COMP2300-COMP6300-ENGN2219
Computer Organization &
Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian
National
University

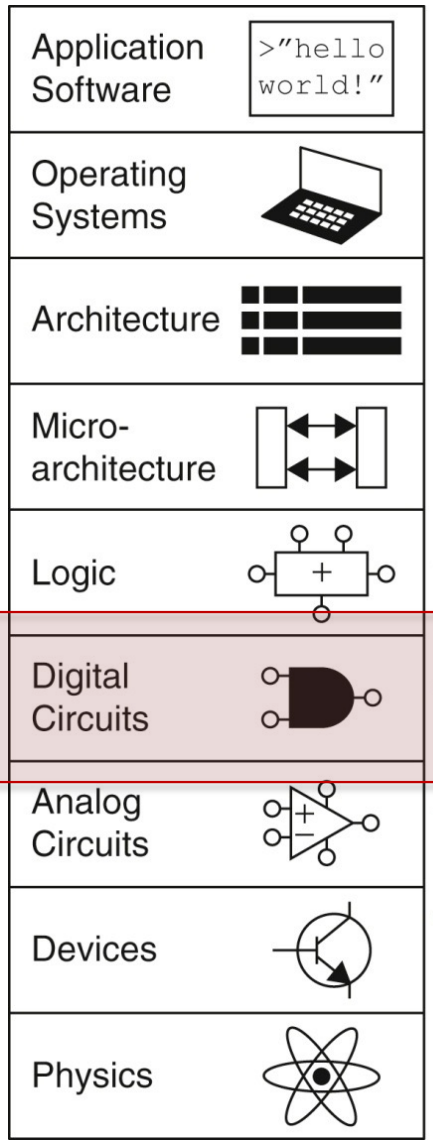
Plan: Week 2

Week 1: *Digital abstraction and binary digits*

Week 2: *Number systems for binary variables, Logic gates*

This Week: *Boolean logic & Logic gates (contd)*

This Week: *Combinational logic (more than just gates)*


Application Software
>"hello world!"
Programs
Operating Systems
Device Drivers
Architecture
Instructions Registers
Micro-architecture
Datapaths Controllers
Logic
Adders Memories
Digital Circuits
AND Gates NOT Gates
Analog Circuits
Amplifiers Filters
Devices
Transistors Diodes
Physics
Electrons

We are here

This Lecture

- Finish slides from last week
  - Bitwise operations and bit masks
- Brief overview of MOS transistors
- Combinational circuits

Bitwise Operations

- All logical operators can be applied to two bit-patterns (i.e., a group of bits) of *m* bits each, where *m is any # bits (8, 16, ...)*
  - Apply the operation individually to each pair of bits
  - If A and B are **8-bit** input sources (or source operands), then their AND/OR (C) is also **8-bit**

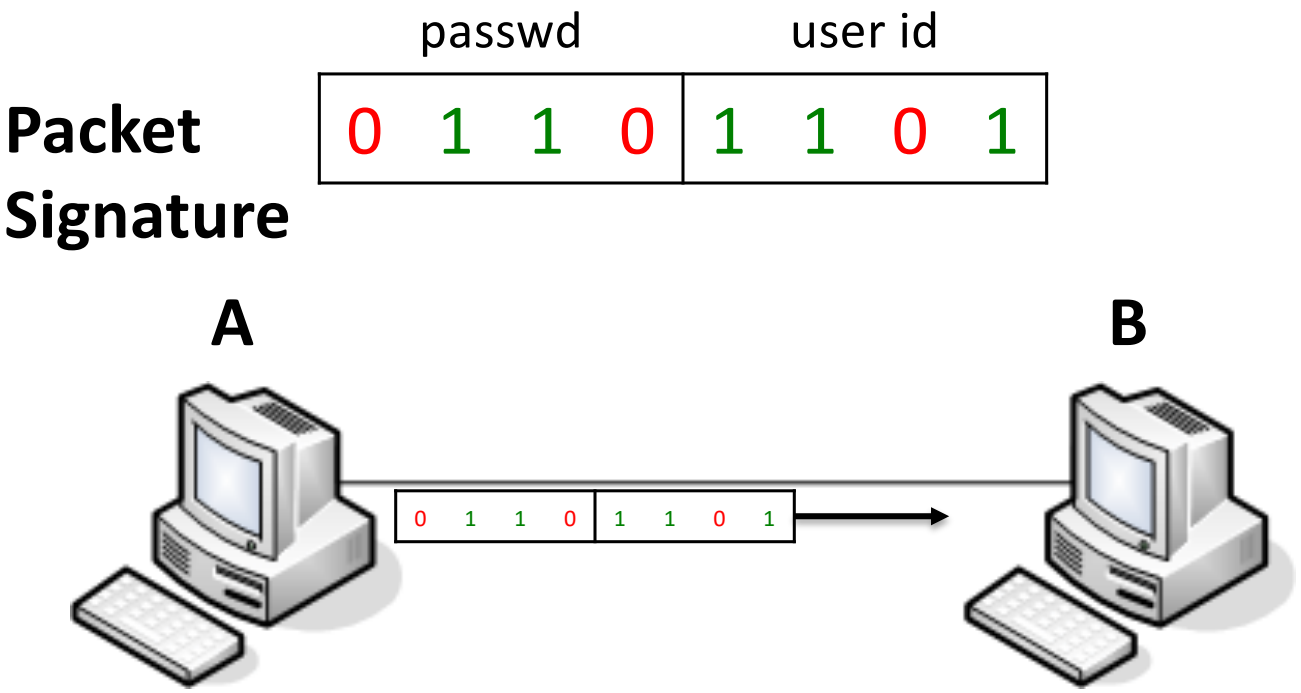
C = AB (bit-wise AND)

A	0	0	0	0	1	1	0	1
B	1	1	1	1	1	1	1	1
C	0	0	0	0	1	1	0	1

C = A + B (bit-wise OR)

A	0	0	0	0	1	1	0	1
B	0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	0	1

Bit Masks



B wants to create a *new packet* with user id set to **A**'s id and passwd bits set to 0 (e.g., to send a packet to another computer)

Bit Masks

- Suppose we are interested in extracting the least significant four bits from A, while ignoring the right-most four bits
  - If we AND A with B, and choose B as 00001111, then we get the desired bit pattern in C
  - **Bit mask:** A binary pattern (B) that separates the bits of A into two halves

C = AB (bit-wise AND)

A	0	1	1	0	1	1	0	1
B	0	0	0	0	1	1	1	1
C	0	0	0	0	1	1	0	1

Exercises

- Suppose we have a bit pattern, A = 11000010, and the rightmost two bits are of particular significance. Find a bitmask and a logical operation to mask out the values in the rightmost positions in a new bit pattern B. (**All other bits in B are set to 0.**)
- Suppose we have a bit pattern, A = 10110010, and the leftmost two bits are of particular significance. Find a bitmask and a logical operation to mask out the values in the leftmost positions in a new bit pattern B. (**All other bits in B are set to 1.**)

Exercise

- How can we find out if two bit-patterns A and B are identical?
- Verify that, 1 AND X = X, where X is a binary variable. Also, verify that, 0 OR X = X.
- Verify that, B AND B = B, where B is a binary variable. Also, verify that, B OR B = B.
- Verify that, B AND B’ = 0, where B is a binary variable. Also, verify that, B OR B’ = 1.

Announcement

Assessment Scheme

Here are the assessments for COMP2300/6300/ENGN2219.

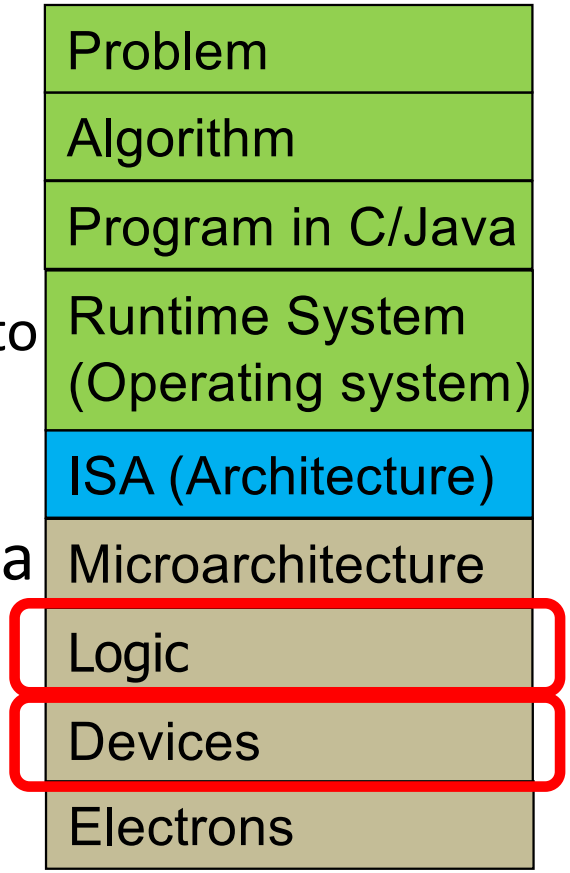
All times are in **Canberra Time**.

Assessment Item	Weighting	Due Date	Feedback Date (Approx)
Checkpoint 1[1]	5%	13/03/2023 - 10:00 am	17/03/2023
Assignment 1	25%	03/04/2023 - 10:00 am	17/04/2023
Checkpoint 2[1]	5%	08/05/2023 - 10:00 am	12/05/2023
Assignment 2	25%	22/05/2023 - 10:00 am	05/06/2023
Final Exam	40%	TBD	TBD

Transistors

- All computers are built from billions of small and simple structures called transistors
  - 1970: Few 1000s of transistors
  - Apple's M2 Max: **50+ Billion** transistors
  - **Moore's Law:** Transistor count double in 18 months
  - Computers with improved capability over time due to a large # transistors at the bottom

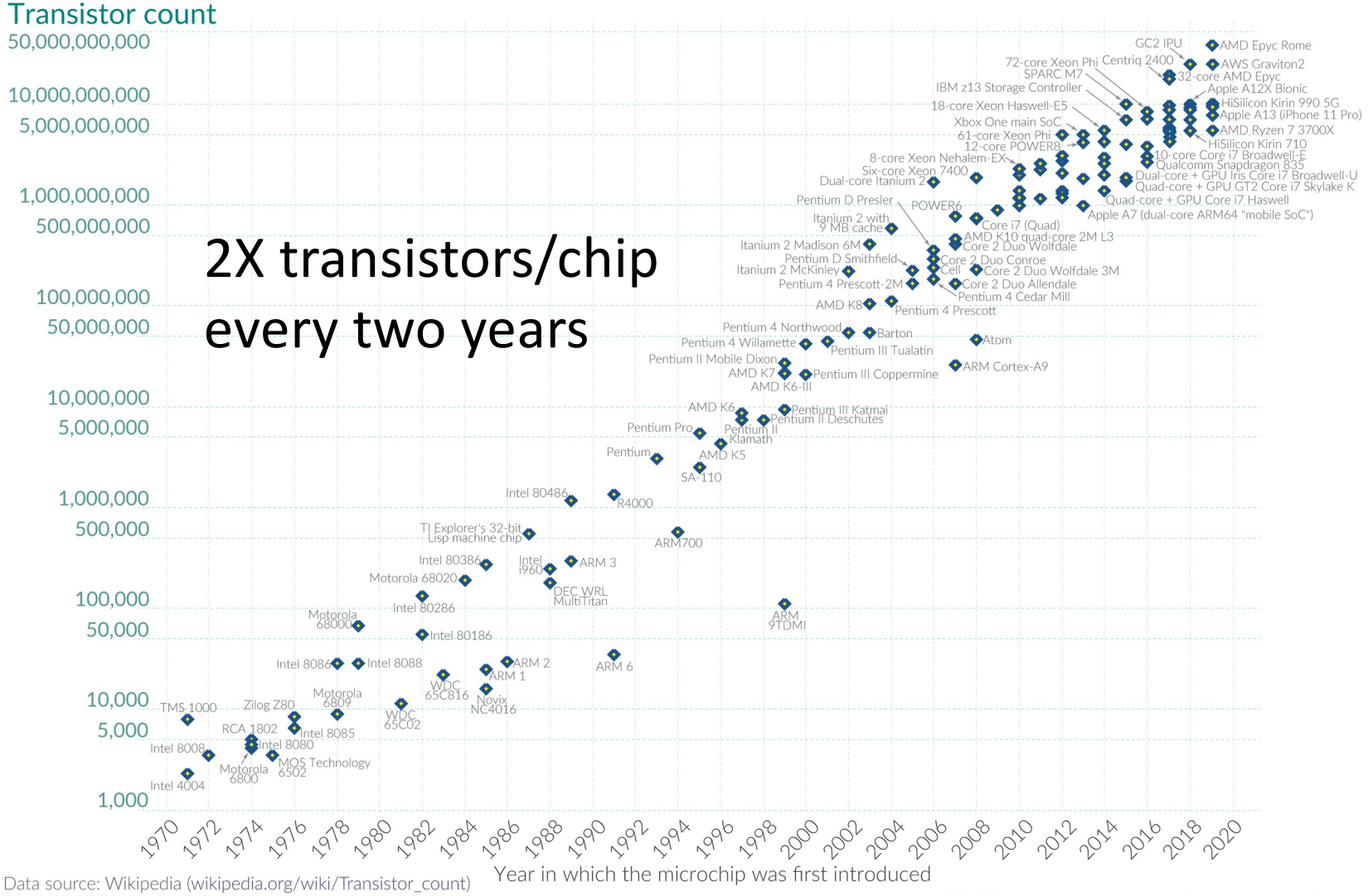
- **Our coverage:** Working of a MOS transistor (as a switch and a logic element)
  - Implementation of logic gates at the device level using transistors



Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



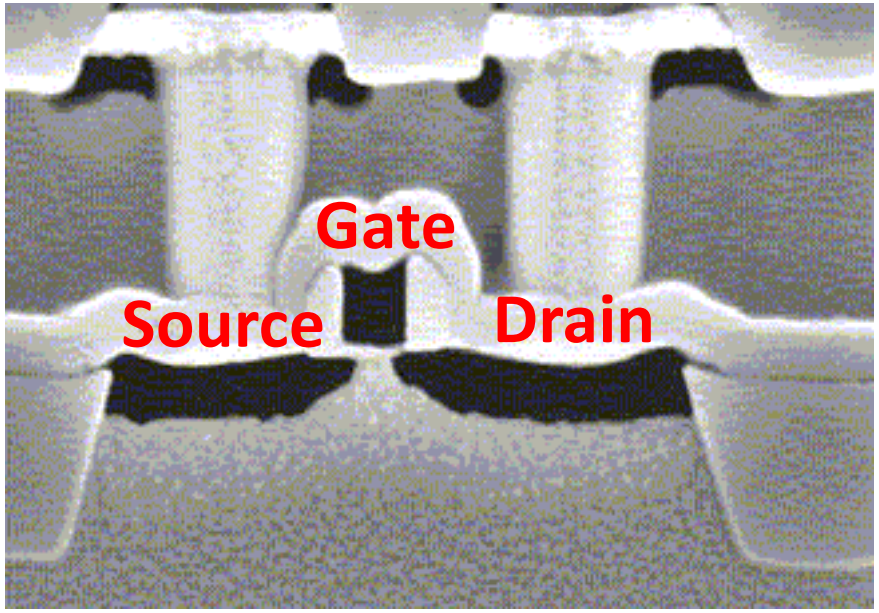
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

Transistors

- Sections 1.6 and 1.7 in Harris & Harris provide more technical explanations that we will cover

MOS Transistor

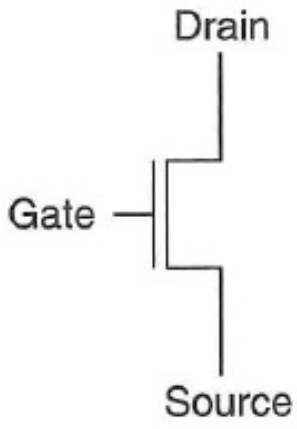
- **MOS** stands for
  - Conductors (**M**etal)
  - Insulators (**O**xide)
  - **S**emiconductors



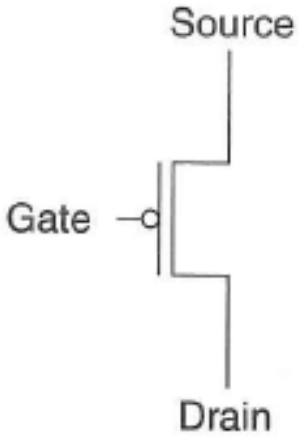
- MOS transistor has three terminals
- Combine many of these to form logic gates that we have seen
  - *The electrical properties of metal-oxide semiconductors are well beyond the scope of what we want to understand in this course*
  - They are below our **lowest level of abstraction**
  - If transistors **misbehave**, an architect is at their mercy (unlikely to happen)

Two Types of MOS Transistors

- Two types: n-type and p-type
- They both operate “logically,” very similar to the way wall switches work

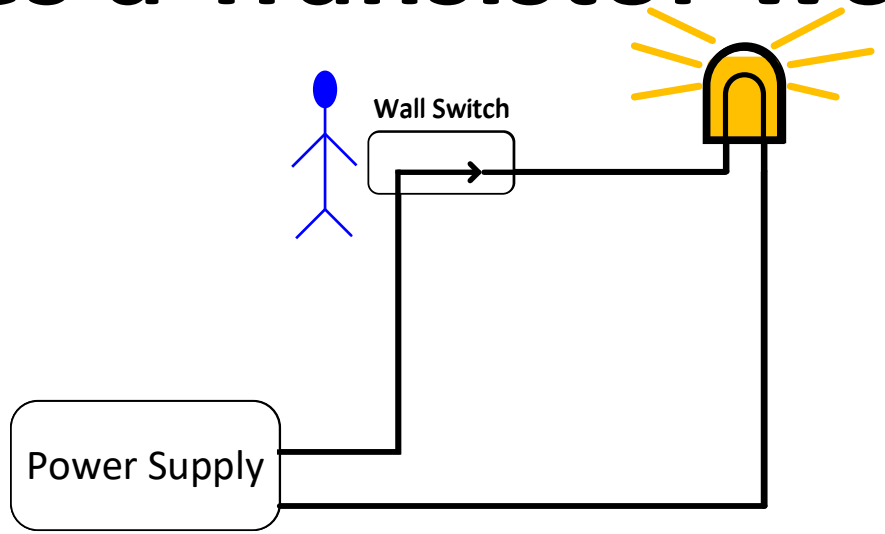


n-type



p-type

How Does a Transistor work?



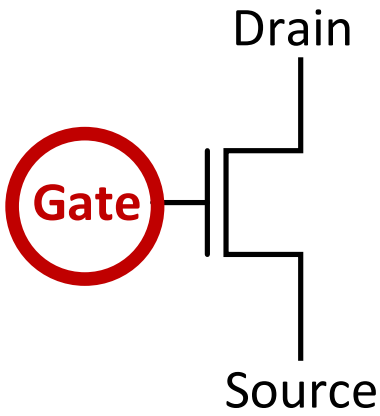
- For the lamp to glow, electrons must flow
- For electrons to flow, there must be a closed circuit from the power supply to the lamp and back to the power supply
- The lamp can be turned on and off by simply manipulating the wall switch to make or break the closed circuit

How Does a Transistor work?

- Instead of the wall switch, we could use an n-type of a p-type MOS transistor to make or break the closed circuit

If the gate of the n-type transistor is supplied with a **high** voltage, the connection from source to drain acts like a piece of wire (**we have a closed circuit**)

If the gate of the n-type transistor is supplied with **zero** voltage, the connection between source and drain is broken (**we have an open circuit**)

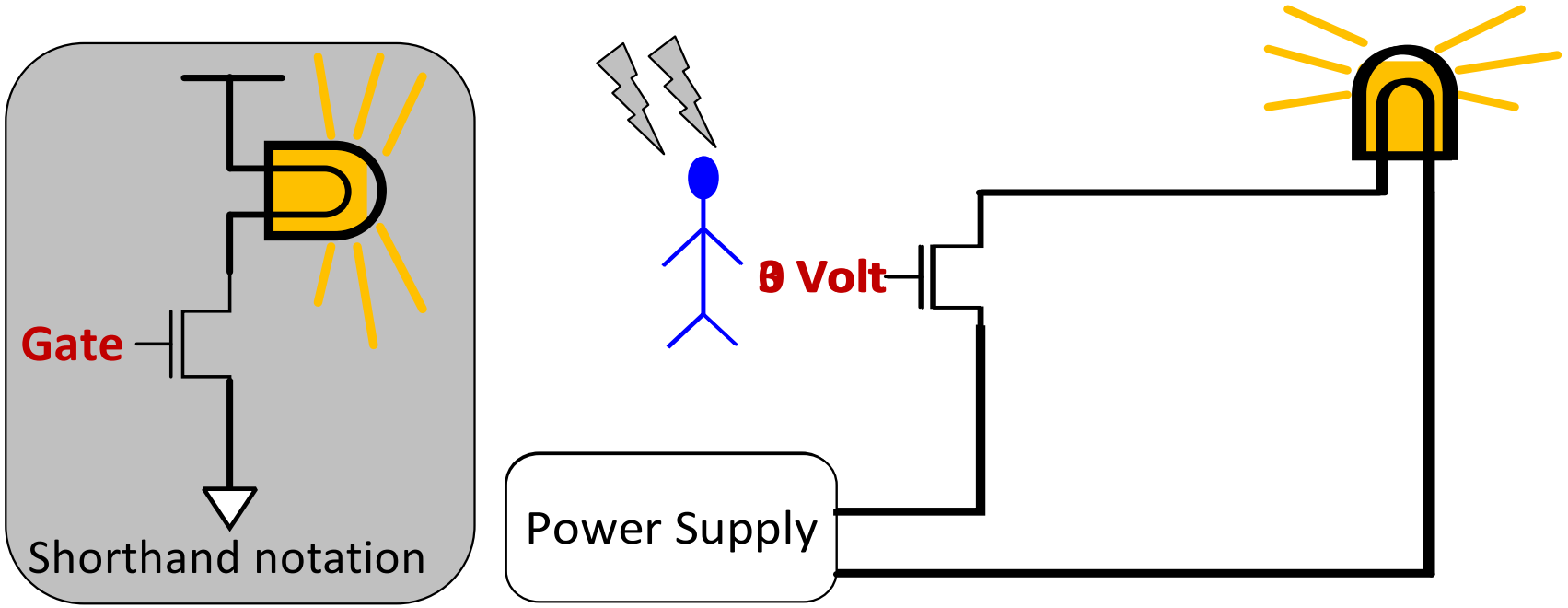


Schematic of an n-type MOS transistor

- Depending on the technology, high voltage can range from **0.3V** to **3V**

How Does a Transistor work?

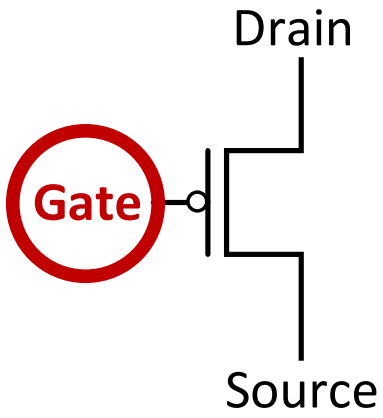
- The n-type transistor in a circuit with a battery and a bulb



How Does a Transistor work?

- The p-type MOS transistor works in exactly the opposite fashion from the n-type transistor

The circuit is **open** when the gate is supplied with 3V

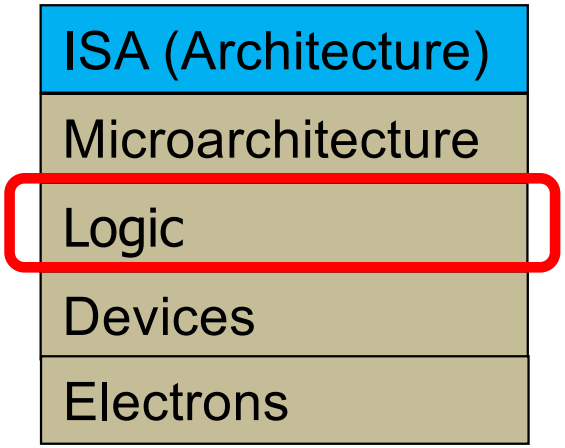


The circuit is **closed** when the gate is supplied with 0V

- Depending on the technology, high voltage can range from **0.3V** to **3V**

One Level Higher in Abstraction

- Now, we know how a **MOS transistor** works
- Let's revisit how we build logic structures out of MOS transistors
- We called these logic structures logic gates and they implement simple **Boolean** functions

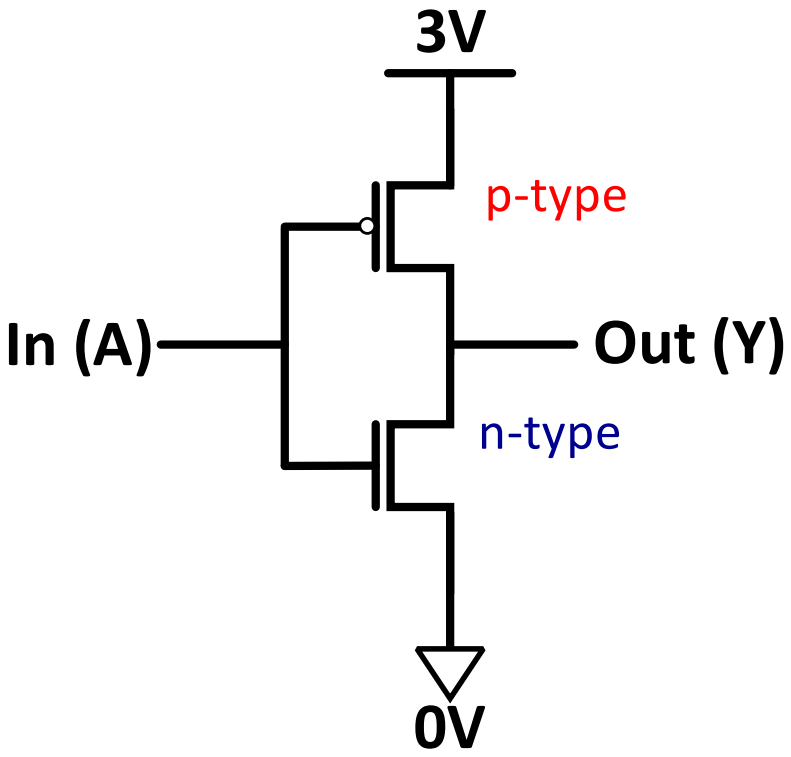


Making Logic Gates Using CMOS Technology

- Modern computers use both n-type and p-type transistors, called Complementary MOS (CMOS) technology
  - **nMOS + pMOS = CMOS**

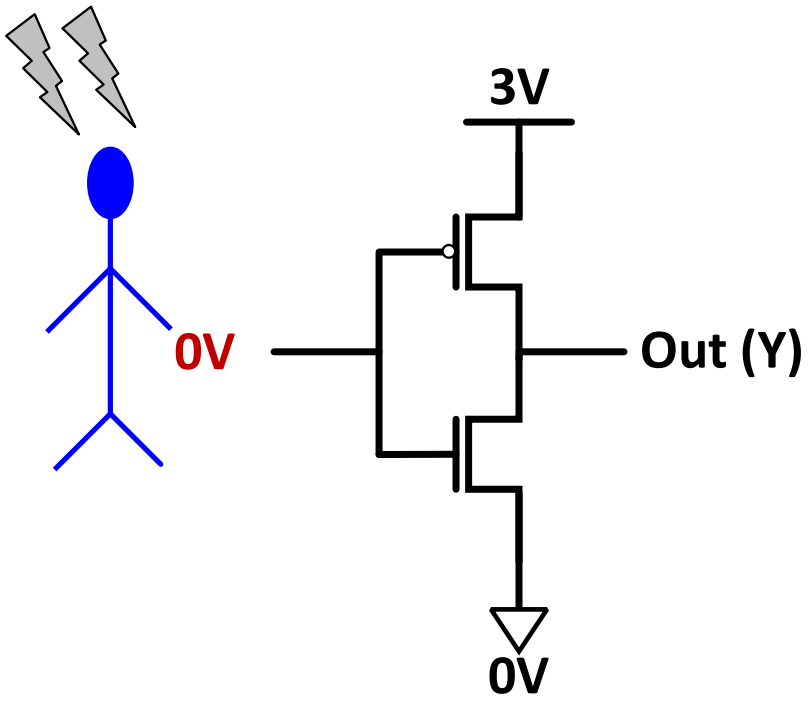
CMOS Technology Fundamentals

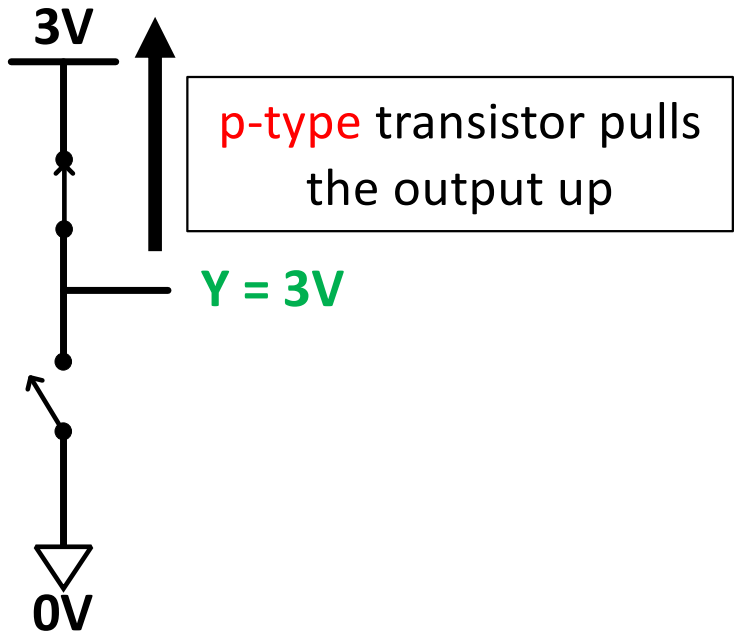
- Let's look at the simplest logic structure that exists in a modern computer
  - What does this circuit do?



CMOS Technology Fundamentals

- What happens when the input is connected to 0V?



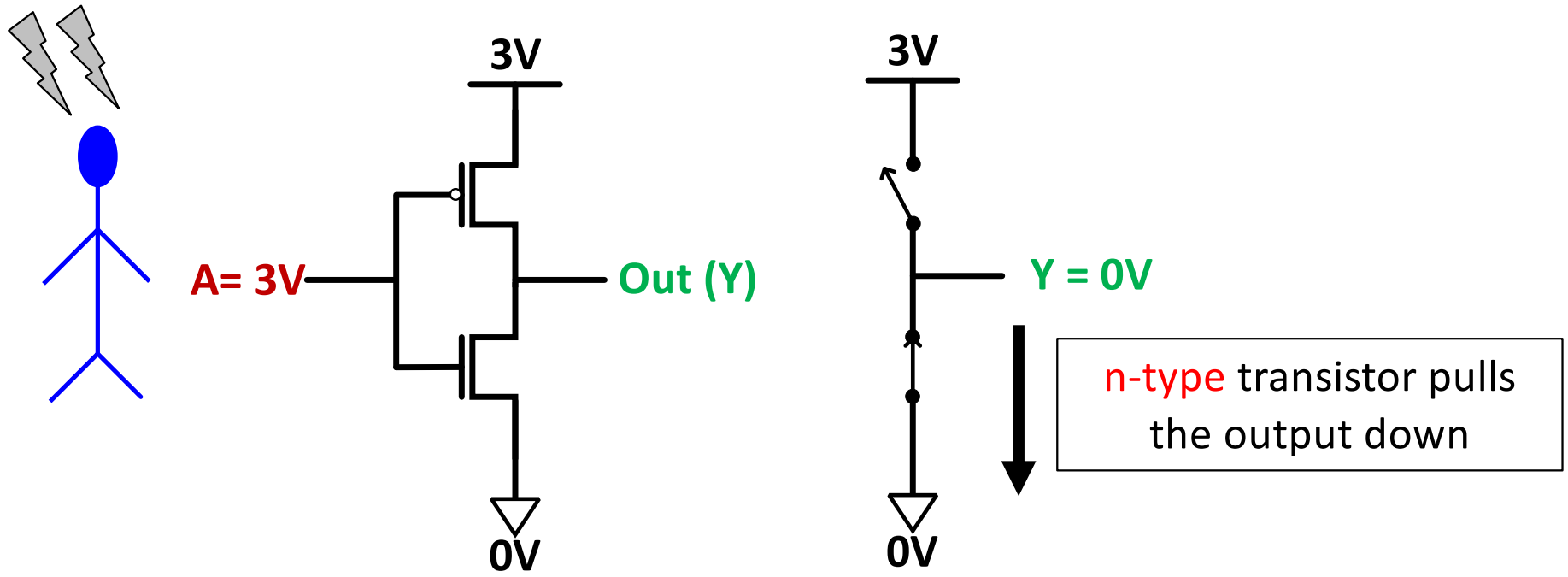


CMOS Technology Fundamentals

- **p**-type transistors are good at pulling u**p** the voltage

CMOS Technology Fundamentals

- What happens when the input is connected to 3V?



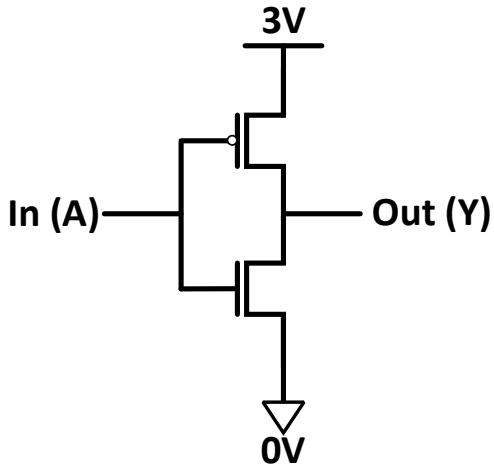
CMOS Technology Fundamentals

- **n**-type transistors are good at pulling dow**n** the voltage

CMOS NOT Gate (Inverter)

- We have seen a NOT gate at the transistor level
  - If A = **0V** then Y = **3V**
  - If A = **3V** then Y = **0V**

- Interpretation of voltage levels
  - Interpret **0V** as logical (binary) 0 value
  - Interpret **3V** as logical (binary) 1 value



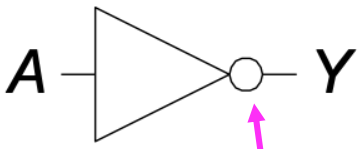
A	P	N	Y
0	**ON**	OFF	1
1	OFF	**ON**	0

$$Y = \bar{A}$$

CMOS NOT Gate (Inverter)

A	Y
0	1
1	0

The NOT gate has only one input (unary)



bubble → invert

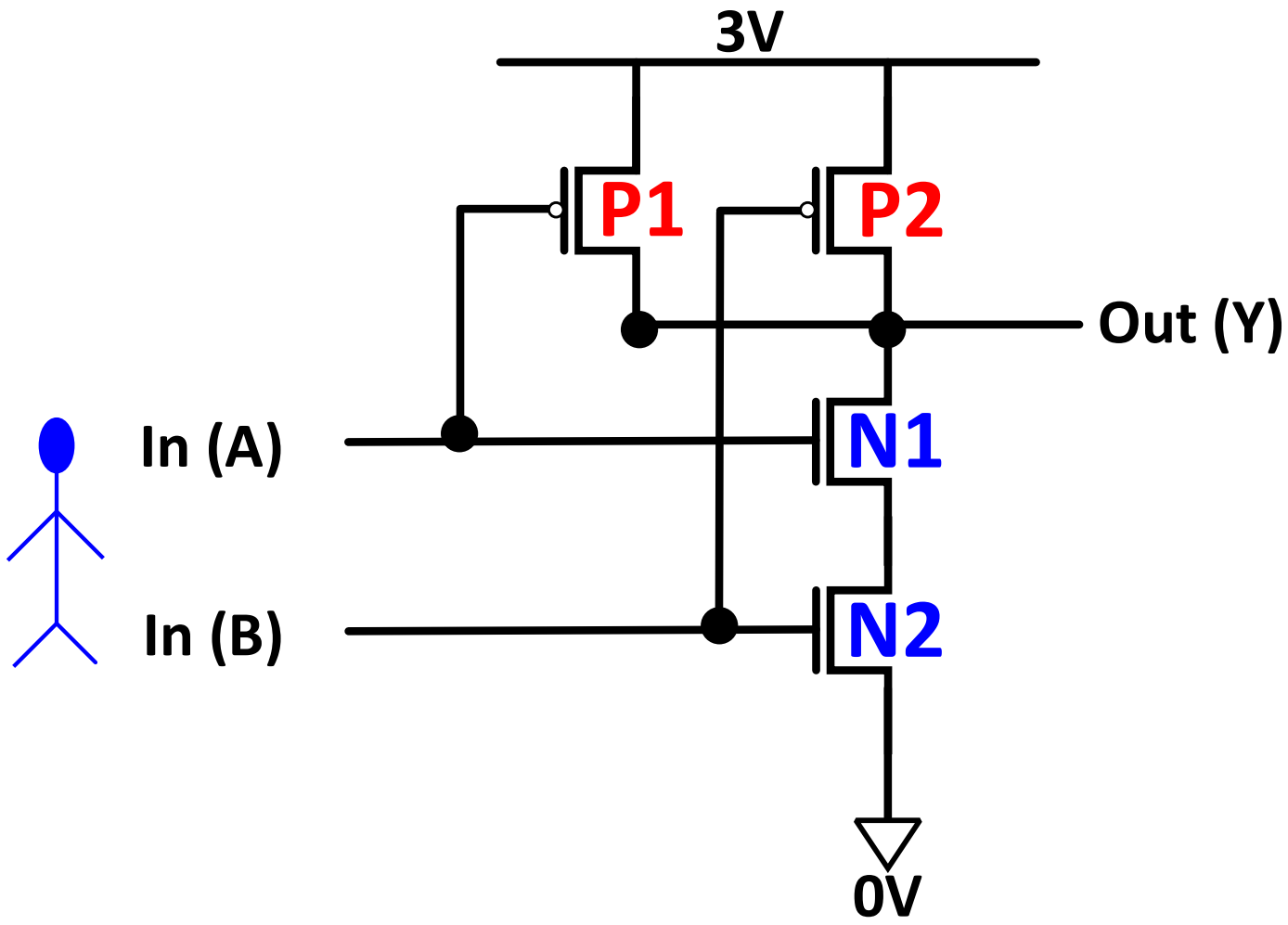
$Y = A'$

$Y = \bar{A}$

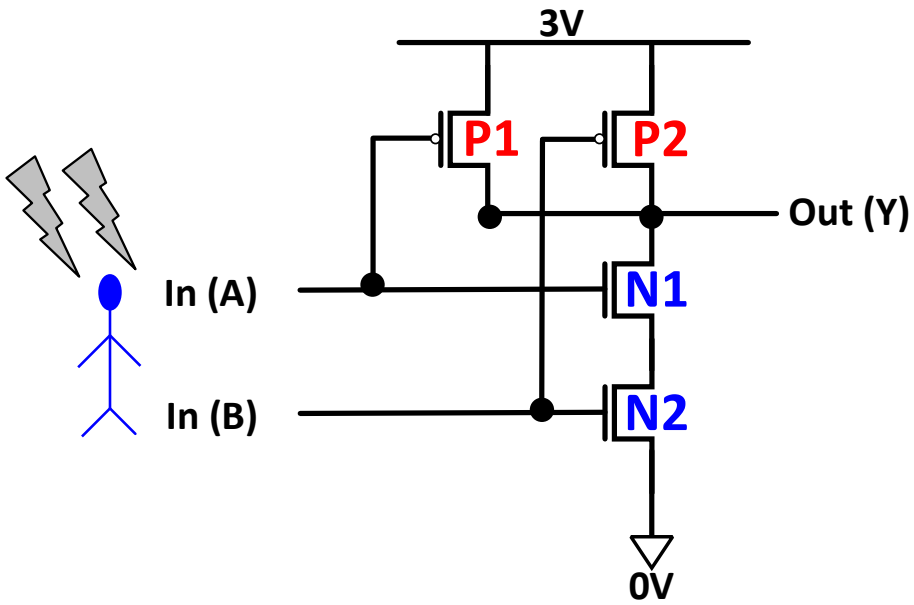
Truth Table NOT Logic Gate Boolean Equation

NOT Function: *The output Y is the inverse of the input A*

Another CMOS Gate: What is this?



CMOS NAND Gate



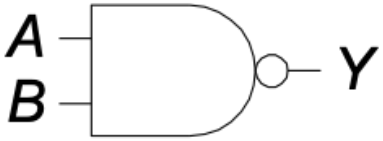
A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

- P1 and P2 are in parallel; only one must be **ON** to pull up the voltage to 3V
- N1 and N2 are connected in series; both must be **ON** to pull down the voltage to 0V

CMOS NAND Gate

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Truth Table



NAND Logic Gate

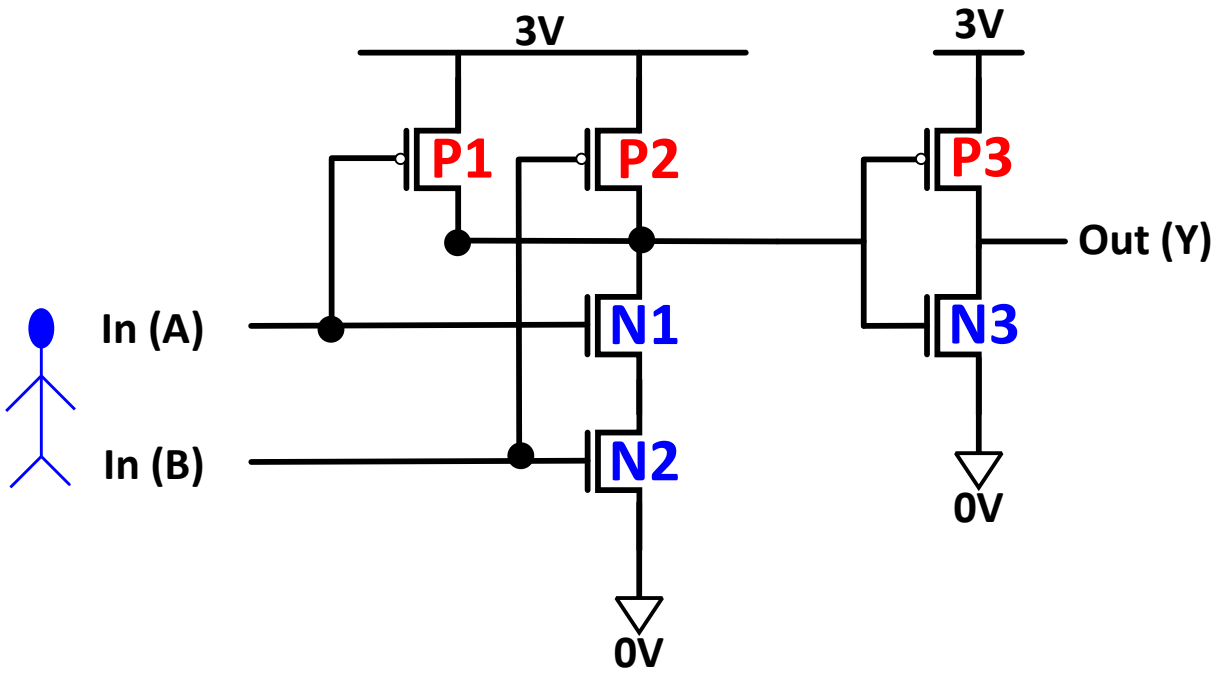
$Y = (AB)'$

$Y = \overline{AB}$

Boolean Equation

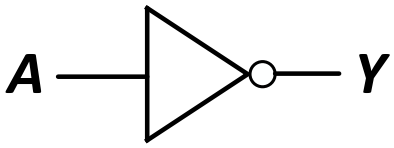
NAND Function: *The output Y is 1 unless both inputs are 1*

CMOS AND Gate

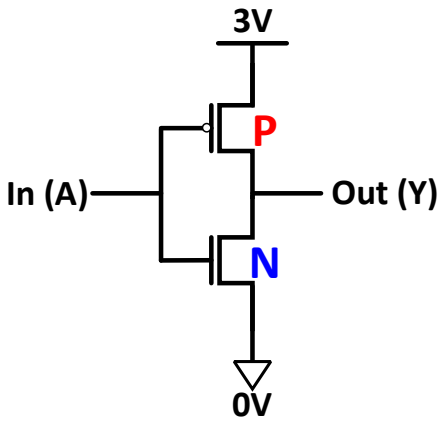


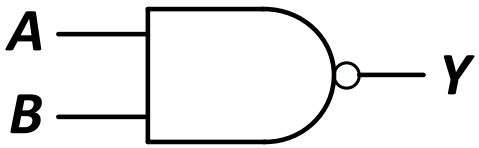
- We make an AND gate using one NAND gate and one NOT gate
- **Your task:** Can we not use fewer transistors for the AND gate?

CMOS NOT, NAND, and AND Gates

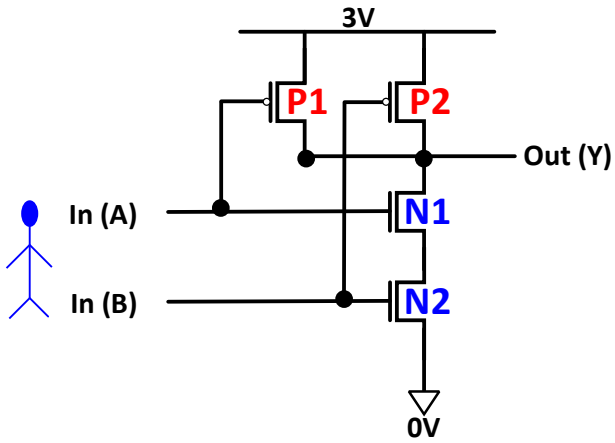


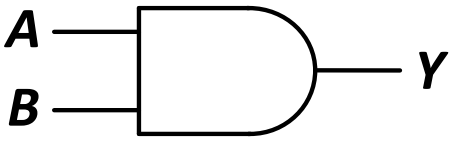
A	Y
0	1
1	0



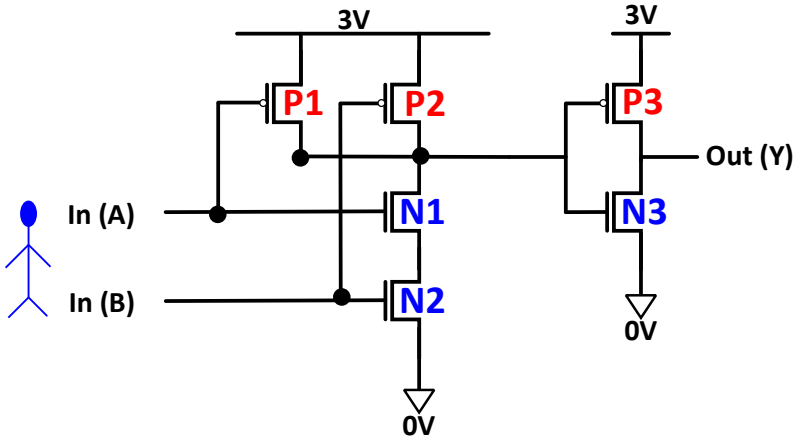


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

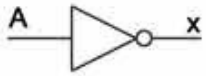
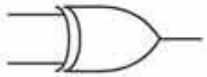




A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

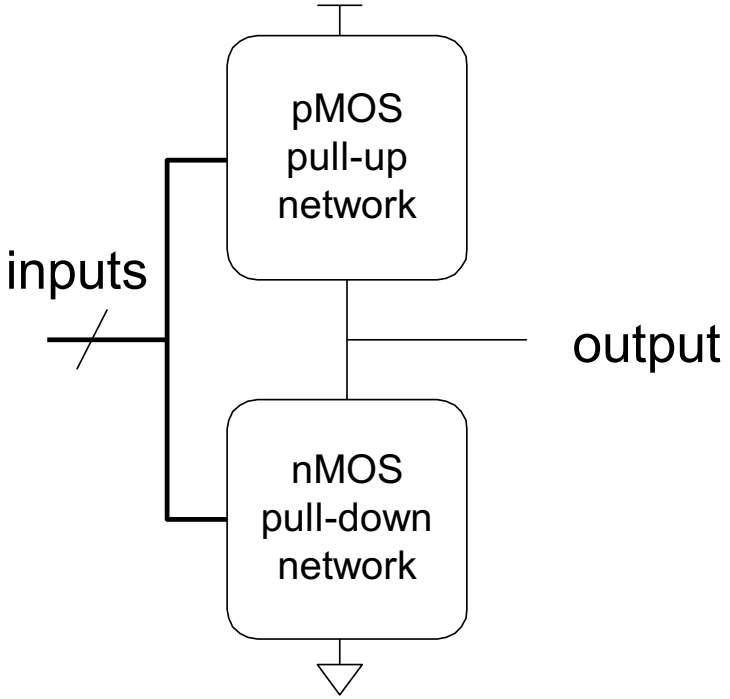


Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
Symbol							
Truth Table	A | X 0 | 1 1 | 0	B A | X 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	B A | X 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0	B A | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	B A | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0	B A | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	B A | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

General CMOS Gate Structure

- We have a general form to construct any inverting logic gate, such as, NOT, NAND, NOR
  - The networks may consist of transistors in series of in parallel
  - When transistors are in **parallel**, the network is **ON** if one of the transistors is **ON**
  - When transistors are in **series**, then network is **ON** only if all transistors are **ON**

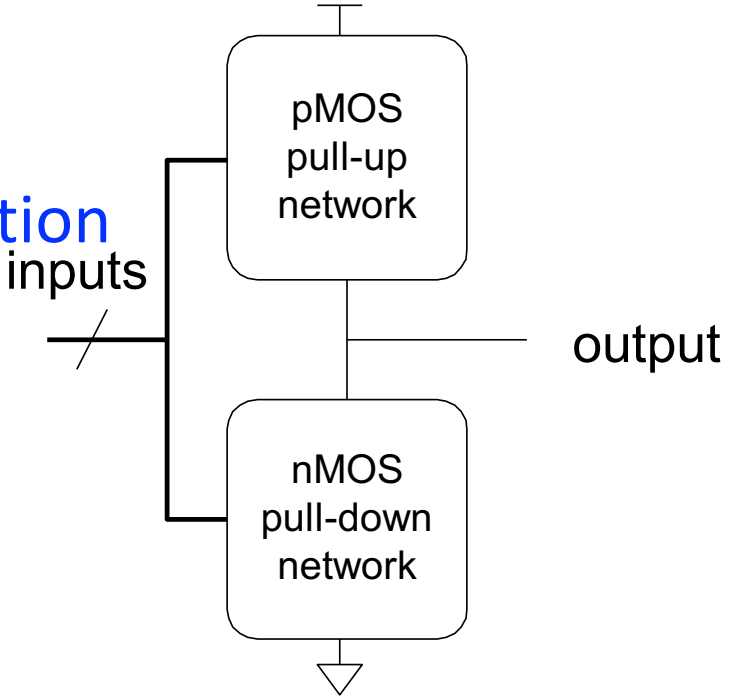


pMOS transistors are used for pull-u**p**

nMOS transistors are used for pull-dow**n**

General CMOS Gate Structure

- Exactly one network should be ON, and the other network should be OFF at any given time
- If both networks are ON simultaneously, there is a short circuit → incorrect operation
- If both networks are OFF simultaneously, the output is floating → undefined



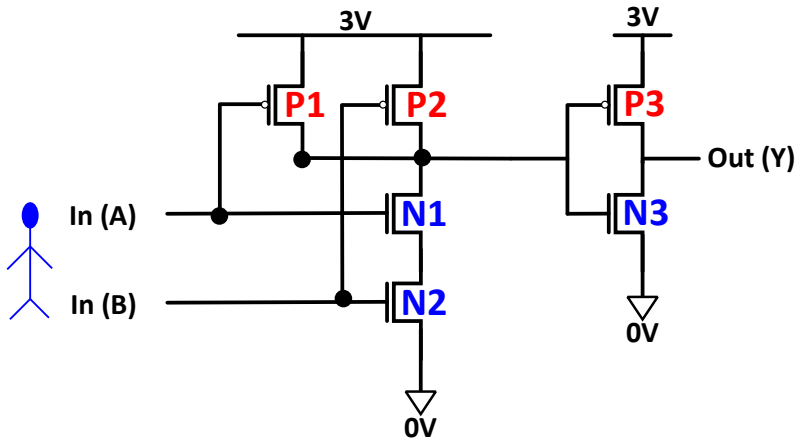
pMOS transistors are used for pull-u**p**

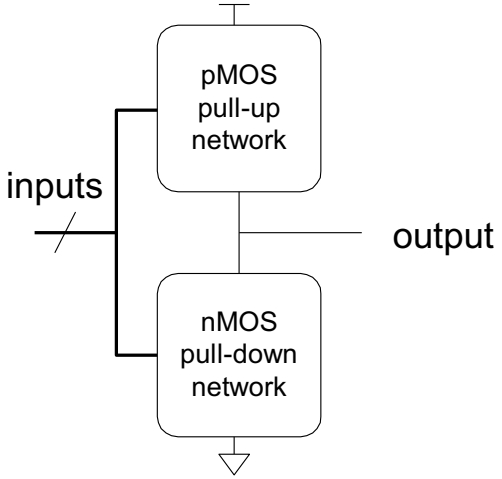
nMOS transistors are used for pull-dow**n**

Why This Structure?

- MOS transistors are imperfect switches

- pMOS transistors pass 1's well but 0's poorly
  - **p**MOS transistors are good at "pulling u**p**" the output
- nMOS transistors pass 0's well but 1's poorly
  - **n**MOS transistors are good at "pulling down" the output





Latency

- Which one is faster?
  - Transistor in series
  - Transistors in parallel

- Series connections are slower than parallel connections
  - More **resistance** on the wire

- **Remember: Latency of series vs. parallel circuits extend from transistors to gates and larger circuits**

- See Section 1.7.8 for more details

Gates with More Than Two Inputs

- We can create larger gates with more than 2 inputs
  - 3-input NOR gate or 11-input NAND gate

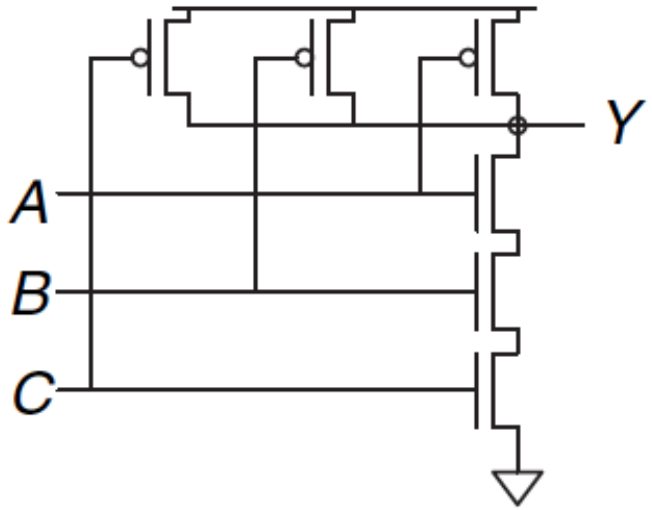
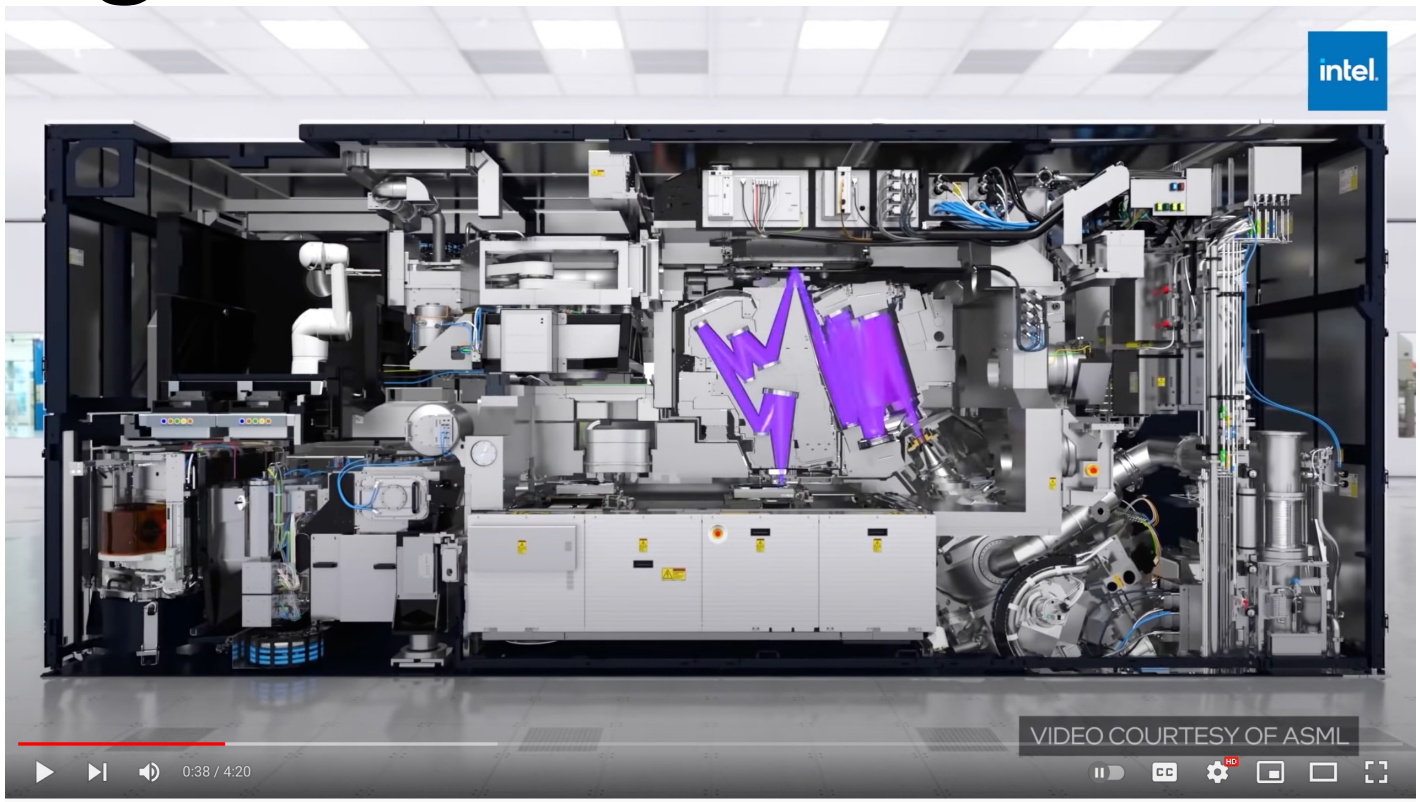


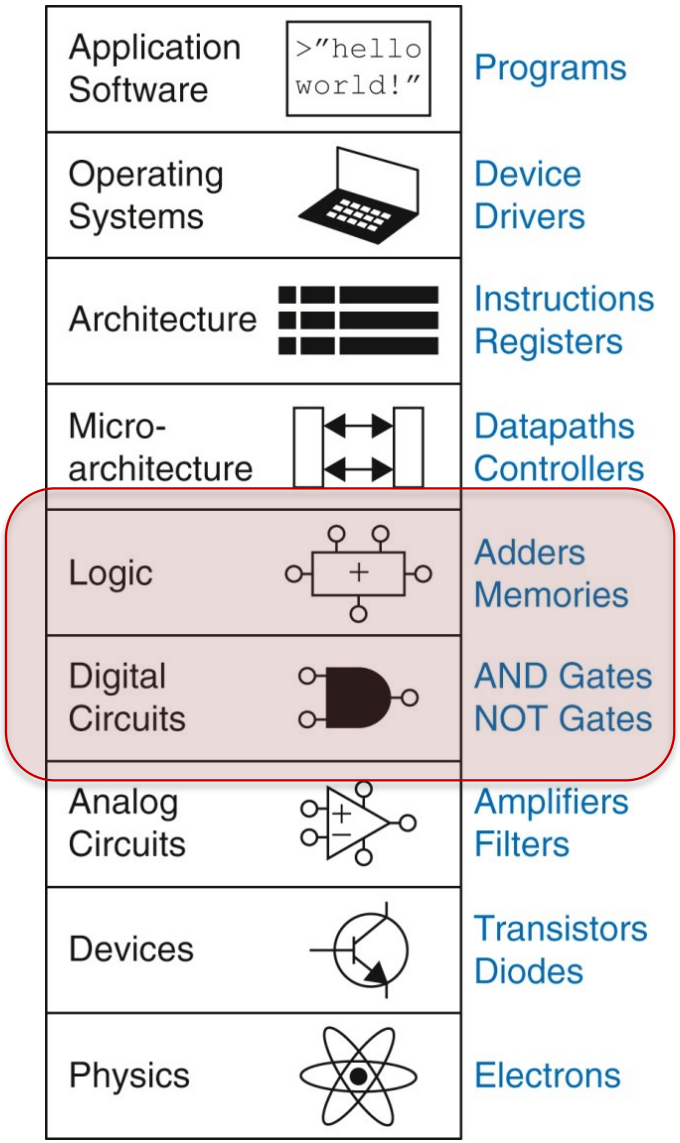
Figure 1.35 Three-input NAND gate schematic

Manufacturing Tech is the Enabler

- **Precision Manufacturing**
  - Extreme Ultraviolet (EUV) light to pattern <10nm structures



https://www.youtube.com/watch?v=Jv40Viz-KTc



Broadening our horizon "one layer at a time"

Classification of Digital Circuits

- **Combinational Circuit:** Output depends **only** on the current values of the inputs
  - Memory-less (a **distinct** and **critical** feature)
  - All logic gates are combinational

- **Sequential Circuit:** Output depends *not only* on the current inputs, but also past values of the inputs
  - The sequence of inputs **over time** decide the **next** output
  - Sequential circuits have state/memory
  - Example: Elevator controller (**State:** ON_GROUND, IN_TRANSIT, AT_TOP)

Section 2.1 of H&H

Combinational Behavior

- **Example:** Suppose a combinational circuit, consisting of an AND gate, with two inputs, A and B

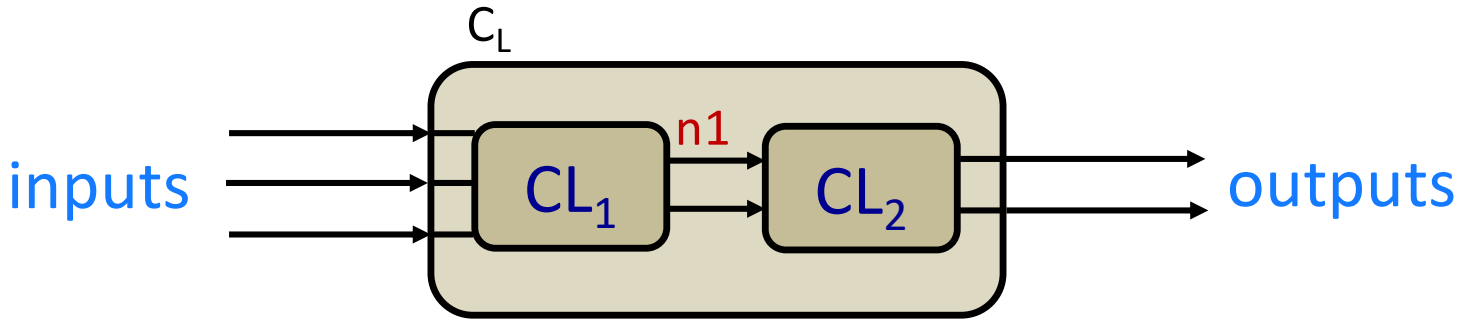
time →	*t0*	*t1*	*t2*	*t3*	*t4*	*t5*	*t6*
A	0	1	1	0	1	0	1
B	0	1	0	0	1	0	1
Output	0	1	0	0	1	0	1

- At time t6, the **sequence** of changes to A and B between t0 – t5 is irrelevant.
- Output is strictly determined by the values of A and B at t6

Combinational Circuits

- **Functional specification:** What is the behavior of the circuit?
- What is the output for a given combination of input values?
- **Timing specification:** How long does the circuit takes to produce the output?
  - **Worst-case:** 10 nanoseconds
  - **Best-case:** 1 nanoseconds

Combinational Circuits



- **Hierarchy:** The top-level circuit, C_L, is made up for of two combinational sub-circuits, CL_1 and CL_2
- **Nodes:** n1 is an internal wire or node
- **Abstraction:** The input and output interface, and the functional and timing specification is enough for someone to use C_L. Inner composition of C_L is not important.

Implementing Combinational Logic

- Steps in implementing combinational Logic
  - Initial specification (e.g., in English)
  - Construct the truth table
  - Derive the Boolean equation

  Functional specification

  - Simplify the Boolean equation (use Boolean algebra)
  - Implement the equation using logic gates

Specification

[Happiness detector] Mr. X is not happy if there is an assignment deadline, or their favorite bar is closed. Design a circuit that outputs 1 only if Mr. X is happy.

[Multiplexer] Design a circuit with three inputs: D_0, D_1, select; and one output. The output is D_0 if select is 0, and D_1 if select is 1.

[Half Adder] Design a circuit that adds two binary variables: A and B. The circuit has two outputs: sum and carry-out (C_{out}).

[Full Adder] Design a circuit that adds three binary variables: A, B, and a carry-in (C_{in}). The circuit has two outputs: sum and carry-out (C_{out}).

Constructing Truth Tables

- Identify inputs and outputs (interface)
  - The interface maybe implicitly specified or require some thought
- Write all the possible combinations of input values
  - For each input combination, determine the output
  - All inputs to the left, outputs to the right

Truth Table: Happiness Detector

Specification: Mr. X is not happy if there is an assignment deadline, or their favorite bar is closed. Design a circuit that outputs 1 only if Mr. X is happy.

Interface

Assignment deadline? (D)

- 0: there is not a deadline
- 1: there is a deadline

Bar is closed? (B)

- 0: open
- 1: closed

Happy (H): 1 → ☺, 0 → ☹

Truth Table

D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Deriving a Boolean Equation

- Boolean equation is an alternative way to represent the function of a combinational logic block

- Enables systematic transformation of the function into simpler functions
  - Different hardware implementations
  - Simplification process can be automated via **Computer-Aided Design (CAD)** and **Electronic Design Automation (EDA)**

- Different Boolean expressions lead to different logic gate implementations
  - **Different hardware area, cost, latency, energy properties**

Definitions

- **Complement:** For any binary variable X, its complement is X'
- **Literal:** True form (X) and complementary form (X')
- **Implicant:** AND of one or more literals is product or implicant
  - X, Y, XY, X'Y'Z, XYZ, XY'Z' are all **implicants** for a function of three variables
- **Minterm:** product (AND) involving all the inputs to the function
  - XYZ is a **minterm** for a function of three variables X, Y, and Z
- **Maxterm:** sum (OR) than includes all input variables
  - X + Y + Z, X' + Y + Z'

Section 2.2 of H&H

Minterms and Maxterms

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

- Each **minterm** is obtained from an **AND** term of **n** variables
  - Use **prime** of the variable if the bit is 0 and **unprimed** if 1
  - The subscript *j* in the symbol for each minterm (m_j) denotes the **decimal equivalent** of the binary number of the minterm designated

Operation Precedence

- NOT has the highest precedence
- Next: AND
- Last: OR
- Example: Y = A + BC’
  - First, we find C’
  - Then, we find BC’ (product/AND)
  - Finally, we perform A + (the *result* of BC’)

Standardized Representations

- Also called canonical representations
- Sum of Products (SOP) form
  - Today's lecture
- Product of Sums (POS) form
  - After learning about Boolean algebra (Lecture # 4 on Friday)

Sum of Products (SOP)

- **Sum of Products Form (SOP)**
  - Also known as **disjunctive normal form** or minterm expansion
  - SOP is **canonical/standard** form of a Boolean function

- We have a truth table of a Boolean Function **F** and we need to express the function in terms of inputs in a **standard manner**
  - Give it a unique algebraic signature

- Truth table is an expensive representation
  - More compact and unique signature of a Boolean function

- All Boolean equations can be written in SOP form

Key Idea of SOP

- ***Express the truth table as a two-level Boolean expression***
  - contains all input variable combinations that result in a **1** output
  - if **ANY** of the combinations of input variables that result in a **1** is **TRUE**, then the output is **1**
  - **F** = **OR** of all input variable combinations that result in a **1**

- ***Why does it work?***
  - Output is **1** whenever the corresponding minterm is **1**
  - Minterm is **1** when the corresponding input combinations result in the minterm evaluating to **1**

SOP: Simple Example (1 minterm)

To write the Boolean equation for a truth table, sum each of the minterms for which the output is 1

A	B	Y1	minterm	name
0	0	0	A'B'	m_0
0	1	1	A'B	m_1
1	0	0	AB'	m_2
1	1	0	AB	m_3

Boolean Eq

Y1 = A'B

Y1 is 1 *only* when A = 0 and B = 1

Conversely, when A' = 1 and B = 1

SOP: Example (2 minterms)

To write the Boolean equation for a truth table, sum each of the minterms for which the output is 1

A	B	Y1	minterm	name
0	0	0	A'B'	m_0
0	1	1	A'B	m_1
1	0	0	AB'	m_2
1	1	1	AB	m_3

Boolean Eq

Y1 = A'B + AB

Y1 is 1 ***either*** when A = 0 and B = 1

OR, when A = 1 and B = 1

$Y1 = \sum(1,3)$

Another SOP Example

- **The function evaluates to TRUE (1) if any of the products (minterms) causes the output to be 1**

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = \sum(3, 4, 5, 6, 7)$$

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

SOP in Sigma Notation

$$\mathbf{F} = \sum(\mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7})$$

- 3, 4, 5, 6, 7 denote the decimal equivalent of the binary number of the minterm designated in the truth table

SOP Summary

- A Boolean function can be expressed **algebraically** from a given truth table by forming a **minterm** for each combination of the variables that produces a **1** in the function and then taking the OR of all those terms

- The **minterms** whose sum defines the Boolean function are those that give the 1's of the function in a truth table

- The sum of products canonical form can also be written in **sigma** notation using the summation symbol, $\sum(\boldsymbol{m1}, \boldsymbol{m2}, \ldots)$

Equation: Happiness Detector

Specification: Mr. X is not happy if there is an assignment deadline, or their favorite bar is closed. Design a circuit that outputs 1 only if Mr. X is happy.

Truth Table

D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Eq

H = D'B'

H = (D)' AND (B)'

From Equation to Gates

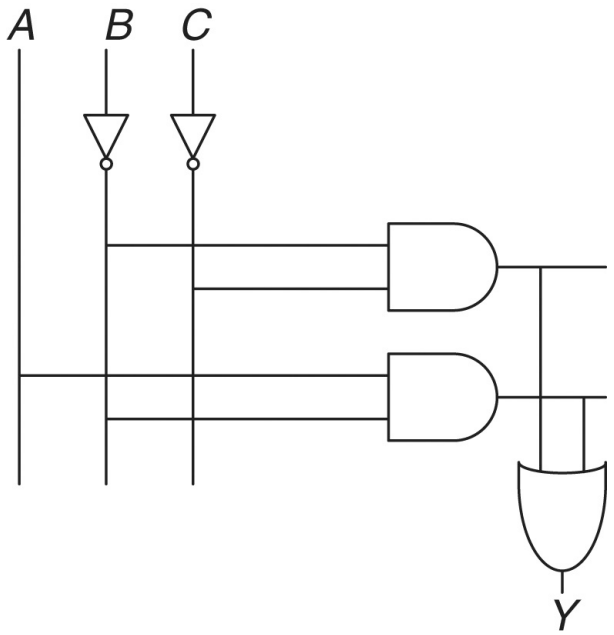
Schematic: A diagram of a digital circuits with elements (gates) and the wires that connect them together

Example Boolean Eq

Y = AB’ + B’C’

Schematic

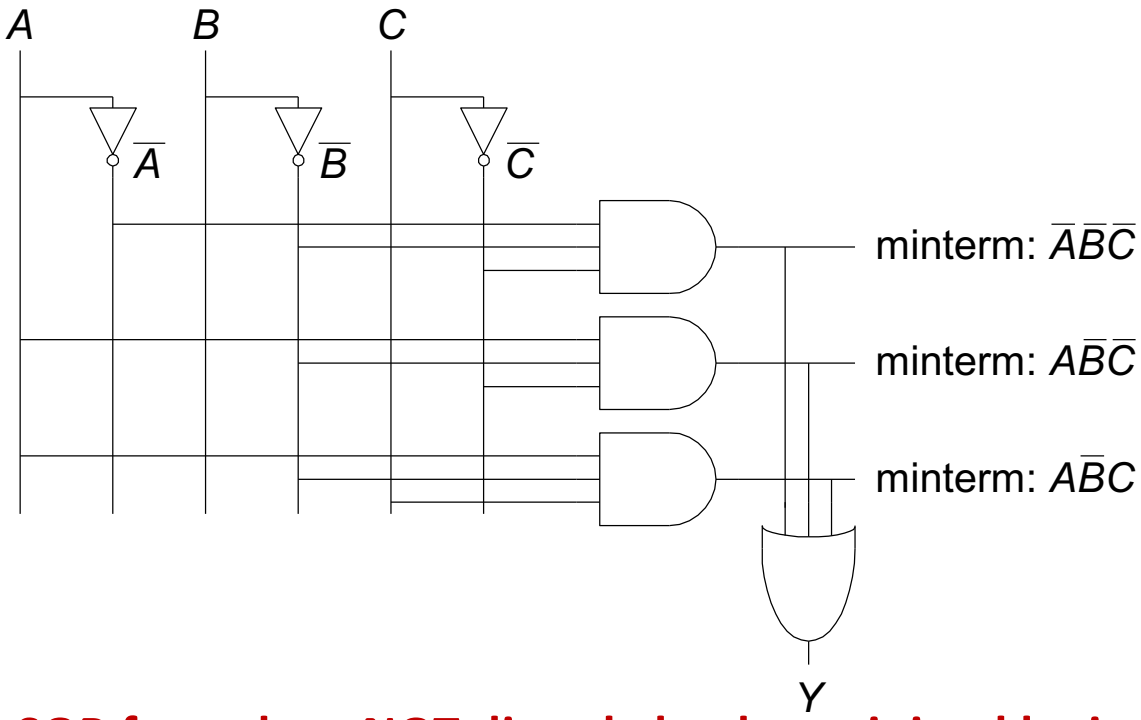
1. Inputs are on the left (or top) side
2. Outputs are on the right
3. Gates flow from left to right
4. Use straight wires
5. Wires connect at a T junction
6. A dot where wires cross indicates a connection



From Equation to Gates

- **Another example**

$$Y = \left(\overline{A} \cdot \overline{B} \cdot \overline{C}\right) + \left(A \cdot \overline{B} \cdot \overline{C}\right) + \left(A \cdot \overline{B} \cdot C\right)$$

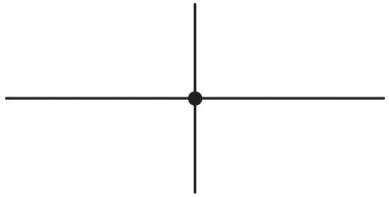


Key to remember: **SOP form does NOT directly lead to minimal logic (next lecture)**

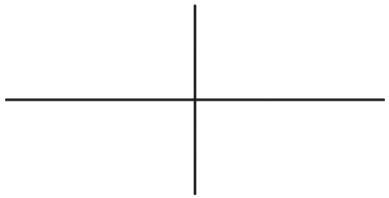
Rules for Connecting Wires

wires connect
at a T junction

wires connect
at a dot



wires crossing
without a dot do
not connect



Schematic: Happiness Detector

Specification: Mr. X is not happy if there is an assignment deadline, or their favorite bar is closed. Design a circuit that outputs 1 only if Mr. X is happy.

Truth Table

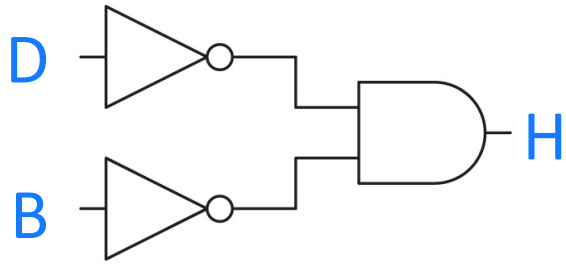
D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Eq

H = D'B'

H = (D)' AND (B)'

Logic Gate Implementation



Schematic: Happiness Detector

Specification: Mr. X is not happy if there is an assignment deadline, or their favorite bar is closed. Design a circuit that outputs 1 only if Mr. X is happy.

Truth Table

D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

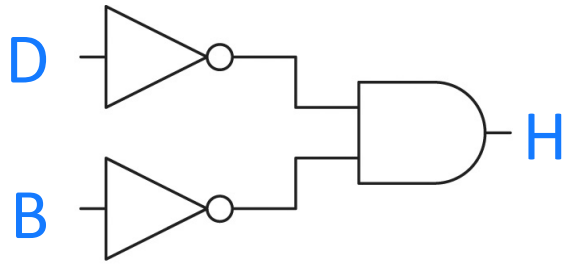
Which (monolithic) gate is this?

Boolean Eq

H = D'B'

H = (D)' AND (B)'

Logic Gate Implementation



Schematic: Happiness Detector

Specification: Mr. X is not happy if there is an assignment deadline, or their favorite bar is closed. Design a circuit that outputs 1 only if Mr. X is happy.

Truth Table

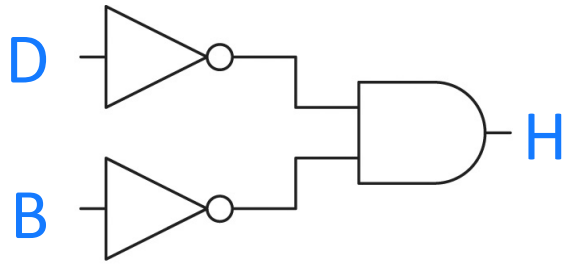
D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Eq

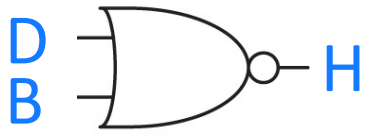
H = D'B'

H = (D)' AND (B)'

Logic Gate Implementation



Which (monolithic) gate is this? **Answer: NOR gate**



Schematic: Happiness Detector

Why does the happiness detector lack an OR gate in the two-level representation as a gate-level schematic?

Multiplexer: T. Table + Eq

Specification: Design a circuit with three inputs: D_0, D_1, select (S); and one output (Y). The output is D_0 if select is 0, and D_1 if select is 1.

$$Y = S'D_1'D_0 + S'D_1D_0 + SD_1D_0' + SD_1D_0$$

$$Y = S'D_0\underbrace{(D_1' + D_1)}_{=1} + SD_1\underbrace{(D_0' + D_0)}_{=1}$$

$$Y = S'D_0\,(1) + SD_1\,(1)$$

$$Y = S'D_0 + SD_1$$

Section 2.8.1 of H&H

Truth Table

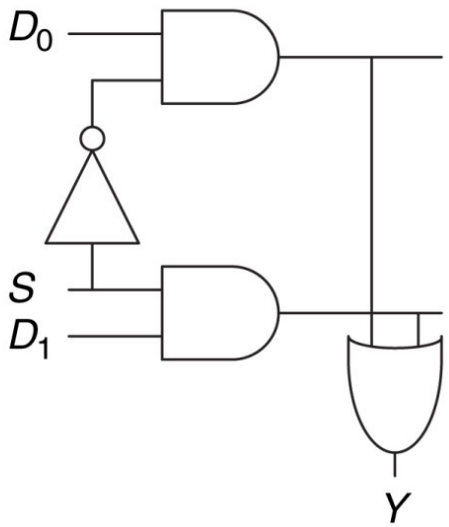
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Multiplexer: Gate-Level Schematic

Specification: Design a circuit with three inputs: D_0, D_1, select (S); and one output (Y). The output is D_0 if select is 0, and D_1 if select is 1.

$Y = S'D_0 + SD_1$

Gate-Level Schematic

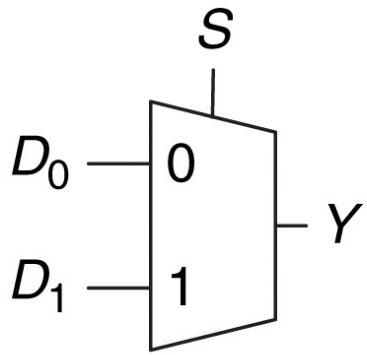


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2:1 Multiplexer (Mux)

- A 2:1 multiplexer (**mux**)
  - Two data inputs (D_0 and D_1)
  - Another input called the select signal
  - Output is either D_0 or D_1 depending on the value of select

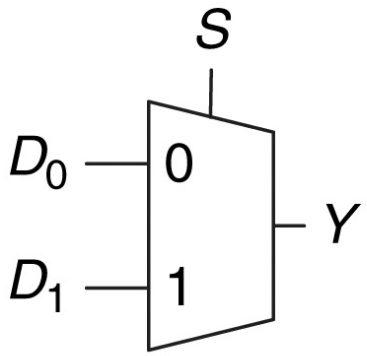
- We will use the high-level schematic for 2:1 mux and ignore the gate-level **implementation** details



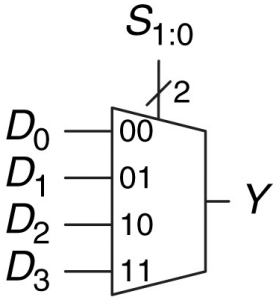
High-level Schematic

Multiplexer Applications

- Heavily used in **control** circuitry
  - Decision making
  - Which of the many ***competing outcomes*** to ***select***?
- Select one of the many signals and send it to another unit
- Think of if/else blocks in high-level programs



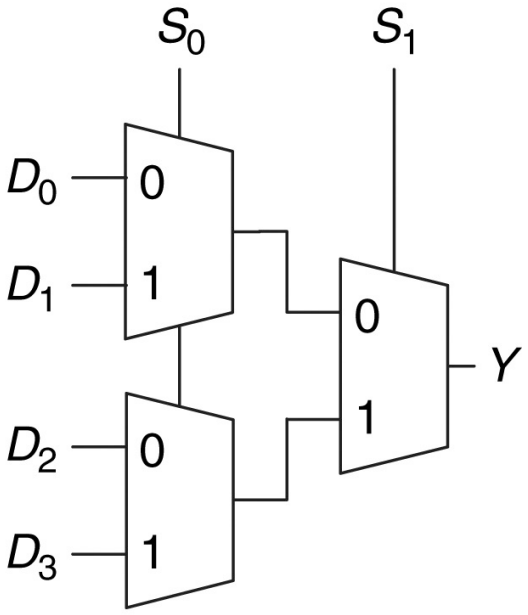
Wider (4:1) Multiplexer



- A 4:1 mux has two select signals S_0 and S_1
- A / and 2 implies a **bus width** of 2 to contrast with 1-bit wire or input
- One option is to construct the truth table and derive the Boolean equations
  - How many rows will there be in the table? (tedious!)
- Let's use intuition to build a 4:1 mux from two 2:1 multiplexers

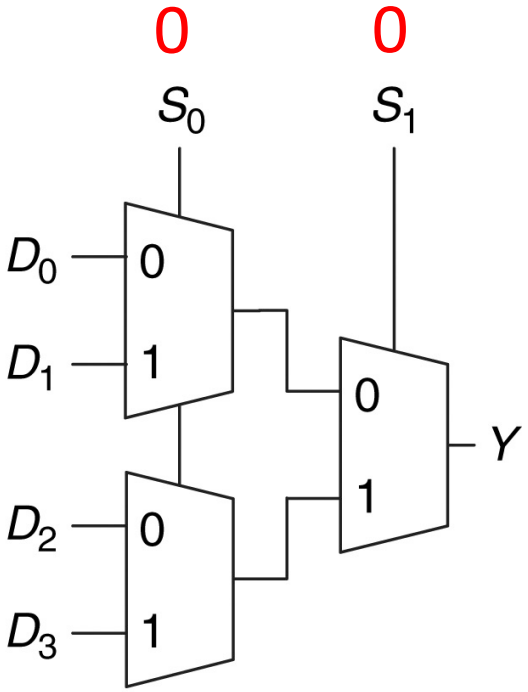
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



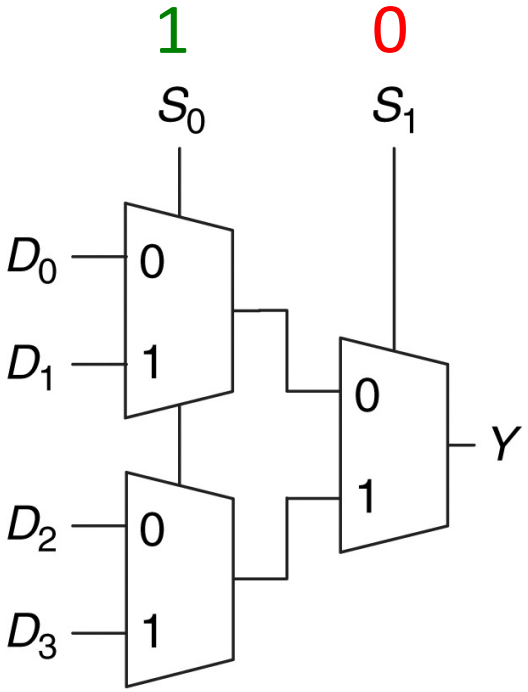
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



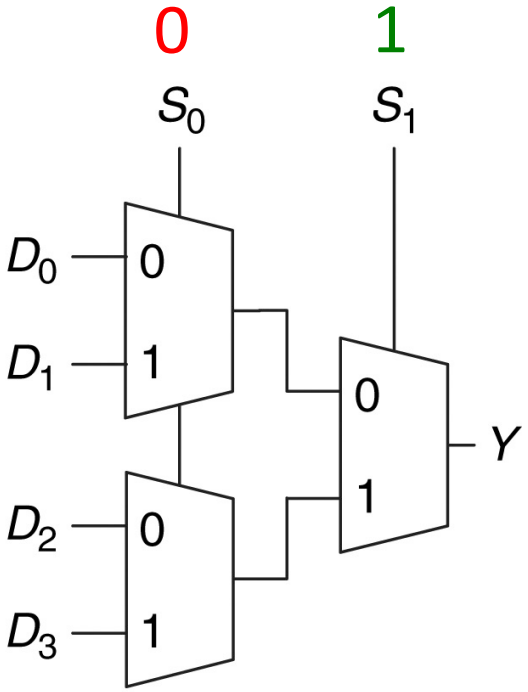
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



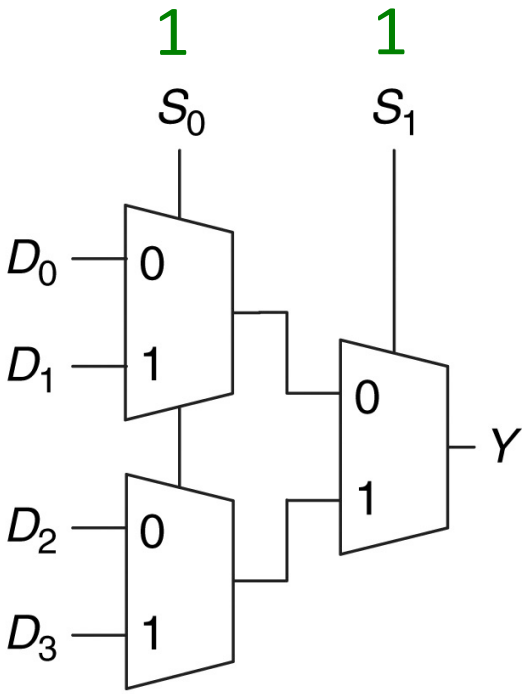
Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



Wider (4:1) Multiplexer

S_0	S_1	Y
0	0	D_0
1	0	D_1
0	1	D_2
1	1	D_3



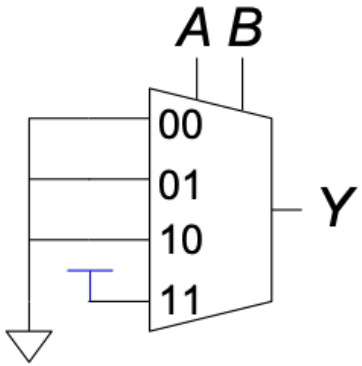
Logic Using Multiplexers

- Any truth table can be seen as a lookup table
  - Lookup 00, we see 0; Lookup 10, we see 1

- Multiplexers can be used as lookup tables to perform logic functions
  - Connect the data inputs to 0 or 1
  - Use inputs (A/B) as select lines

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

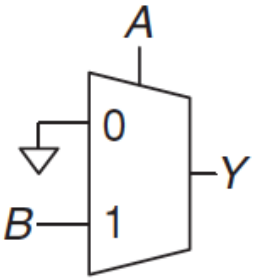
$Y = AB$



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	Y
0	0
1	B



Logic Using Multiplexers

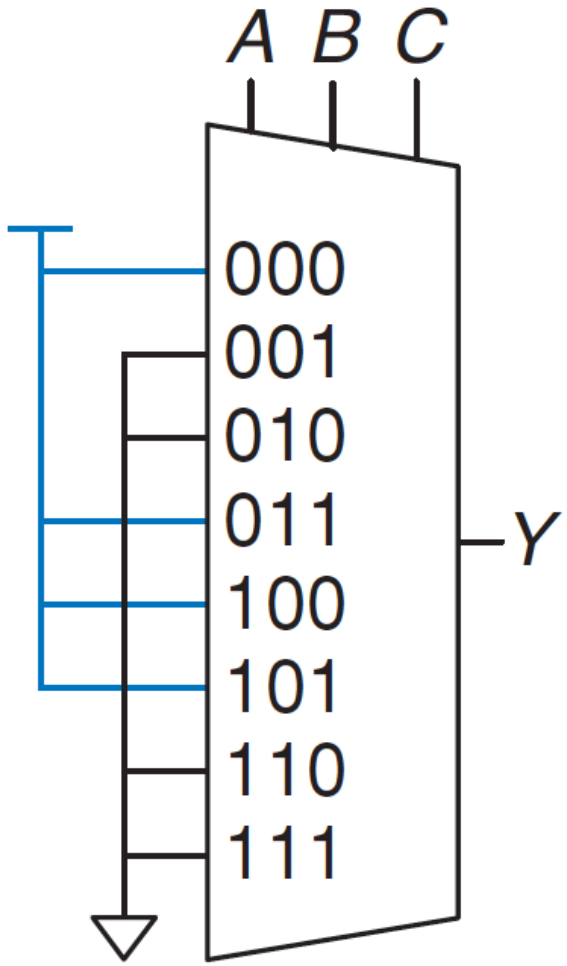
- Multiplexers can implement logic gate
  - For example, we can build a 2-input AND gate from a 2:1 multiplexer
- Can be (re)programmed to perform any N-input logic function
- Key idea: Connect multiplexer inputs to 0 (zero/ground) or 1 (high) by inspecting the truth table

A 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs

Multiplexer Logic: 3-Input Example

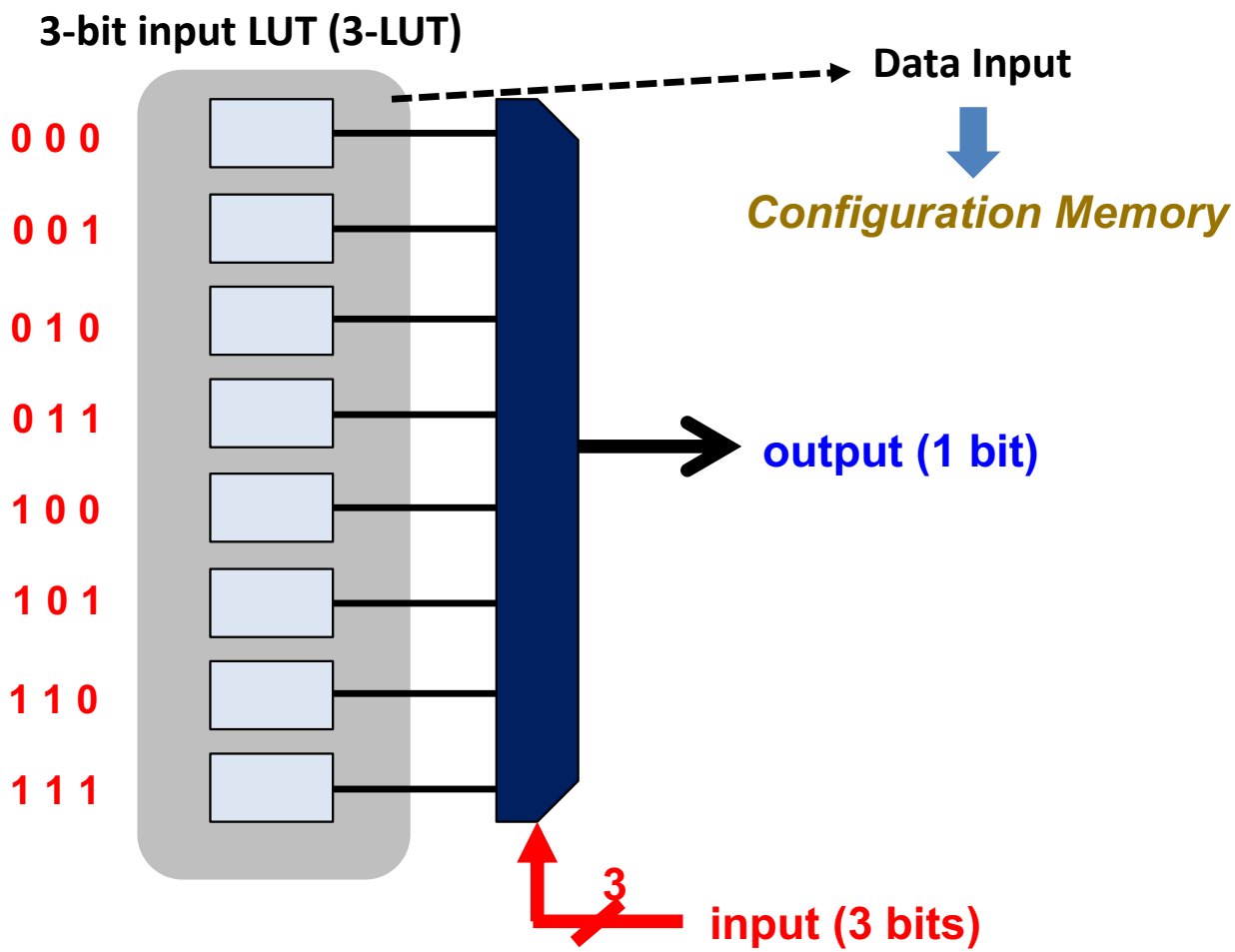
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$$Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$$



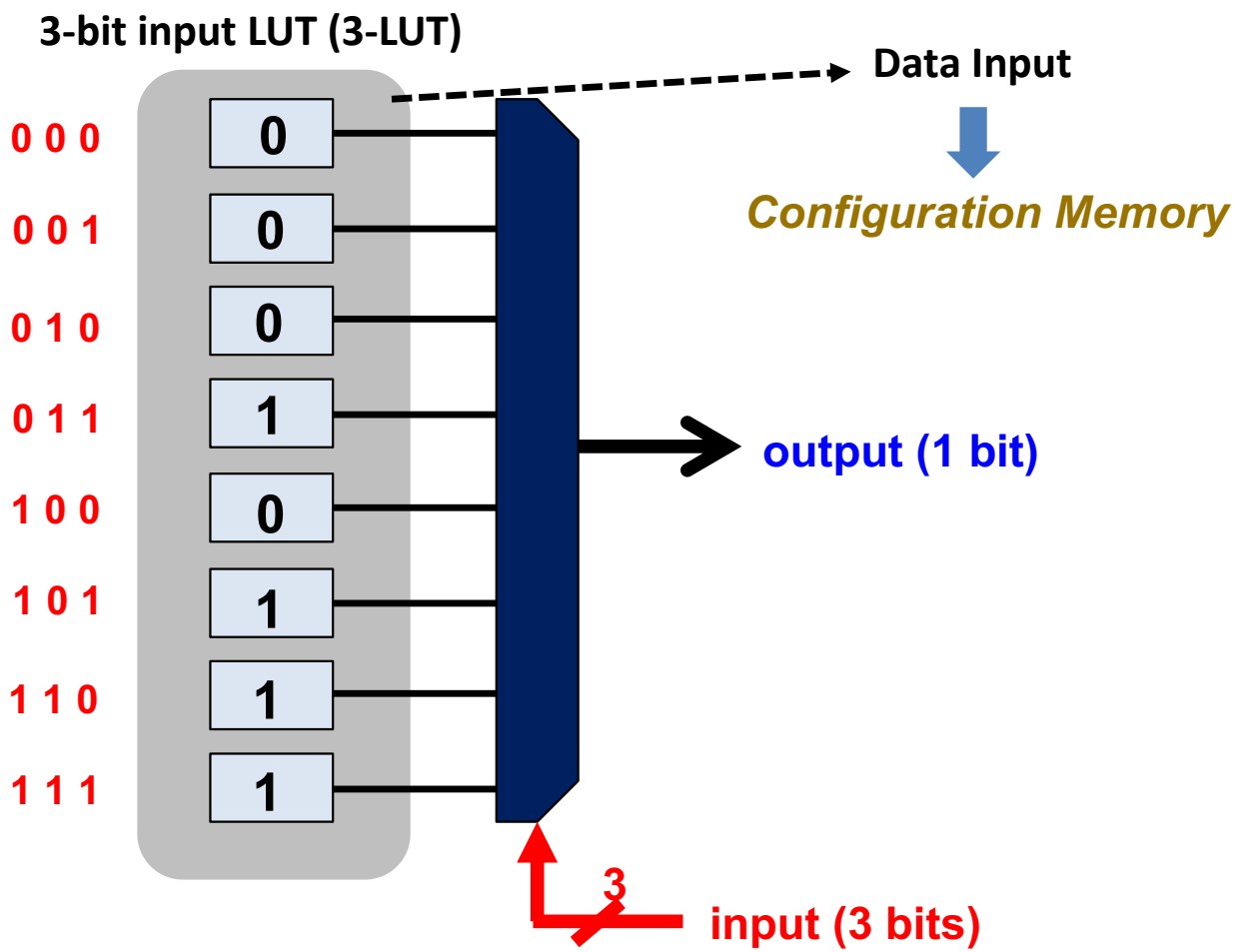
8-Input Lookup Table (LUT)

- 3-LUT can implement any 3-bit input function
  - Building block of **F**ield **P**rogrammable **G**ate **A**rray (**FPGA**)



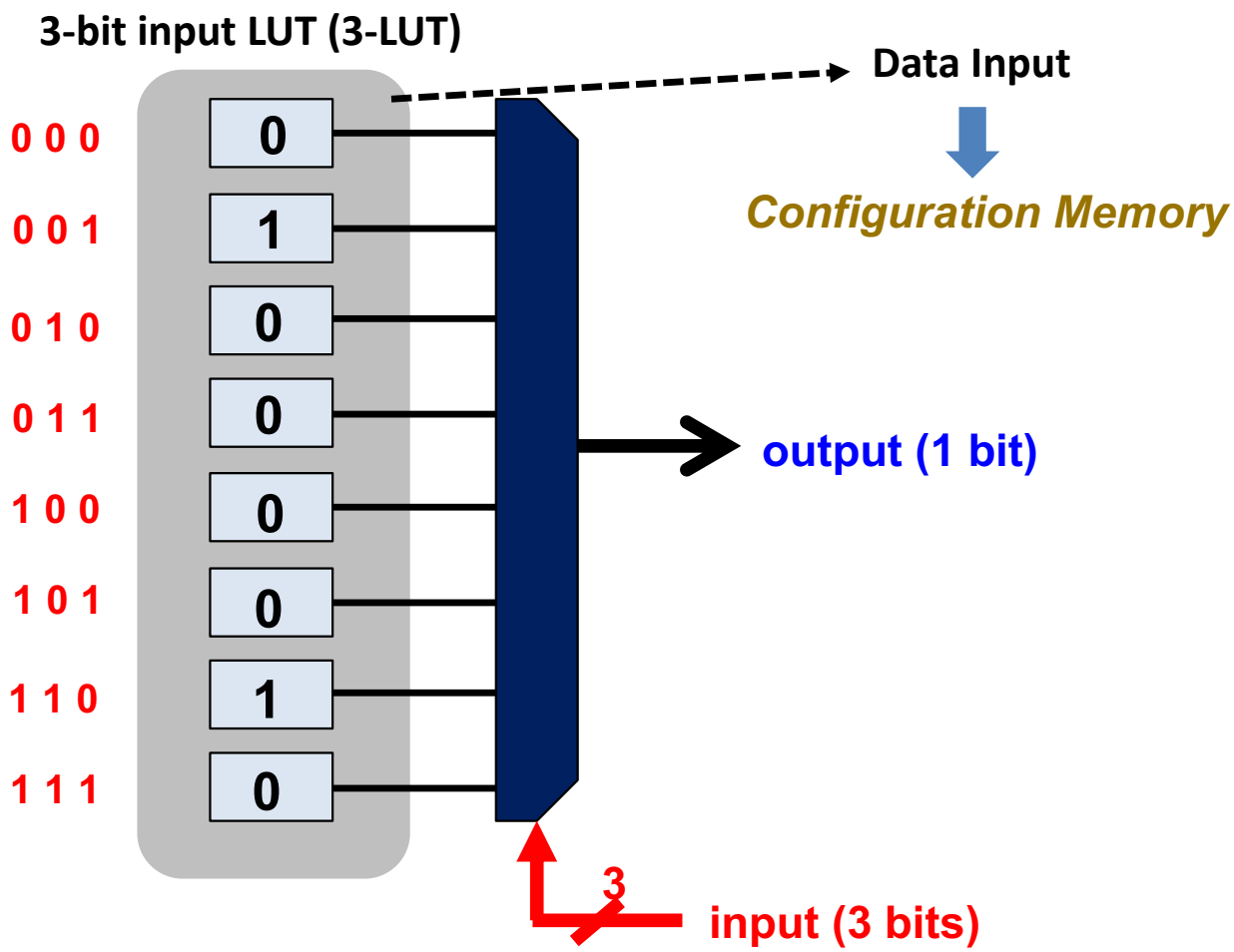
8-Input Lookup Table (LUT)

- 3-LUT can implement any 3-bit input function
  - Building block of field programmable gate array (**FPGA**)

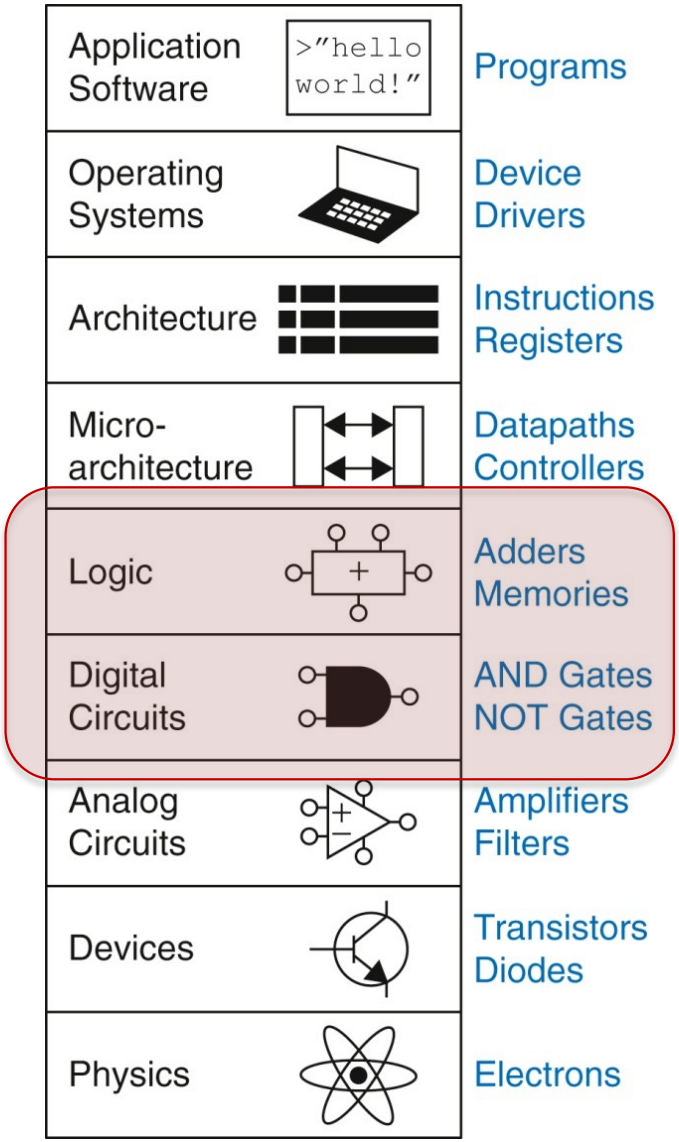


8-Input Lookup Table (LUT)

- 3-LUT can implement any 3-bit input function
  - Building block of field programmable gate array (**FPGA**)

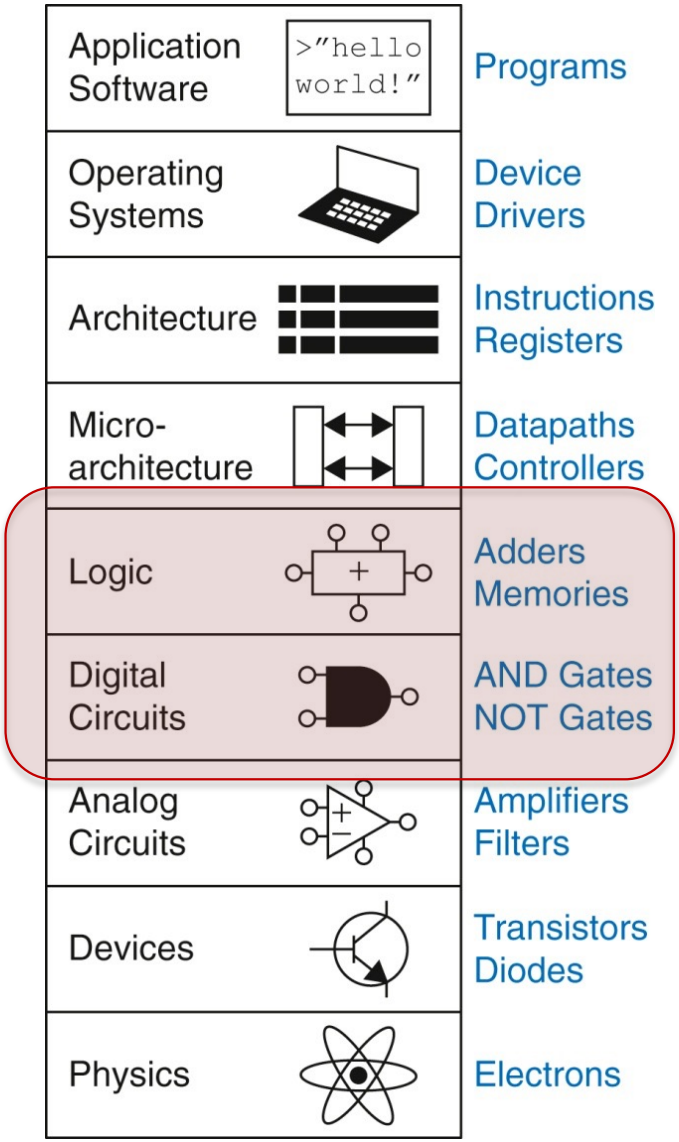


Friday Lecture



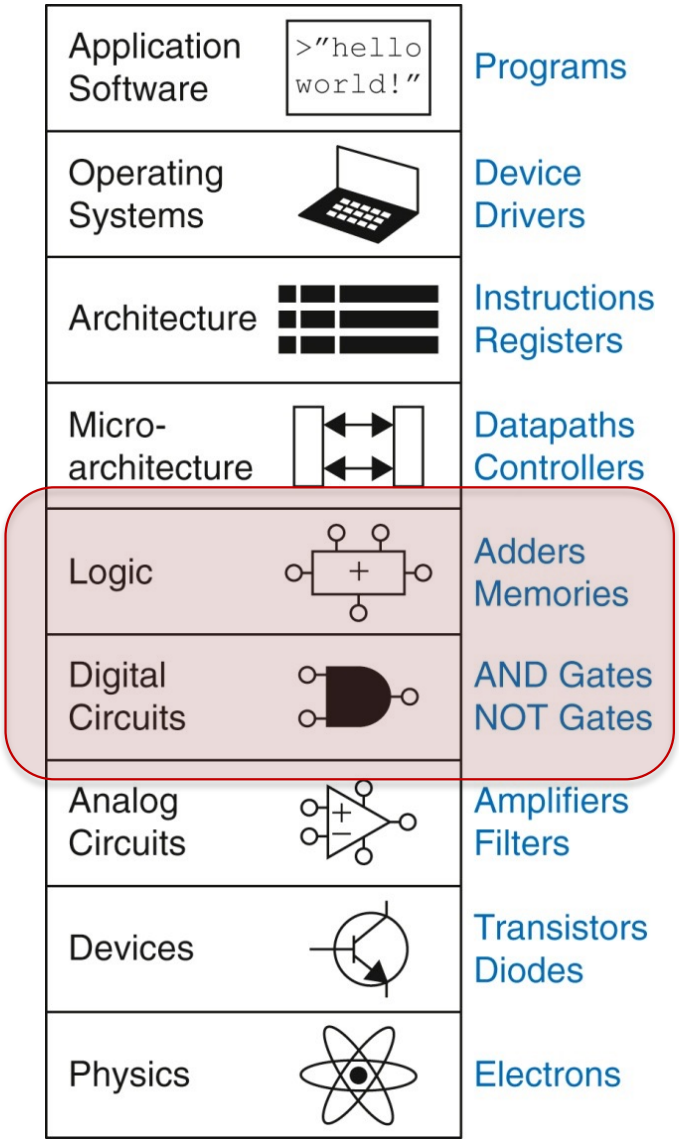
Topics Covered So Far

- Binary number system
- Transistor (basic building block)
- Logic gates
- Combinational circuits
  - English specification
  - Transformation to truth tables
  - Sum of Products (SOP)
  - Two-level implementation
- Multiplexers & lookup tables
- Half/Full adder



Today

- Revision of SOP
- More combinational circuits
  - Adders
  - ALU
  - Decoder
  - Comparator
  - PLA
  - Tri-state buffer
- Timing issues in combinational circuits
- Logic minimization with Boolean algebra



Sum of Products: Revision

- Find all the input combinations (**minterms**) for which the output of the function is TRUE.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C + AB\overline{C} + ABC$$

0 1 1 1 0 0 1 0 1 1 1 0 1 1 1

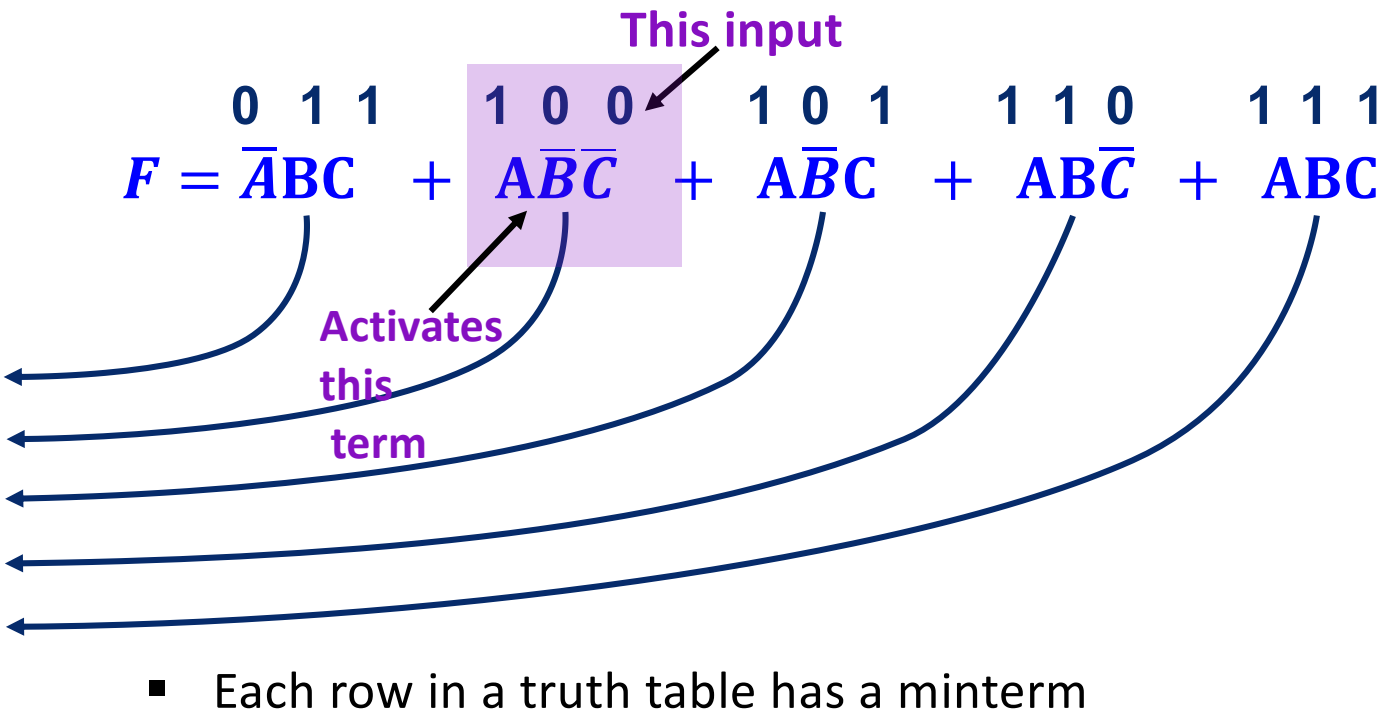
$$F = \sum(3, 4, 5, 6, 7)$$

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

Sum of Products: Revision

- Find all the input combinations (**minterms**) for which the output of the function is TRUE.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



$$F = \sum(3, 4, 5, 6, 7)$$

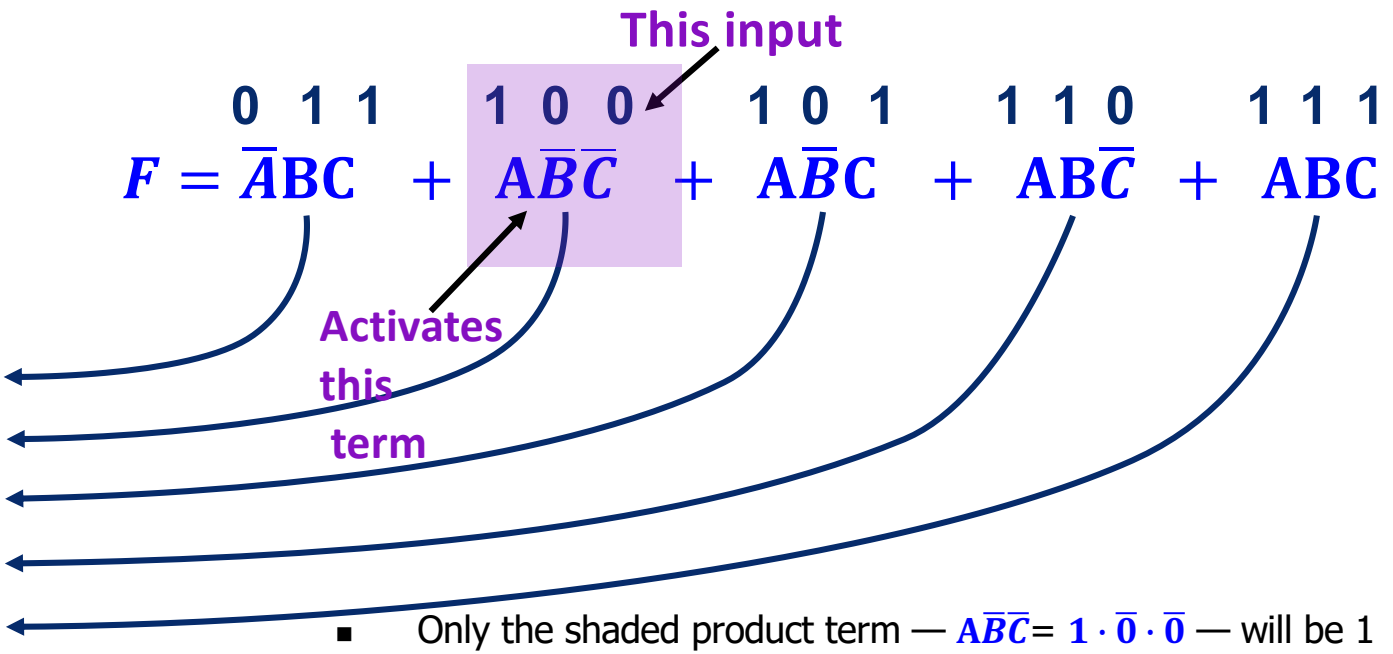
- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

Sum of Products: Revision

- Find all the input combinations (**minterms**) for which the output of the function is TRUE.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = \sum(3, 4, 5, 6, 7)$$



- Only the shaded product term — $A\overline{B}\overline{C} = 1 \cdot \overline{0} \cdot \overline{0}$ — will be 1
- No other product terms will "turn on" — they will all be 0
- So if inputs A B C correspond to a product term in expression,
  - We get 0 + 0 + … + 1 + … + 0 + 0 = 1 for output
- If inputs A B C do not correspond to any product term in expression
  - We get 0 + 0 + … + 0 = 0 for output

SOP in Sigma Notation

$$\mathbf{F} = \sum(\mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7})$$

- 3, 4, 5, 6, 7 denote the decimal equivalent of the binary number of the minterm designated in the truth table

Half Adder

Specification: Design a circuit that adds two binary variables: A and B.
The circuit has two outputs: sum and carry-out (C_{out}).

Truth Table

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

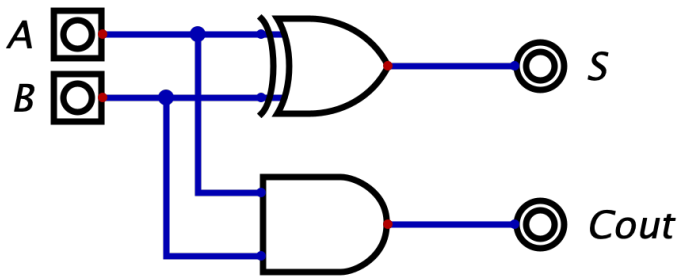
Boolean Eq

$S = A'B + AB'$

$S = A \oplus B$

$C_{out} = AB$

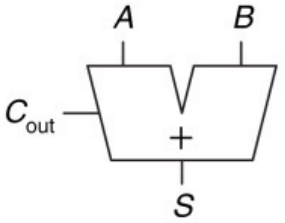
Schematic



Section 5.2.1 of H&H

Ripple Carry Adder

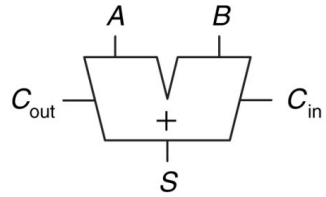
- **Limitation of half adder:** No *carry input*



- **Problem:** Adding multiple bits requires the need to add carry out from the previous column to the next column

```
   1
 1001
+0101
-----
 1110
```

- **Full adder** solves the problem
  - Accepts **three** inputs including a carry input
  - Signals flow from *right to left to reflect the carry propagation* in arithmetic circuits



Full Adder: T. Table + Eq

Specification: Design a circuit that adds two binary variables: A and B. The circuit has two outputs: sum and carry-out (C_{out}).

$S = C_{in}'A'B + C_{in}'AB' + C_{in}A'B' + C_{in}AB$

$C_{out} = C_{in}'AB + C_{in}A'B + C_{in}AB' + C_{in}AB$

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder: T. Table + Eq

Specification: Design a circuit that adds two binary variables: A and B. The circuit has two outputs: sum and carry-out (C_{out}).

$S = C_{in}'A'B + C_{in}'AB' + C_{in}A'B' + C_{in}AB$

$C_{out} = C_{in}'AB + C_{in}A'B + C_{in}AB' + C_{in}AB$

Insight about C_{out}

- 1 when both A and B are 1
  - Carry **G**eneration (**G**)
- 1 when there is a C_{in} and one of A and B is 1
  - Carry **P**ropagation (**P**)

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder: T. Table + Eq

Specification: Design a circuit that adds two binary variables: A and B. The circuit has two outputs: sum and carry-out (C_{out}).

$S = C_{in}'A'B + C_{in}'AB' + C_{in}A'B' + C_{in}AB$

$C_{out} = C_{in}'AB + C_{in}A'B + C_{in}AB' + C_{in}AB$

Simplification via Boolean algebra

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

$C_{out} = C_{in}(A \oplus B) + AB$

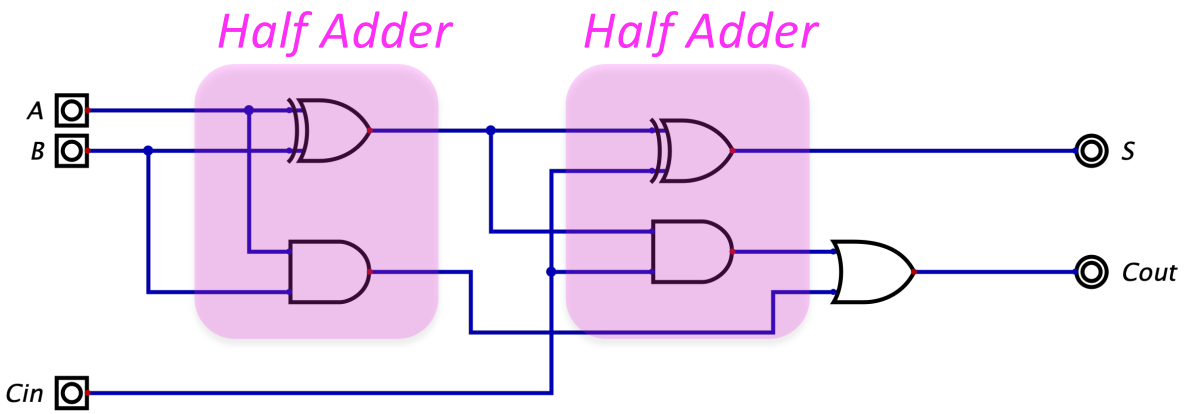
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder: Schematic

Specification: Design a circuit that adds two binary variables: A and B. The circuit has two outputs: sum and carry-out (C_{out}).

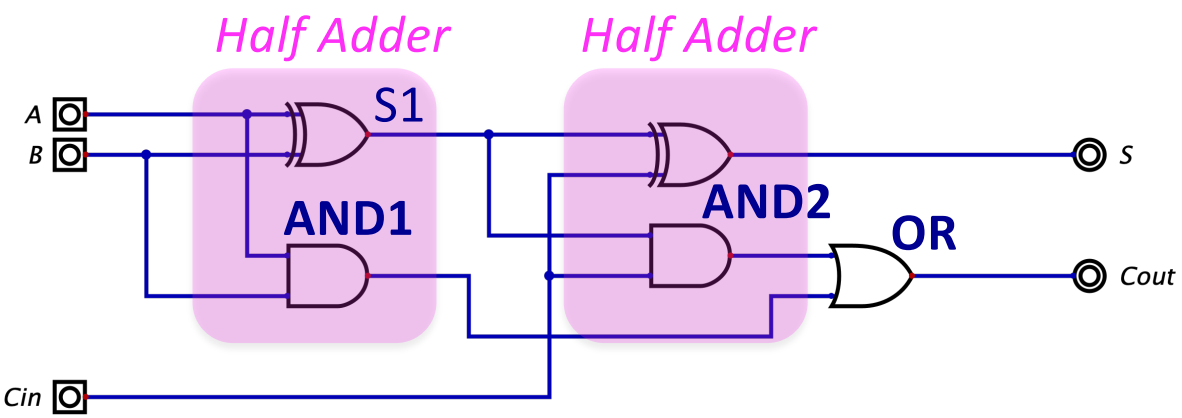
$S = A \oplus B \oplus C_{in}$

$C_{out} = C_{in}(A \oplus B) + AB$



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder = Two Half Adders



- What does **AND1** do?
  - Computes the carry out from A + B (call it S1)
- What does **AND2** do?
  - Computes the carry out from S1 + C_{in}
- What does **OR** do?
  - C_{out} is 1 if either the output of **AND1** is 1 or the output of **AND2** is 1
  - What does the truth table reveal about C_{out}?

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

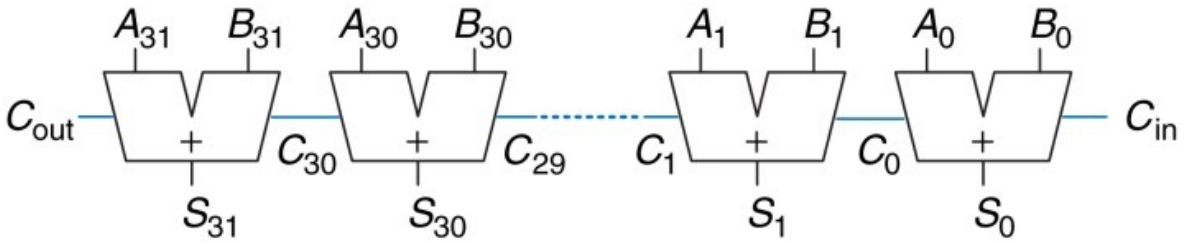
Ripple Carry Adder

- What if we want to add two N-bit numbers?

$$\begin{array}{rr} & 1 \\ & 1001 \\ + & 0101 \\ \hline & 1110 \end{array}$$

Ripple Carry Adder

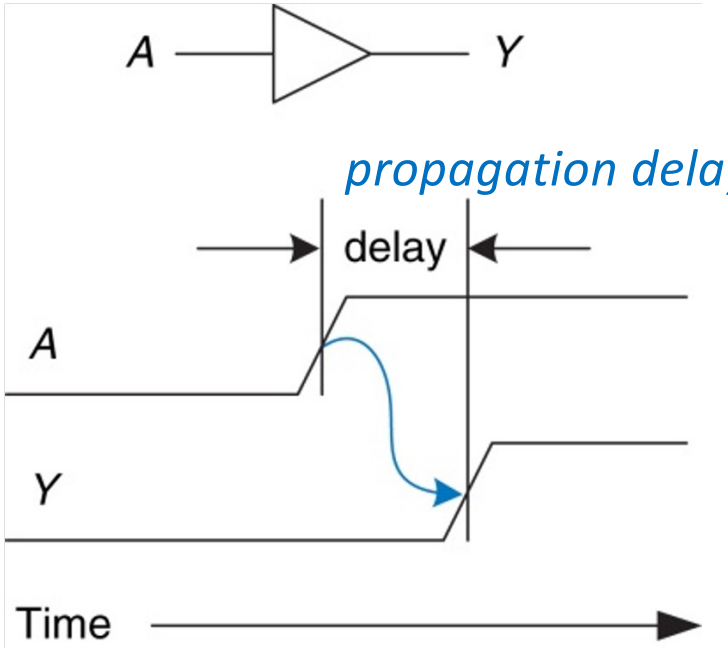
- What if we want to add two N-bit numbers?
  - Connect a chain of full adders from right to left



- **Ripple carry adder has a critical drawback!**

Timing in Combinational Circuits

- Every combinational circuit has a delay (seconds)
  - The time it takes for the *output to reach a final stable value* when the input **changes** (typically nanoseconds or picoseconds)



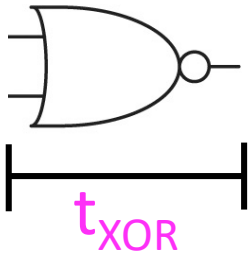
Section 2.9 of H&H

Examples

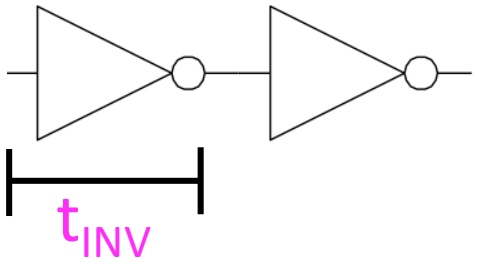
- **Inputs of the AND gate change from (0,0) to (1,1)**
  - Output of AND gate change from 0 to 1
  - How long does it take to for the output to change?

- **When A, B, and C_{in} are inputs to a full adder**
  - How long does it take to observe the final (and stable) S and C_{out}?

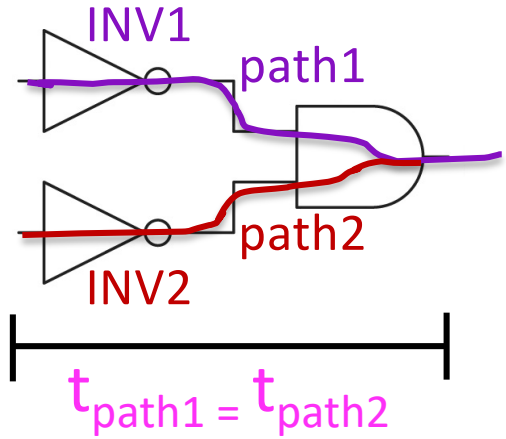
Examples of Timing/Delay



Each gate has a delay



Chain of gates:
Sum the delay of each gate in the chain 2 X t_{INV}



Multiple paths from input to output

$t_{path1} = t_{INV1} + t_{AND}$

$t_{path2} = t_{INV2} + t_{AND}$

Critical and Shortest Path

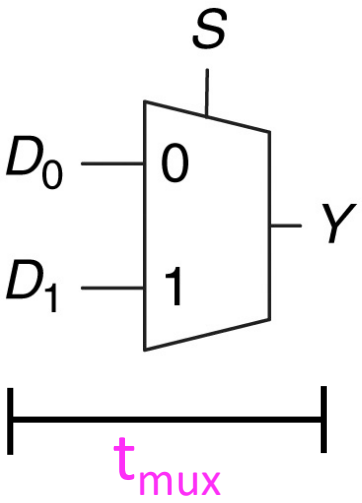
- Most useful combinational circuits have multiple paths from input to output
  - **Critical path:** The slowest path (with **longest delay**)
  - Critical path limits the speed at which the circuit operates
  - In contrast, the shortest path is the fastest

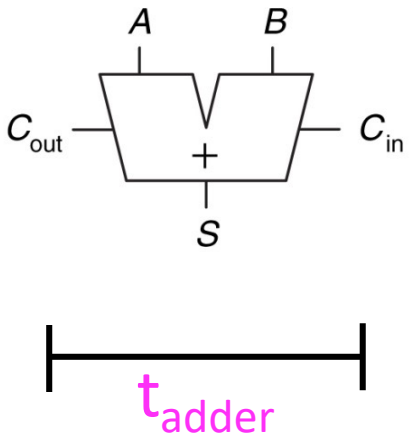
- For simplification, we will ignore the delay of nodes (wires)
  - *Although the delay is non-trivial, it is studied best at the analog level of abstraction*

Section 2.9 of H&H

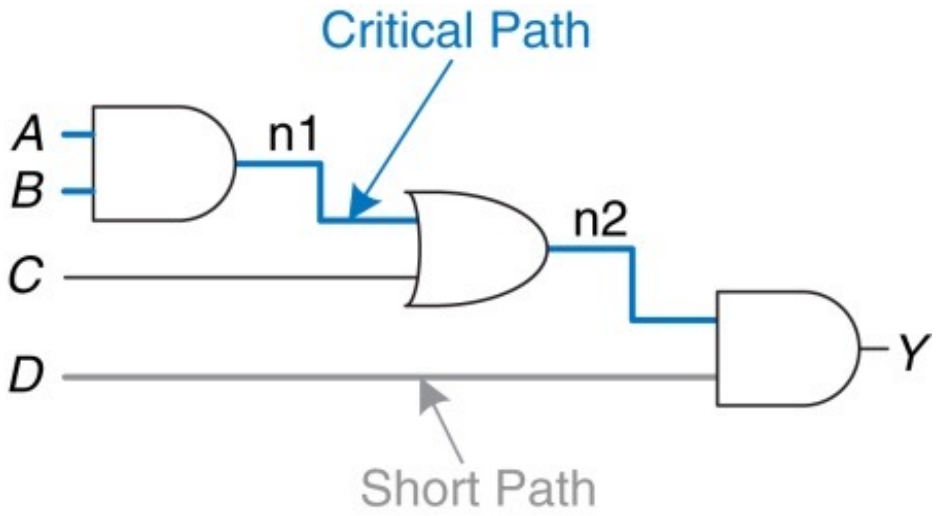
Multiplexer and Adder Delay

- Assume component-level delay and don't worry about delay of individual gates





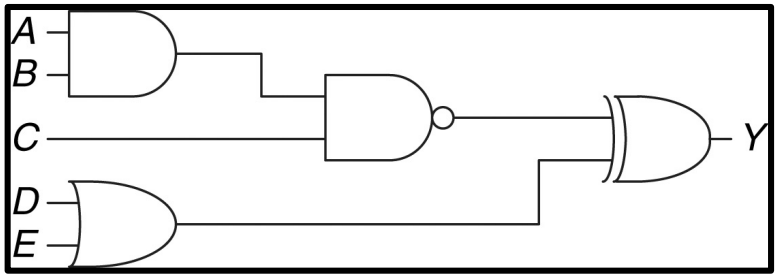
Example (1)



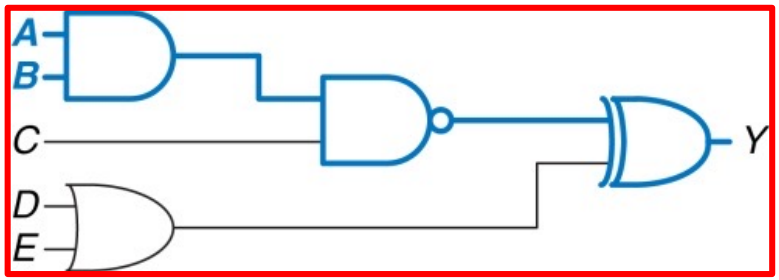
The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path

Example (2)

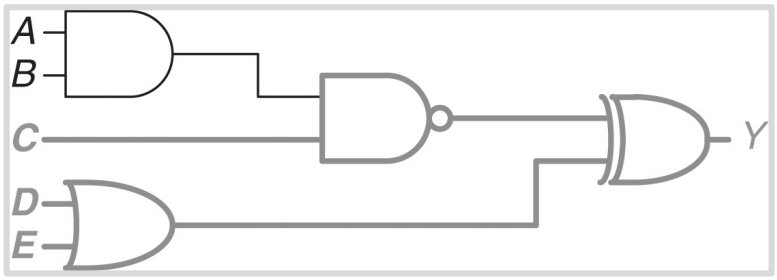
Example Circuit



Critical Path

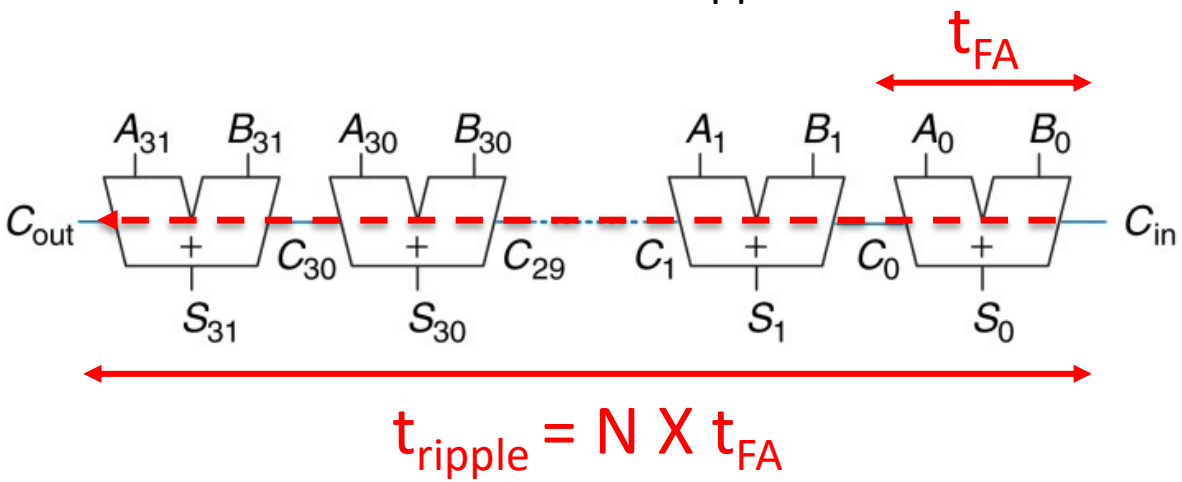


Shortest Path



Drawback: Ripple Carry Adder

- If we abstract the delay of full adder as t_{FA}, then what is the delay of the ripple carry adder, t_{ripple}?



- The critical path consists of N full adders (slow when N is large)
  - The critical path runs through the chain of full adders
  - Every full adder is on the critical path

Section 5.2.1 of H&H

Carry-Lookahead Adder

- **Motivation:** When the delay of a circuit grows with the number of input bits, the design is not scalable
  - *We try to find a way to optimize the circuit to reduce the delay*

- Ideally, we want circuits that *take constant time regardless of the input size*

- **Optimization:** We try to optimize the circuit using intuition/insight and keep the delay reasonable
  - There is a tradeoff (nothing is free ....

Section 5.2.1 of H&H

Carry-Lookahead Adder (CLA)

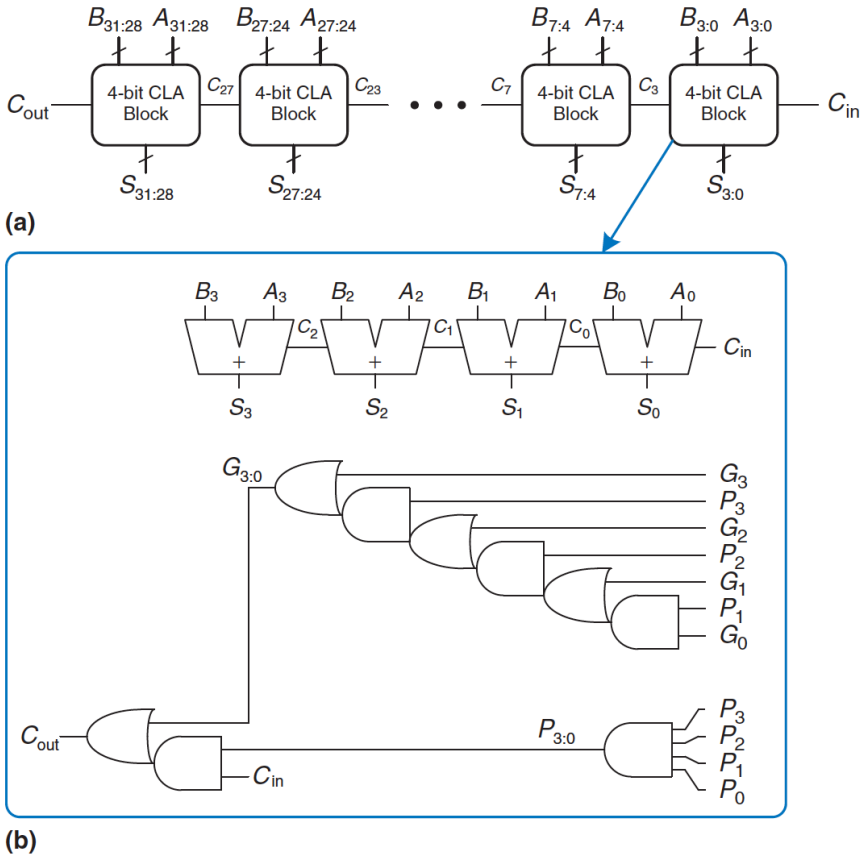
- Another one in the class of **carry propagate adders** that **accelerates** carry generation

- **Insight of CLA:** As soon as C_{in} is known, C_{out} for an k-bit ripple carry adder can be calculated

- When do we have a carry out from a column?
  - A = 1 AND B = 1, C_{out} is 1 → Carry **G**eneration
  - C_{in} = 1, A = 1 OR B = 1, C_{out} is 1 → Carry **P**ropagation

Optional study: Section 5.2.1 of H&H

CLA Design



Specialized logic for fast carry generation

Optional study: Section 5.2.1 of H&H

Things to Consider

- Each CLA block is busy generating a carry for the next block simultaneously (i.e., in parallel)

- **Is there still a bottleneck in the design?**
  - What is the propagation delay of an N-bit carry lookahead adder?

Lessons from CLA

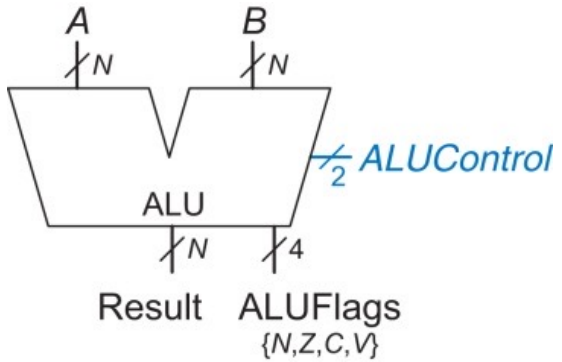
- **Speed-Area Tradeoff:** In digital systems, there is a tradeoff b/w performance (speed) and hardware cost (area/power)
  - CLA speeds up addition, but requires extra logic gates that take up additional area and dissipate more power

- **Logic Specialization:** Logic specialization for frequently used but slow circuits/tasks is often necessary
  - CLA uses **specialized logic** for fast carry generation

Arithmetic and Logic Unit (ALU)

- The circuits we have looked so far can do one useful thing
  - XNOR gate **performs** equality testing
  - Adder **performs** addition
  - Multiplexer **performs** selection

- ALU is our first **general purpose** circuit
  - Performs a variety of arithmetic/logical operations
  - ADD, SUB, AND, OR, XOR, ...



N-bit ALU

- It has a 2-bit control input
  - The language ALU speaks or the instructions it understands

Section 5.2.4 of H&H

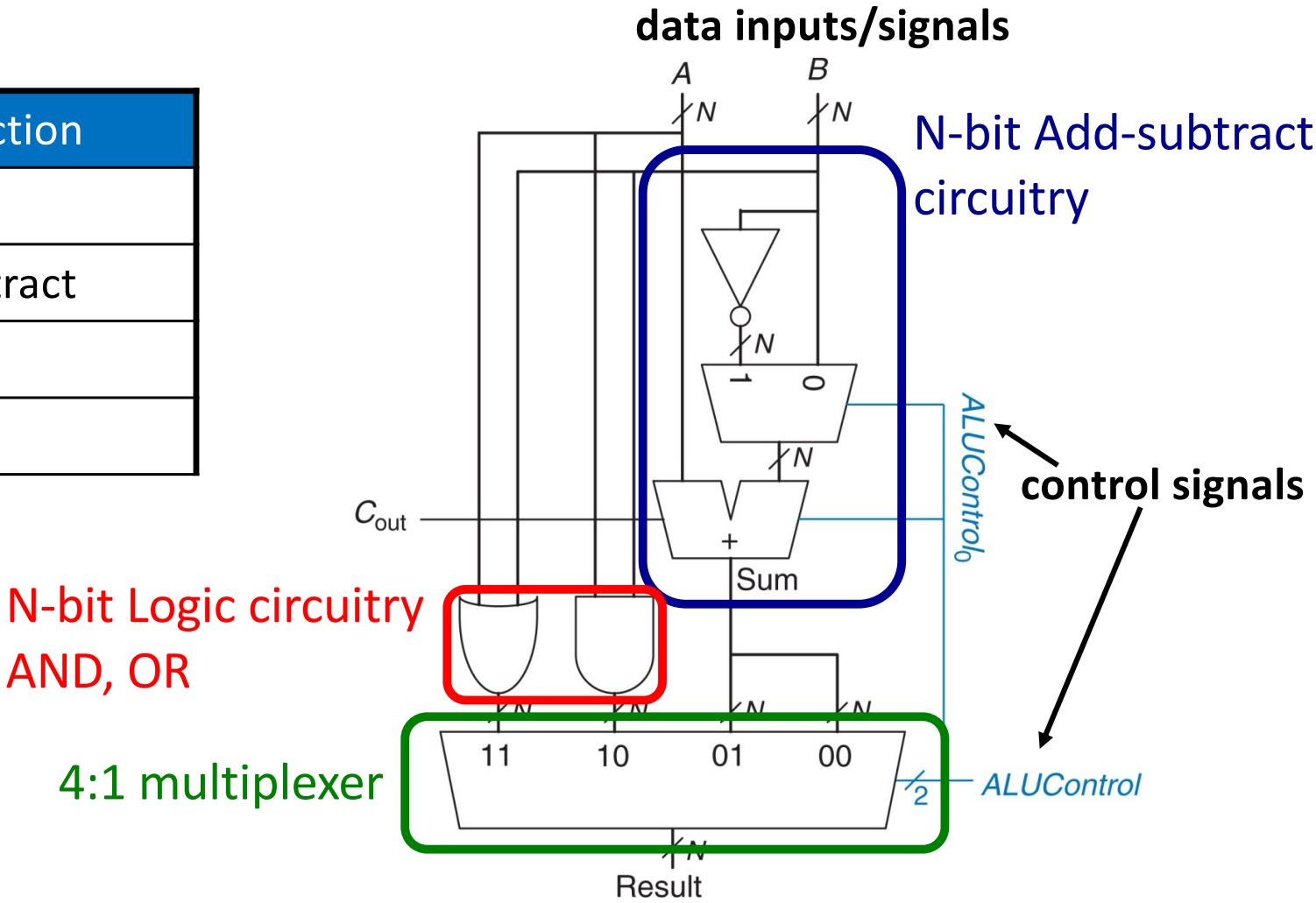
ALU Interface/Instructions

- N-bit data inputs and outputs
- 2-bit control input (ALUControl)
  - Specifies one of four function
  - Setting ALUControl to 00, 01, 10, and 11 is giving ALU **instructions**
- The assignment of binary codes to ALU functions is **not** arbitrary
  - It is clever (01 for Subtract in particular) as we will reveal

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

ALU Implementation

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR



Add-Subtract Circuitry

- A + B
  - Normal addition

- A – B
  - A + (–B)
  - In 2's complement, –B = B' + 1
  - An inverter performs B'
  - We send $ALUControl_0$ as the carry input of the adder
  - $ALUControl_0$ is 1 when the ALU function is Subtract

The Nature of Hardware

- **Parallelism: Hardware is inherently parallel**
  - All logic gates in the ALU work in parallel when the circuit is presented with valid input
- **Redundancy: Generality leads to redundancy**
  - ALU is a general-purpose circuit that can perform a variety of operations. Some work/effort is wasted
  - The output of **OR/AND** is wasted when ALUControl is 01
- **Control: Control circuitry comes with a cost**
  - ALU consumes more area than the individual functional units it combines (4:1 multiplexer is for controlling output)

ALUFLAGS N Z C V

- We need meta-information about the ALU result
  - Is the result negative (N)?
  - Is the result zero (Z)?
  - Is there a carry out (C)?
  - Is there an overflow (V)?

- Many scientific algorithms rely on flags for the **next step**
  - If overflow: discard result, and redo
  - Carry out is the carry in for another operation
  - If the result is negative: do {...}; else do {...}

Flags are only relevant for arithmetic operations
($ALUControl_1 = 0$)

ALUFLAGS N Z C V

- Negative
  - Check the MSB of result
- Zero
  - NOR all bits of the result (same as invert then AND)
- Carry
  - AND $ALUControl_1$ with C_{out} from the adder
- Overflow
  - **Option # 1:** Use A and B to compute overflow
  - **Option # 2:** Use A and the output of 2:1 multiplexer to compute overflow

Option # 1 for Overflow

- The following scenarios generate overflow: overflow flag is 1

	$ALControl_0$	A_{31}	B_{31}	S_{31}
Scenario # 1	0 (Add)	0	0	1
Scenario # 2	0 (Add)	1	1	0
Scenario # 3	1 (Subtract)	0	1	1
Scenario # 4	1 (Subtract)	1	0	0

Case # 1 in plain English: When doing **A + B** , if A and B are +ve, and the sum is –ve

Case # 2: A + B, if A and B are –ve, and the sum is +ve

Case # 3: A – B, if A is +ve and and B is –ve, and the sum is –ve

Case # 4: A – B, if A is –ve and and B is +ve, and the sum is +ve

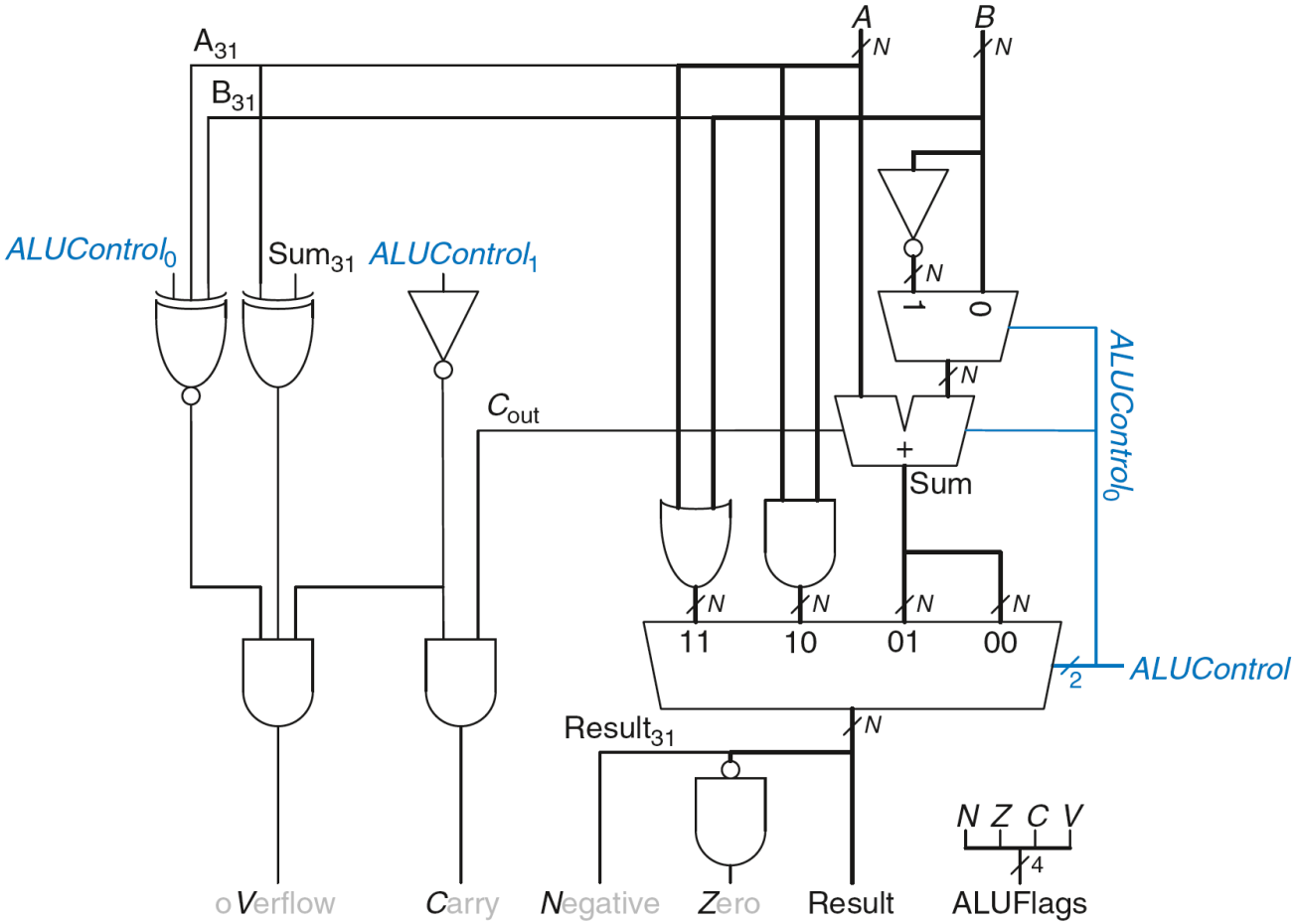
Option # 1 for Overflow

- The following scenarios generate overflow: overflow flag is 1

	$ALControl_0$	A_{31}	B_{31}	S_{31}
Scenario # 1	0 (Add)	0	0	1
Scenario # 2	0 (Add)	1	1	0
Scenario # 3	1 (Subtract)	0	1	1
Scenario # 4	1 (Subtract)	1	0	0

- Overflow is 1 whenever there is an even number of 1's among $ALUControl_0$, A_{31}, and B_{31}
  - XNOR $ALUControl_0$, A_{31}, and B_{31}
- Overflow is 1 whenever A_{31} and S_{31} are different
  - XOR A_{31} and S_{31}

Option # 1 for Overflow



Option # 2

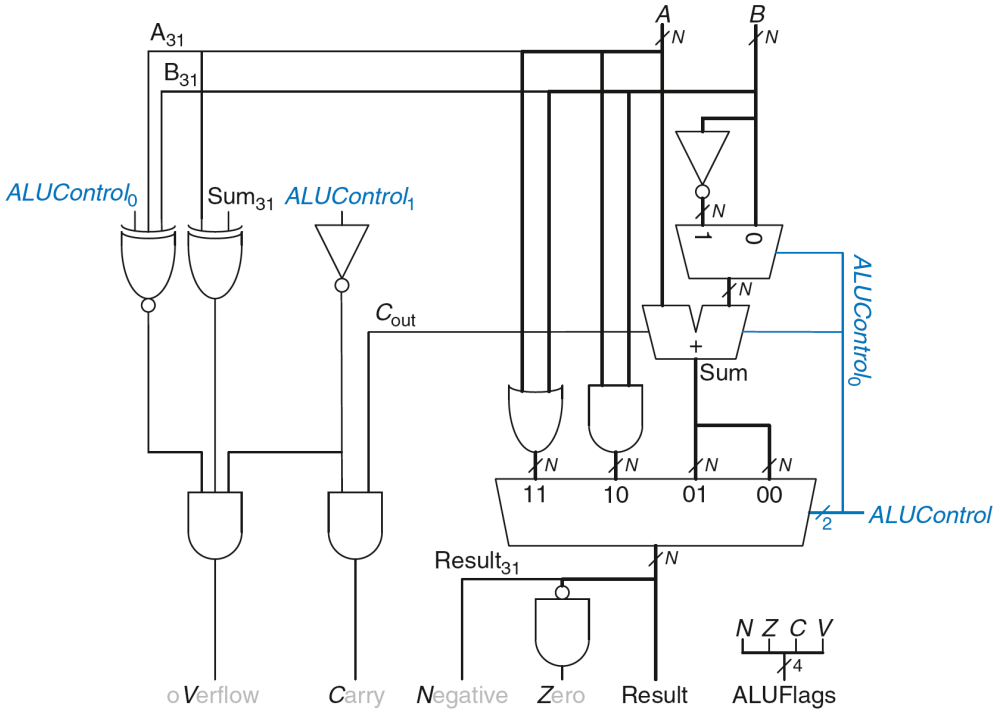
- Use A and the output of 2:1 mux
  - B if the instruction is an Add and –B if the instruction is a subtract
- Easy to reason conceptually
  - If A – B is the same as A + (–B) then everything is an add
  - There is no need to consider subtract separately when reasoning about overflow generation
- The circuitry is also much simpler
  - **Homework assignment:** Figure out the circuitry for overflow generation with option # 2

ALU Timing Analysis

Homework

picoseconds (10^{-12} seconds) = ps

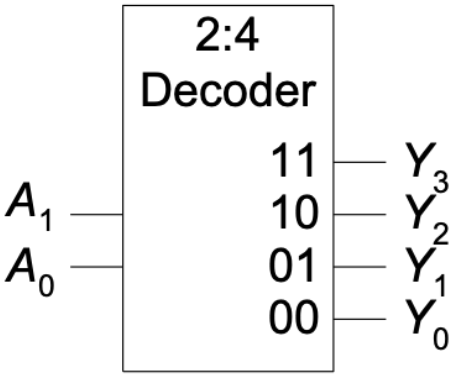
Element	Delay
Inverter	t_{INV} = 1 ps
2:1 Mux	t_{mux2} = 5 ps
4:1 Mux	t_{mux4} = 8 ps
Adder	t_{adder} = 14 ps
AND	t_{AND} = 2 ps
OR	t_{OR} = 2 ps



- Find t_{Result} in ps for the four ALU instructions/functions. Ignore overflow generation
  - Which function takes the longest time (and is the critical path)? Ignore wire delay
- Express t_{Result} in the form of an equation for Add and Subtract. What is the difference?

Decoders

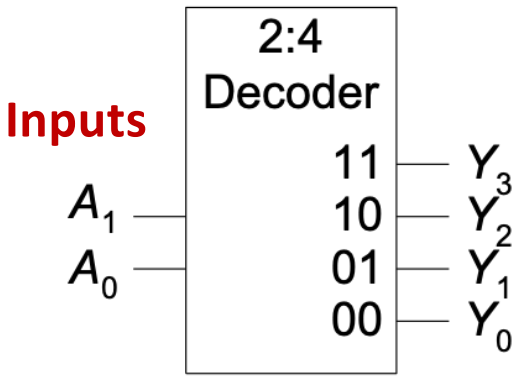
- N inputs and 2^N outputs
- For each input combination, only one of the outputs is 1
  - The outputs are affectionately called one-hot



Section 2.8.2 of H&H

Decoders

- N inputs and 2^N outputs
- For each input combination, only one of the outputs is 1
  - The outputs are affectionately called one-hot



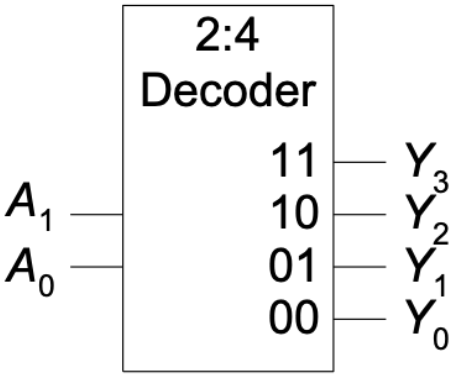
2:4 Decoder Truth Table

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Outputs

Decoders

- N inputs and 2^N outputs
- For each input combination, only one of the outputs is 1
  - The outputs are affectionately called one-hot



2:4 Decoder Truth Table and Boolean Equations

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$Y_0 = A_1'A_0'$

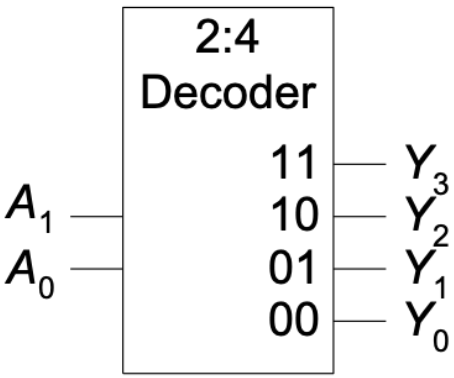
$Y_1 = A_1'A_0$

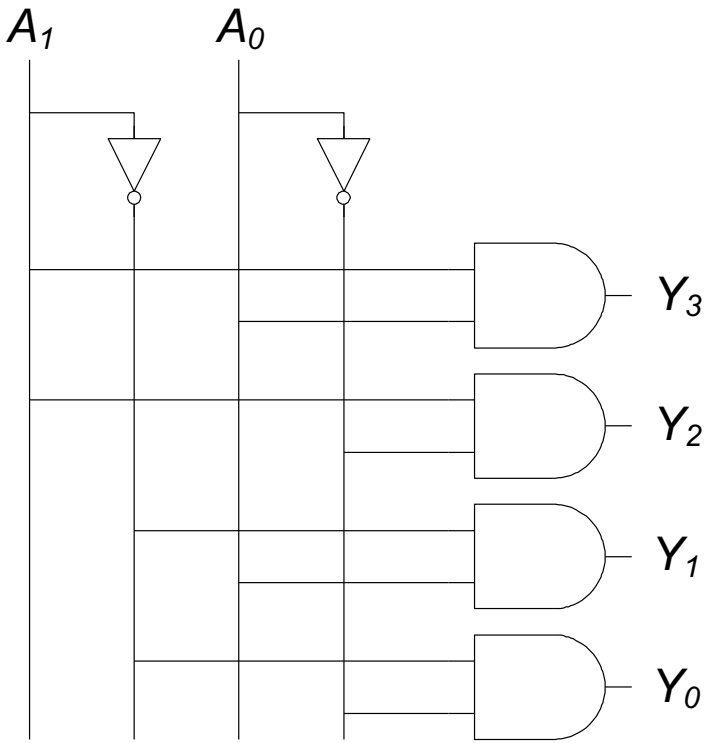
$Y_2 = A_1A_0'$

$Y_3 = A_1A_0$

Decoders

- N inputs and 2^N outputs
- For each input combination, only one of the outputs is 1





$Y_0 = A_1'A_0'$

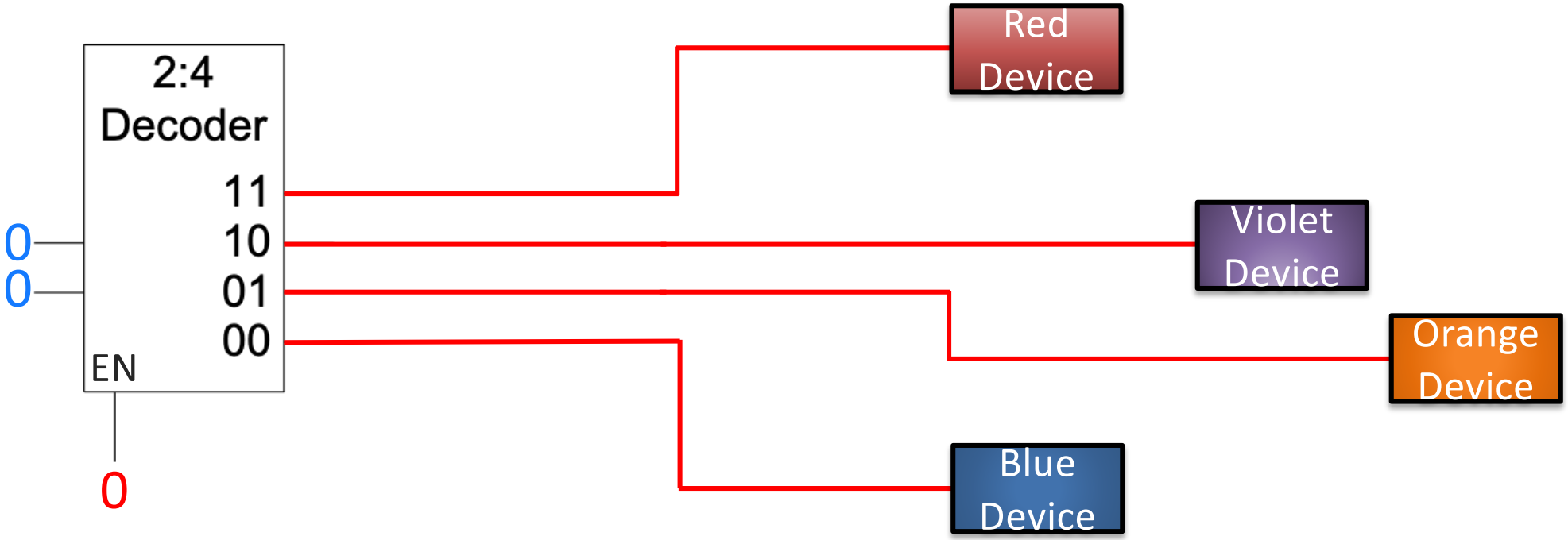
$Y_1 = A_1'A_0$

$Y_2 = A_1A_0'$

$Y_3 = A_1A_0$

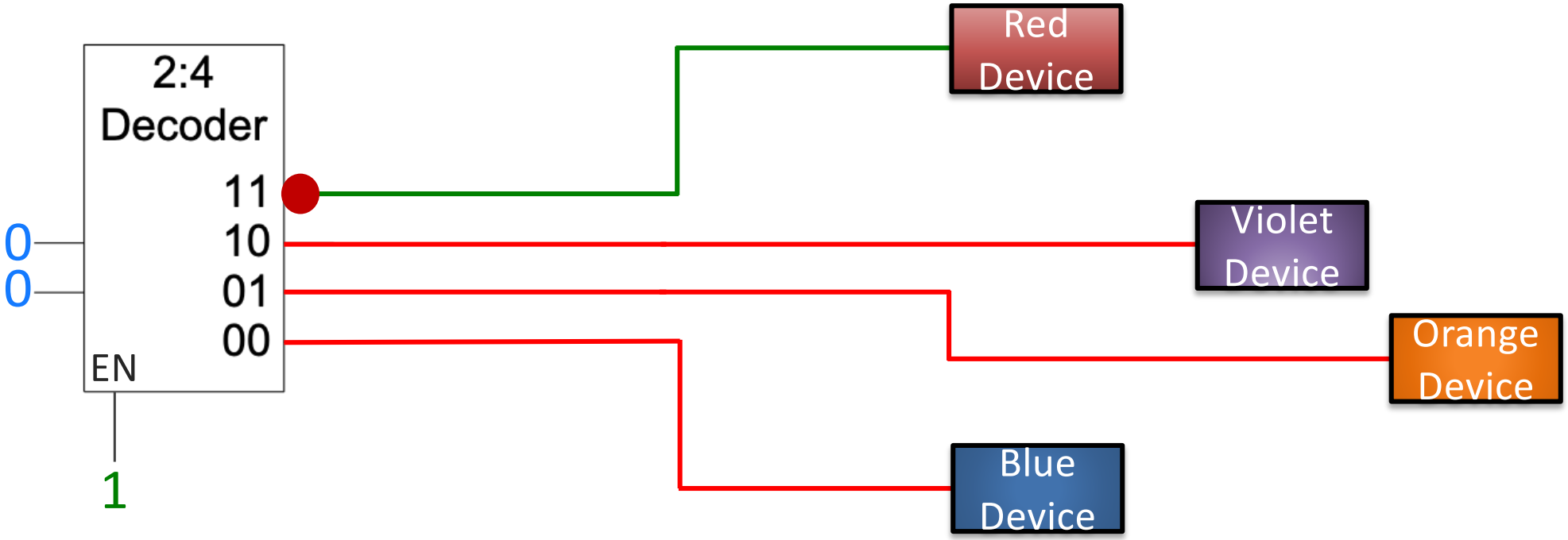
Uses of Decoders

- For each input combination, only one of the outputs is 1



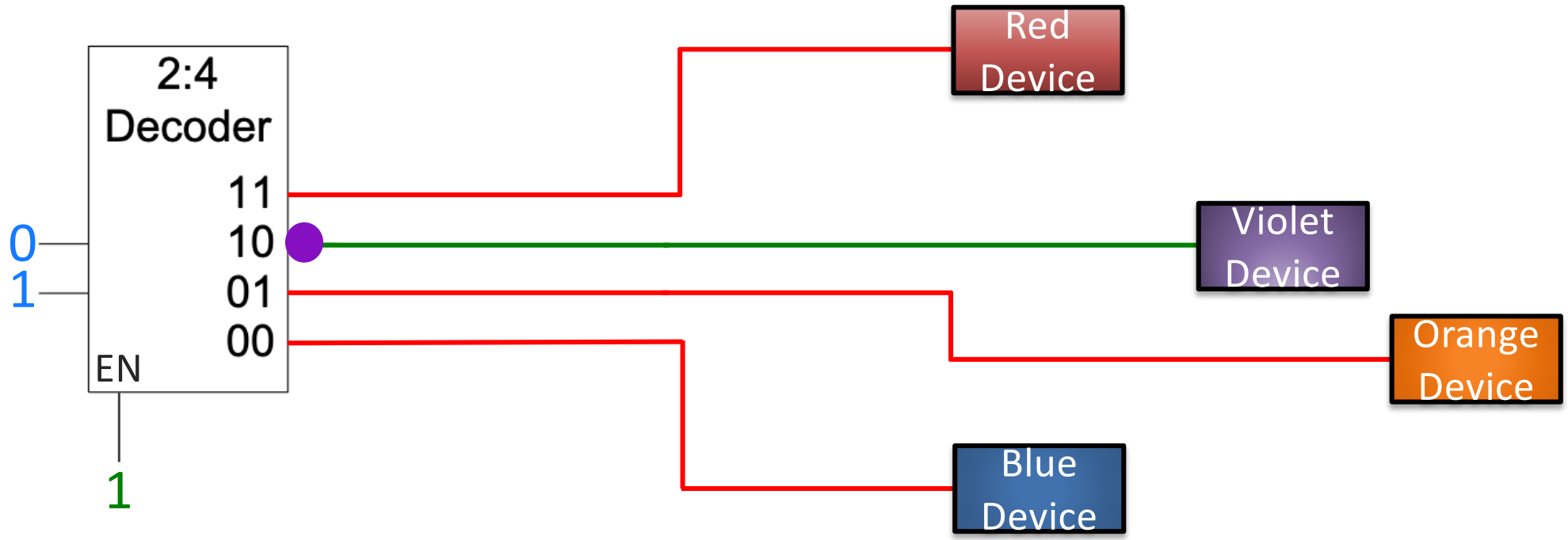
Uses of Decoders

- For each input combination, only one of the outputs is 1



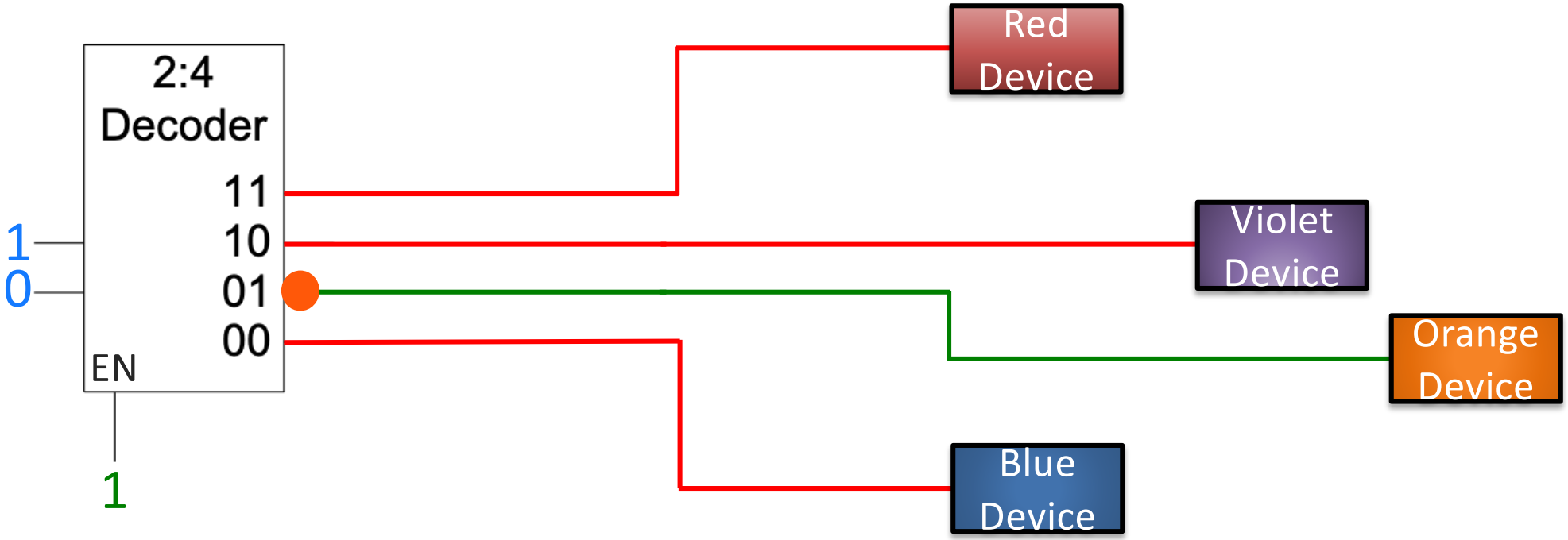
Uses of Decoders

- For each input combination, only one of the outputs is 1



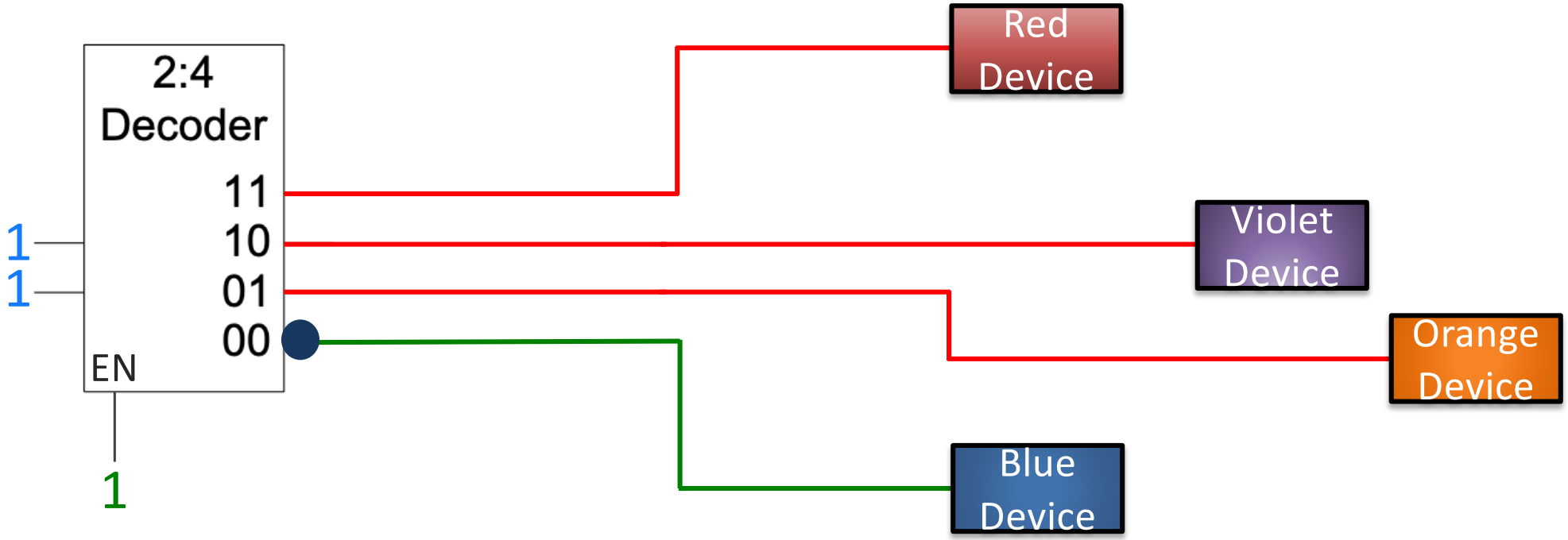
Uses of Decoders

- For each input combination, only one of the outputs is 1



Uses of Decoders

- For each input combination, only one of the outputs is 1



Uses of Decoders

- Think of 00, 01, 10, and 11 codes as instructions
  - To four devices
  - Each device reacts to a specific instruction *in a specific way*

- *We have created a new 2-bit language*
  - *With an interpreter/decoder*

Logic Using Decoders

- Decoders can be combined with OR gates to build logic functions

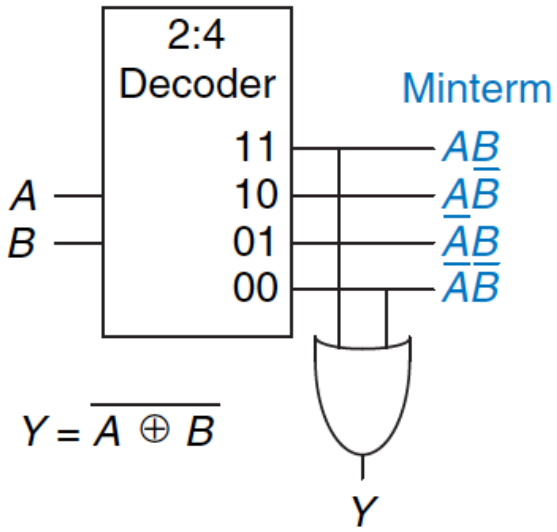
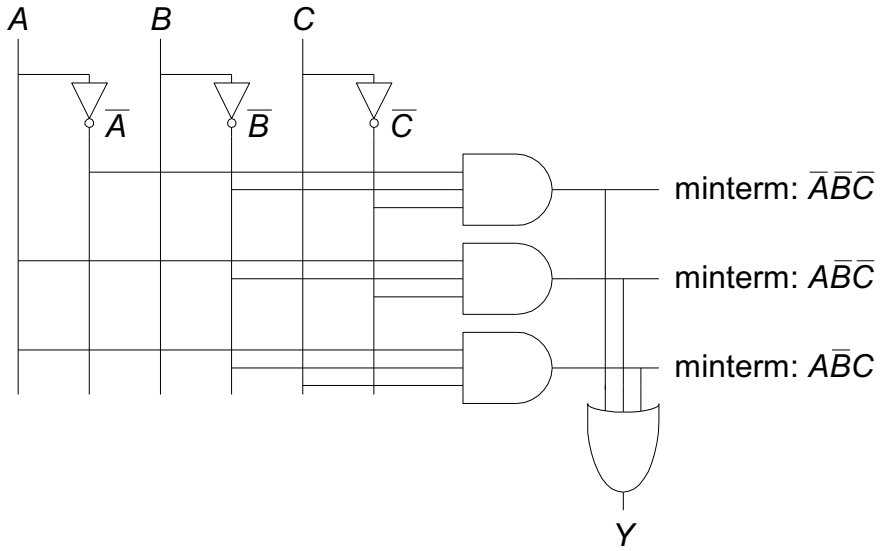


Figure 2.65 Logic function using decoder

Programmable Logic Array (PLA)

- SOP (sum-of-products) leads to two-level logic
- Example: **Y = A'B'C' + AB'C' + AB'C**

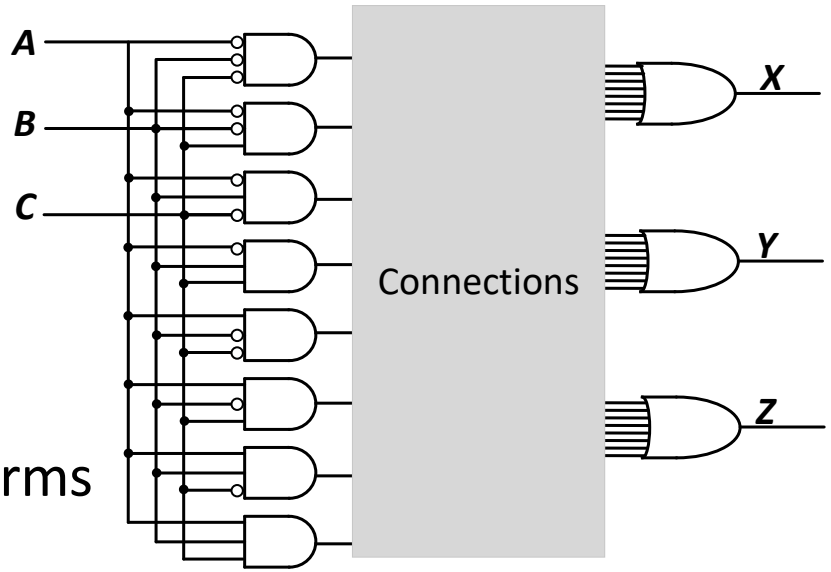


- A PLA enables the two-level SOP implementation of **any** N-input M-output function

Section 5.6.1 of H&H

Programmable Logic Array (PLA)

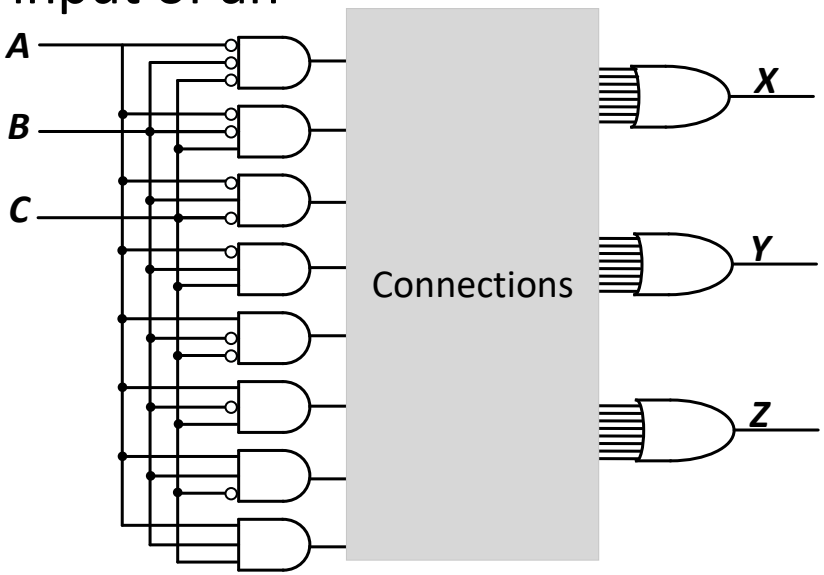
- Common building block for implementing any collection of logic functions
- An **array** of AND gates followed by an **array** of OR gates
- How many AND gates?
  - **Recall SOP:** the number of possible minterms
- How many OR gates?
  - The number of output columns in the truth table



Section 5.6.1 of H&H

Programmable Logic Array (PLA)

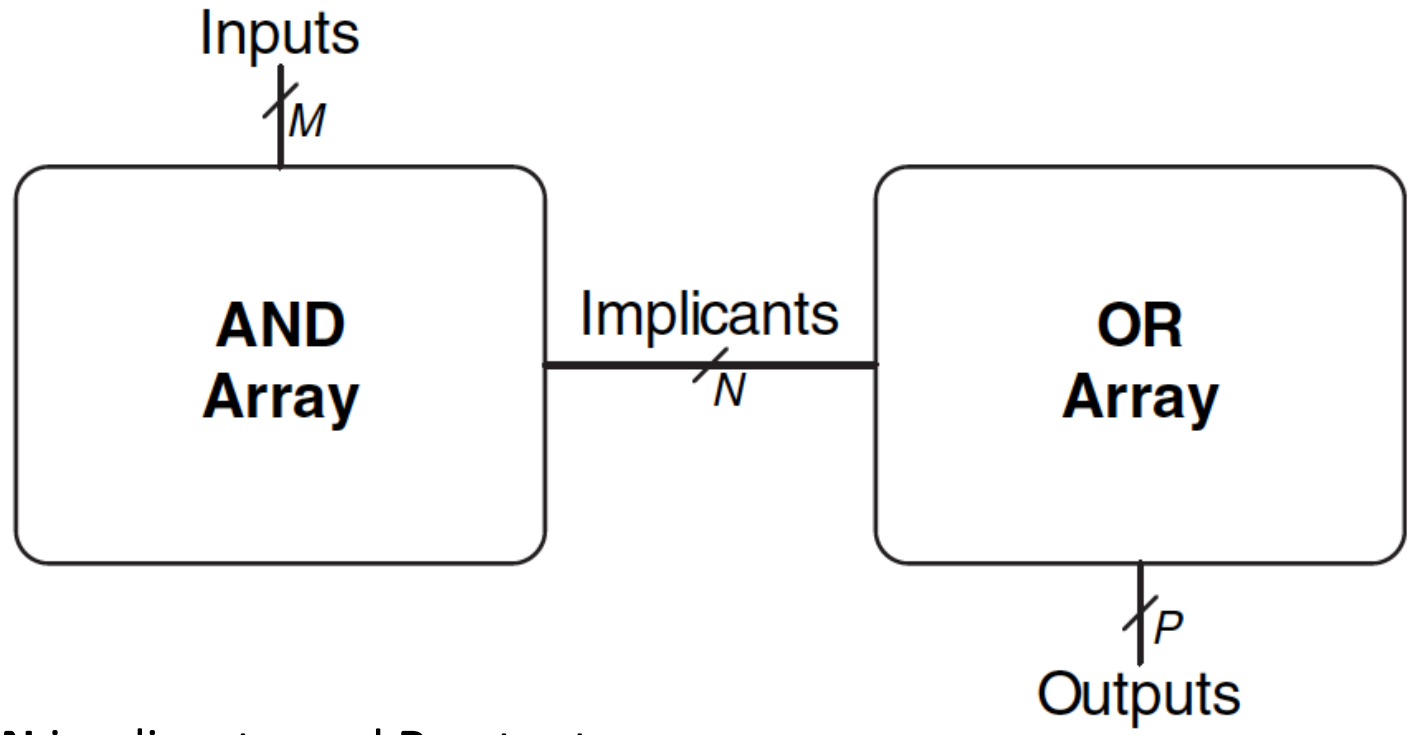
- **How do we implement a logic function?**
  - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP



- **Programming a PLA:** we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function
- **Programmable devices we have talked about:** CPU/processor, FPGA, PLA

Section 5.6.1 of H&H

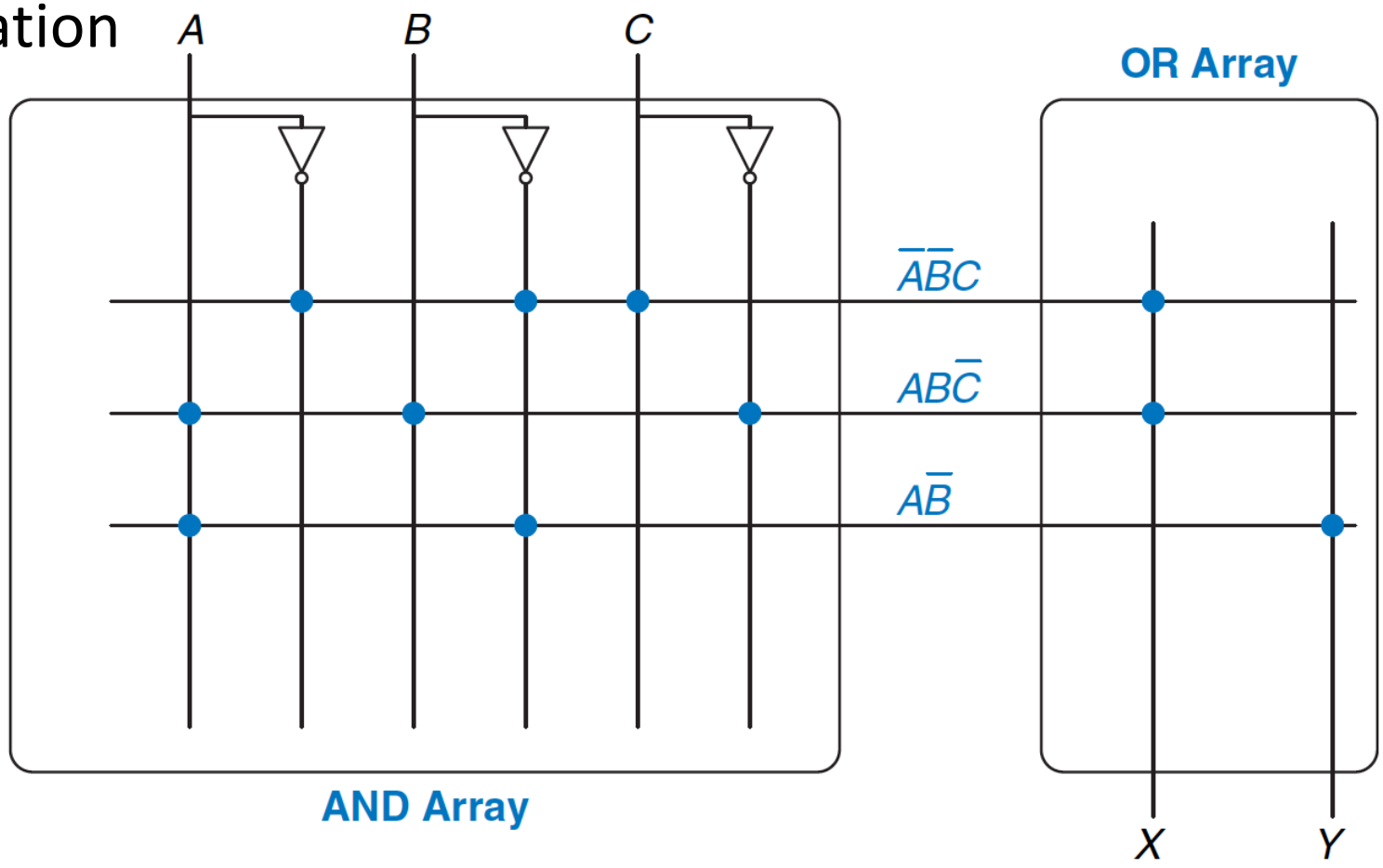
PLA Example (I)



- **M** inputs, **N** implicants, and **P** outputs
- Chips are manufactured in bulk with the same layout (**low cost**)
- Programmed **once** to implement the required function by **programming** **connections**

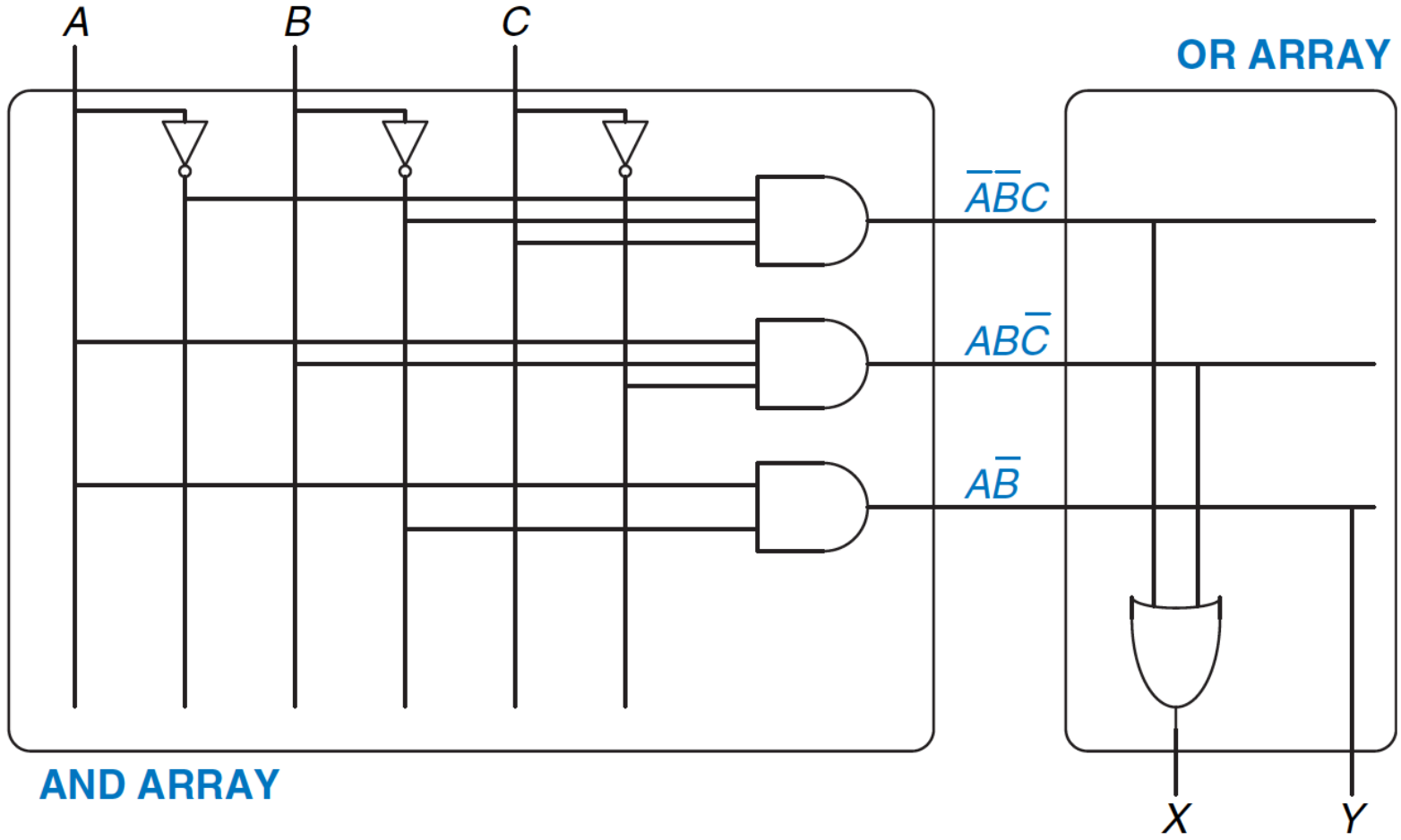
PLA Example (II)

Dot Notation



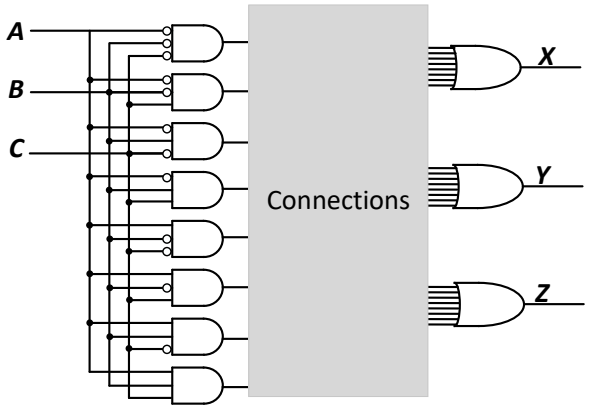
Section 5.6.1 of H&H

PLA Example (III)



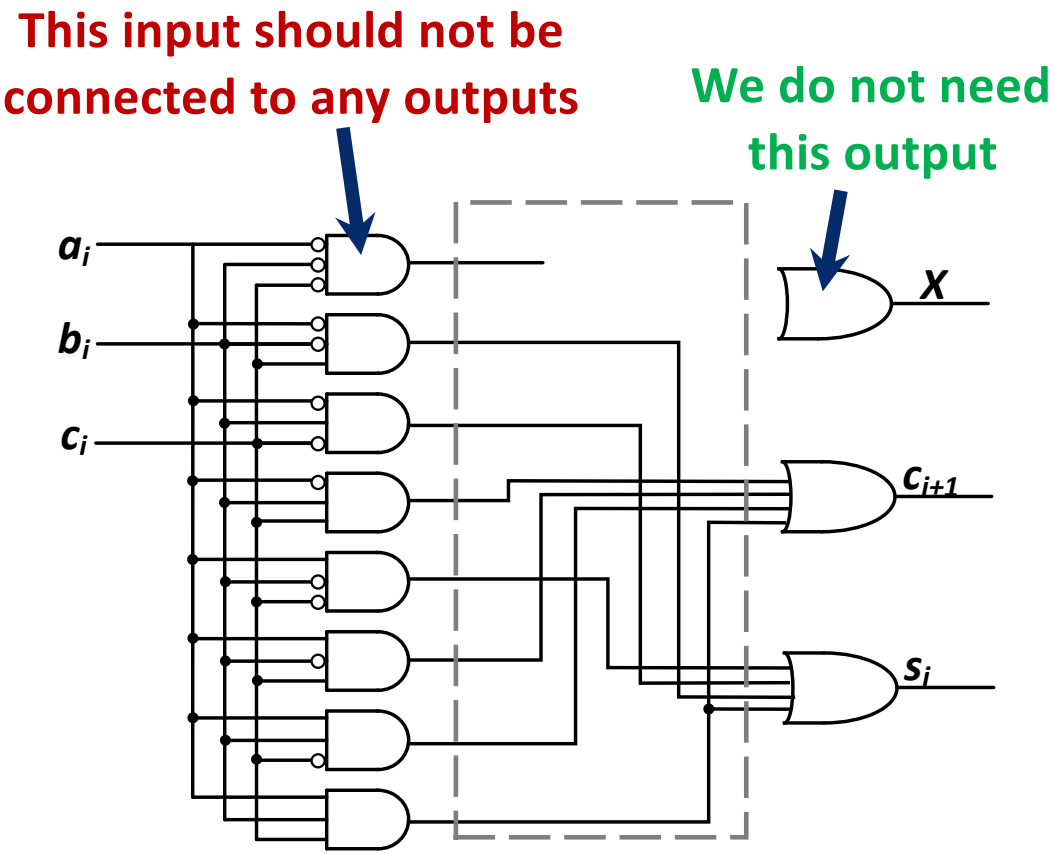
Implementation: Pick the **literals** & **implicants** by programming connections

Full Adder Implementation w/t PLA



Truth table of a full adder

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



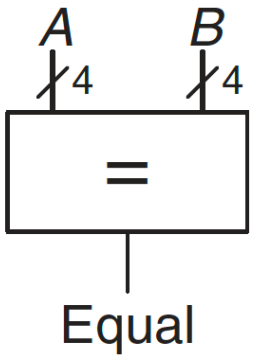
Implementation: Pick the **implicants** by programming connections

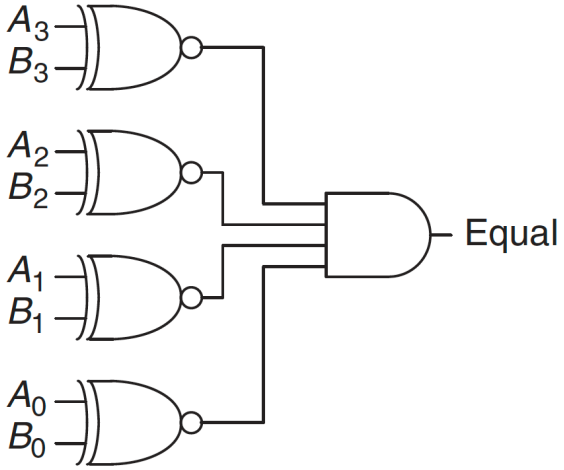
Lessons from PLA

- **Programmability:** Programmable devices incur a **cost**. A lot of logic in a PLA is wasted as only a few **implicants** are needed typically to implement a function. On the other hand, PLAs can be **programmed** after factory manufacturing which is their key **programmability advantage**.

Comparator (Equality Checker)

- Checks if two **N-input** values are exactly the same
  - Example: 4-bit Comparator

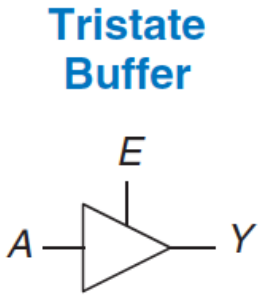




- What about magnitude comparison?

Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

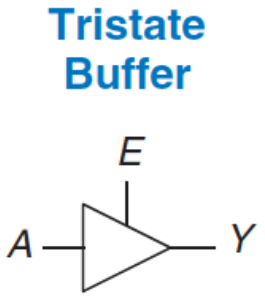
Figure 2.40 Tristate buffer

A tri-state buffer acts like a switch

Section 2.6.2 of H&H

Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer

A tri-state buffer acts like a switch

- When E is HIGH, the output Y is whatever A is
  - Same behavior as a regular buffer

Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

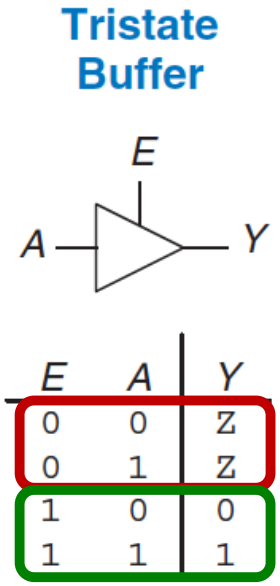


Figure 2.40 Tristate buffer

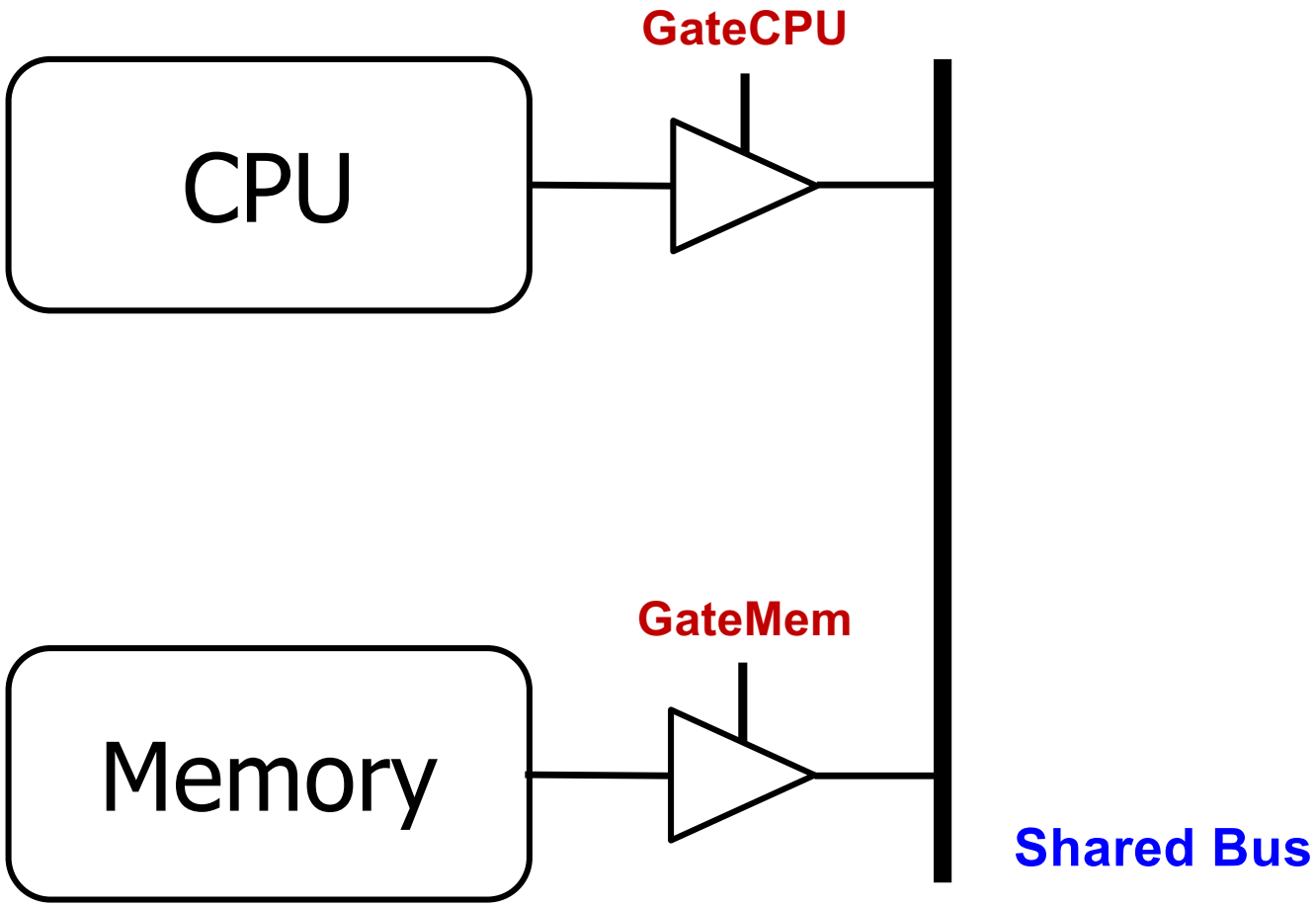
A tri-state buffer acts like a switch

- When E is LOW, output is a floating signal (Z)
- **Floating:** Signal not driven by any circuit (open circuit, floating wire)

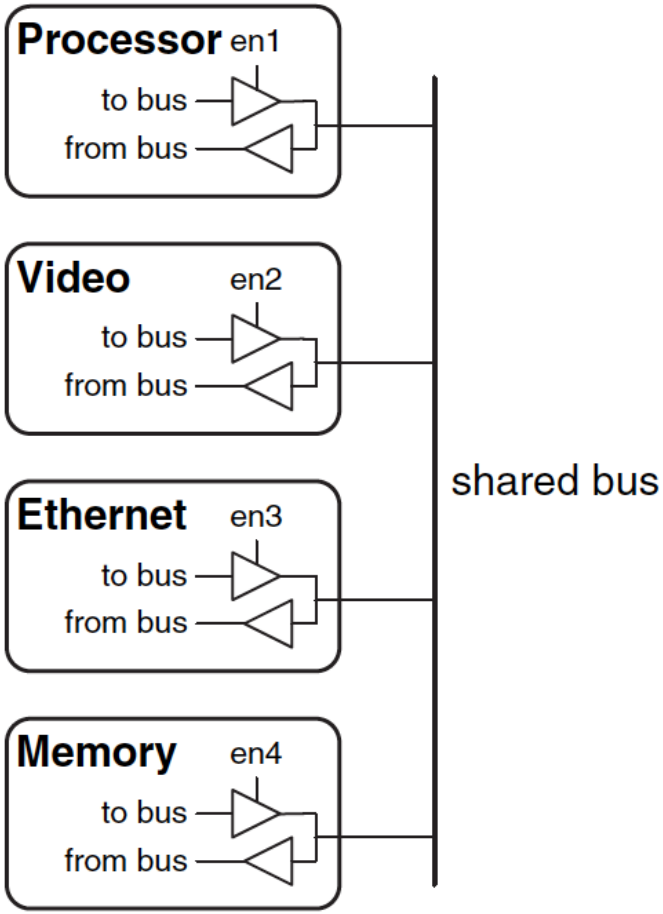
Example: Use of Tri-State Buffers

- Imagine a wire shared by the CPU and memory or two I/O peripherals
- At any time only one of them can place a value on the wire, but not both
- You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

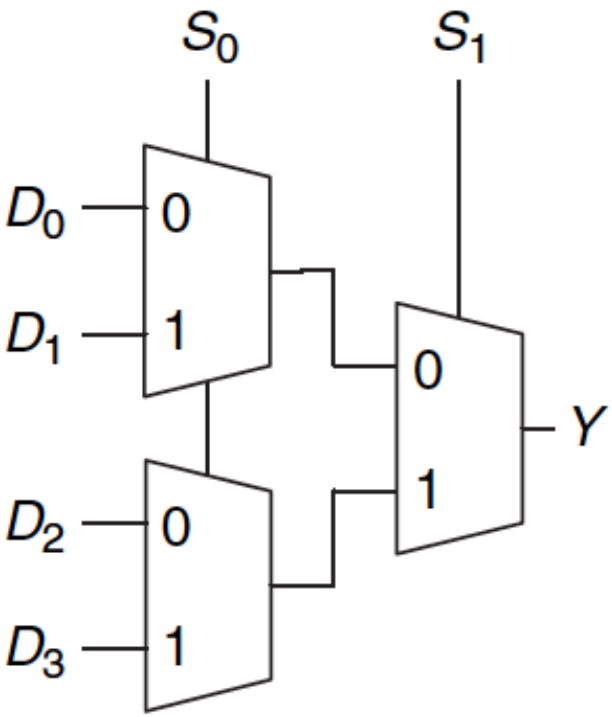
Example: Use of Tri-State Buffers

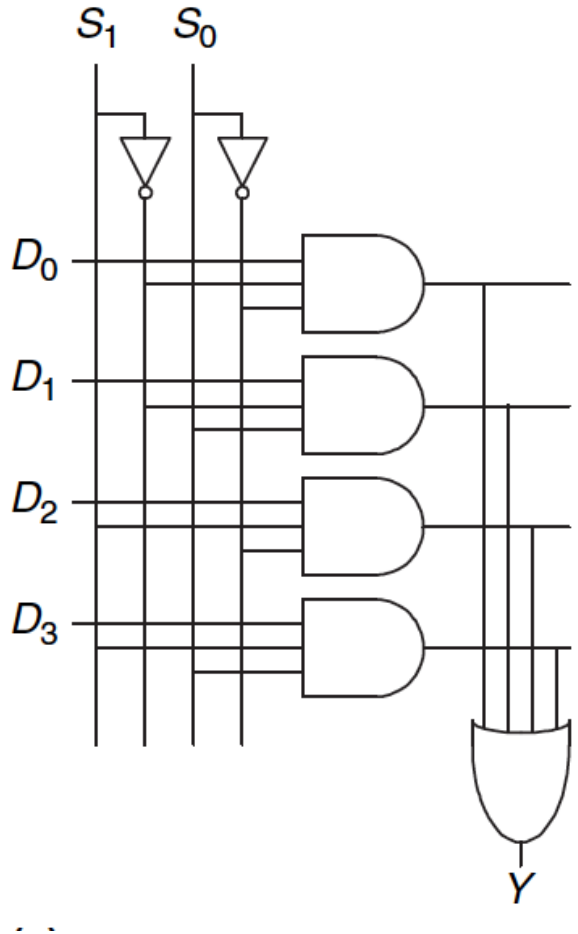


Another Example

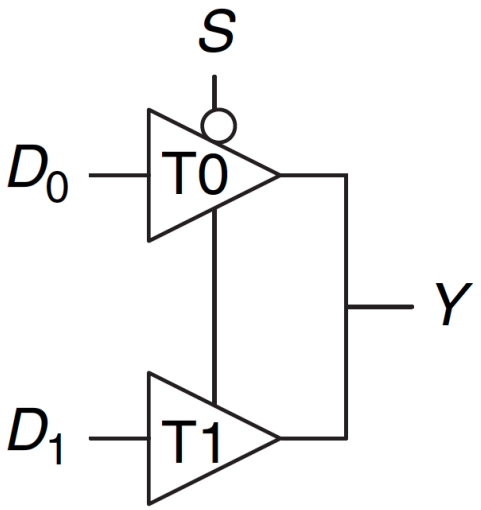


Recall: A 4:1 Multiplexer



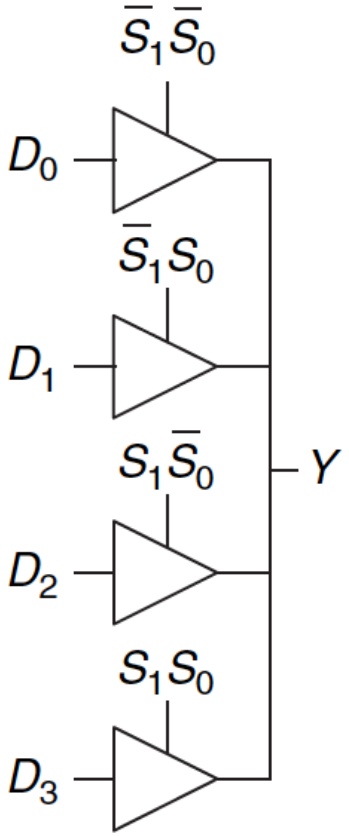


Multiplexer Using Tri-State Buffers



$$Y = D_0\overline{S} + D_1 S$$

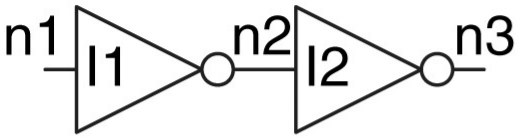
Figure 2.56 **Multiplexer using tristate buffers**

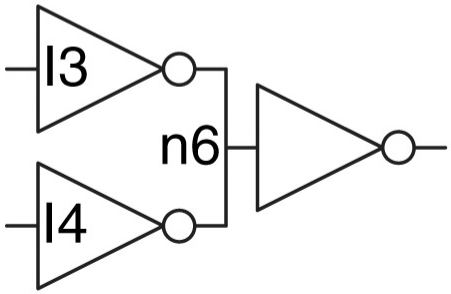


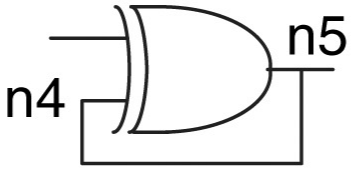
Combinational Composition Rules

- Every circuit element is itself combinational
- Each node is either an input to the circuit or connects to exactly one output terminal of a circuit element
- They circuit contains no cyclic paths; every path through the circuit visits each circuit node at most once

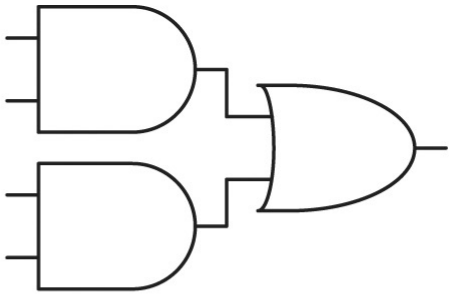
Which circuits are combinational?







Assume n5 is 0
and the other
input of XOR is 1



Implementing Combinational Logic

- Steps in implementing combinational Logic
  - Initial specification (e.g., in English)
  - Construct the truth table
  - Derive the Boolean equation

  Functional specification

  - *Simplify the Boolean equation (use Boolean algebra)*
  - Implement the equation using logic gates

Boolean Algebra (Logic Minimization)

- The sum-of-products canonical form does not lead to the simplest logic gate implementation
- In many cases, we can reduce the total number of gates
- We can reduce the number of literals in the equation
- We use Boolean algebra to simplify Boolean equations
  - Similar in spirit to simplification in ordinary algebra except we are dealing with 0 and 1

Section 2.2 of H&H

Boolean Algebra

- Boolean algebra consists of
  - Axioms (**correct by definition**)
  - Theorems of one variable
  - Theorems of several variables

- Any theorem can be proved via the axioms
  - An axiom is the ground truth and cannot be proven wrong

- The **Principle of Duality**
  - If the symbols 0 and 1 and the operators **AND** and **OR** are interchanged, the statement will still be correct

Boolean Axioms

Number	Axiom	Dual	Name
A1	B = 0 if B ≠ 1	B = 1 if B ≠ 0	Binary Field
A2	$\overline{0} = 1$	$\overline{1} = 0$	NOT
A3	0 • 0 = 0	1 + 1 = 1	AND/OR
A4	1 • 1 = 1	0 + 0 = 0	AND/OR
A5	0 • 1 = 1 • 0 = 0	1 + 0 = 0 + 1 = 1	AND/OR

Dual: Replace: • with +
0 with 1

Boolean Theorems of One Variable

Number	Theorem	Dual	Name
T1	$B \bullet 1 = B$	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	$B + \overline{B} = 1$	Complements

Dual: Replace: • with +
0 with 1

Theorems: Several Variable

#	Theorem	Dual	Name
T6	$B \bullet C = C \bullet B$	$B+C = C+B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	$(B + C) + D = B + (C + D)$	Associativity
T8	$B \bullet (C + D) = (B \bullet C) + (B \bullet D)$	$B + (C \bullet D) = (B+C)\,(B+D)$	Distributivity
T9	$B \bullet (B+C) = B$	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	$(B+C) \bullet (B+\overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) =$ $(B \bullet C) + (\overline{B} \bullet D)$	$(B+C) \bullet (\overline{B}+D) \bullet (C+D) =$ $(B+C) \bullet (\overline{B}+D)$	Consensus

Warning: T8’ (dual of T8) differs from traditional algebra: OR (+) distributes over AND (•)

Proving Theorems

- **Method 1: Perfect induction**
  - Proof by exhaustion: Check all possible input combinations
  - Two expressions are equal if they produce the same value for every possible input combination

- **Method 2: Use other theorems/axioms to simplify equations**
  - As in ordinary algebra, make one side of the equation look like the other side of the equation

Example: Perfect Induction

Number	Theorem	Name
T6	B•C = C•B	Commutativity

B	*C*	*BC*	*CB*
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Example: Perfect Induction

Number	Theorem	Name
T9	B• (B+C) = B	Covering

B	*C*	*(B+C)*	*B(B+C)*
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Method 2: T9 (Covering)

Number	Theorem	Name
T9	B• (B+C) = B	Covering

Method 2: Prove true using other axioms and theorems.

B•(B+C)	= B•B + B•C	T8: Distributivity
	= **B** + B•C	T3: Idempotency
	= B•(1 + C)	T8: Distributivity
	= B•(**1**)	T2: Null element
	= B	T1: Identity

Method 2: T10 (Combining)

Number	Theorem	Name
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	Combining

Prove true using other axioms and theorems:

$B \bullet C + B \bullet \overline{C} = B \bullet (C + \overline{C})$ T8: Distributivity

$= B \bullet (\mathbf{1})$ T5': Complements

$= B$ T1: Identity

Simplifying Boolean Equations

- A basic principle for simplifying sum-of-product equations
  - **P**A + **P**A' = **P**
  - **P** is any implicant
  - Y = A'B + AB = B(A'+A) = B(1) = B

- **An equation is minimized if**
  - it uses the fewest number of implicants
  - if there are multiple equations with the same number of implicants, then the one with the fewest literals

Section 2.2 of H&H

Simplification Example – 1

$Y = AB + AB'$

Y = A T10: Combining

or

= $A(B + B')$ T8: Distributivity

= $A(1)$ T5': Complements

= A T1: Identity

Simplification Example – 2

$Y = A(AB + ABC)$

$= A(AB(1 + C))$	T8: Distributivity
$= A(AB(1))$	T2’: Null Element
$= A(AB)$	T1: Identity
$= (AA)B$	T7: Associativity
$= AB$	T3: Idempotency

Simplification Example – 3A

Y = AB'C + ABC + A'BC

= AC(B + B') + A'BC T8: Distributivity

= AC(1) + A'BC T5: Complements

= AC + A'BC T1: Identity

- The two implicants AC and BC share the minterm ABC
- Are we stuck with simplifying only one of the minterm pairs?

Simplification Example – 3B

Y = AB'C + ABC + A'BC

= AB'C + **ABC + ABC** + A'BC	T3': Idempotency
= (AB'C+ABC) + (ABC+A'BC)	T7': Associativity
= AC + BC	T10: Combining

- The two implicants AC and BC are called prime implicants
- They cannot be combined with any other implicants in the equation to get a new implicant with fewer literals

Simplification Example – 4

$Y = A'B'C' + AB'C' + AB'C$

De Morgan's Theorem

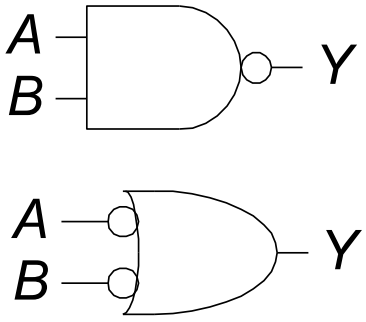
#	Theorem	Dual	Name
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = \overline{B_0} + \overline{B_1} + \overline{B_2} \ldots$	$\overline{B_0 + B_1 + B_2 \ldots} = \overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots$	DeMorgan's Theorem

- The complement of the product is the sum of the complements
- **Dual:** The complement of the sum is the product of the complements

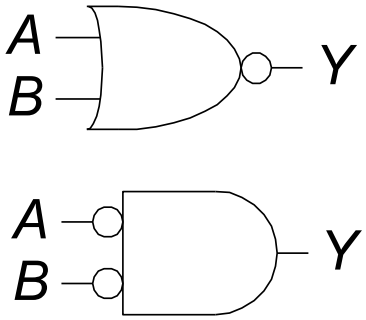
Section 2.2 of H&H

De Morgan's Theorem

- $Y = \overline{AB} = \overline{A} + \overline{B}$



- $Y = \overline{A + B} = \overline{A} \bullet \overline{B}$

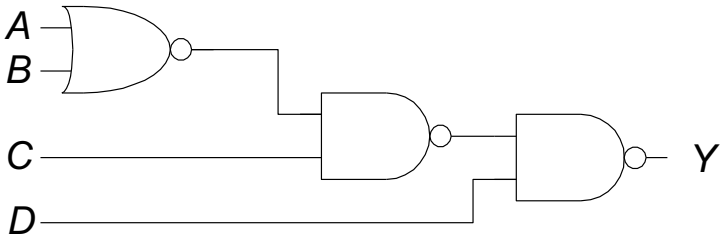


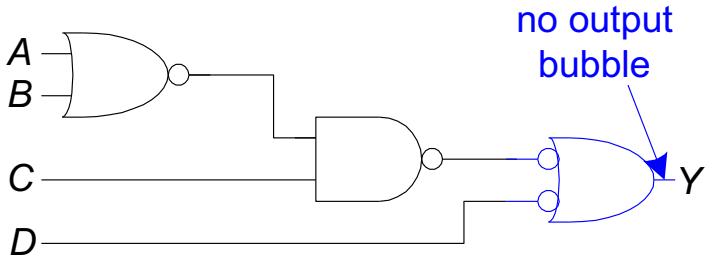
Bubble Pushing Rules

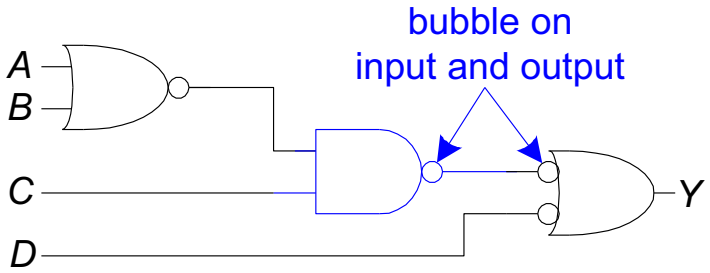
- Pushing bubbles backward/forward *changes the body of the gate* from **AND/OR** to **OR/AND**
- Pushing a bubble *from output back to inputs* put bubbles on all gate inputs
- Pushing *bubbles on all gate inputs forward* towards the output puts a bubble on the output

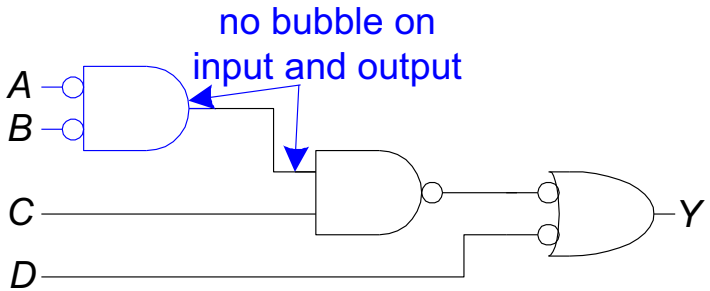
Section 2.5.1 and 2.5.2 of H&H

Bubble Pushing Example







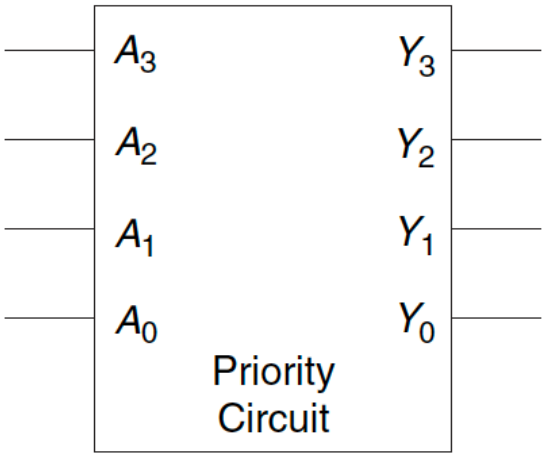


$Y = \overline{A}\overline{B}C + \overline{D}$

Priority Circuit

Example 2.7 of H&H

- **Priority circuit**
  - Inputs: “Requestors” with priority levels
  - Outputs: “Grant” signal for each requestor
  - **Example:** 4-bit priority circuit
- Room reservation system with priority levels
- Computer bus demanded by four CPUs



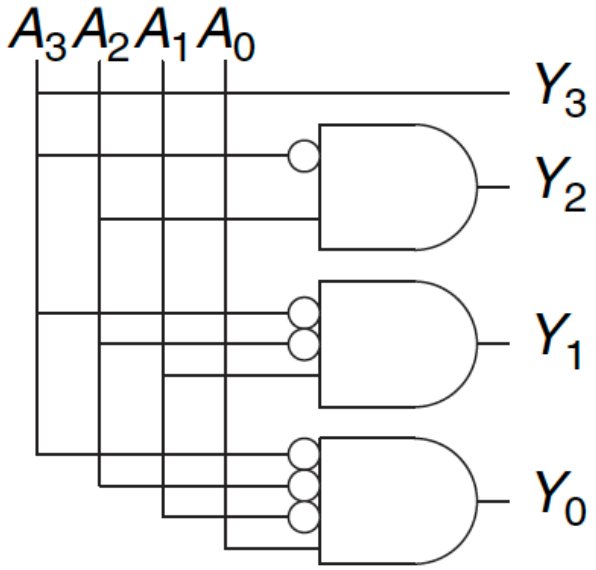
A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Priority Circuit

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 **Priority circuit truth table with don't cares (X's)**



X (Don't Care) means *We don't care what the value of this input is*

$Y_3 = A_3$

$Y_2 = A_3'A_2$

$Y_1 = A_3'A_2'A_1$

$Y_0 = A_3'A_2'A_1'A_0$

Optional Self-Study

- Product of Sums (POS)
  - Interesting but not entirely needed if you understand SOP well
  - Follows from Demorgan

Section 2.2.3 of H&H

Alternative Canonical Form: POS

- Product of Sums (**POS**)
- Find all the input combinations (maxterms) for which the output of the function is FALSE
- The function evaluates to FALSE (i.e., the output is 0) if any of the Sums (**maxterms**) causes the output to be 0
- Think: DeMorgan of SOP of $\overline{F}$

Alternative Canonical Form: POS

Product of Sums (POS)

product

$$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)$$

sums

Each sum term represents one of the "zeros" of the function

This input

$$F = \underbrace{(A + B + C)}_{0\ 0\ 0} \underbrace{(A + B + \overline{C})}_{0\ 0\ 1} \underbrace{(A + \overline{B} + C)}_{0\ 1\ 0}$$

Activates this term

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

For the given input, only the shaded sum term will equal 0

$$A + \overline{B} + C = 0 + \overline{1} + 0$$

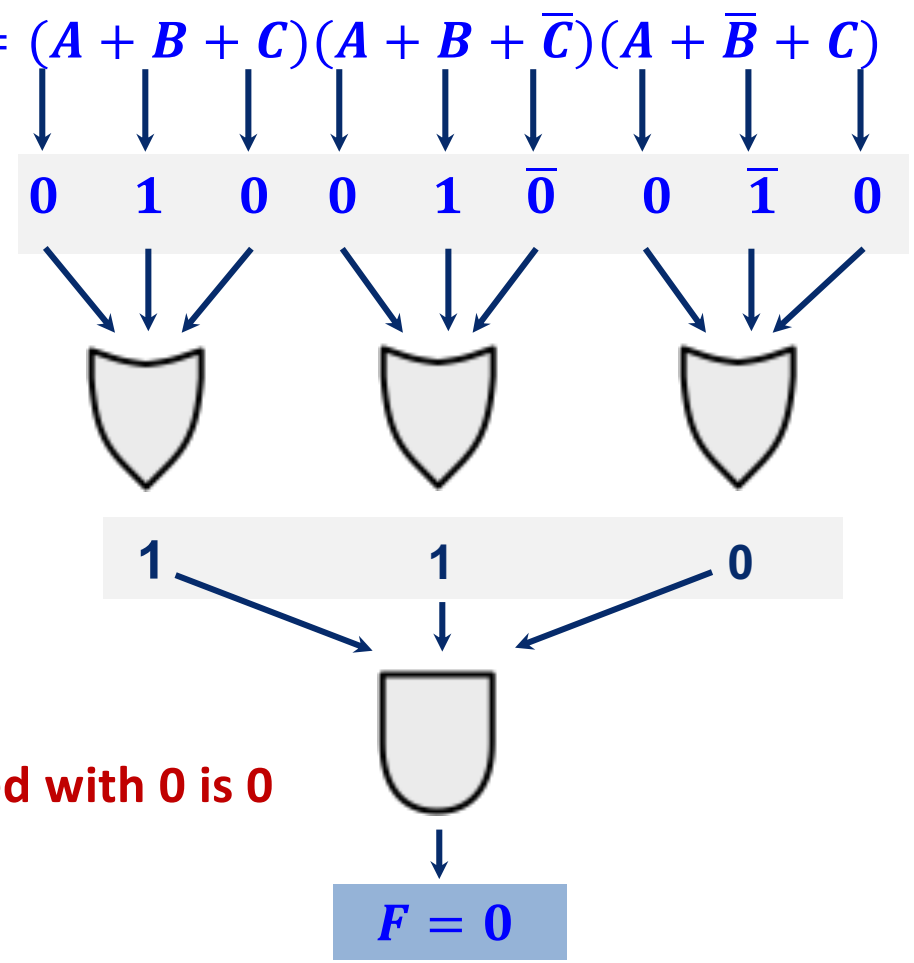
Anything ANDed with 0 is 0; Output F will be 0

Consider A=0, B=1, C=0

0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Input

0 1 0



Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is F = 0

Optional Self-Study

- More combinational circuits
  - Shifters
  - Rotators
  - Multiplication
  - Division
  - FPGAs

Section 5.2.5, 5.2.6, 5.2.7, 5.6.2 of H&H