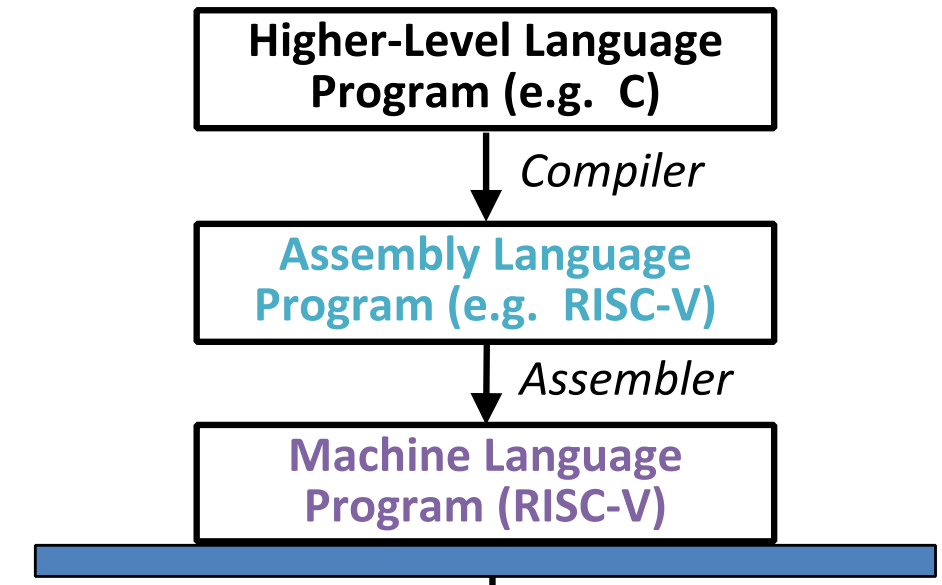
Computer Architecture, Fall 2019

# *Combinational Digital Logic*

# Agenda

- **Hardware Design Overview**
- Switches and Transistors
- Combinational Logic
  - Combinational Logic Gates
  - Truth Tables
  - Boolean Algebra
  - Circuit Simplification

# Overview




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

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

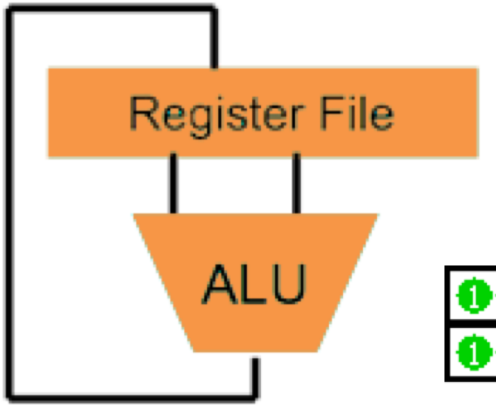
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

*Machine Interpretation*

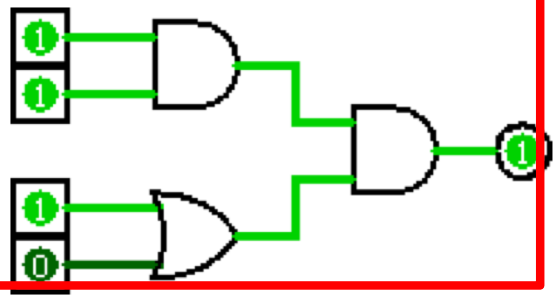
**Hardware Architecture Description (e.g. block diagrams)**

*Architecture Implementation*

**Logic Circuit Description (Circuit Schematic Diagrams)**




We are here



# Why is this important?

- Why study hardware design?
  - We need some digital systems knowledge to build our own processor

# Why is this important?

- We need some digital systems knowledge to build our own processor

- Why build our own processor?
  - Because we believe computer scientists should have an answer to the question “How does a computer work?”
  - So you can understand how code is actually executed on a computer

# Why is this important?

- So you can understand how code is actually executed on a computer

- Why do we care about how code executes?
  - Reliability
  - Performance
  - Security

# Hardware Design

Why study hardware design and processors and computers if you only care about high-level software?

Example of the power of layered abstractions

- Transistors -> Combinational Logic
- Combinational Logic -> Sequential Logic
- Sequential Logic -> Processors
- Processors -> Machine Language
- Machine Language -> Assembly
- Assembly -> High-level Programming Languages
- High-level Programming Languages -> Word,Minecraft,Twitter

At each step we can “abstract away” the lower steps and (mostly) forget they exist

# Synchronous Digital Systems (SDS)

*Hardware of a processor, such as a RISC-V processor, is an example of a Synchronous Digital System*

## *Synchronous:*

- All operations coordinated by a central clock
  - "Heartbeat" of the system! (processor frequency)

## *Digital:*

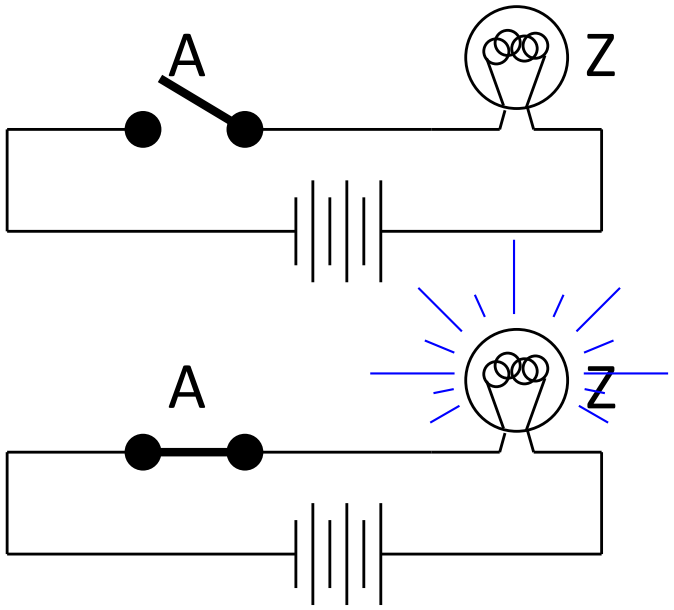
- Represent all values with two discrete values
- Electrical signals are treated as 1's and 0's
  - 1 and 0 are complements of each other
- High/Low voltage for True/False, 1/0

# Agenda

- Hardware Design Overview
- Combinational Logic
  - Combinational Logic Gates
  - Truth Tables
  - Boolean Algebra
  - Circuit Simplification

# Switches

- The basic element of physical implementations
- Convention:  if input is a “1,” the switch is *asserted*




*Open* switch if A is “0” (unasserted) and turn OFF light bulb (Z)

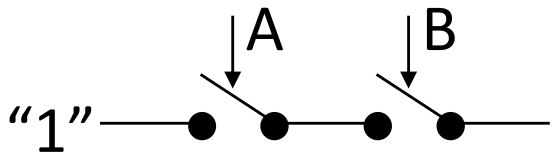
*Close* switch if A is “1” (asserted) and turn ON light bulb (Z)

In this example, Z ≡ A.

# Switch Logic

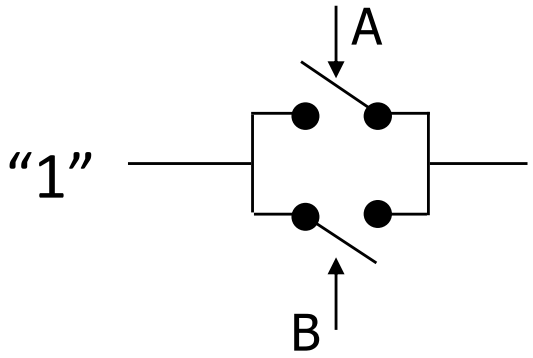
- Can compose switches into more complex ones (Boolean functions)
  - Arrows show action upon assertion (1 = close)

**AND:**




Z ≡ A and B

**OR:**




Z ≡ A or B

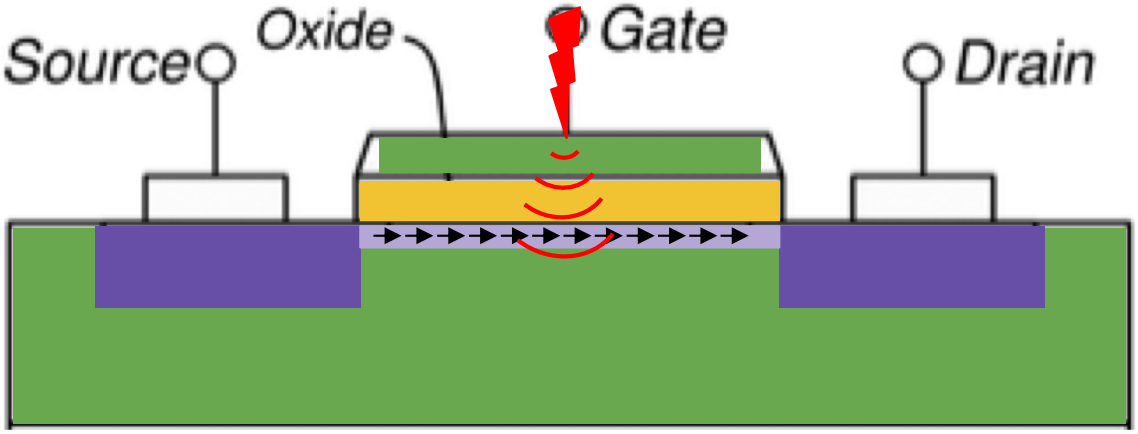
# Computers need switches

To create an electric computer:

- We need the ability to control switches with electricity
- Switches controlling groups of other switches
  - So electricity should flow through the switch

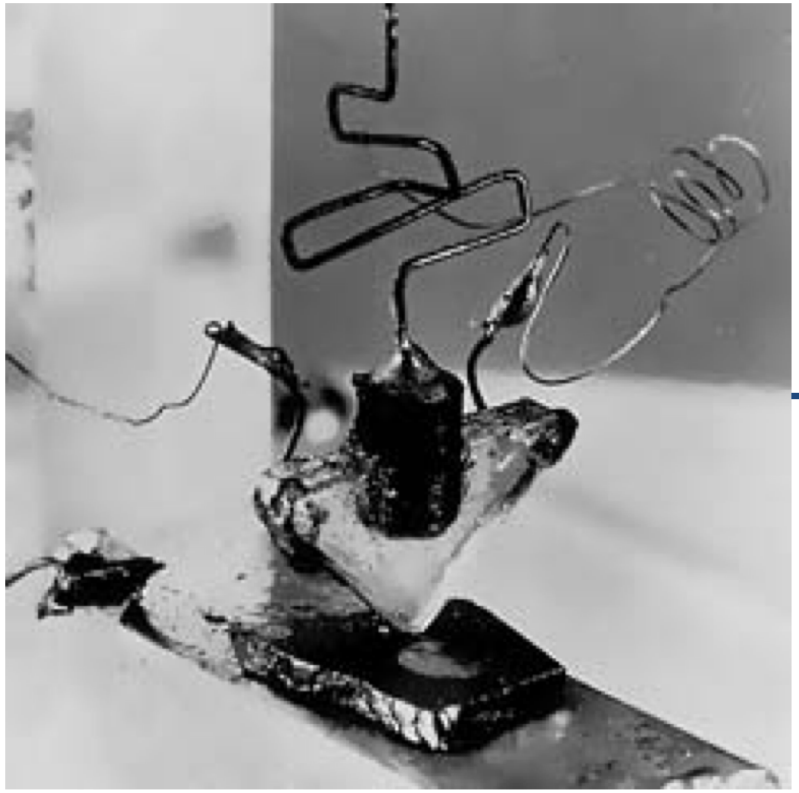
# Transistors are hardware switches!

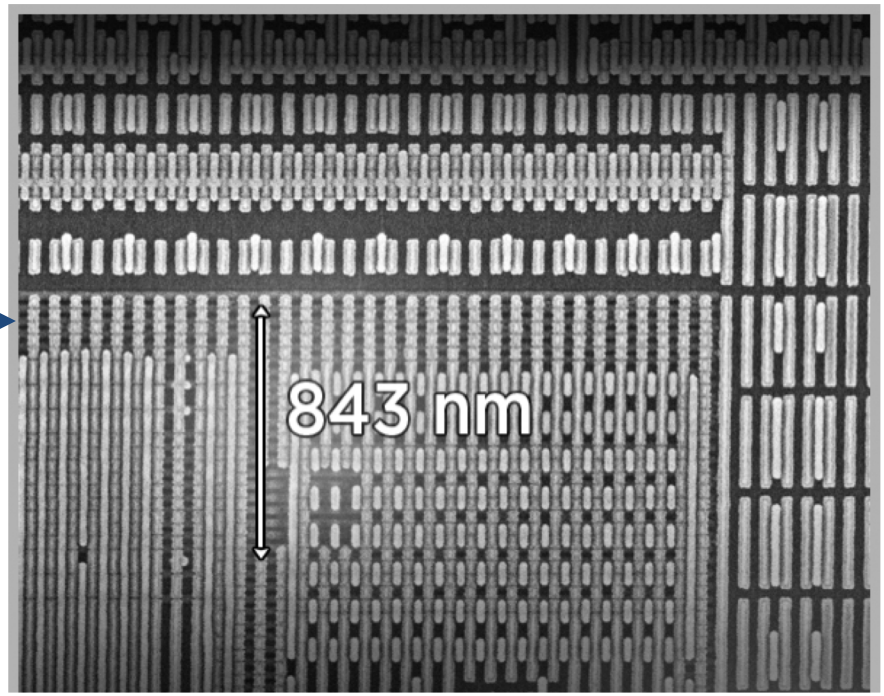
- **MOSFET**:
  - **M**etal **O**xide **S**emiconductor **F**ield **E**ffect **T**ransistor



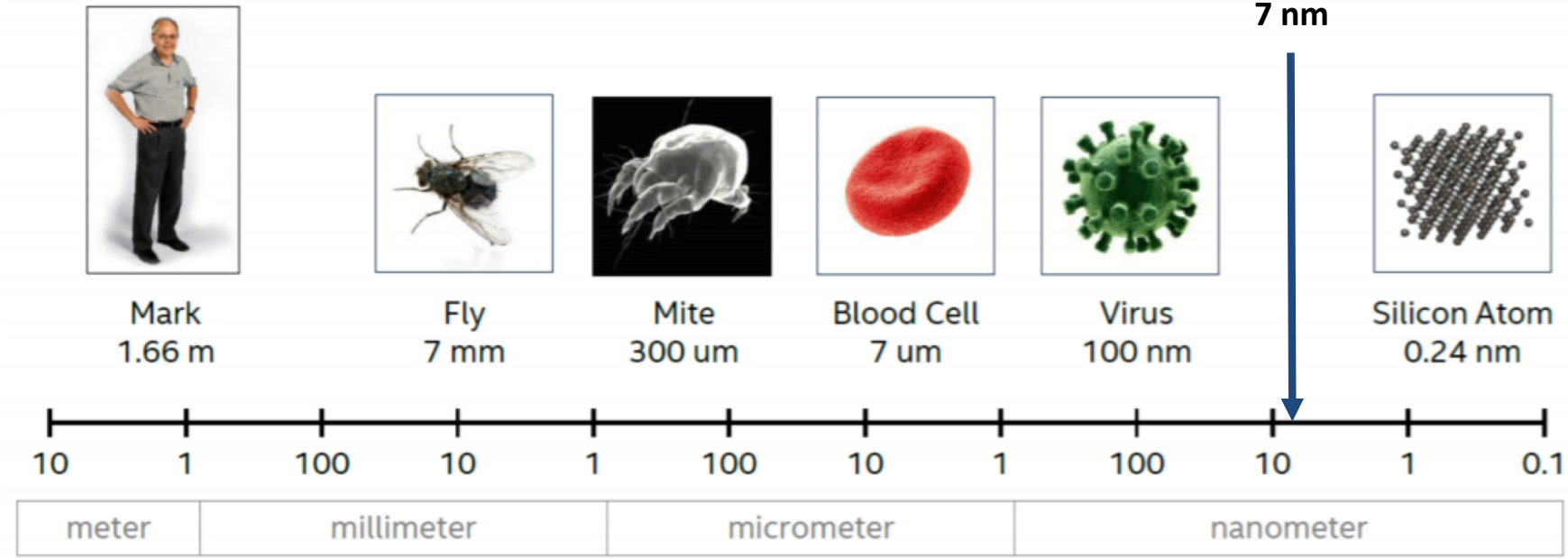

**The bottom of the abstraction layers for this class!**

# 1947 → Today





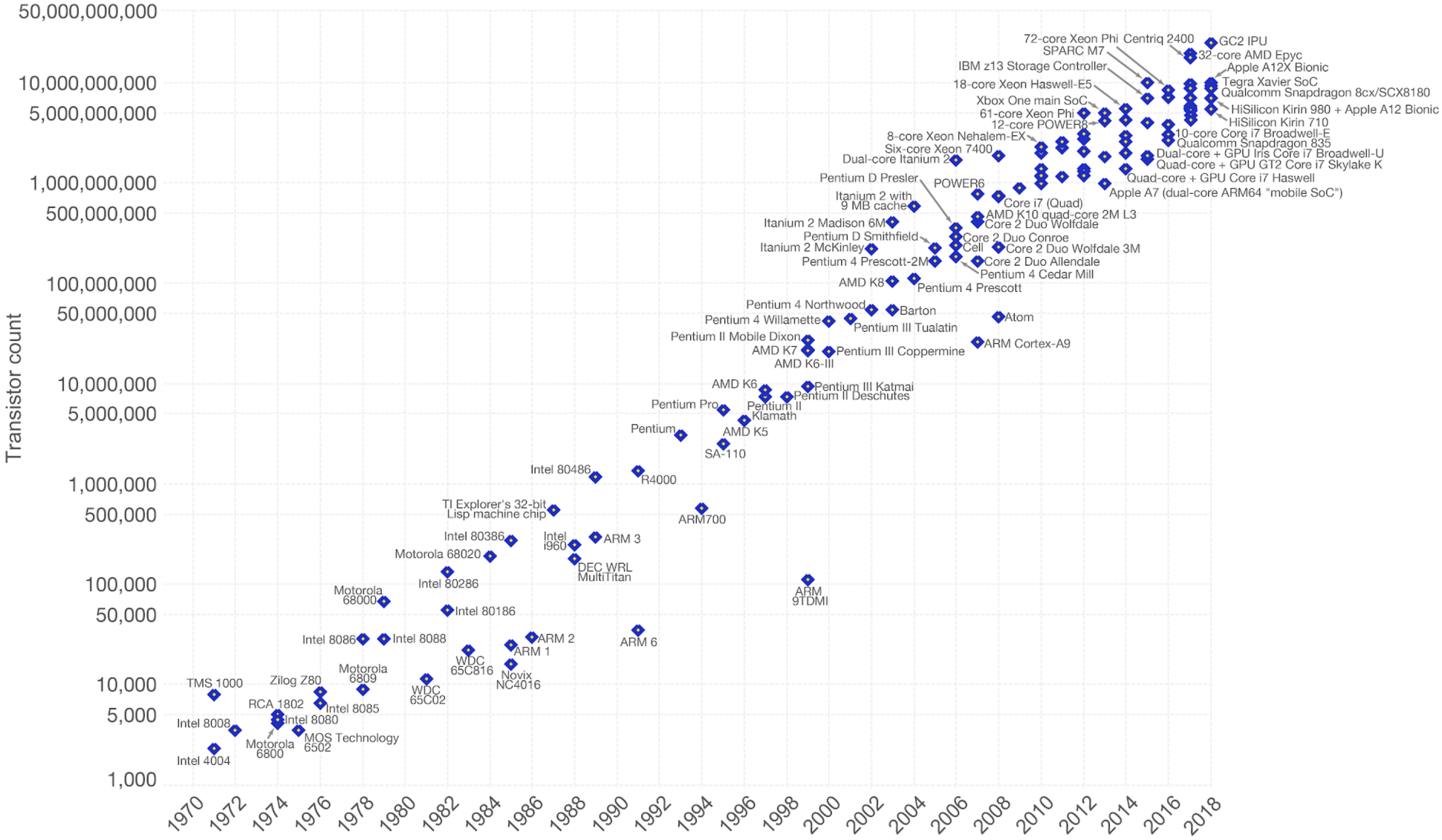

**2019 Processor Technology**
**7 nm**




Source: Mark Bohr, IDF14

# Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

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. There you find more visualizations and research on this topic.

# Historical Trend

Trend:

- Hardware gets more powerful every year.
  - (Really due to hard work of thousands of engineers)

Therefore:

- Software gets more resources and faster compute!
  - (And has to keep up with ever-changing hardware)

# Agenda

- Hardware Design Overview
- Switches and Transistors
- **Combinational Logic**
  - **Combinational Logic Gates**
  - Truth Tables
  - Boolean Algebra
  - Circuit Simplification

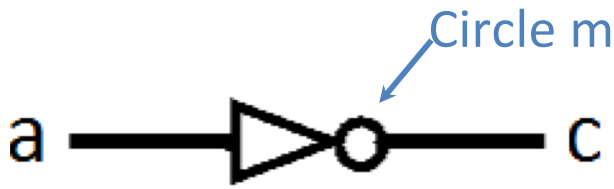
# Type of Circuits

- *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)

# Logic Gates (1/2)

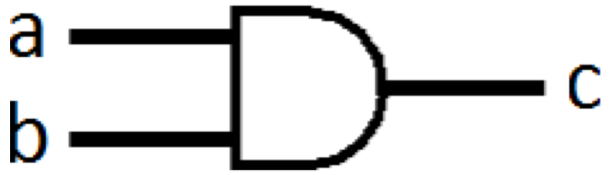
- Special names and symbols:

**NOT**




| a | c |
|---|---|
| 0 | 1 |
| 1 | 0 |

**AND**




| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

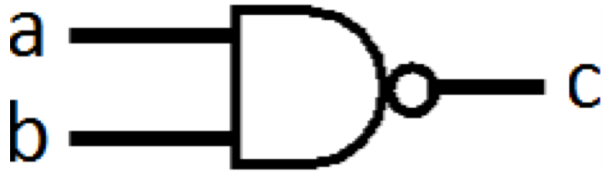
**OR**


| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# Logic Gates (2/2)

Inverted versions are easier to implement in CMOS

**NAND**




| a | b | c |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

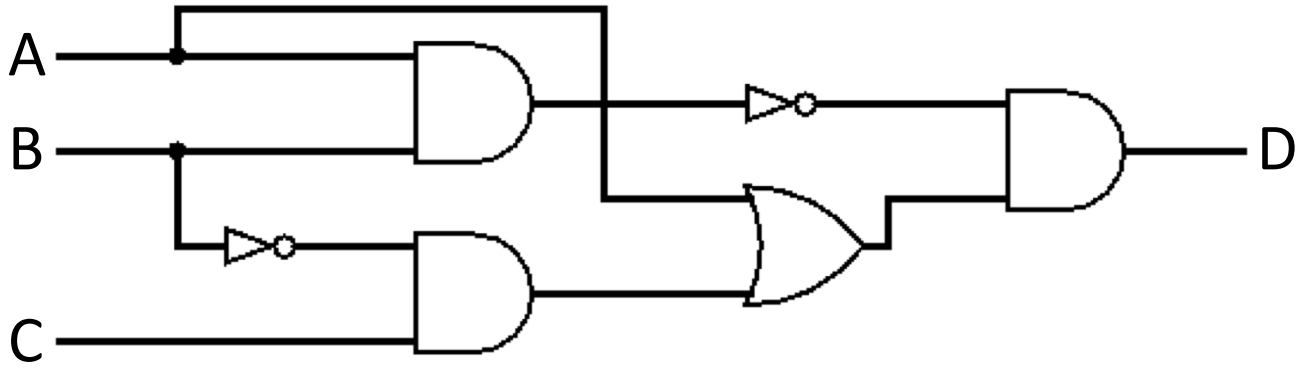
**NOR**


| a | b | c |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**XOR**


| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Combining Multiple Logic Gates




(NOT(**A** AND **B**)) AND (**A** OR (NOT **B** AND **C**))

# Agenda

- Hardware Design Overview
- Switches and Transistors
- **Combinational Logic**
  - Combinational Logic Gates
  - **Truth Tables**
  - Boolean Algebra
  - Circuit Simplification

# Representations of Combinational Logic

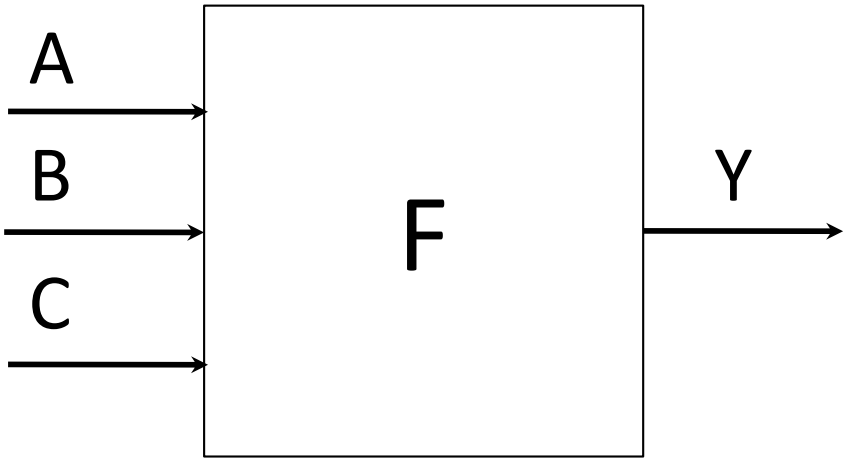
- ✓ Text Description
- ✓ Circuit Diagram
  - – Transistors and wires
  - – Logic Gates
- ✓ Truth Table
- ✓ Boolean Expression

✓ *All are equivalent*

# Truth Tables

- Table that relates the inputs to a combinational logic circuit to its output
  - Output *only* depends on current inputs
  - Use abstraction of 0/1 instead of high/low V
  - Shows output for *every* possible combination of inputs
- How big?
  - 0 or 1 for each of N inputs, so $2^N$ rows

# General Form



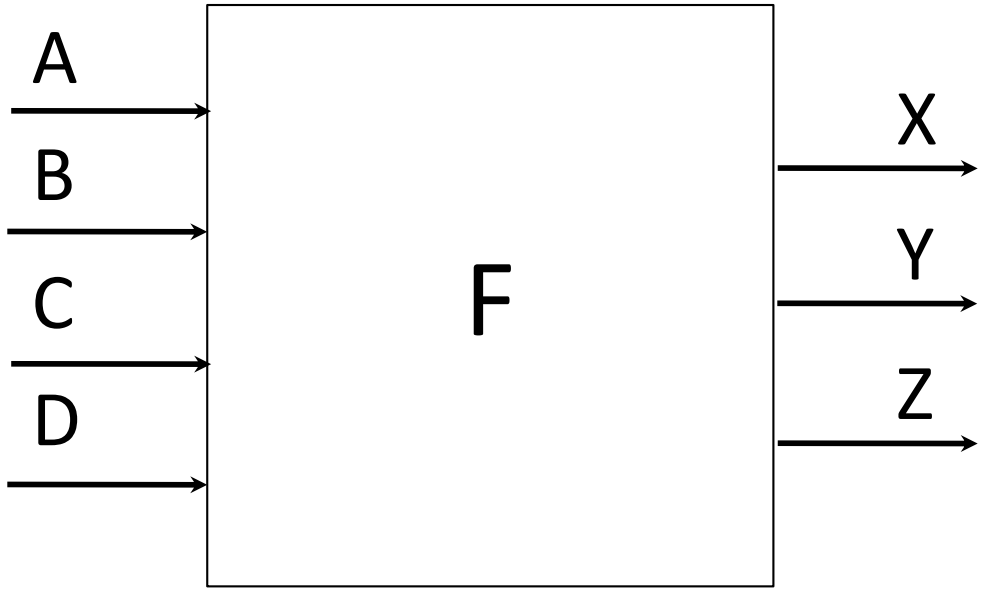

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | F(0,0,0) |
| 0 | 0 | 1 | F(0,0,1) |
| 0 | 1 | 0 | F(0,1,0) |
| 0 | 1 | 1 | F(0,1,1) |
| 1 | 0 | 0 | F(1,0,0) |
| 1 | 0 | 1 | F(1,0,1) |
| 1 | 1 | 0 | F(1,1,0) |
| 1 | 1 | 1 | F(1,1,1) |

**R Rows**

If N inputs, how many distinct functions F do we have?

Function maps each row to 0 or 1, so $2^R$ possible functions

# Multiple Outputs




- For 3 outputs, just three indep. functions: X(A,B,C,D), Y(A,B,C,D), and Z(A,B,C,D)
  - Can show functions in separate columns without adding any rows!

**Question:** Convert the following statements into a Truth Table for: (x XOR y) OR (NOT z)

| X | Y | Z | (A) | (B) | (C) | (D) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |

**Question:** Convert the following statements into a Truth Table for: (x XOR y) OR (NOT z)

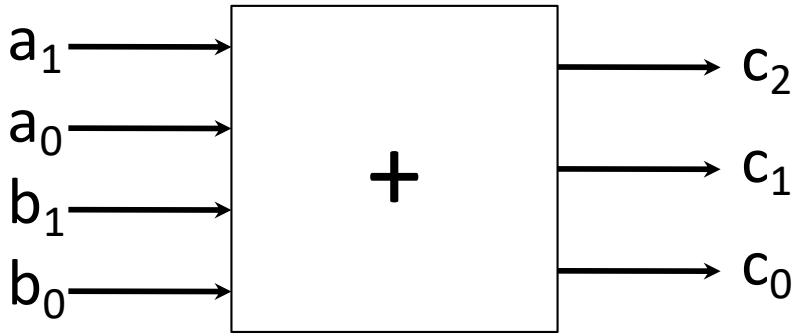
| X | Y | Z | (A) | (B) | (C) | (D) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |

# More Complicated Truth Tables

**3-Input Majority**

| a | b | c | y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**2-bit Adder**




| A | | B | | C | | |
|---|---|---|---|---|---|---|
| $a_1$ | $a_0$ | $b_1$ | $b_0$ | $c_2$ | $c_1$ | $c_0$ |
| | | | . | How many rows? | | |
| | | | . | | | |
| | | | . | | | |

# Agenda

- Detailed CALL Example
- Hardware Design Overview
- Switches and Transistors
- CMOS Networks
- **Combinational Logic**
  - Combinational Logic Gates
  - Truth Tables
  - **Boolean Algebra**
  - Circuit Simplification

# My Hand Hurts…

- Truth tables are huge
  - Write out EVERY combination of inputs and outputs (thorough, but inefficient)
  - Finding a particular combination of inputs involves scanning a large portion of the table
- There must be a shorter way to represent combinational logic
  - Boolean Algebra to the rescue!

# Boolean Algebra

- Represent inputs and outputs as variables
  - Each variable can only take on the value 0 or 1
- Overbar or ¬ is NOT: "logical complement"
  - e.g. if A is 0, $\overline{A}$ is 1. If A is 1, then ¬A is 0

For slides, will use ¬A

- Plus (+) is 2-input OR: "logical sum"
- Product (·) is 2-input AND: "logical product"
  - All other gates and logical expressions can be built from combinations of these

    **¬AB** + **A¬B** == (NOT(**A** AND **B**)) OR (**A** AND NOT **B**)

# Laws of Boolean Algebra

These laws allow us to perform simplification:

| | | |
|---|---|---|
| $x \cdot \overline{x} = 0$ | $x + \overline{x} = 1$ | complementarity |
| $x \cdot 0 = 0$ | $x + 1 = 1$ | laws of 0's and 1's |
| $x \cdot 1 = x$ | $x + 0 = x$ | identities |
| $x \cdot x = x$ | $x + x = x$ | idempotent law |
| $x \cdot y = y \cdot x$ | $x + y = y + x$ | commutativity |
| $(xy)z = x(yz)$ | $(x + y) + z = x + (y + z)$ | associativity |
| $x(y + z) = xy + xz$ | $x + yz = (x + y)(x + z)$ | distribution |
| $xy + x = x$ | $(x + y)x = x$ | uniting theorem |
| $\overline{x}y + x = x + y$ | $(\overline{x} + y)x = xy$ | uniting theorem v.2 |
| $\overline{x \cdot y} = \overline{x} + \overline{y}$ | $\overline{x + y} = \overline{x} \cdot \overline{y}$ | DeMorgan's Law |

# Truth Table to Boolean Expression

- Read off of table
  - For 1, write variable name
  - For 0, write complement of variable
- *Sum of Products (SoP)*
  - Take rows with 1's in output column, sum products of inputs
  - c= ¬a b + a¬b
- *Product of Sums (PoS)*
  - Take rows with 0's in output column, product the sum of the *complements of the inputs*
  - c = (a + b) · (¬a + ¬b)

| a | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

We can show that these are equivalent!

# Agenda

- Detailed CALL Example
- Hardware Design Overview
- Switches and Transistors
- CMOS Networks
- **Combinational Logic**
  - Combinational Logic Gates
  - Truth Tables
  - Boolean Algebra
  - **Circuit Simplification**

# Manipulating Boolean Algebra

- SoP and PoS expressions can still be long
  - We wanted to have shorter representation than a truth table!
- Boolean algebra follows a set of rules that allow for simplification
  - Goal will be to arrive at the simplest equivalent expression
  - Allows us to build simpler (and faster) hardware

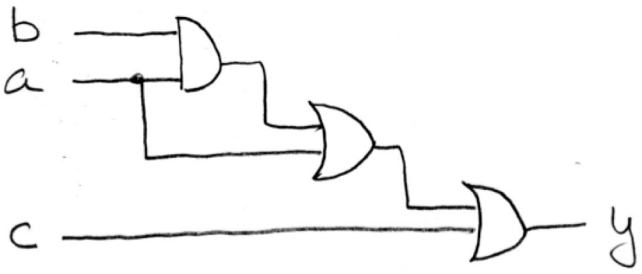
# Faster Hardware?

- **Recall:** Everything we are dealing with is just an abstraction of transistors and wires
  - Inputs propagating to the outputs are voltage signals passing through transistor networks
  - There is always some *delay* before a CL's output updates to reflect the inputs
- Simpler Boolean expressions ↔ smaller transistor networks ↔ smaller circuit delays ↔ faster hardware

# Boolean Algebraic Simplification Example

$$y = ab + a + c$$

# Circuit Simplification




$$\downarrow$$

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

$$\downarrow$$

$$= ab + a + c$$
$$= a(b + 1) + c$$
$$= a(1) + c$$
$$= a + c$$

$$\downarrow$$


1) original circuit (Transistors and/or Gates)

2) equation derived from original circuit

3) algebraic simplification

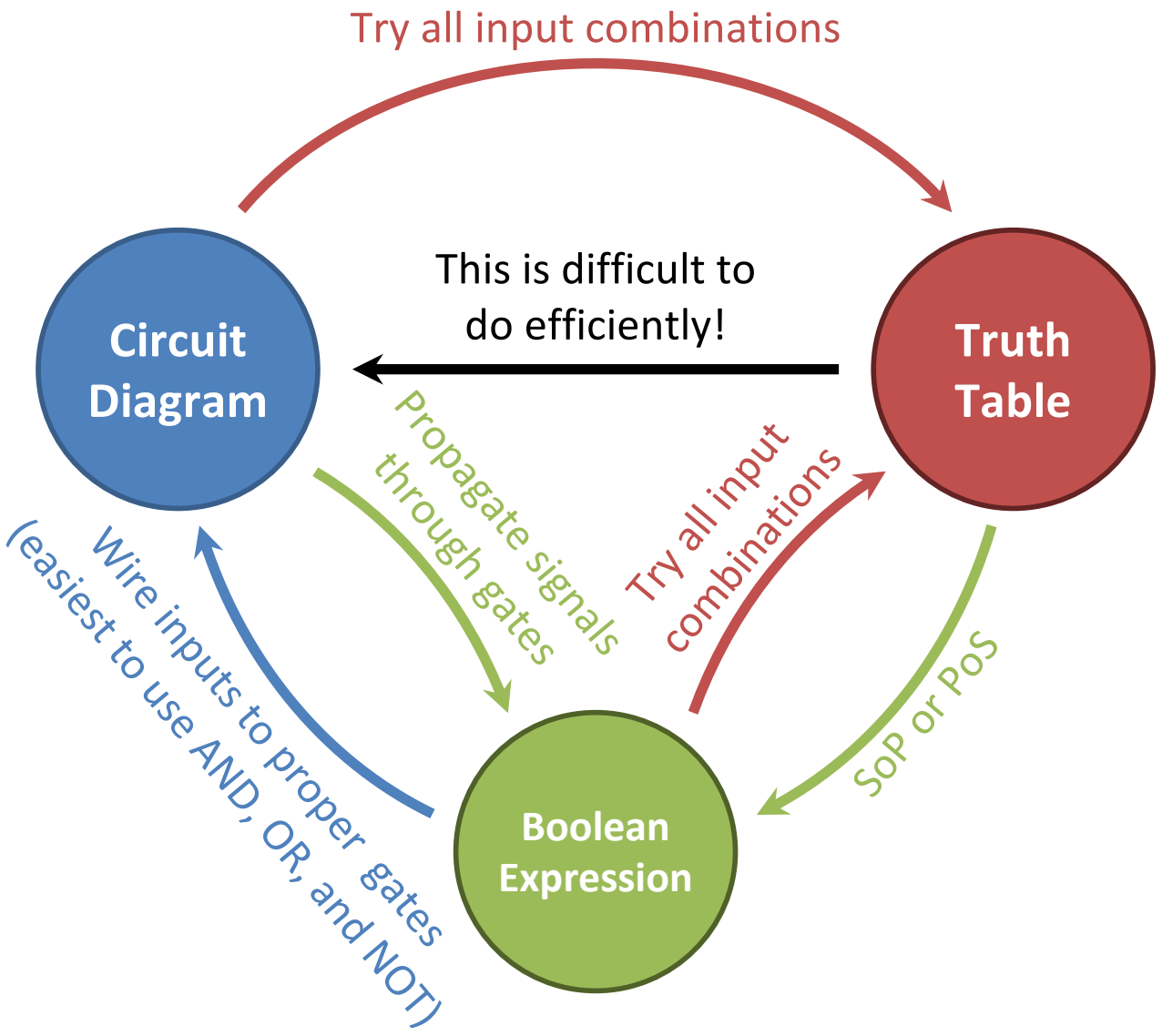
4) simplified circuit

# Representations of Combinational Logic

- ✓ Text Description
- ✓ Circuit Diagram
  - – Transistors and wires
  - – Logic Gates
- ✓ Truth Table
- ✓ Boolean Expression

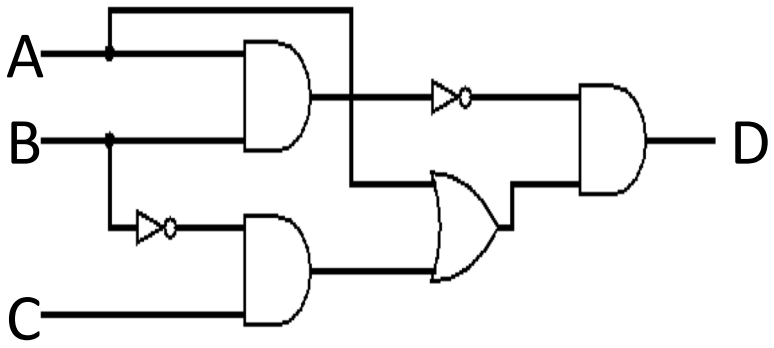
✓ *All are equivalent*

# Converting Combinational Logic



# Summary

- Beginnings of hardware design and layered abstractions
- Transistors -> CMOS Networks -> Combinational Logic




✓ Text Description
✓ Circuit Diagram
  – Transistors and wires
  – Logic Gates
✓ Truth Table
✓ Boolean Expression

✓ *All are equivalent*

# *Sequential Digital Logic*

# Great Idea #1: Levels of Representation & Interpretation

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

*Compiler*

**Assembly Language Program (e.g. RISCV)**

*Assembler*

**Machine Language Program (RISCV)**

```
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
```

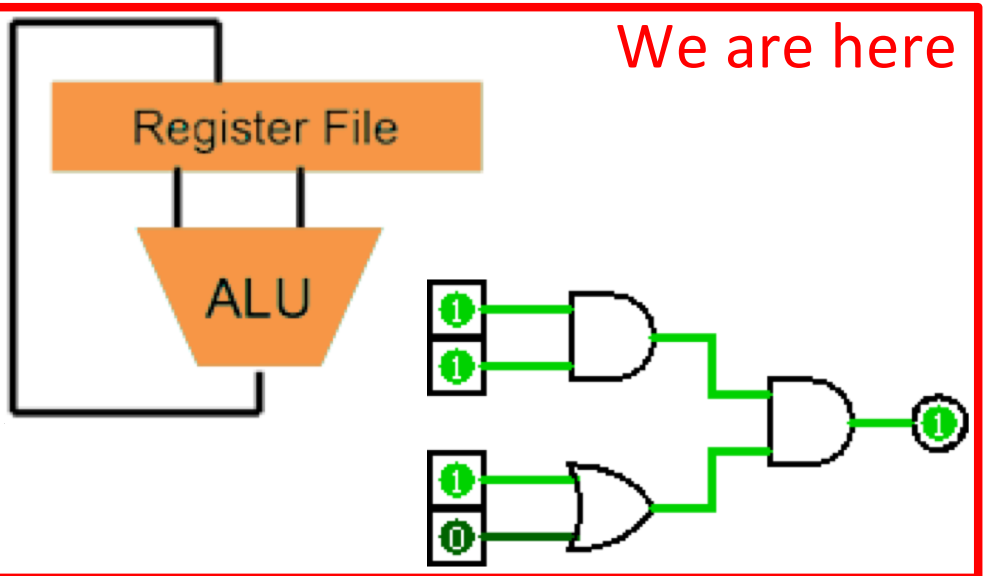
*Machine Interpretation*

**Hardware Architecture Description (e.g. block diagrams)**

*Architecture Implementation*

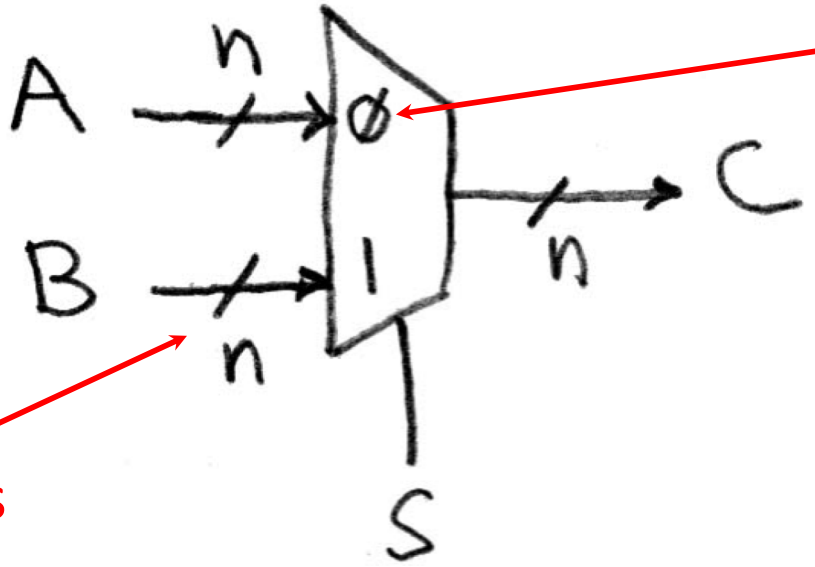
**Logic Circuit Description (Circuit Schematic Diagrams)**

We are here



# 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

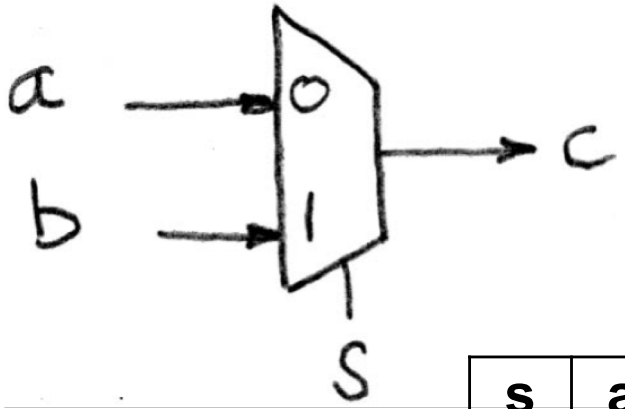



This input is passed to output if selector bits match shown value

Represents that this wire has n bits

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

- **Schematic:**



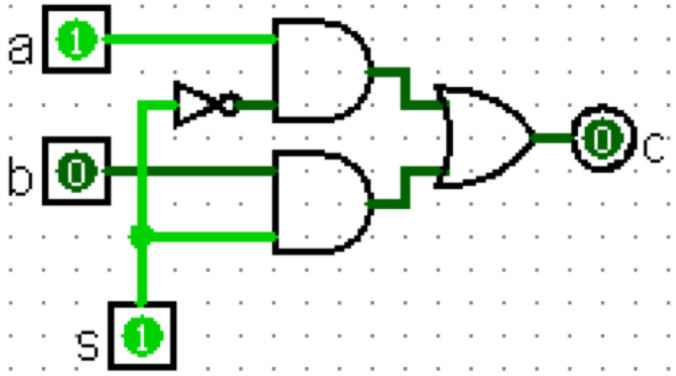

- **Truth Table:**

| 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:**

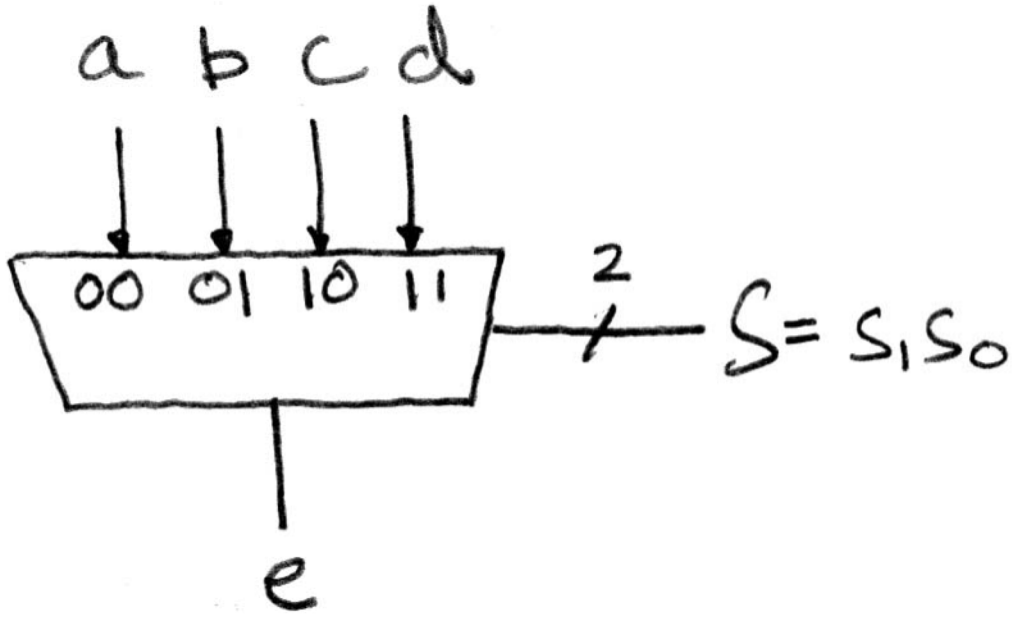
$$
\begin{aligned}
c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\
&= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\
&= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\
&= \bar{s}(a(1) + s((1)b) \\
&= \bar{s}a + sb
\end{aligned}
$$

- **Circuit Diagram:**



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

- **Schematic:**

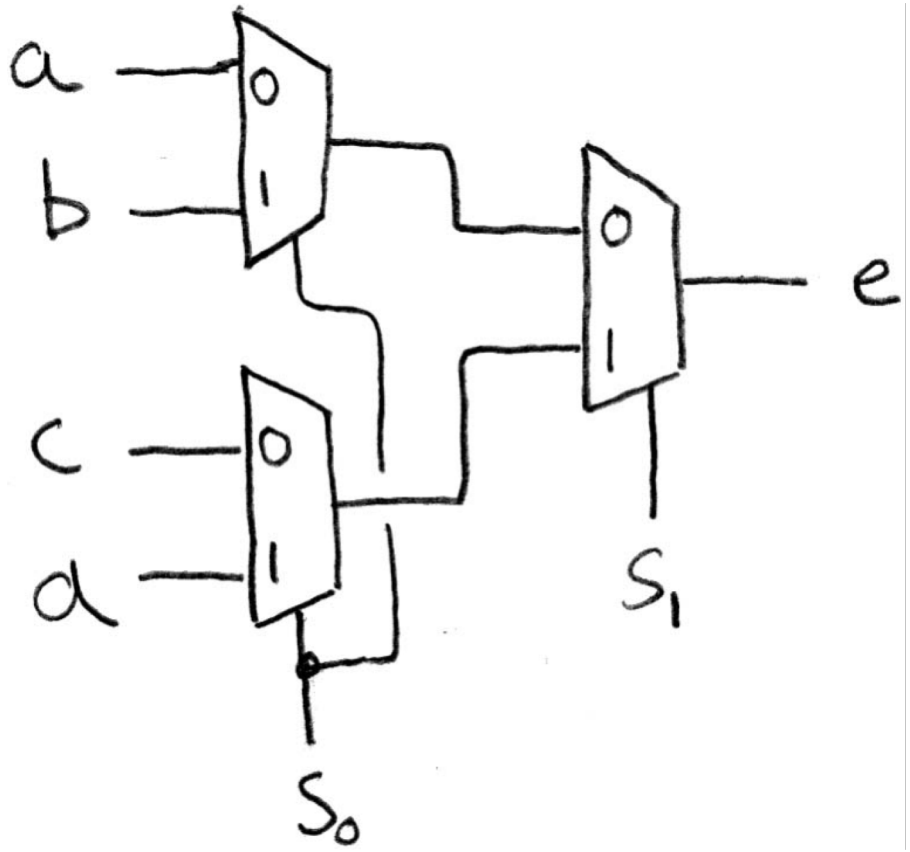



- **Truth Table:** How many rows? $2^6$
- **Boolean Expression:**

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

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

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



# Agenda

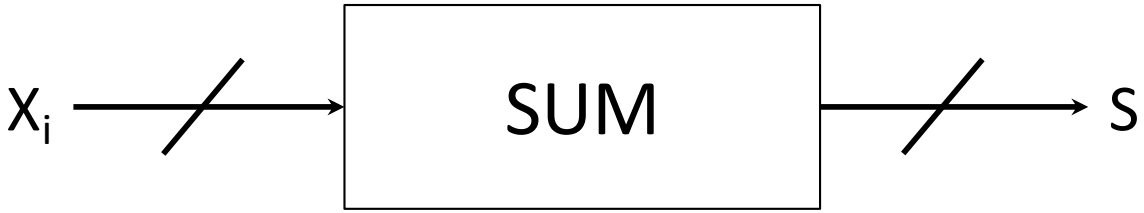
- Muxes
- **Sequential Logic Timing**
- Maximum Clock Frequency
- Functional Units
- Summary

# 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)

# Accumulator Example

An example of why we would need sequential logic



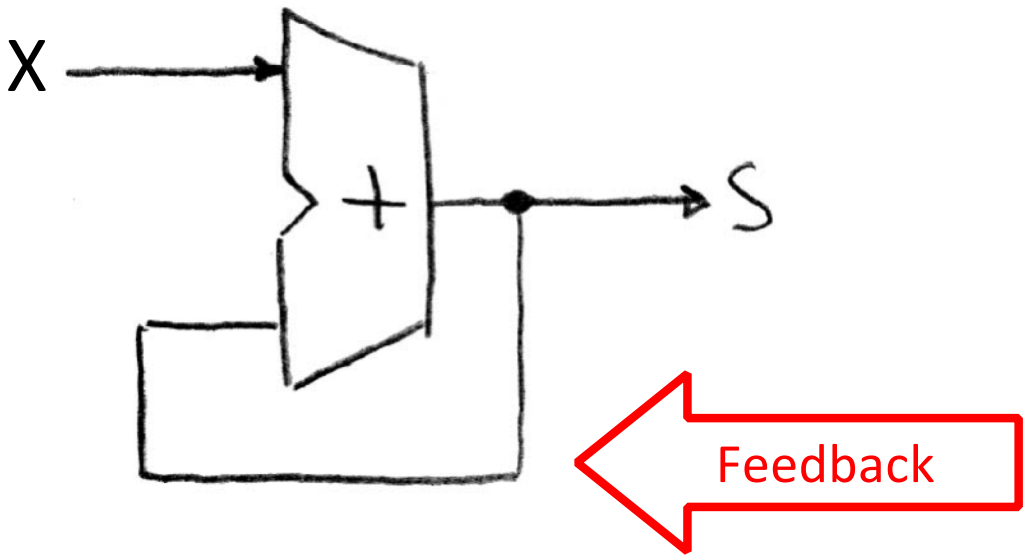

Want:

```
S=0;
for X1,X2,X3 over time...
        S = S + Xi
```

Assume:

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

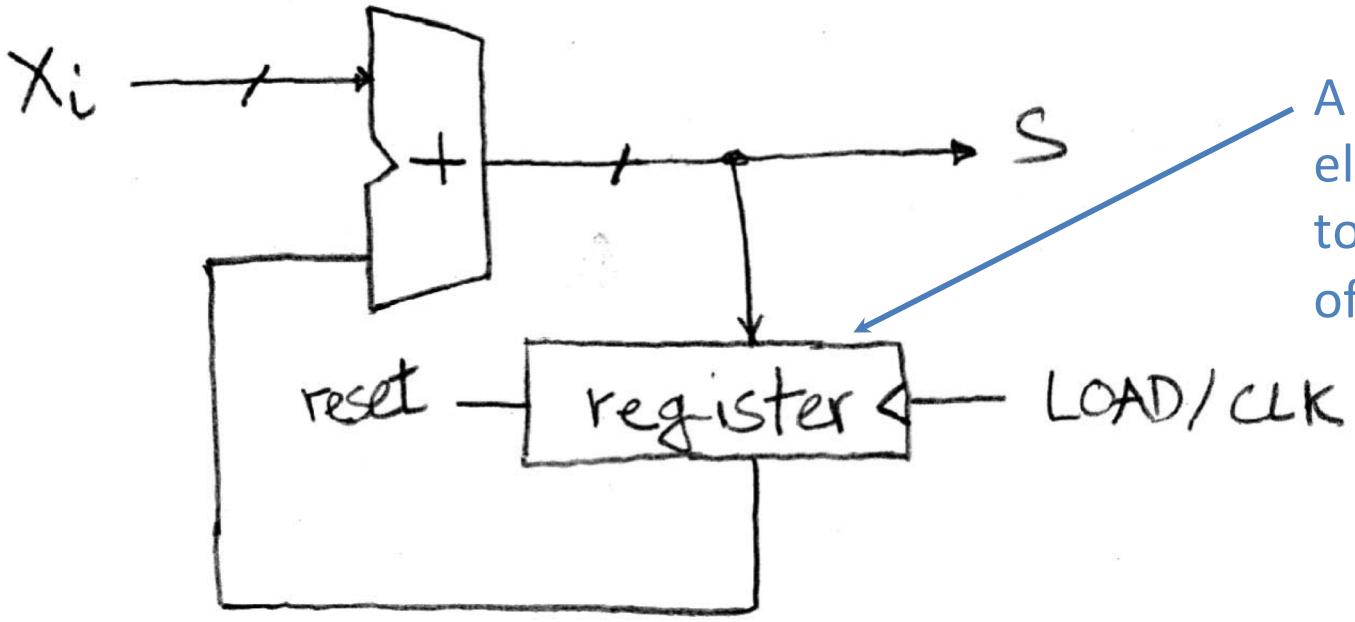
# First Try: Does this work?




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

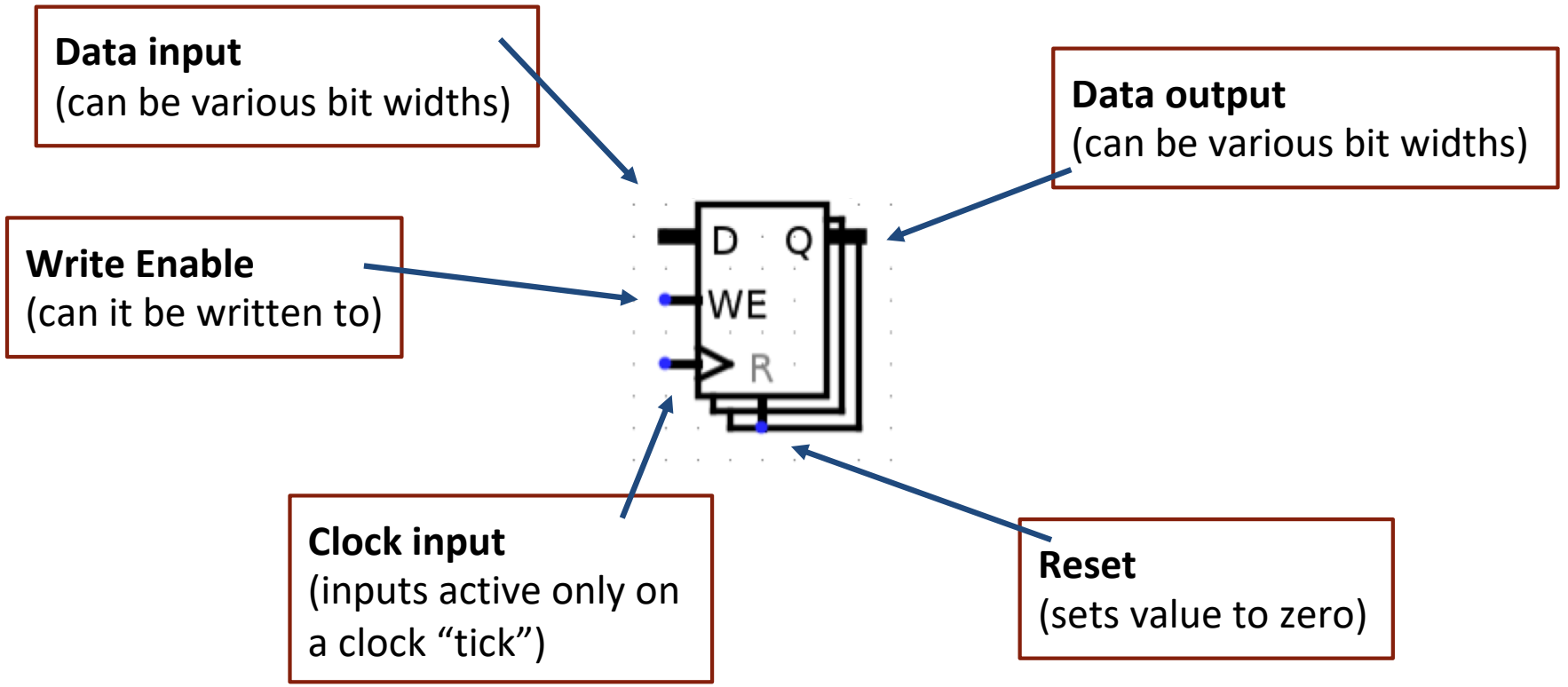
# Uses for State Elements

- 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

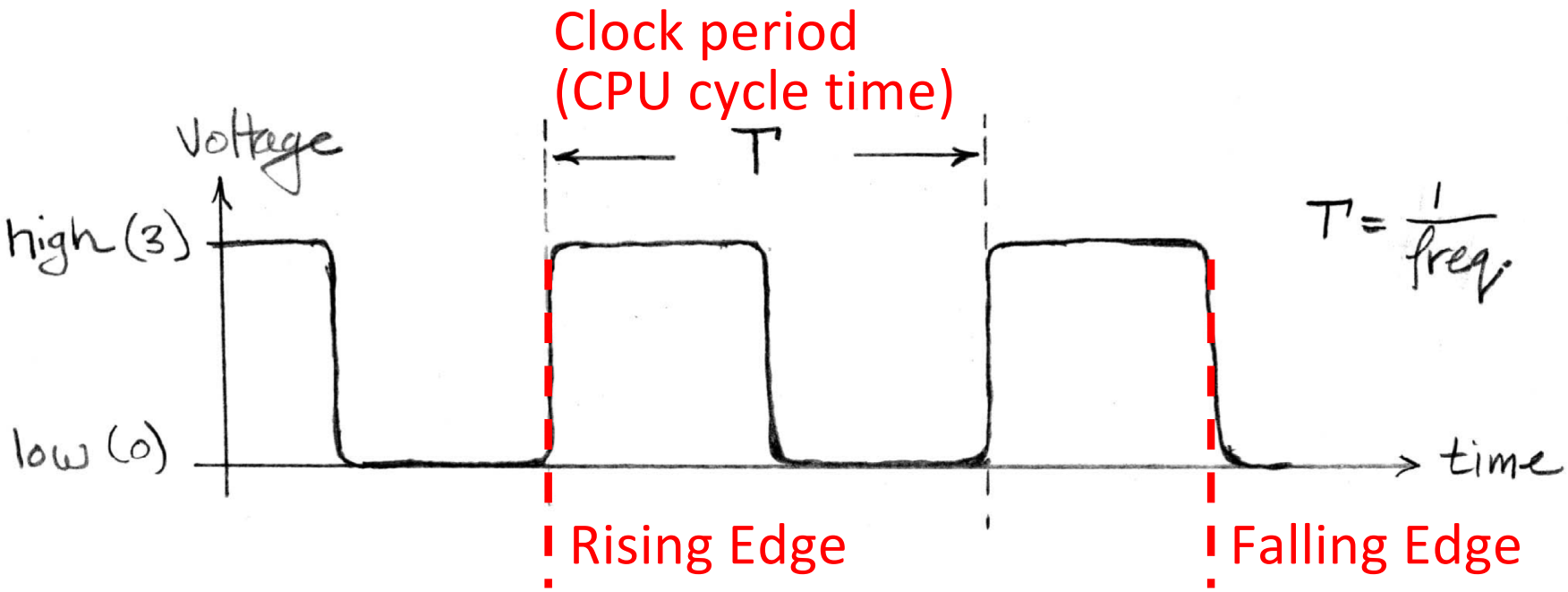
# Registers

Same as registers in assembly:

- small memory storage locations

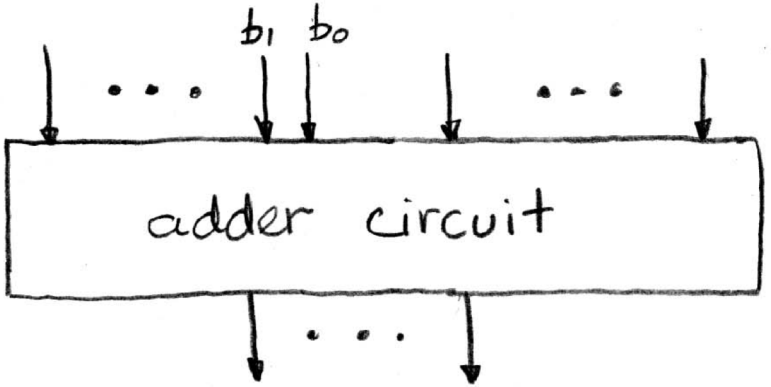


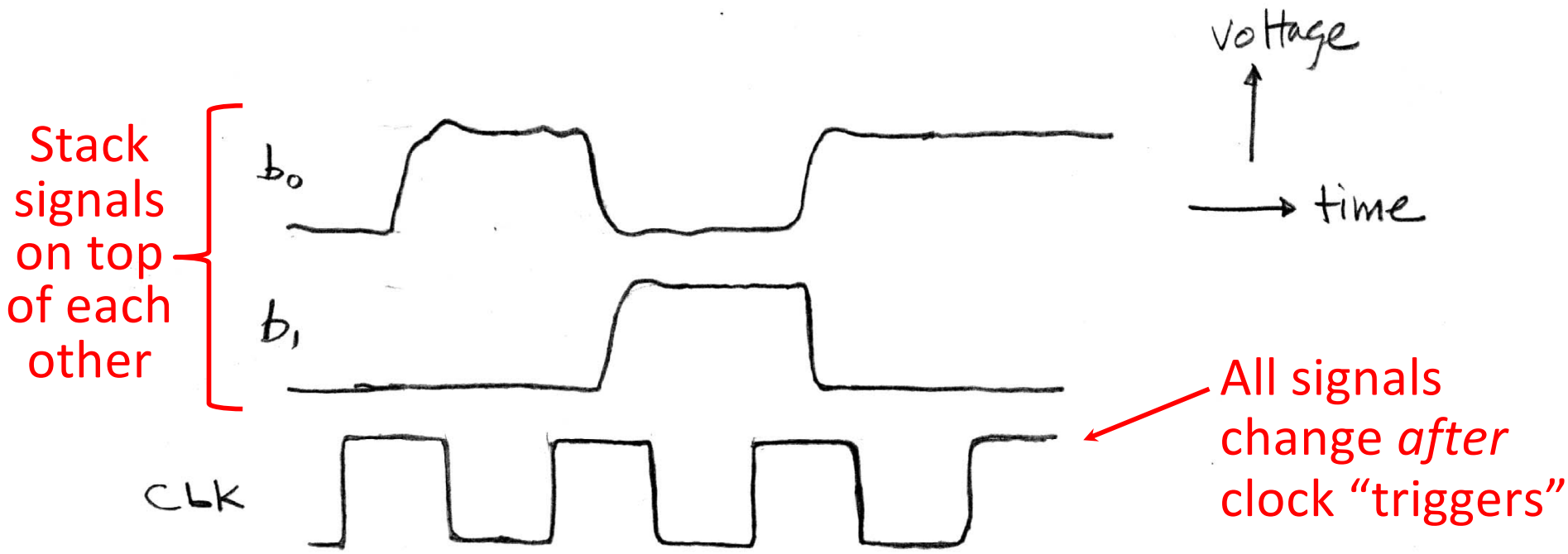
# Signals and Waveforms: Clocks




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

# Signals and Waveforms



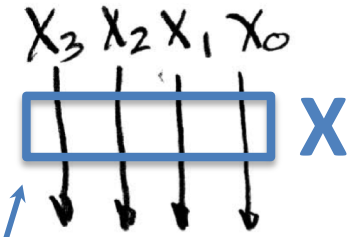



# Dealing with Waveform Diagrams

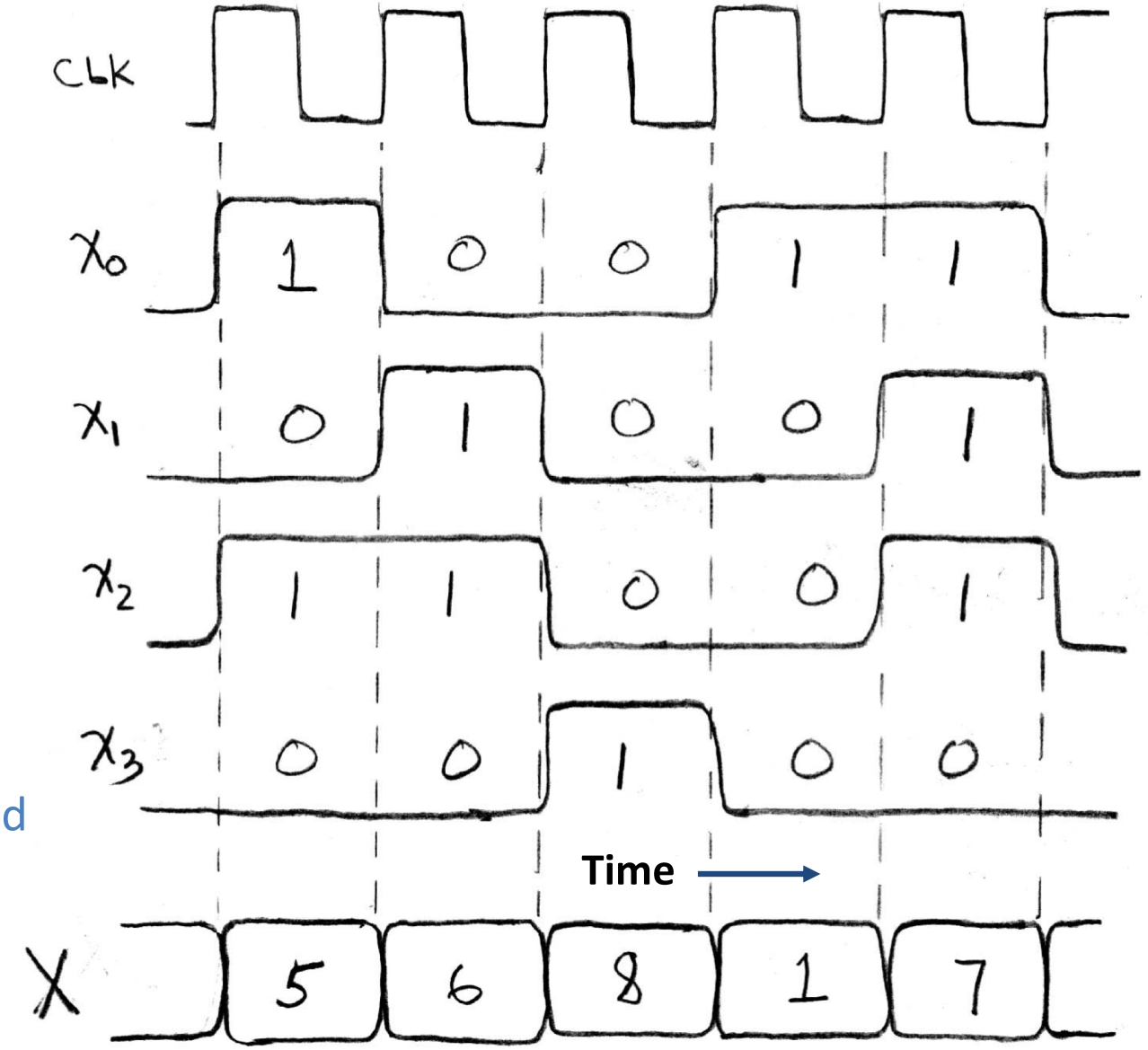
- Easiest to start with CLK on top
  - Solve signal by signal, from inputs to outputs
  - Can only draw the waveform for a signal if *all* of its input waveforms are drawn

- When does a signal update?
  - A *state element* updates based on CLK triggers
  - A *combinational element* updates ANY time ANY of its inputs changes

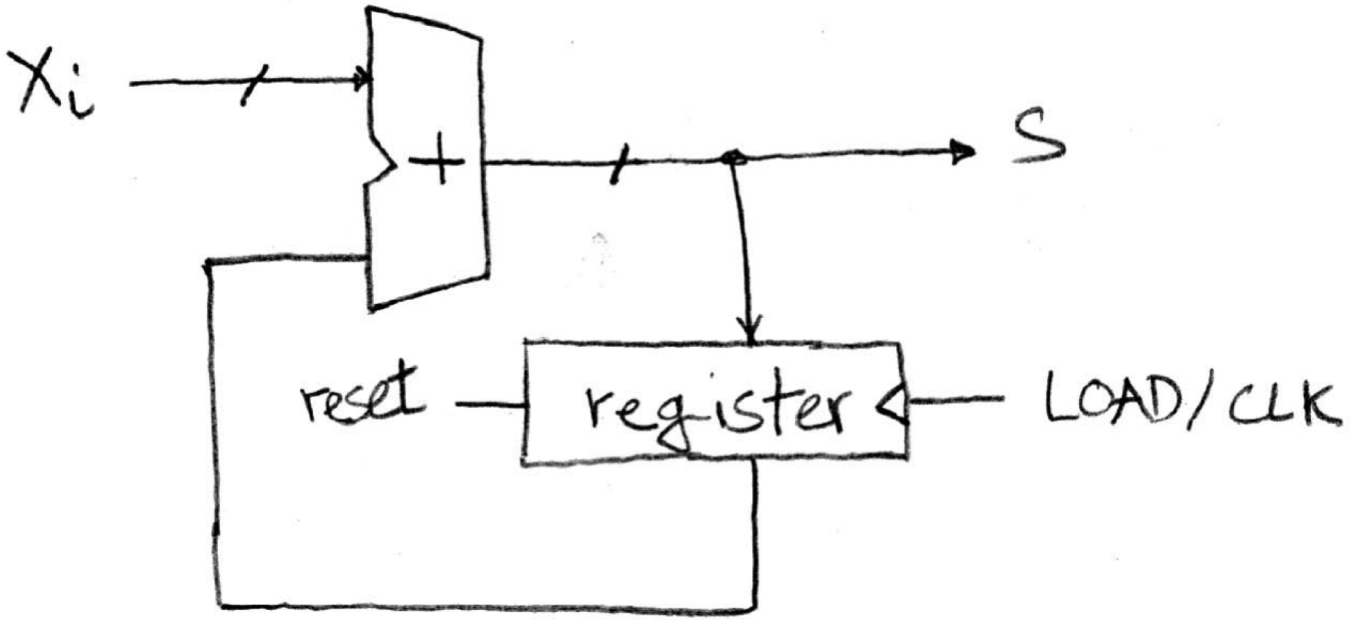
# Signals and Waveforms: Grouping



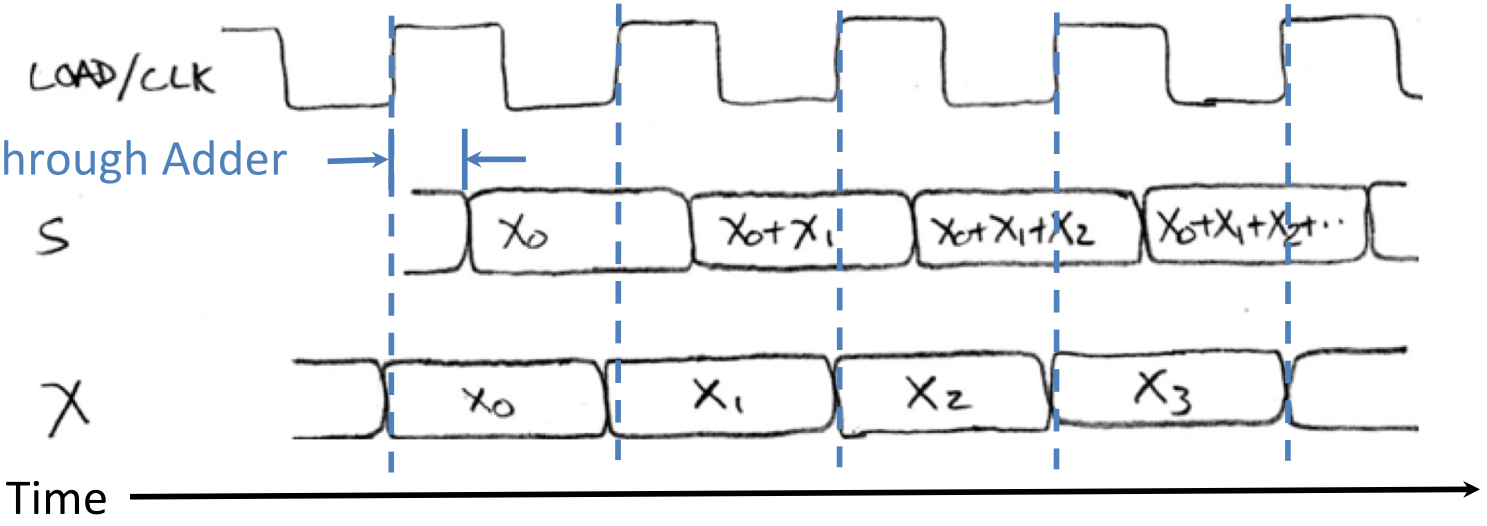

A group of wires when interpreted as a bit field is called a *bus*



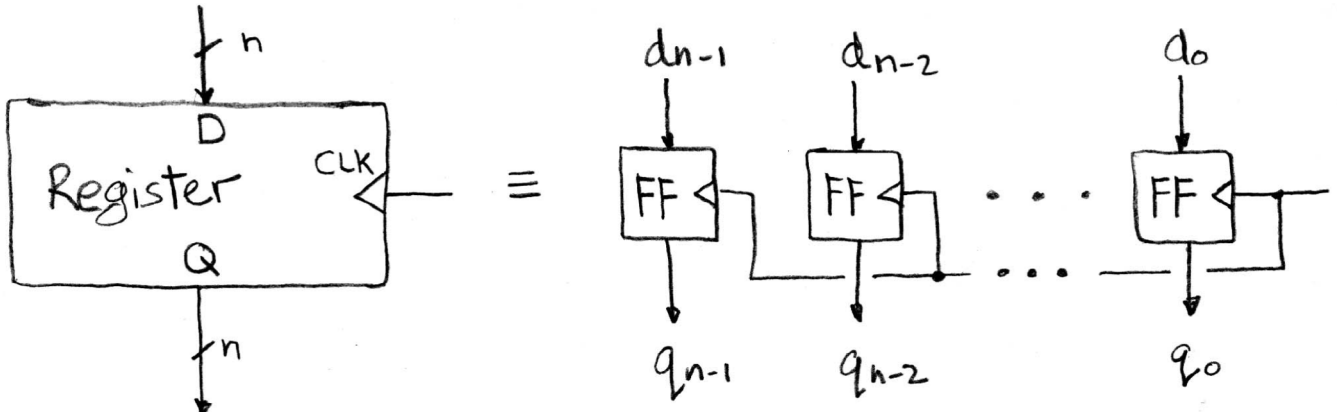
# Second Try: How About This?




Rough timing …



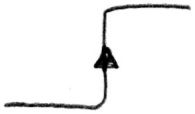
# Register Internals



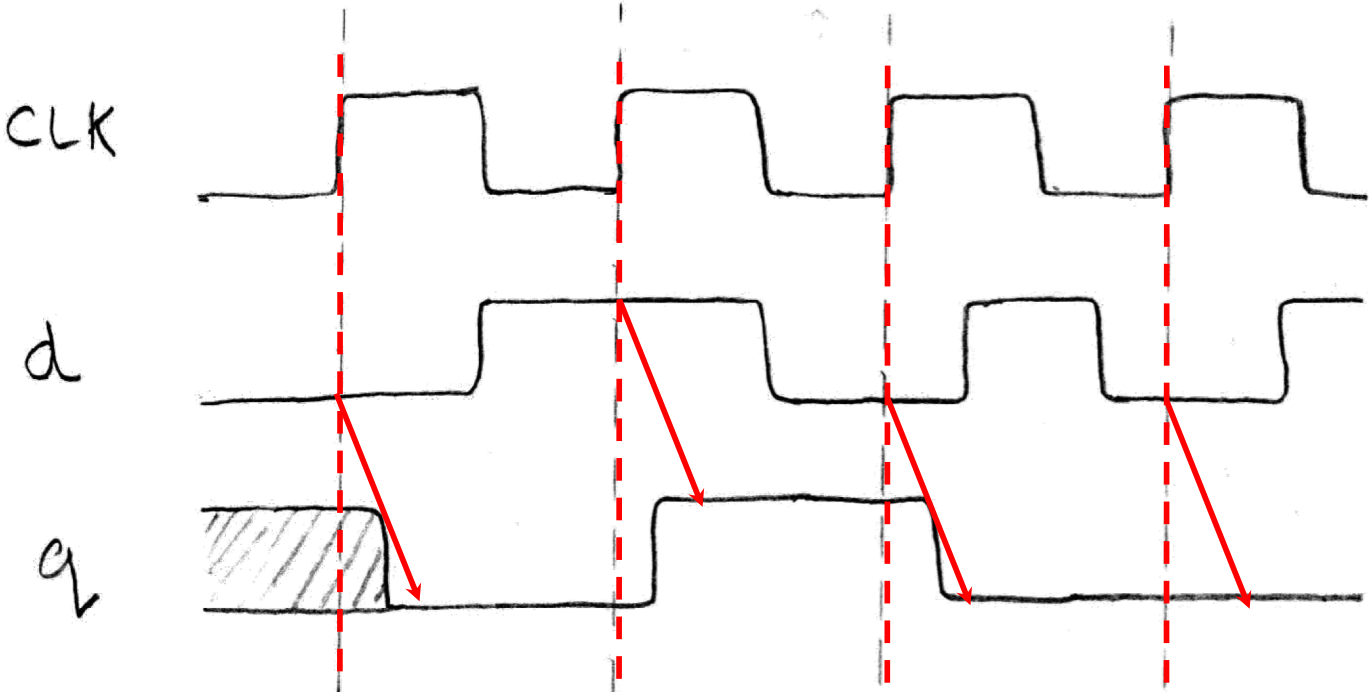

- n instances of a *"Flip-Flop"*
  - Output flips and flops between 0 and 1
- Specifically this is a "D-type Flip-Flop"
  - D is "data input", Q is "data output"
  - In reality, has 2 outputs (Q and $\overline{Q}$), but we only care about 1
- http://en.wikibooks.org/wiki/Practical_Electronics/Flip-flops

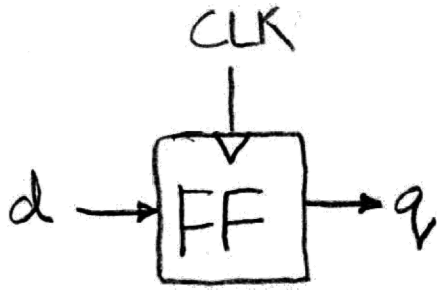
# Flip-Flop Timing Behavior (1/2)

- Edge-triggered D-type flip-flop
  - This one is “rising edge-triggered”



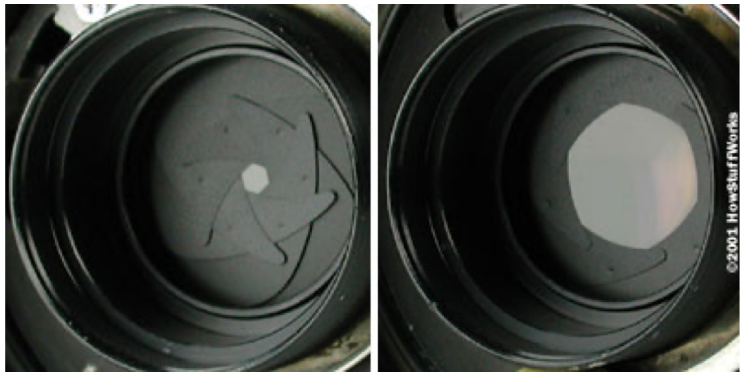
- “On the rising edge of the clock, input d is sampled and transferred to the output. At other times, the input d is ignored and the previously sampled value is retained.”
- Example waveforms:



# 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

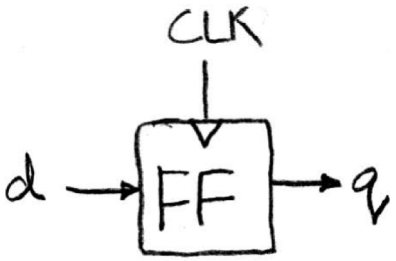


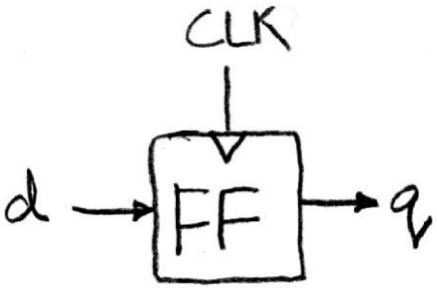

# Flip-Flop Timing Terminology (2/3)

- 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)

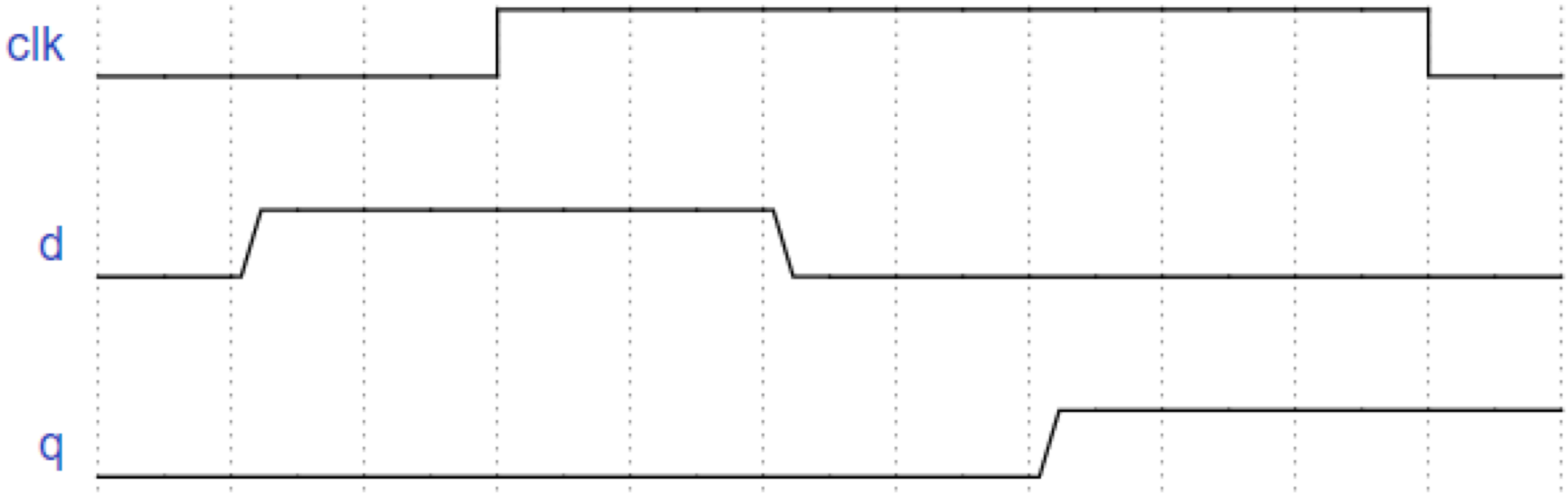
# Flip-Flop Timing Terminology (3/3)

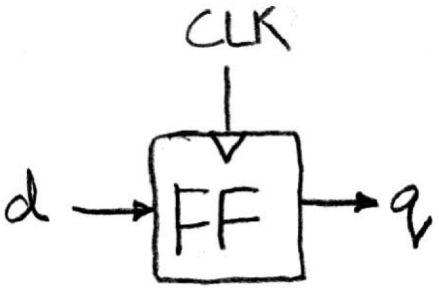
- Now applied to hardware:
  - *Setup Time:* how long the input must be stable *before* the clock trigger for proper input read
  - *Hold Time:* how long the input must be stable *after* the clock trigger for proper input read
  - *"Clock-to-Q" Delay:* how long it takes the output to change, measured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



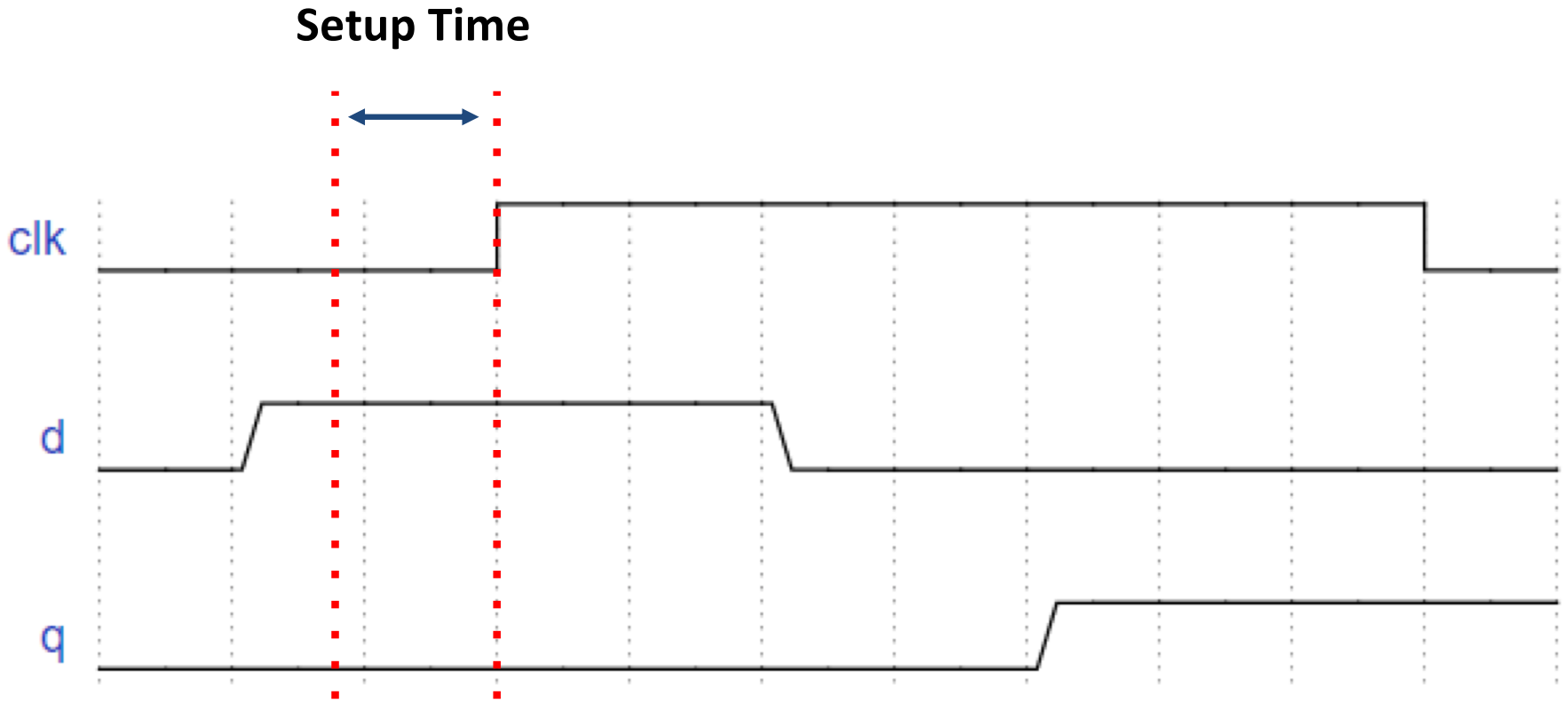


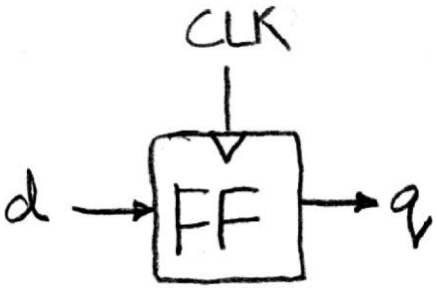

# Flip-Flop Timing Behavior



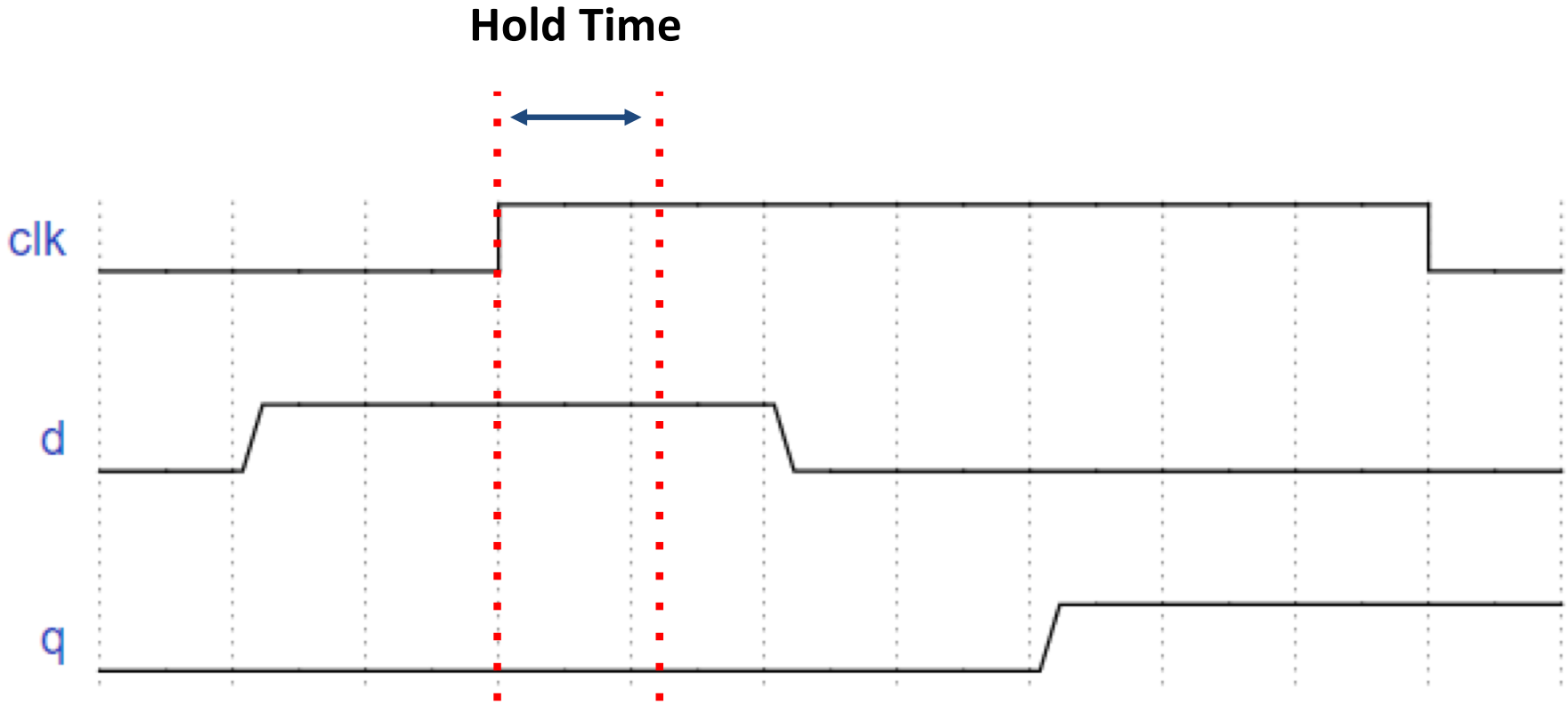


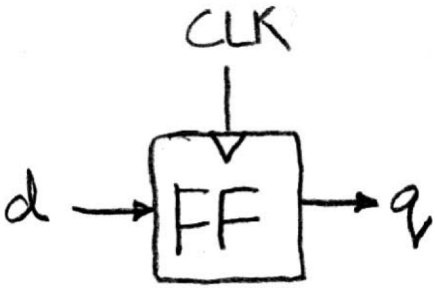

# Flip-Flop Timing Behavior



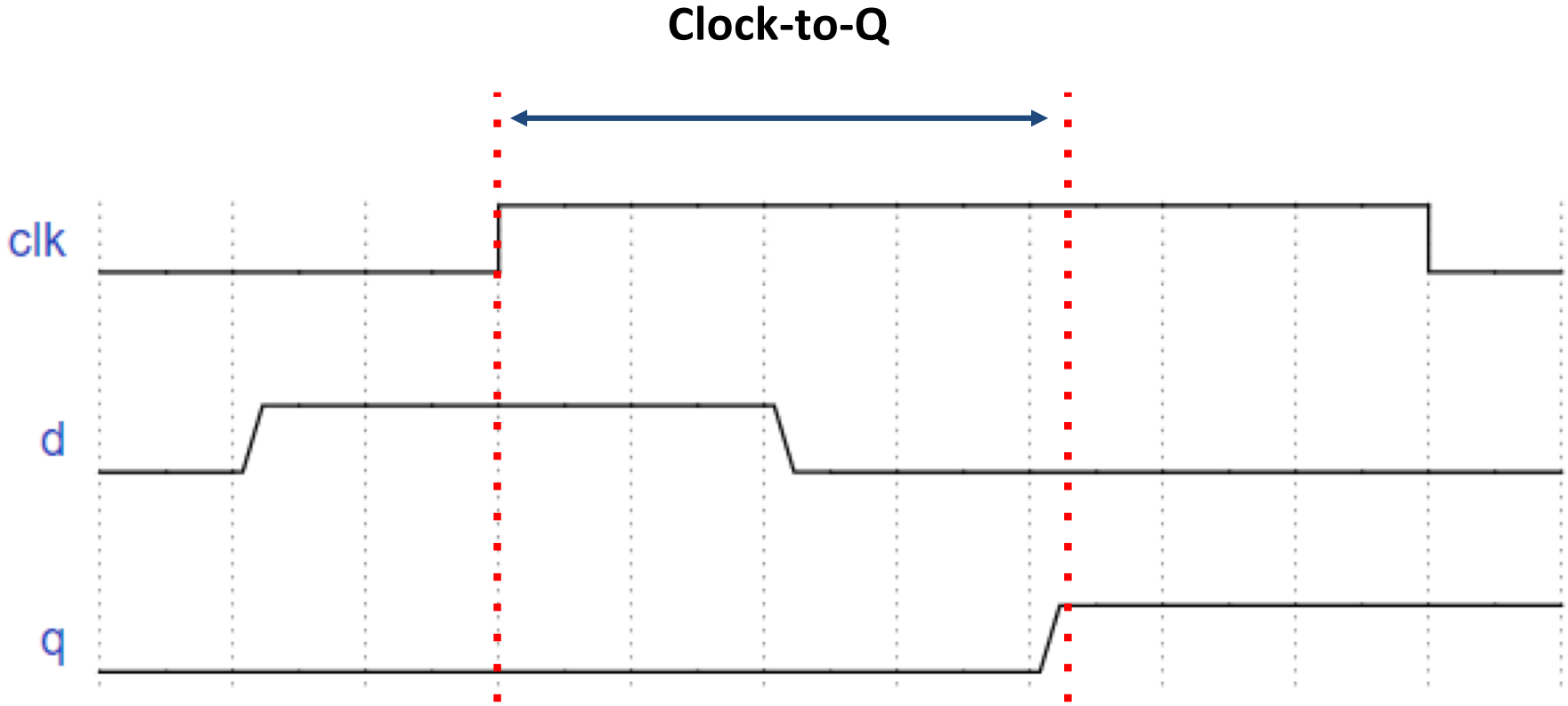



# Flip-Flop Timing Behavior

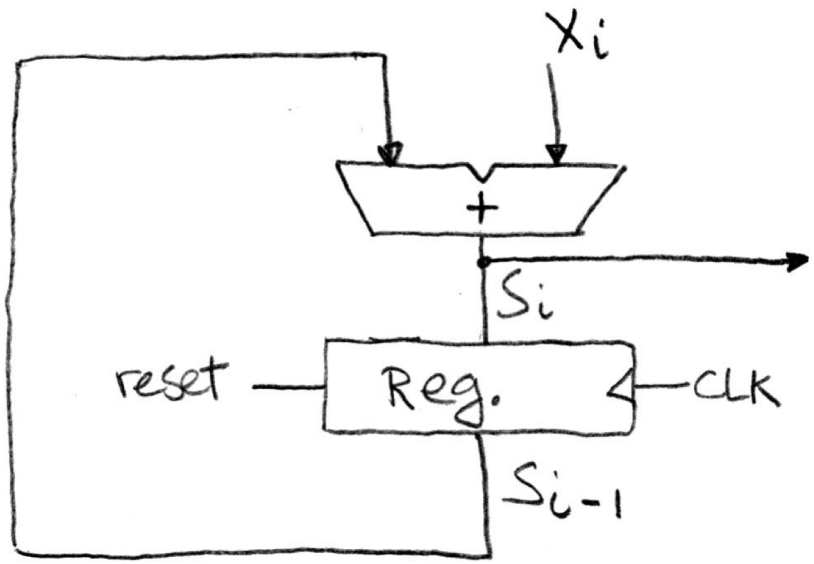




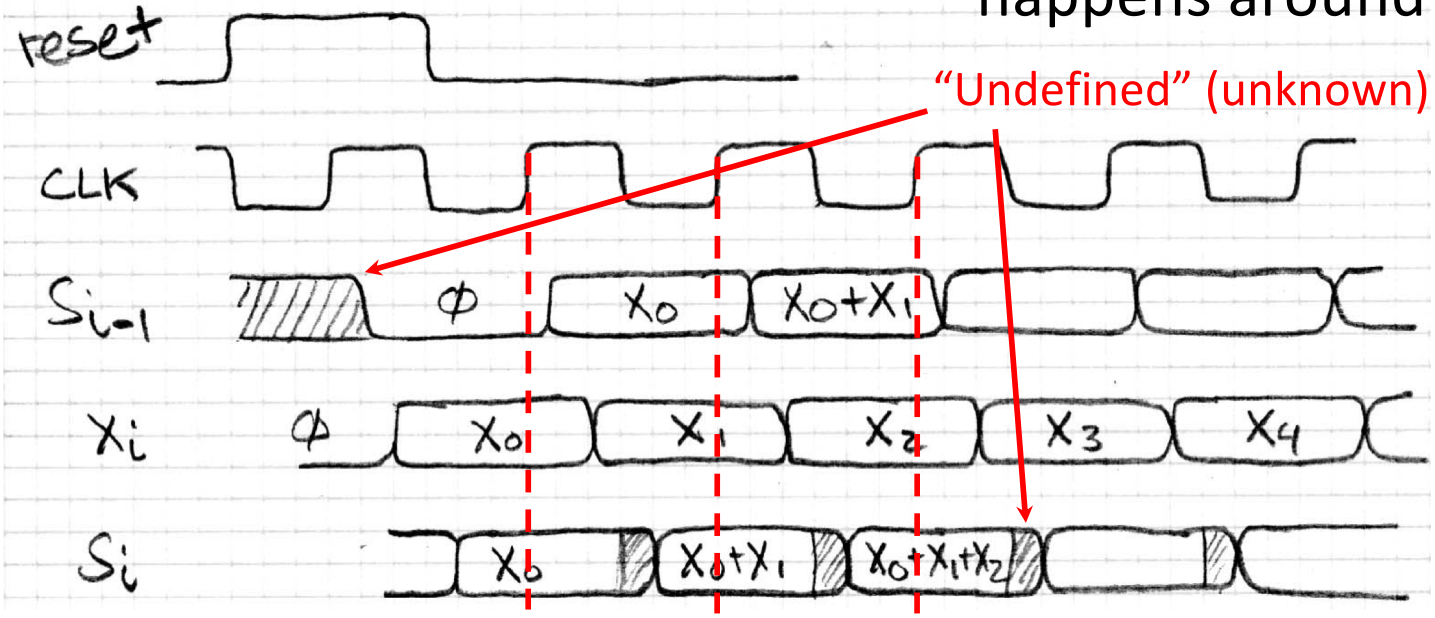

# Flip-Flop Timing Behavior



# 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



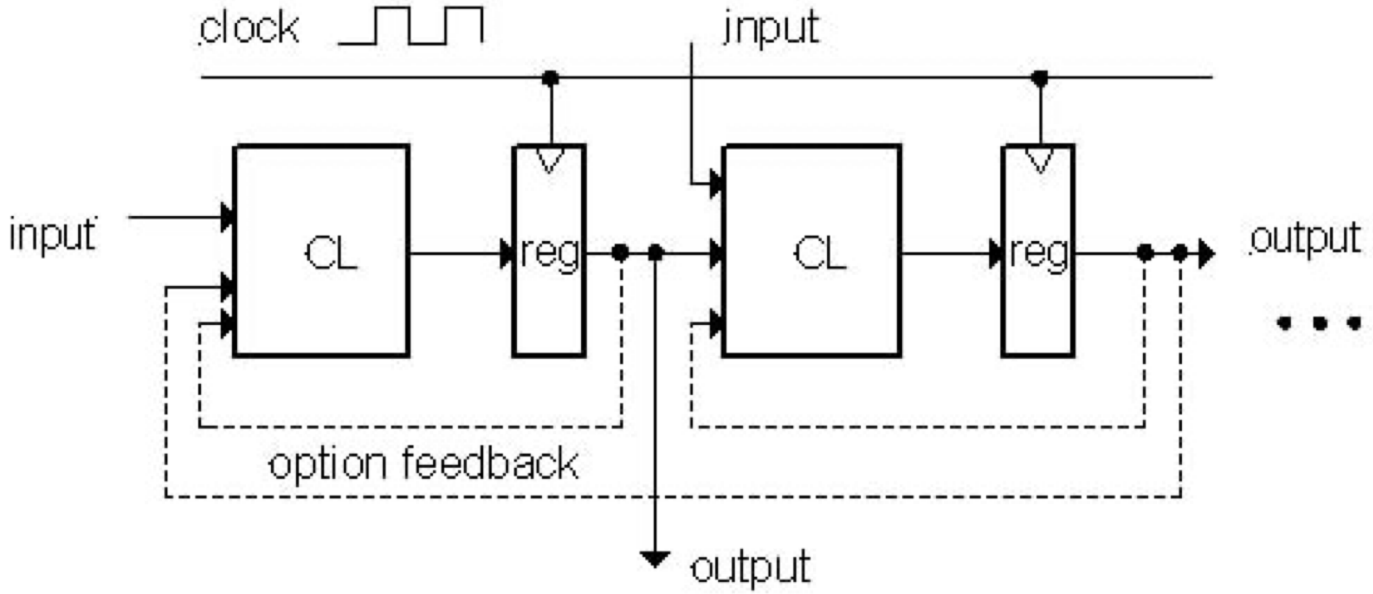
# Review of Timing Terms

- 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**
- Functional Units
- Summary

# Model for Synchronous Systems



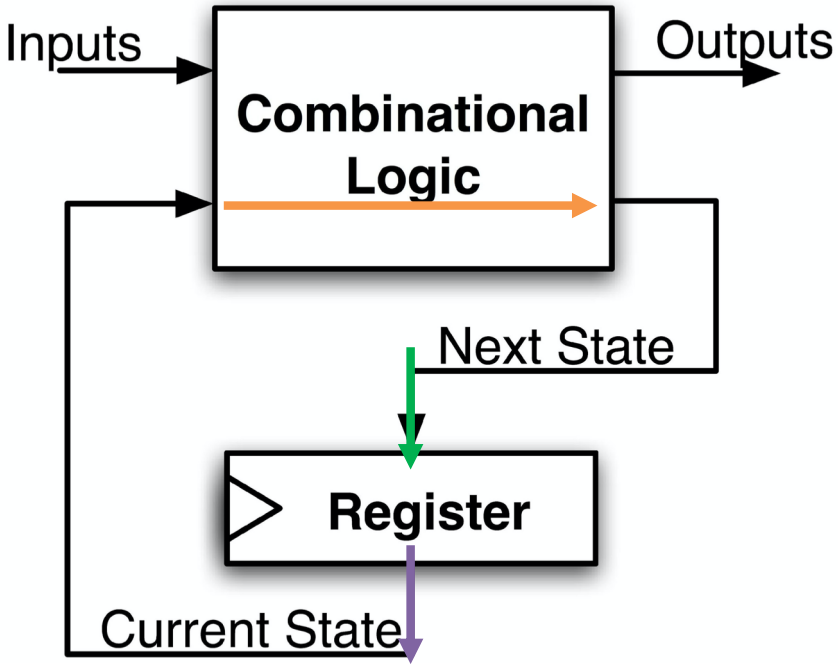

- Combinational logic blocks separated by registers
  - Clock signal connects only to sequential logic elements
  - Feedback is optional depending on application
- How do we ensure proper behavior?
  - How fast can we run our clock?

# 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

# 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)




Assumes Max Delay > Hold Time

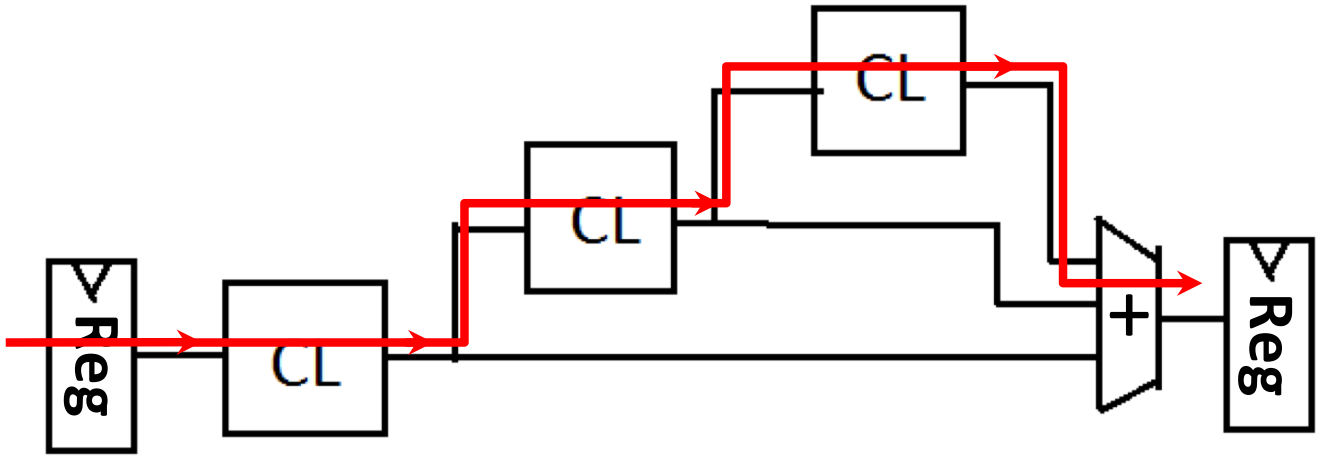
Max Delay = CLK-to-Q Delay
+ CL Delay
+ Setup Time

Min Period = Max Delay

Max Freq = 1/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

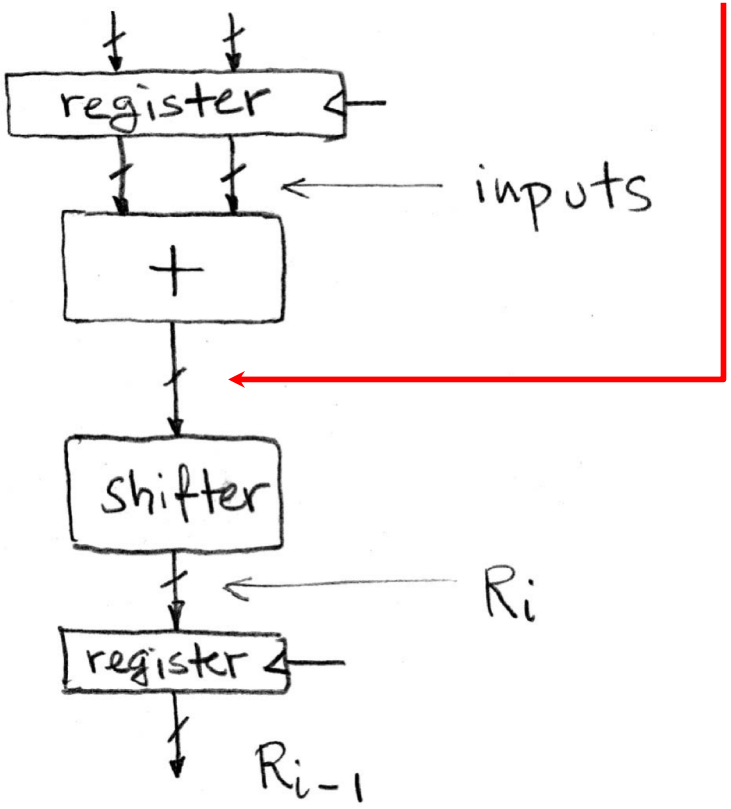
# How do we go faster?

Pipelining!

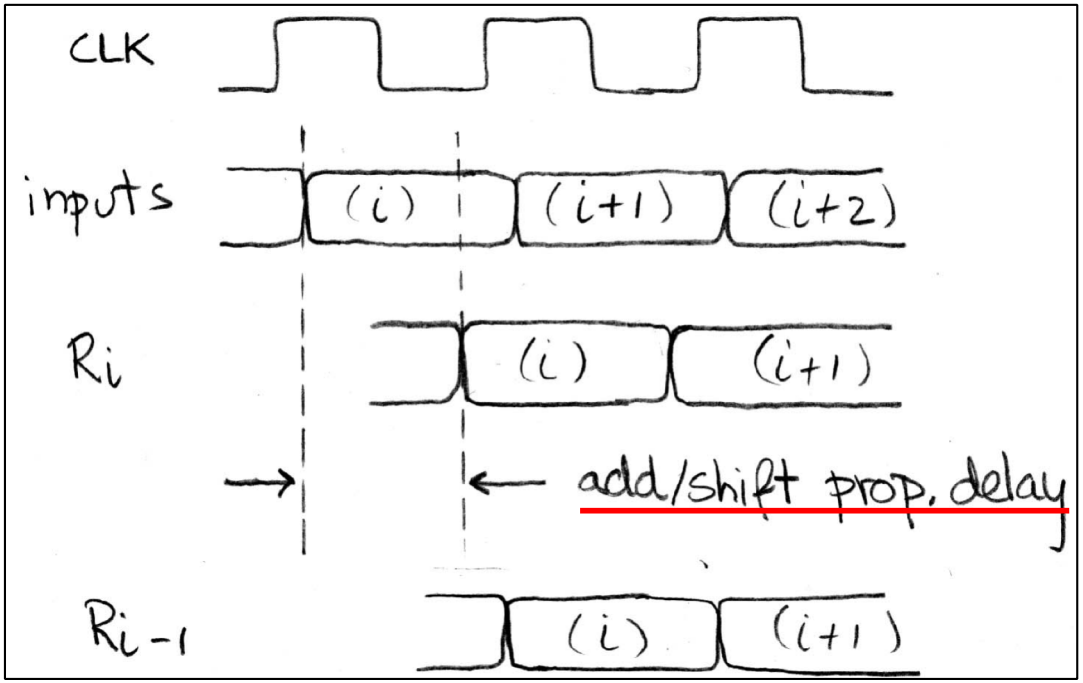
- Split operation into smaller parts and add a register between each one.

# Pipelining and Clock Frequency (1/2)

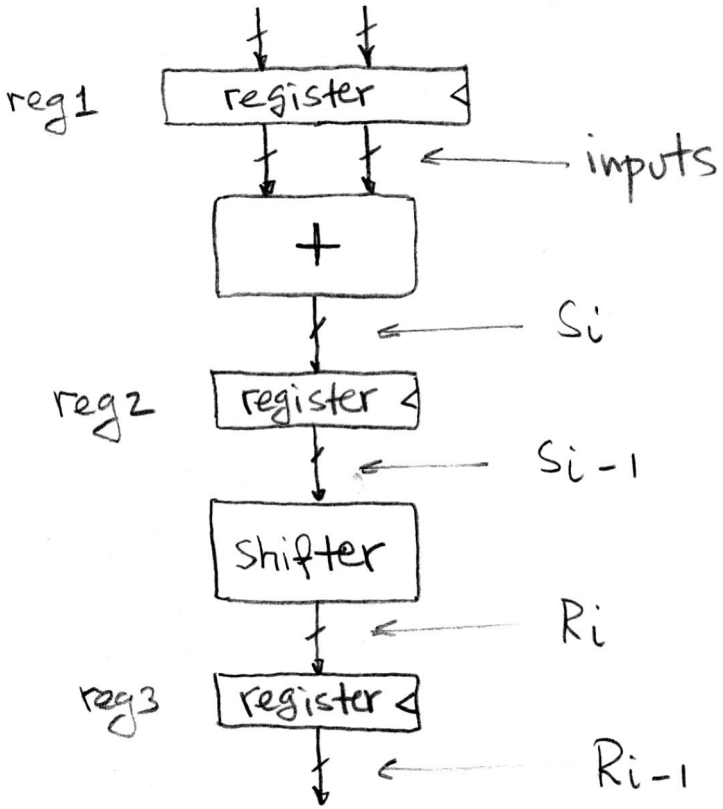
- Clock period limited by propagation delay of adder and shifter
  - Add an extra register to reduce the critical path!

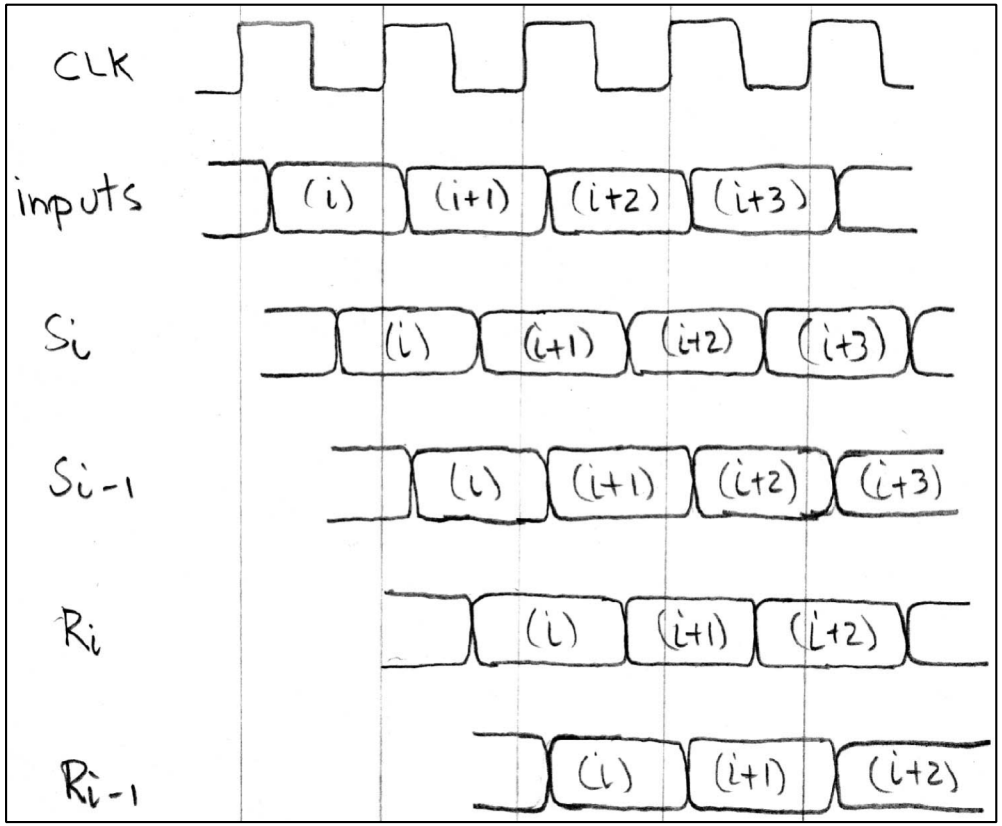



**Timing:**



# Pipelining and Clock Frequency (2/2)




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

# A Pipelining Analogy

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

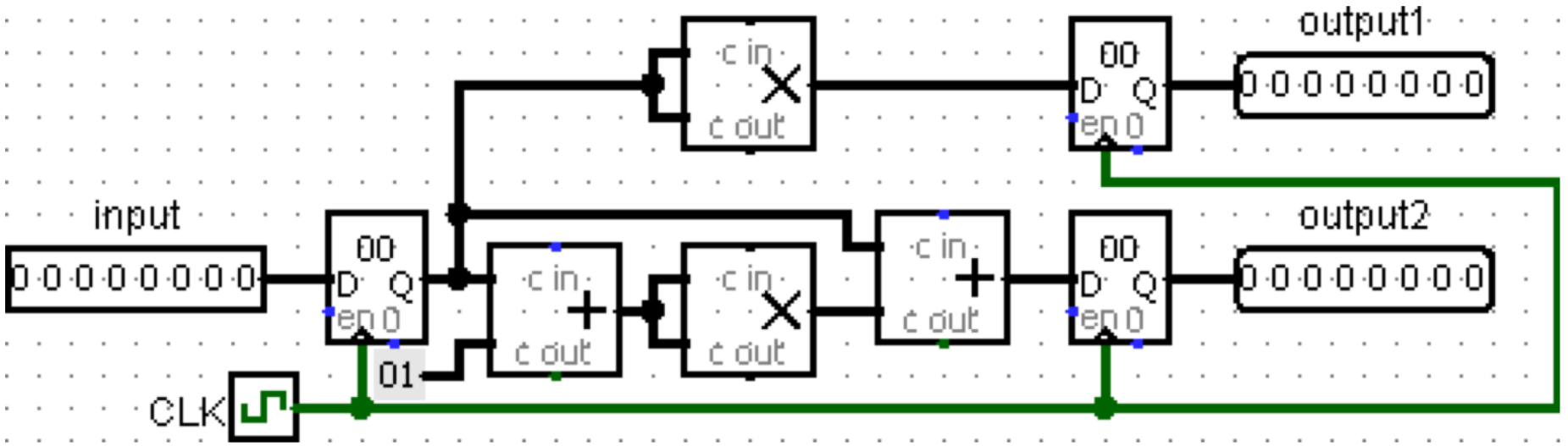
# Pipelining Basics

- 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.

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

What is the maximum clock-to-q time?

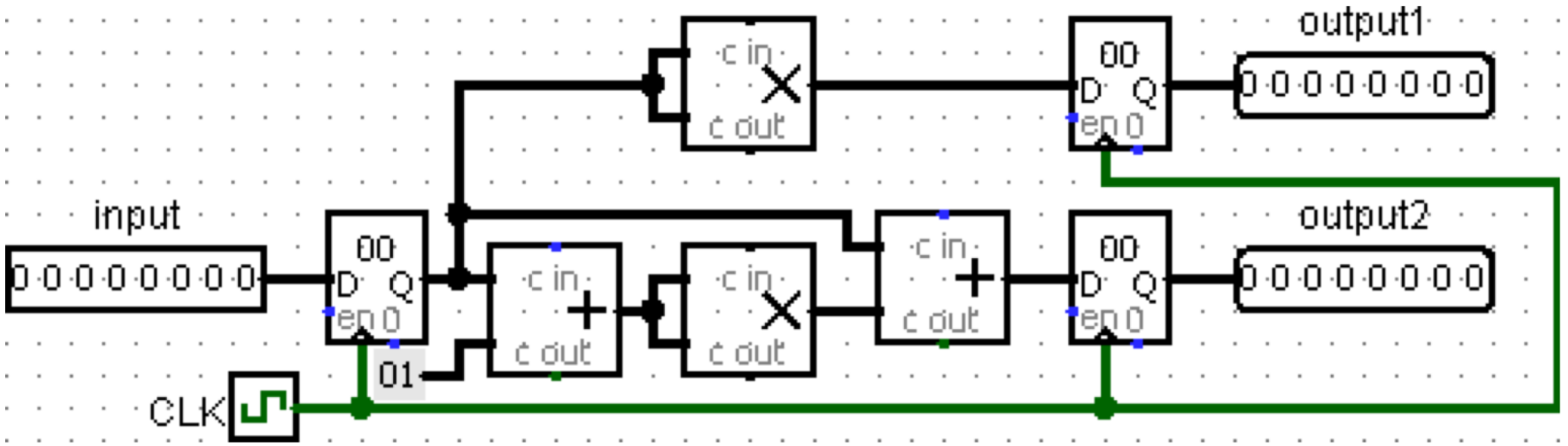



| | |
|---|---|
| (A) | 550 ps |
| (B) | 750 ps |
| (C) | 500 ps |
| (D) | 700 ps |

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

$t_{add}$ = 100 ps. $t_{mult}$ = 200 ps. $t_{setup}$ = $t_{hold}$ = 50 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
- **Functional Units**
- Summary

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

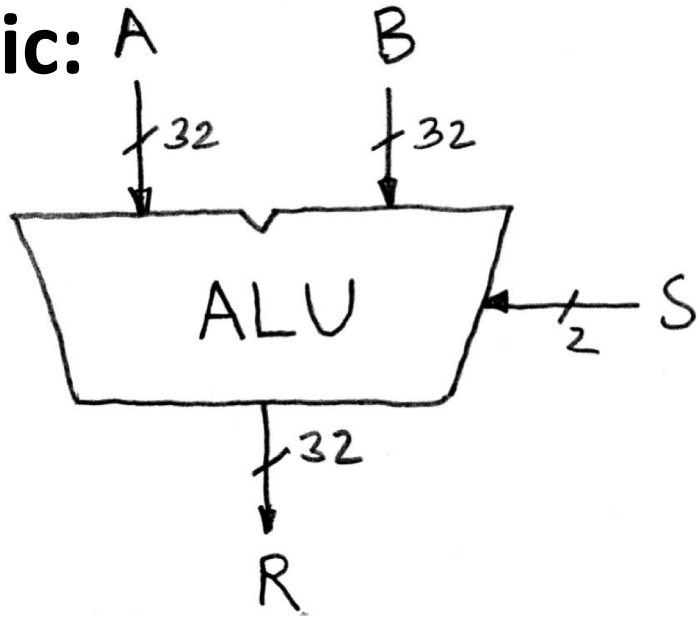
- Now that we know sequential logic, we can explore some pieces of a processor!

- 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

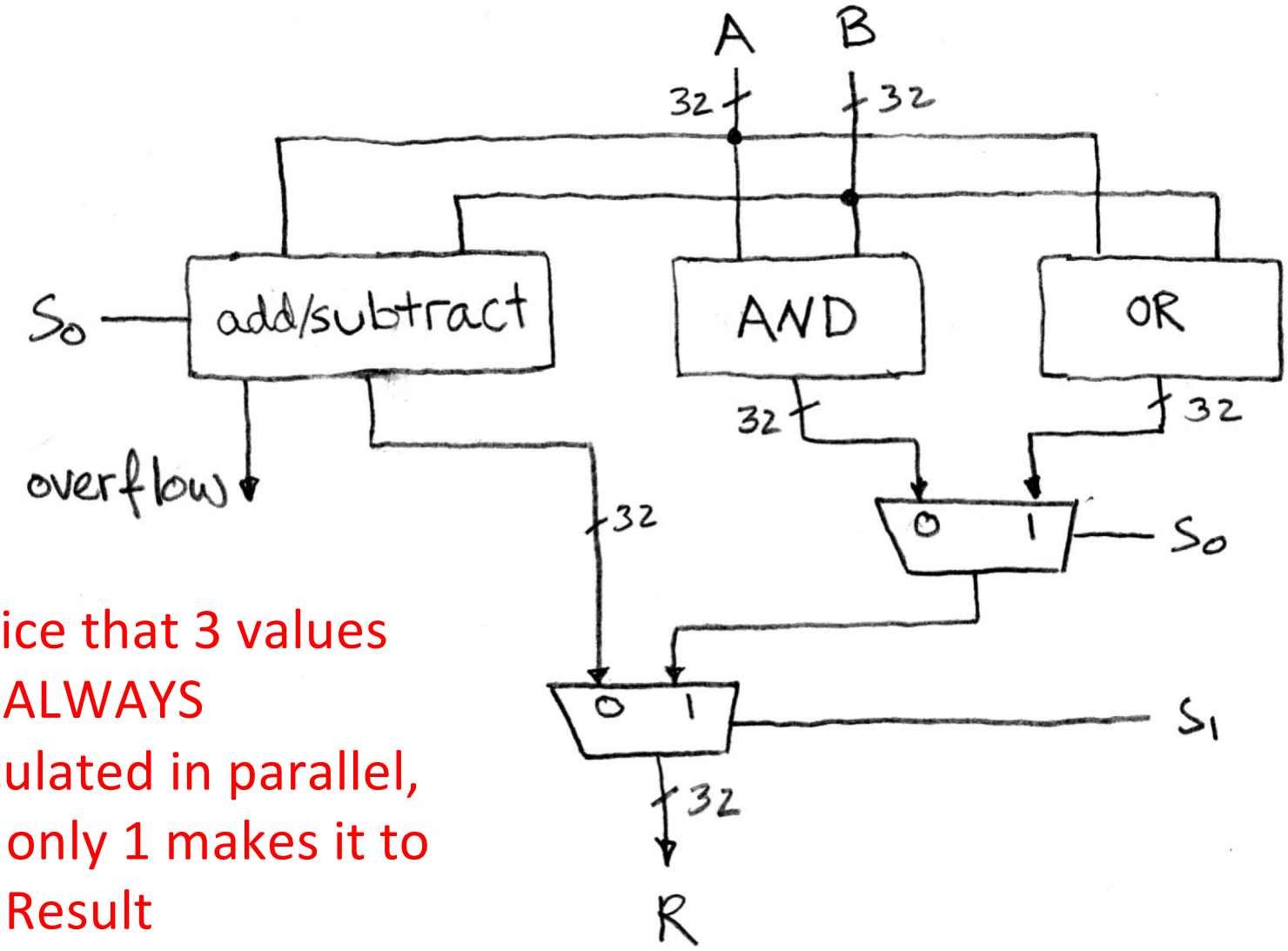
# Arithmetic Logic Unit (ALU)

- 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 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

# Agenda

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

# 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