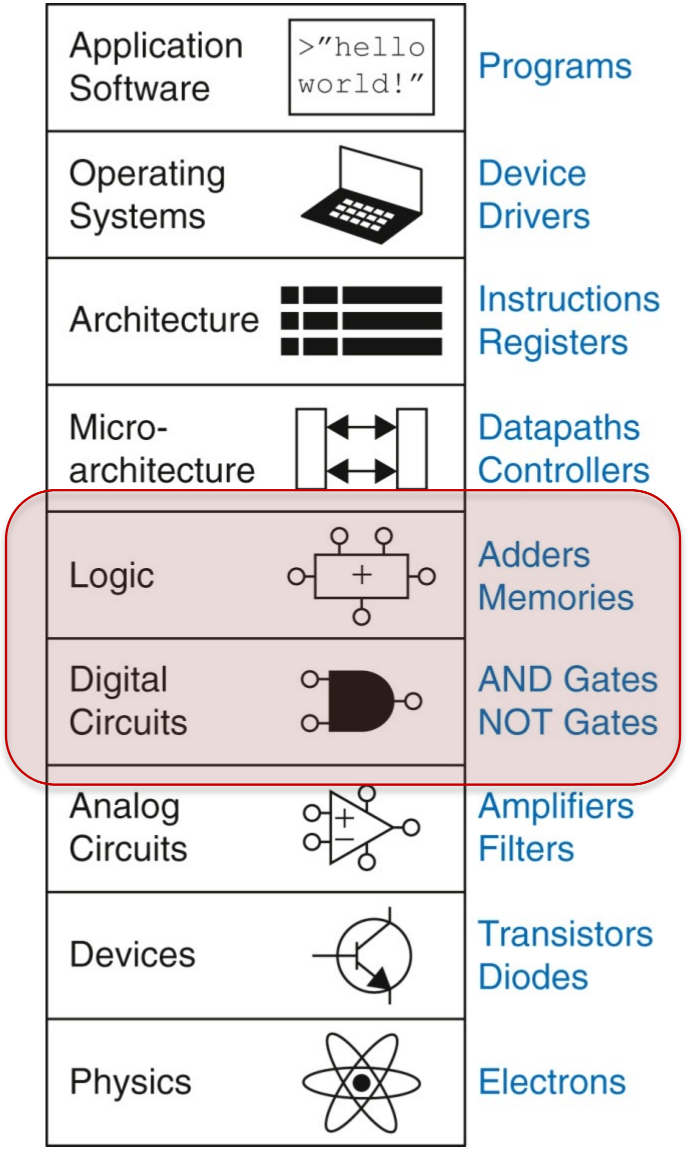
COMP2300-COMP6300-ENGN2219
Computer Organization & Program Execution

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

Australian National University

Plan of Week 2

- **Week 1:** Digital abstraction and binary digits
- **Week 1:** Number systems for binary variables
- **This Week:** Boolean logic & Logic gates (contd)
- **This Week:** Combinational logic (more than just gates)



Broadening our horizon

"one layer at a time"

Classification of Digital Circuits

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

- **Sequential Circuit:** Output depends on the current and history of inputs
  - The sequence of inputs **over time** determine the output
  - Sequential circuits have a state or memory
  - **Example:** Elevator controller (**State:** on the ground, in transit, at the 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

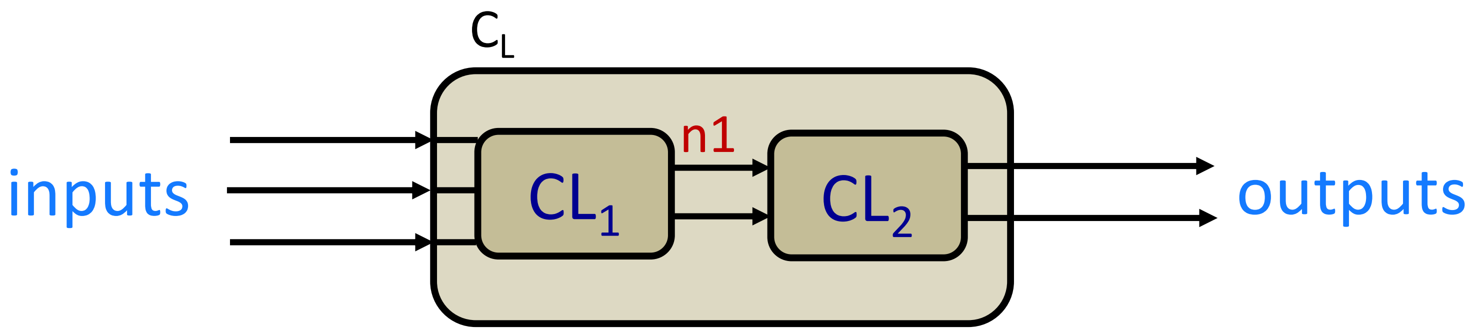
time →	*t0*	*t1*	*t2*	*t3*	*t4*	*t5*	*t6*
A	0	1	1	0	1	0	1
B	0	1	0	0	1	0	1
Output	0	1	0	0	1	0	1

- 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

Implementing Combinational Logic

- Steps in implementing combinational Logic
  - Initial specification (e.g., in English)
  - Construct the truth table
  - Derive the Boolean equation
  - Simplify the Boolean equation (use Boolean algebra)
  - Implement the equation using logic gates

Functional specification

Specification

[Happiness detector] Alex is not happy if there is a work-related deadline or the beach is closed due to bad weather. Design a circuit that outputs 1 only if Alex 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 may be implicit 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: Alex is not happy if there is a work-related deadline or the beach is closed due to bad weather. Design a circuit that outputs 1 only if Alex is happy.

Interface

- **Deadline?** (D)
  - 0: there is not a deadline
  - 1: there is a deadline
- **Beach** 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

Truth Table: Beach

Specification: **IF** it is warm and sunny, **OR** it is my birthday, **THEN** I am going to the beach. Write the truth table where the output is 1 when I am going to the beach

Deriving a Boolean Equation

- The truth table is the unique signature of a Boolean function
  - But it is an expensive representation
- **Why is that?**

Deriving a Boolean Equation

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

- Enables the systematic transformation of the function into simpler functions (using Boolean algebra, we will see later)
  - Different hardware implementations
  - The simplification process can be automated via **Computer-Aided Design (CAD)** and **Electronic Design Automation (EDA)**

- Different Boolean expressions of the same Boolean function lead to different logic gate-level implementations
  - **Different hardware area, cost, latency, energy properties**

Definitions

- **Complement:** variable with a bar or prime (') over it
  $\overline{A}$, $\overline{B}$, $\overline{C}$, A', B', C'

- **Literal:** variable or its complement
  A , $\overline{A}$, B , $\overline{B}$, C , $\overline{C}$

- **Implicant:** product (AND) of literals
  $(A \cdot B \cdot \overline{C})$, $(\overline{A} \cdot C)$, $(B \cdot \overline{C})$

- **Minterm:** product (AND) that includes **all** input variables
  $(A \cdot B \cdot \overline{C})$, $(\overline{A} \cdot \overline{B} \cdot C)$, $(\overline{A} \cdot B \cdot \overline{C})$

- **Maxterm:** sum (OR) that includes **all** input variables
  $(A + \overline{B} + \overline{C})$, $(\overline{A} + B + \overline{C})$, $(A + B + \overline{C})$

Section 2.2 of H&H

Minterms

			Minterms	
x	y	z	**Term**	**Designation**
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'yz'$	m_2
0	1	1	$x'yz$	m_3
1	0	0	$xy'z'$	m_4
1	0	1	$xy'z$	m_5
1	1	0	xyz'	m_6
1	1	1	xyz	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

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 **maxterm** is obtained from an **OR** term of **n** variables

Operation Precedence

- NOT has the highest precedence
- Next is AND
- Last is 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

- Enable a single, universally agreed on way of representing a Boolean function from its truth table
  - Also called canonical representations

- Sum of Products (SOP) form

- Product of Sums (POS) form

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**

Two-Level Canonical Forms: SOP

Sum of Products Form (SOP)

Also known as **disjunctive normal form** or **minterm expansion**

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

$$F = \overset{0\ 1\ 1}{\underline{\bar{A}BC}} + \overset{1\ 0\ 0}{\underline{A\bar{B}\bar{C}}} + \overset{1\ 0\ 1}{\underline{A\bar{B}C}} + \overset{1\ 1\ 0}{\underline{AB\bar{C}}} + \overset{1\ 1\ 1}{\underline{ABC}}$$

- 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)

All Boolean equations can be written in SOP form

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

SOP Form – Why Does it Work?

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

This input

$$F = \overset{0\ 1\ 1}{\bar{A}BC} + \overset{1\ 0\ 0}{A\bar{B}\bar{C}} + \boxed{\overset{1\ 0\ 1}{A\bar{B}C}} + \overset{1\ 1\ 0}{AB\bar{C}} + \overset{1\ 1\ 1}{ABC}$$

Activates this term

- Only the shaded product term — $A\bar{B}C$ $= \mathbf{1 \cdot \bar{0} \cdot 1}$ — 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

The function evaluates to TRUE (i.e., output is 1)
if **any** of the Products (minterms) causes the output to be 1

Standard Notation for SOP Form

- Standard "shorthand" notation
  - If we agree on the order of the variables in the rows of truth table…
    - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

A	B	C	F	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	
1	0	0	1	**100 = decimal 4 so this is minterm #4, or m4**
1	0	1	1	
1	1	0	1	
1	1	1	1	**111 = decimal 7 so this is minterm #7, or m7**

f =

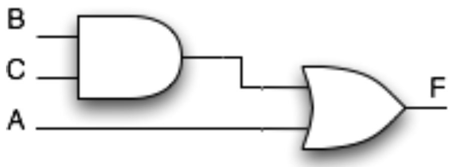
We can write this as a sum of products

Or, we can use a summation notation

Canonical SOP Form

A	B	C	minterms
0	0	0	$\bar{A}\bar{B}\bar{C}$ = m0
0	0	1	$\bar{A}\bar{B}C$ = m1
0	1	0	$\bar{A}B\bar{C}$ = m2
0	1	1	$\bar{A}BC$ = m3
1	0	0	$A\bar{B}\bar{C}$ = m4
1	0	1	$A\bar{B}C$ = m5
1	1	0	$AB\bar{C}$ = m6
1	1	1	ABC = m7

Shorthand Notation for Minterms of 3 Variables



2-Level AND/OR Realization

F in canonical form:

$$F(A,B,C) = \sum m(3,4,5,6,7)$$
$$= m3 + m4 + m5 + m6 + m7$$

F =

canonical form ≠ minimal form

F

More SOP Examples

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

$Y1 = \sum(1,3)$

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, $\sum(\boldsymbol{m1}, \boldsymbol{m2}, \dots)$

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

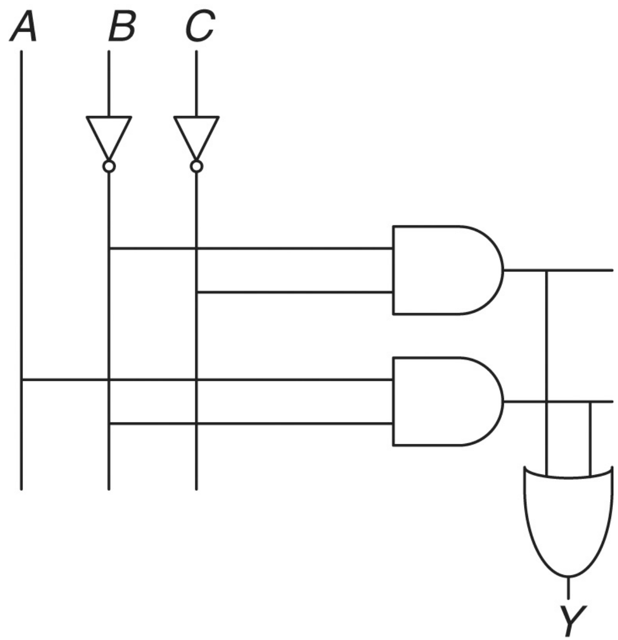
Schematic: A diagram of a digital circuits with elements (gates) and the wires that connect them together

Example Boolean Eq

Y = AB' + B'C'

Schematic

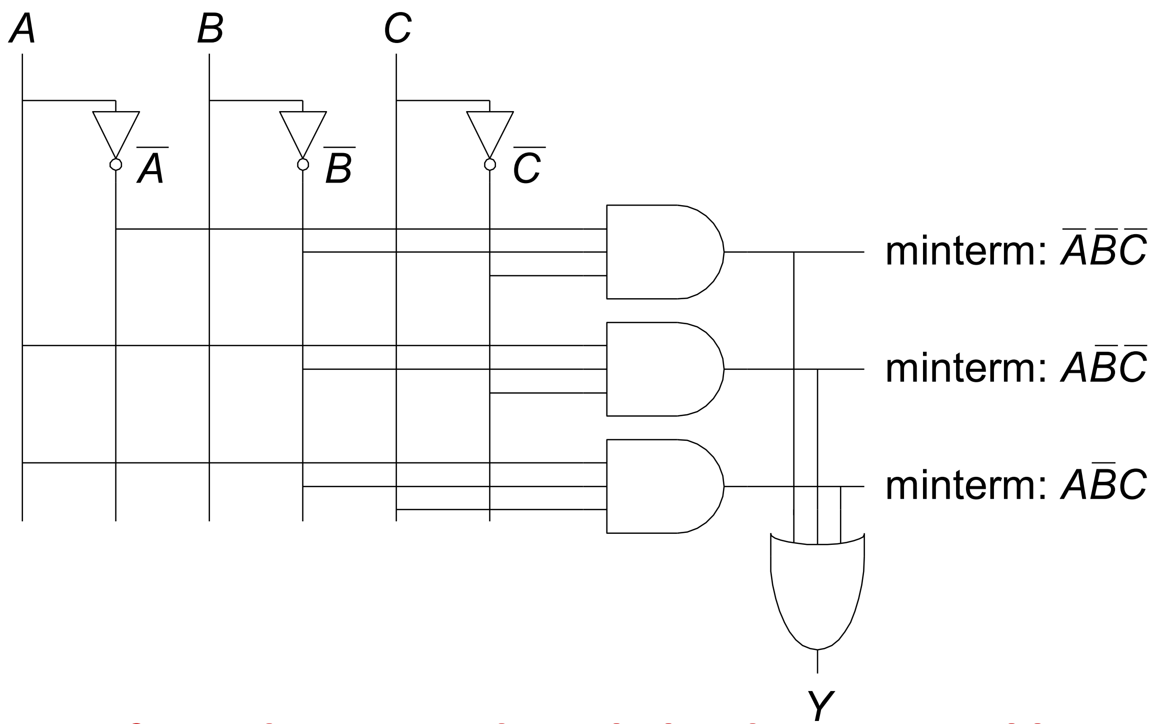
1. Inputs are on the left (or top) side
2. Outputs are on the right
3. Gates flow from left to right
4. Use straight wires
5. Wires connect at a T junction
6. A dot where wires cross indicates a connection



From Equation to Gates

- **Another example**

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



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

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

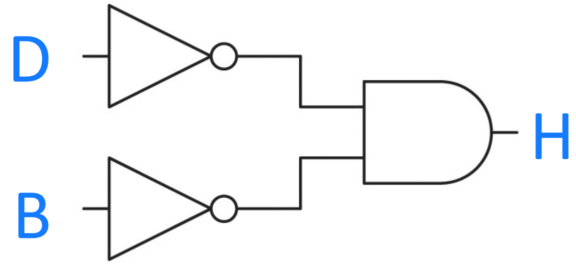
D	B	H
0	0	1
0	1	0
1	0	0
1	1	0

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

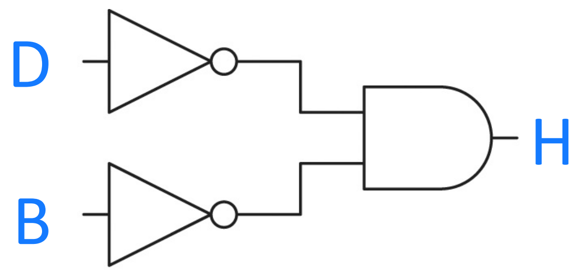
Which (monolithic) gate is this?

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

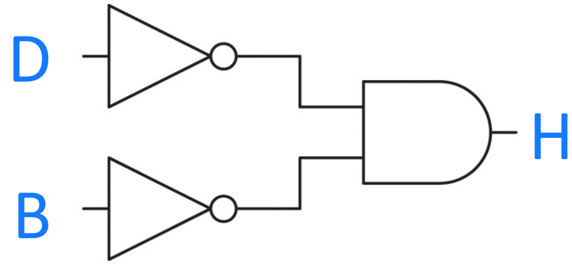
Boolean Eq

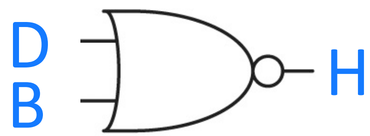
H = D'B'

H = (D)' AND (B)'

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

Logic Gate Implementation





Schematic: Happiness Detector

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

Combinational Building Blocks used in Modern Computers

Multiplexers

Multiplexer: T. Table + Eq

Specification: 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.

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

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

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

$$Y = S'D_0 + SD_1$$

Boolean algebra:
Distribution of product over sums

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

The minimum you can do is write the truth table systematically and express the Boolean function using the SOP canonical form

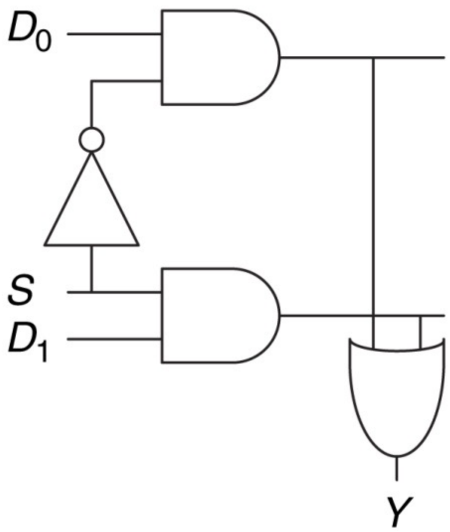
But, remember, ...
canonical form ≠ minimal form

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

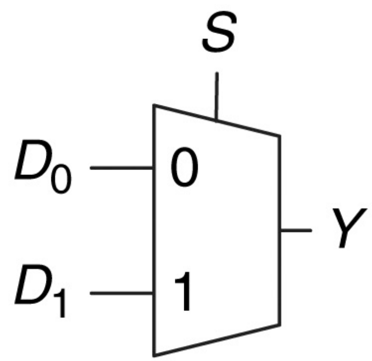


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

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

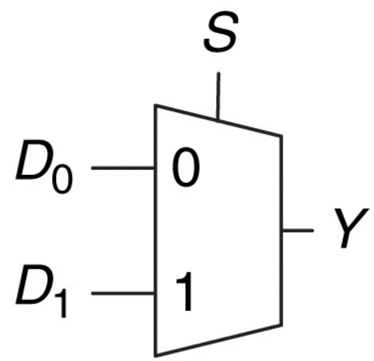
- We will use the high-level schematic for 2:1 mux and ignore the gate-level **implementation** details



High-level Schematic

Multiplexer Applications

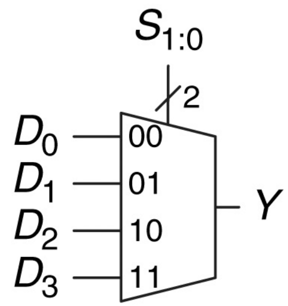
- Heavily used in **control** circuitry
  - Decision making
  - Which of the many ***competing outcomes*** to ***select***?



- Select one of the many signals and send it to another unit

- Think of if/else blocks in high-level programs

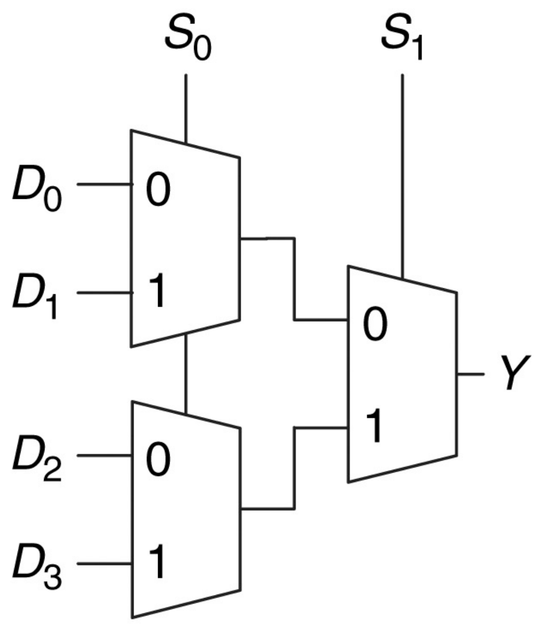
Wider (4:1) Multiplexer



- 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

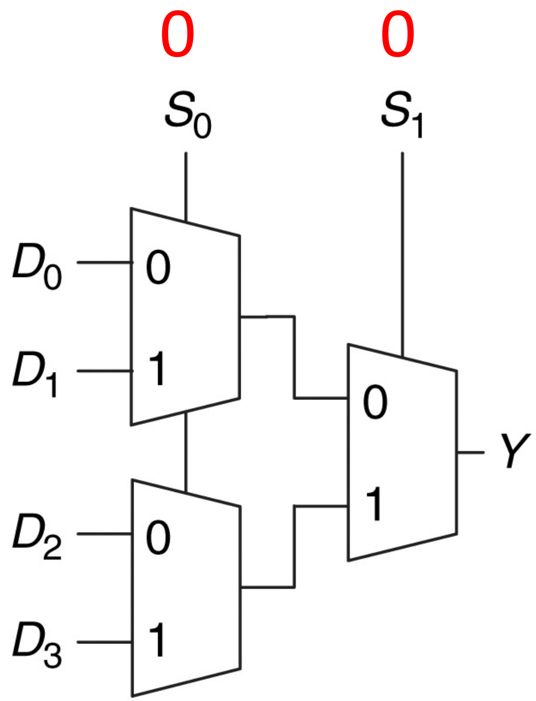
Wider (4:1) Multiplexer

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



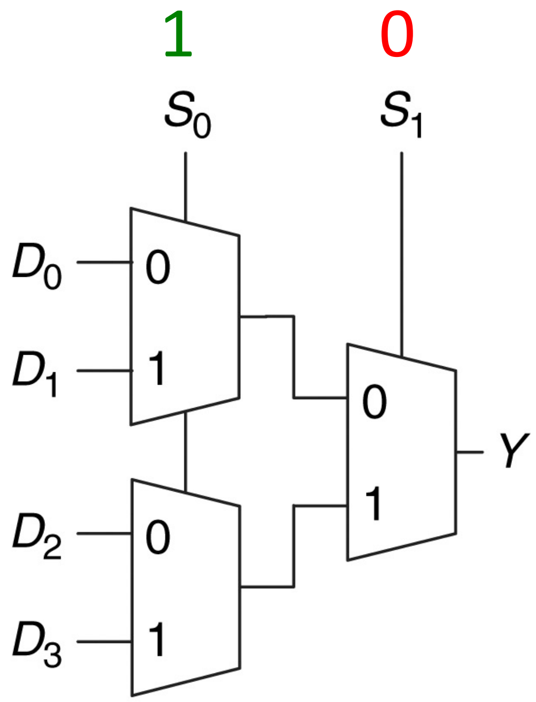
Wider (4:1) Multiplexer

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



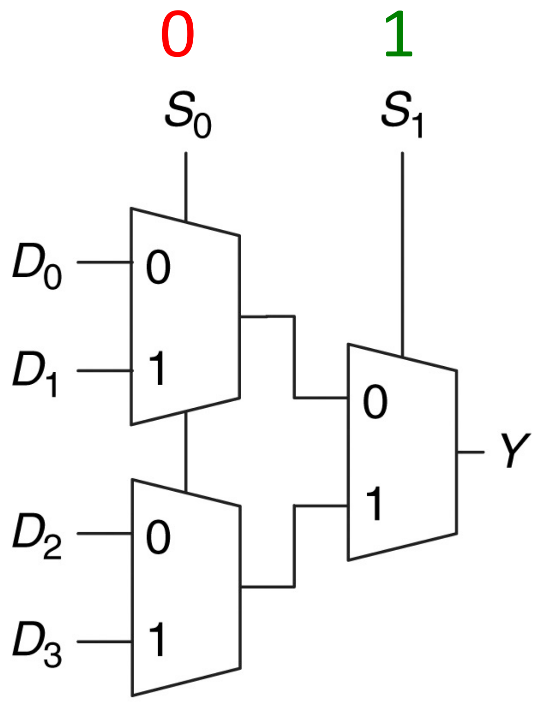
Wider (4:1) Multiplexer

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



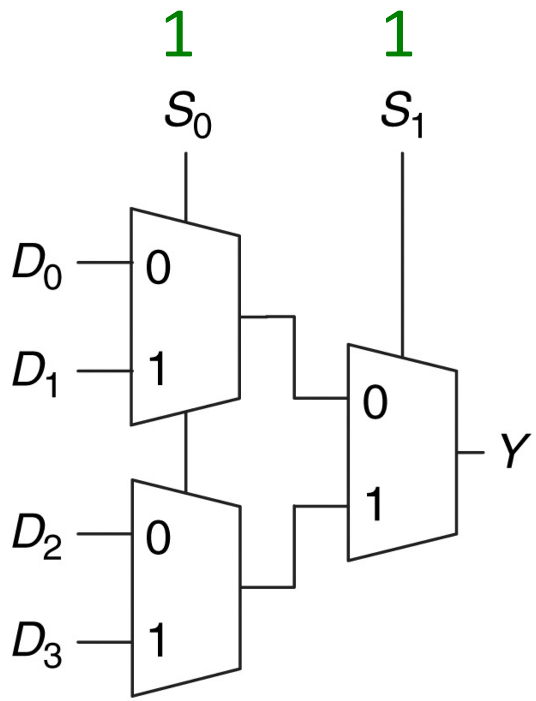
Wider (4:1) Multiplexer

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

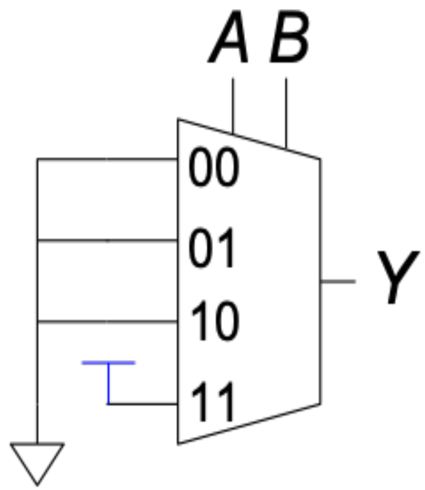
Logic Using Multiplexers

- Any truth table can be seen as a lookup table (LUT)
  - Lookup 00, and we see either 0 or 1
  - It is like looking up a dictionary

- **Muxes are used as LUTs 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

$Y = AB$

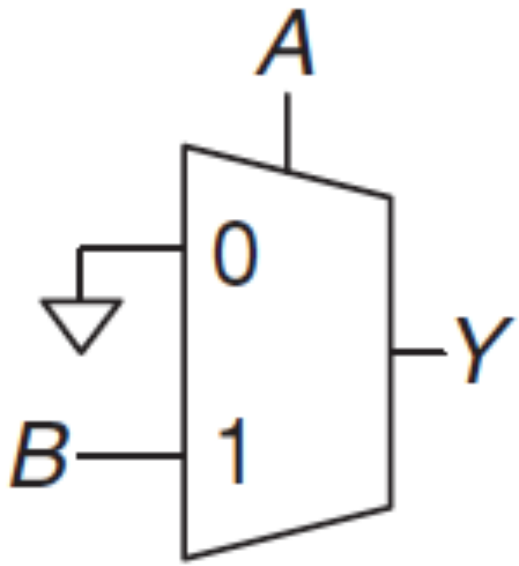


Logic Using Multiplexers

$Y = AB$

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

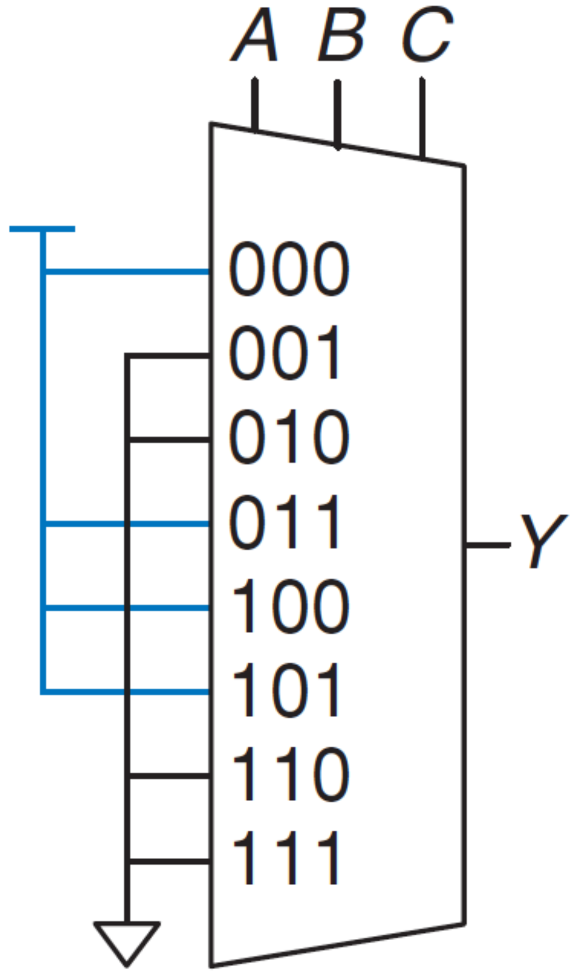
- **Key idea:** Connect multiplexer inputs to 0 (zero/ground) or 1 (high) by inspecting the truth table

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

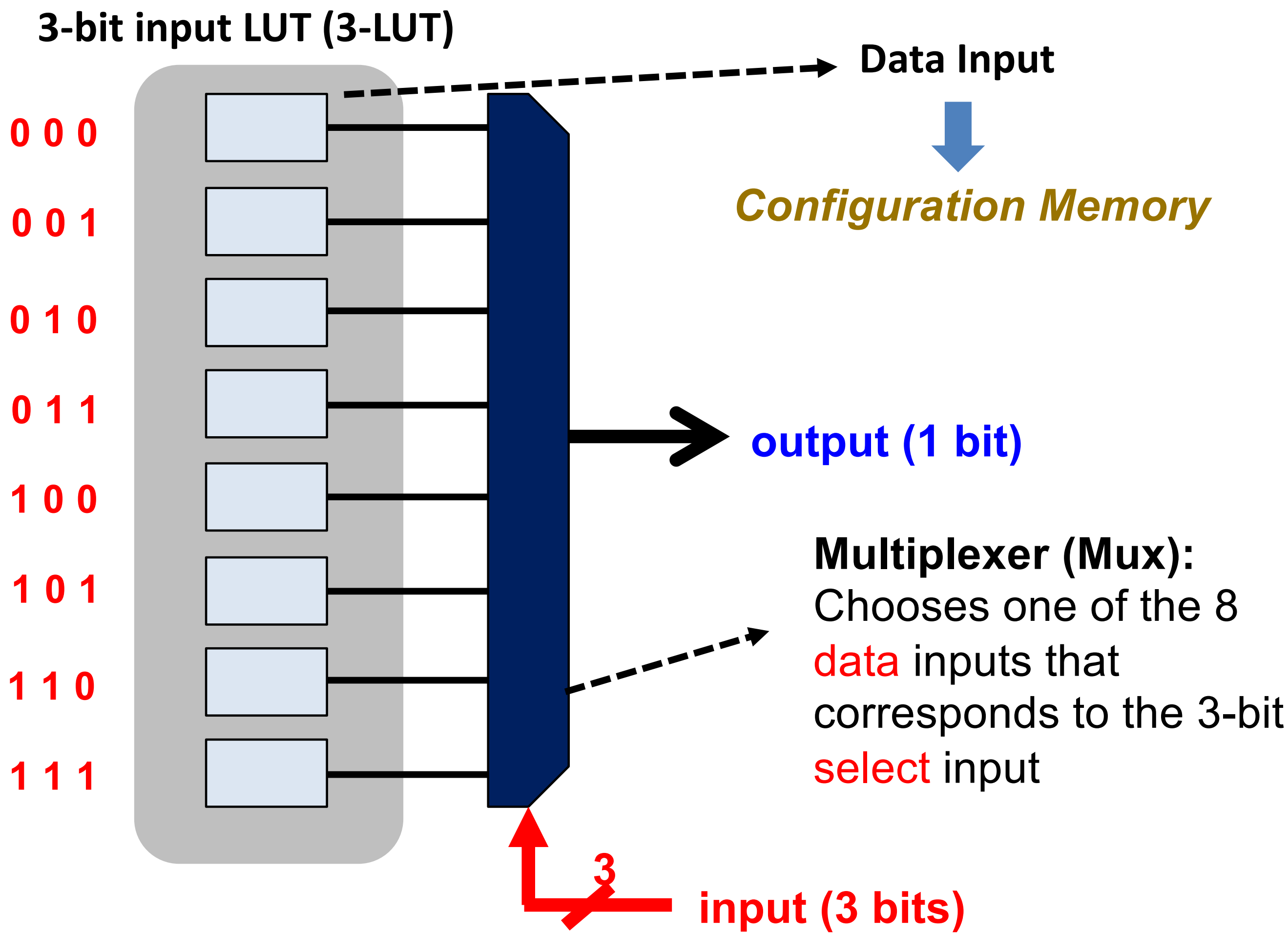
$$Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$$



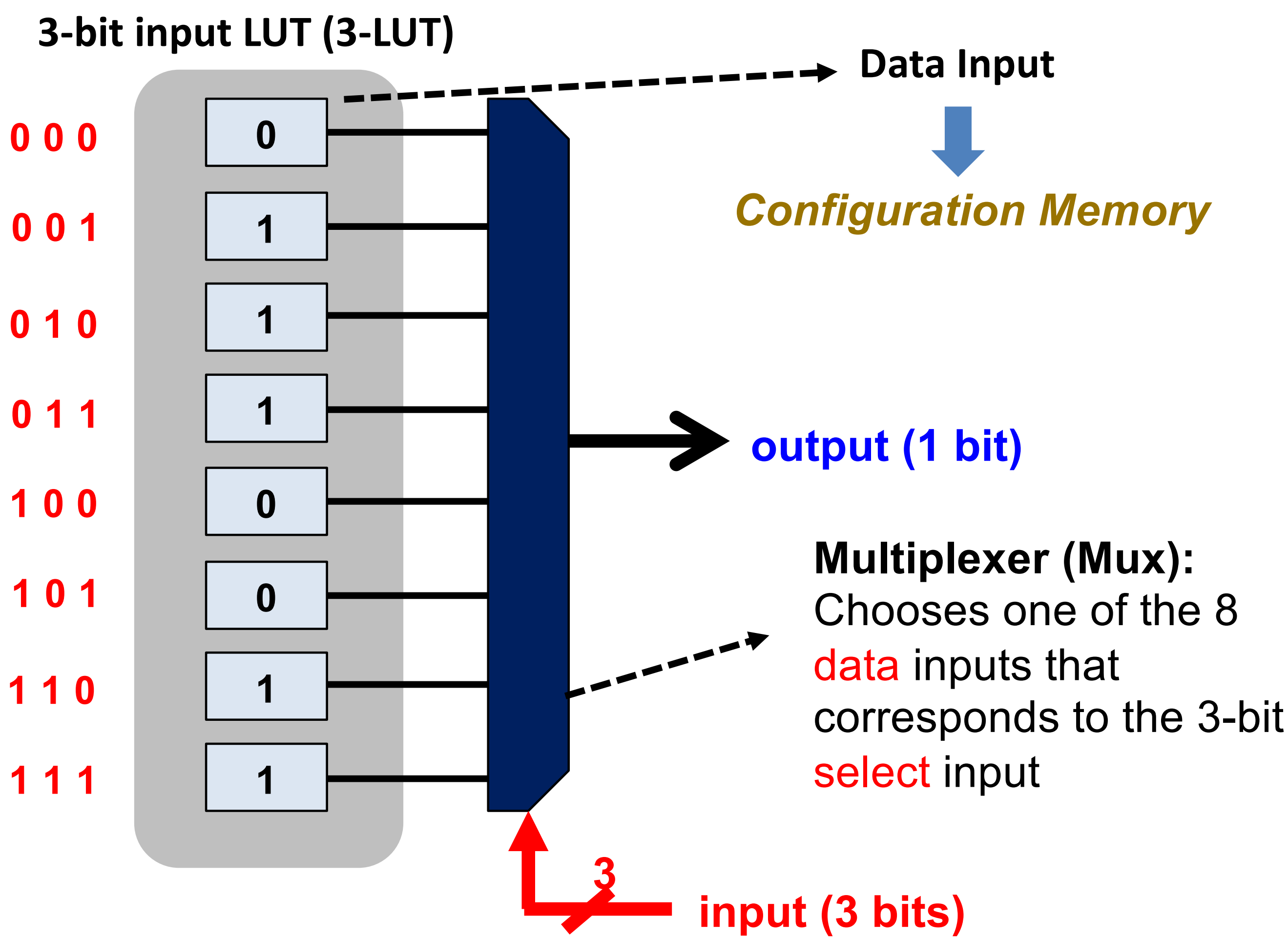
3-Input Lookup Table (LUT)

- LUTs are building blocks of **F**ield **P**rogrammable **G**ate **A**rray (**FPGA**)
- Many LUTs in an FPGA chip to implement logic functions with many variables
- The data inputs are stored as configuration memory

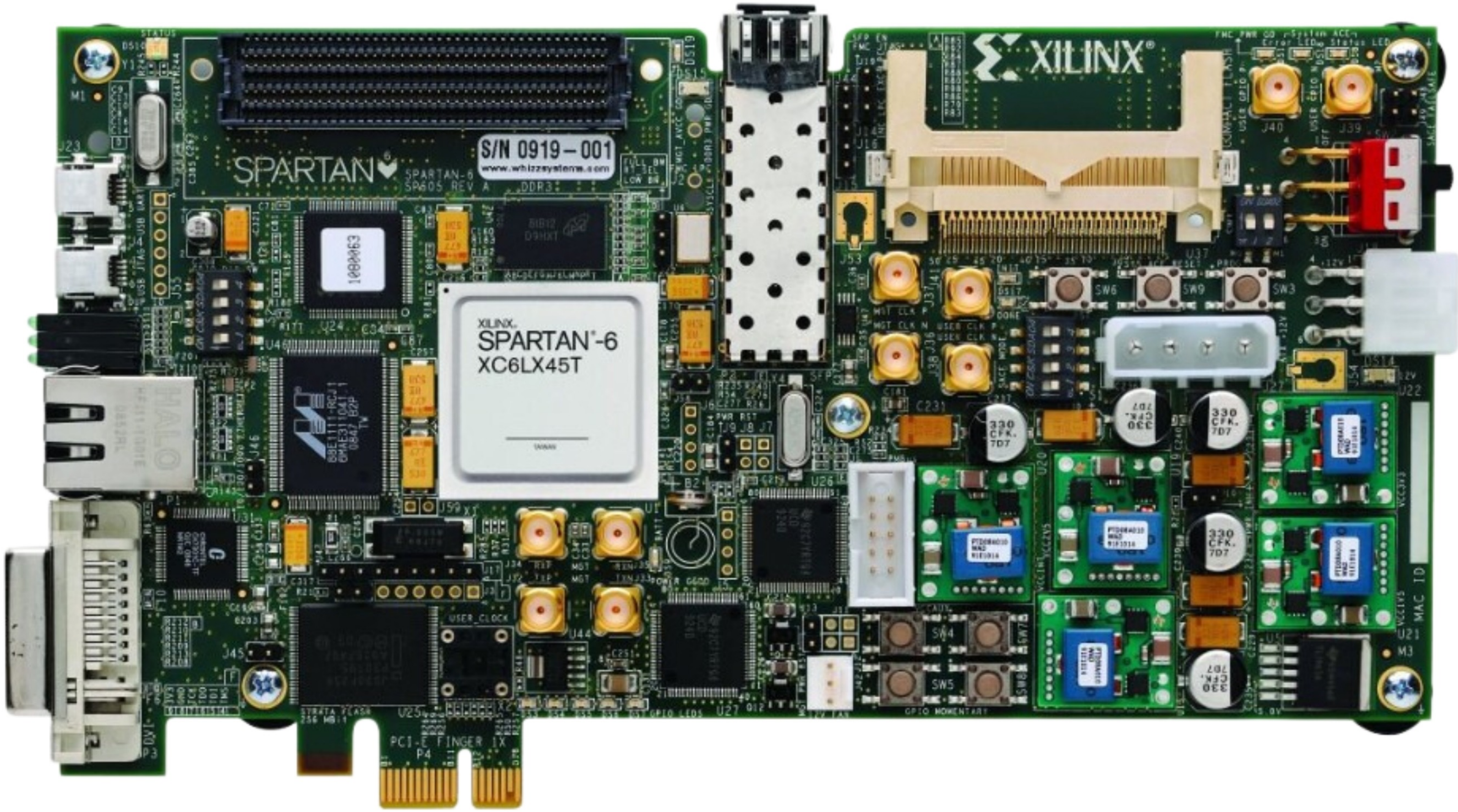
3-Input Lookup Table (LUT)



3-Input Lookup Table (LUT)

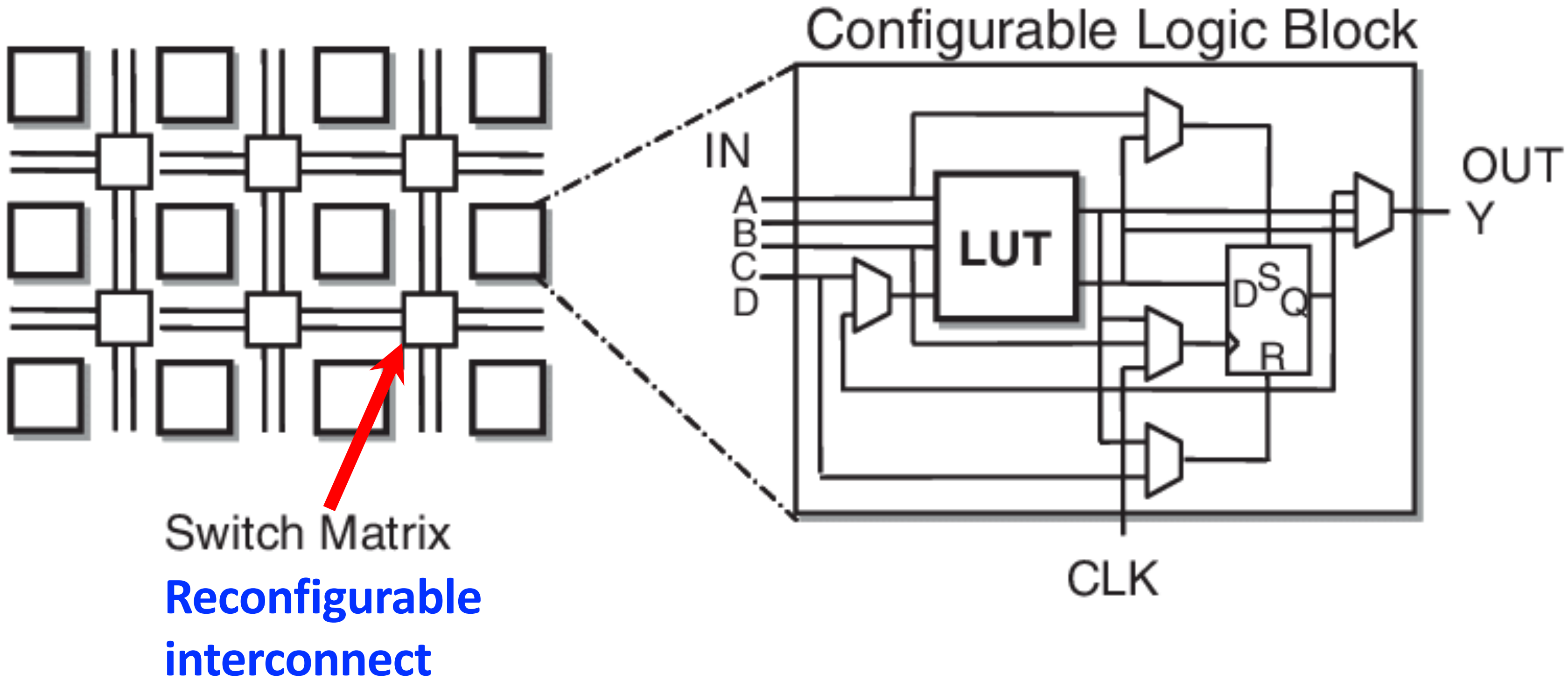


Modern FPGA



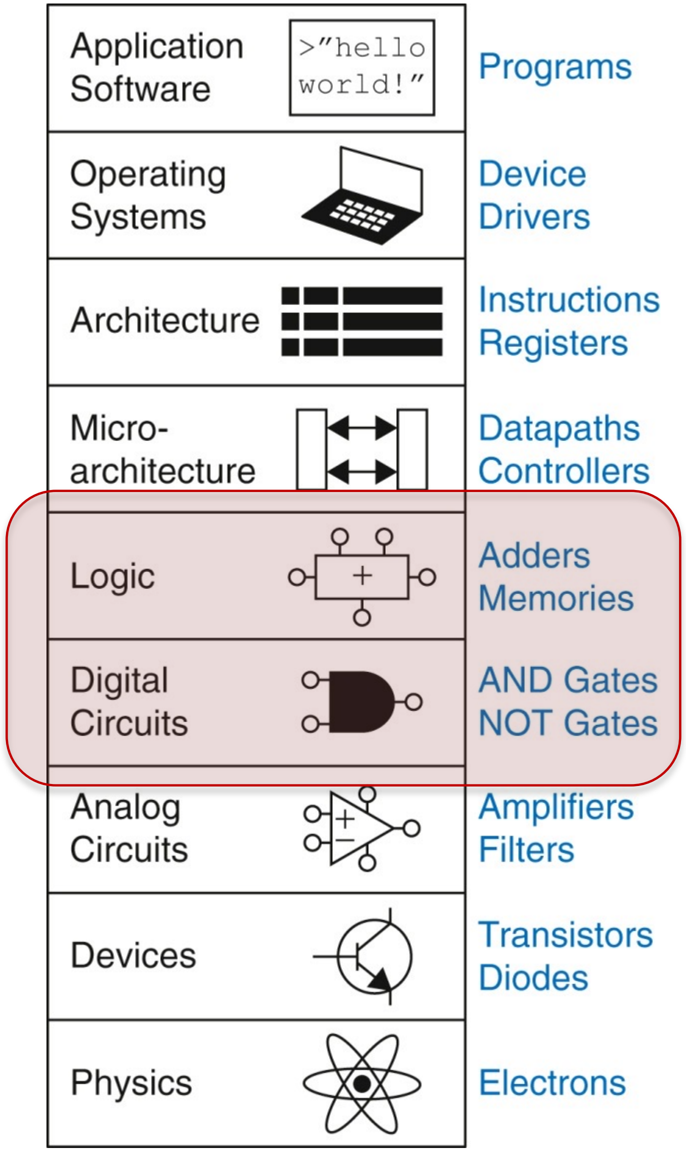
Modern FPGA

- Each 3-LUT performs the subset of the logic function (N is large)
- Signals are routed b/w CLBs using **reconfigurable** connections



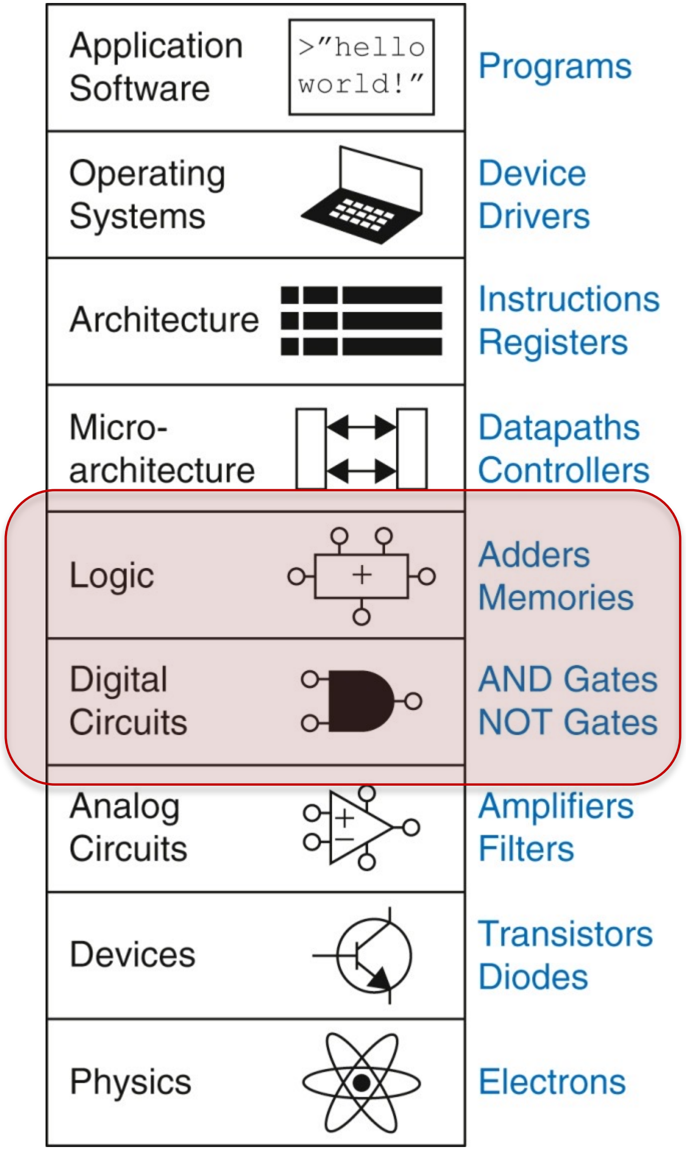
Topics Covered So Far

- 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



Continuing ....

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



Adders & Timing in Combinational Circuits

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

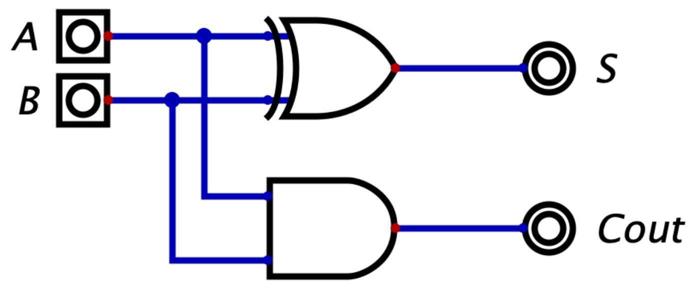
Boolean Eq

$S = A'B + AB'$

$S = A \oplus B$

$\boxed{C_{out} = AB}$

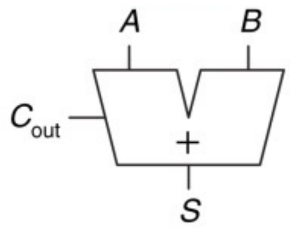
Schematic



Section 5.2.1 of H&H

Full Adder

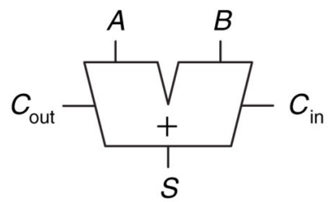
- **Limitation of half adder:** No carry input



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

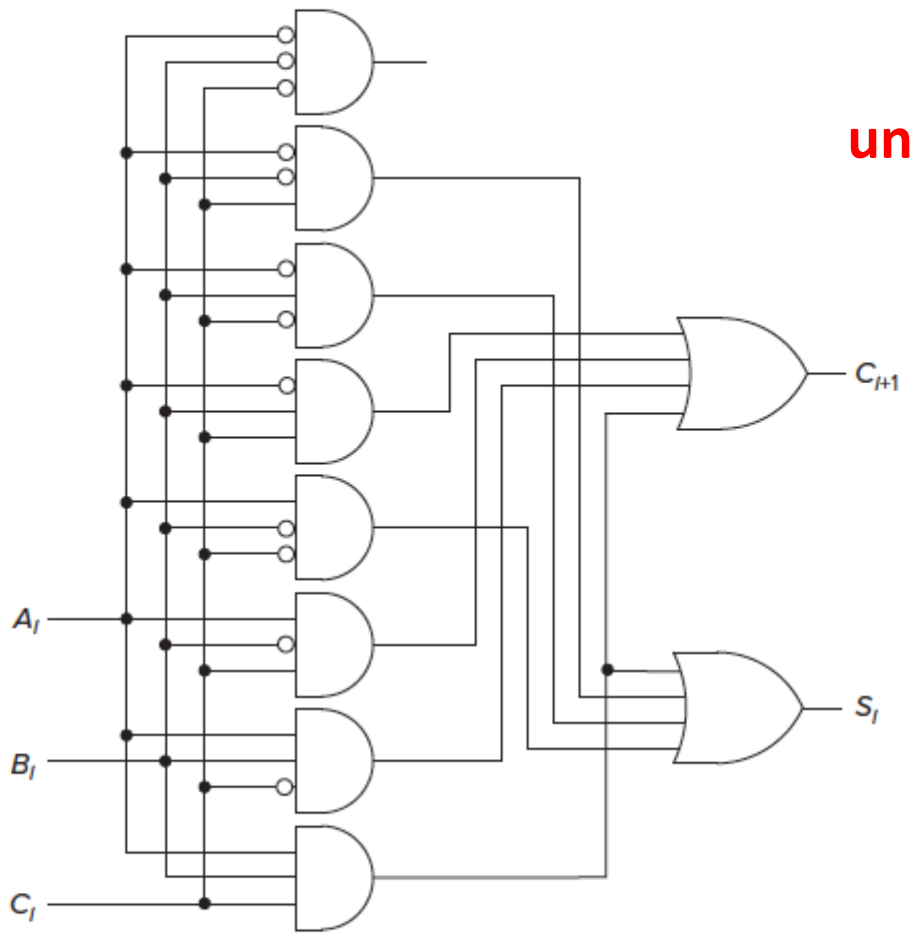
```
   1
  1001
+ 0101
------
  1110
```

- **Full adder** solves the problem



  - Accepts **three** inputs, including a carry input
  - Signals flow from right to left reflecting the carry propagation in arithmetic circuits

Full Adder: T. Table + Eq



Sum of products form != minimal form

unused minterm

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

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

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

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

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

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

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

<u>Simplification via Boolean algebra</u>

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

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

Full Adder: T. Table + Eq

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

Insight about C_{out}

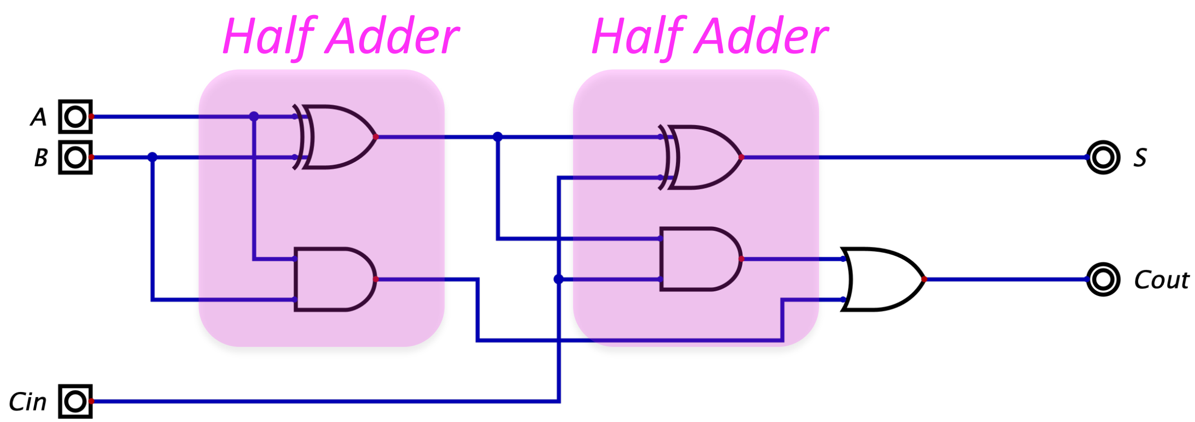
- 1 when both A and B are 1
  - Carry **G**eneration (**G**)
- 1 when there is a C_{in} and one of A and B is 1
  - Carry **P**ropagation (**P**)

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: Schematic

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

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



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

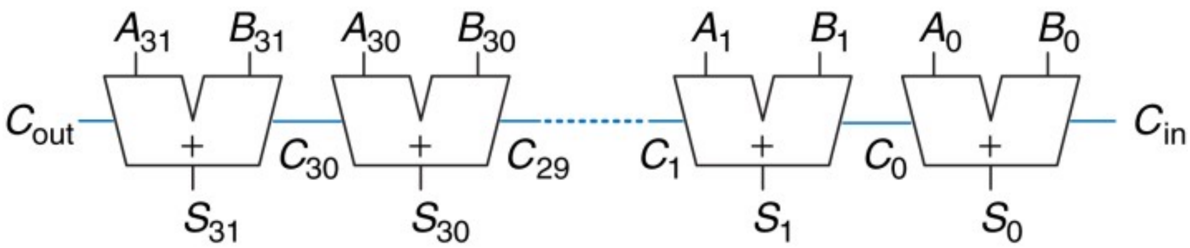
Ripple Carry Adder

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

$$\begin{array}{r} {\color{blue}1} \\ 1001 \\ +\quad 0101 \\ \hline {\color{blue}1110} \end{array}$$

Ripple Carry Adder

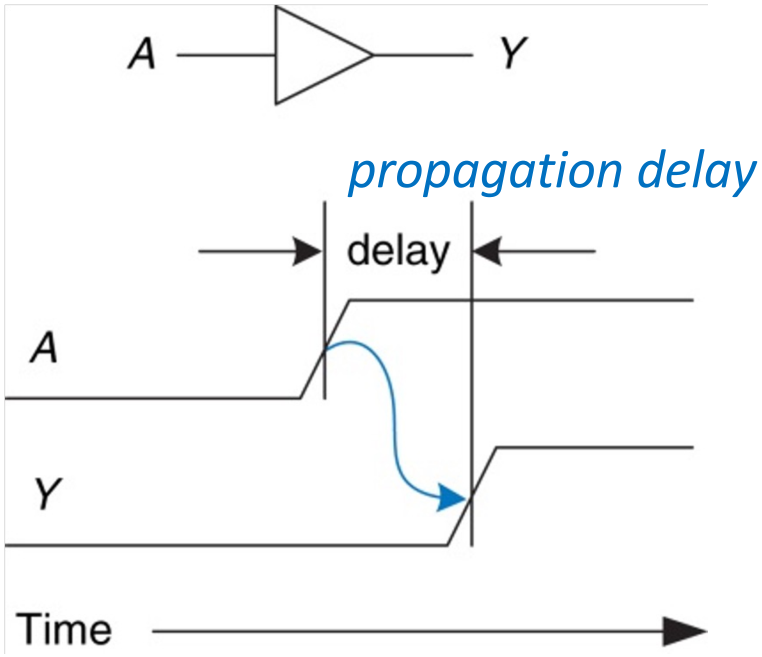
- What if we want to add two N-bit numbers?
  - Connect a chain of full adders from right to left



- **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)



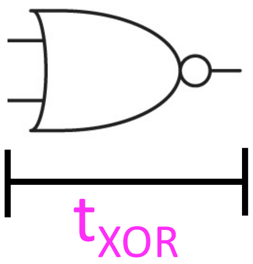
Section 2.9 of H&H

Examples

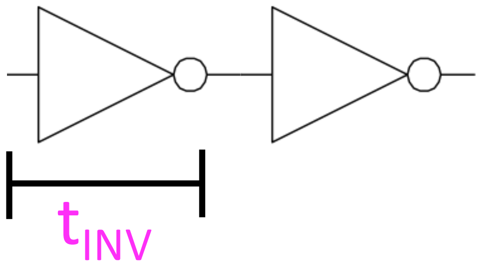
- **Inputs of the AND gate change from (0,0) to (1,1)**
  - Output of AND gate change from 0 to 1
  - How long does it take to for the output to change?

- **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}?

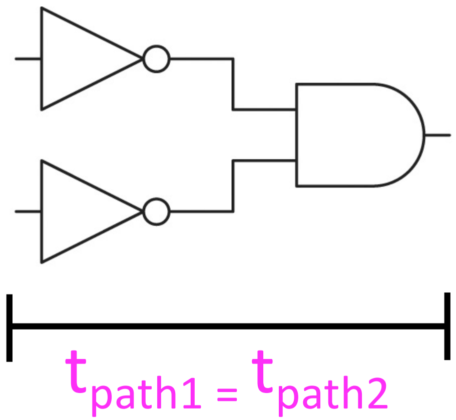
Examples of Timing/Delay



Each gate has a delay



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



Multiple paths from input to output

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

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

Critical and Shortest Path

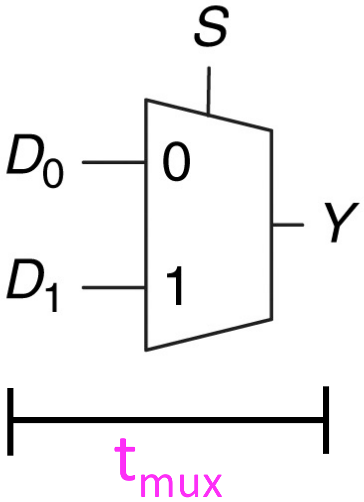
- Most useful combinational circuits have multiple paths from input to output
  - **Critical path:** The slowest path (with **longest delay**)
  - **Critical path limits the speed at which the circuit operates**
  - In contrast, the **shortest path** is the fastest

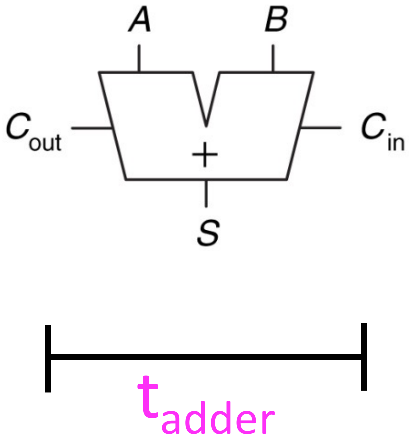
- For simplification, we will ignore the delay of nodes (wires)
  - Although the delay is non-trivial, it is studied best at the analog level of abstraction

Section 2.9 of H&H

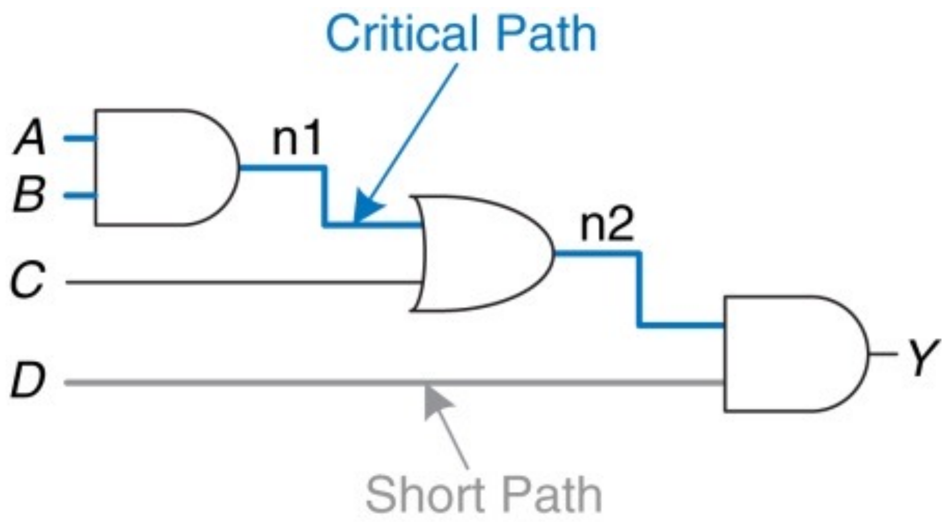
Multiplexer and Adder Delay

- Assume component-level delay and don't worry about delay of individual gates (unless necessary)





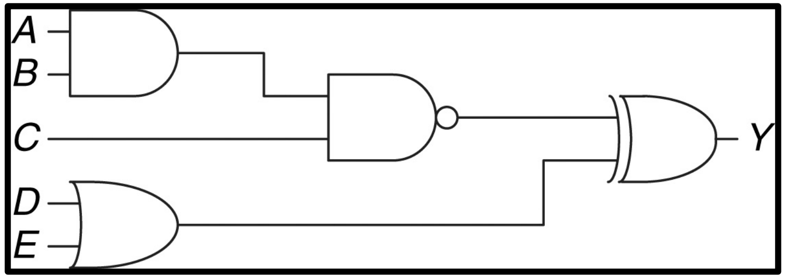
Example (1)



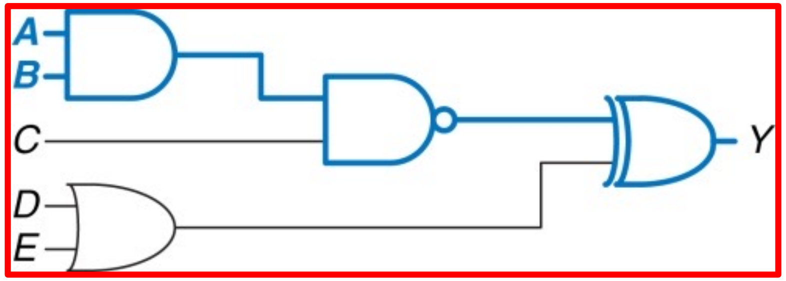
- The **propagation delay** of a combinational circuit is the sum of the propagation delays through each element on the critical path

Example (2)

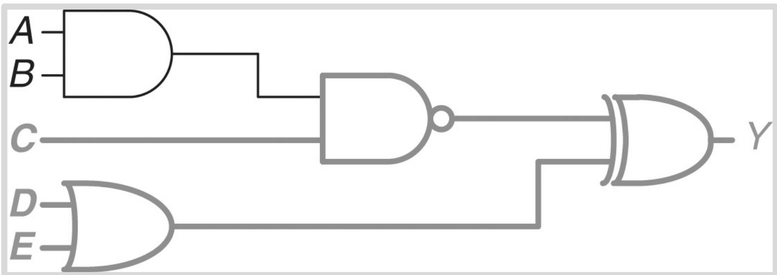
Example Circuit



Critical Path

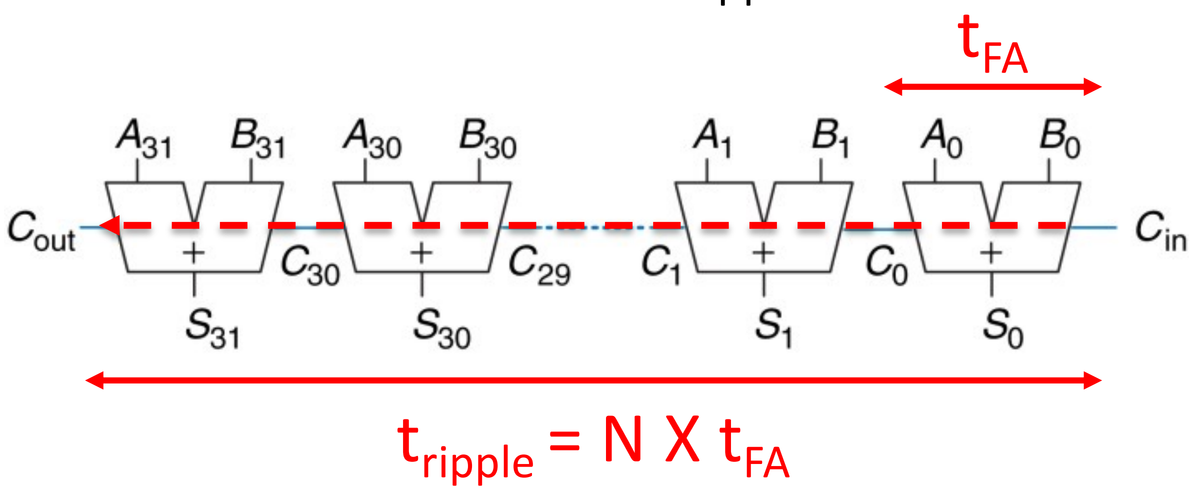


Shortest Path



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 and insight and keep the delay reasonable
  - There is aways 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
  - Recursively combine **G** and **P** signals to compute the carry out

Section 5.2.1 of H&H

CLA Equations

one column

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1}$$

$$G_{3:0} = G_3 + P_3(G_2 + P_2(G_1 + P_1G_0))$$

$$P_{3:0} = P_3P_2P_1P_0$$

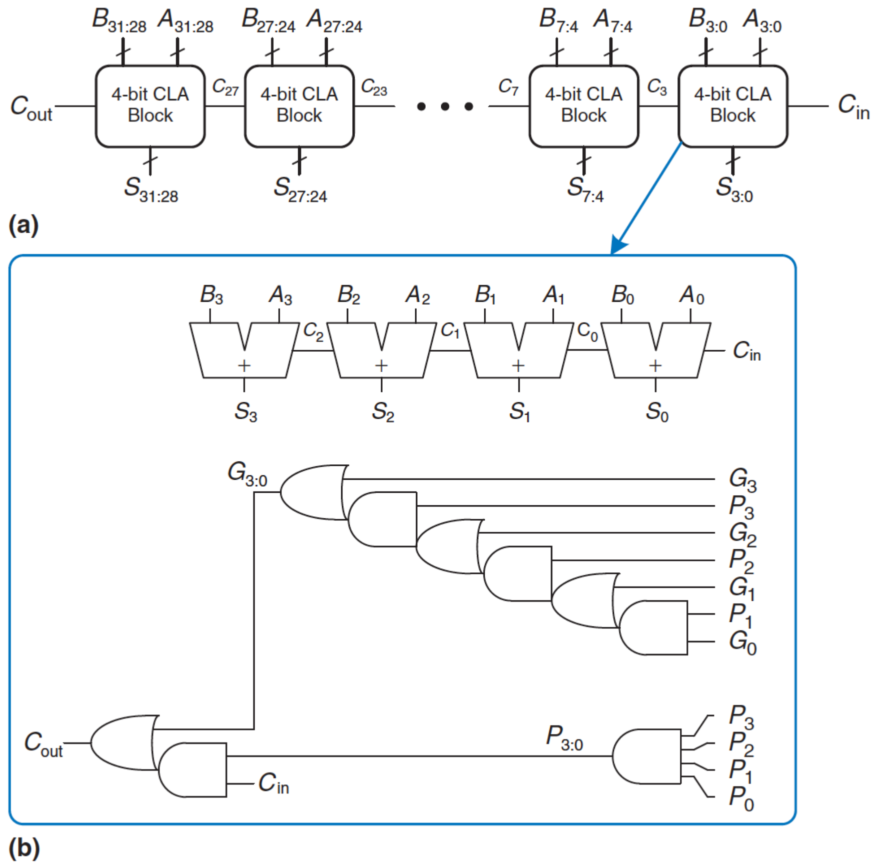
$$C_i = G_{i:j} + P_{i:j}C_{j-1}$$

4-bit block

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

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 (in parallel)

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

$$\boxed{t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{k} - 1\right) t_{AND\_OR} + k t_{FA}}$$

Lessons from CLA

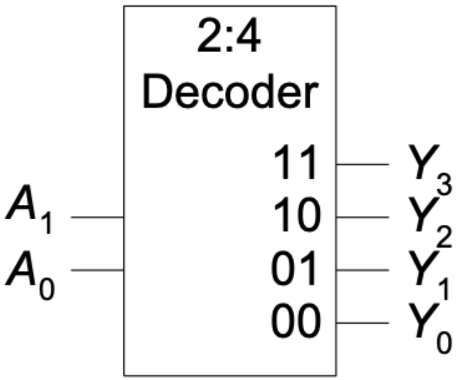
- **Speed-Area Tradeoff:** In digital systems, there is a tradeoff b/w performance (speed) and hardware cost (area/power)
  - **CLA speeds up addition but requires extra logic gates that take up additional area and dissipate more power**

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

Decoders

Decoders

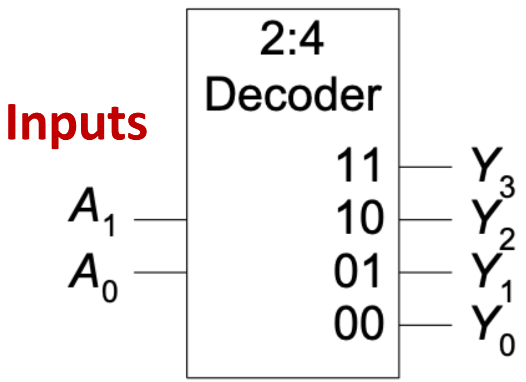
- N inputs and 2^N outputs
- For each input combination, only one of the outputs is 1 **(one-hot)**
  - It detects an input pattern and outputs a 1 corresponding to it



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



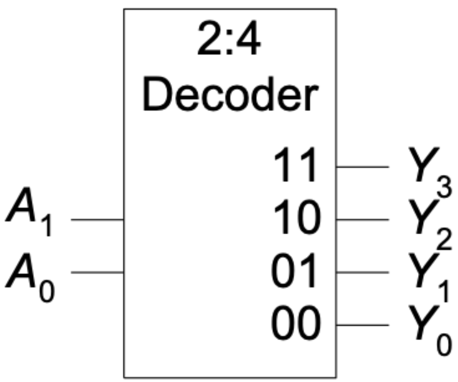
<u>2:4 Decoder Truth Table</u>

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

$Y_0 = A_1'A_0'$

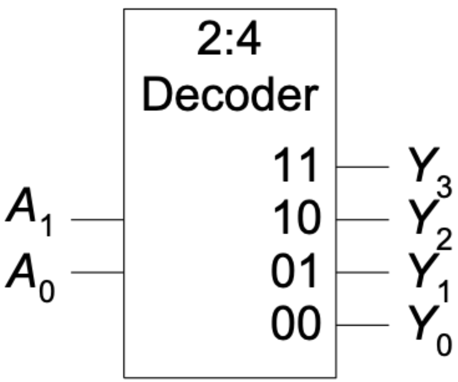
$Y_1 = A_1'A_0$

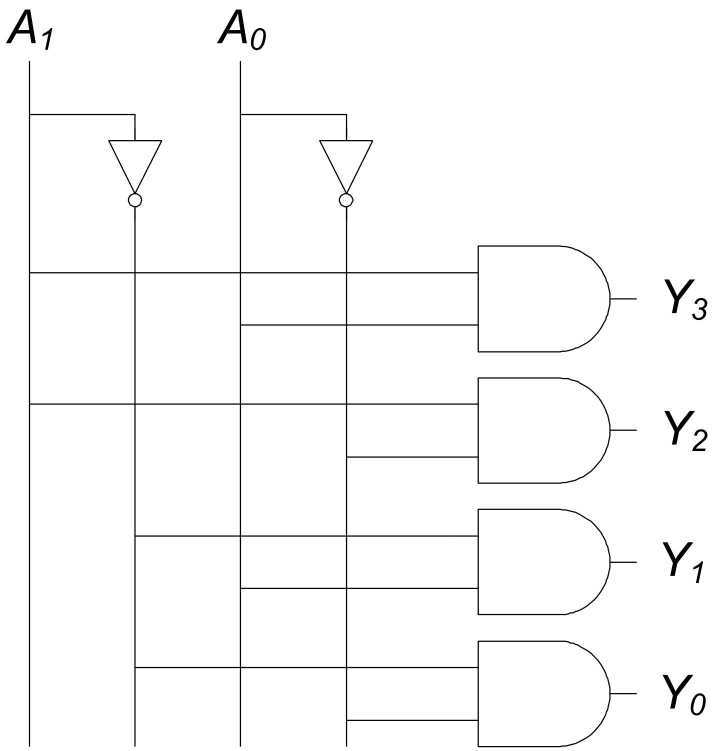
$Y_2 = A_1A_0'$

$Y_3 = A_1A_0$

Decoders

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





$Y_0 = A_1'A_0'$

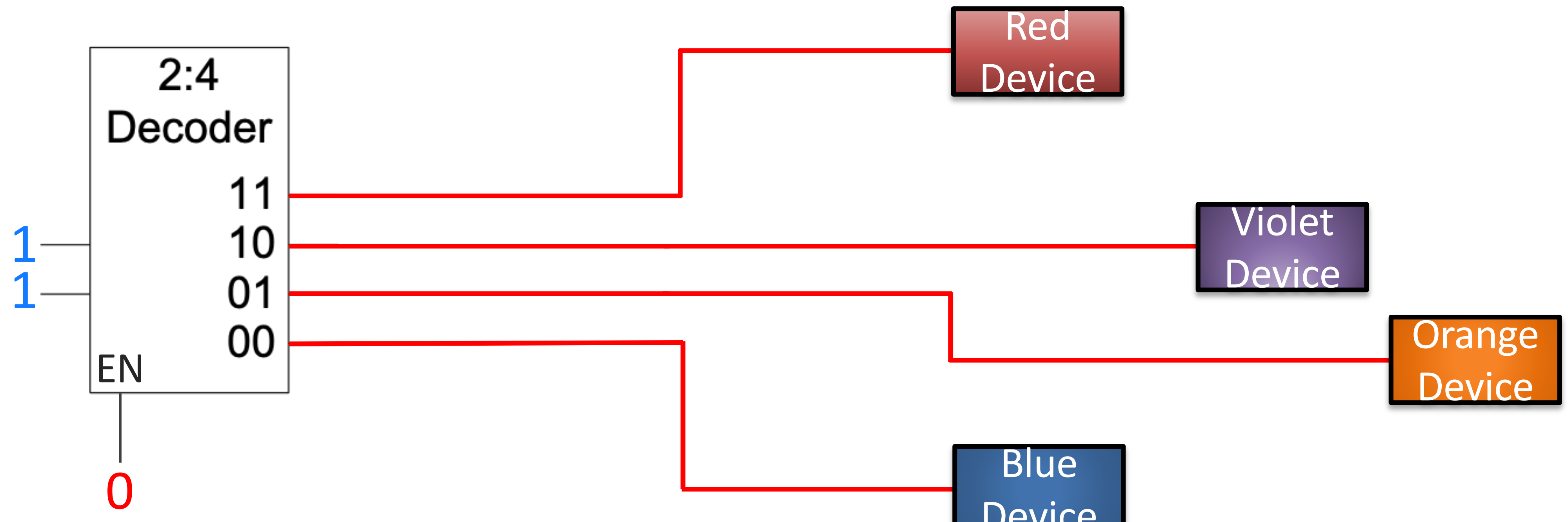
$Y_1 = A_1'A_0$

$Y_2 = A_1A_0'$

$Y_3 = A_1A_0$

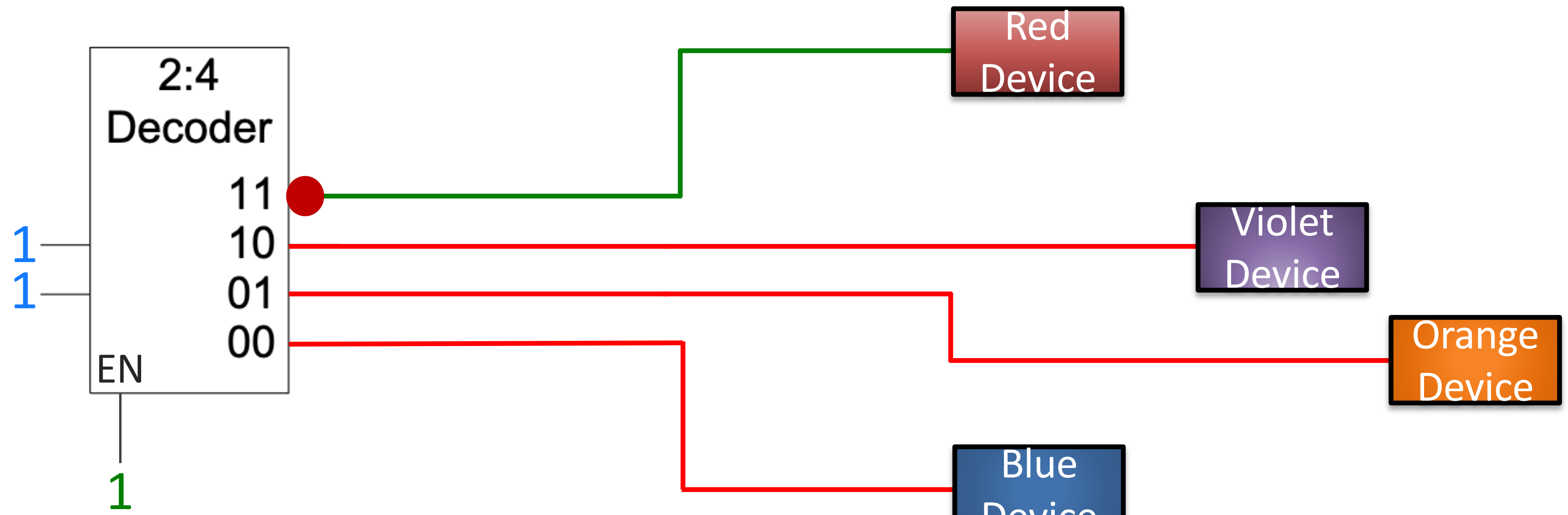
Uses of Decoders

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



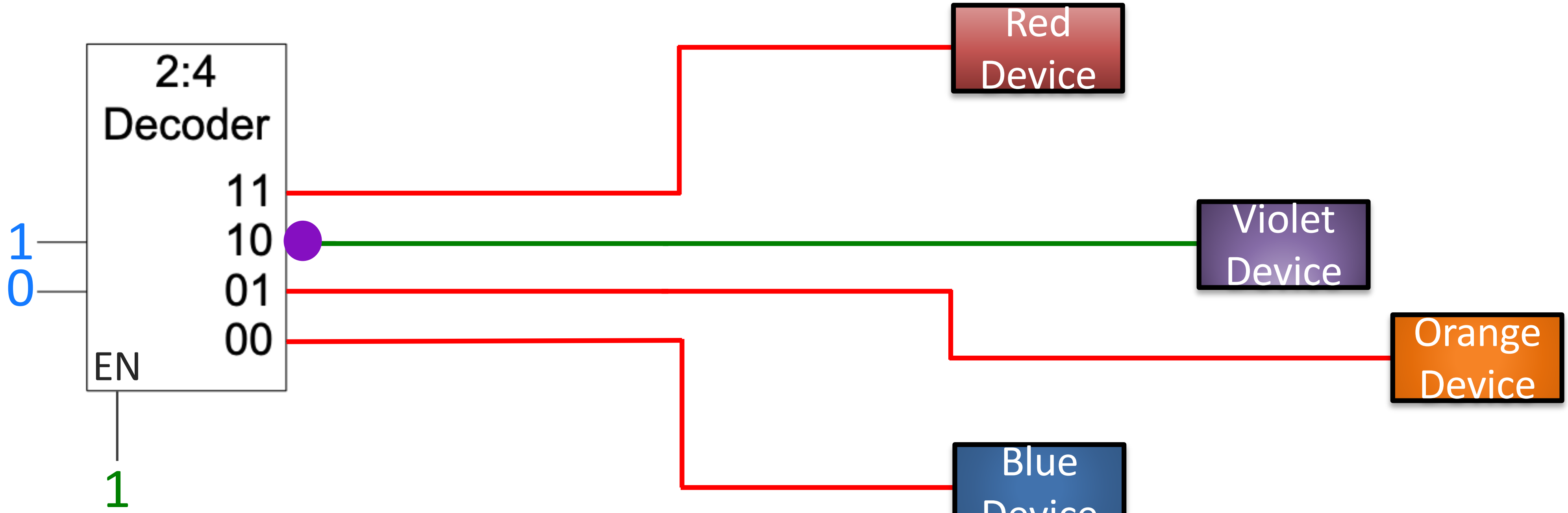
Uses of Decoders

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



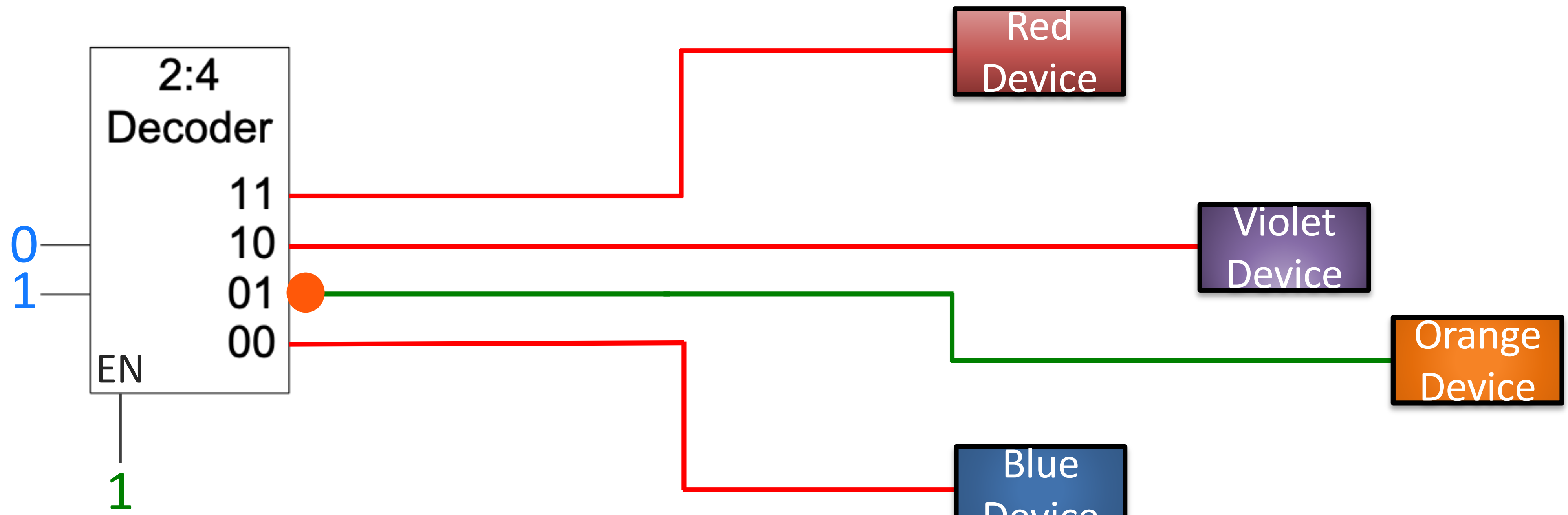
Uses of Decoders

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



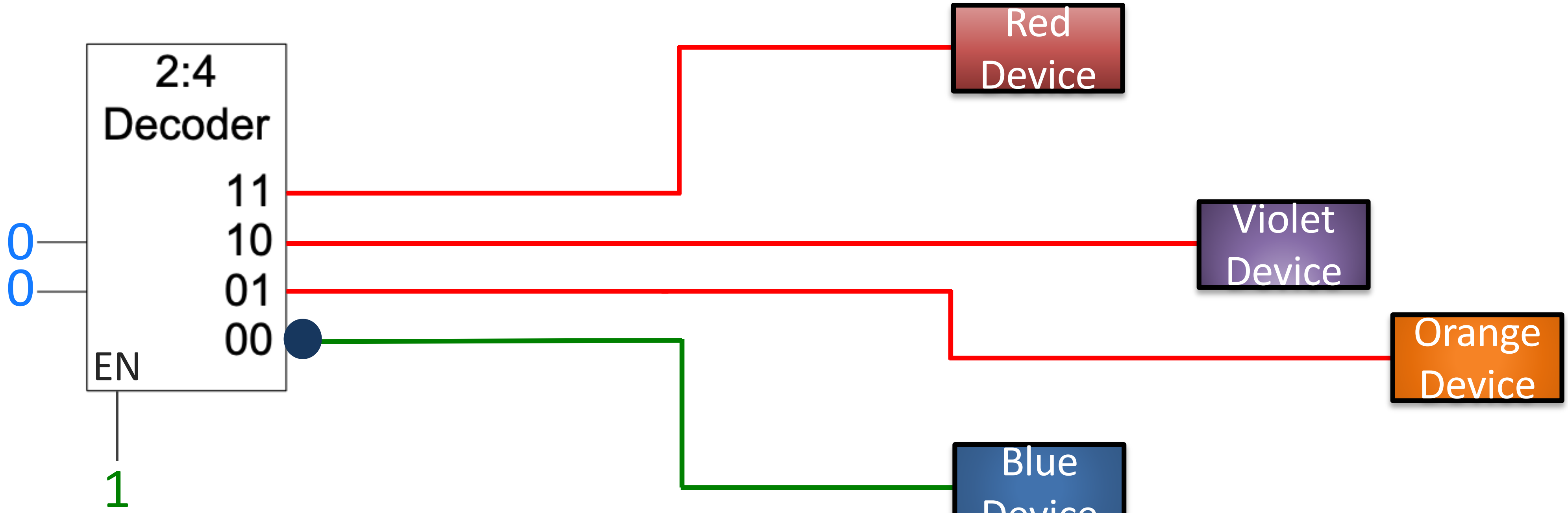
Uses of Decoders

- 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 different devices
  - Each device reacts to a specific instruction in a specific way

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

- We will need the decoder for building the control unit of our QuAC computer that decodes instructions

Logic Using Decoders

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

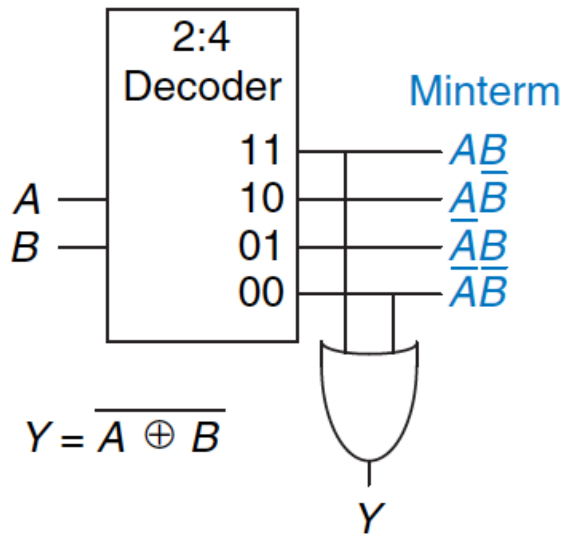


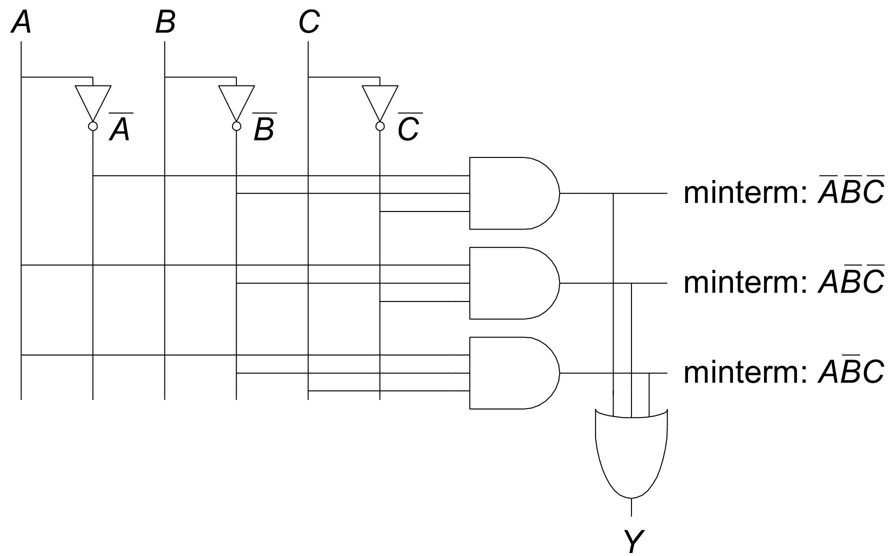
Figure 2.65 **Logic function using decoder**

PLA

Programmable Logic Array (PLA)

- SOP (sum-of-products) leads to two-level logic

- Example: **Y = A'B'C' + AB'C' + AB'C**

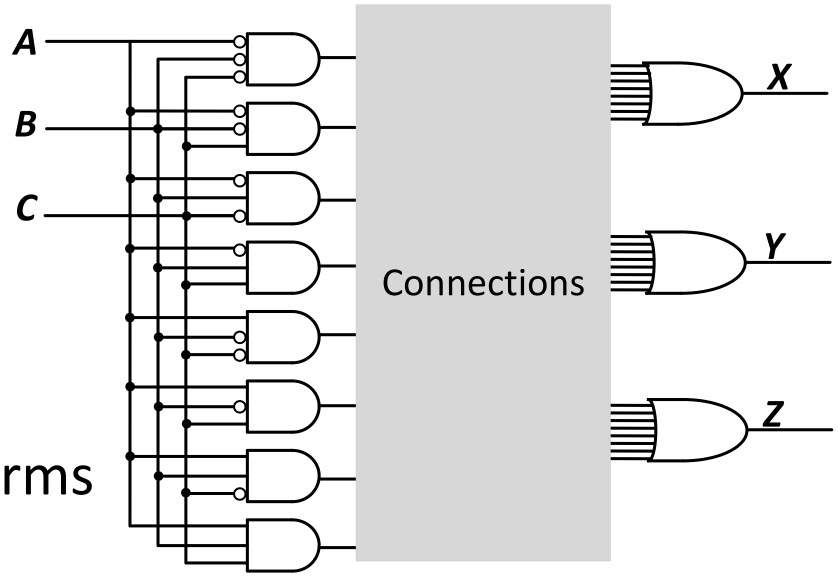


- We can use a PLA to implement **any** N-input P-output function
- PLA is built once in the factory and programmed later in the house to implement any logic function

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

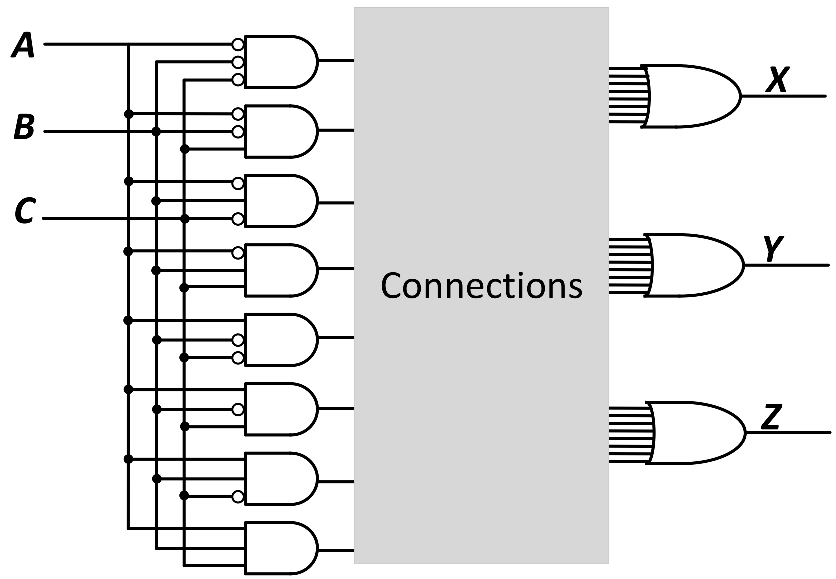
- How many OR gates?
  - The number of output columns in the truth table

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

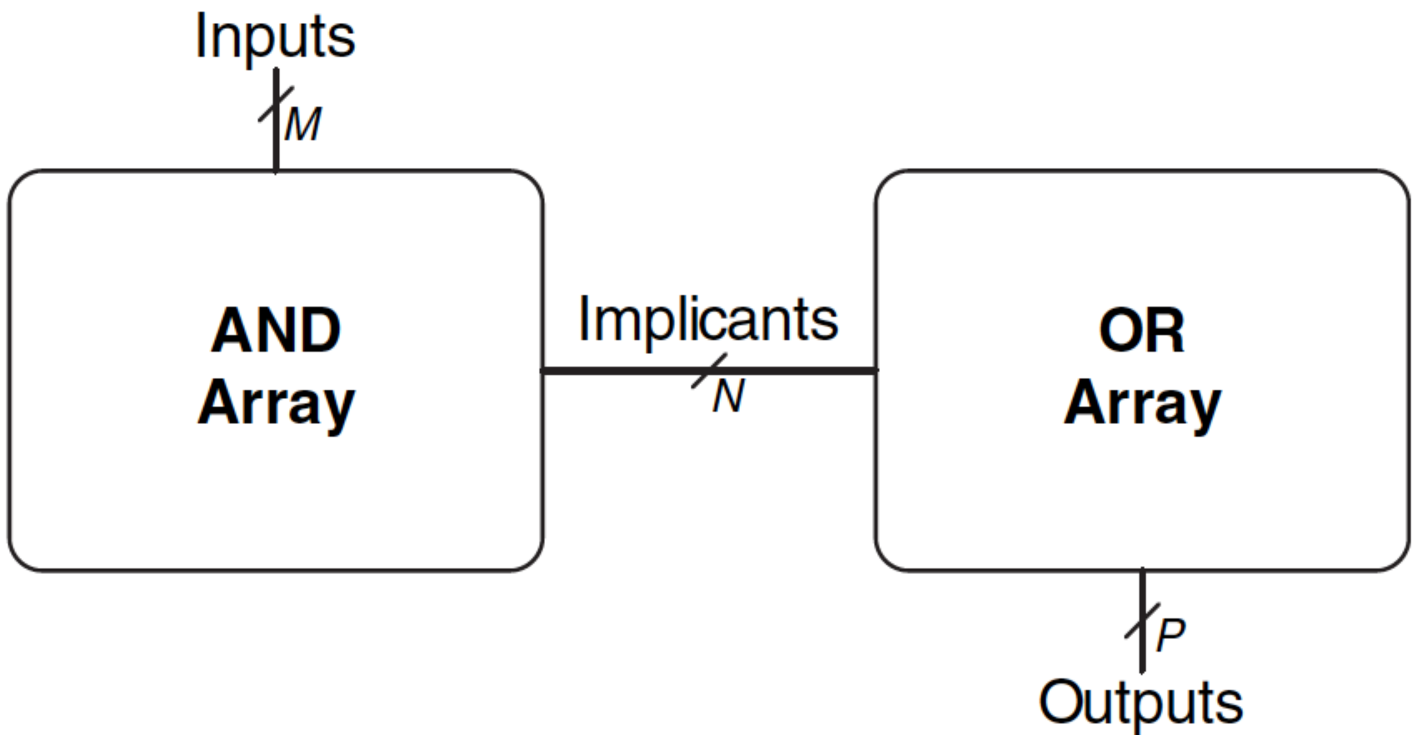


Section 5.6.1 of H&H

Programmable Devices

- Programmable devices we have talked about
  - CPU/processor (programmed using instructions stored in memory, aka, executable file)
  - FPGA (programmed by storing bits inside LUTS, aka, bit file)
  - PLA (programmed by burning fuses)

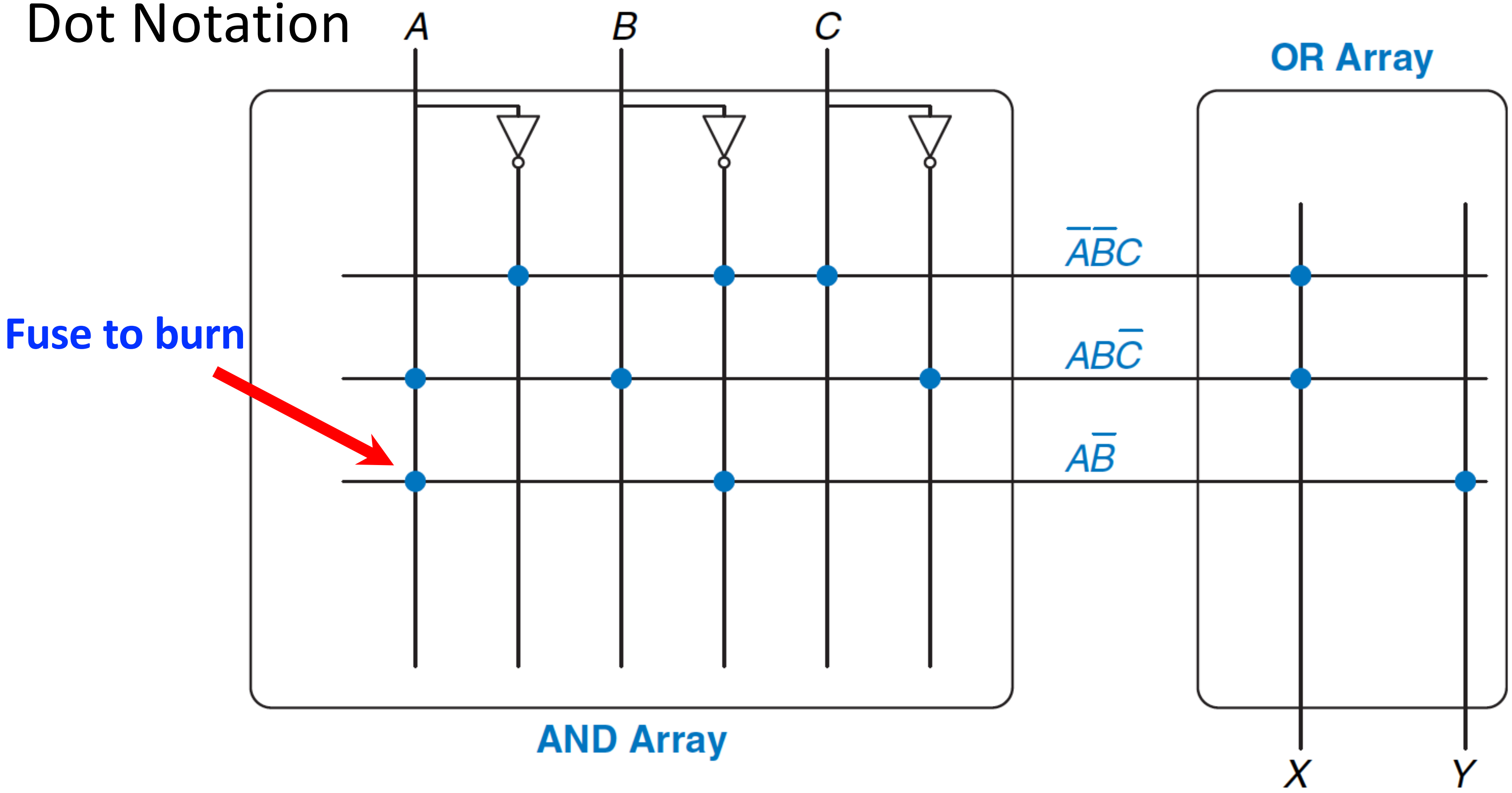
PLA Example (I)



- **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**

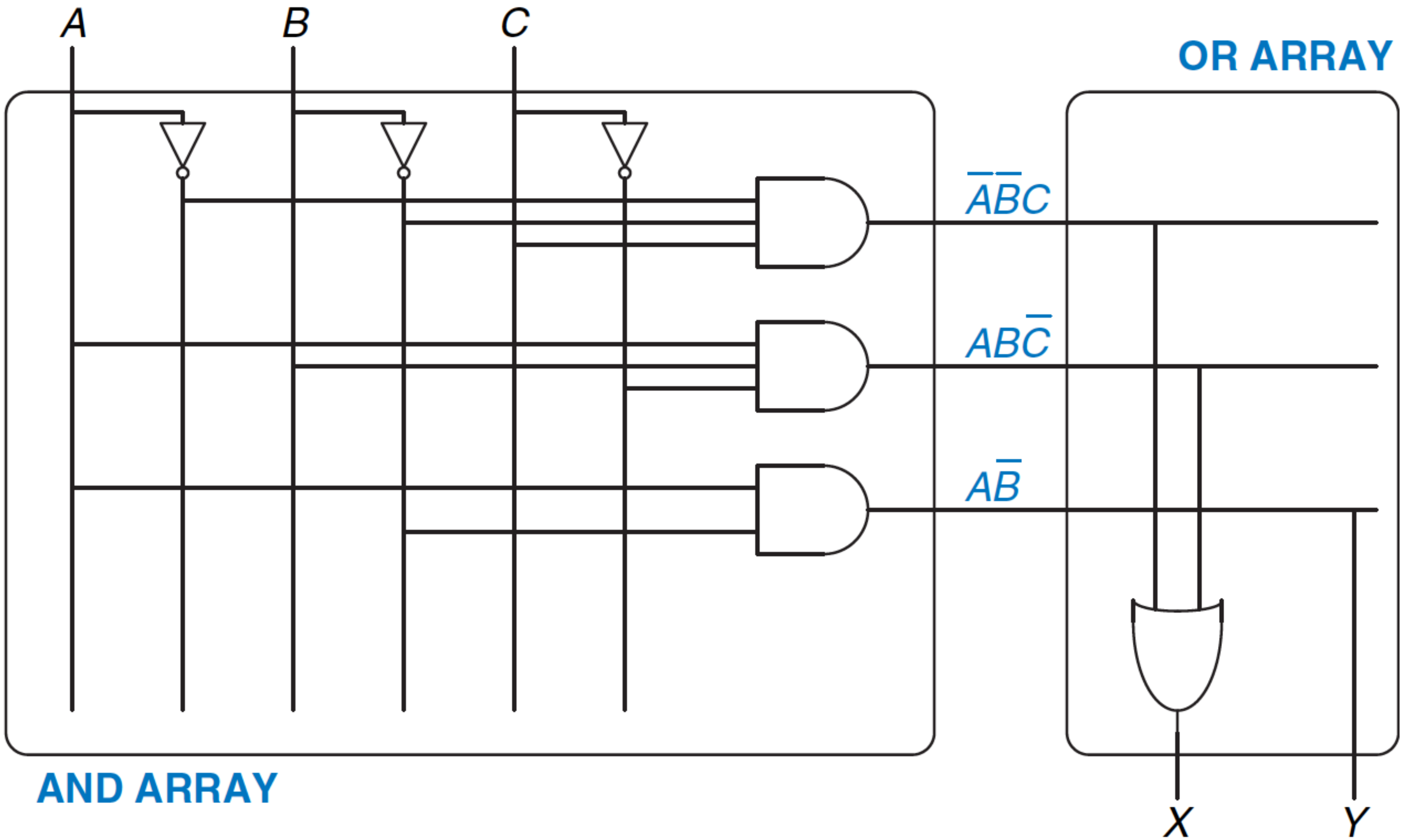
PLA Example (II)

Dot Notation



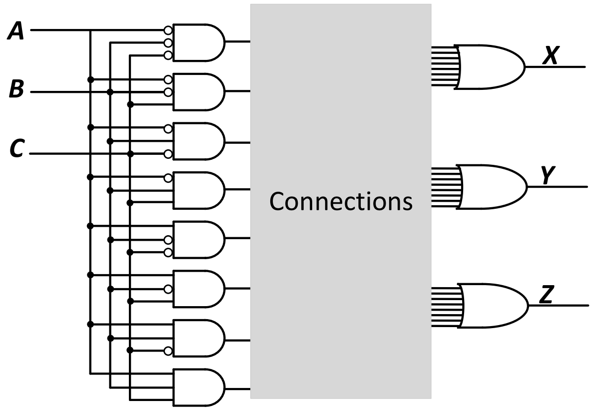
Section 5.6.1 of H&H

PLA Example (III)



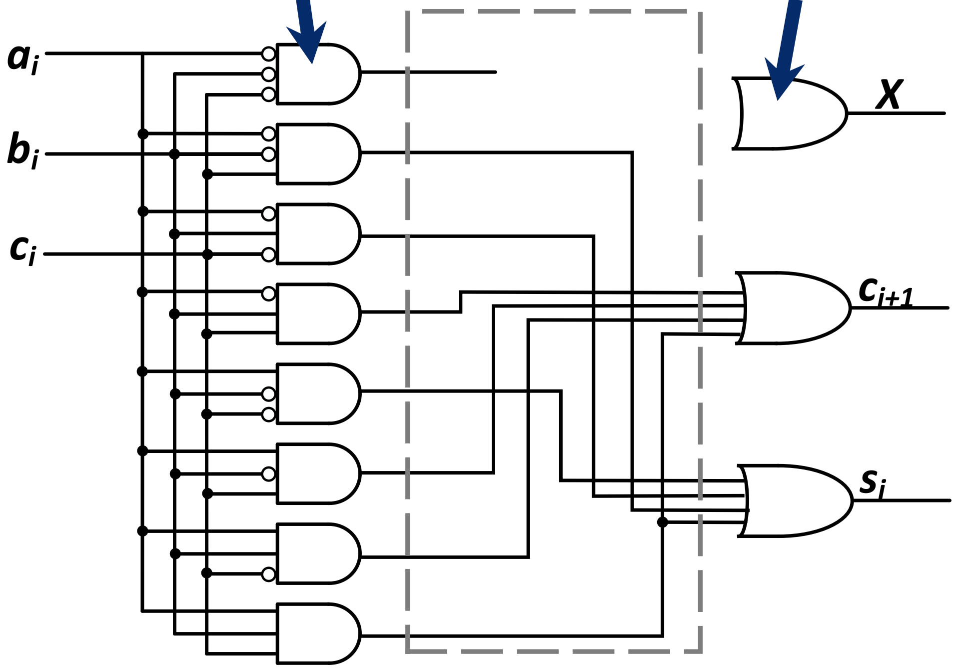
Implementation: Pick the **literals** & **implicants** by programming connections

Full Adder Implementation w/t PLA



This input should not be connected to any outputs

We do not need this output



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

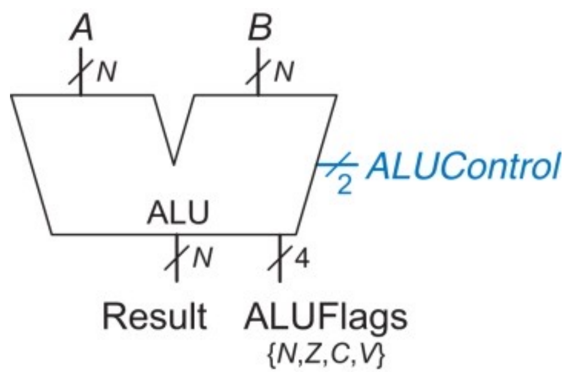
Lessons from PLA

- **Programmability:** Programmable devices incur a **cost**
  - Some logic in PLA is redundant if a subset of **minterms** are needed
  - On the other hand, PLAs can be **programmed** after **bulk** manufacturing which is their key **programmability advantage**

ALU

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



N-bit ALU

Section 5.2.4 of H&H

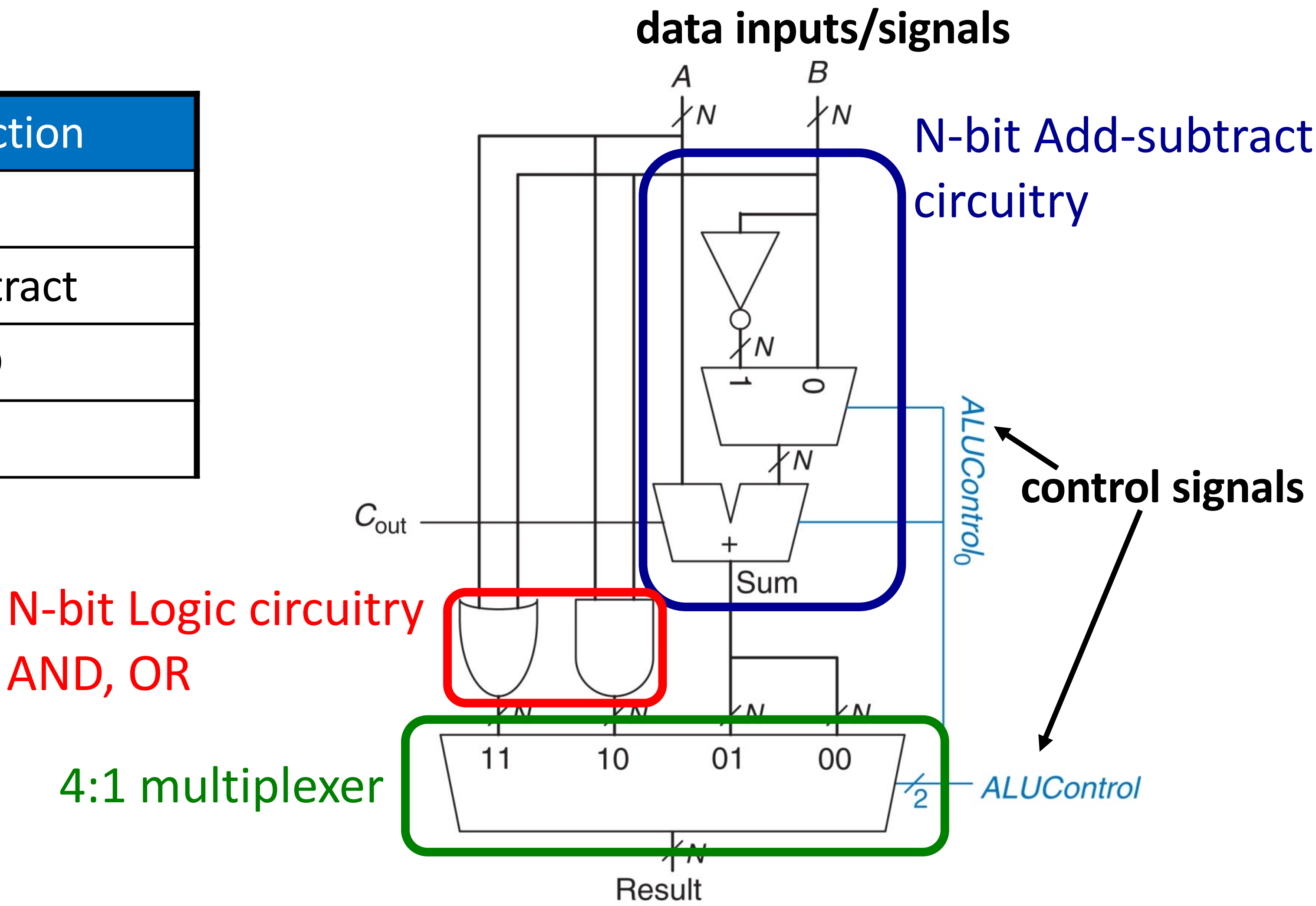
ALU Interface/Instructions

- N-bit data inputs and outputs
- 2-bit control input (**ALUControl**)
  - Specifies one of four functions
  - 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

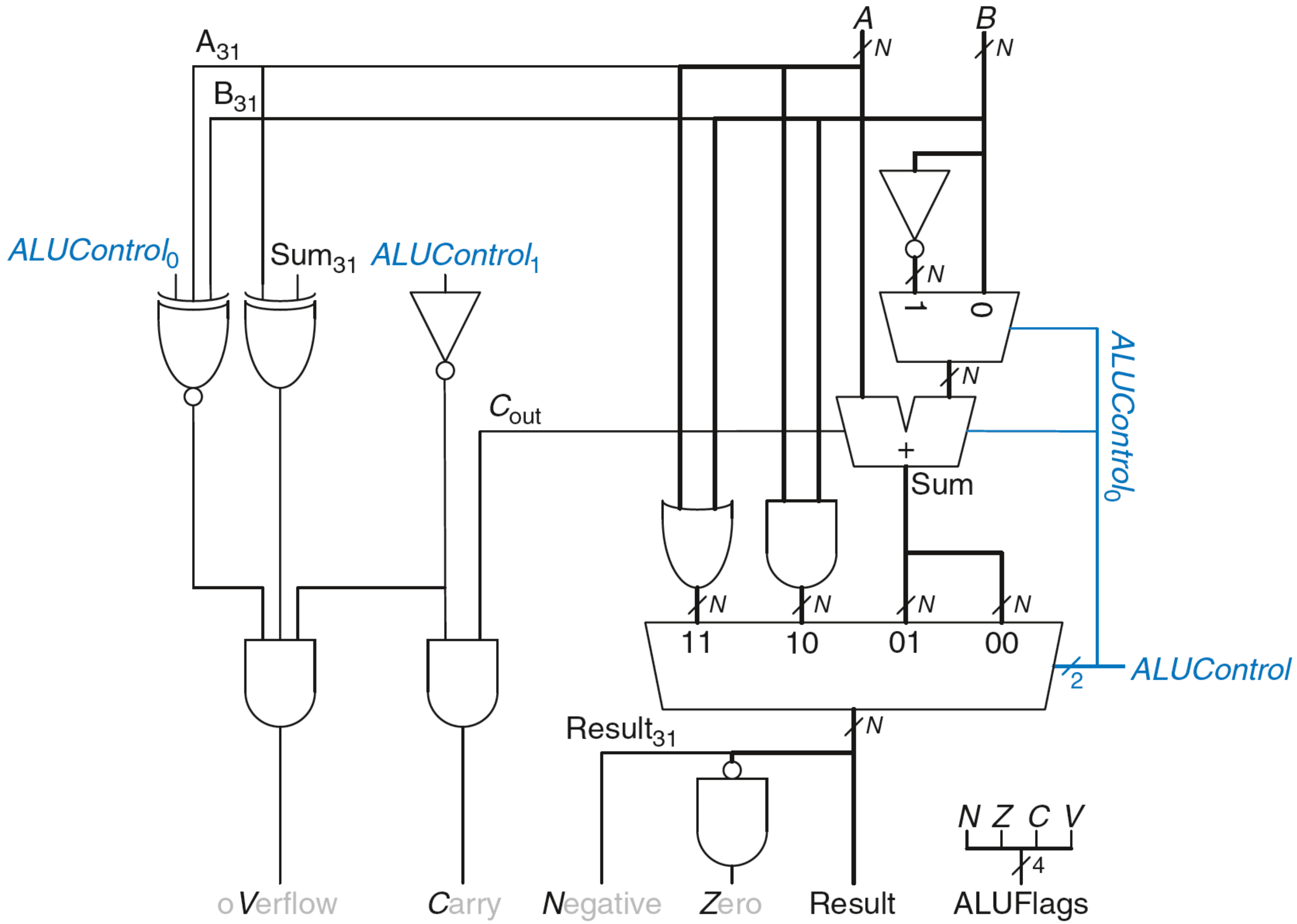
Option # 1 for Overflow

- 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

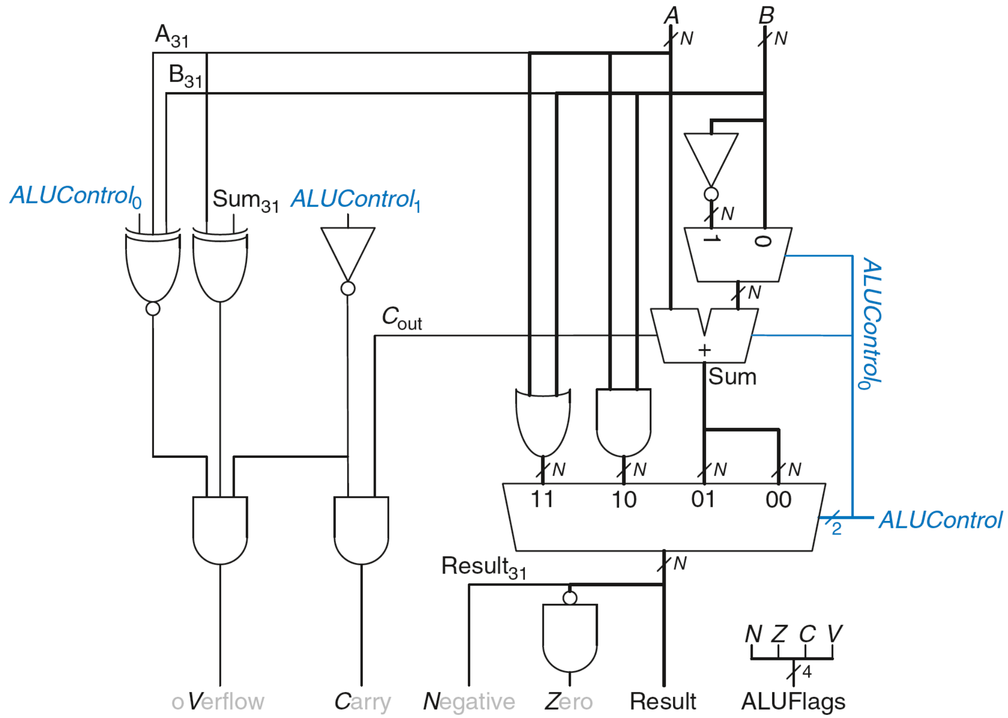
- Use A and the output of 2:1 mux
  - B if the instruction is an Add and –B if the instruction is a subtract
- Easy to reason conceptually
  - If A – B is the same as A + (–B) then everything is an add
  - There is no need to consider subtract separately when reasoning about overflow generation
- The 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

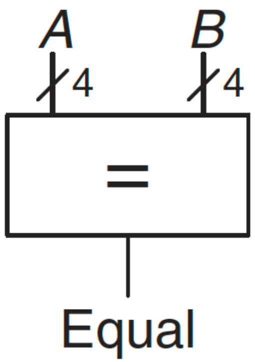


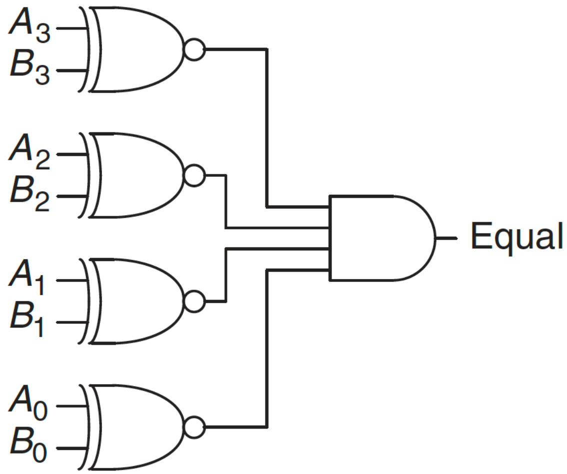
- Find t_{Result} in ps for the four ALU functions. (Ignore overflow generation)
    - **Which function takes the longest time (and is the critical path)? Ignore wire delay**
- Express t_{Result} in the form of an equation for Add and Subtract. What is the difference?

Comparator

Comparator (Equality Checker)

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





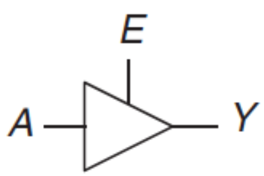
- **What about magnitude comparison (relative values of A and B)?**

Tri-State Buffer

Tri-State Buffer

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

Tristate Buffer



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

Figure 2.40 Tristate buffer

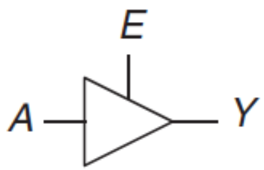
A tri-state buffer acts like a switch but can pass both 0's and 1's if E is asserted

Section 2.6.2 of H&H

Tri-State Buffer

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

Tristate Buffer



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

Figure 2.40 Tristate buffer

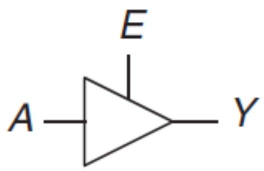
A tri-state buffer acts like a switch

- 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

Tristate
Buffer



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

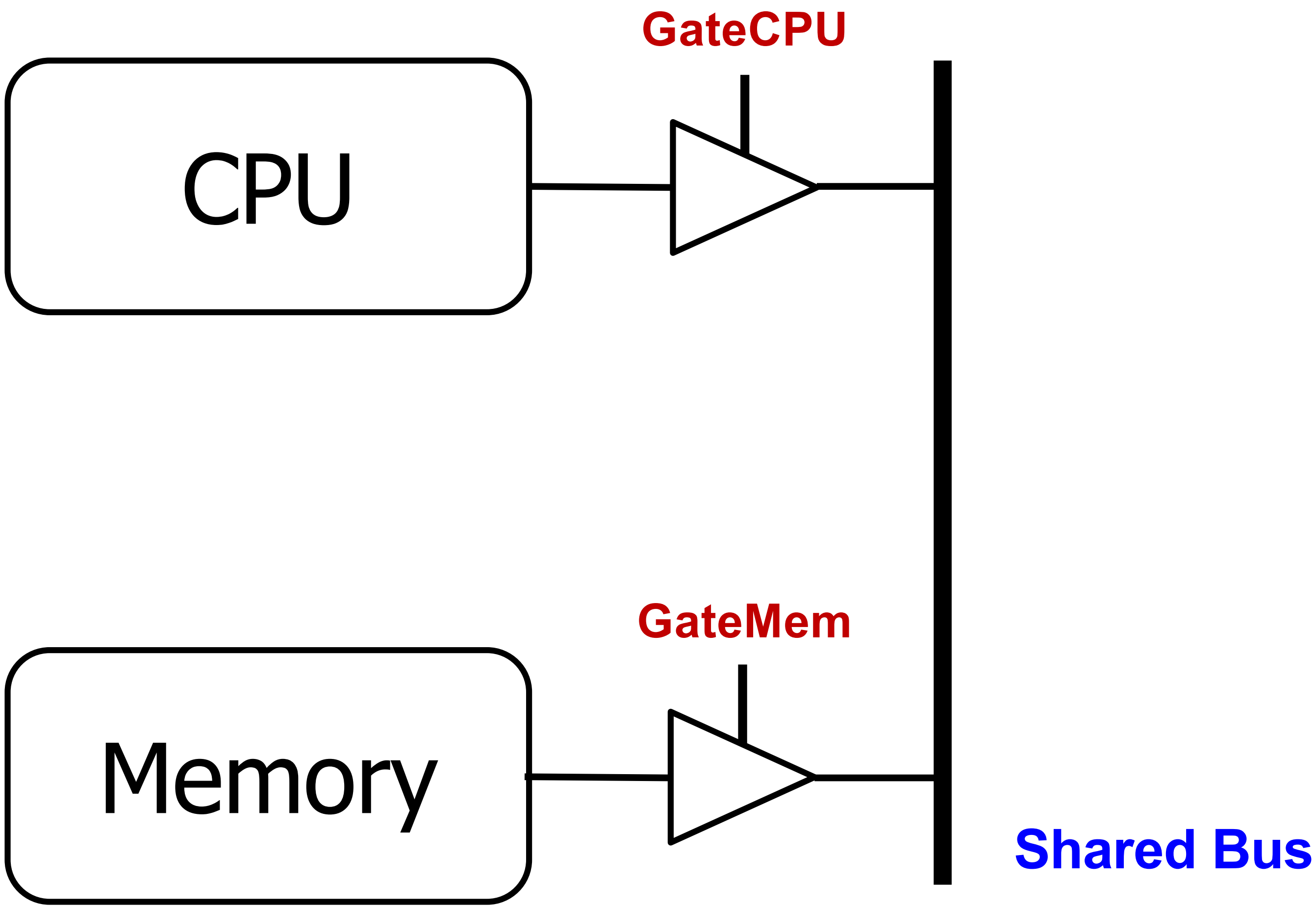
Use of Tri-State Buffers

- Imagine a wire shared by the CPU and memory or two I/O peripherals

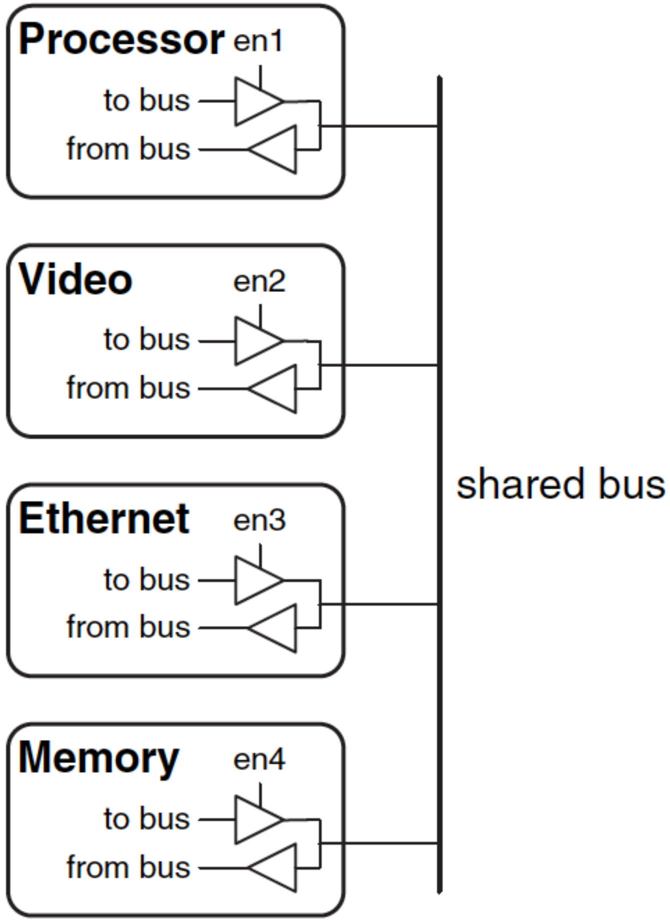
- At any time, **only one of them can place a value on the wire, but not both**

- Use two tri-state buffers
  - One driven by CPU, and one driven by memory
  - Ensure at most one is enabled at any time

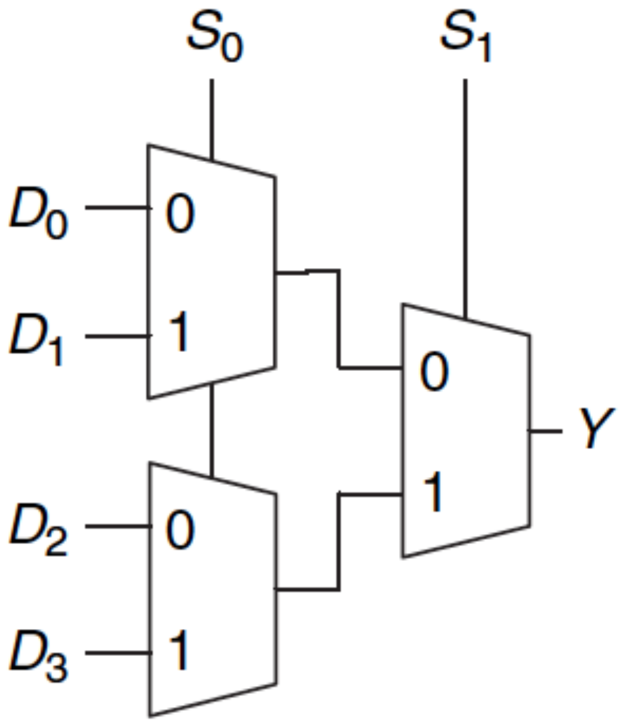
Example: Use of Tri-State Buffers

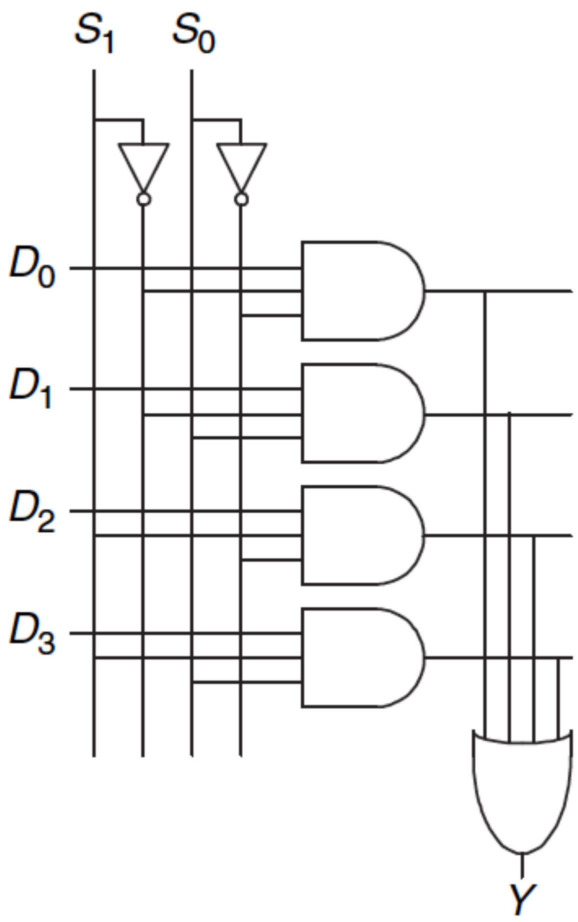


Another Example

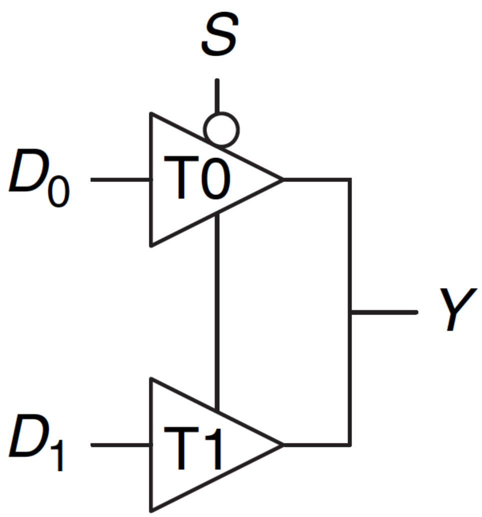


Recall: A 4:1 Multiplexer



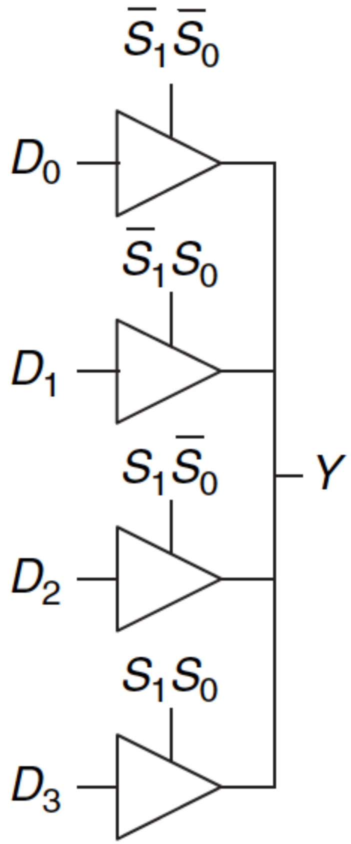


Multiplexer Using Tri-State Buffers



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

Figure 2.56 **Multiplexer using tristate buffers**

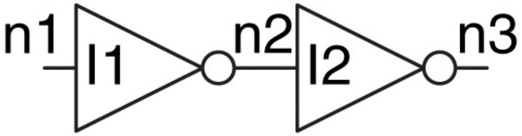


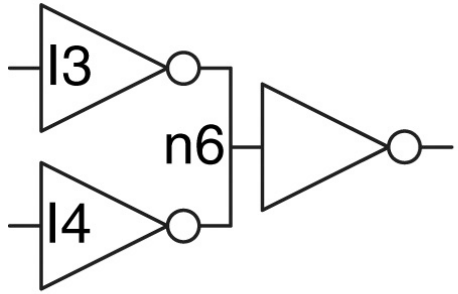
Combinational Composition Rules

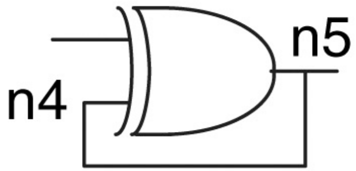
Combinational Composition Rules

- 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**
- The circuit contains **no cyclic paths**. Every path through the circuit visits each circuit node at most once

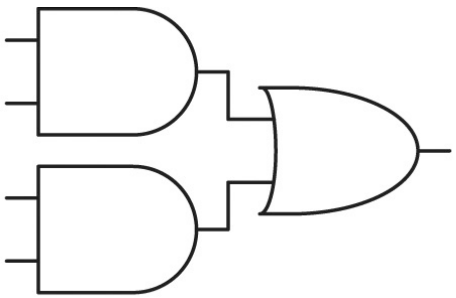
Which circuits are combinational?

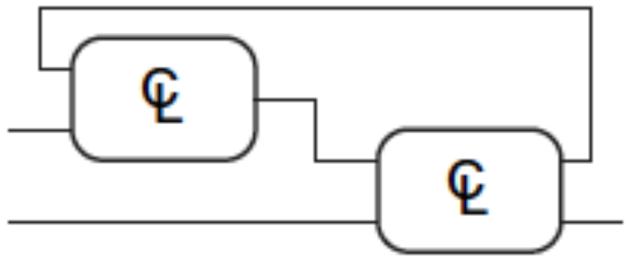






Assume n5 is 0
and the other
input of XOR is 1





We Study Boolean Algebra for

Logic Minimization

Because we care about minimizing area, cost, logic complexity, energy, footprint, ...

Boolean Algebra (Logic Minimization)

- The **sum-of-products (SOP)** canonical form does not lead to the simplest logic gate implementation
- We can eliminate some **minterms** → Less # logic gates
- We can reduce the **# literals** in minterms → Smaller gates
- We use Boolean algebra to simplify Boolean equations
  - Similar in spirit to simplification in ordinary algebra except we are dealing with 0 and 1 **(much easier)**

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 $B \neq 1$	$B = 1$ if $B \neq 0$	Binary Field
A2	$\overline{0} = 1$	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND/OR

Dual: Replace: $\bullet$ 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: $\bullet$ 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	$B \bullet (B+C) = B$	Covering

Method 2: Prove true using other axioms and theorems.

$B \bullet (B+C)$	$= B \bullet B + B \bullet C$	T8: Distributivity
	$= \mathbf{B} + B \bullet C$	T3: Idempotency
	$= B \bullet (1 + C)$	T8: Distributivity
	$= B \bullet (\mathbf{1})$	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

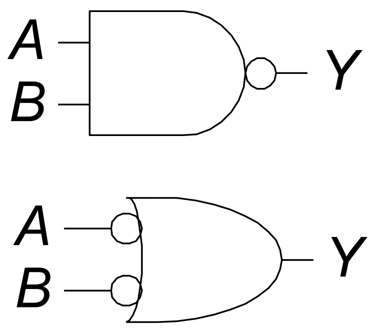
#	Theorem	Dual	Name
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = \overline{B_0} + \overline{B_1} + \overline{B_2} \ldots$	$\overline{B_0 + B_1 + B_2 \ldots} = \overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots$	DeMorgan’s Theorem

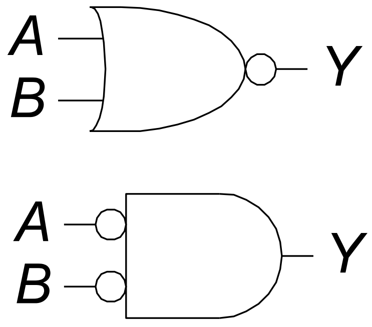
- The complement of the product is the sum of the complements
- **Dual:** The complement of the sum is the product of the complements

Section 2.2 of H&H

De Morgan’s Theorem

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





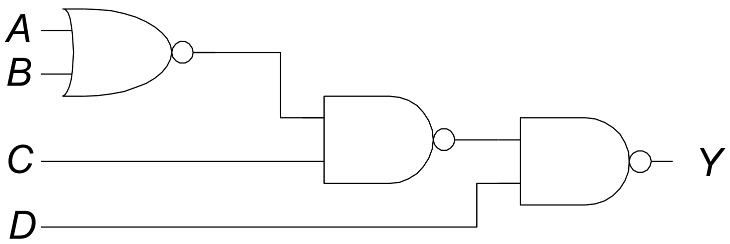
- $Y = \overline{A + B} = \overline{A} \bullet \overline{B}$

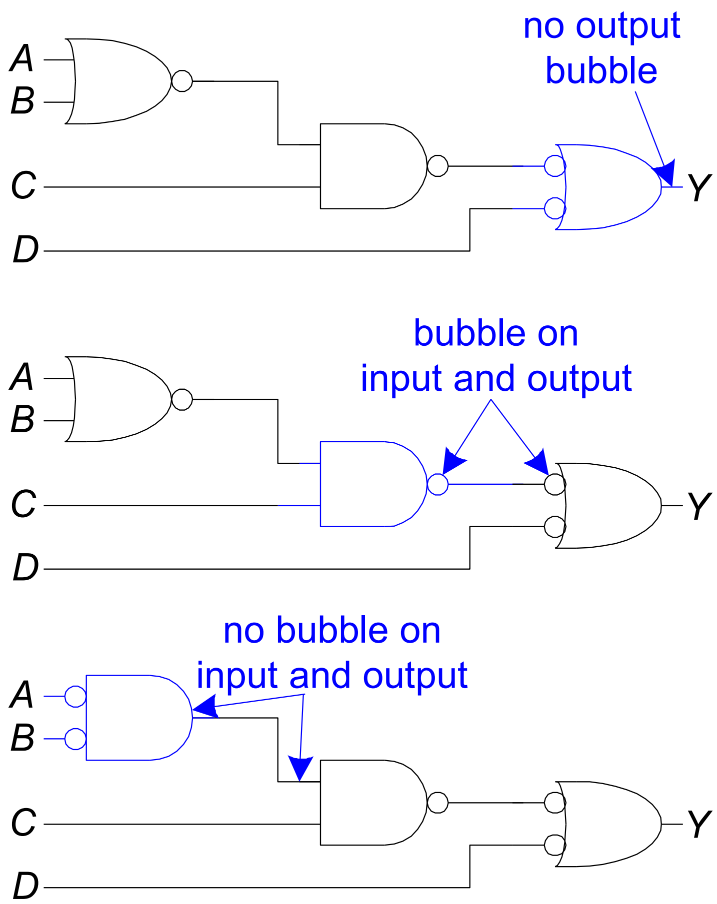
Bubble Pushing Rules

- 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



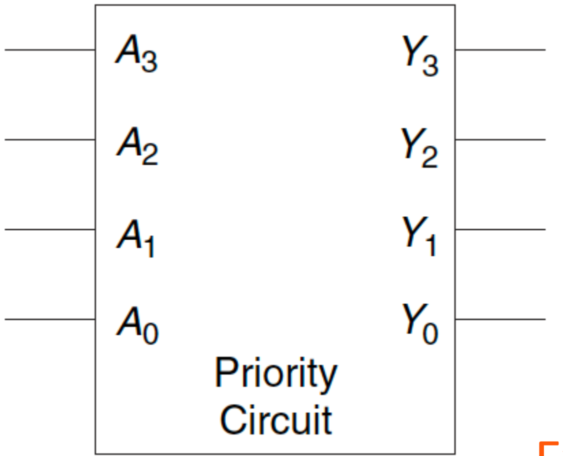


$Y = \overline{A}\,\overline{B}C + \overline{D}$

Priority Circuit

Priority Circuit

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



Requestors | Grant Signals

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

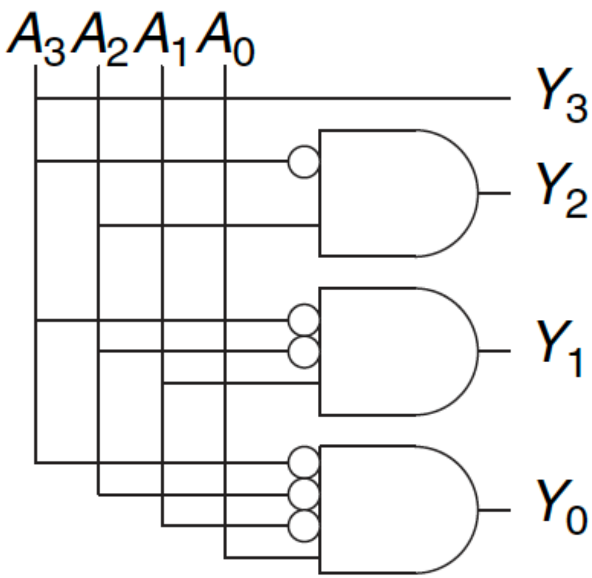
Example 2.7 of H&H

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)



$Y_3 = A_3$

$Y_2 = A_3'A_2$

$Y_1 = A_3'A_2'A_1$

$Y_0 = A_3'A_2'A_1'A_0$

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

Logical Completeness

- Any logic function can be implemented with a PLA
- PLA needs only **AND**, **OR**, and **NOT** gates
- **The set of gates {AND, OR, NOT} is logically complete because we can build a circuit from a truth table without needing any other gate**

Logical Completeness of NAND

- Can we implement a **NOT** gate using a **NAND** gate?
- What about implementing **AND** gate using **NAND** gate **?**
- What about implementing **OR** gate using **NAND** gate**?**

- If all of above is true, then **we can build computers from one gate only, the NAND gate**

- Prove yourself that *NAND is logically complete*

- Most computer today are built using billion of NAND gates

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**)
- DeMorgan of SOP of $\overline{\boldsymbol{F}}$
- 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

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

$$F = \underset{0\quad 0\quad 0}{(A + B + C)}\ \underset{0\quad 0\quad 1}{(A + B + \overline{C})}\ \underset{0\quad 1\quad 0}{(A + \overline{B} + C)}$$

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

Activates this term

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

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

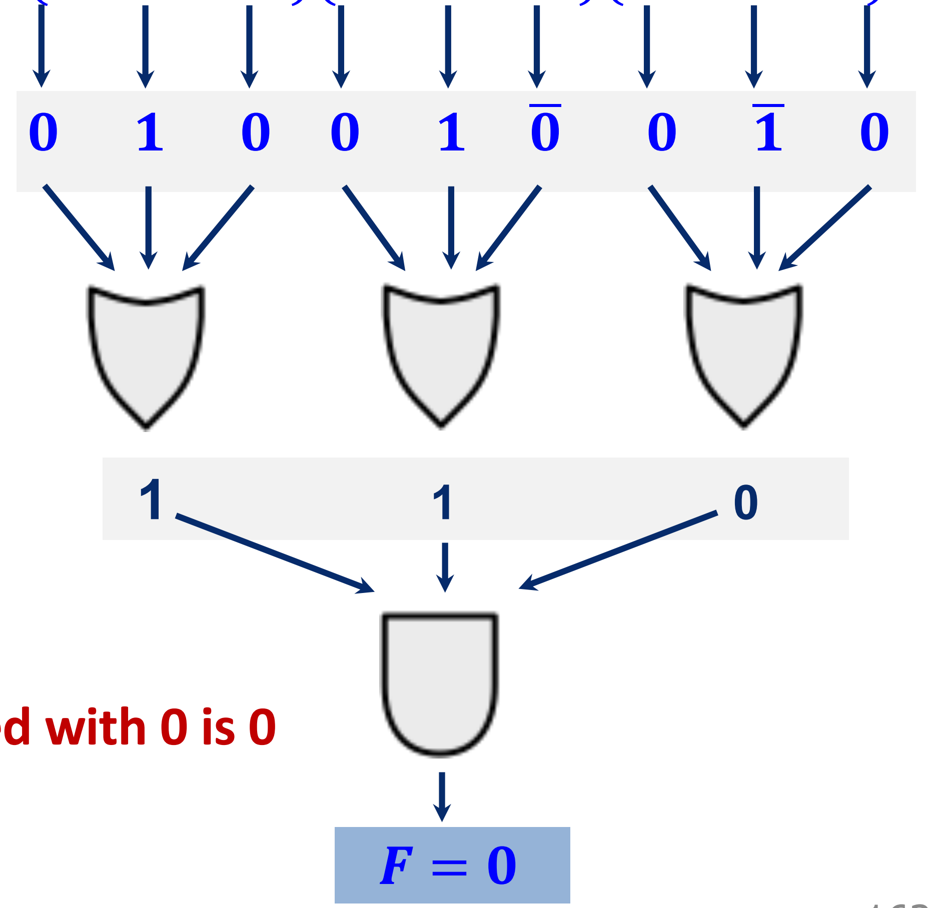
Anything ANDed with 0 is 0; Output F will be 0

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

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

Input

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



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

What We Have Done So Far

- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational logic fundamentals
- Boolean algebra
- Using Boolean algebra to implement combinational circuits
- Basic combinational logic blocks
- Simplifying combinational logic circuits