# CS 61C: Great Ideas in Computer Architecture

## *Functional Units, Finite State Machines*

**Instructor:** Justin Hsia

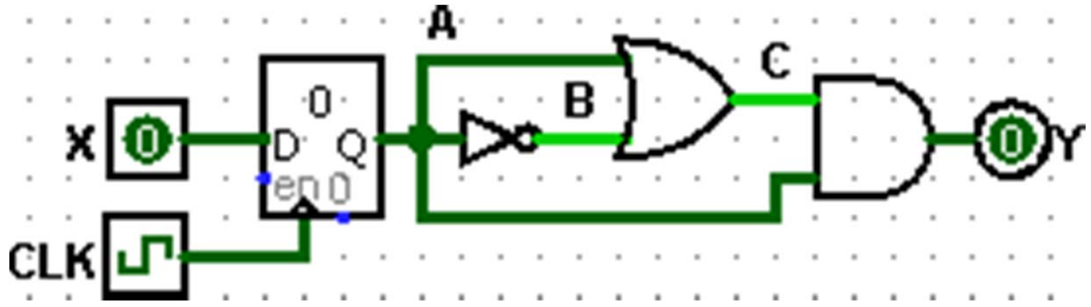
## Review of Last Lecture

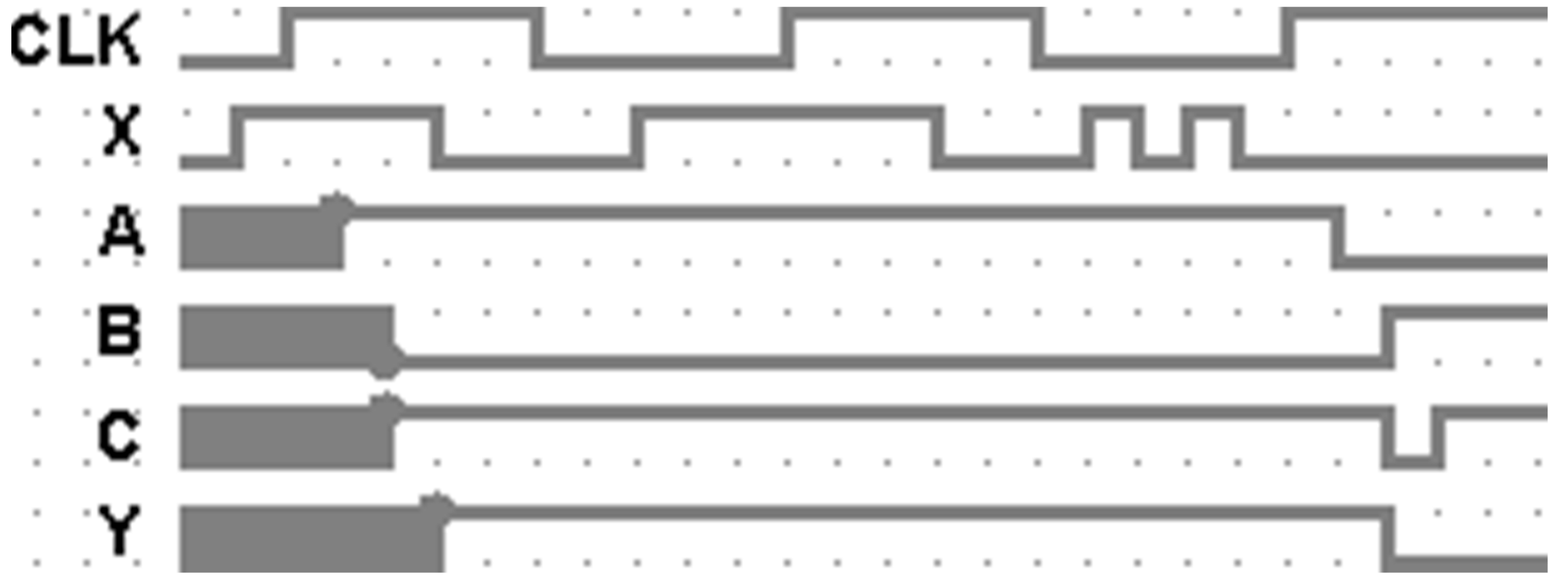
- Synchronous Digital Systems
  - Pulse of a Clock controls flow of information
  - All signals are seen as either 0 or 1
- Hardware systems are constructed from *Stateless* Combinational Logic and *Stateful* “Memory” Logic (registers)
- Combinational Logic: equivalent circuit diagrams, truth tables, and Boolean expressions
  - Boolean Algebra allows minimization of gates
- State registers implemented from Flip-flops

## Dealing with Waveform Diagrams

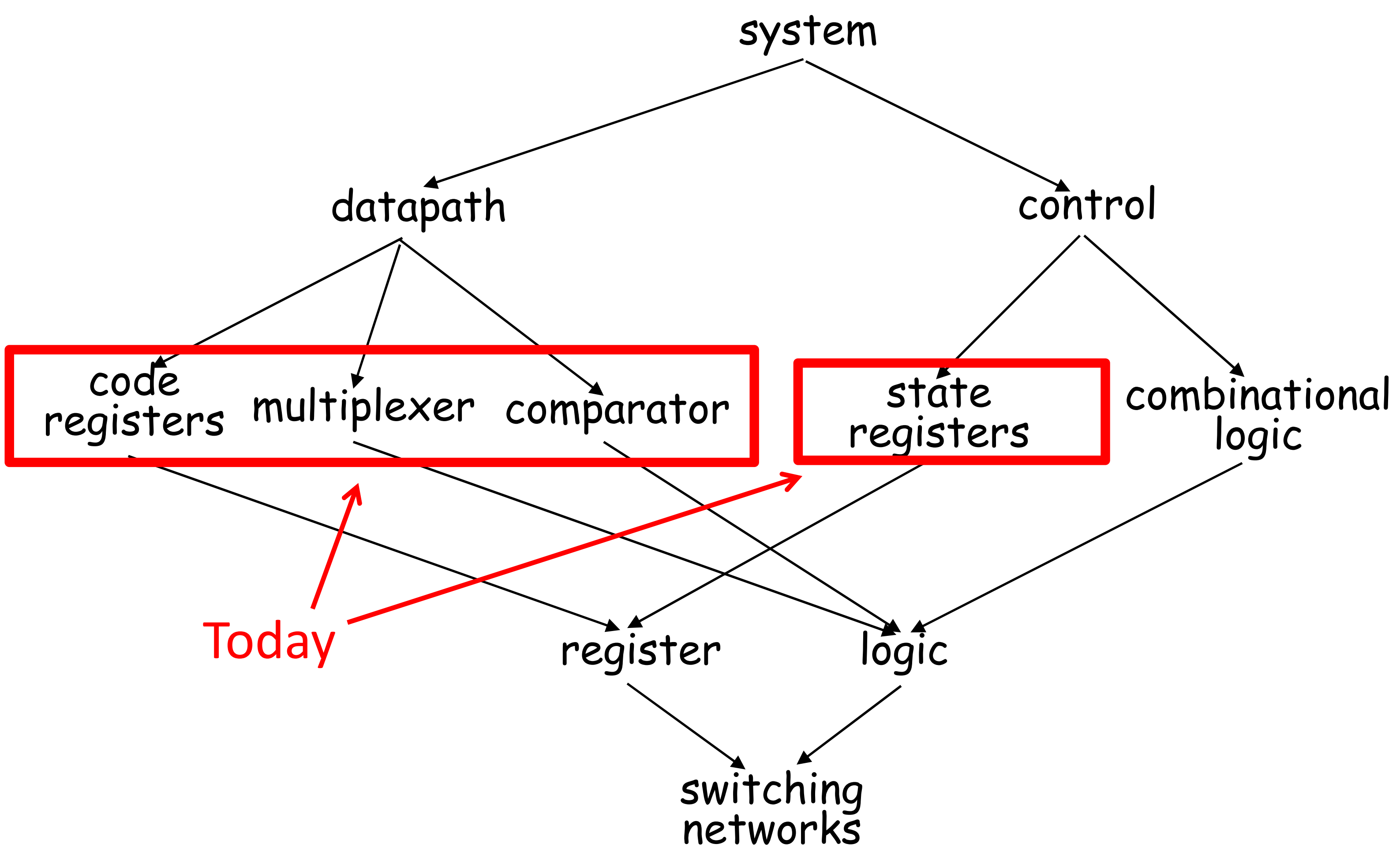
- Easiest to start with CLK on top
  - Solve signal by signal, from inputs to outputs
  - Can only draw the waveform for a signal if *all* of its input waveforms are drawn
- When does a signal update?
  - A *state element* updates based on CLK triggers
  - A *combinational element* updates ANY time ANY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

**Example:** T = 10 ns. $t_{setup} = t_{hold} = 0$. $t_{clk\text{-}to\text{-}q} = 1$ ns. $t_{prop} = 1$ ns for all gates. Each "tick" below is 1 ns.
*Solve for the waveform of the output Y.*





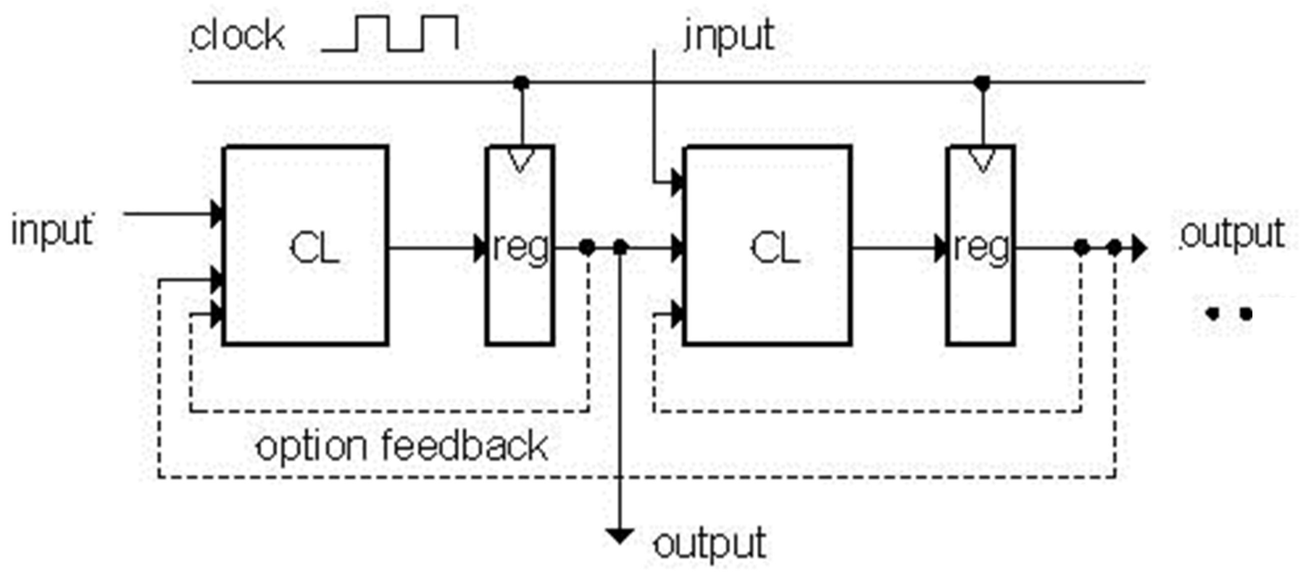
# Hardware Design Hierarchy



## Agenda

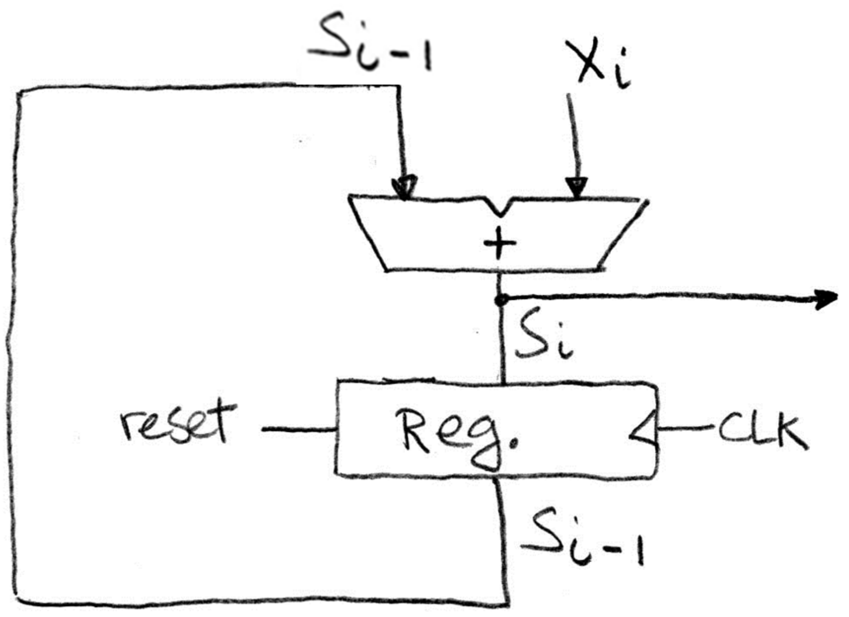
- State Elements Continued
- Administrivia
- Logisim Introduction
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter

# Model for Synchronous Systems

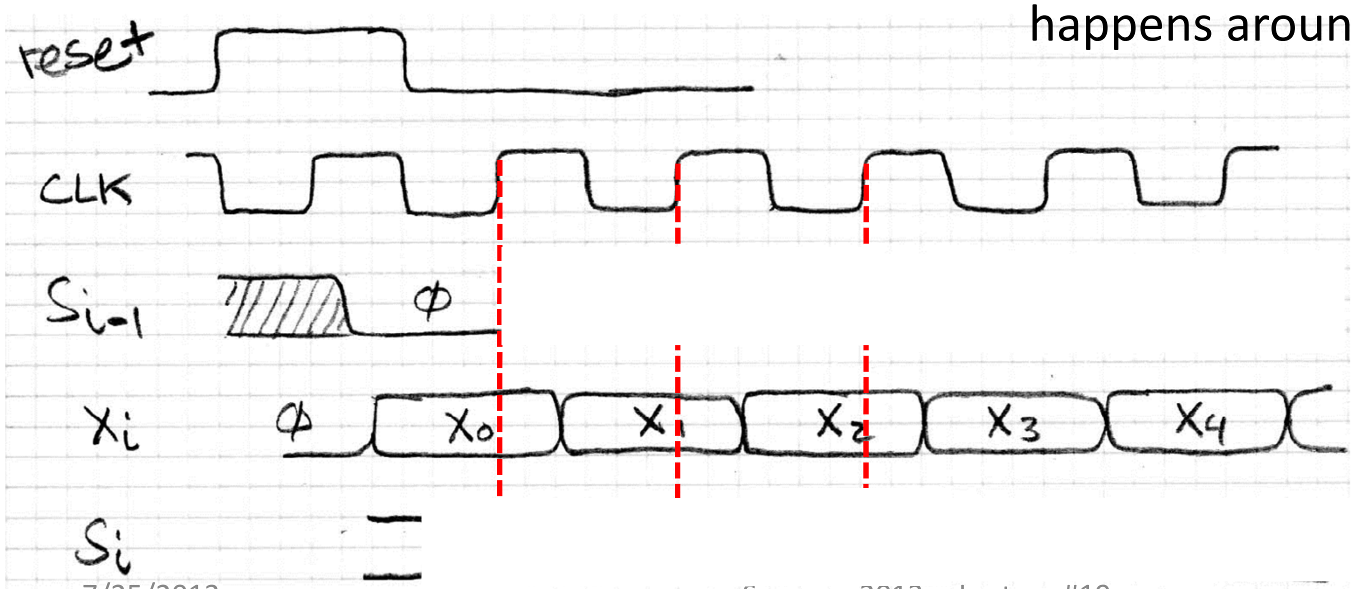


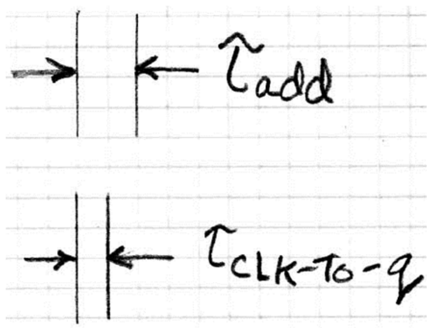
- Collection of Combinational Logic blocks separated by registers
  - Feedback is optional depending on application
- *Clock (CLK):* square wave that synchronizes the system
  - Clock signal connects only to clock input of registers
- *Register:* several bits of state that samples input on rising edge of CLK

# Accumulator Revisited ...Again



- reset signal shown
- In practice X 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



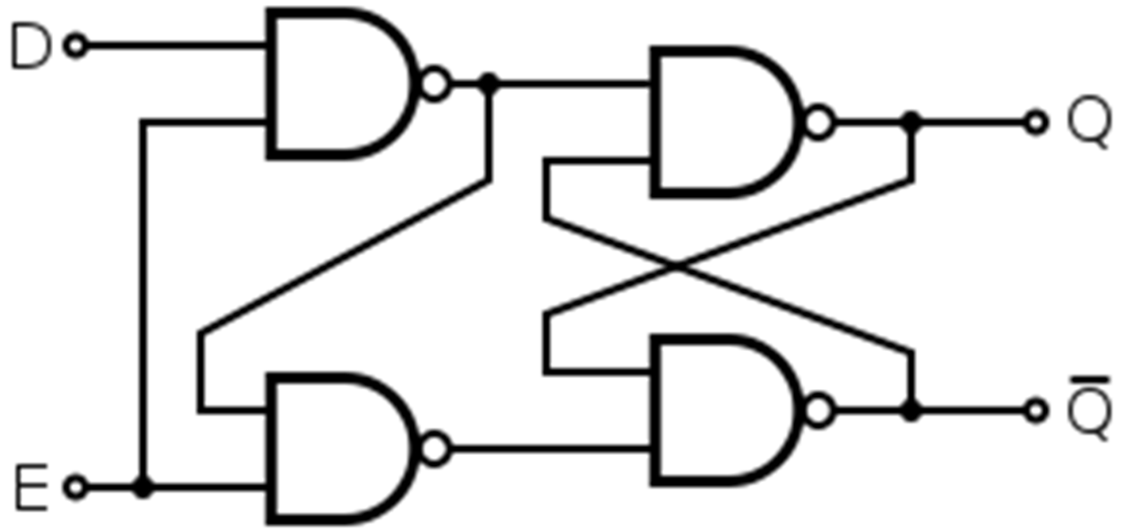


## Register Timing Terms (Review)

- *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

## Where Does Timing Come From?

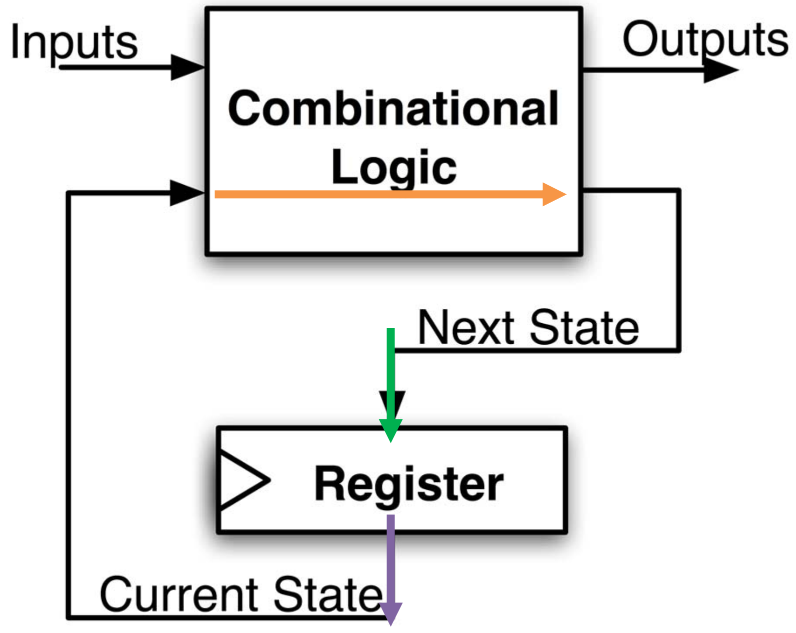
- Example D flip-flop implementation:



- Changing the D signal around the time E (CLK) changes can cause unexpected behavior

# Maximum Clock Frequency

- What is the max frequency of this circuit?
  - Limited by how much time needed to get correct Next State to Register

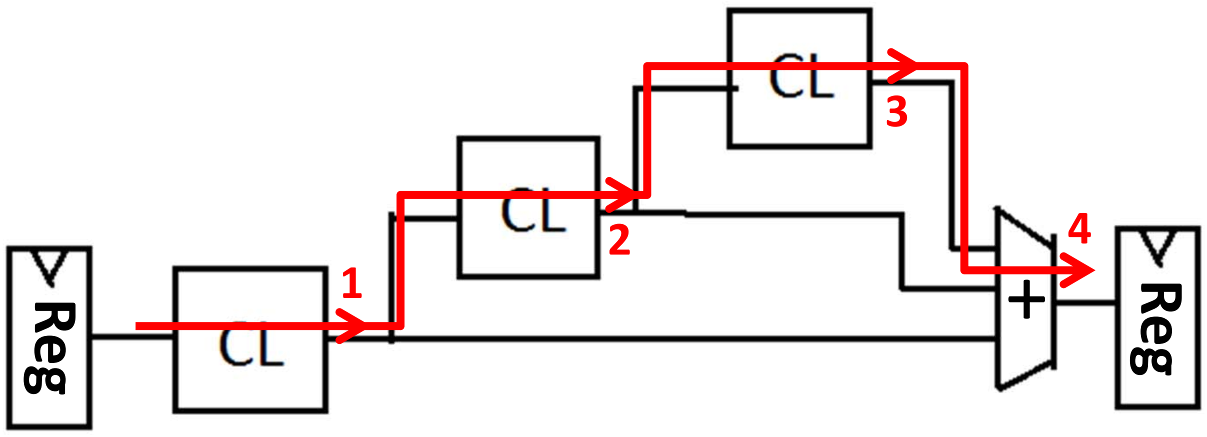


Max Delay = Setup Time
+ CLK-to-Q Delay
+ CL Delay

Max Freq = 1/Max Delay

# The Critical Path

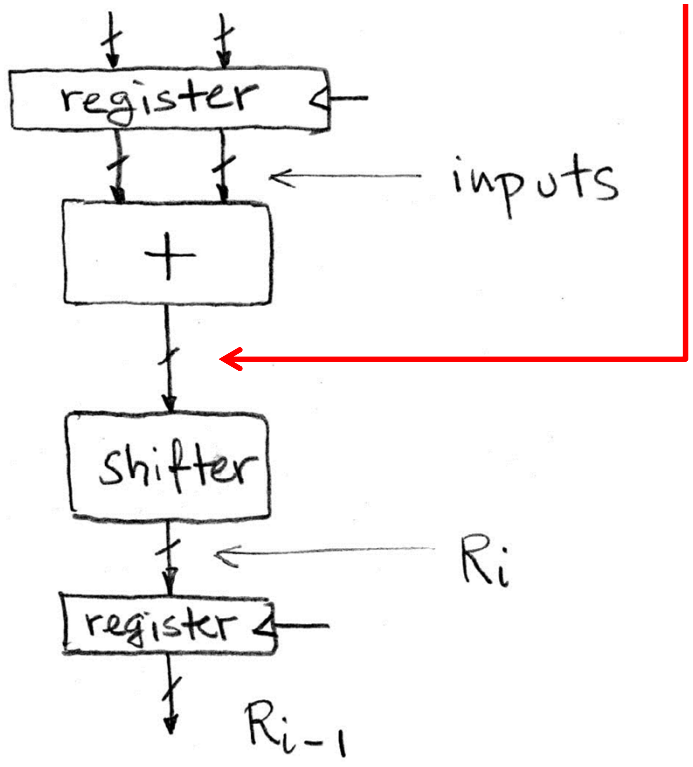
- The *critical path* is the longest delay between *any* two registers in a circuit
- The clock period must be *longer* than this critical path, or the signal will not propagate properly to that next register



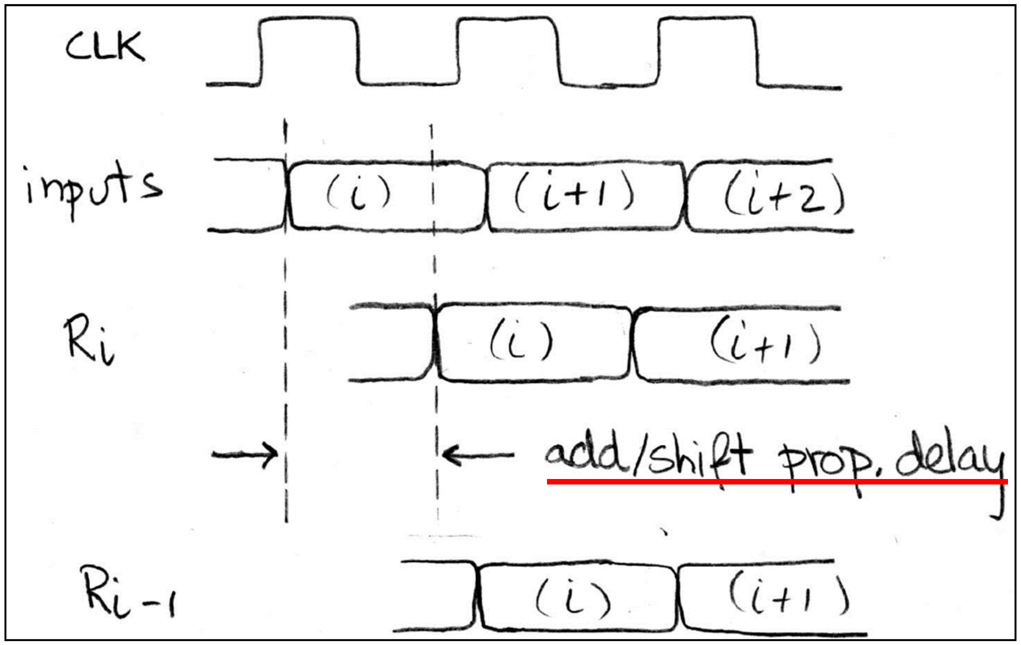
**Critical Path =**
CL Delay 1
+ CL Delay 2
+ CL Delay 3
+ Adder Delay

# 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!

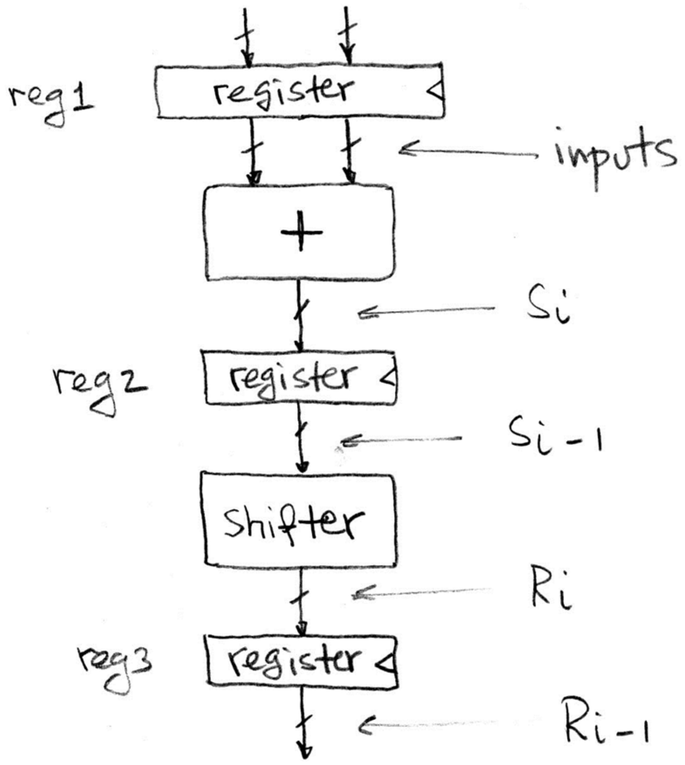


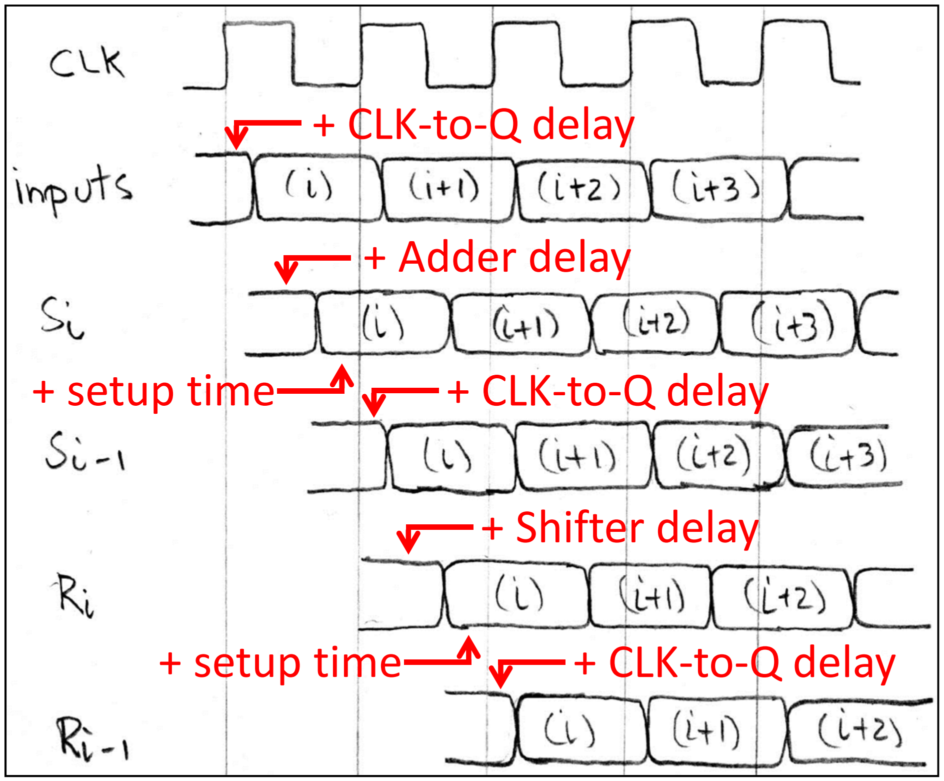
**Timing:**



# Pipelining and Clock Frequency (2/2)

- Extra register allows higher clock freq (more outputs per sec)
- However, takes two (shorter) clock cycles to produce first output (higher latency for initial output)





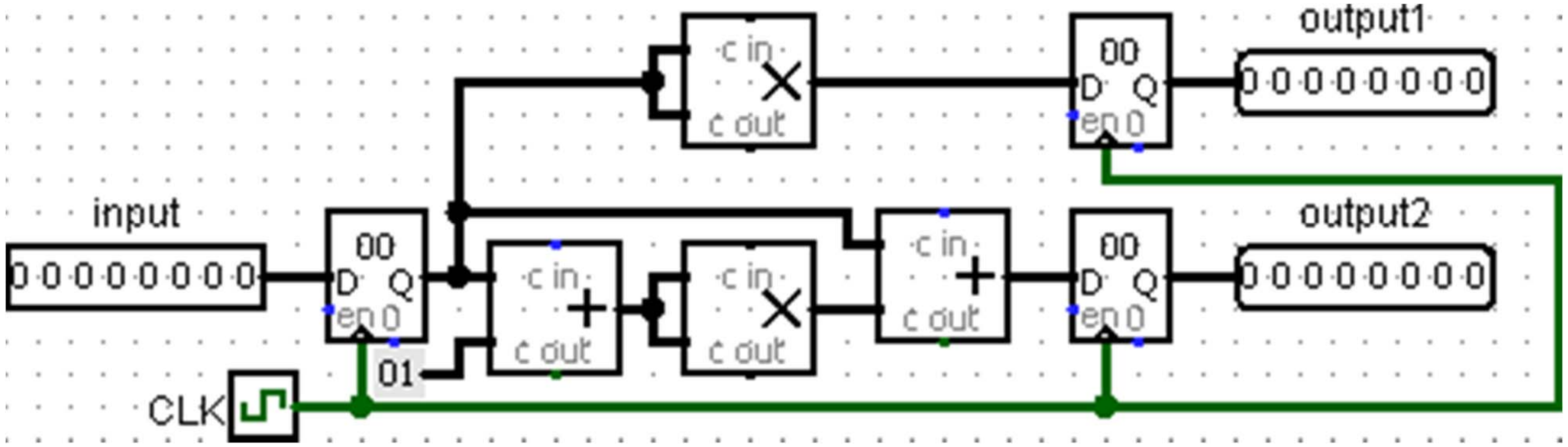
# Pipelining Basics

- 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) next week

**Question:** Want to run on 1 GHz processor.

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

What is the maximum $t_{clk\text{-}to\text{-}q}$ we can use?



[✓ (A) 550 ps]

(B) 750 ps

(C) 500 ps

(D) 700 ps

## Agenda

- State Elements Continued
- Administrivia
- Logisim Introduction
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter

## Administrivia

- Midterm re-grade requests due today
  - We will re-grade the entire test
- Project 2 Part 2 will be posted today
- HW5 posted today, due **Thu Aug. 1**

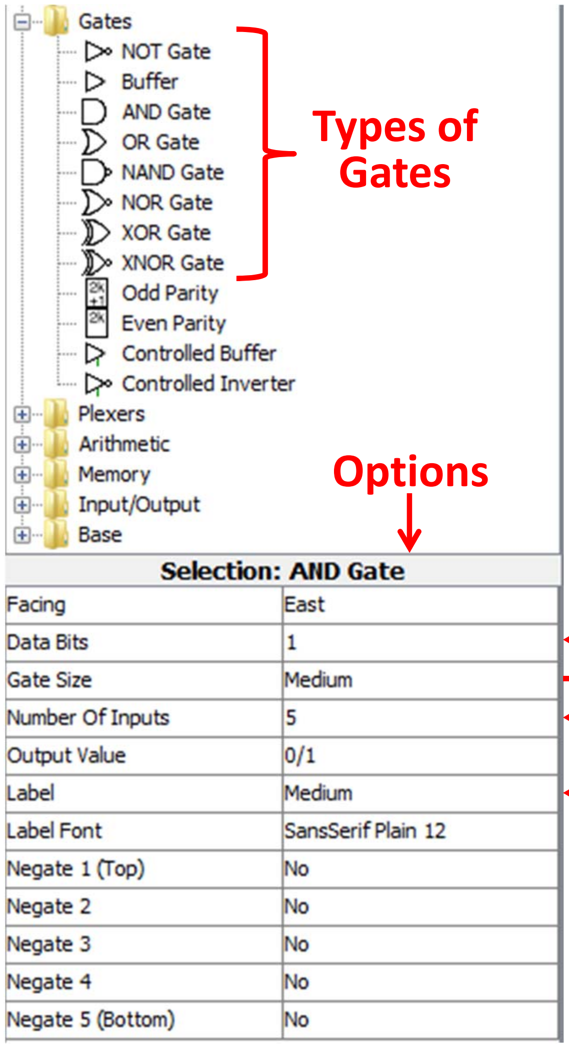
## Agenda

- State Elements Continued
- Administrivia
- Logisim Introduction
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter

## 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
- http://ozark.hendrix.edu/~burch/logisim/

# Gates in Logisim



**Selection: AND Gate**

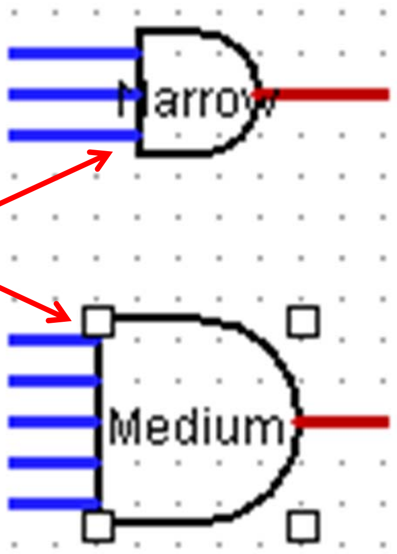
| | |
|---|---|
| Facing | East |
| Data Bits | 1 |
| Gate Size | Medium |
| Number Of Inputs | 5 |
| Output Value | 0/1 |
| Label | Medium |
| Label Font | SansSerif Plain 12 |
| Negate 1 (Top) | No |
| Negate 2 | No |
| Negate 3 | No |
| Negate 4 | No |
| Negate 5 (Bottom) | No |

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

bus width n

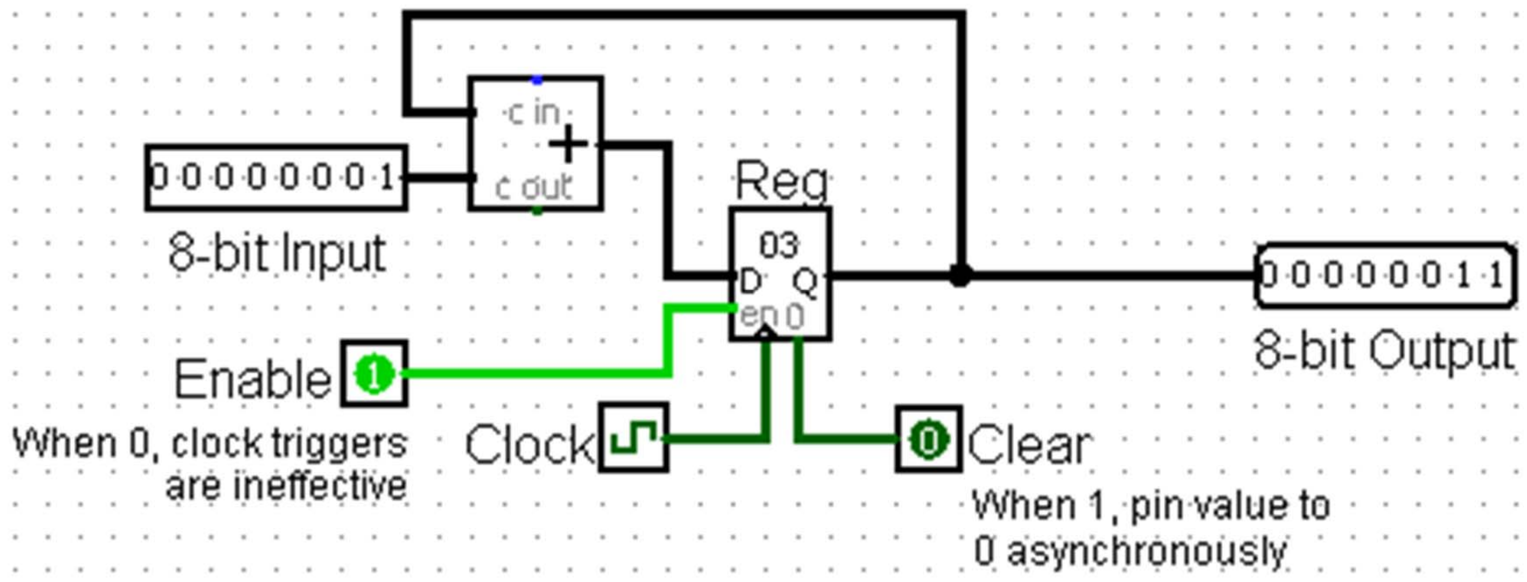
# inputs

labeling not necessary, but can help



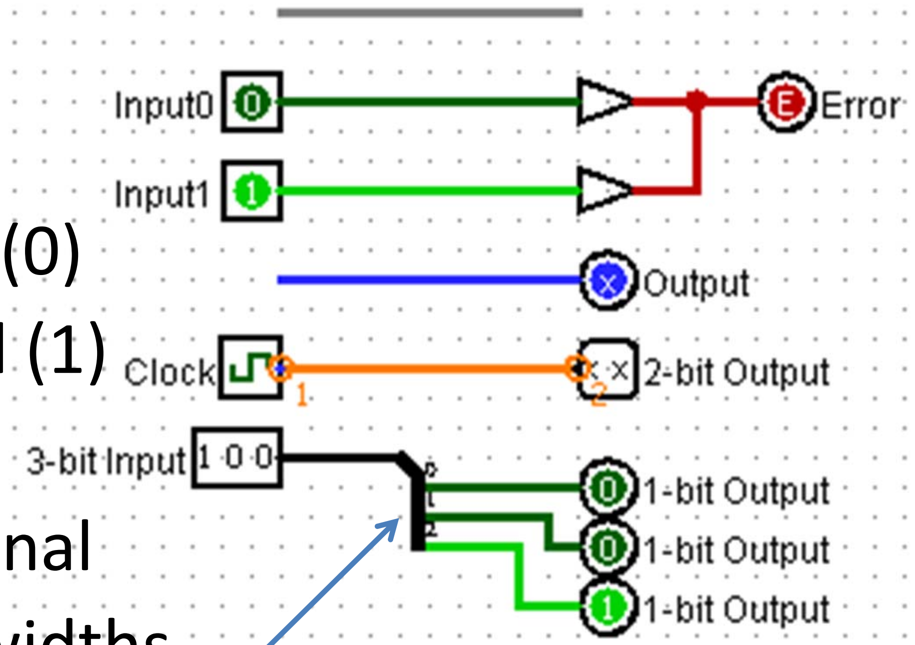
# Registers in Logisim

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



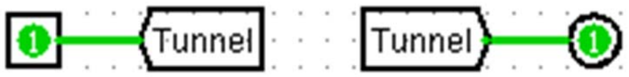
# Wires in Logisim

- 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



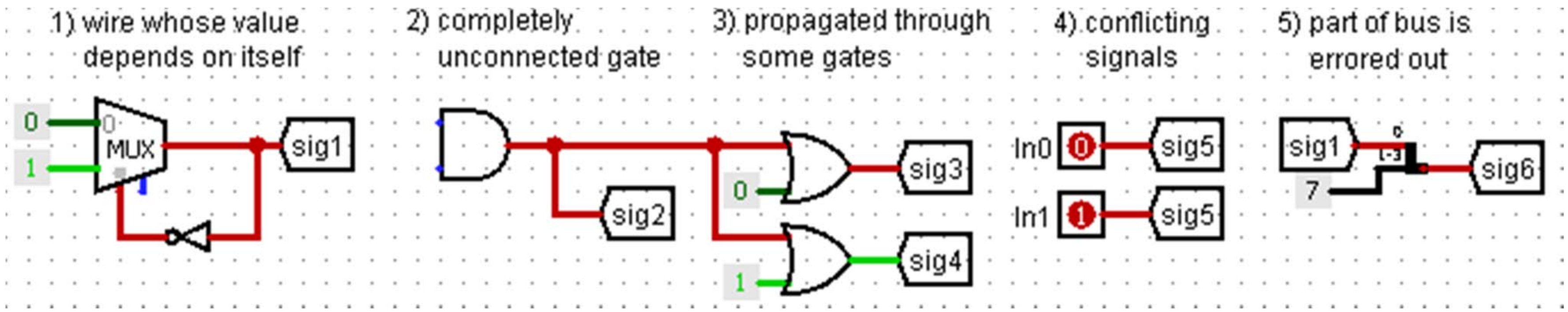
"Splitter" used to adjust bus widths

- **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:

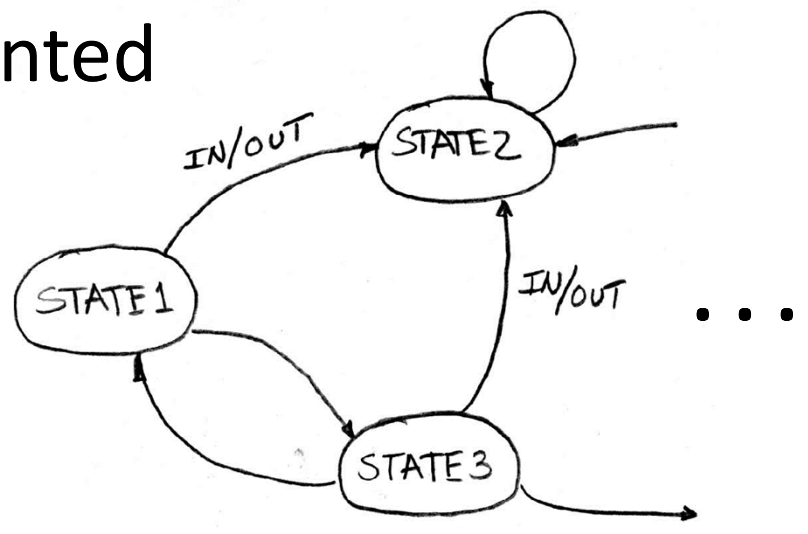

1) wire whose value depends on itself
2) completely unconnected gate
3) propagated through some gates
4) conflicting signals
5) part of bus is errored out
0
1
MUX
sig1
sig2
0
1
sig3
sig4
In0
In1
sig5
sig5
sig1
7
sig6

## Agenda

- State Elements Continued
- Administrivia
- Logisim Introduction
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter

## Finite State Machines (FSMs)

- You may have seen FSMs in other classes
- Function can be represented with a *state transition diagram*
- With combinational logic and registers, any FSM can be implemented in hardware!

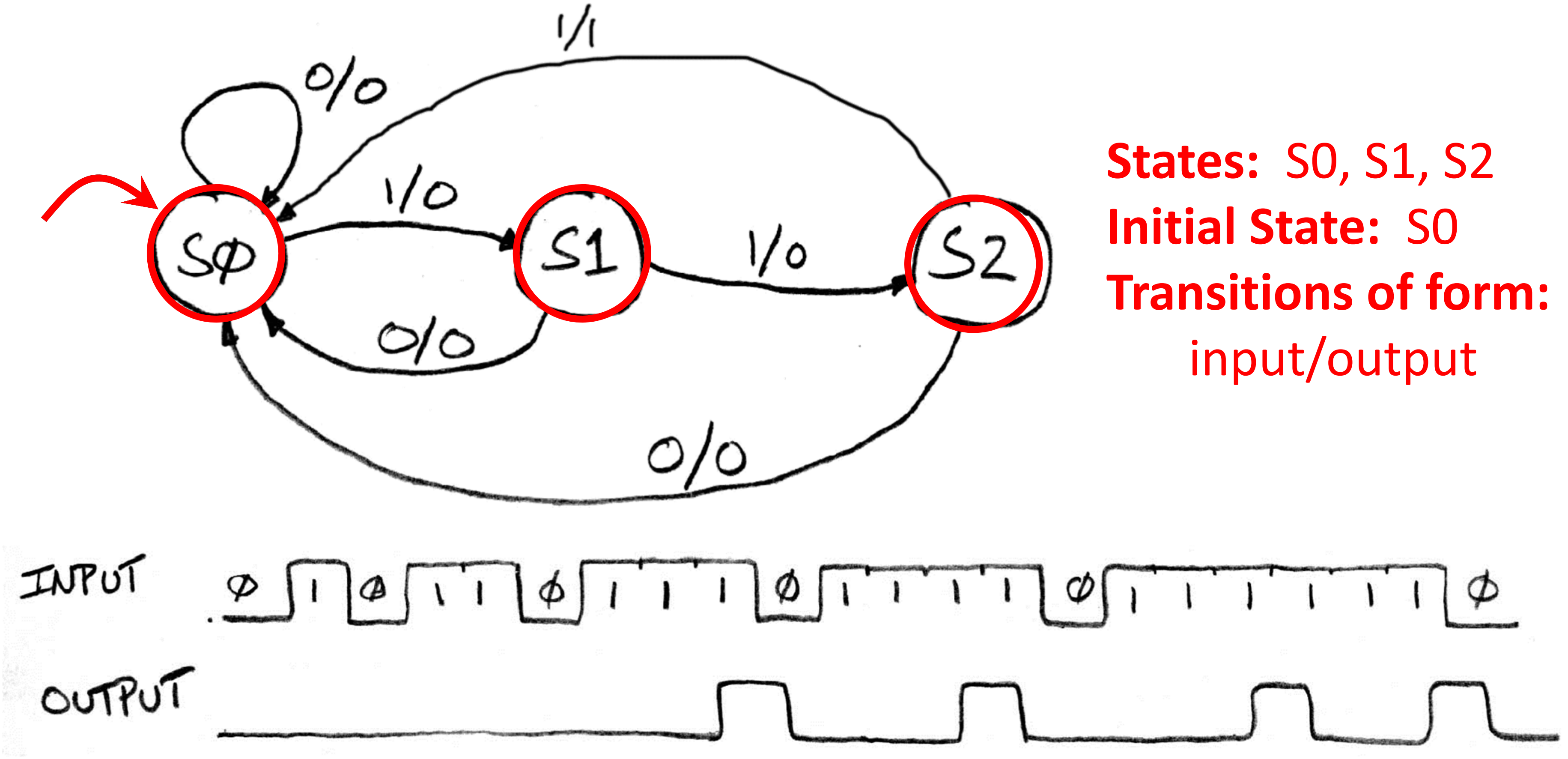


# FSM Overview

- 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

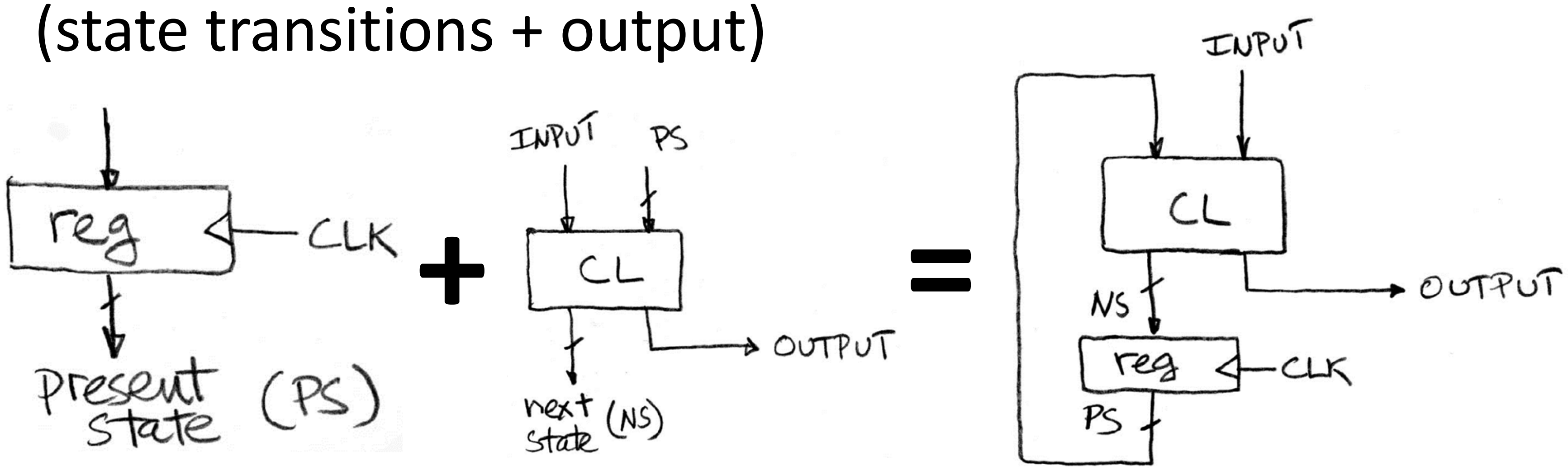
- FSM to detect 3 consecutive 1's in the Input



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

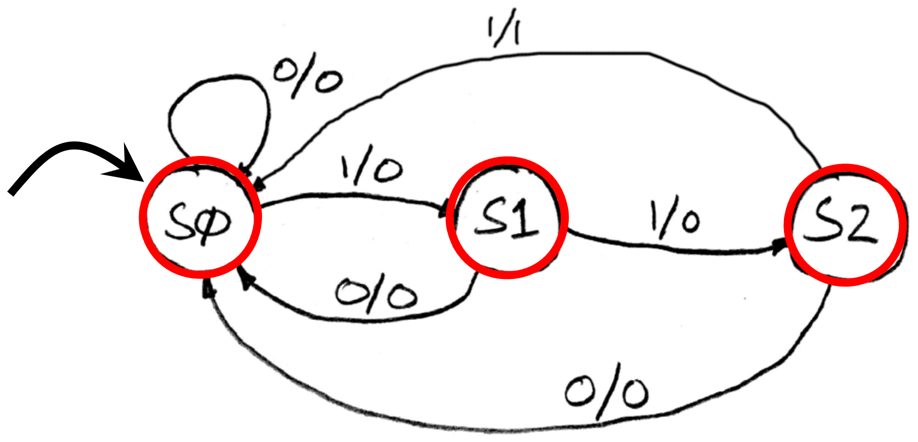
## Hardware Implementation of FSM

- 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:
  - $NS_1 = CS_1'CS_0In + CS_1CS_0In' + CS_1CS_0In$

    Differs by 2

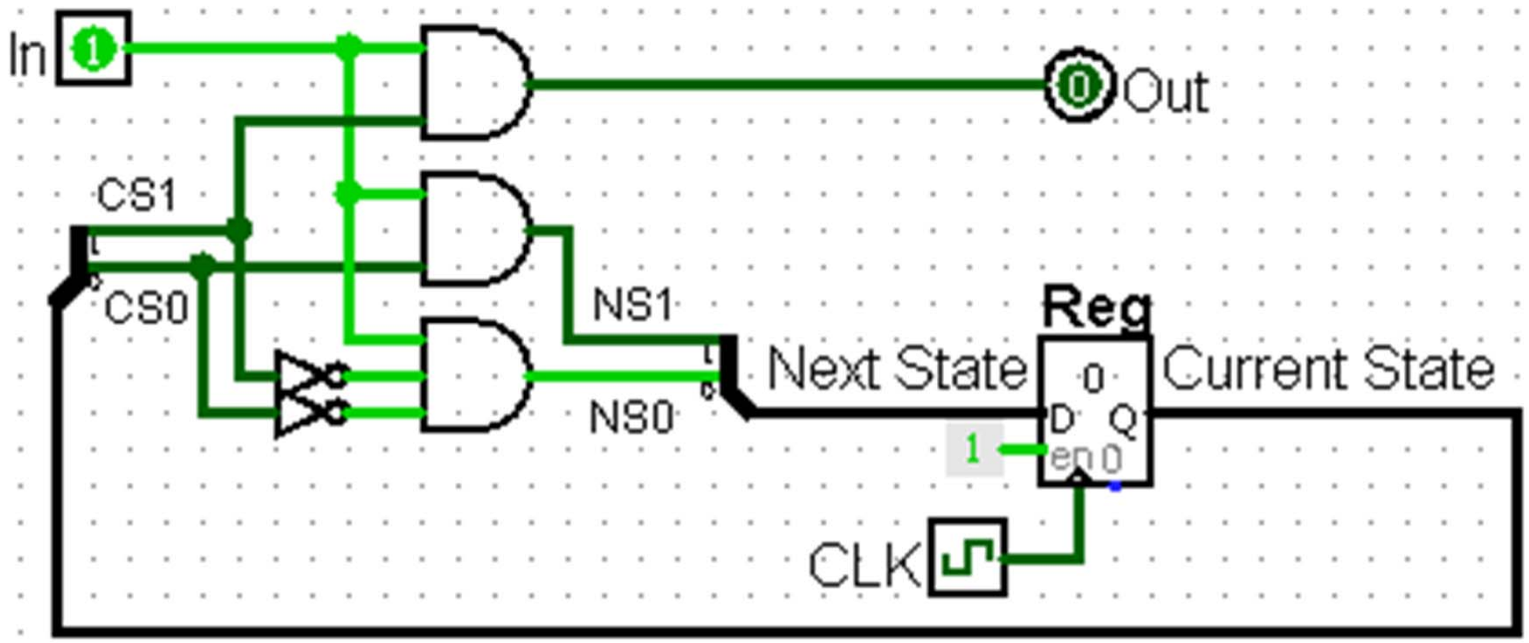
    Is neighbor
  - $NS_1 = CS_0In$
- Karnaugh Map:
  - $NS_1 = CS_0In$

| 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 |

| In \ CS | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | **0** | **0** | **X** | **0** |
| 1 | **0** | **1** | **X** | **0** |

## 3 Ones FSM in Hardware

- 2-bit **Register** needed for state
- **CL:** $NS_1 = CS_0 In$, $NS_0 = CS_1'CS_0'In$, $Out = CS_1 In$

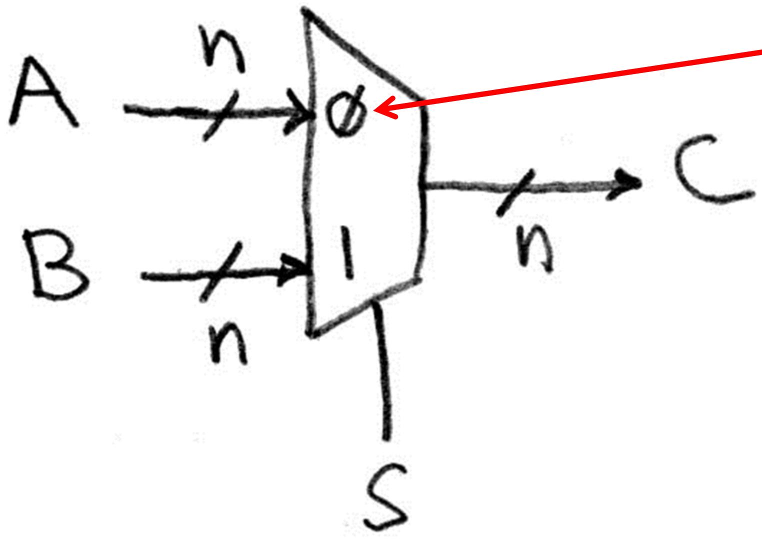


## Agenda

- State Elements Continued
- Administrivia
- Logisim Introduction
- Finite State Machines
- Multiplexers
- ALU Design
  – Adder/Subtracter

## Data Multiplexor

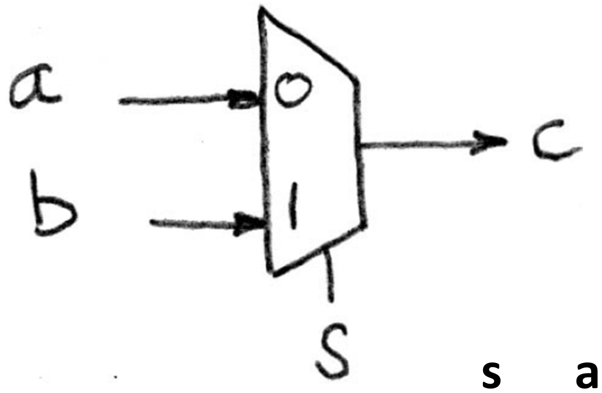
- Multiplexor (“MUX”) is a *selector*
  - Place one of multiple inputs onto output (N-to-1)
- Shown below is an n-bit 2-to-1 MUX
  - Input S selects between two inputs of n bits each



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

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

- **Schematic:**



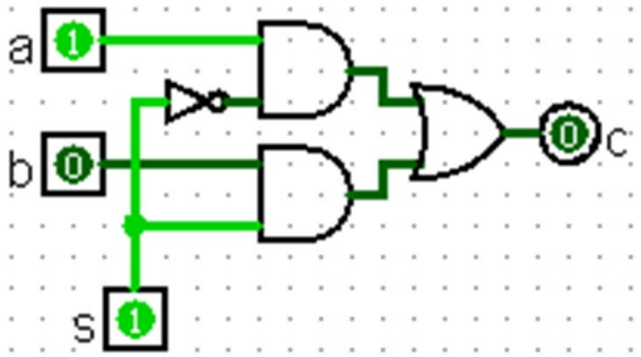
- **Truth Table:**

| 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:**

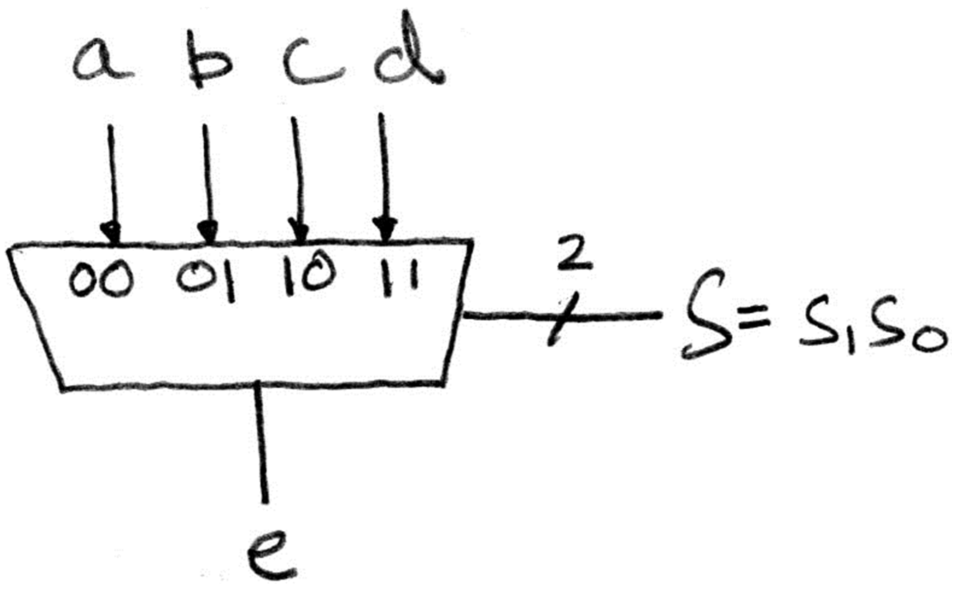
$$\begin{aligned} c &= \overline{s}a\overline{b} + \overline{s}ab + s\overline{a}b + sab \\ &= \overline{s}(a\overline{b} + ab) + s(\overline{a}b + ab) \\ &= \overline{s}(a(\overline{b} + b)) + s((\overline{a} + a)b) \\ &= \overline{s}(a(1) + s((1)b) \\ &\boxed{= \overline{s}a + sb} \end{aligned}$$

- **Circuit Diagram:**



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

- **Schematic:**

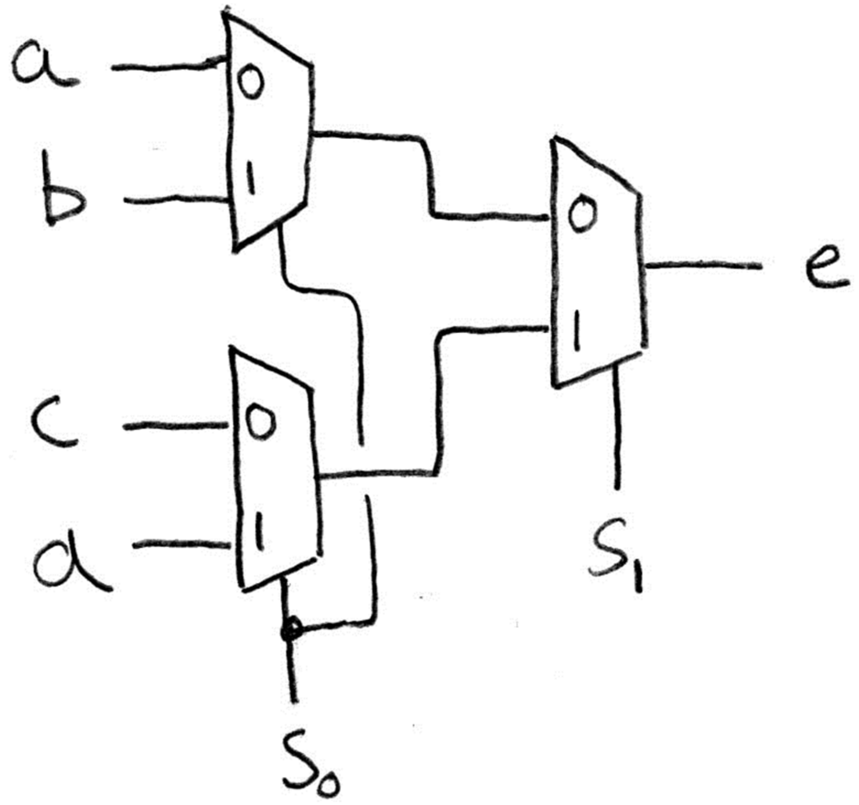


- **Truth Table:** How many rows? $2^6$
- **Boolean Expression:**

$$e = s_1's_0'a + s_1's_0b + s_1s_0'c + s_1s_0d$$

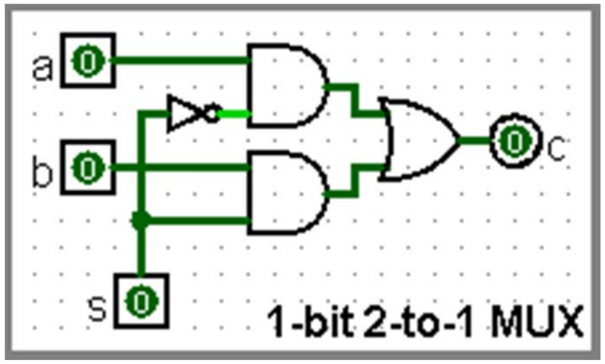
## 1-bit 4-to-1 MUX (2/2)

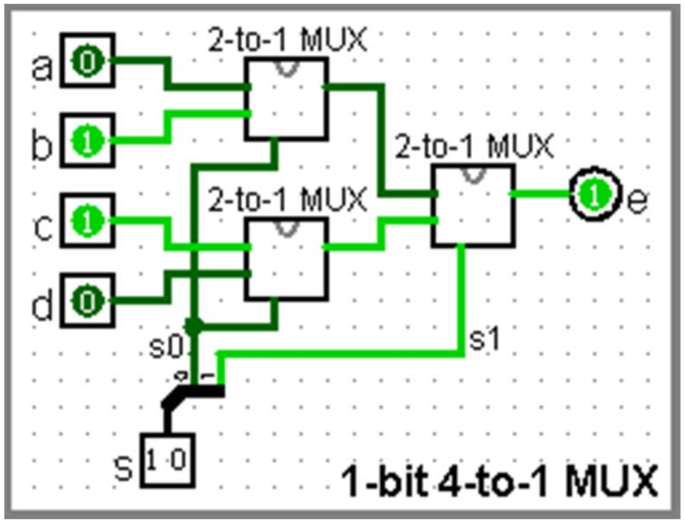
- Can we leverage what we've previously built?
  - Alternative hierarchical approach:

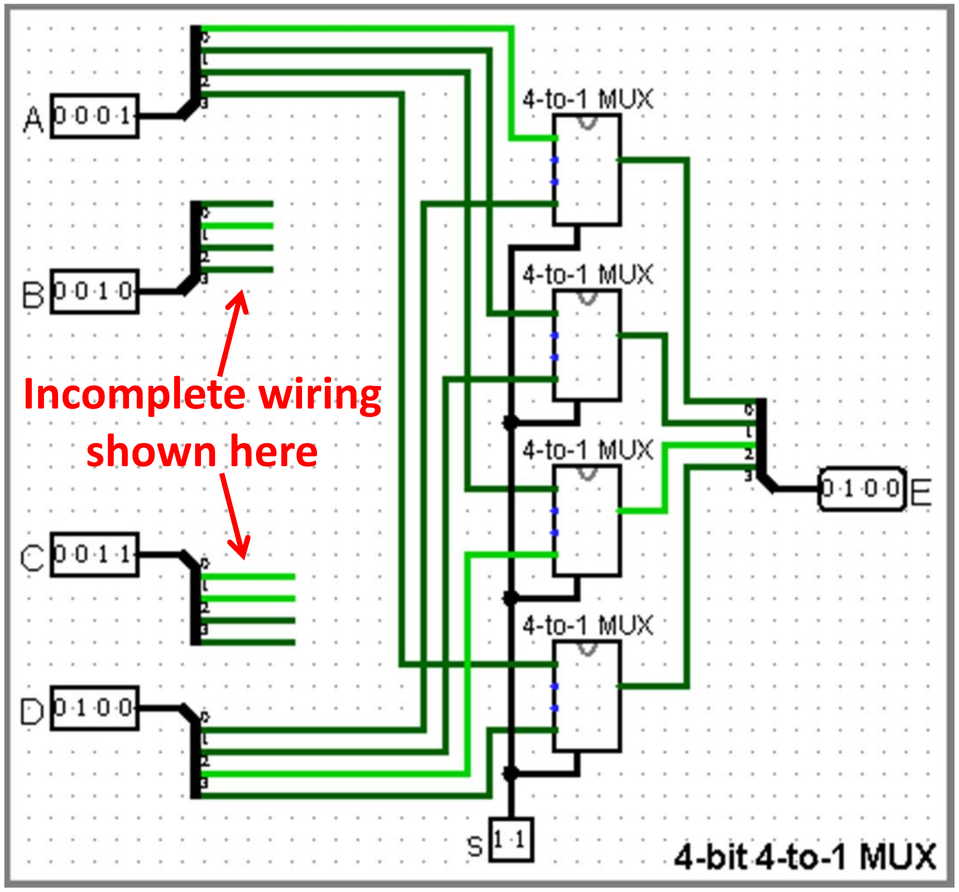


# Subcircuits Example

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







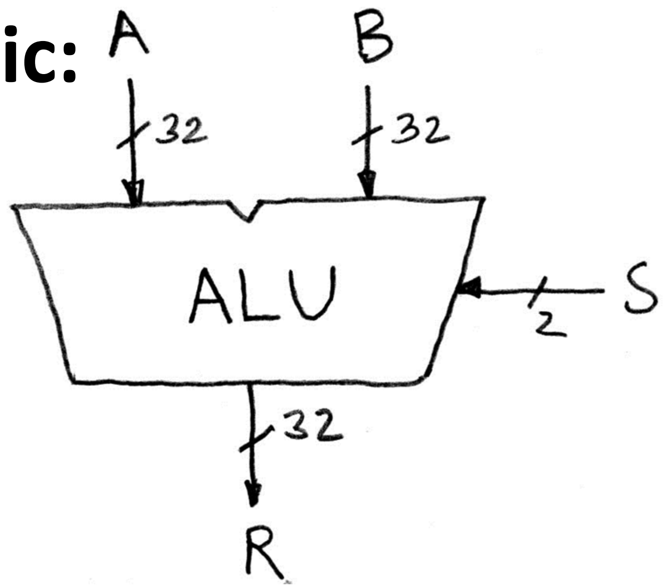
## Get To Know Your Instructor

## Agenda

- State Elements Continued
- Administrivia
- Logisim Introduction
- Finite State Machines
- Multiplexers
- ALU Design
  - Adder/Subtracter

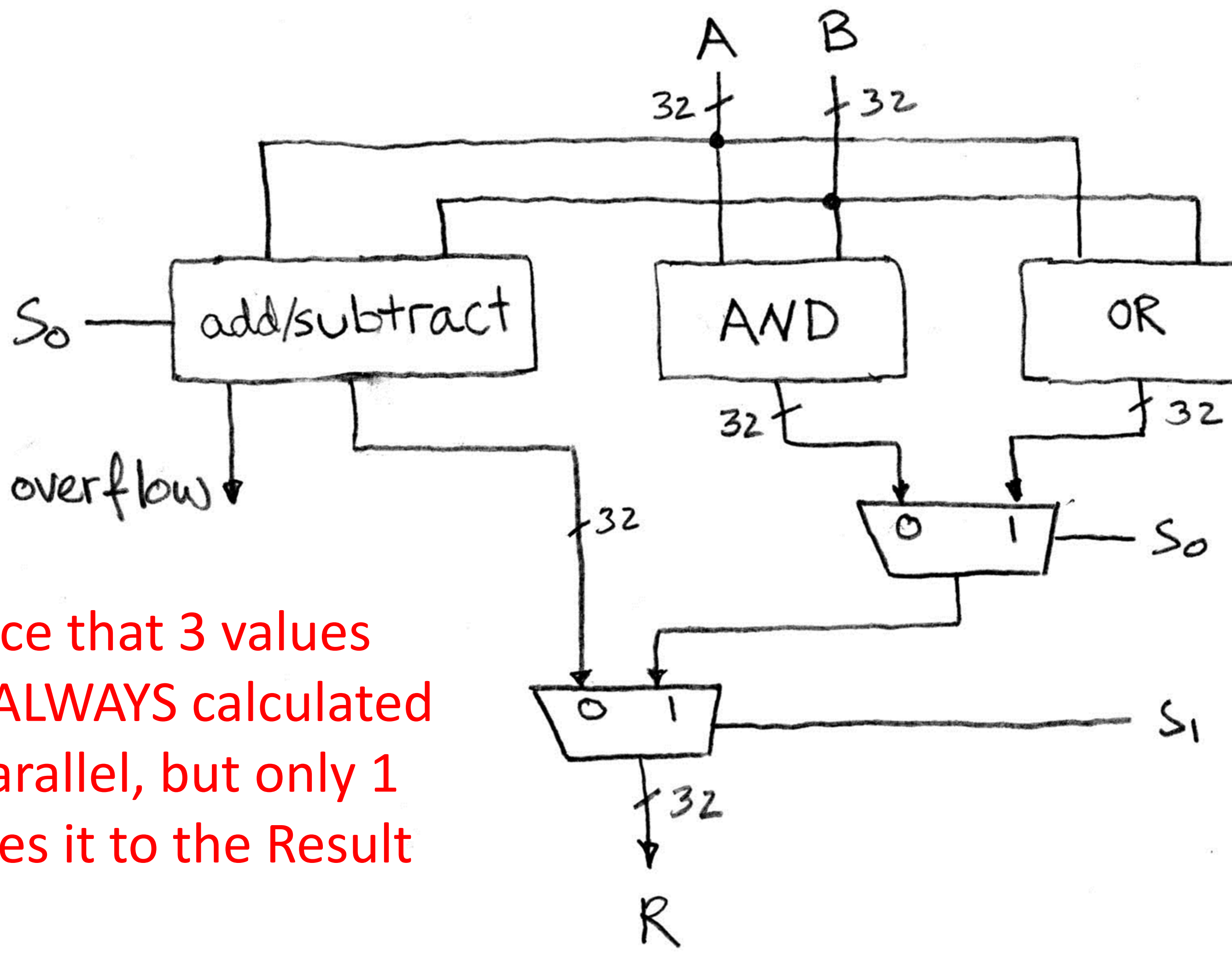
## Arithmetic and Logic Unit (ALU)

- 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 AND B
when S=11, R = A OR B

# Simple ALU Schematic



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

# Adder/Subtractor: 1-bit LSB Adder

$$\begin{array}{cccc|c|} \cline{5-5} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \\ \cline{5-5} \end{array}$$

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

Possible carry-in $c_1$

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

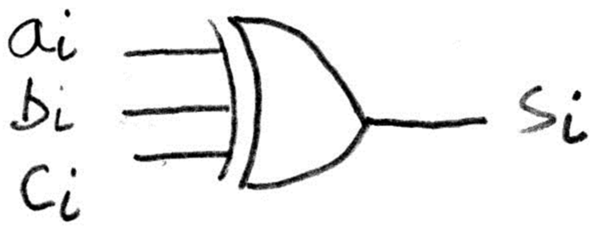
| $a_i$ | $b_i$ | $c_i$ | $s_i$ | $c_{i+1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

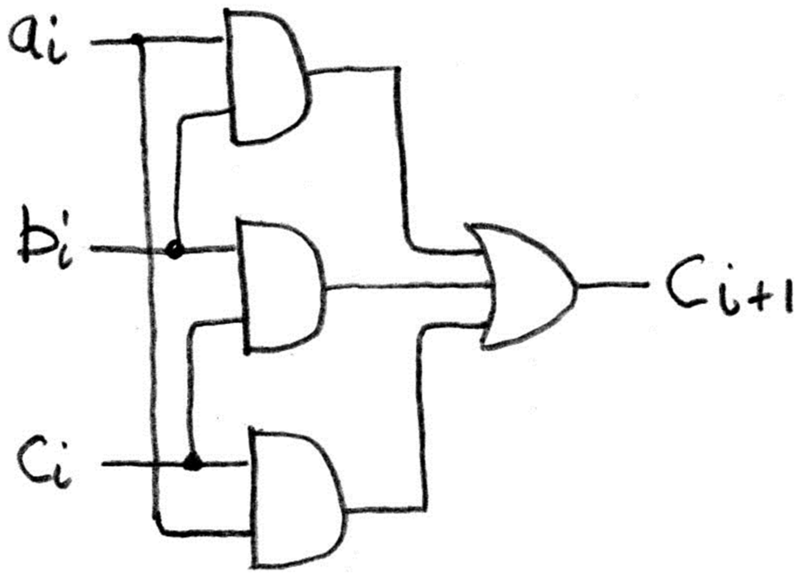
Here defining XOR of many inputs to be 1 when an *odd* number of inputs are 1

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

# Adder/Subtractor: 1-bit Adder

- **Circuit Diagrams:**

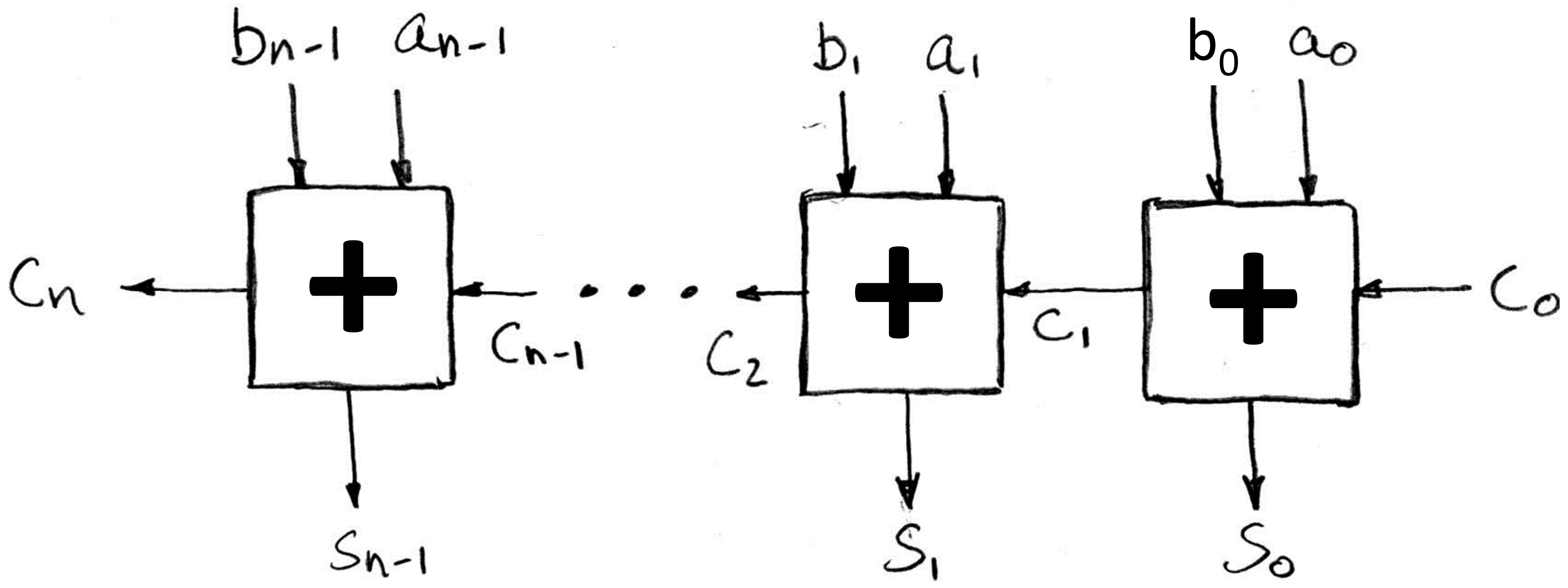




$$\begin{aligned} s_i &= \mathrm{XOR}(a_i, b_i, c_i) \\ c_{i+1} &= \mathrm{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i \end{aligned}$$

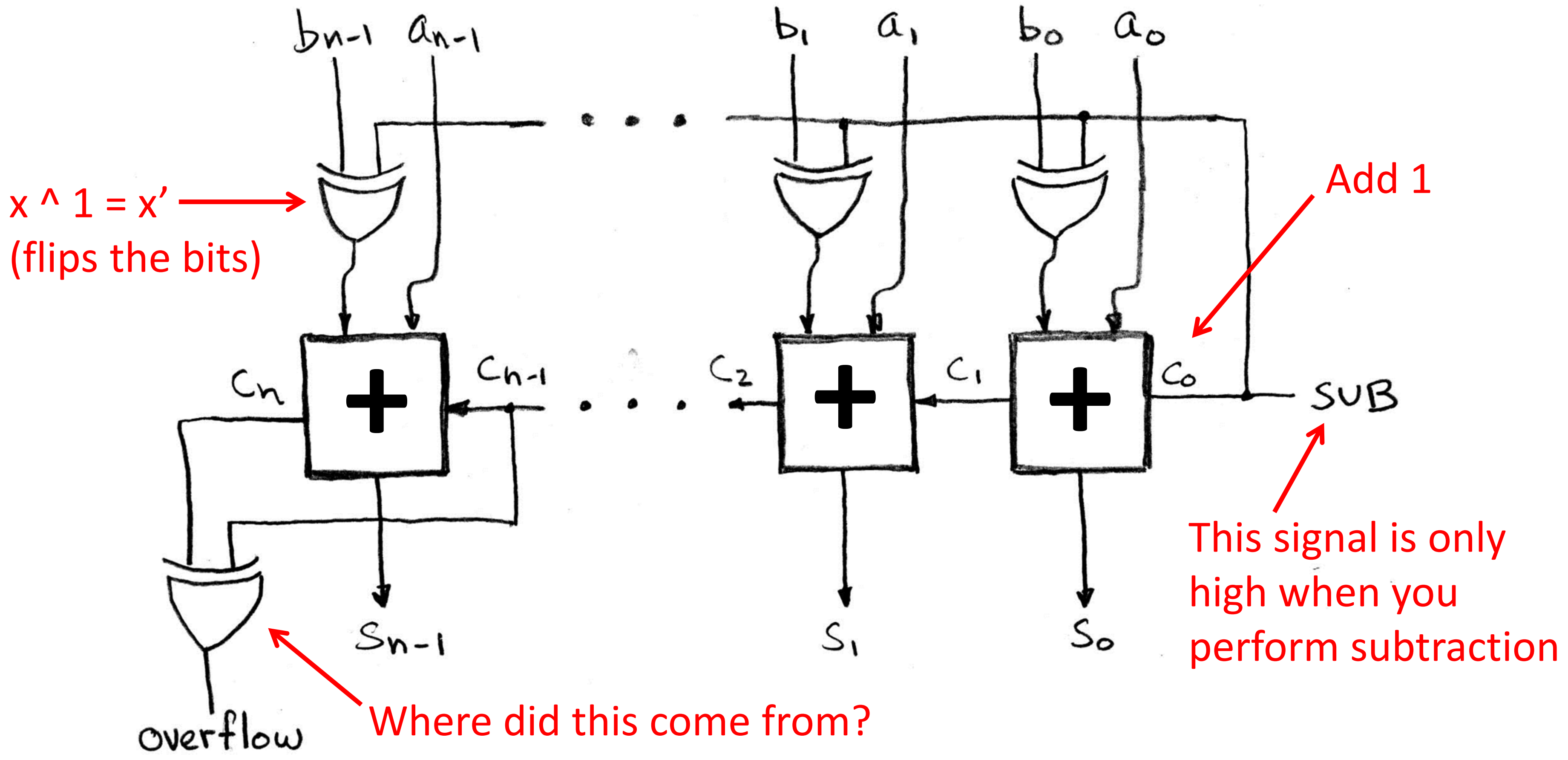
## N x 1-bit Adders → N-bit Adder

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

## Summary

- Critical path constrains clock rate
  - Timing constants: setup, hold, and clk-to-q times
  - Can adjust with extra registers (*pipelining*)
- Finite State Machines extremely useful
  - 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

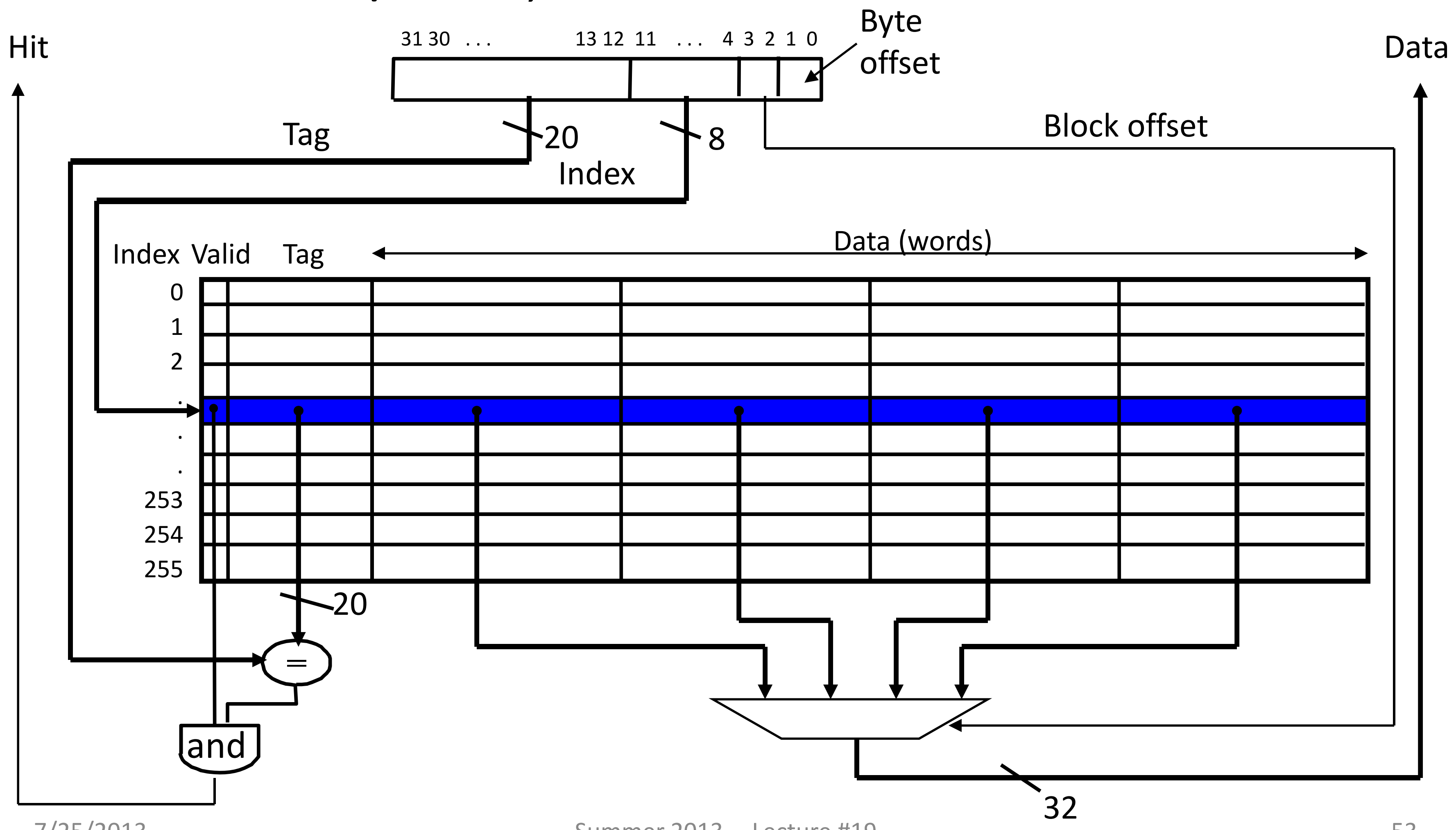
# BONUS SLIDES

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.

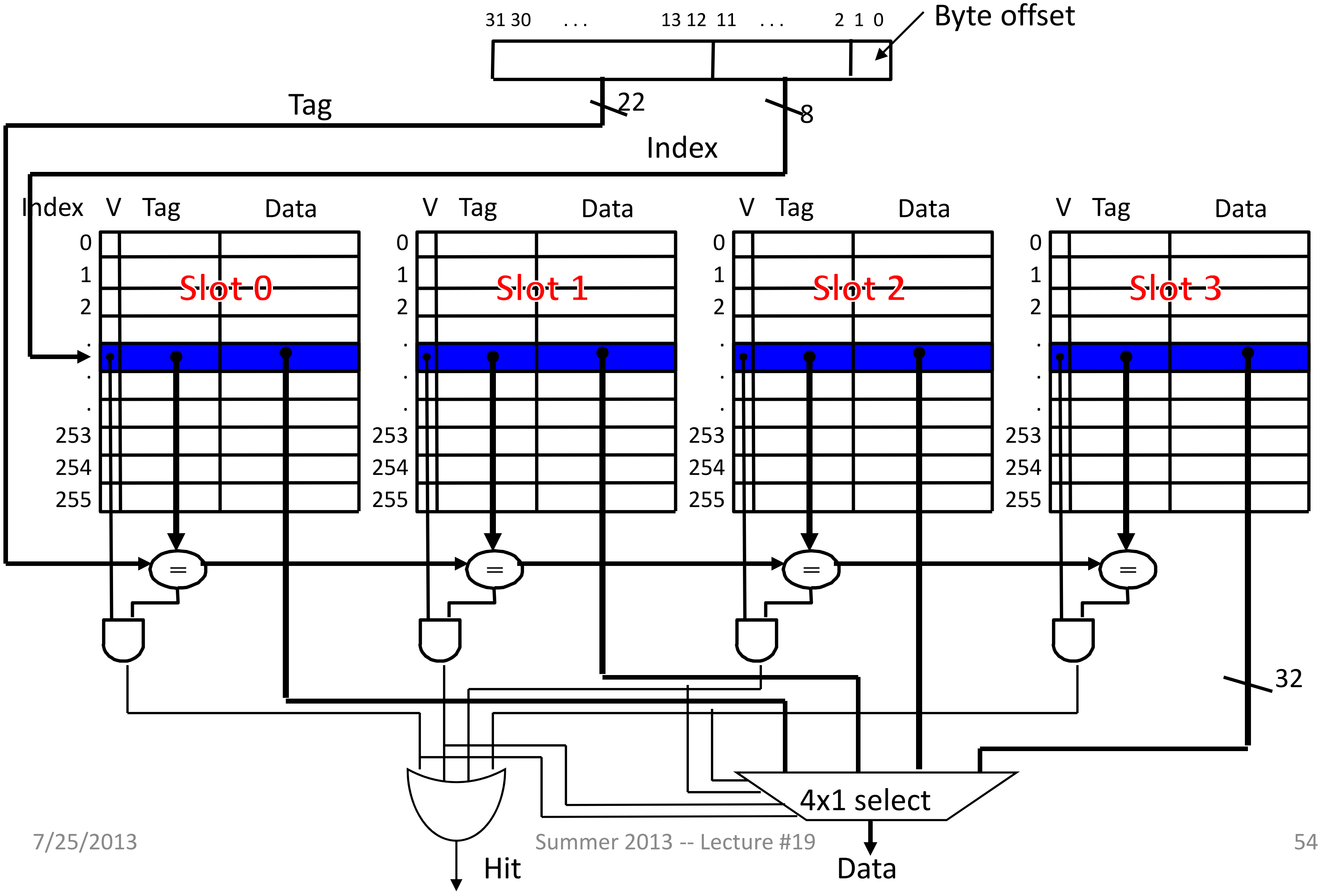
# Direct-Mapped Cache Internals

- Four words/block, cache size = 1 Ki words



# 4-Way Set Associative Cache

- $2^8$ = 256 sets each with four slots for blocks



31 30 . . . 13 12 11 . . . 2 1 0
Byte offset
Tag
22
8
Index
Index
V
Tag
Data
0
1
2
253
254
255
Slot 0
Slot 1
Slot 2
Slot 3
=
32
4x1 select
Hit
Data