# CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

## Lecture 10: Combinational Logic, FSMs

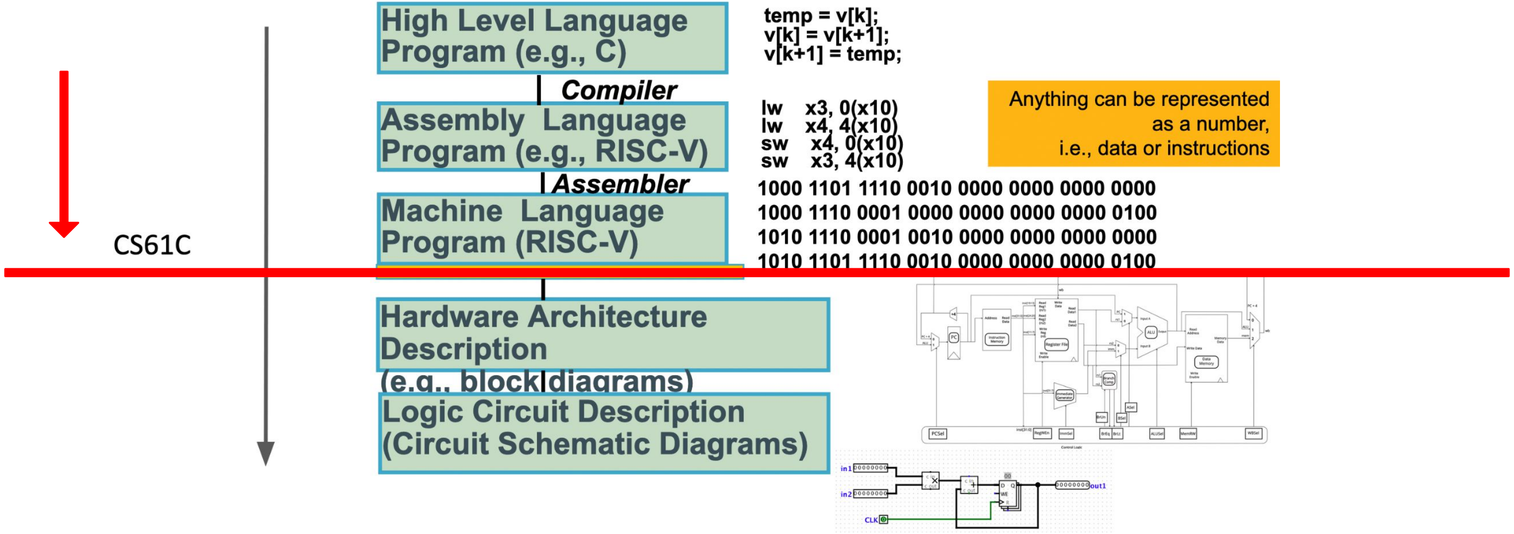
Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

# Announcements

- Midterm Friday! Logistics on Ed/the course website.
    - Review sessions Tuesday 5-7pm, Thursday 3-5pm (different content - see Ed for details)
    - Additional conceptual OH M/W 5-6pm in Soda 411
- Extensions
    - 7 day limit
- OH clarification
    - No lab tickets in OH on Tuesdays/Thursdays
- Labs 3 and 4 due tomorrow (Tuesday), Homework 3 due Wednesday (long!)

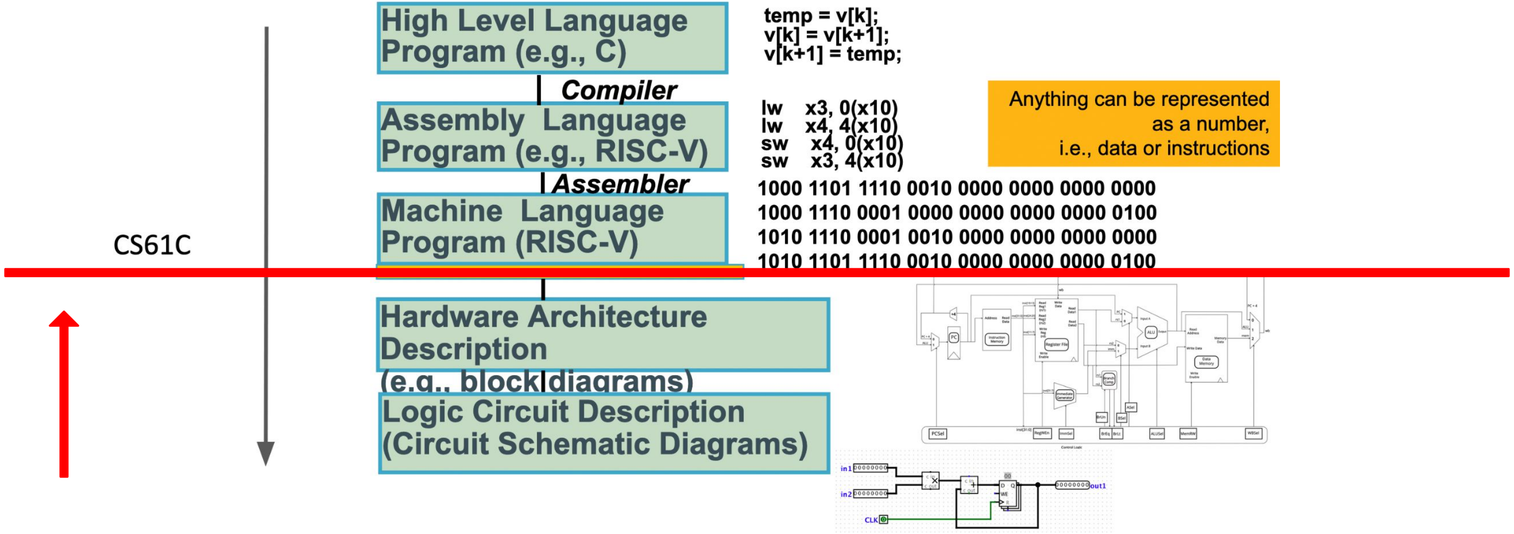
# So far…

- C, RISC-V generally exist on the software side of the stack
  - Nuance: the ISA does determine much of how a processor is implemented
- We built our way down from C to machine code



# Next 5-ish lectures:

- How a modern processor is built, starting with basic elements as building blocks



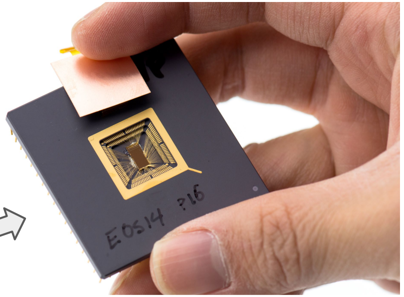
# Another way to put it…

Previously:

**How do we run this code:**

```c
#define SPOCK 1701
int KIRK = 1701;
int sulu(int scotty) {
    return scotty * scotty;
}
int main(int argc, char *argv[])  {
    int *chekov = malloc(sizeof(int) * 1701);
    if (chekov) free(chekov);
    sulu(SPOCK); // ← snapshot just before it returns
    return 0;
}
```

**On this piece of metal?**




Today:

**What’s in this thing?**

# Why study hardware?

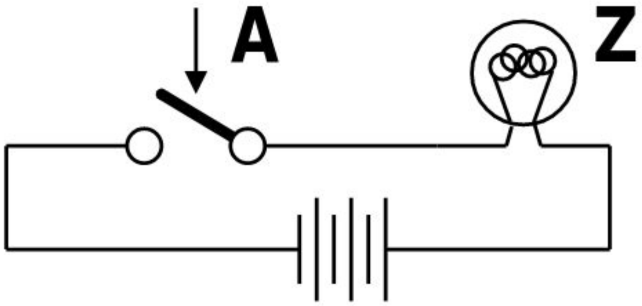
1. The cliché:
    - To really understand how computers work and become a better performance programmer.
2. Understand capabilities and limitations of HW in general and processors in particular.
    - Why is my computer so slow? Why does my battery run down?
    - Someday, you may need to decide between different hardware platforms, or even design custom HW for extra performance.
3. $$$$
    - In addition to hardware companies like Apple, Intel, or NVIDIA, traditionally software companies Google, Amazon, Meta do their own hardware design! Even some financial firms employ hardware engineers.
4. Background for more in-depth HW courses (EECS 151, CS 152)
    - More fun!

# CS 61C Hardware Roadmap

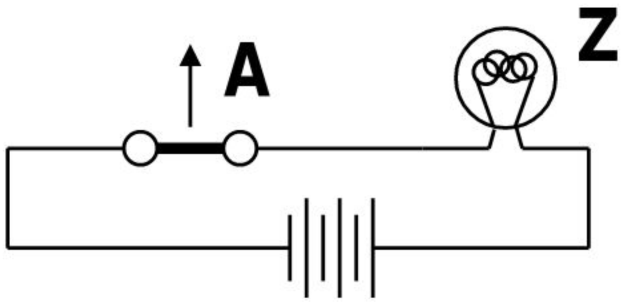
| |
|---|
| Processor (CPU) Architecture<br>● Datapath<br>● Control Logic<br>● Pipelining |
| Synchronous Digital Systems<br>● Sequential Logic<br>● Combinational Logic |
| **Transistors** |

# Transistors

# Switches

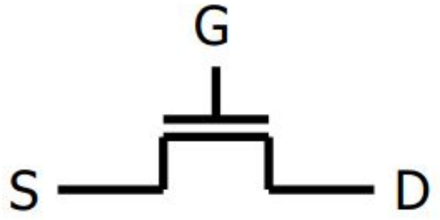



When the switch is open, the light bulb isn’t on.



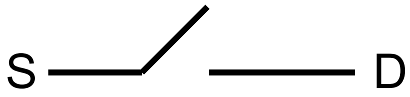

When the switch is closed, the light bulb is on.

# Transistors: n-type




An n-type transistor is drawn like this.

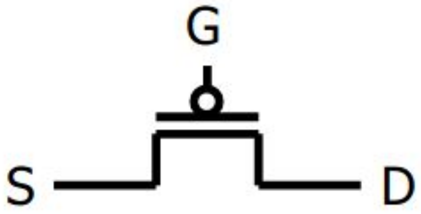
Drain, Gate, and Source are terminals with some voltage value (let's say 0 to 1).



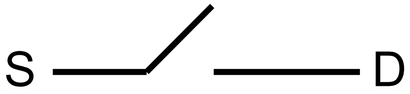

If G ≤ S, then the switch is open.
e.g. G=0, S=0


If G > S, then the switch is closed.
e.g. G=1, S=0

# Transistors: p-type







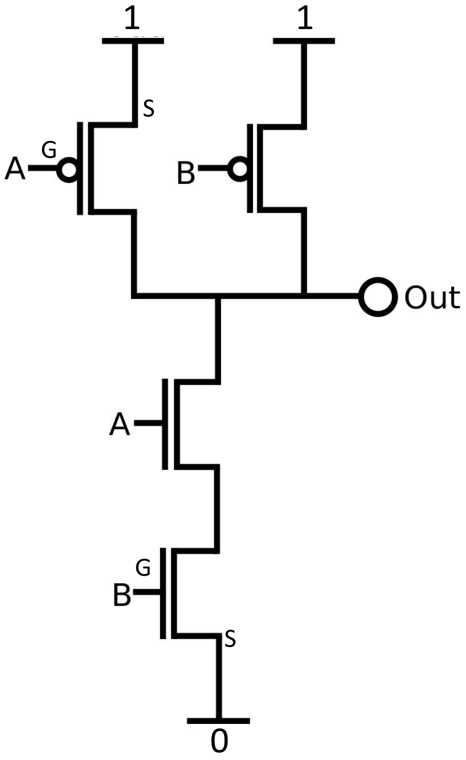

A p-type transistor is drawn like this.

Drain, Gate, and Source are terminals with some voltage value (let’s say 0 to 1).

If G < S, then the switch is closed.
e.g. G=0, S=1

If G ≥ S, then the switch is open.
e.g. G=1, S=1

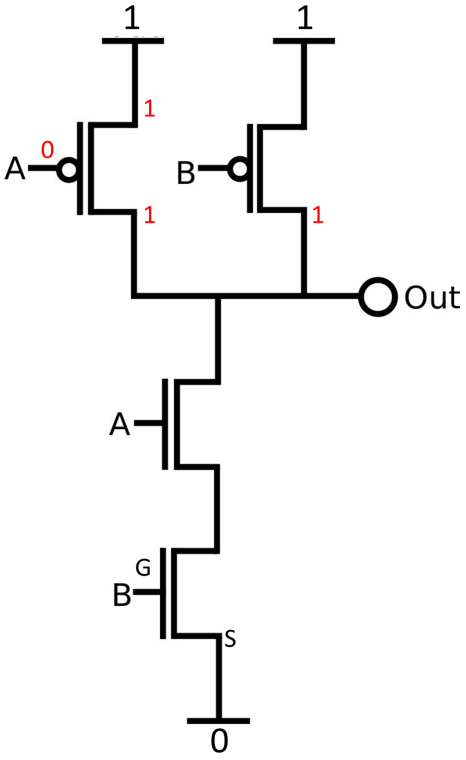
# Transistors: CMOS




What's the out voltage if A is low, and B is low?

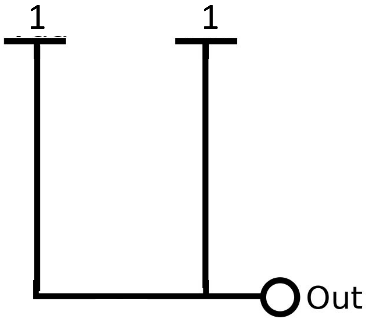
| A | B | Out |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

# Transistors: CMOS

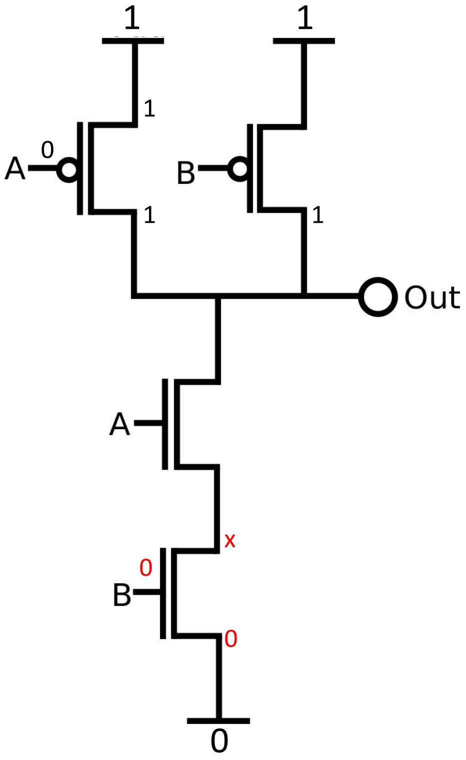



What's the out voltage if A is low, and B is low?

| A | B | Out |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

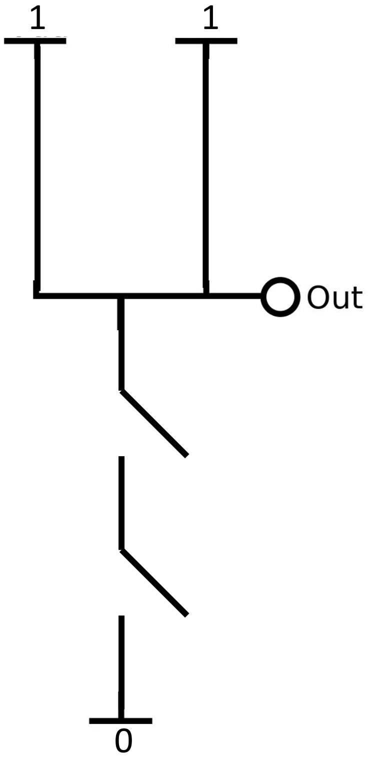


# Transistors: CMOS

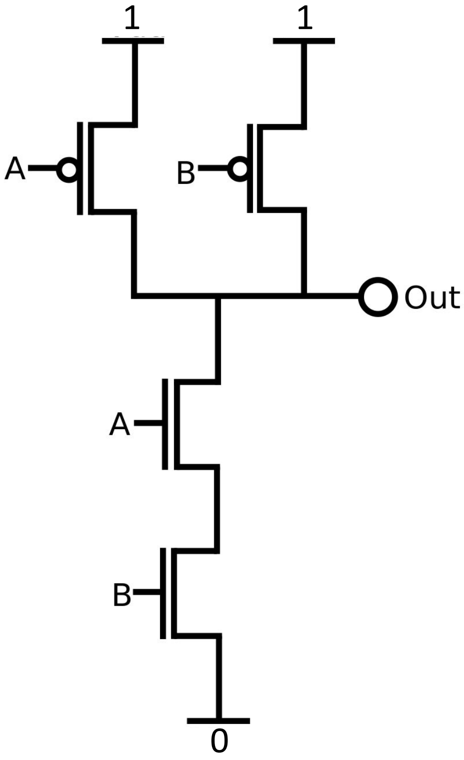



What's the out voltage if A is low, and B is low?

| A | B | Out |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

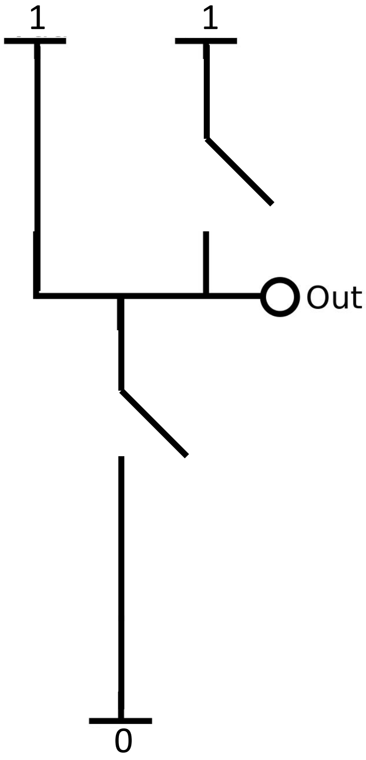


# Transistors: CMOS

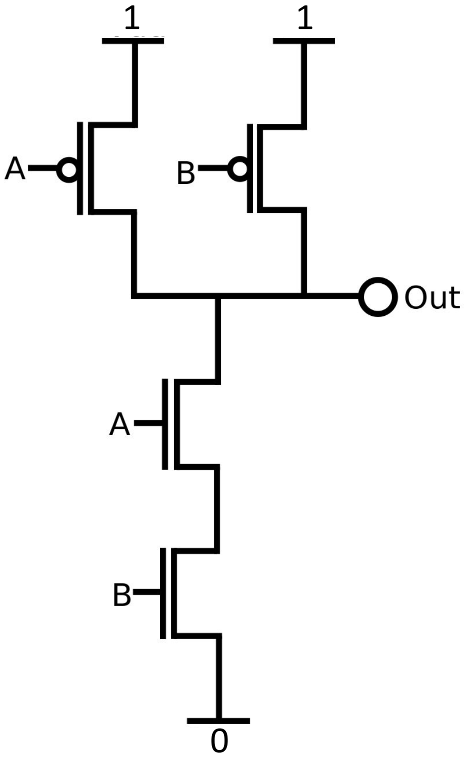



What's the out voltage if A is low, and B is high?

| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |

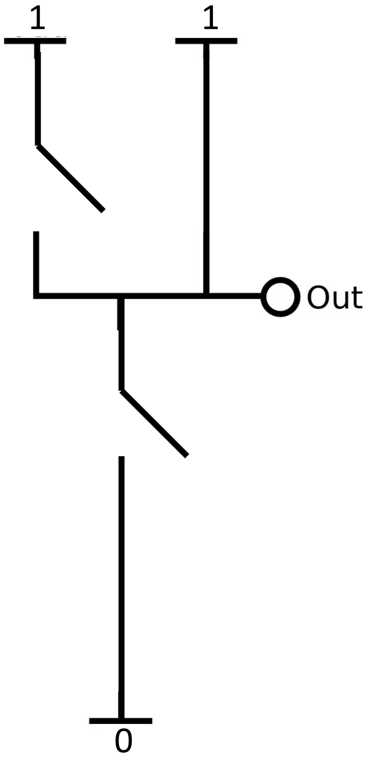


# Transistors: CMOS

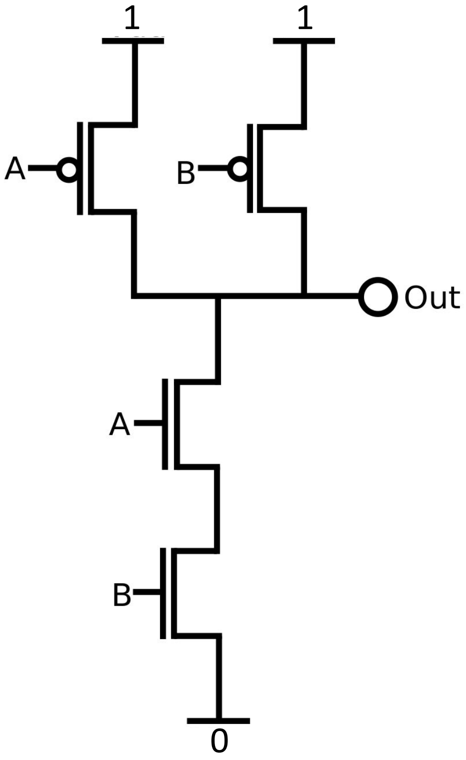



What's the out voltage if A is high, and B is low?

| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | |
| 1 | 1 | |

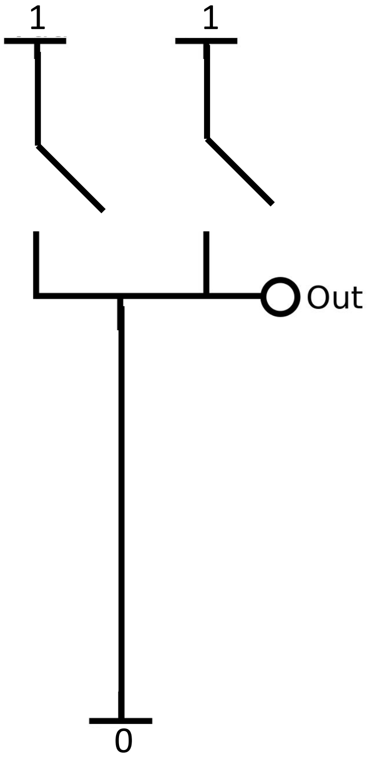


# Transistors: CMOS

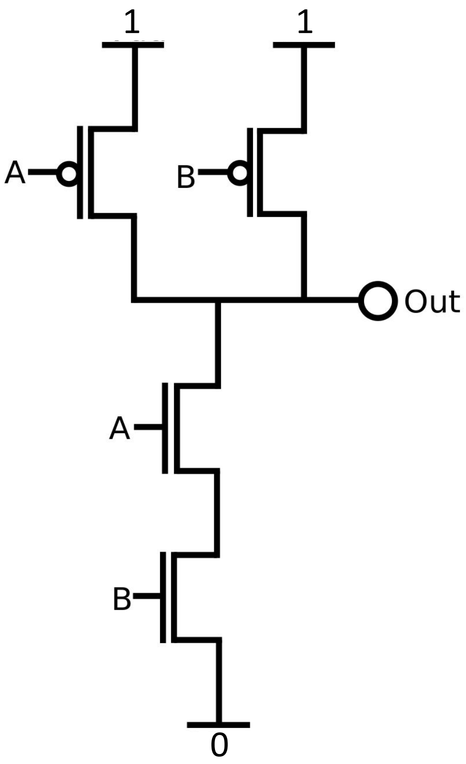



What's the out voltage if A is high, and B is high?

| A | B | Out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | |



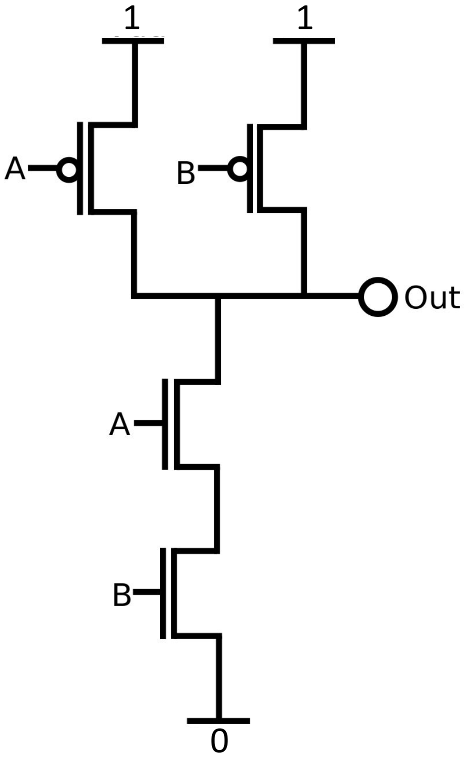
# Transistors: CMOS




| A | B | Out |
| --- | --- | --- |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

We just built a logic gate out of transistors!

# Transistors: CMOS




C = Complementary. We use p-type and n-type transistors together to build logic.

# CS 61C Hardware Roadmap

Processor (CPU)

- Datapath
- Control Logic
- Pipelining

Synchronous Digital Systems

- Sequential Logic
- **Combinational Logic**

Transistors

Note: Transistors will not be in scope for exams.

# Basic Logic Gates

# Logic Gates

- Operators with:
  - One or more 1-bit inputs
  - One 1-bit output
- Can be built out of transistors
- Can be represented as:
  - A block in a circuit diagram
  - A truth table, listing the output for every possible input
  - A Boolean algebra expression
- Used to perform bitwise operations
  - Recall: We saw bitwise operations (NOT, OR, AND, XOR) in C and RISC-V

# NOT

- One 1-bit inputs, labeled A
- One 1-bit output, labeled Out
- Can be represented as:


$$Out = \neg A$$

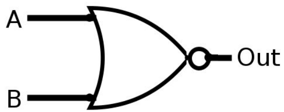
| A | Out |
|---|---|
| 0 | 1 |
| 1 | 0 |

A diagram

An algebraic expression

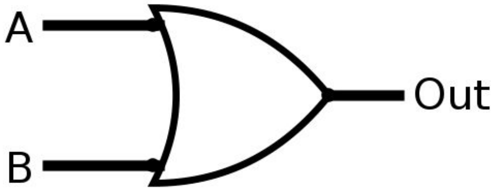
A truth table

# OR




NOR gate looks like this.

- Two 1-bit inputs, labeled A and B
- One 1-bit output, labeled Out
- Can be represented as:




A diagram

$$Out = A + B$$

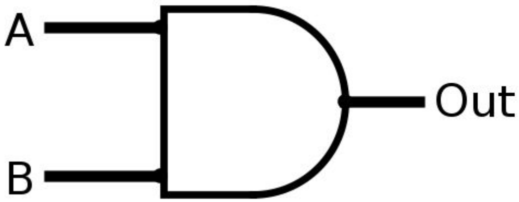
An algebraic expression

| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

A truth table

# AND

- Two 1-bit inputs, labeled A and B
- One 1-bit output, labeled Out
- Can be represented as:




A diagram

Out = AB

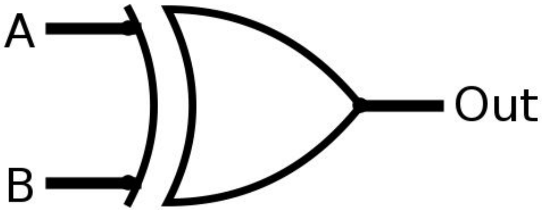
An algebraic expression

| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

A truth table

# XOR

- Two 1-bit inputs, labeled A and B
- One 1-bit output, labeled Out
- Can be represented as:




A diagram

Out = A ⊕ B

An algebraic expression

| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

A truth table

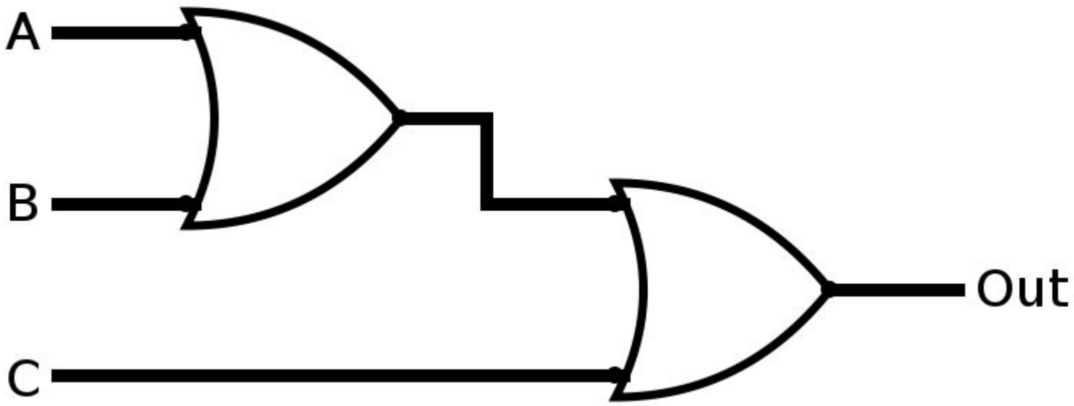
# Combinational Logic

# Combinational Logic

- We can combine logic gates to make larger circuits
  - One or more inputs
  - One or more outputs
  - Perform more complicated logic operations
- These circuits can also be represented as
  - A block in a circuit diagram
  - A truth table, listing the output for every possible input
  - A Boolean algebra expression

# Mystery Circuit #1

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out



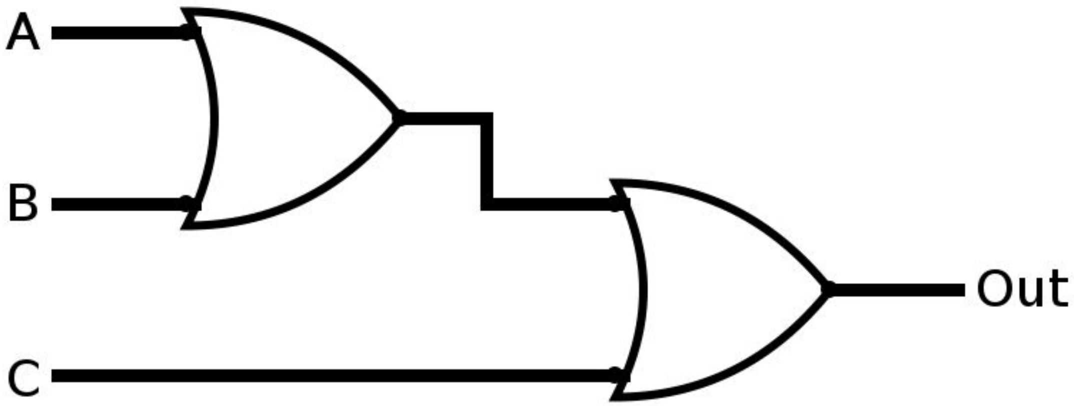

Let’s try to write the truth table from the circuit diagram.

How many rows are in the truth table?
(Hint: How many unique inputs are there?)

# Mystery Circuit #1

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out

| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | |
| 0 | 0 | 1 | |
| 0 | 1 | 0 | |
| 0 | 1 | 1 | |
| 1 | 0 | 0 | |
| 1 | 0 | 1 | |
| 1 | 1 | 0 | |
| 1 | 1 | 1 | |

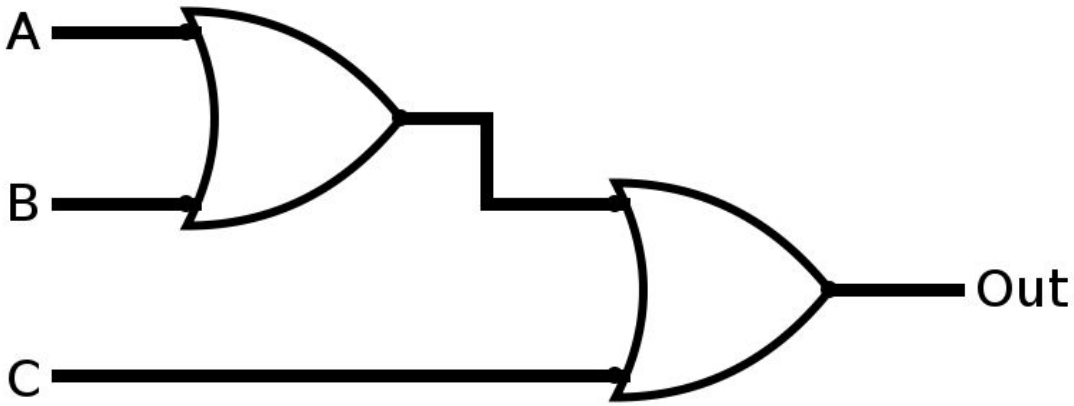



Try inputs on the circuit to fill out the truth table!

# Mystery Circuit #1

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out

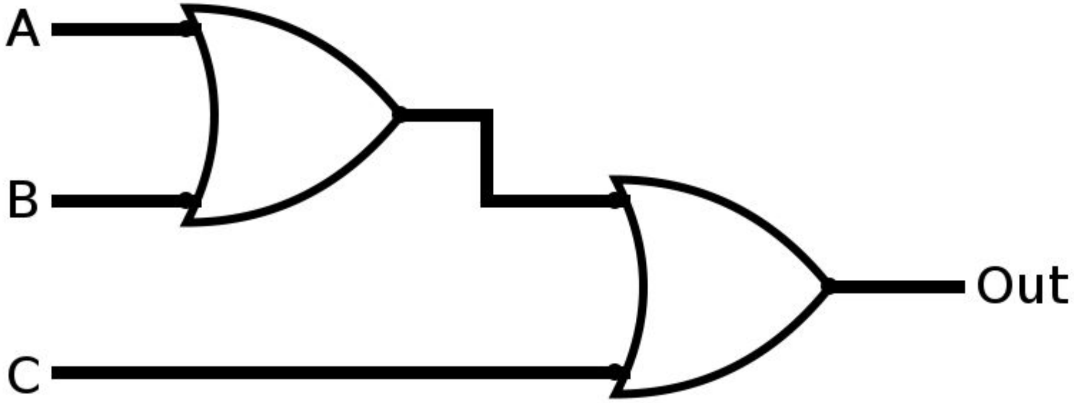
| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |




Let's write the algebraic expression from the circuit.

# Mystery Circuit #1

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out

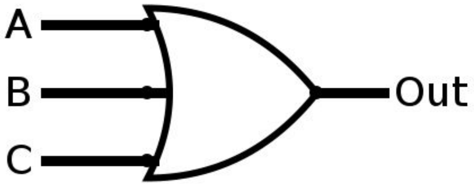


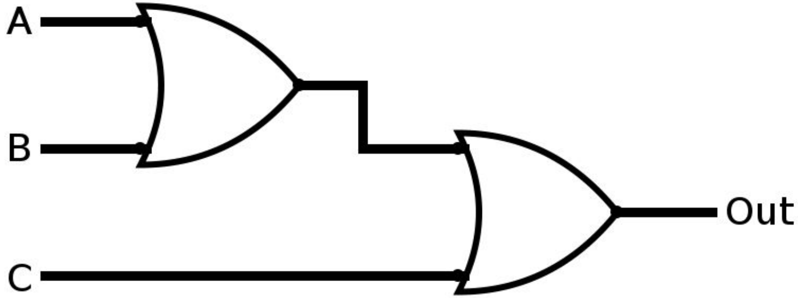

| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Out = (A + B) + C

# Mystery Circuit #1: 3-way OR

- Intuitively: Outputs 1 when at least one of the inputs is 1
- Sometimes drawn like this:

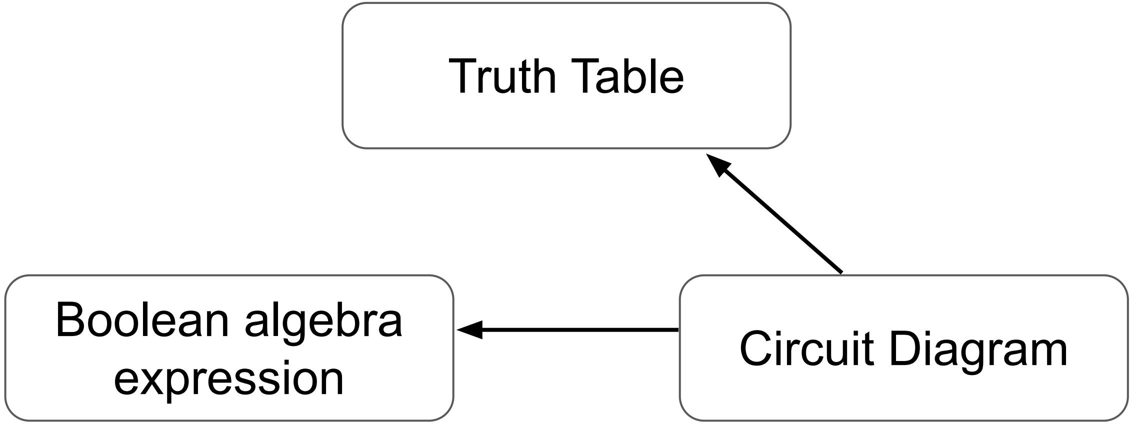



| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Out = (A + B) + C

# Converting Between Representations




We converted a circuit diagram to a truth tables and algebraic expression.

Can we convert between the other representations too?

# Mystery Circuit #2

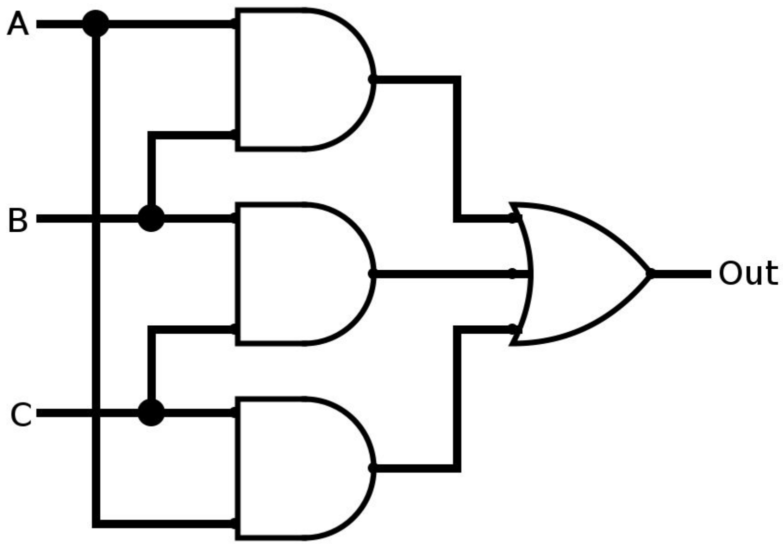
- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out

Given the algebraic expression, let's draw the circuit diagram.

Out = AB + BC + AC

# Mystery Circuit #2

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out




Out = AB + BC + AC

# Mystery Circuit #2

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out

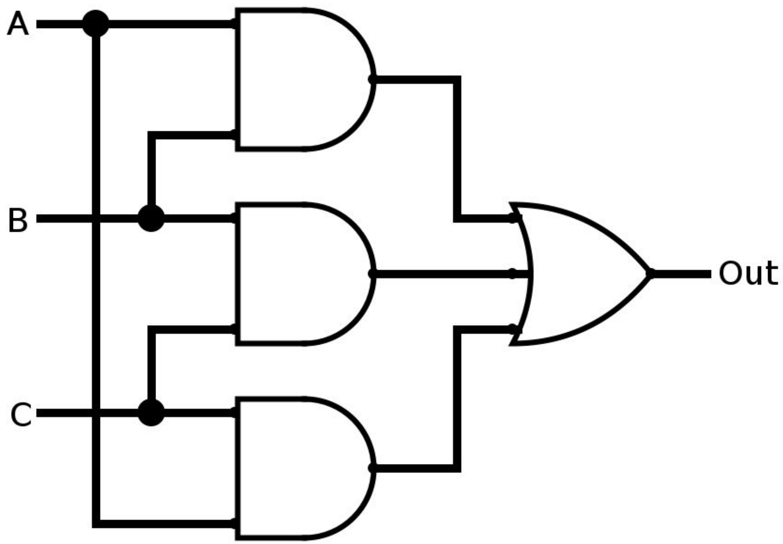
| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | |
| 0 | 0 | 1 | |
| 0 | 1 | 0 | |
| 0 | 1 | 1 | |
| 1 | 0 | 0 | |
| 1 | 0 | 1 | |
| 1 | 1 | 0 | |
| 1 | 1 | 1 | |

Given the algebraic expression, let's write the truth table.

$$Out = AB + BC + AC$$

# Mystery Circuit #2

- Three 1-bit inputs, labeled A, B, C
- One 1-bit output, labeled Out

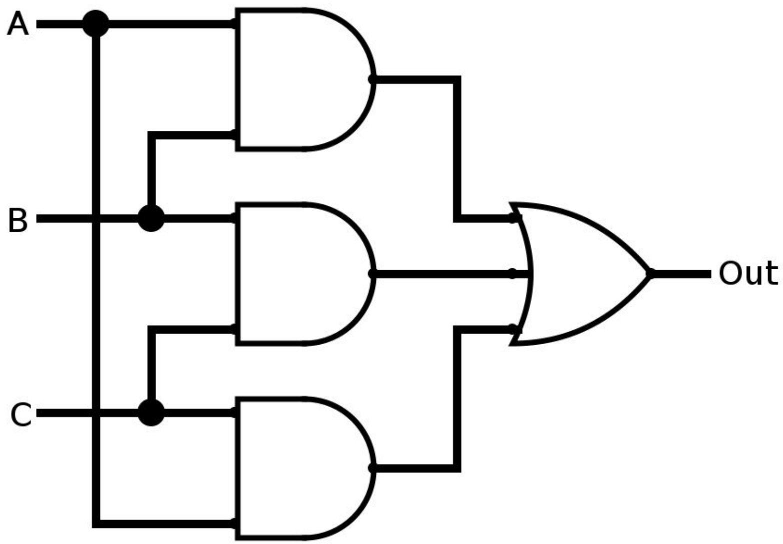



| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Out = AB + BC + AC

# Mystery Circuit #2: Majority Circuit

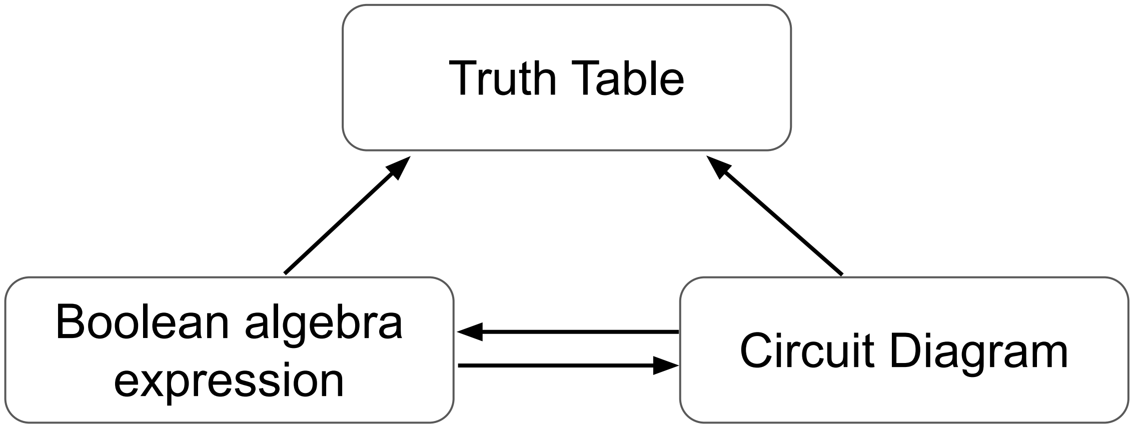
- Intuitively: Outputs the most common value (0 or 1) among the three inputs




| A | B | C | Out |
|---|---|---|---|
| 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 |

Out = AB + BC + AC

# Converting Between Representations



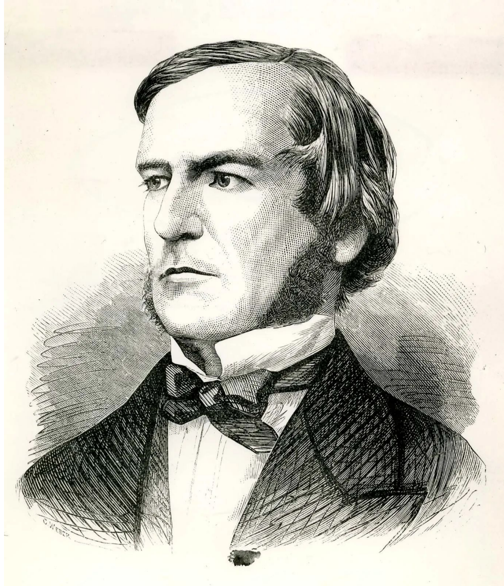

We just converted algebraic expressions to truth tables and circuit diagrams.

Before we see the next conversion, let's talk a bit more about Boolean algebra.

# Boolean Algebra

# History of Boolean Algebra

- Early computer designers built ad hoc circuits from switches
- Began to notice common patterns in their work: ANDs, ORs, …
- Master’s thesis (by Claude Shannon, 1940) made link between work and 19th Century mathematician George Boole
- Called it “Boolean Algebra” in his honor



# Boolean Algebra Operations

- Recall the Boolean algebra operations from earlier
  - We saw these operations in C and RISC-V
  - We can compute these operations with logic gates

| **Operation** | **Notation** |
|---|---|
| A and B | $AB$ |
| A or B | $A + B$ |
| not A | $\neg A$ |
| A xor B | $A \oplus B$ |

Note: There are a variety of accepted symbols used for these operations.

# Simplifying Boolean Algebra

- Given a complicated Boolean algebra expression, we can use some rules to simplify them
  - Useful way to simplify circuits: convert to Boolean algebra, simplify, and convert back
  - Simplifying circuits = less hardware

# Laws of Boolean Algebra

| | | |
|---|---|---|
| $(x)(\neg x) = 0$ | $x + \neg x = 1$ | Complementarity |
| $(x)(0) = 0$ | $x + 1 = 1$ | Laws of 0s and 1s |
| $(x)(1) = x$ | $x + 0 = x$ | Identities |
| $(x)(x) = x$ | $x + x = x$ | Idempotent law |
| $xy = yx$ | $x + y = y + x$ | Commutative law |
| $(xy)z = x(yz)$ | $(x + y) + z = x + (y + z)$ | Associativity |
| $x(y + z) = xy + xz$ | $x + yz = (x + y)(x + z)$ | Distribution |
| $xy + x = x$ | $(x + y)x = x$ | Uniting theorem |
| $\neg(xy) = \neg x + \neg y$ | $\neg(x + y) = (\neg x)(\neg y)$ | DeMorgan’s Law |

# Boolean Algebra Simplification Example

| | | |
|---|---|---|
| $y$ | $= ab + a + c$ | |
| | $= ab + a(1) + c$ | Identity |
| | $= a(b+1) + c$ | Distribution |
| | $= a(1) + c$ | Law of 1s |
| | $= a + c$ | Identity |

# Sum-of-Products

- Given a truth table, how can we convert it to a Boolean algebra expression?
- Idea: Look for every row where the output is 1
  - For Out=1, we either have:
  - (A=0 and B=0 and C=1), or
  - (A=1 and B=0 and C=1), or
  - (A=1 and B=1 and C=1)
- In Boolean algebra: $(\neg A)(\neg B)(C) + (A)(\neg B)(C) + (A)(B)(C)$
- This is called the sum-of-products form
  - For each row where Out=1, create one product (AND the inputs together)
  - Then take the sum of all the products (OR the products together)

| A | B | C | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

# Mystery Circuit #3

- Three 1-bit inputs, labeled A, B, S
- One 1-bit output, labeled Out

Given the truth table, let's write the algebraic expression.

Strategy: Start with sum-of-products form, then simplify with Boolean algebra laws.

| A | B | S | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

# Mystery Circuit #3

- Three 1-bit inputs, labeled A, B, S
- One 1-bit output, labeled Out

| A | B | S | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

$$\text{Out} = (a)(\neg b)(\neg s) + (a)(b)(\neg s) + (\neg a)(b)(s) + abs$$

$$= (\ (a)(\neg b) + ab\ )(\neg s) + (\ (\neg a)(b) + ab\ )s$$

$$= (\ (a)(\neg b + b)\ )(\neg s) + (\ (\neg a + a)(b)\ )s$$

$$= (a(1))(\neg s) + ((1)b)s$$

$$= a(\neg s) + bs$$

# Mystery Circuit #3

- Three 1-bit inputs, labeled A, B, S
- One 1-bit output, labeled Out

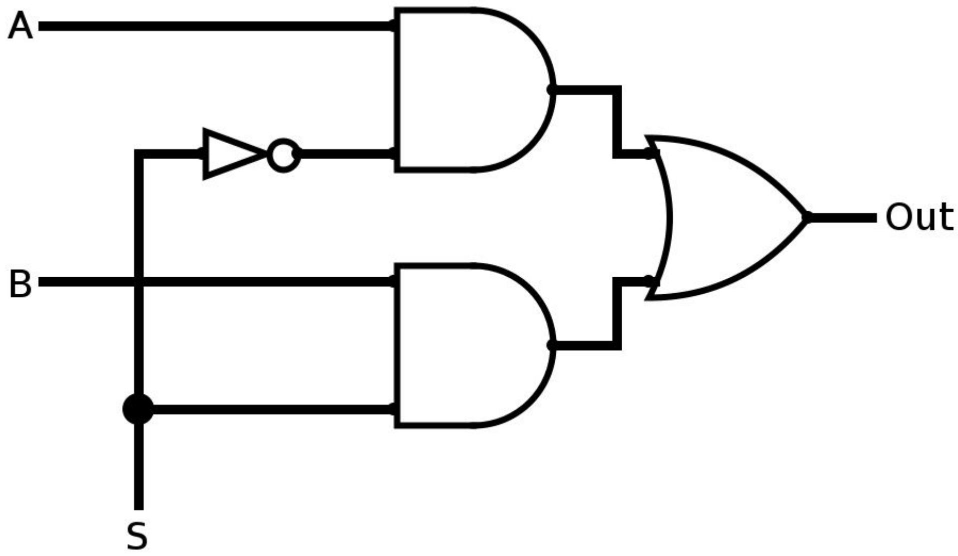
Given the simplified algebraic expression, we can draw a circuit diagram with fewer logic gates (compared to drawing a diagram from sum-of-products form).

| A | B | S | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

$$\text{Out} = (A)(\neg S) + (B)(S)$$

# Mystery Circuit #3

- Three 1-bit inputs, labeled A, B, S
- One 1-bit output, labeled Out

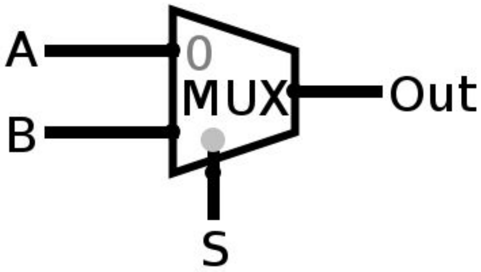



| A | B | S | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

$$Out = (A)(\neg S) + (B)(S)$$

# Mystery Circuit #3: Multiplexer (MUX)

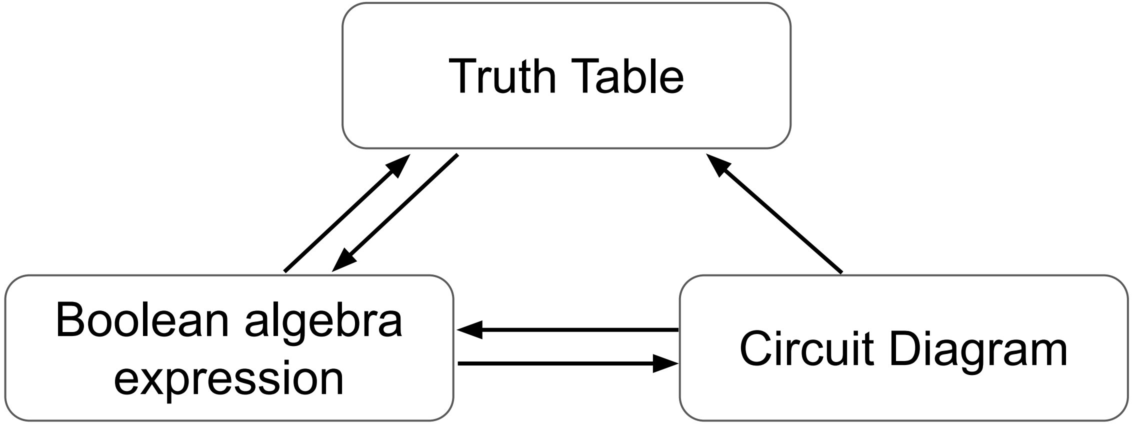
- Intuitively:
  - If S is 0, output the value in A.
  - If S is 1, output the value in B.
  - S is the “select” bit
- Usually drawn like this:




| A | B | S | Out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

Out = (A)(¬S) + (B)(S)

# Converting Between Representations




Now, we can convert truth tables to algebraic expressions.

# Adders

# 32-bit Adder

- Inputs:
  - 32-bit input A
  - 32-bit input B
- Outputs:
  - 32-bit output Sum
- How many rows in the truth table?
  - $2^{32}$ possible inputs for each of A and B
  - $2^{64}$ possible inputs in total
- Can we build this more efficiently?
  - Yes - connect smaller building blocks together!

# 1-bit Adder

How do we add binary numbers by hand?

| | | | | |
|---|---|---|---|---|
| | 0 | 0 | 1 | 0 |
| + | 0 | 1 | 1 | 1 |

# 1-bit Adder

Consider the inputs and outputs in one column:

| | | | | |
|---|---|---|---|---|
| | (1) | (1) | (0) | |
| | 0 | 0 | 1 | 0 |
| + | 0 | 1 | 1 | 1 |
| | 1 | 0 | 0 | 1 |

Inputs:

- 0, 1: Bits being added
- (1): Carry-in bit

Outputs:

- 0: Sum
- (1): Carry-out bit

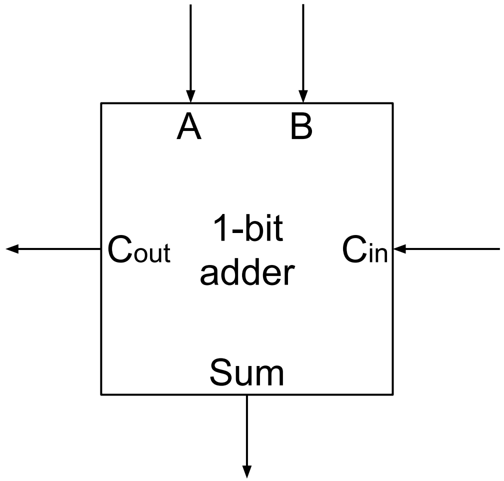
# 1-bit Adder

- Three 1-bit inputs, labeled A, B, $C_{in}$
- Two 1-bit outputs, labeled Sum and $C_{out}$

| A | B | $C_{in}$ | Sum | $C_{out}$ |
|---|---|---|---|---|
| 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 |

# 1-bit Adder

- Three 1-bit inputs, labeled A, B, $C_{in}$
- Two 1-bit outputs, labeled Sum and $C_{out}$



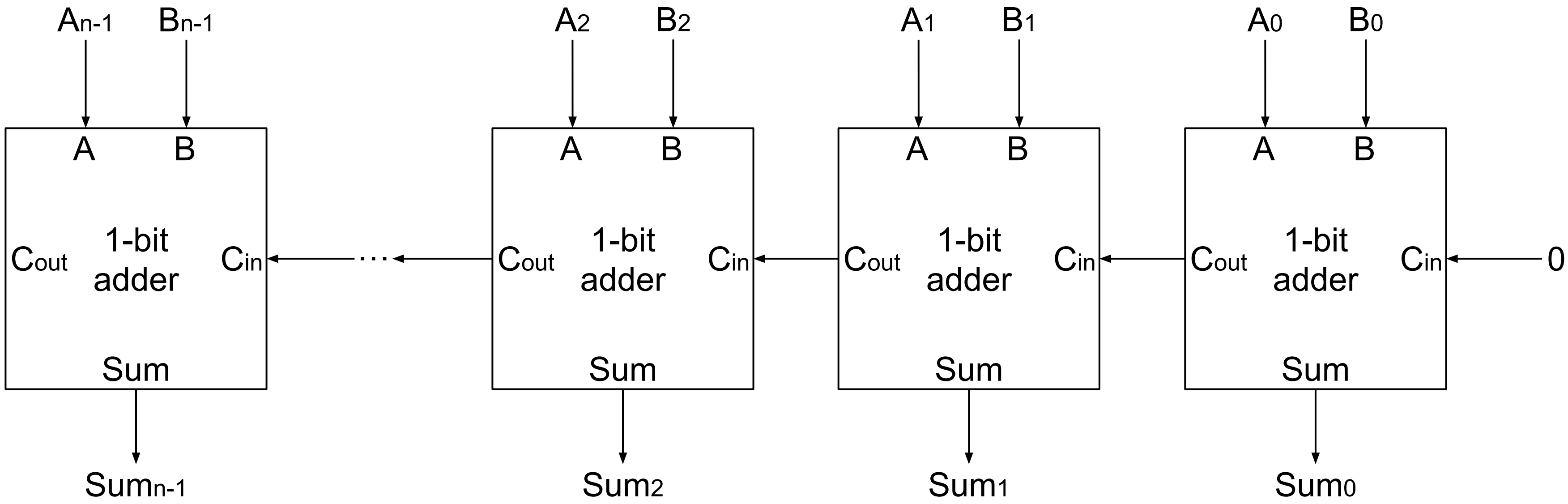

| A | B | $C_{in}$ | Sum | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

$$C_{out} = BC_{in} + AC_{in} + AB$$

$$Sum = A \oplus B \oplus C_{in}$$

# *n*-bit Adder

- Idea: Chain *n* 1-bit adders together to add *n*-bit numbers together



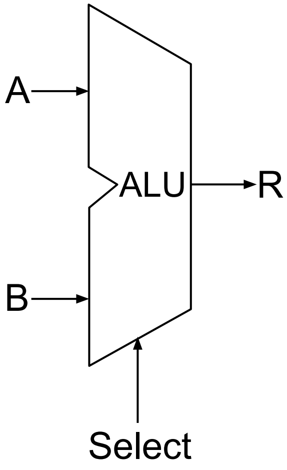
# Arithmetic Logic Unit (ALU)

# Why Build an ALU?

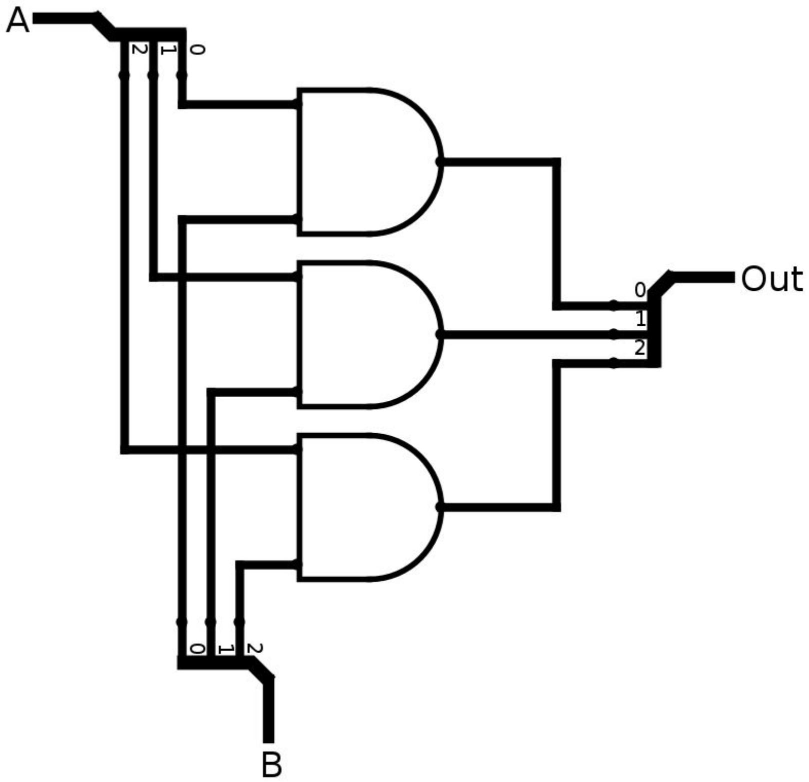
- Starting Wednesday, we'll be building a circuit to execute RISC-V instructions
- In that circuit, we need logic to compute arithmetic operations
  - Examples: Addition, subtraction, AND, OR, XOR, bit-shifting, etc.
- Today, we'll build a simplified circuit that computes four arithmetic operations:
  - Addition, AND, OR, XOR
  - Use a "select" input to choose which operation to perform
- In Project 3A, you'll build a circuit that performs all the arithmetic operations we need

# ALU Design Specifications

- Inputs:
  - 32-bit input A
  - 32-bit input B
  - 2-bit operation selector S
- Outputs:
  - 32-bit result R
- Behavior:
  - If **`S=0b00`**, set **`R=A+B`** (addition)
  - If **`S=0b01`**, set **`R=A&B`** (bitwise AND)
  - If **`S=0b10`**, set **`R=A|B`** (bitwise OR)
  - If **`S=0b11`**, set **`R=A^B`** (bitwise XOR)



# ALU Design: Splitting Bits

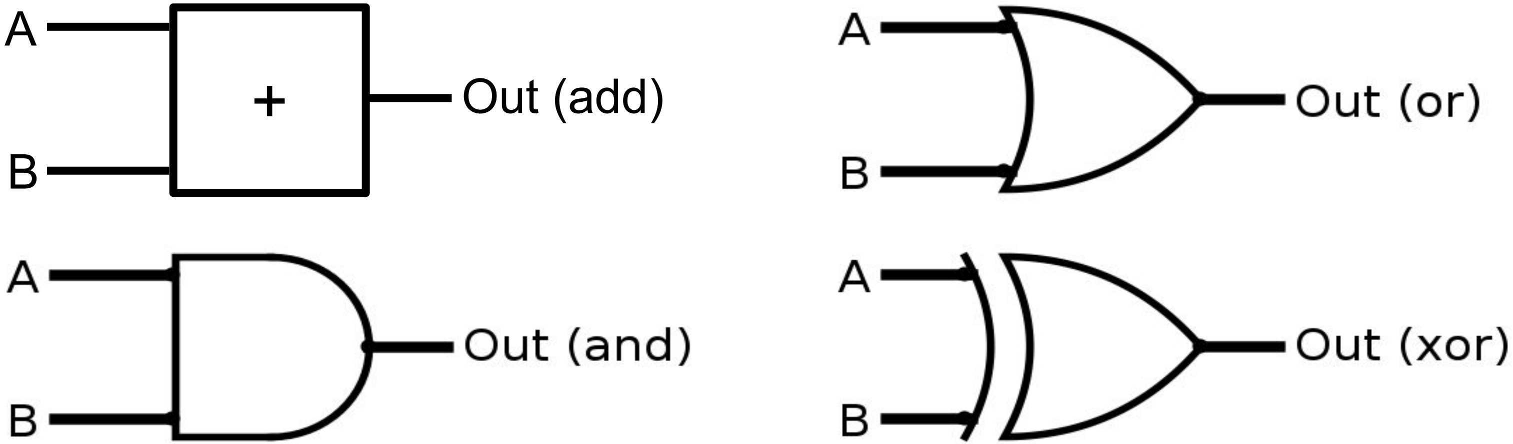



We can split longer inputs into smaller sets of bits to perform operations on.

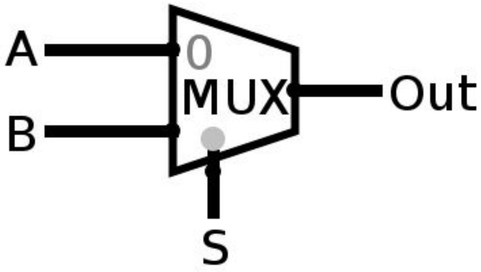
Example: A and B are 3-bit inputs. We split them into individual bits to perform a 3-bit bitwise AND, then recombine the results for the output.

# ALU Design: Arithmetic Operations

- Use logic gates we've seen today to implement the four arithmetic operations
    - 32-bit adder: Chain 32 1-bit adders together
    - 32-bit AND, OR, XOR gates: Split input and perform bitwise operations on each bit



# ALU Design: Selecting Operations

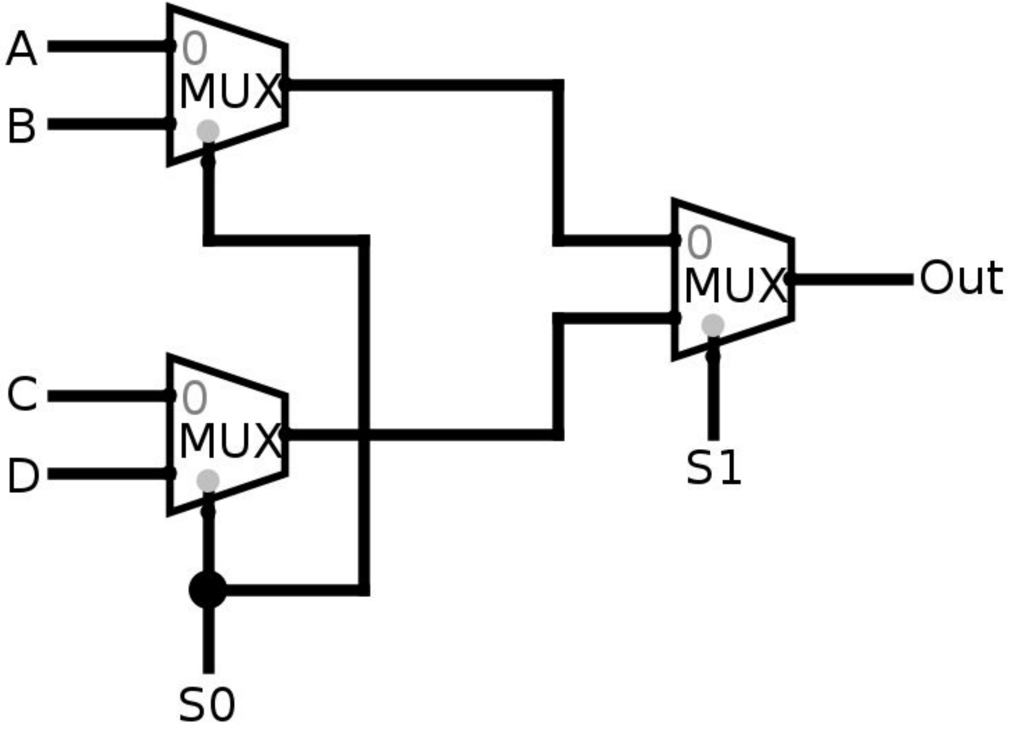



Recall the multiplexer (mux): select one of two inputs, based on a select bit.

How can we make a 4-to-1 mux (select one of four inputs)?

How many select bits do we need for a 4-to-1 mux?

# ALU Design: 4-to-1 mux

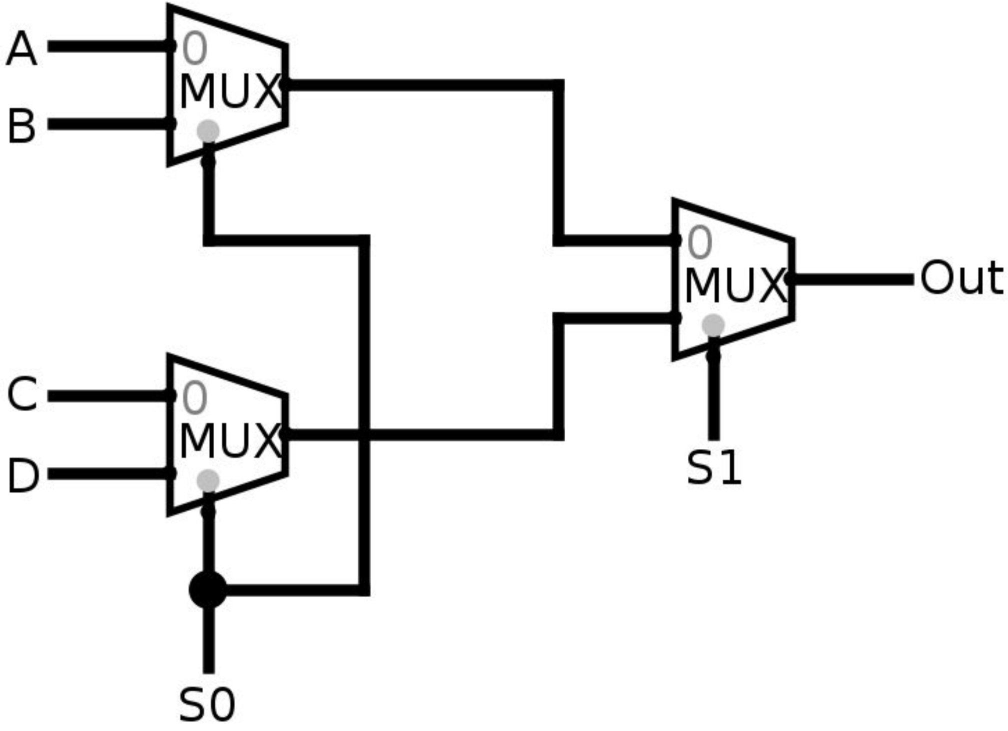



S = 0b00: Choose A
S = 0b01: Choose B
S = 0b10: Choose C
S = 0b11: Choose D

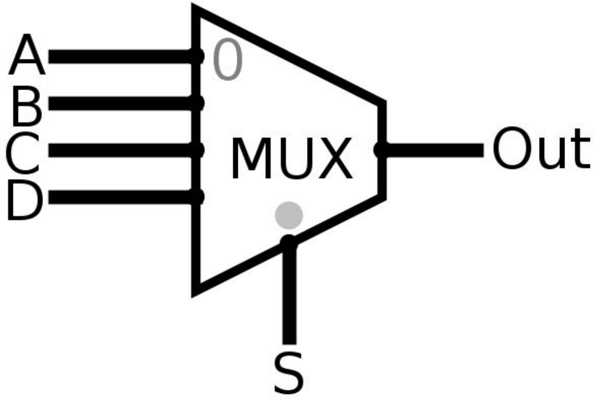
Lower bit of S (S0) chooses A/C or B/D.

Upper bit of S (S1) chooses A/B or C/D.

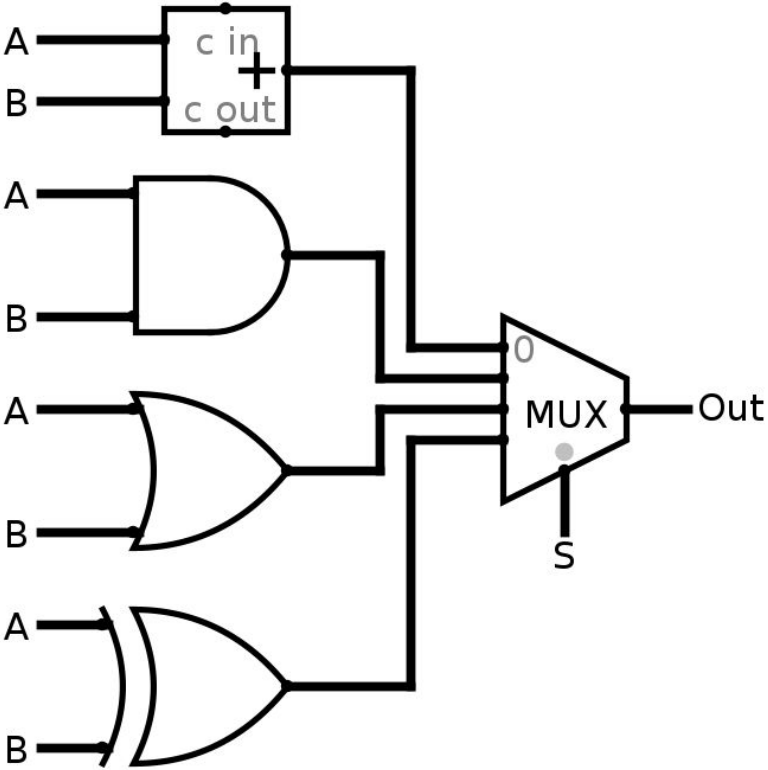
# ALU Design: 4-to-1 mux




Sometimes drawn like this:



# ALU Circuit




Idea: Compute all four operations, and use the mux to select one to output

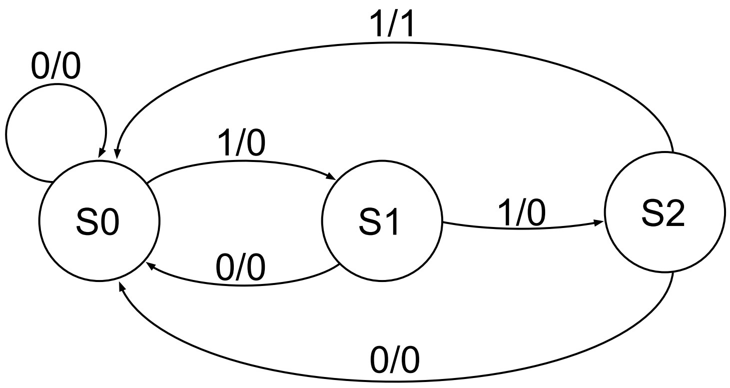
# Finite State Machines

# Finite State Machines (FSMs)

- An FSM consists of:
  - A set of states
  - A transition function: f(current state, input) → next state, output
- To run an FSM, repeat these steps:
  - Receive an input
  - Based on current state and input, move to the next state and generate an output

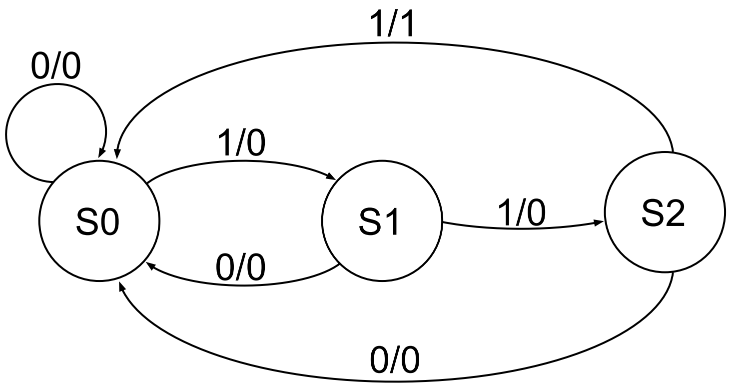
# FSM: State Transition Diagram

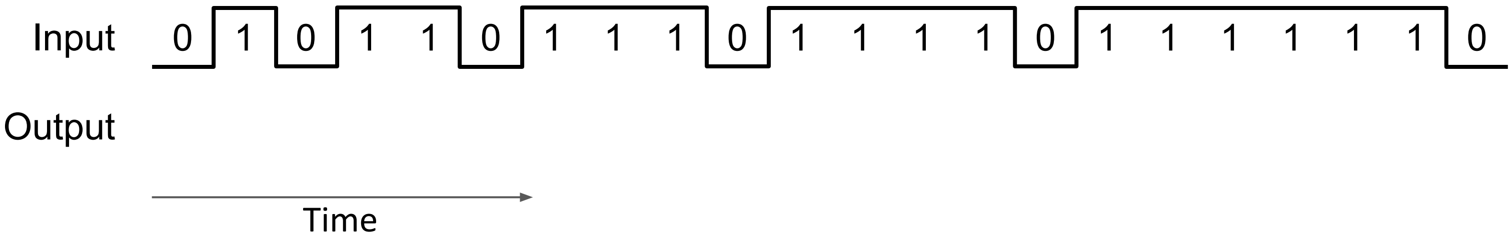
- S0, S1, S2 are the states
- Each arrow represents a transition
  - Arrow from current state to next state
  - Left label on arrow is the input
  - Right label on arrow is the output
- Example of arrow transition:
  - Arrow from S0 to S1, labeled 1/0
  - If you’re at state S0 and receive input 1, then move to state S1, and output 0
- Example of arrow transition:
  - Arrow from S2 to S0, labeled 1/1
  - If you’re at state S2 and receive input 1, then move to state S0, and output 1



# FSM: Waveform

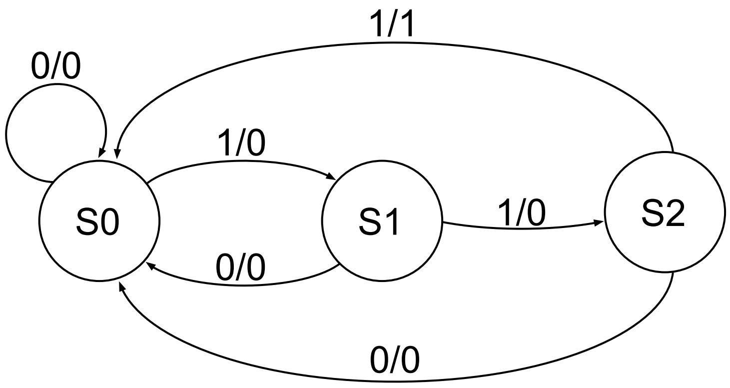
Given this state transition diagram and input signal, what is the output signal?

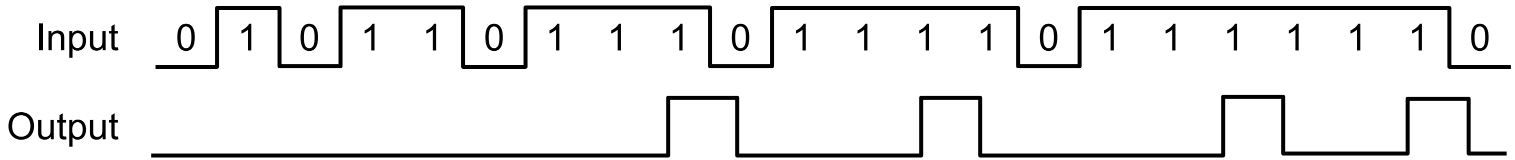


# FSM: Waveform

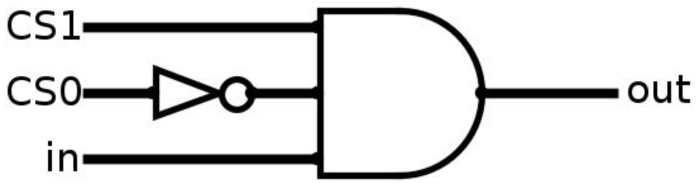
What pattern in the input is this FSM detecting?



# FSM: Transition Function Circuit

- Transition function is combinational logic with:
  - Two inputs: current state (CS) and input (in)
  - Two outputs: next state (NS) and output (out)
  - Each state is labeled with a unique binary number



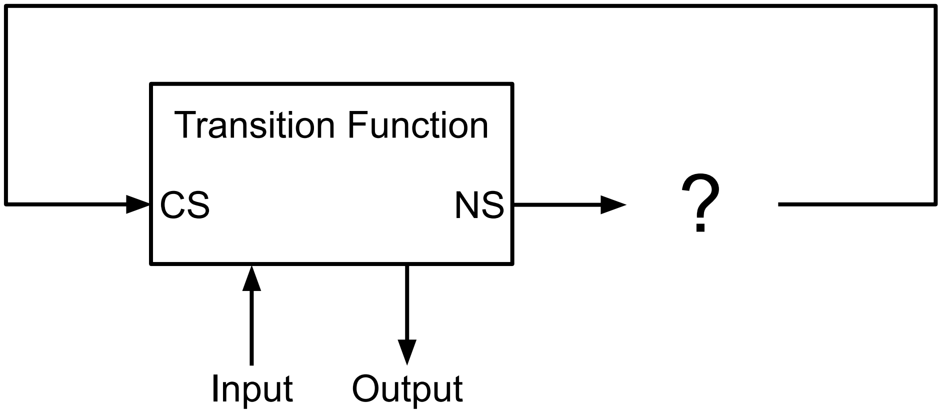

You can also derive a circuit and Boolean algebra expression for the other two outputs, NS0 and NS1.

| 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 |

$$out = (CS1)(\neg CS0)(in)$$

# FSM: State

How do we differentiate between timesteps?
How do we “save” which state we’re in?



# Summary

- Logic gates can be built out of transistors
  - Hardware implementation of bitwise operations
- More complicated circuits can be built out of logic gates
- Logic gates and circuits can be represented with:
  - Circuit diagrams
  - Truth tables
  - Boolean algebra expressions
- Manipulating Boolean algebra expressions
  - Laws of Boolean algebra
  - Sum-of-products representation
- Applications of circuits
  - Arithmetic logic unit (ALU)
  - Finite State Machines (FSMs)
- Next time: How do we store values in circuits?