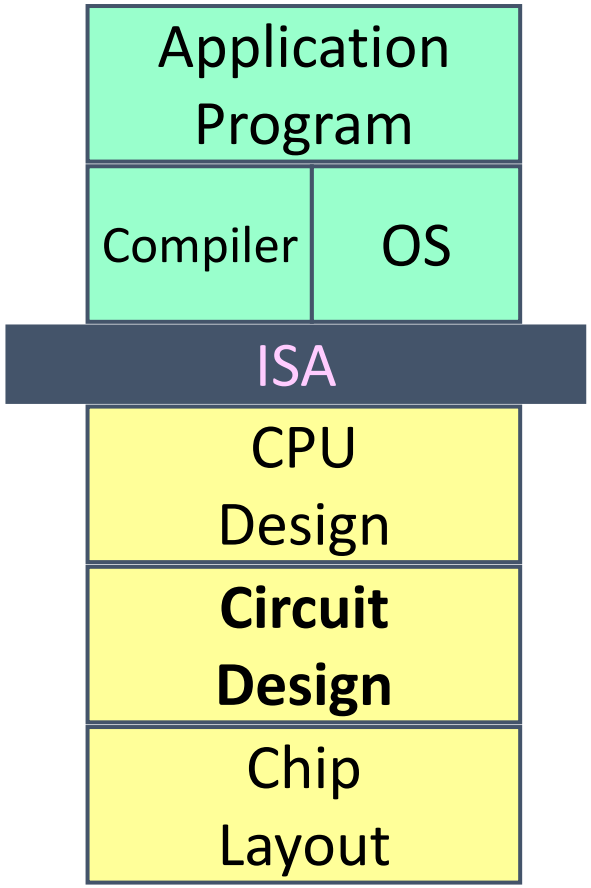
Logic Design

CMPU 224 – Computer Organization

Jason Waterman

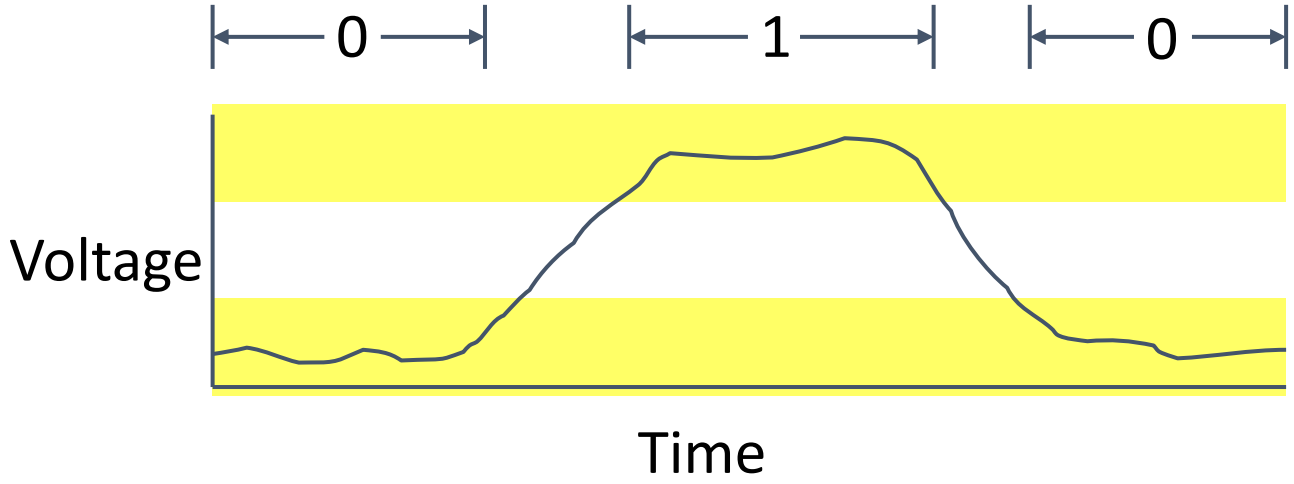
Overview of Logic Design

- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage
- Bits are Our Friends
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information



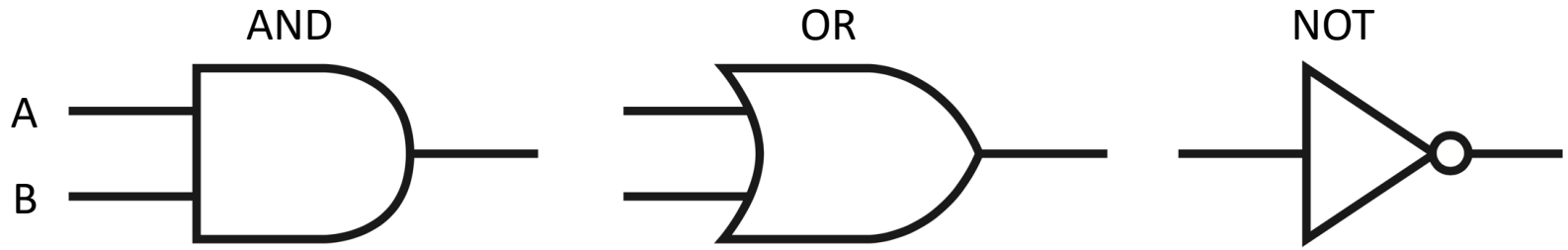
Digital Signals

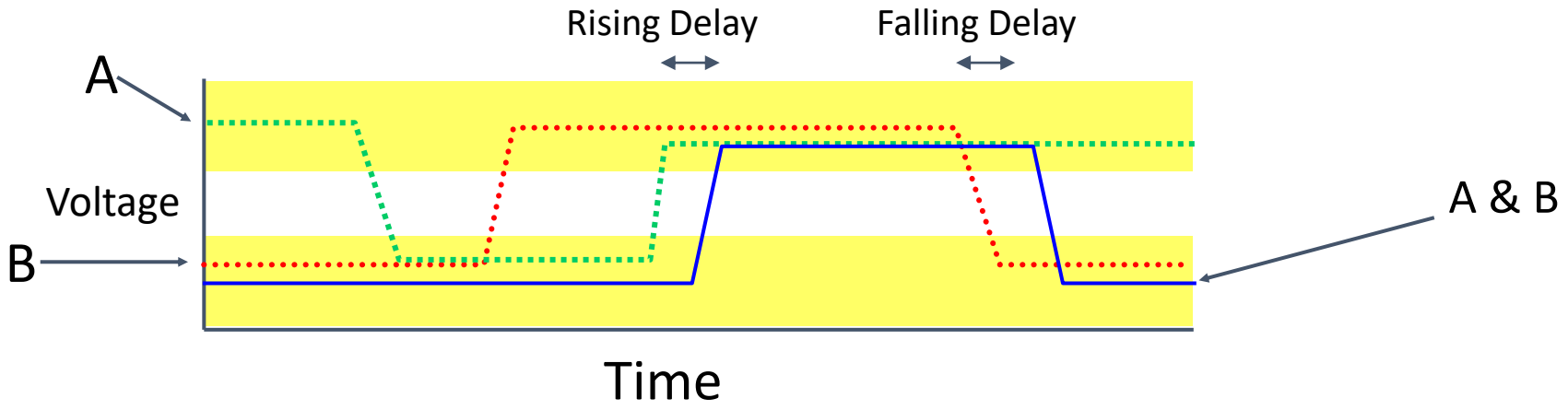
- Use voltage thresholds to extract discrete values from continuous signal
- Simplest version: 1-bit signal
  - Either high range (1) or low range (0)
  - With guard range between them
- Not strongly affected by noise or low-quality circuit elements
  - Can make circuits simple, small, and fast



Computing with Logic Gates

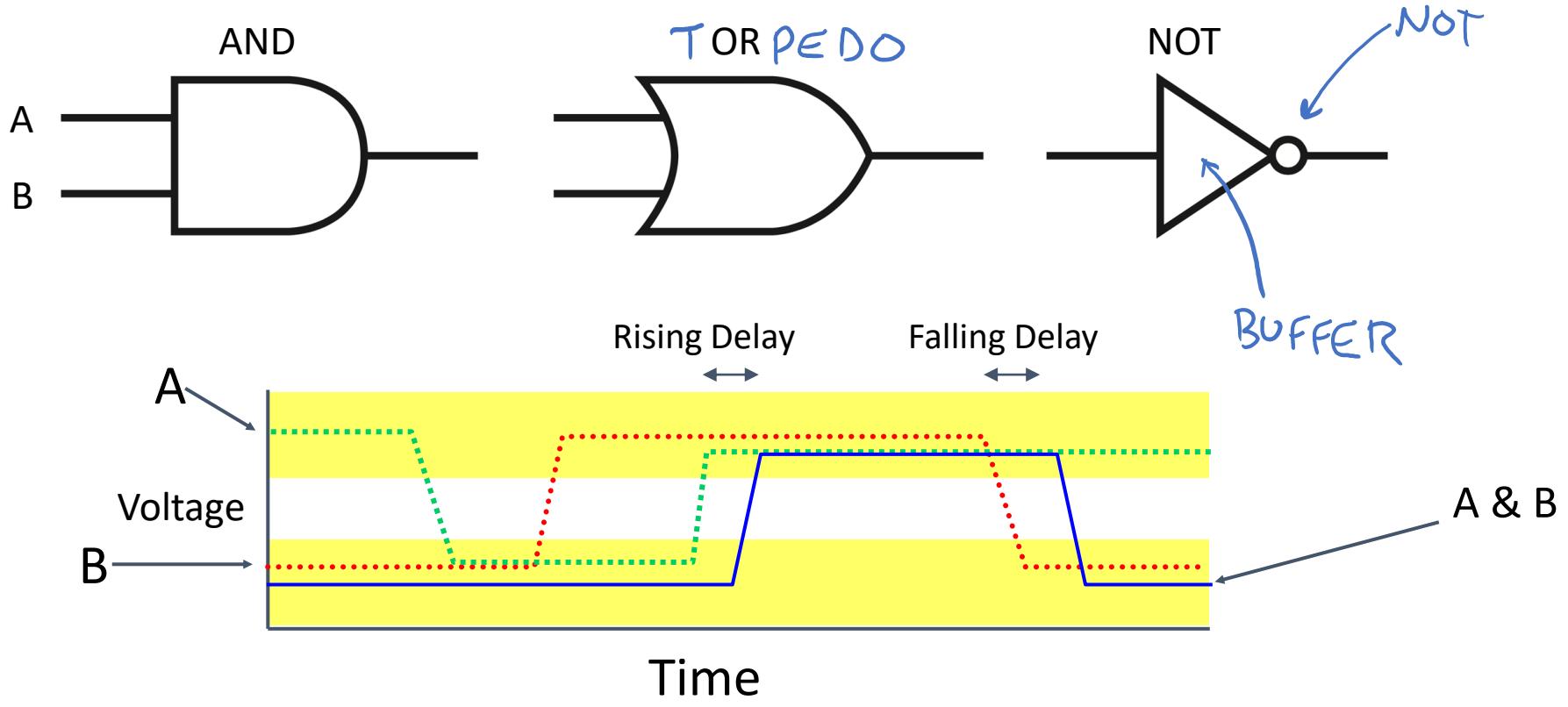
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some small delay





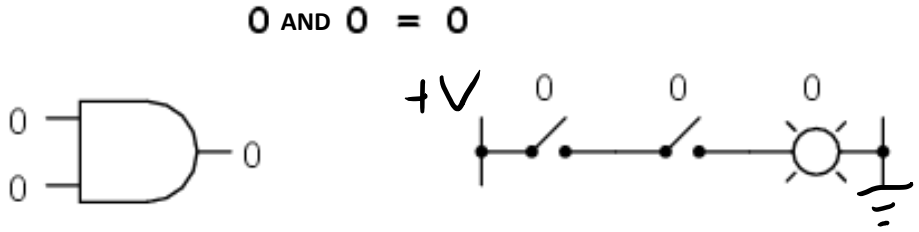
Computing with Logic Gates

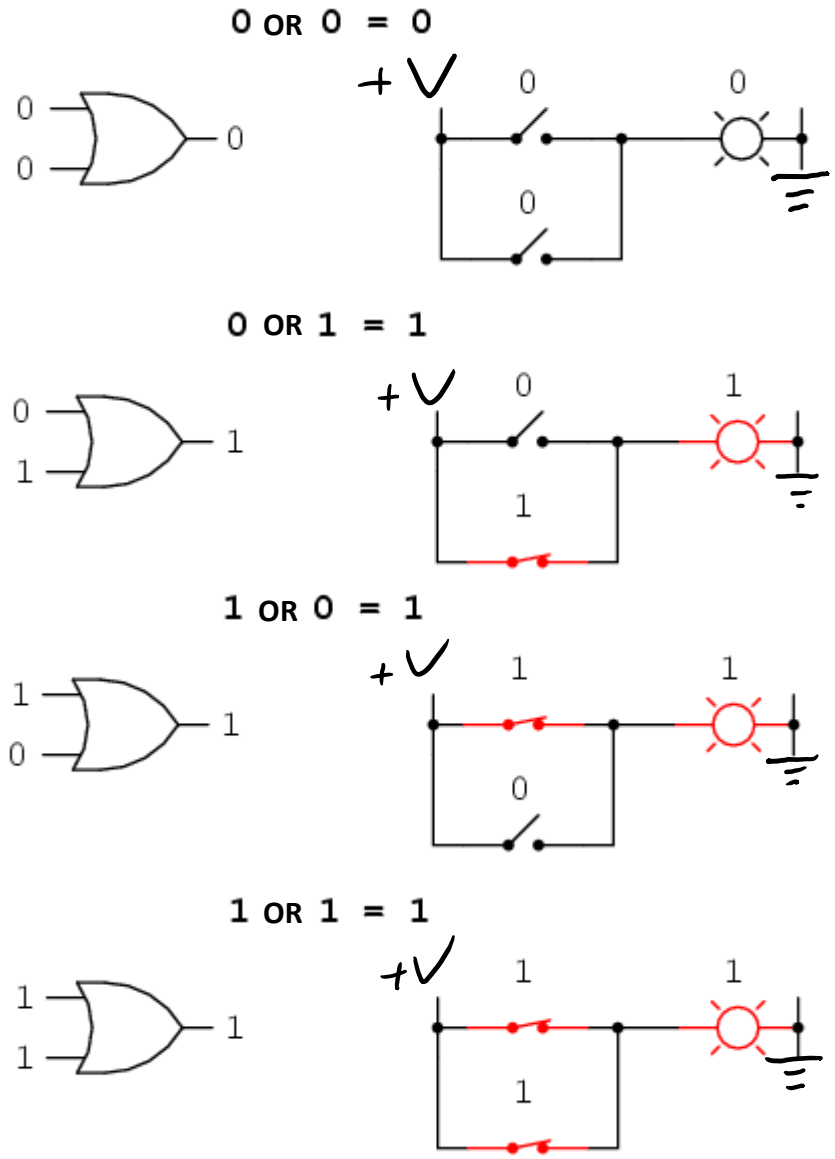
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some small delay



What's a logic gate?

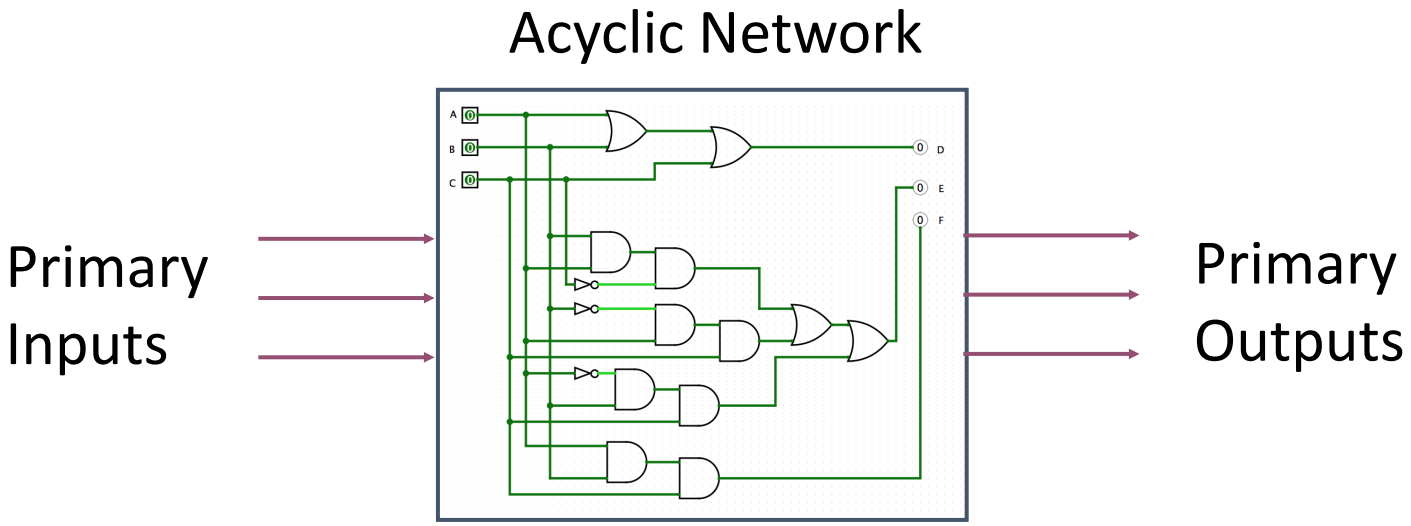
- The are referred to as logic gates because of their ability to "open" and "close" pathways for signals
  - Made up of transistors which act as tiny switches
- OR gate shown on the right, AND gate shown below





Combinational Circuits

- Acyclic Network of Logic Gates
  - Continuously responds to changes on primary inputs
  - Primary outputs become (after some delay) Boolean functions of primary inputs
  - Output is solely dependent on the current input (no memory)



Truth Tables

- Combinational logic blocks can be completely specified by defining the values of the outputs for each possible set of input values
- For a logic block of n inputs there are 2^n entries in the truth table
- Can completely describe any combinational circuit, but it quickly grows in size

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Boolean Algebra

- Often easier to express logic function with logic equations instead of truth table
- Use Boolean algebra
  - All variables have the values 0 or 1
  - Three operators:
    - `OR`, written as `|` (e.g., `A | B`) and its output is 1 if either of the variables is 1
    - `AND`, written as `&` (e.g., `A & B`) and its output is 1 only if both variables are 1
    - `NOT`, a unary operator written as `!` (e.g., `!A`) and its output is 1 only if the input is 0

Logic Equations

- **Any** combinational circuit can be written as a series of Boolean equations
- `OutputVar` = Boolean Equation of `InputVars`
- In the previous truth table:
  - D is one when at least one of the inputs is 1
  - E is one when exactly two inputs are 1
  - F is one when all three inputs are 1
- This truth table can be written as:
  - D = A | B | C
  - E = (A & B & !C) | (A & !B & C) | (!A & B & C)
  - F = A & B & C

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Sum of Products

- **Any** logic function can be implemented with only `AND`, `OR`, and `NOT` functions
- Can be written as a two-level representation as a logical **sum of products**
  - Logical sum: `OR`
    - `0 + 0 = 0`
    - `0 + 1 = 1`
    - `1 + 0 = 1`
    - `1 + 1 = 1`
  - Logical product: `AND`
    - `0 * 0 = 0`
    - `0 * 1 = 0`
    - `1 * 0 = 0`
    - `1 * 1 = 1`

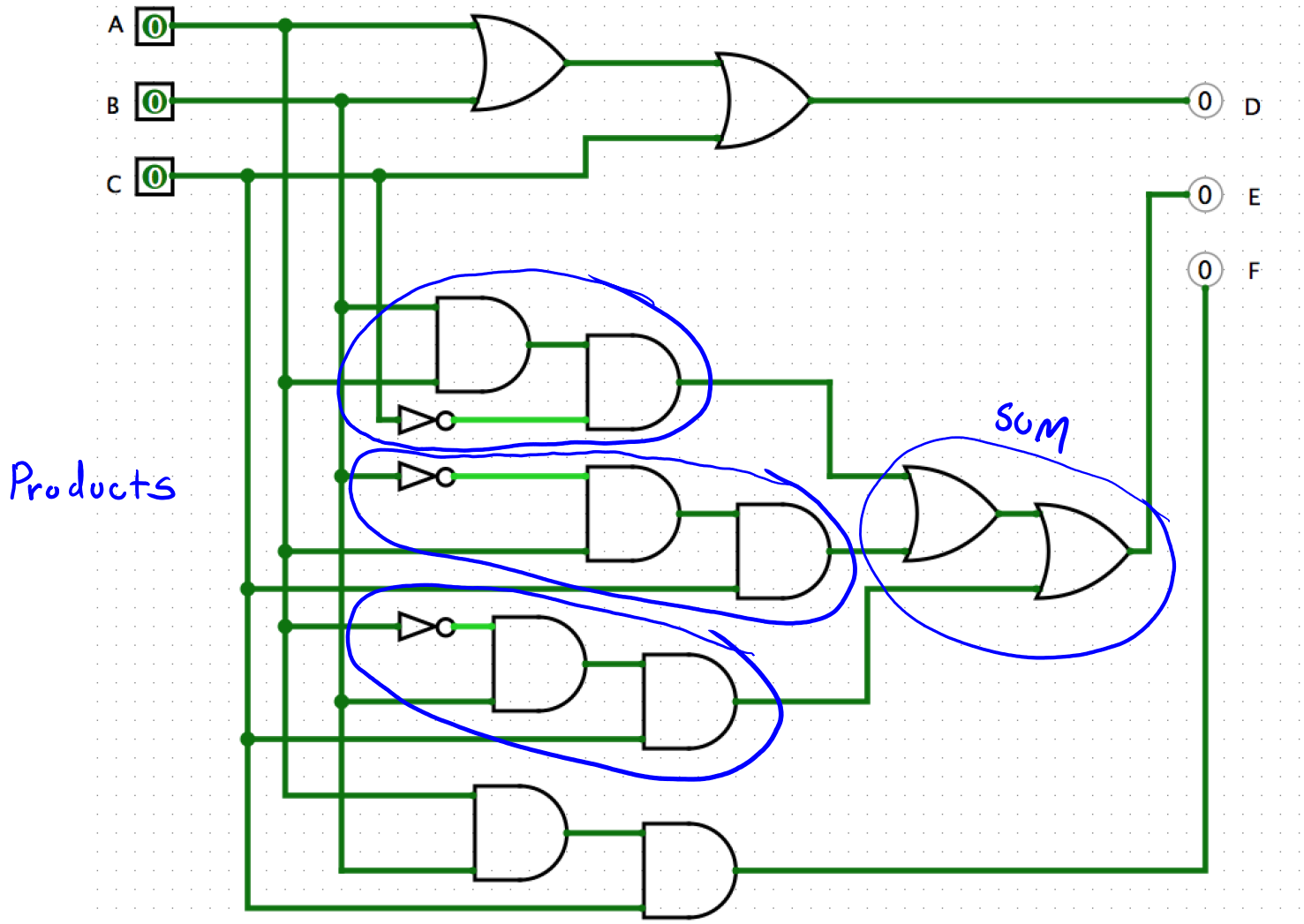
Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- Example:
  - E = `(!A & B & C) | (A & !B & C) | (A & B & !C)`
  - F = `A & B & C`
  - D = `(!A&!B&C) | (!A&B&!C) | (!A&B&C) | (A&!B&!C) | (A&!B&C) | (A&B&!C) | (A&B&C)`

(A & !B) (A & B)

- Sum of products does not always produce the simplest logic circuit that could implement the function!

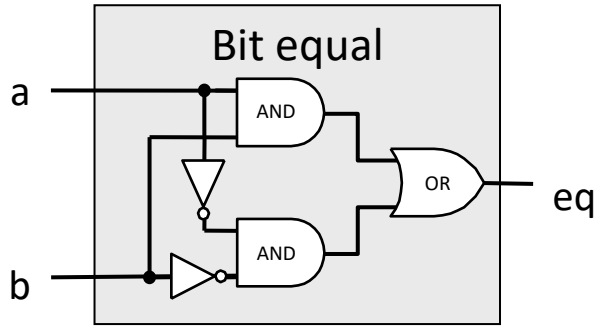
Truth Table Implementation



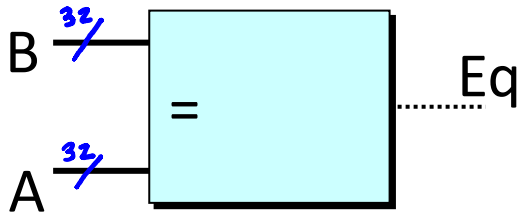
Equality

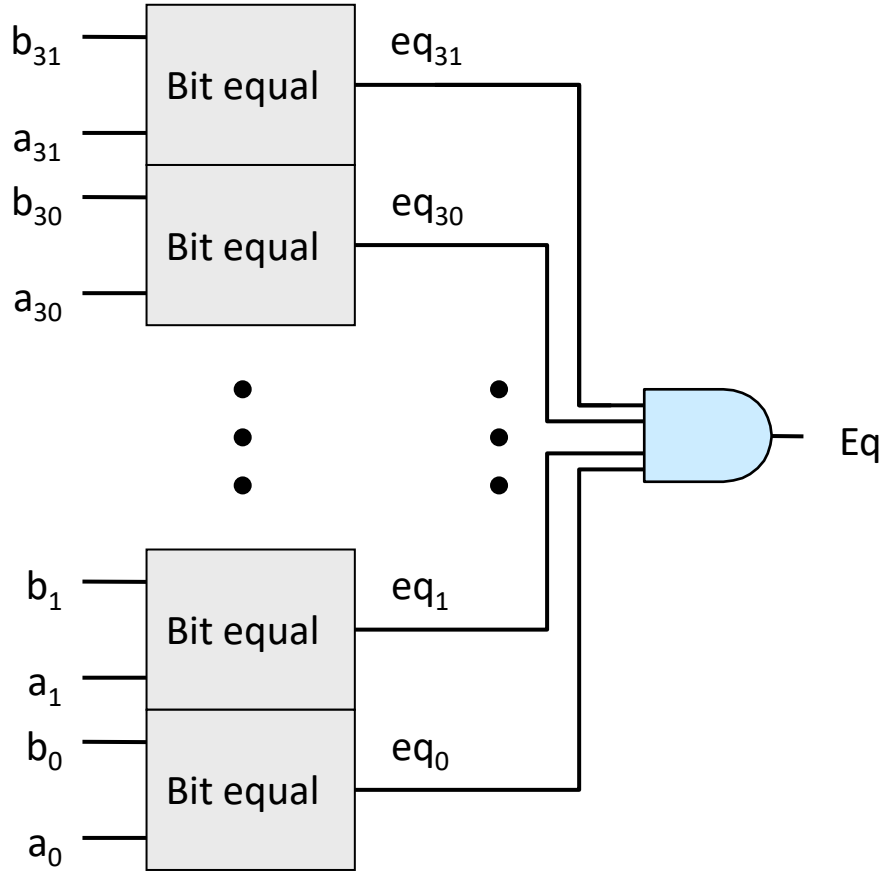
- Generate 1 if a and b are equal

A	B	eq
0	0	1
0	1	0
1	0	0
1	1	1



- Word equality
  - A and B both represent 32-bit words
    - represented by bold line
  - Eq is a 1-bit 'boolean'
    - represented by a dashed line

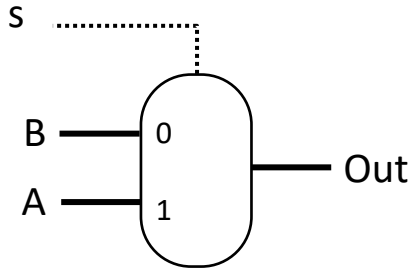


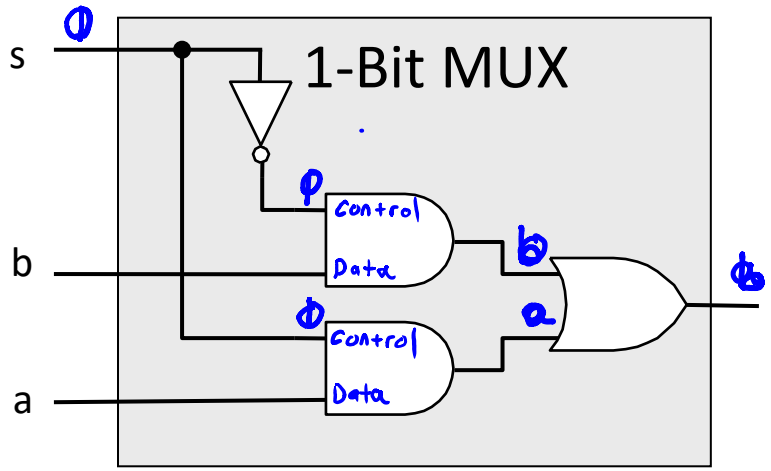


Bit-Level Multiplexor

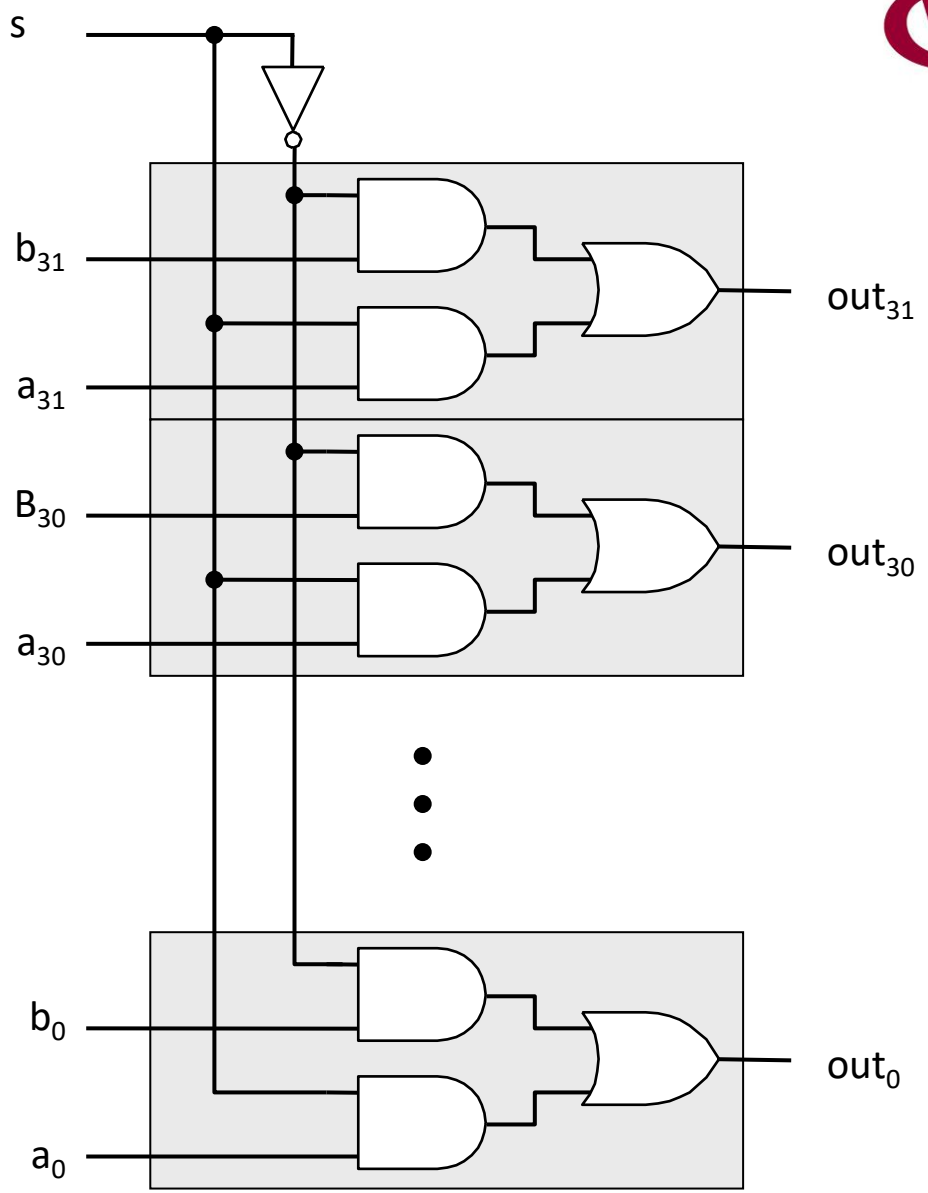
- Control signal `s`
- Data signals `A` and `B`
- Output **`A`** when **`s=1`**, **`B`** when **`s=0`**

Word-Level Representation



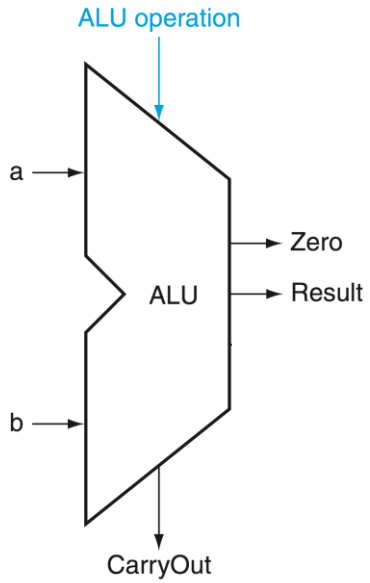


C	D	Out
0	0	0
0	1	0
1	0	0
1	1	1



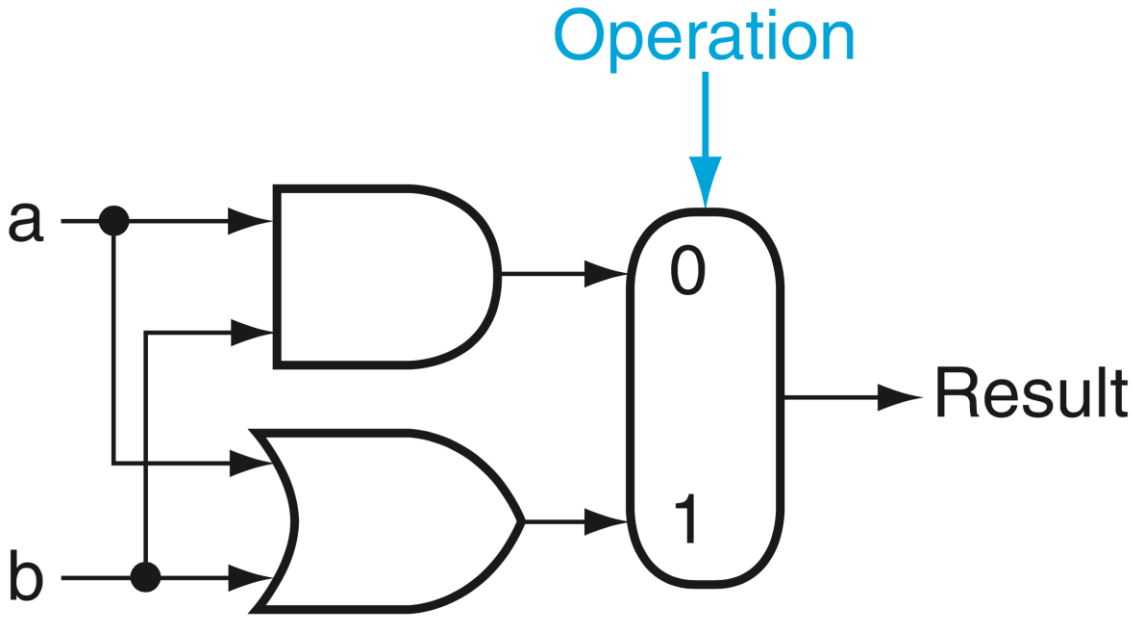
Arithmetic Logic Unit

- Arithmetic Logic Unit (ALU) is the heart of the processor
  - Performs arithmetic operations (addition, subtraction, etc.)
  - Performs logical operations (and, or, not, shifts, etc.)
- We can build a 32-bit ALU with the combinational blocks previously mentioned
- Operations:
  - Bitwise AND
  - Bitwise OR
  - Addition
  - Subtraction
- Flags:
  - Zero
  - Carryout



First Steps: A 1-Bit ALU

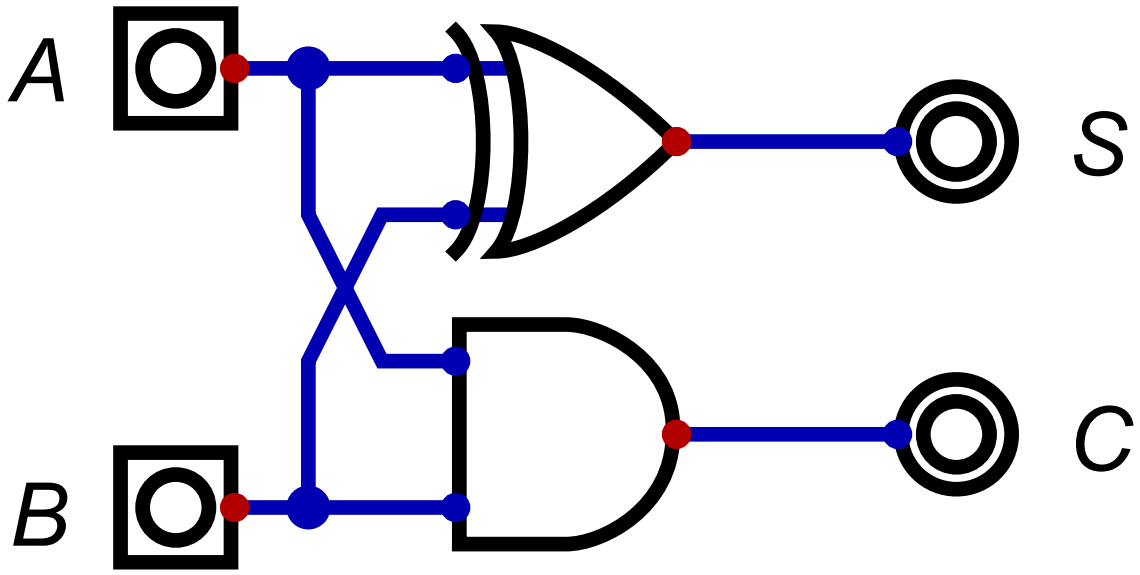
- General idea: Build a 1-bit ALU and duplicate it 32 times
- Start with logical operations which map directly to logic gates
- Simple 1-bit logical unit for `AND` and `OR`
- Operation is an input to a multiplexor which decides the result



1-Bit Half Adder

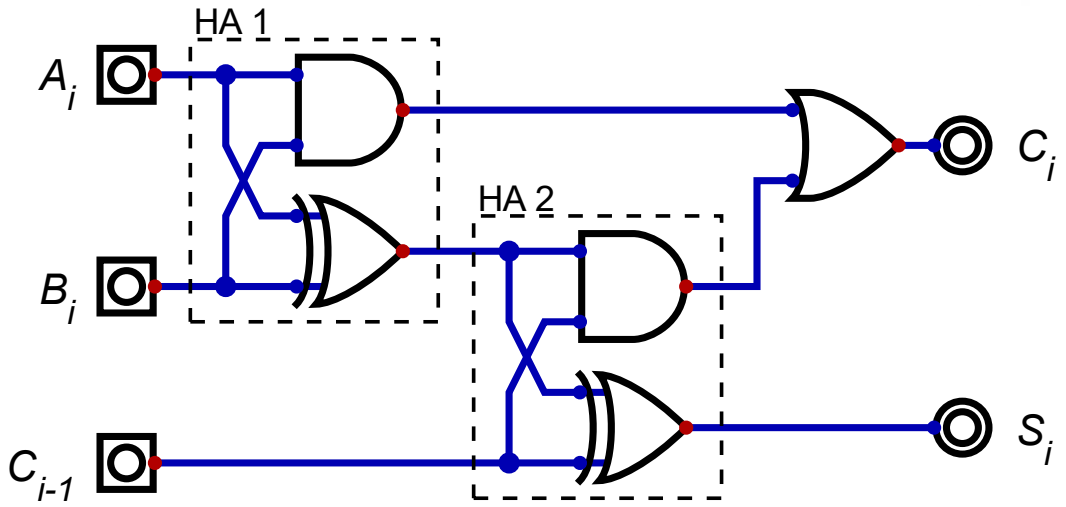
- Like to add two 1-bit numbers A and B
  - Will generate two outputs:
    - Sum (A+B)
    - CarryCout

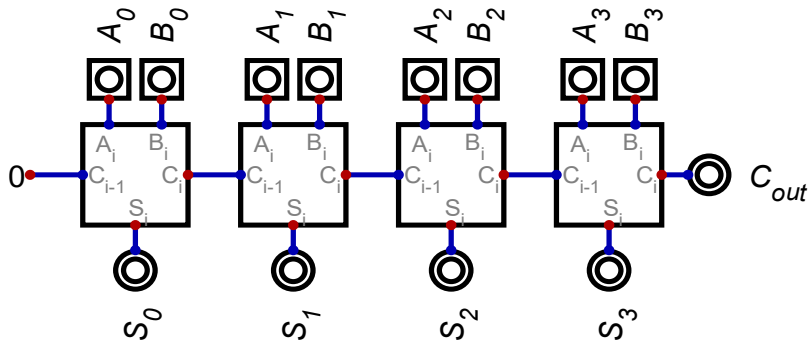
A	B	CarryOut	(A+B) Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



1-Bit Full Adder

- Also called a 3,2 adder because it has three inputs and two outputs
- Can be built from 2 half adders
  - First half adder computes sum of A+B
  - Second half adder sum of C_{in} and (A+B)
- If either addition generates a CarryOut, propagate it forward as C_{out}

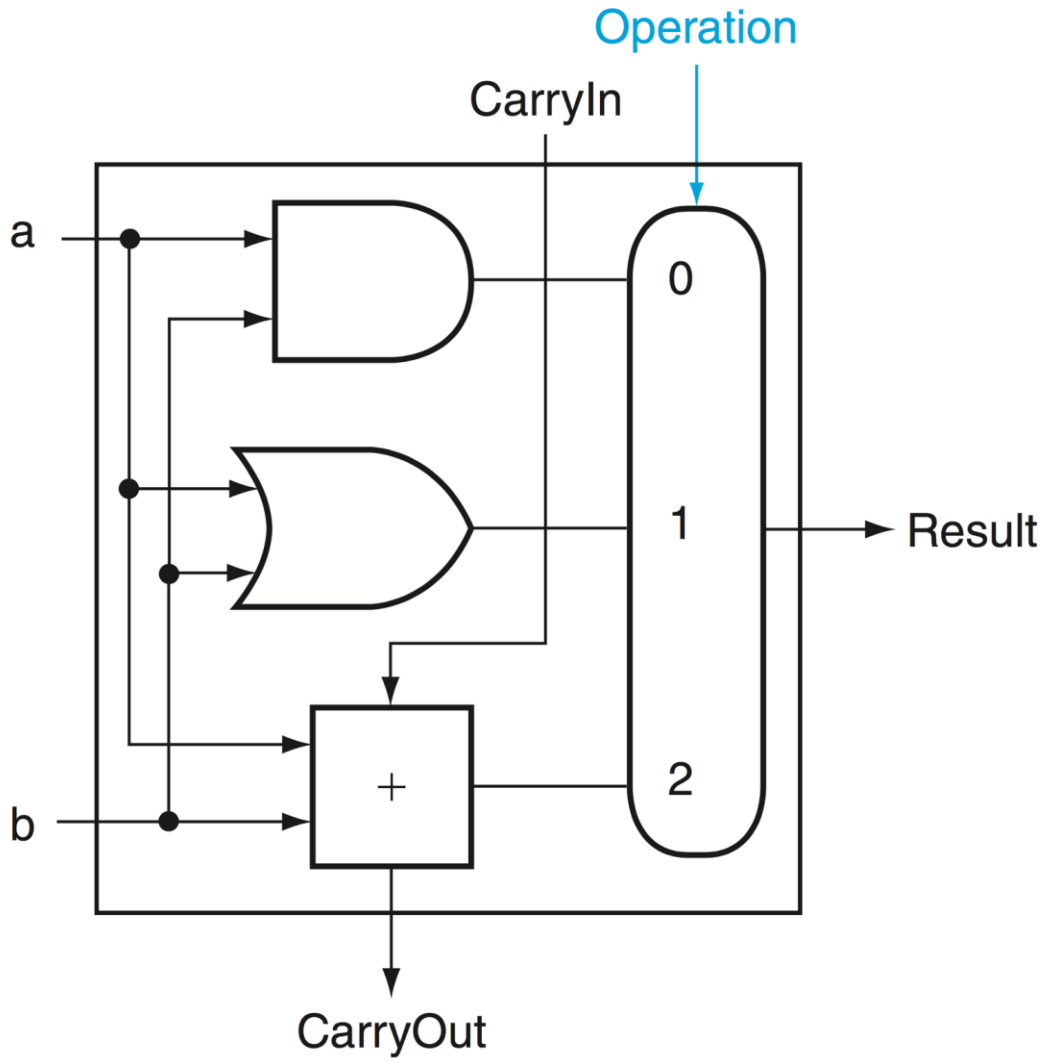




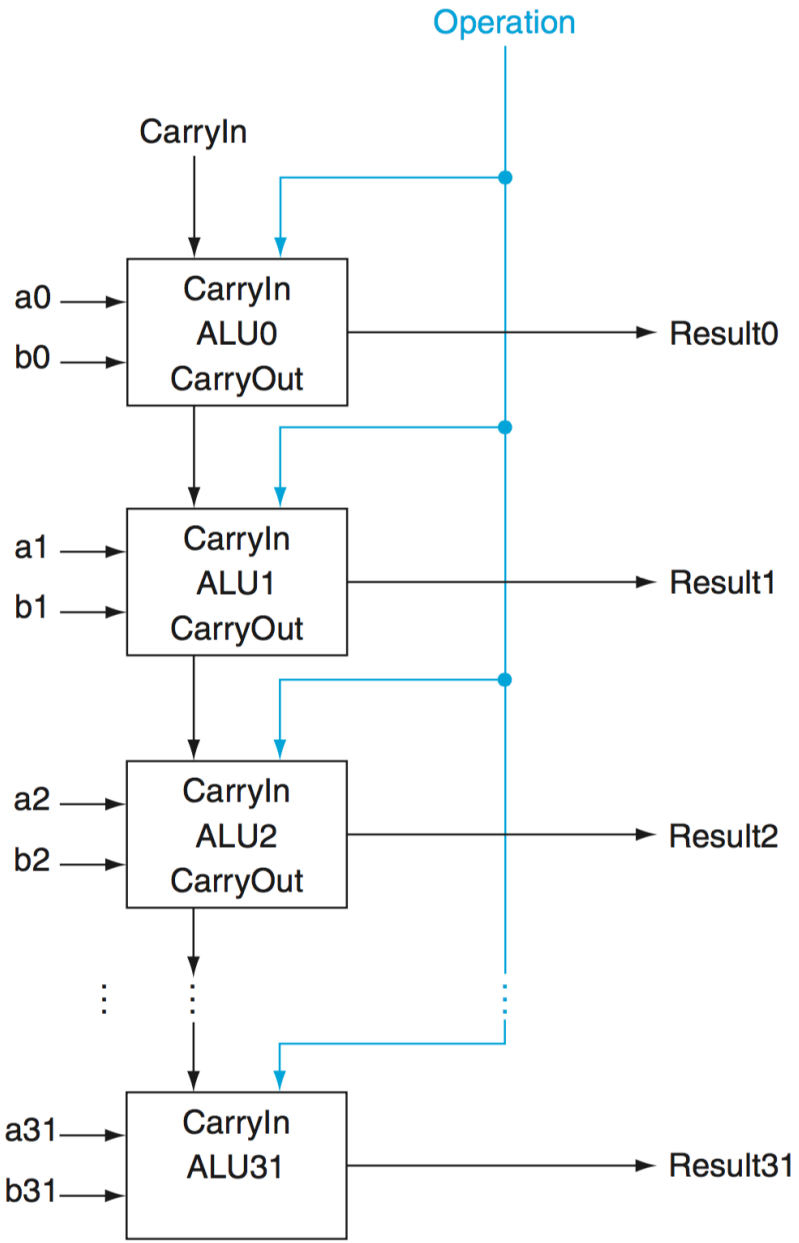
4-Bit Ripple Adder

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

A 1-Bit ALU

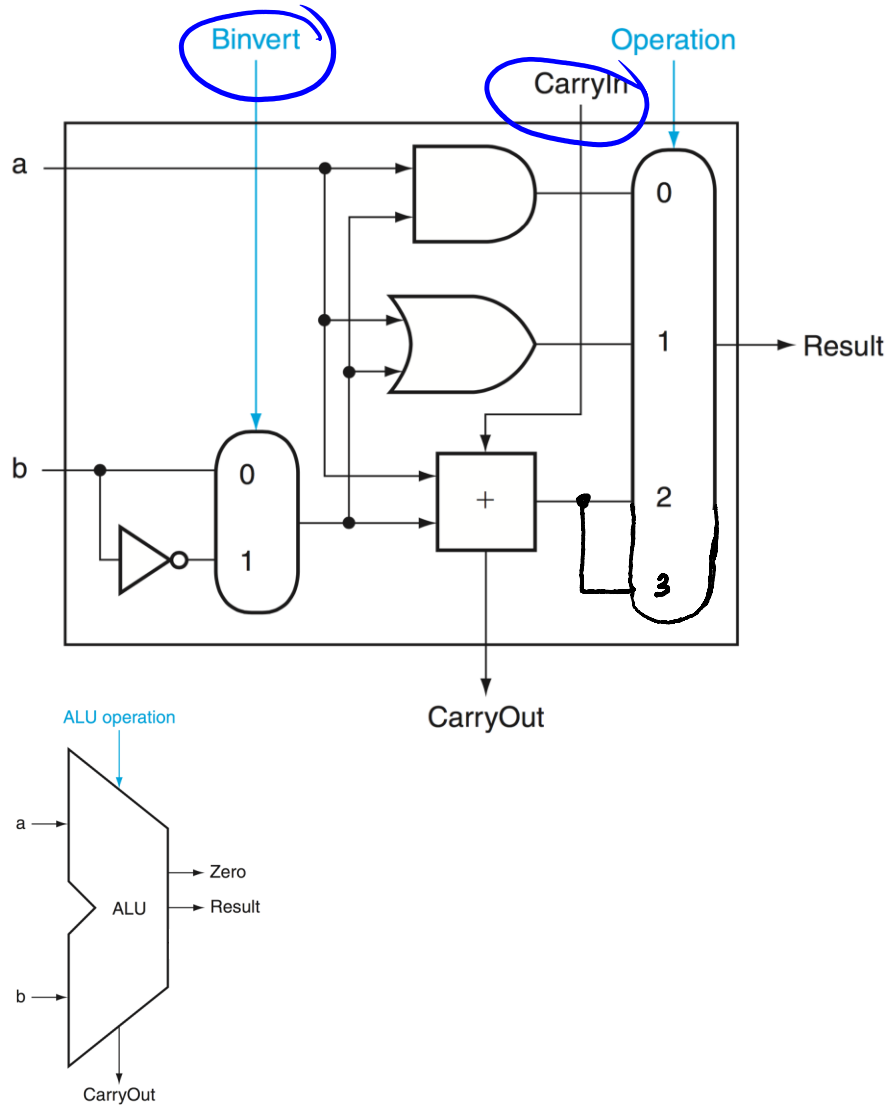


32-Bit ALU


Operation
CarryIn
a0
b0
CarryIn
ALU0
CarryOut
Result0
a1
b1
CarryIn
ALU1
CarryOut
Result1
a2
b2
CarryIn
ALU2
CarryOut
Result2
a31
b31
CarryIn
ALU31
Result31

Adding Subtraction and Flags to the ALU

- To subtract a – b, we take the two's complement of b and add
  - Two's complement of b is ~b + 1
  - Set `Binvert` and `CarryIn` to 1 (use lower bit of Operation signal)
- Flags
  - `CarryOut` – Use the `CarryOut` bit of the 32nd 1-bit adder
  - `Zero` – Or the 32 bits of the result together
- ALU Operation (2-bits)
  - 00 – And
  - 01 – OR
  - 10 – ADD
  - 11 – SUB
- Where does the Operation bits come from?
  - Directly from the encoding of the machine language instruction



Summary

- The ALU (Arithmetic Logic Unit) of a CPU is composed of Digital Logic Circuits
  - Logic gates connected in an acyclic graph
  - Treat as a deterministic black box
  - With a given input, you will get a given output (after a short time delay)
- Any truth table or Boolean equation can be implemented as a logic circuit
  - Sum of Products is one foolproof way to convert to a circuit
  - Will be correct, but not necessarily the smallest possible circuit
- More complex circuits are built out of smaller building blocks
- CPU development is done at a higher level than this (VHDL, Verilog)
  - Software tools convert high-level logic to circuit diagrams
- Next time: How registers are implemented