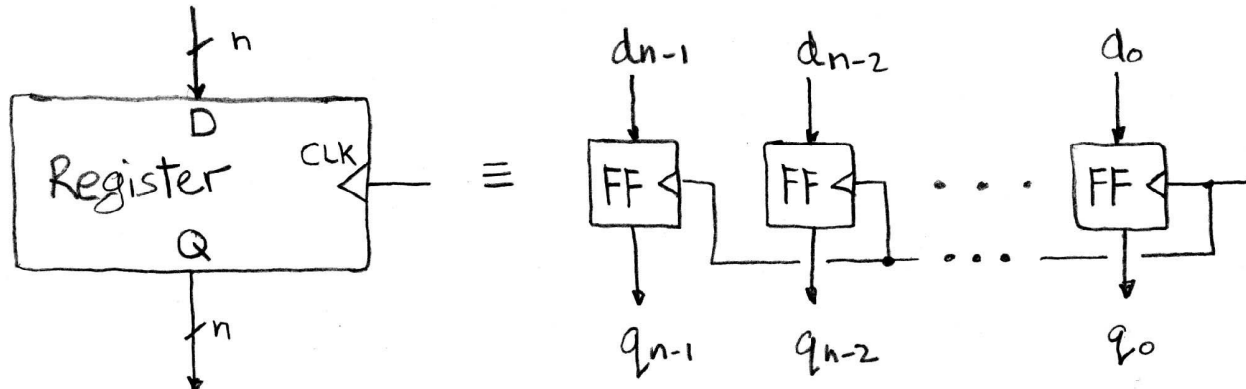
# Second Try: How About This?



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

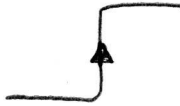


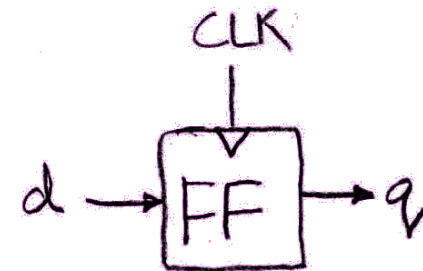
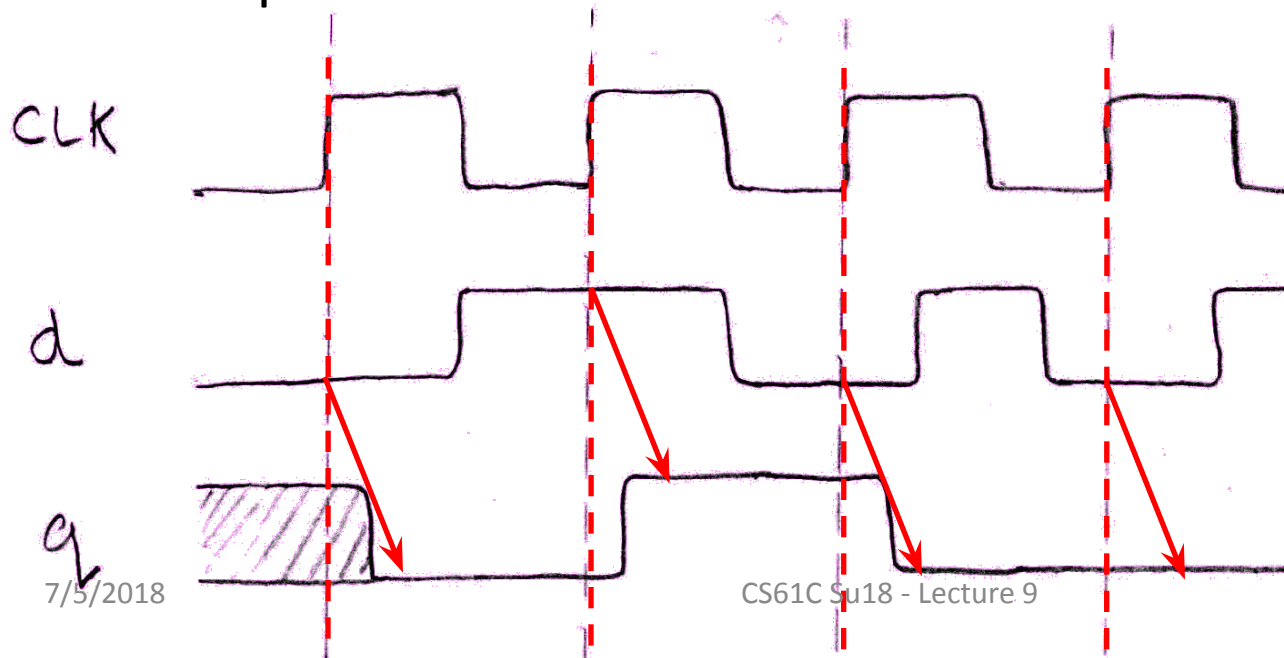
# Register Internals



- 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](http://en.wikibooks.org/wiki/Practical_Electronics/Flip-flops)

# Flip-Flop Timing Behavior (1/2)

- Edge-triggered D-type flip-flop
  - This one is “positive 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
  - *Check for blurriness* once image appears on the display





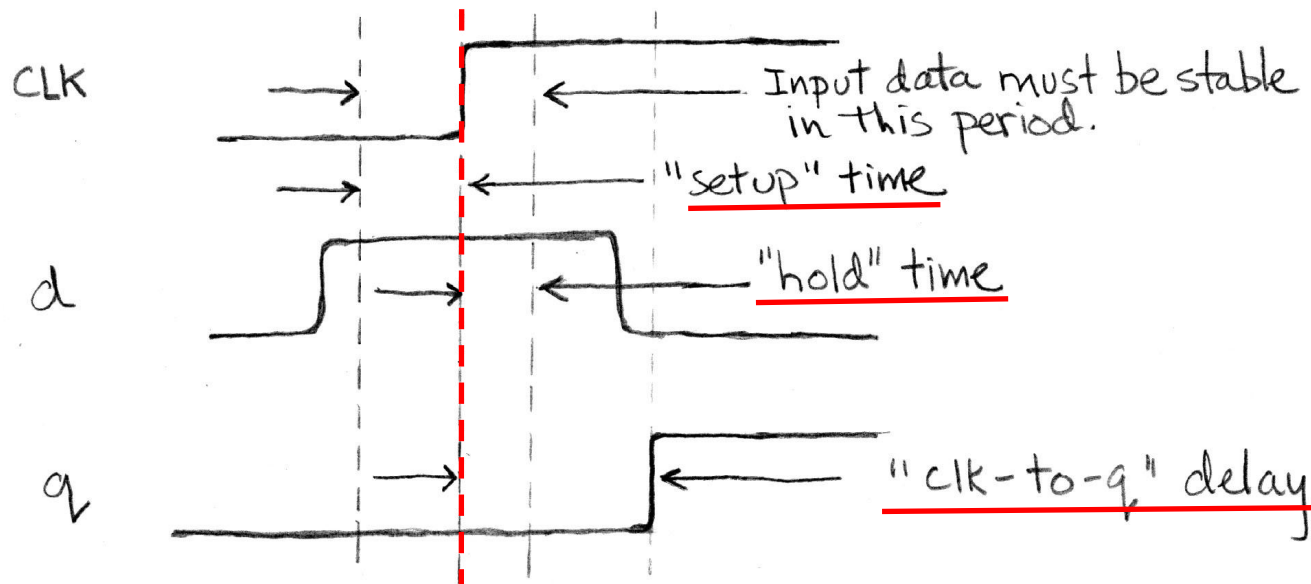
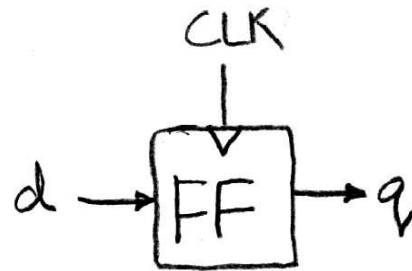
# Flip-Flop Timing Terminology (2/3)

- 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)

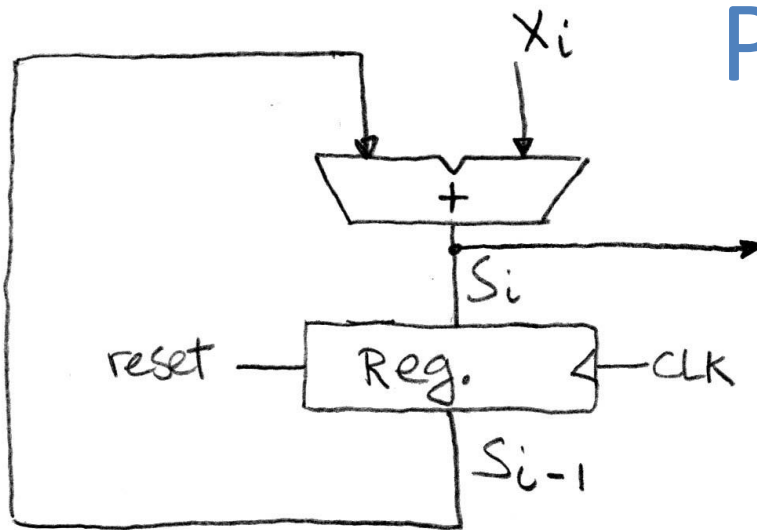
- Now applied to hardware:
  - *Setup Time*: how long the input must be stable *before* the CLK trigger for proper input read
  - *Hold Time*: how long the input must be stable *after* the CLK trigger for proper input read
  - *“CLK-to-Q” Delay*: how long it takes the output to change, measured from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

# Flip-Flop Timing Behavior

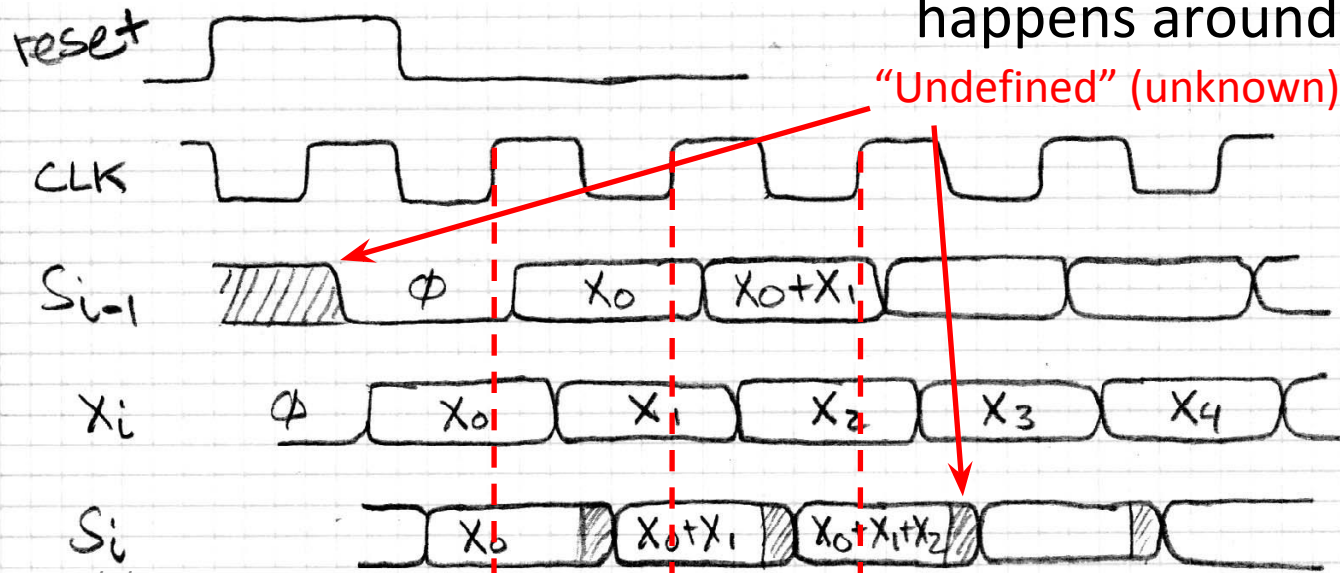


# Accumulator Revisited

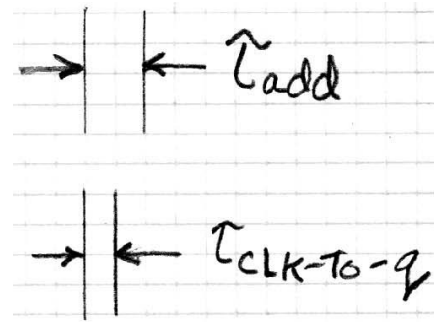
## Proper Timing (2/2)



- reset signal shown
- Also, 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



“Undefined” (unknown) signal



# 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

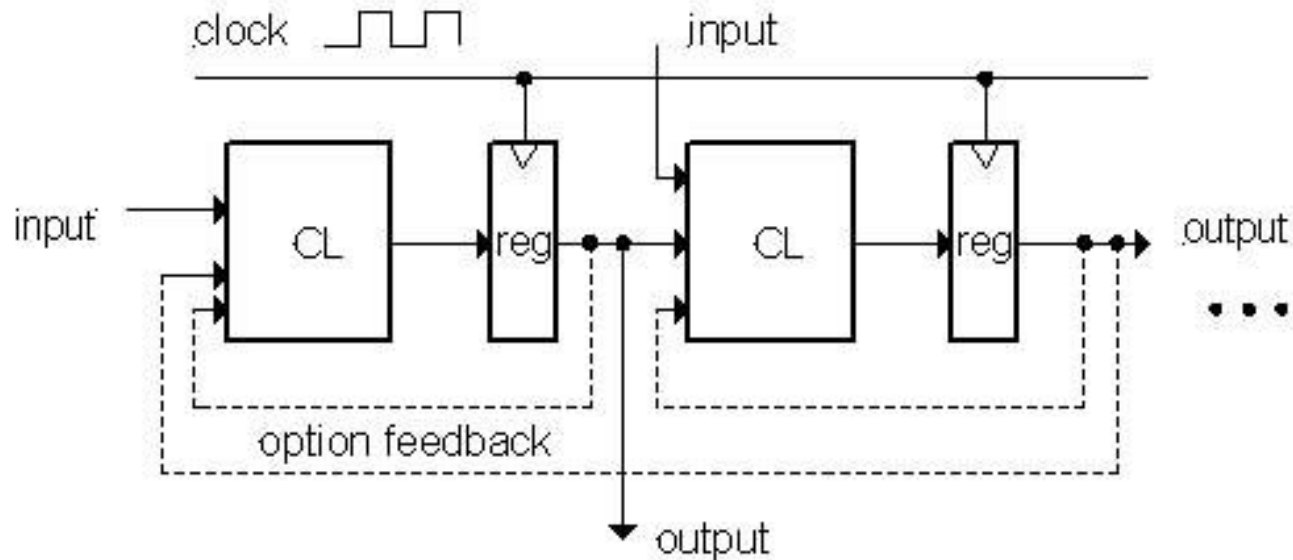
# Review of Timing Terms

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

# Agenda

- **Critical Path and Clock Frequency**
- Administrivia
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter
- Bonus:
  - Pipelining intro
  - Handling overflow
  - Logisim Introduction

# 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?

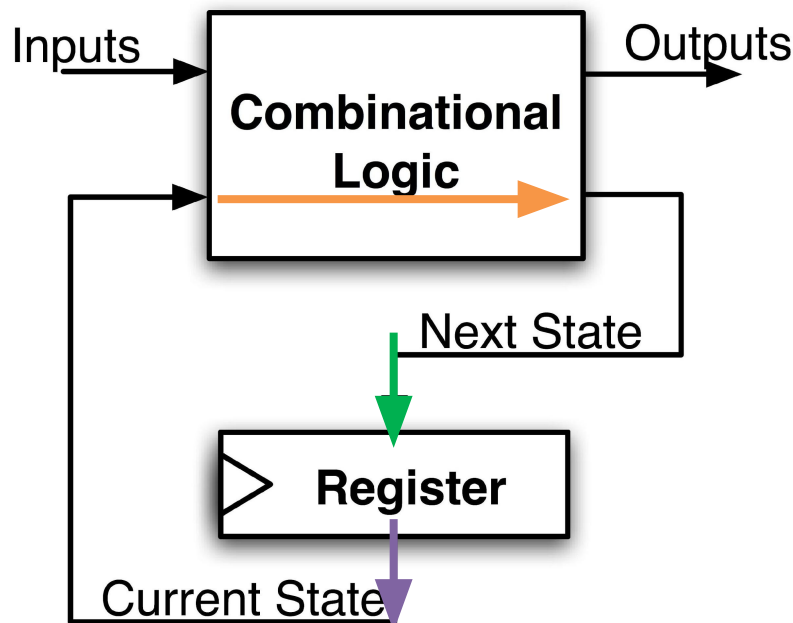


# When Can the Input Change?

- When a register input changes shouldn't violate hold time ( $t_{hold}$ ) or setup time ( $t_{setup}$ ) constraints within a clock period ( $t_{period}$ )
- Let  $t_{input}$  be the time it takes for the input of a register to change, measured from the CLK trigger:
  - Then we need  $t_{hold} \leq t_{input} \leq t_{period} - t_{setup}$
  - Two separate constraints!

# 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)

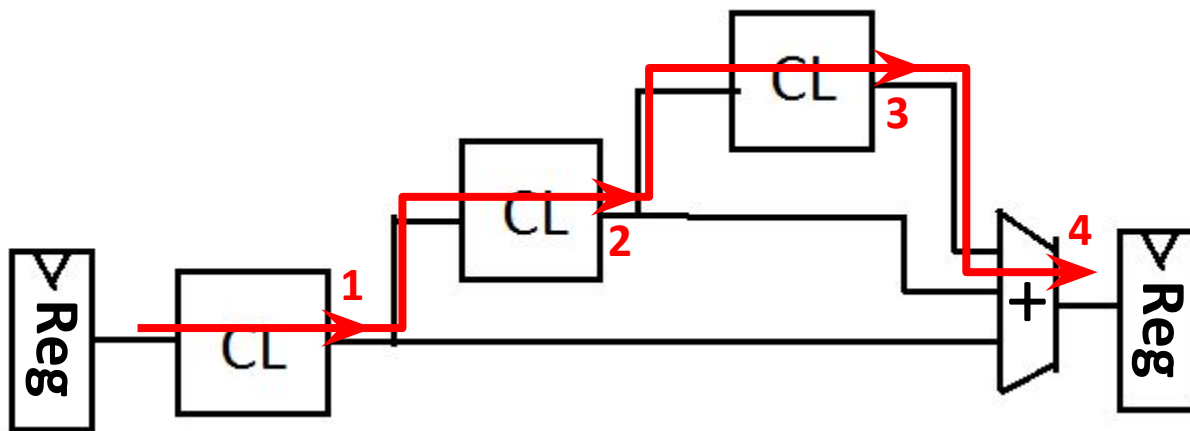


$$\text{Max Delay} = \text{CLK-to-Q Delay} \\ + \text{CL Delay} \\ + \text{Setup Time}$$

$$\text{Min Period} = \text{Max Delay} \\ \text{Max Freq} = 1/\text{Min Period}$$

# 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



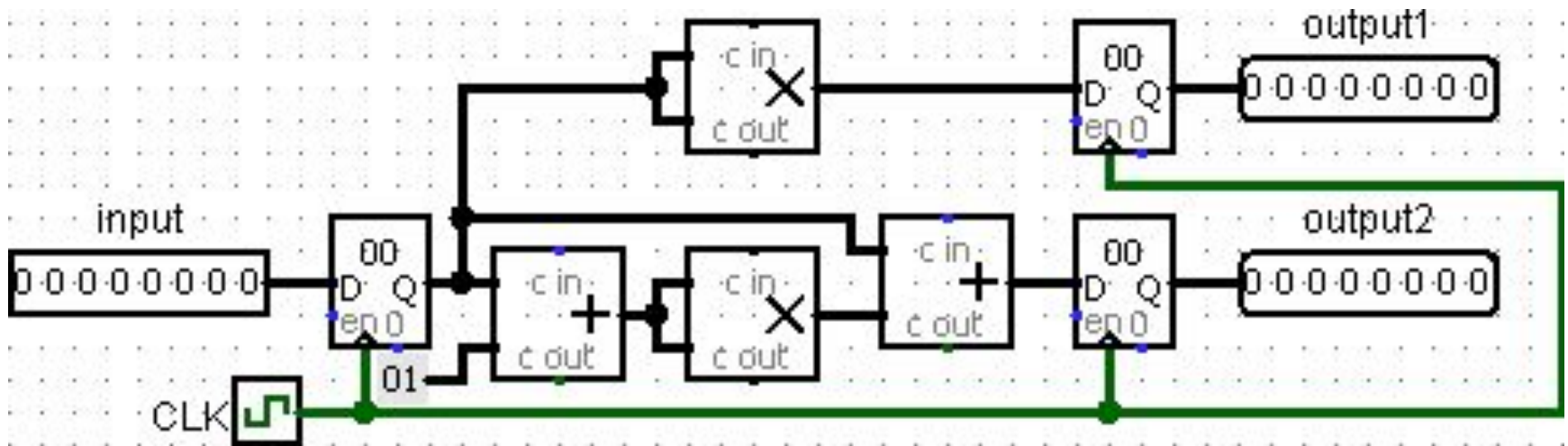
**Critical Path =**

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

**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  $t_{\text{clk-to-q}}$  there can be?



(A) 550 ps

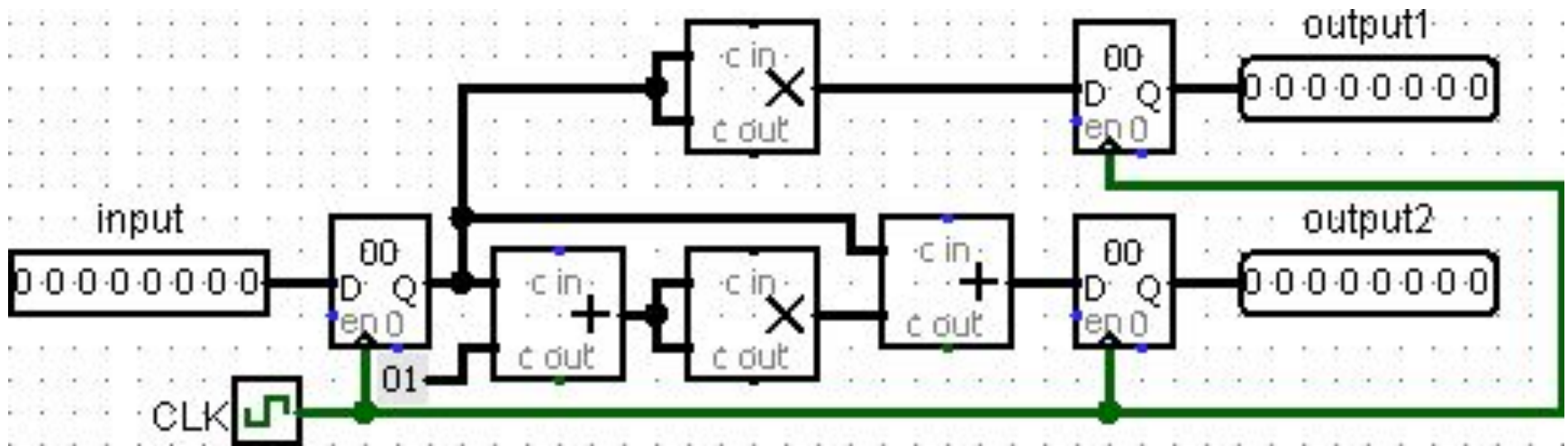
(B) 750 ps

(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  $t_{\text{clk-to-q}}$  we can use?



**(A) 550 ps**

**(B) 750 ps**

**(C) 500 ps**

**(D) 700 ps**

Bottom path is critical path:

$$T_{\text{clk-t-q}} + 100 + 200 + 100 + 50 < 1000 \text{ ps} = 1\text{ns}$$

$$T_{\text{clk-t-q}} + 450 < 1000 \text{ ps}$$

$$T_{\text{clk-t-q}} < 550$$

# Administrivia

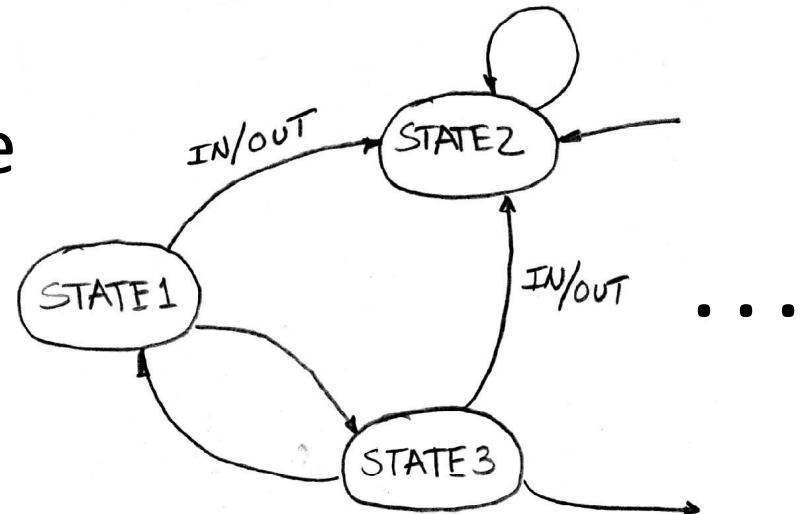
- Homework 2 due tomorrow
- Homework 3 released, due 7/16
- Homework 4 released 7/09, due 7/16
- Proj2-1 due tomorrow
  - Project party tomorrow, 4-6p in the Woz
- Proj2-2 will be released tomorrow
- Please don't leave lecture early!
  - And please DO NOT enter someone else's discussion while the room is still in use

# Agenda

- Critical Path and Clock Frequency
- Administrivia
- **Finite State Machines**
- Multiplexers
- ALU Design
  - Adder/Subtracter
- Bonus:
  - Pipelining intro
  - Handling overflow
  - Logisim Introduction

# Finite State Machines (FSMs)

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



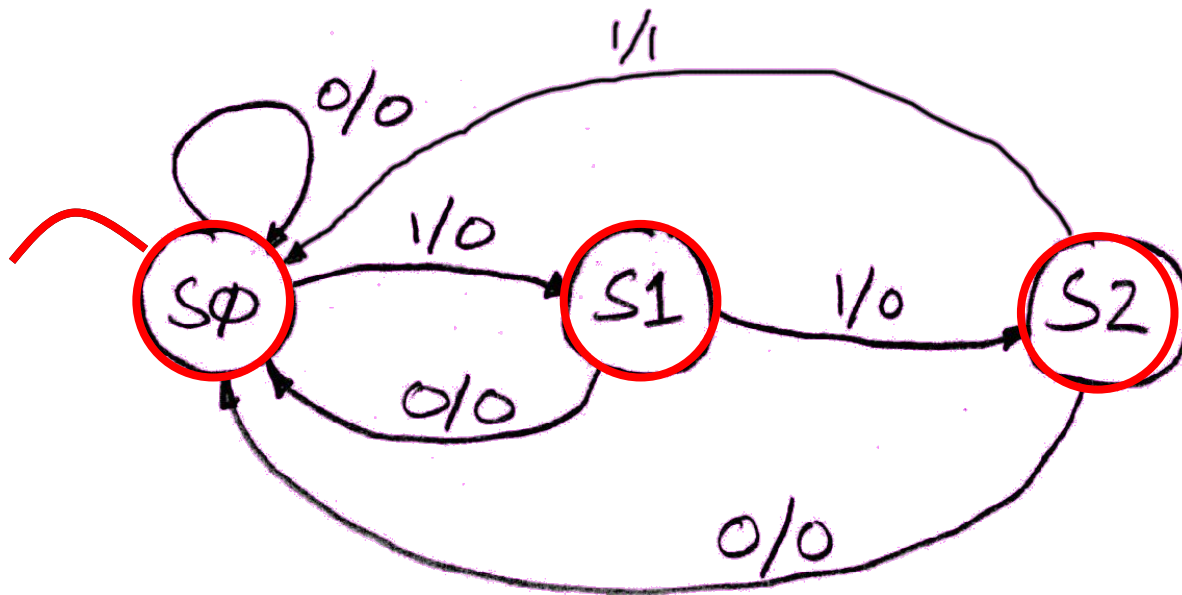


# FSM Overview

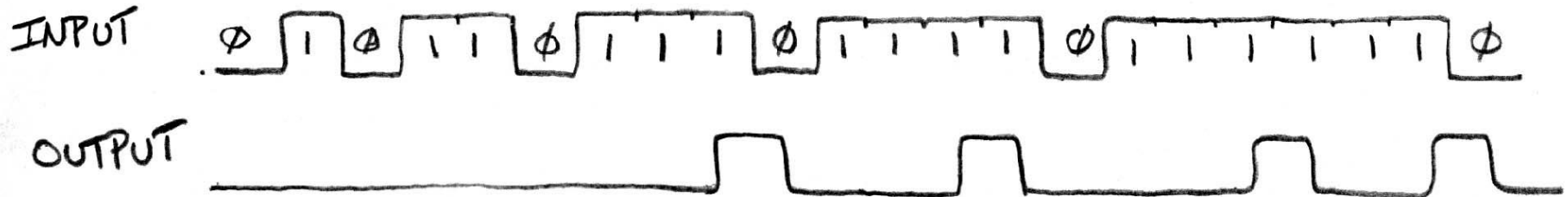
- An FSM (in this class) is defined by:
  - A set of *states*  $S$  (circles)
  - An *initial state*  $s_0$  (only arrow not between states)
  - A *transition function* that maps from the current input and current state to the output and the next state (arrows between states)
- State transitions are 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

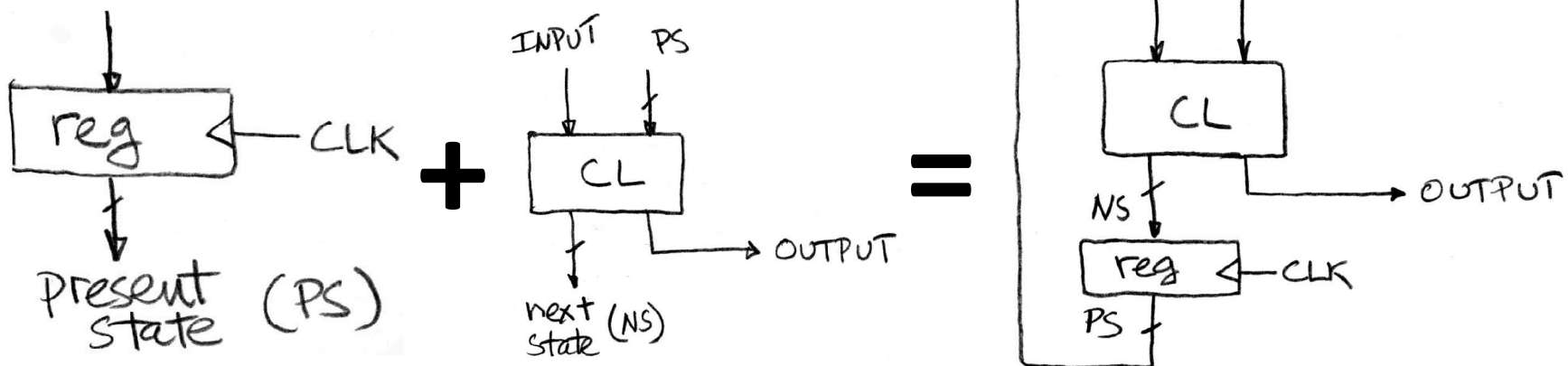


**States:** S0, S1, S2  
**Initial State:** S0  
**Transitions of form:**  
input/output



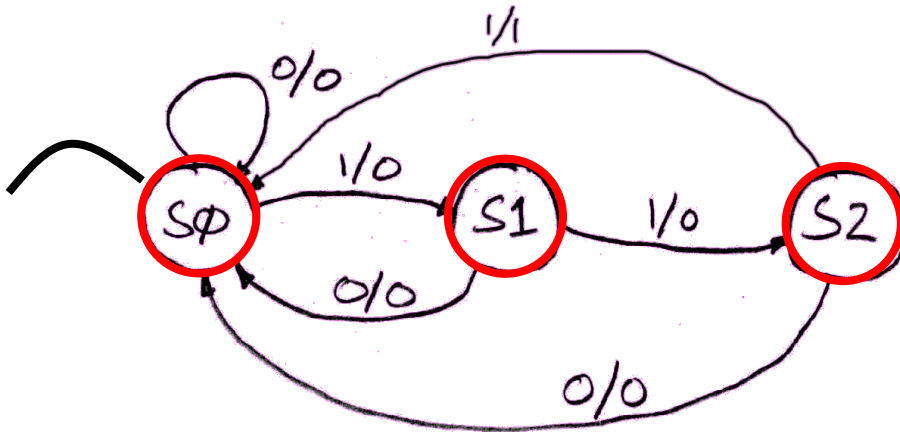
# Hardware Implementation of FSM

- 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

- Implement logic for *EACH* output (2 for NS, 1 for Out)

# Unspecified Output Values (1/2)

- 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


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


# Unspecified Output Values (2/2)

- Let's find expression for  $NS_1$ 
  - Recall:** 2-bit output is just a 2-bit bus, which is just 2 wires

- Boolean algebra:

$$NS_1 = \neg CS_1 CS_0 In + CS_1 \cancel{CS_0} \neg In$$

Differs by 2 

Is neighbor 

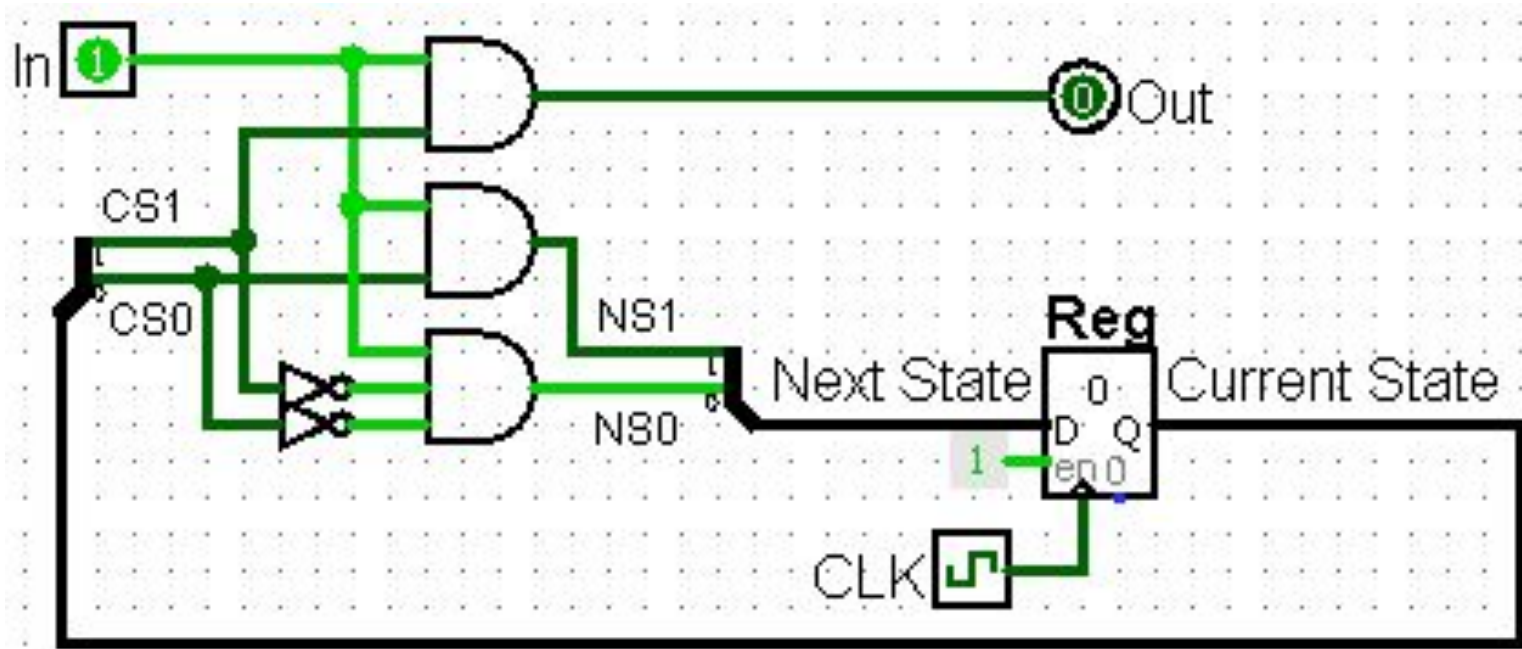
$$NS_1 = CS_0 In (CS_1 + \neg CS_1)$$

$$NS_1 = CS_0 In$$

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:**  $NS_1 = CS_0 In$ ,  $NS_0 = \neg CS_1 \neg CS_0 In$ ,  $Out = CS_1 In$

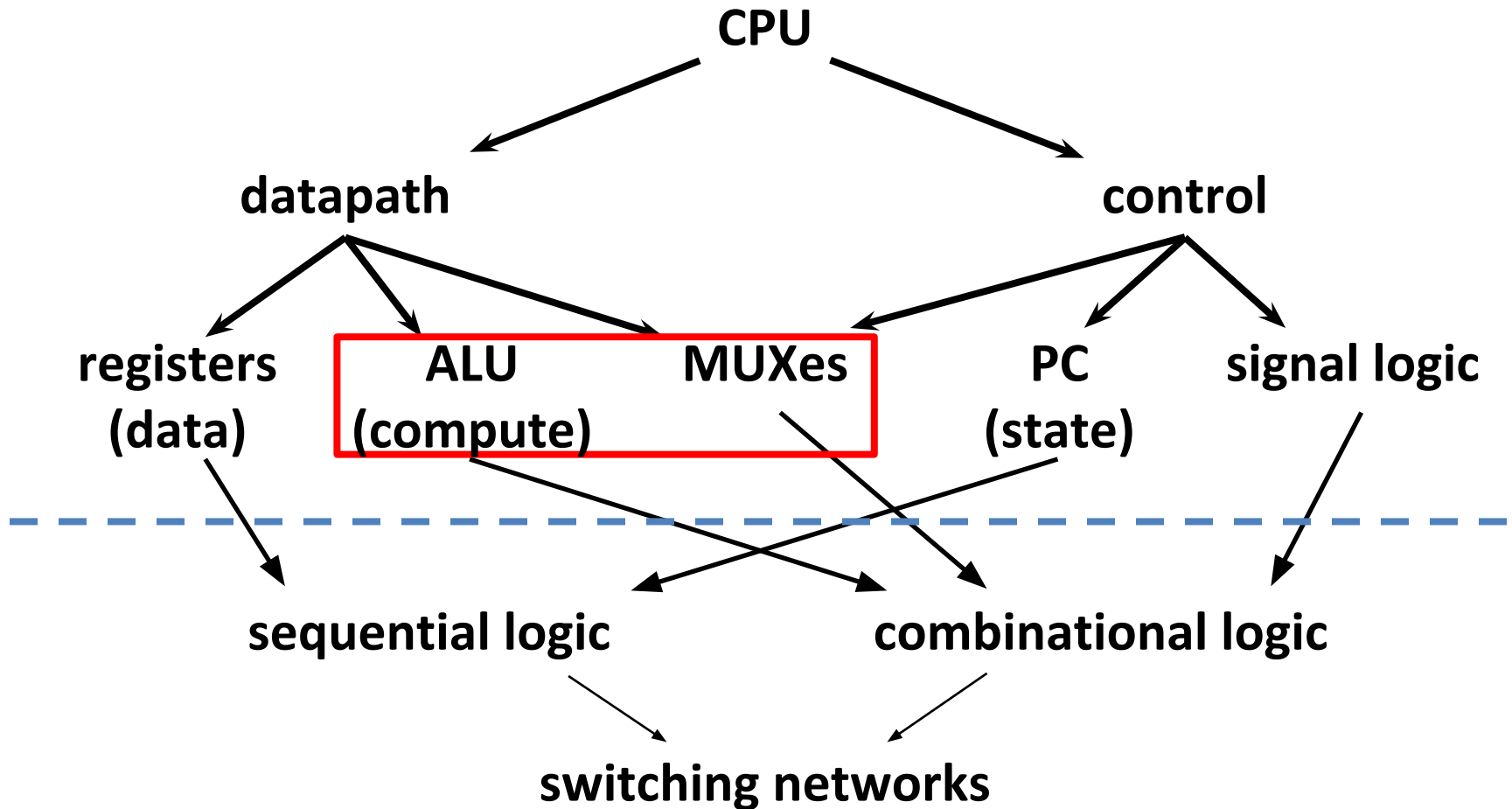


# Agenda

- Critical Path and Clock Frequency
- Administrivia
- Finite State Machines
- **Multiplexers**
- ALU Design
  - Adder/Subtracter
- Bonus:
  - Pipelining intro
  - Handling overflow
  - Logisim Introduction

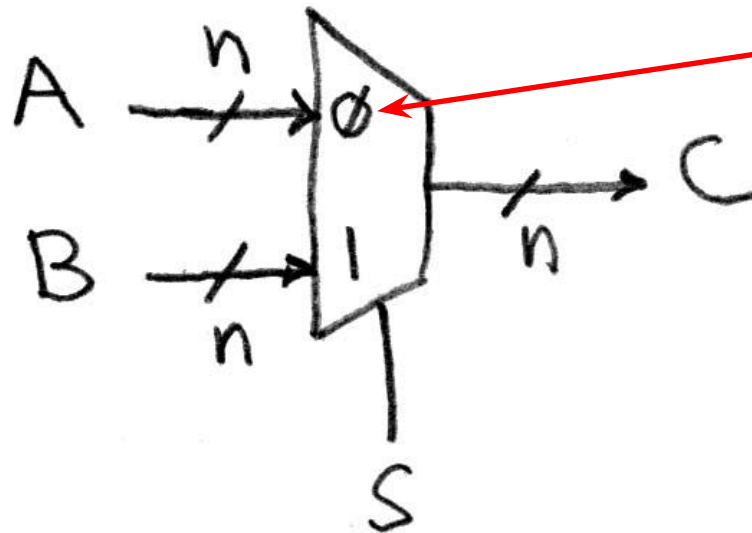


# Hardware Design Hierarchy



# Data Multiplexor

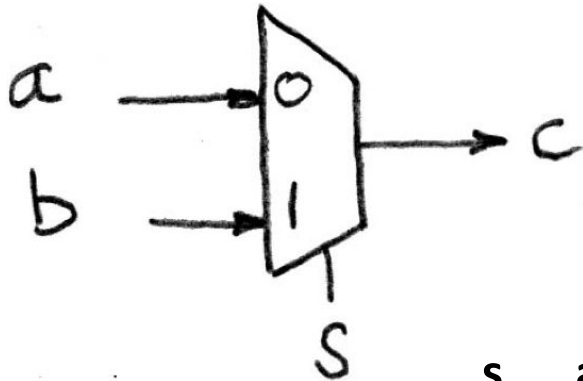
- Multiplexor (“MUX”) is a *selector*
  - Place one of multiple inputs onto output (N-to-1)
- Shown below is an n-bit 2-to-1 MUX
  - Input S selects between two inputs of n bits each



This input is passed to output if selector bits match shown value

# Implementing a 1-bit 2-to-1 MUX

- **Schematic:**



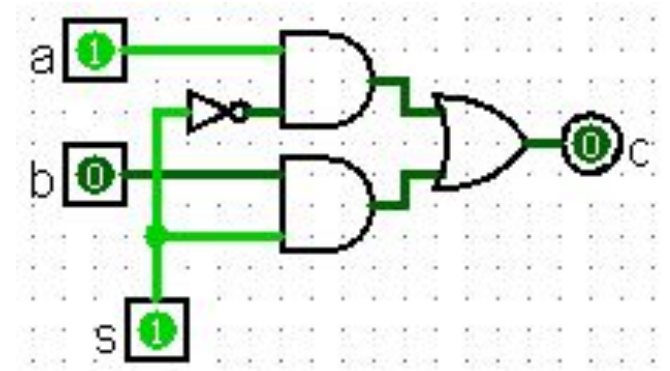
- **Truth Table:**

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:**

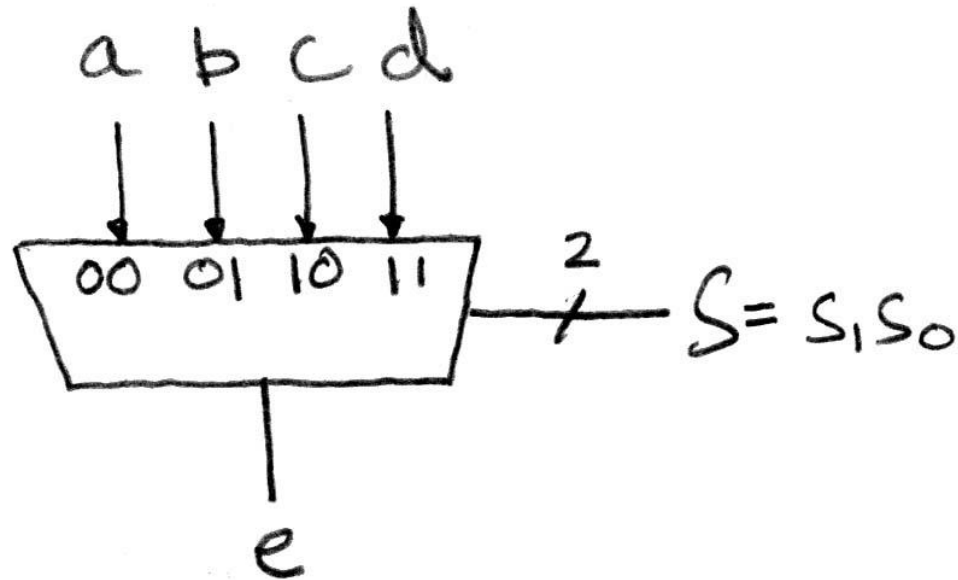
$$\begin{aligned}c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\ &= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\ &= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\ &= \bar{s}(a(1) + s((1)b) \\ &= \bar{s}a + sb\end{aligned}$$

- **Circuit Diagram:**



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

- **Schematic:**

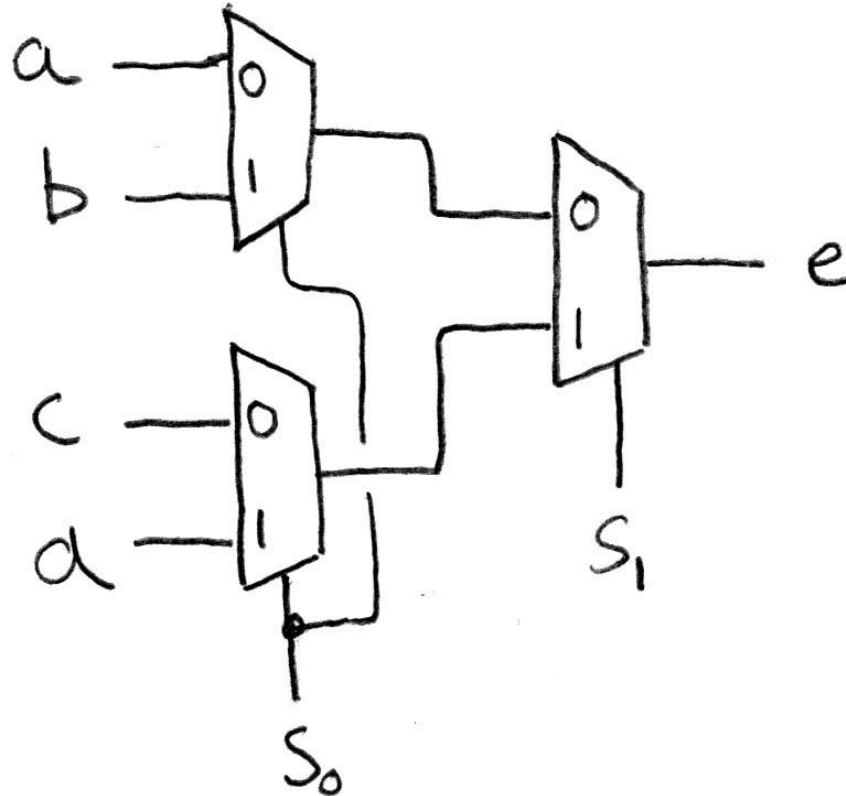


- **Truth Table:** How many rows?  $2^6$
- **Boolean Expression:**

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

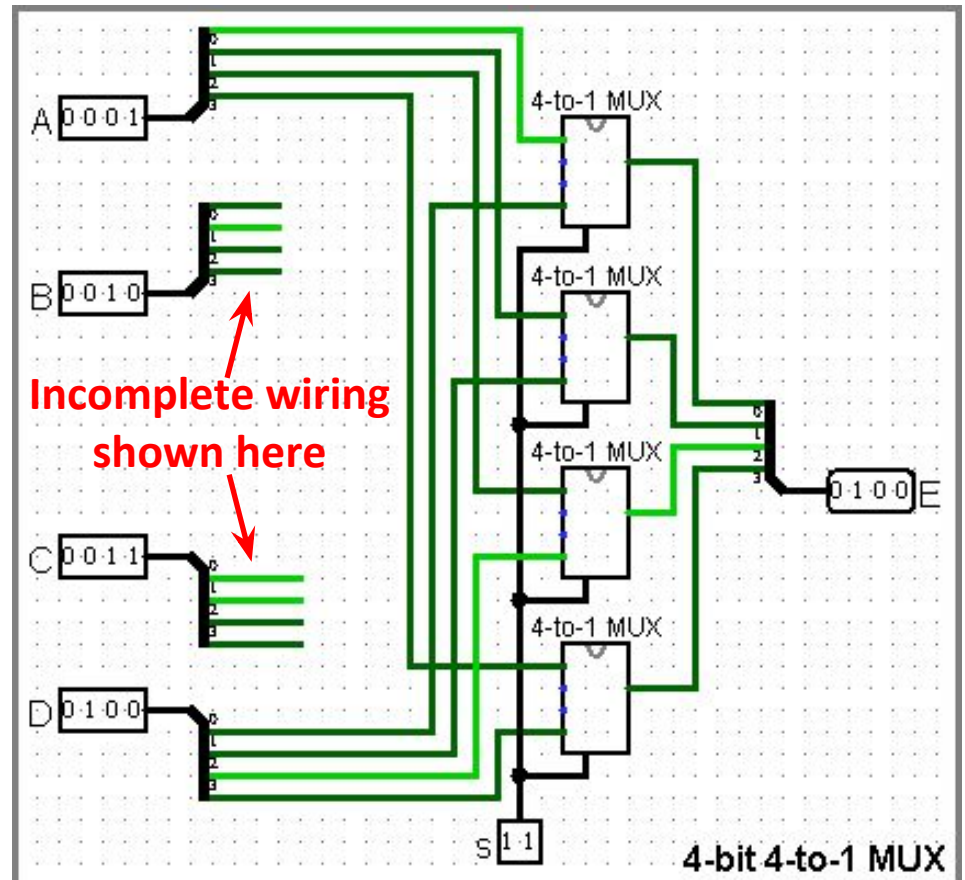
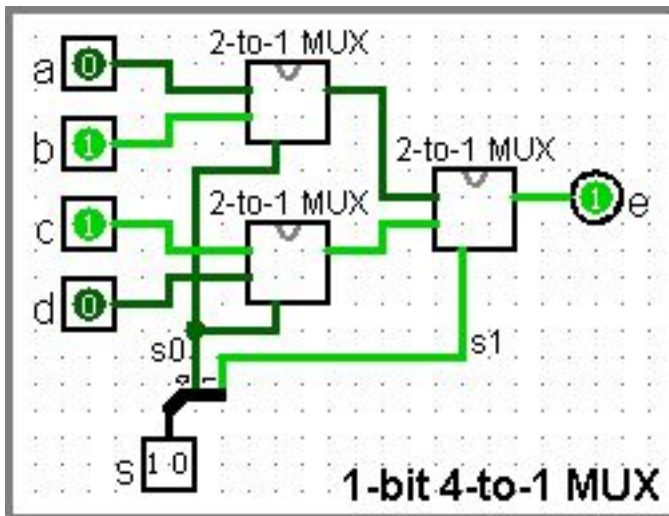
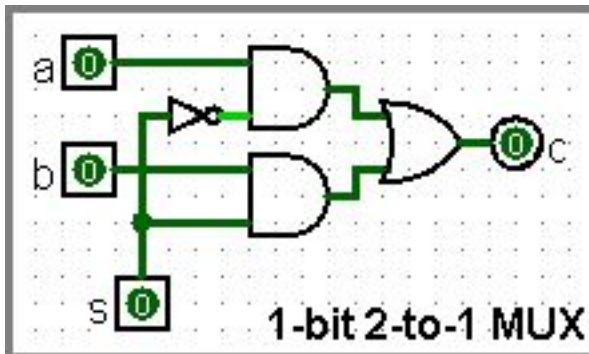
# 1-bit 4-to-1 MUX (2/2)

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



# Subcircuits Example

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



# Meet The Staff



	Damon	Jon
<b>Favorite Villain</b>	Manray	
<b>Protest</b>	No pineapples on pizza	Bring back L8 night
<b>What are you passionate about?</b>	Music	
<b>What would you want to be famous for?</b>	Chess	

# Agenda

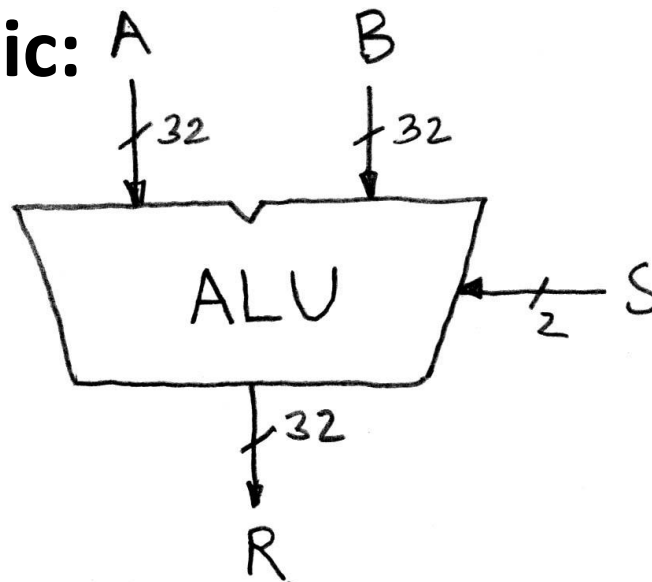
- Critical Path and Clock Frequency
- Administrivia
- Finite State Machines
- Multiplexers
- **ALU Design**
  - **Adder/Subtracter**
- Bonus:
  - Pipelining intro
  - Handling overflow
  - Logisim Introduction



# Arithmetic and Logic Unit (ALU)

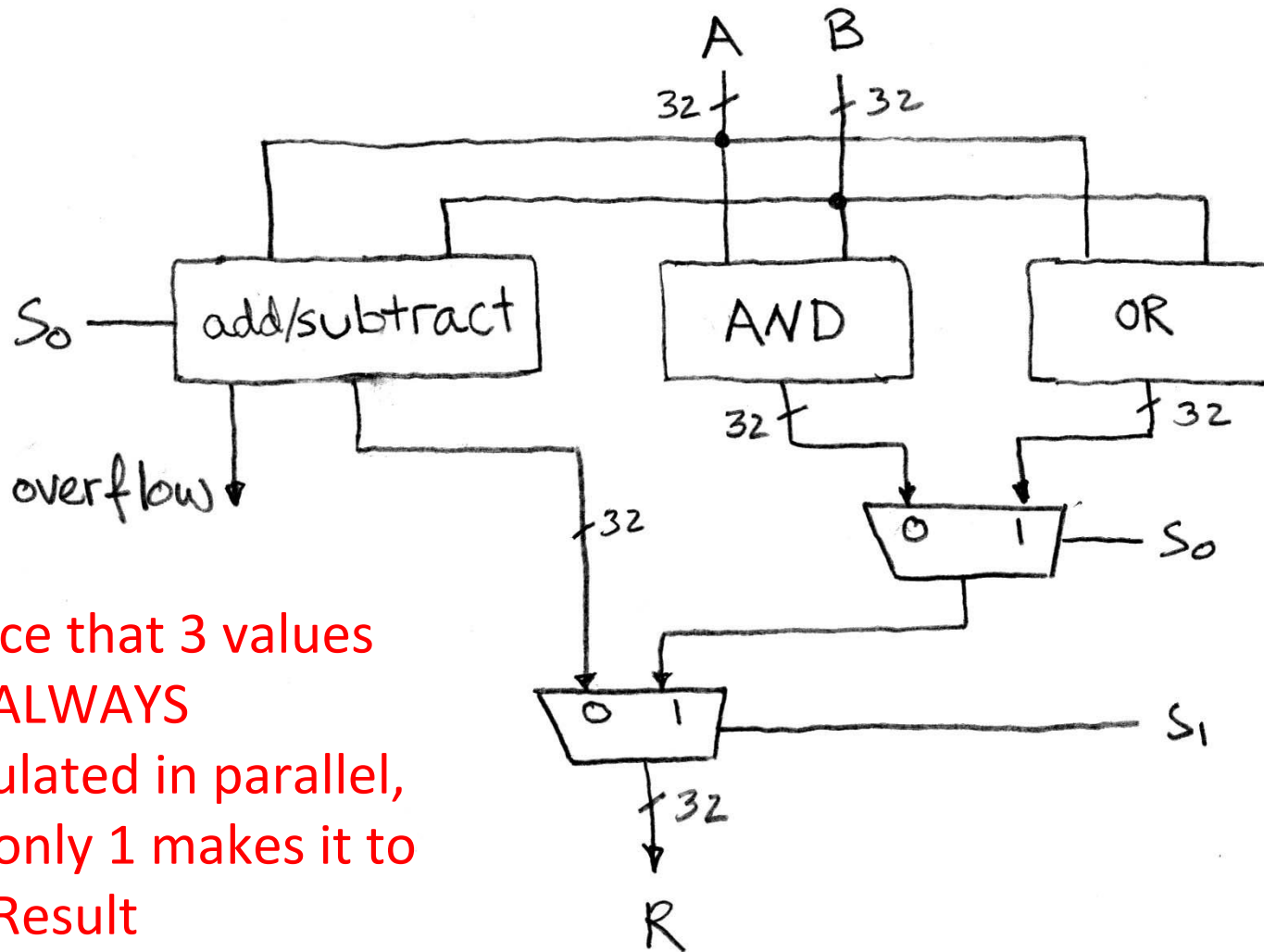
- Most processors contain a special logic block called the “Arithmetic and Logic Unit” (ALU)
  - We’ll show you an easy one that does ADD, SUB, bitwise AND, and bitwise OR

- **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) CL design we've seen before: write out truth table, convert to Boolean, minimize logic, then implement
  - How big might truth table and/or Boolean expression get?
- 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

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

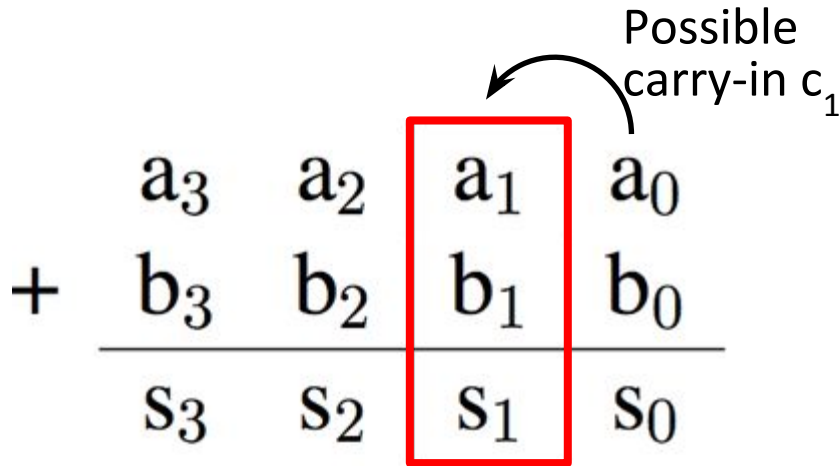
Carry-out bit 

$a_0$	$b_0$	$s_0$	$c_1$
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

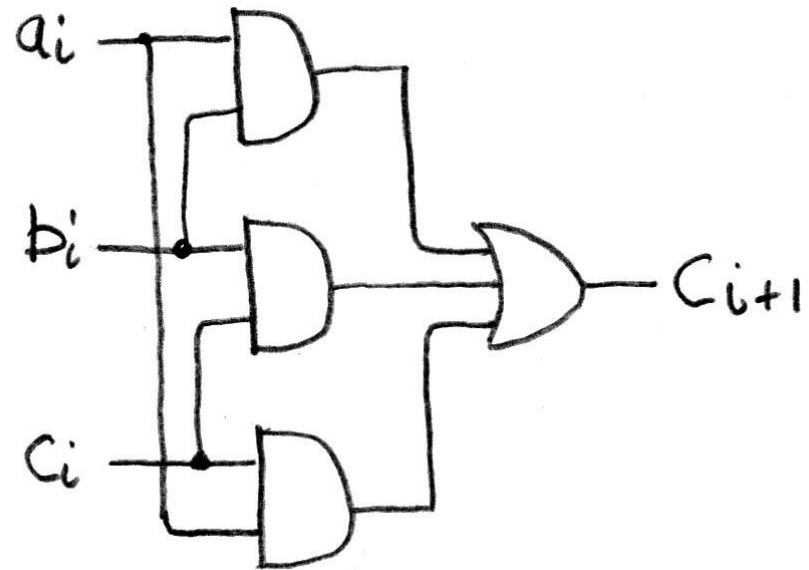
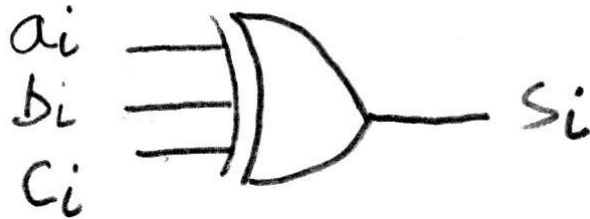
Here defining XOR of many inputs to be 1 when an *odd* number of inputs are 1

$$s_i = \text{XOR}(a_i, b_i, c_i)$$

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

# Adder/Subtractor: 1-bit Adder

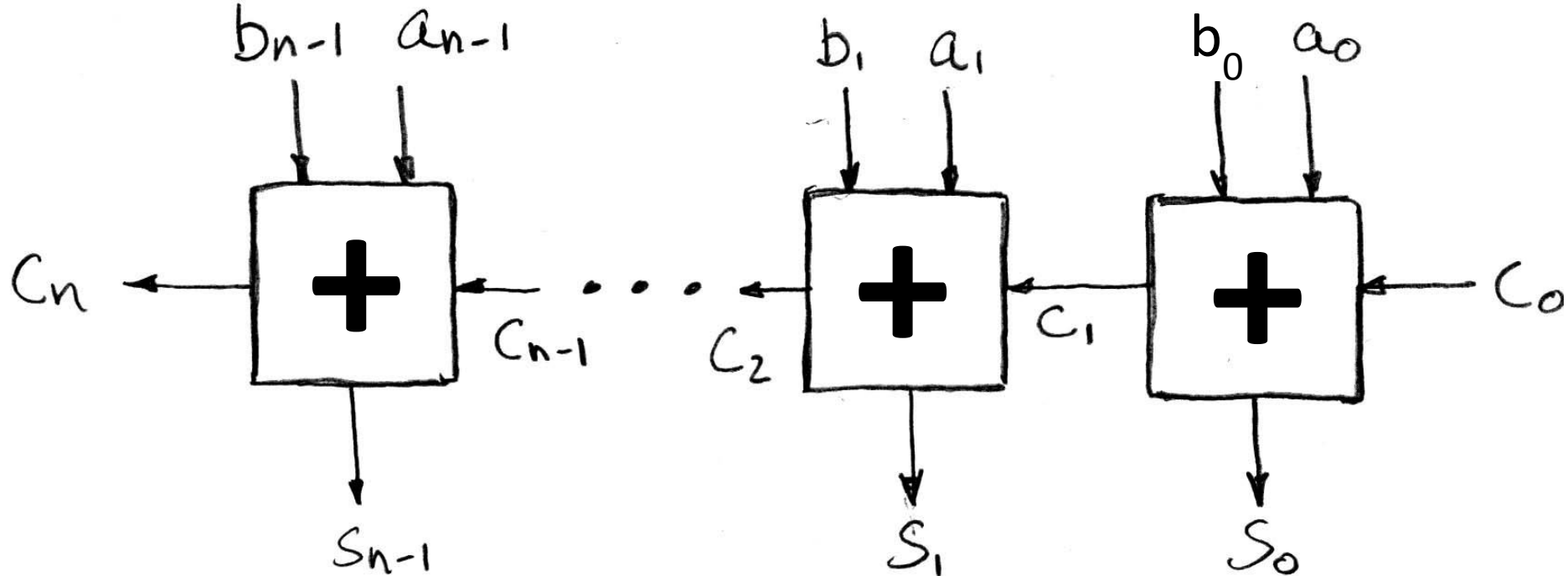
- **Circuit Diagrams:**



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

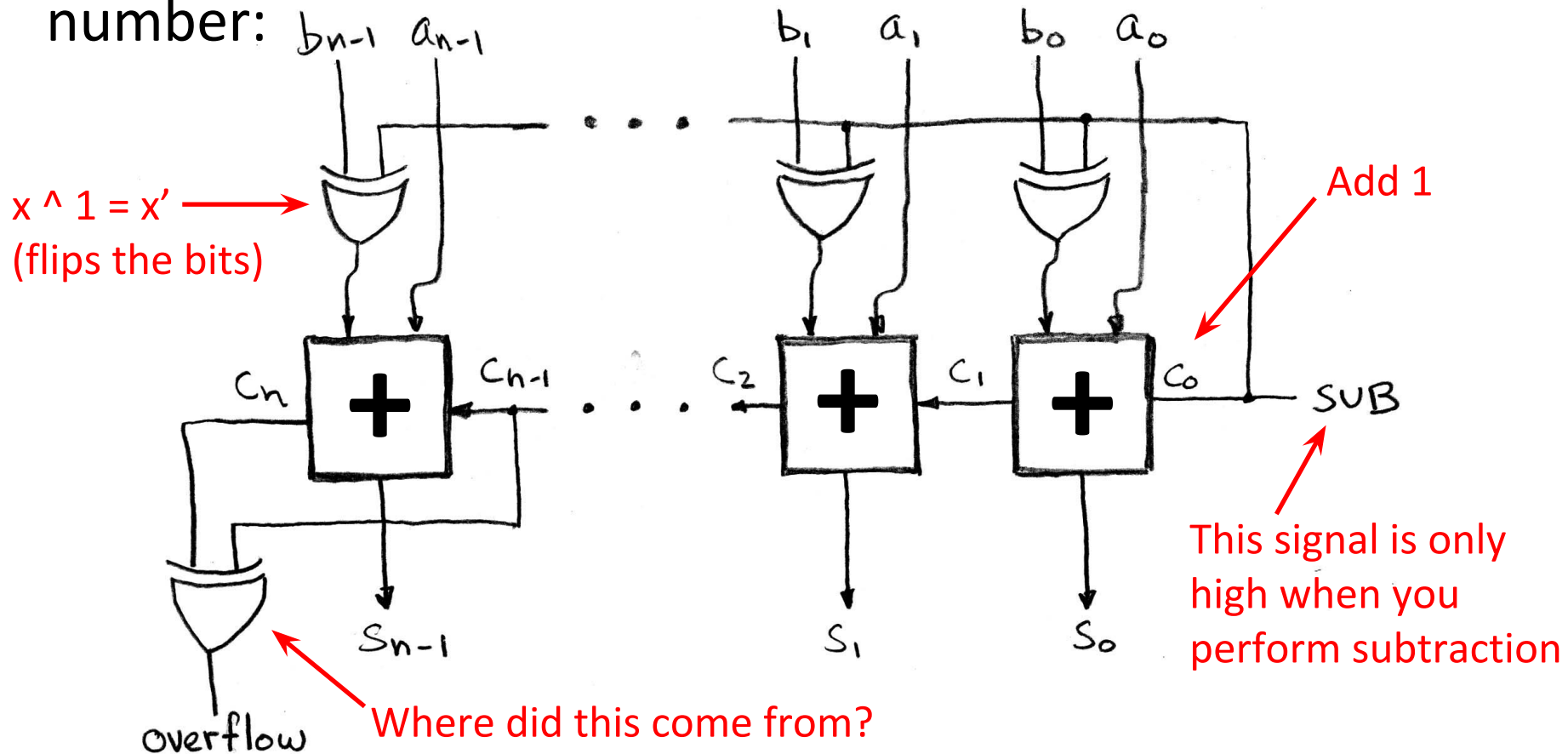
# N x 1-bit Adders → N-bit Adder

- Connect CarryOut<sub>*i*-1</sub> to CarryIn<sub>*i*</sub> to chain adders:



# Two's Complement Adder/Subtractor

- Subtraction accomplished by adding negated number:





# Summary

- Hardware systems are constructed from *Stateless* Combinational Logic and *Stateful* “Memory” Logic (registers)
- State registers implemented from Flip-flops

# Summary

- Critical (longest) path constrains clock rate
  - Need  $t_{hold} \leq t_{input} \leq t_{period} - t_{setup}$
  - Can adjust with extra registers (*pipelining*)
- Finite State Machines visualize state-based computations
  - Can implement systems with Register + CL
- Use MUXes to select among input
  - S input bits selects one of  $2^S$  inputs
  - Each input is a bus n-bits wide
- Build n-bit adder out of chained 1-bit adders
  - Can also do subtraction with additional SUB signal

# 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.

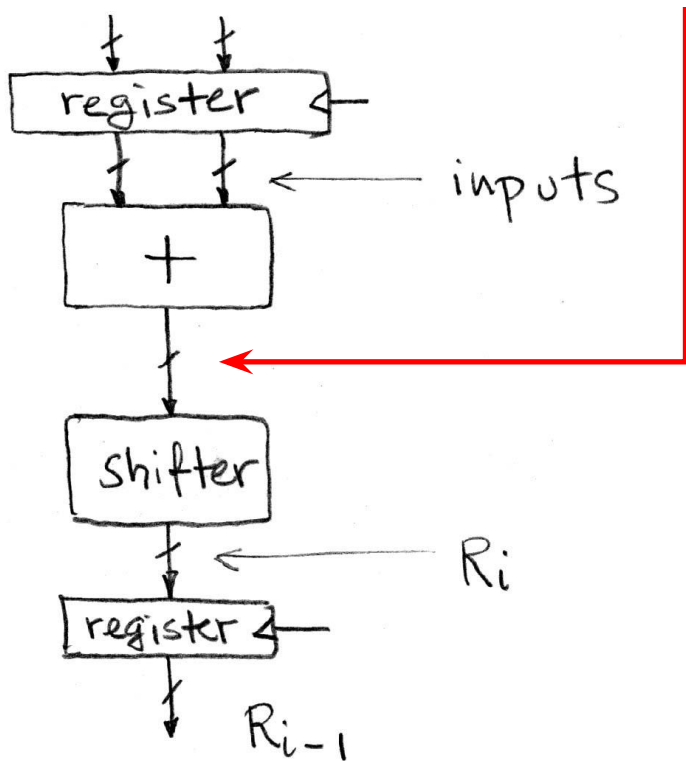
They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.

# Agenda

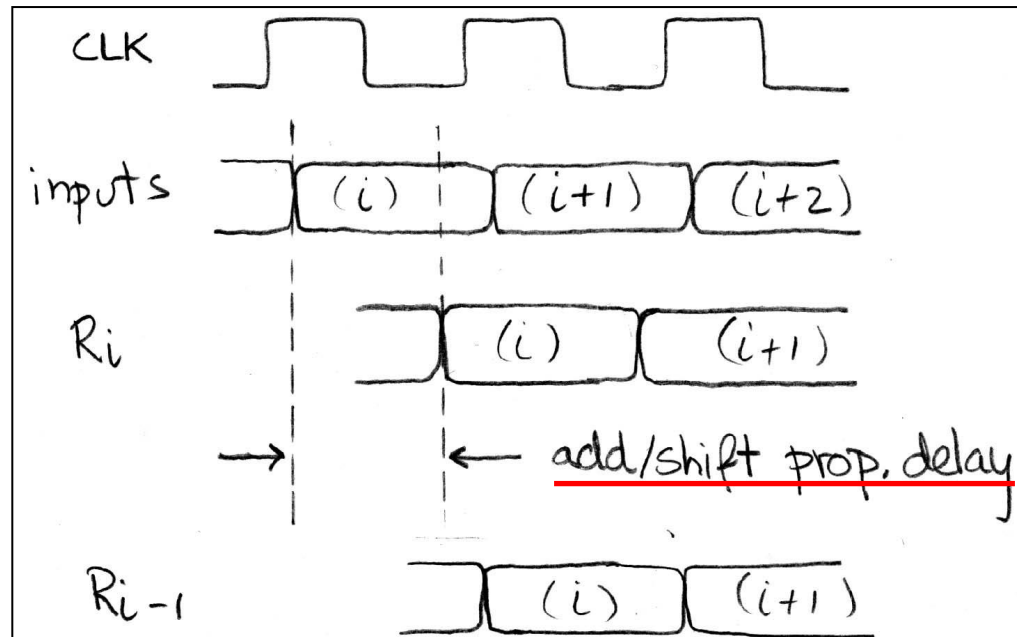
- Critical Path and Clock Frequency
- Administrivia
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter
- Bonus:
  - **Pipelining intro**
  - Handling overflow
  - Logisim Introduction

# Pipelining and Clock Frequency (1/2)

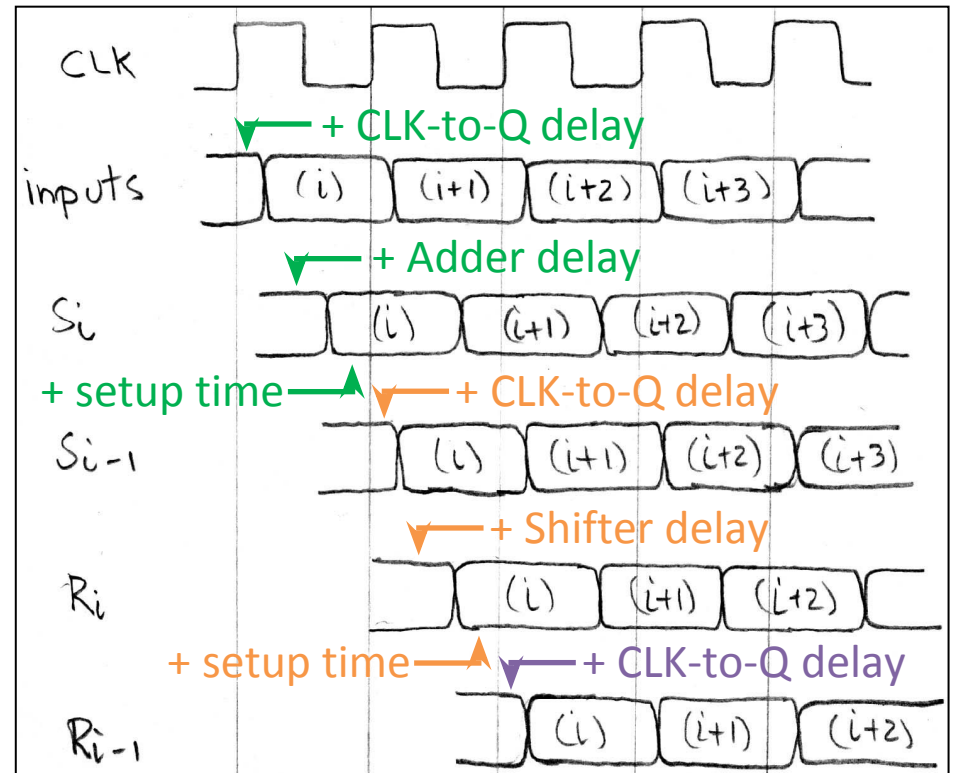
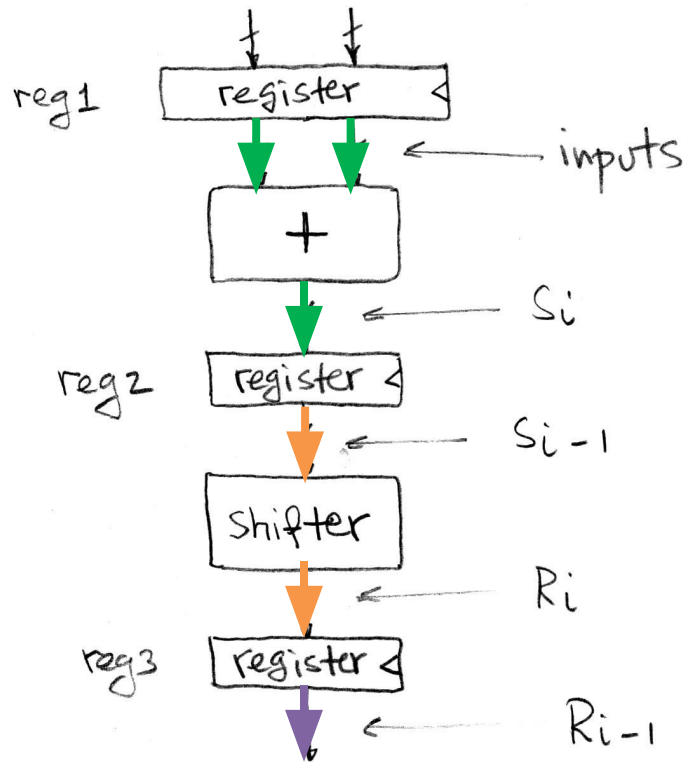
- Clock period limited by propagation delay of adder and shifter
  - Add an extra register to reduce the critical path!



## Timing:



# Pipelining and Clock Frequency (2/2)



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

# 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 (application to CPUs) later

# Agenda

- Critical Path and Clock Frequency
- Administrivia
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter
- Bonus:
  - Pipelining intro
  - **Handling overflow**
  - Logisim Introduction



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

# Agenda

- Critical Path and Clock Frequency
- Administrivia
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter
- Bonus:
  - Pipelining intro
  - Handling overflow
  - **Logisim Introduction**

# 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 2.7.1)
- No programming involved
  - Unlike Verilog, which is a hardware description language (HDL)
  - Click and drag; still has its share of annoying quirks

# Gates in Logisim

**Gates**

- NOT Gate
- Buffer
- AND Gate
- OR Gate
- NAND Gate
- NOR Gate
- XOR Gate
- XNOR Gate
- Odd Parity
- Even Parity
- Controlled Buffer
- Controlled Inverter

**Plexers**

**Arithmetic**

**Memory**

**Input/Output**

**Base**

Types of Gates

Options

↓

**Selection: AND Gate**

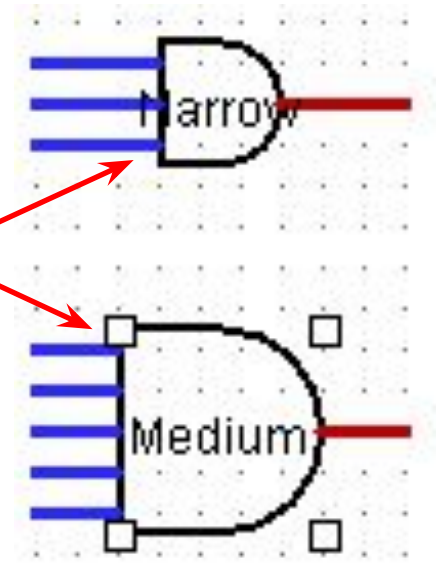
Facing	East
Data Bits	1
Gate Size	Medium
Number Of Inputs	5
Output Value	0/1
Label	Medium
Label Font	SansSerif Plain 12
Negate 1 (Top)	No
Negate 2	No
Negate 3	No
Negate 4	No
Negate 5 (Bottom)	No

← bus width n

← # inputs

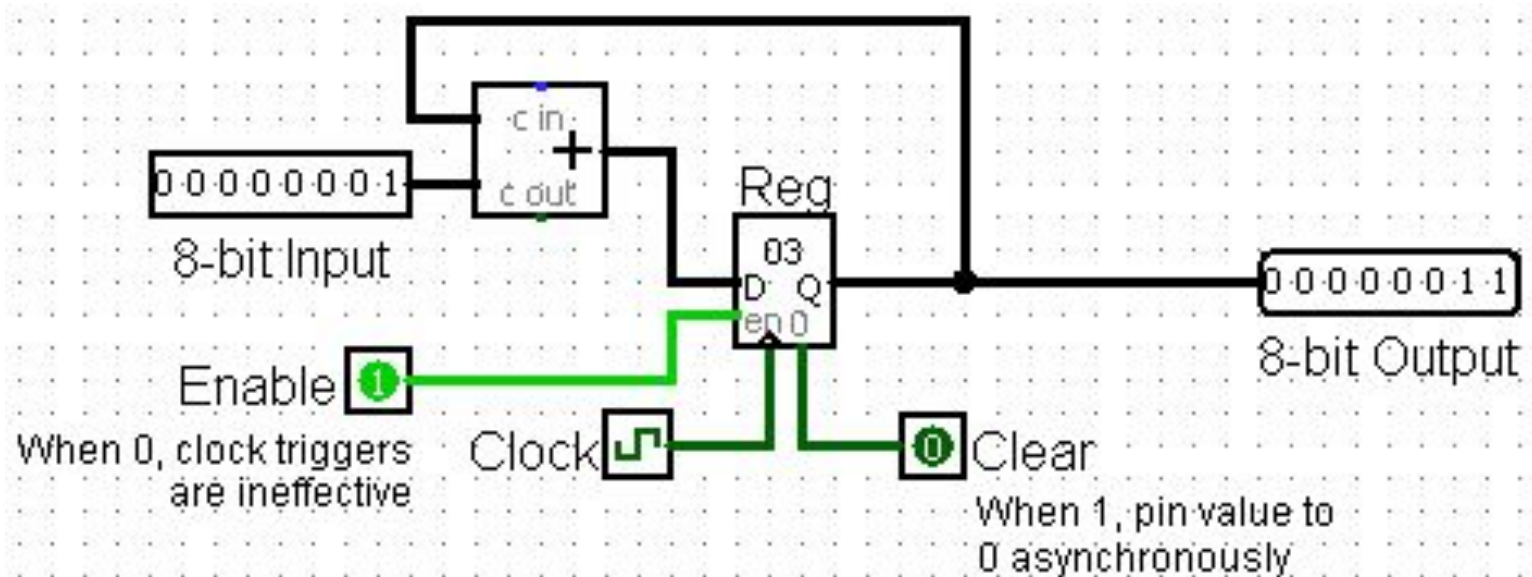
← labeling not necessary, but can help

- Click gate type, click to place
  - Can set options *before* placing or select gate *later* to change



# Registers in Logisim

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

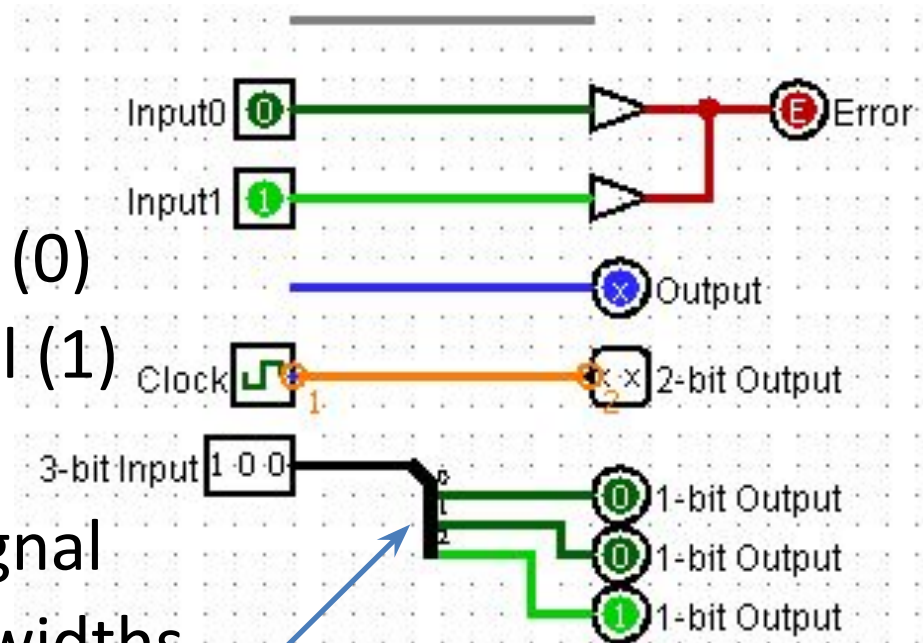


# Wires in Logisim

- 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



“Splitter” used to adjust bus widths

- **Tunnels:** all tunnels with same label are connected



# 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:

