

Chapter 3 & Appendix B

Arithmetic for Computers

Boolean Algebra

- Developed by George Boole in the 1850s
- Mathematical theory of logic
- Shannon was the first to use Boolean Algebra to solve problems in electronic circuit design. (1938)

Variables & Operations

- All variables have the values 1 or 0
 - we often call these values TRUE or FALSE
- Three operators:
 - OR written as $+$, as in $A + B$
 - AND written as \bullet , as in $A \cdot B$
 - NOT written as an overline, as in \overline{A}

Operators: OR

- The result of the OR operator is 1 if either of the operands is a 1
- The only time the result of an OR is 0 is when both operands are 0s
- OR is similar to *addition*, but operates only on binary values

Operators: AND

- The result of an AND is a 1 only when both operands are 1s
- If either operand is a 0, the result is 0
- AND is similar to *multiplication*, but operates on binary values.

Operators: NOT

- NOT is a *unary* operator - it operates on only one operand
- NOT *negates* its operand
- If the operand is a 1, the result of the NOT is a 0

Equations

Boolean algebra uses equations to express relationships. For example:

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

This equation expressed a relationship between the value of X and the values of A , B and C .

Examples

What is the value of each X :

$$X_1 = 1 \cdot (0 + 1)$$

$$X_2 = A + \bar{A}$$

$$X_3 = A \cdot \bar{A}$$

$$X_4 = X_4 + 1 \quad \leftarrow \text{huh?}$$

Laws of Boolean Algebra

Just like in traditional algebra,
Boolean Algebra has postulates and
identities

We can often use these laws to
reduce expressions or put
expressions in to a more desirable
form

Basic Postulates of Boolean Algebra

- Using just the basic postulates - everything else can be derived:

Commutative laws

Distributive laws

Identity

Inverse

Identity Laws

$$A + 0 = A$$

$$A \cdot 1 = A$$

Inverse Laws

$$A + \overline{A} = 1$$

$$A \cdot \overline{A} = 0$$

Commutative Laws

$$A + B = B + A$$

$$A \cdot B = B \cdot A$$

Distributive Laws

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$

Other Identities

Can be derived from the basic postulates.

Laws of Ones and Zeros

Associative Laws

DeMorgan's Theorems

Zero and One Laws

$$A + 1 = 1 \quad \text{Law of Ones}$$

$$A \cdot 0 = 0 \quad \text{Law of Zeros}$$

Associative Laws

$$A + (B + C) = (A + B) + C$$

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

DeMorgan's Theorems

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

Pop Quiz

- What does $A + \overline{\overline{(A + B)}} + A$ simplify to?

Hint: start with the highlighted part

- A: $A \cdot B$
- B: $A \cdot \bar{B}$
- C: $\bar{A} \cdot B$
- D: $\bar{A} \cdot \bar{B}$

Solution

$$F(a, b) = \overline{\overline{A + (A + B)} + A} \quad \text{DeMorgan's}$$

$$= \overline{A + (\bar{A} \cdot \bar{B}) + A} \quad \text{DeMorgan's}$$

$$= \overline{\bar{A} \cdot (\bar{A} \cdot \bar{B}) + A} \quad \text{DeMorgan's}$$

$$= \overline{\bar{A} \cdot (A + B) + A} \quad \text{Write as } f_{AB} + A, \text{ parentheses important!}$$

$$= \overline{(\bar{A} \cdot (A + B)) + A} \quad \text{DeMorgan's}$$

$$= \overline{(\bar{A} \cdot (A + B))} \cdot \bar{A} \quad \text{Continued on next slide}$$

Solution (continued)

$$F(a, b) = \overline{(\bar{A} \cdot (A + B))} \cdot \bar{A}$$

DeMorgan's

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

DeMorgan's

$$= (A + (\bar{A} \cdot \bar{B})) \cdot \bar{A}$$

Distributive

$$= (A \cdot \bar{A}) + ((\bar{A} \cdot \bar{B}) \cdot \bar{A})$$

Inverse

$$= 0 + ((\bar{A} \cdot \bar{B}) \cdot \bar{A})$$

Continued on next slide

Solution (continued!)

$$F(a, b) = 0 + ((\bar{A} \cdot \bar{B}) \cdot \bar{A}) \quad 0 + A = A$$

$$= (\bar{A} \cdot \bar{B}) \cdot \bar{A} \quad \text{Commutative}$$

$$= (\bar{B} \cdot \bar{A}) \cdot \bar{A} \quad \text{Associative}$$

$$= \bar{B} \cdot (\bar{A} \cdot \bar{A}) \quad A \cdot A = A$$

$$= \bar{B} \cdot \bar{A}$$

Other Operators

- Boolean Algebra is defined over the three operators AND, OR and NOT
 - this is a *functionally complete set*
- There are other useful operators:
 - NOR: is a 0 if either operand is a 1
 - NAND: is a 0 only if both operands are 1
 - XOR: is a 1 if the operands are different
- NOTE: NOR or NAND is (by itself) a functionally complete set!

Boolean Functions

- Boolean functions are functions that operate on a number of Boolean variables
- The result of a Boolean function is itself either a 0 or a 1
- Example: $f(a,b) = a+b$

Alternative Representation

- We can define a Boolean function by showing it using algebraic operations
- We can also define a Boolean function by listing the value of the function for all possible inputs

OR as a Boolean Function $f_{or}(a,b)=a+b$

This is called
a “truth table”

a	b	$f_{or}(a,b)$
0	0	0
0	1	1
1	0	1
1	1	1

Truth Tables

<i>a</i>	<i>b</i>	OR	AND	NOR	NAND	XOR
0	0	0	0	1	1	0
0	1	1	0	0	1	1
1	0	1	0	0	1	1
1	1	1	1	0	0	0

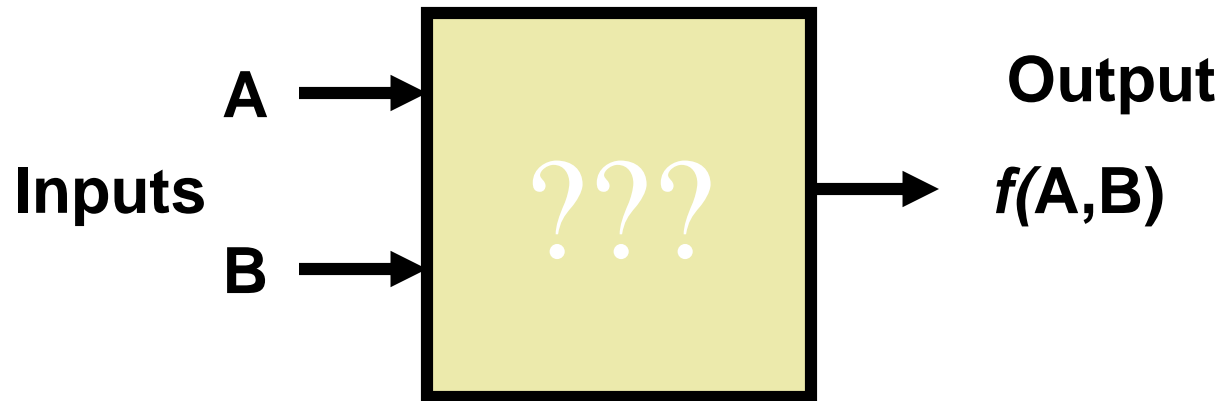
Truth Table for $(X+Y) \cdot Z$

X	Y	Z	$(X+Y) \cdot Z$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Gates

- Digital logic circuits are electronic circuits that are implementations of some Boolean function(s)
- A circuit is built up of *gates*, each *gate* implements some simple logic function

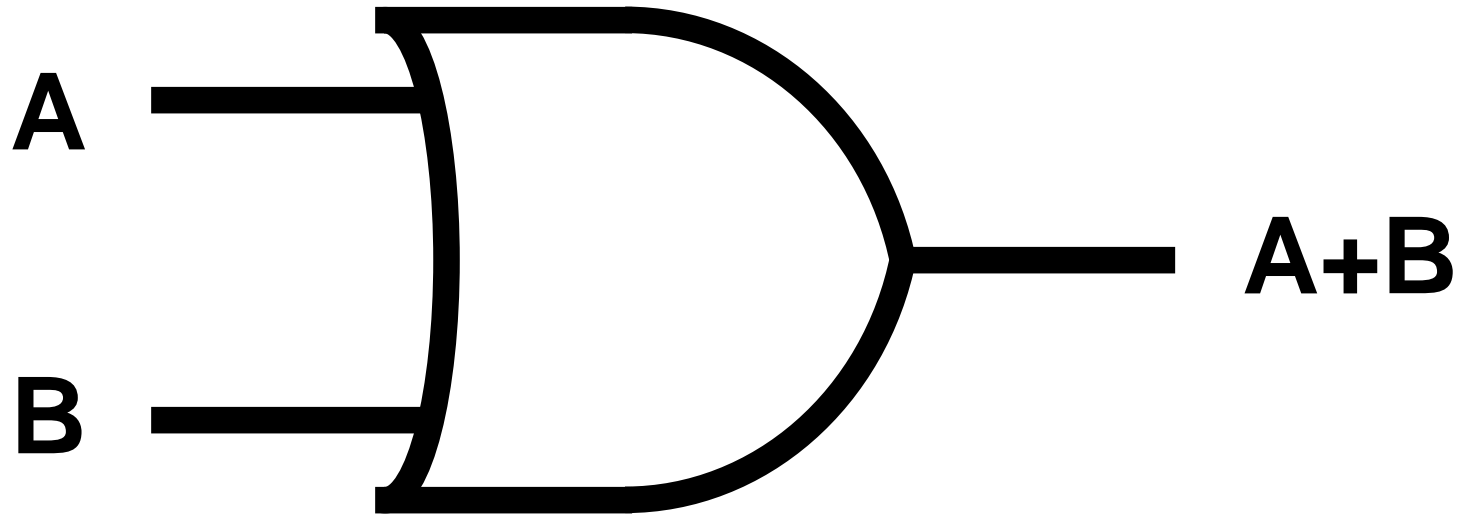
A Gate



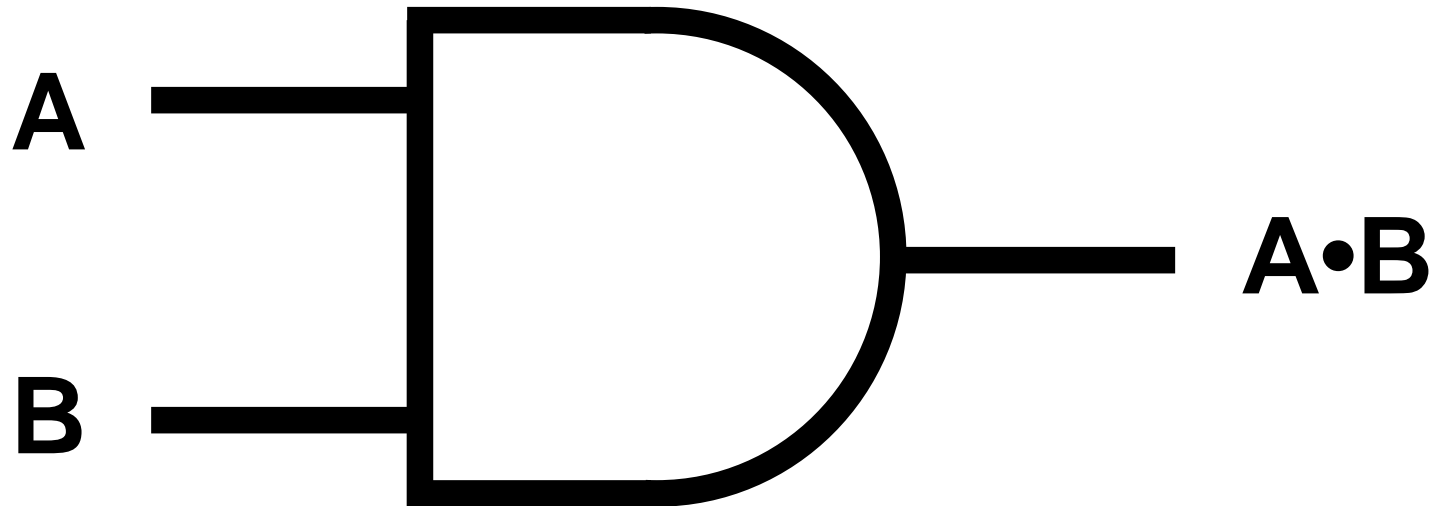
Gates compute something!

- The output depends on the inputs
- If the input changes, the output might change
- If the inputs don't change - the output does not change

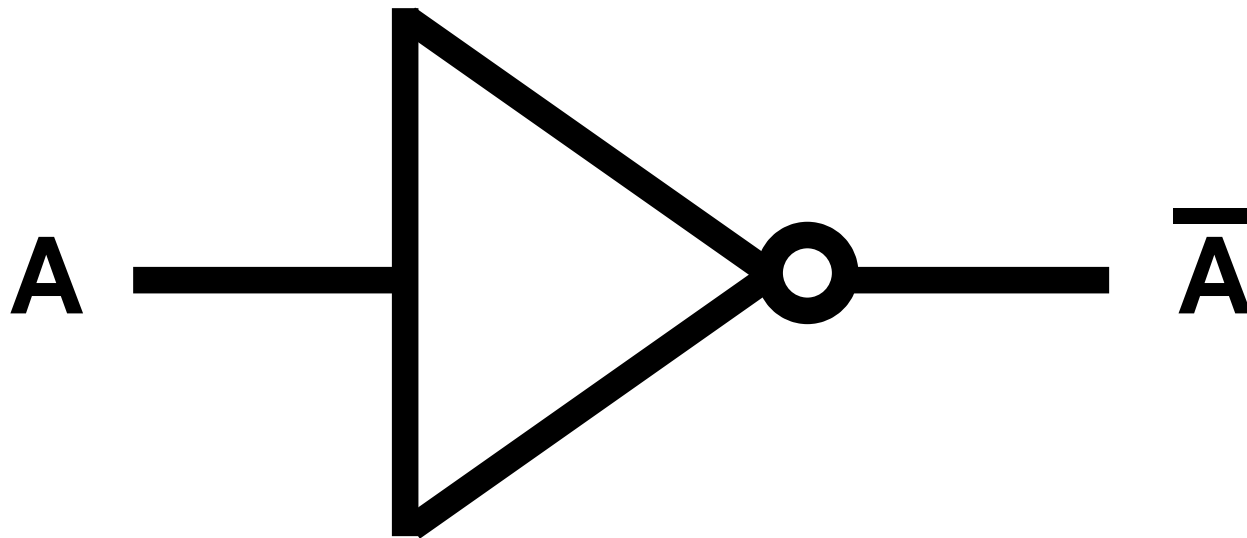
An OR gate



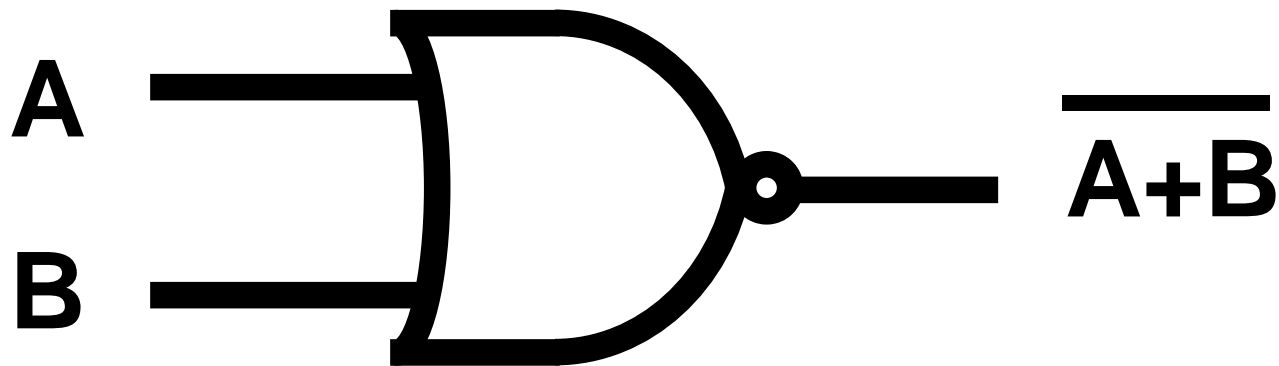
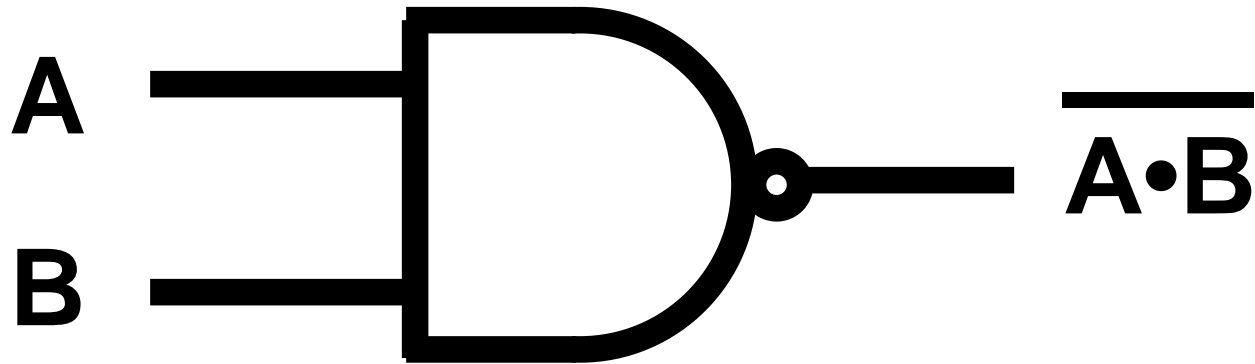
An AND gate



A NOT gate



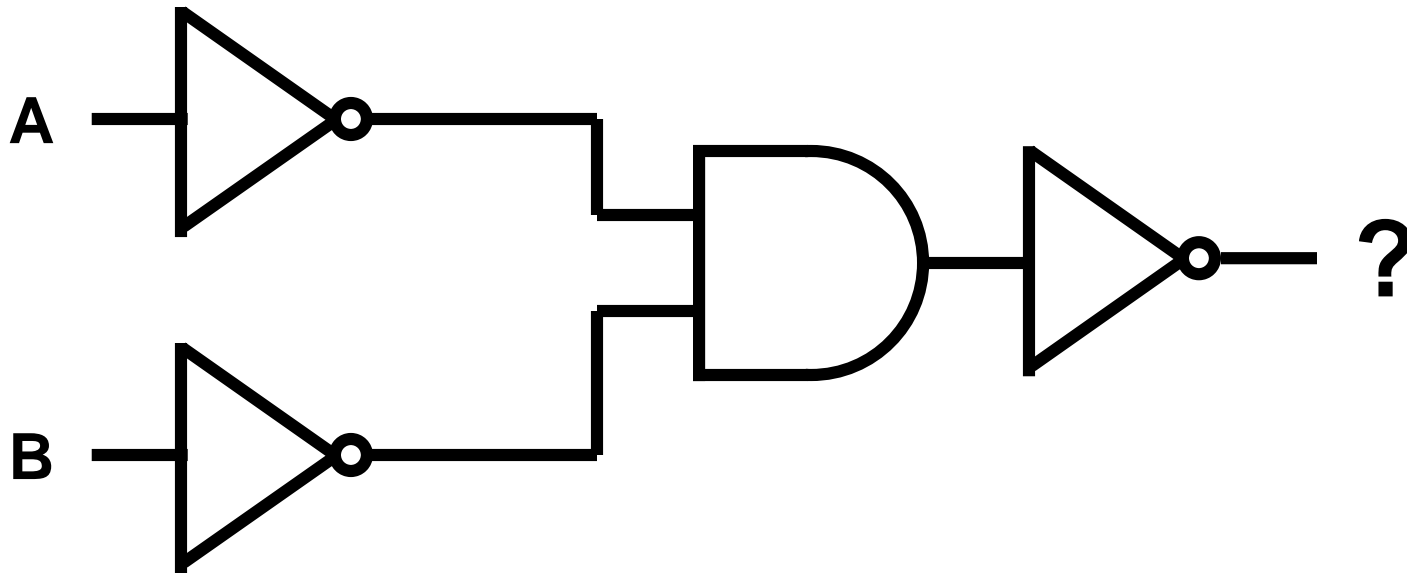
NAND and NOR gates



Combinational Circuits

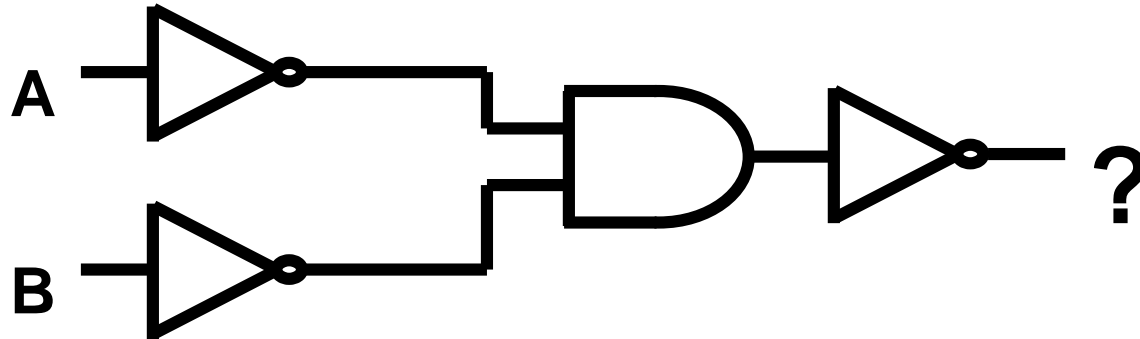
- We can put gates together into circuits
 - output from some gates are inputs to others
- We can design a circuit that represents any Boolean function!

A Simple Circuit



Pop Quiz

- For the circuit below, what is the output?



- A: $\bar{A} + \bar{B}$
- B: $\bar{A} \cdot \bar{B}$
- C: $A + B$
- D: $A \cdot B$

Truth Table for our circuit

a	b	\overline{a}	\overline{b}	$\overline{a} \cdot \overline{b}$	$\overline{\overline{a} \cdot \overline{b}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

Alternative Representations

- Any of these can express a Boolean function:

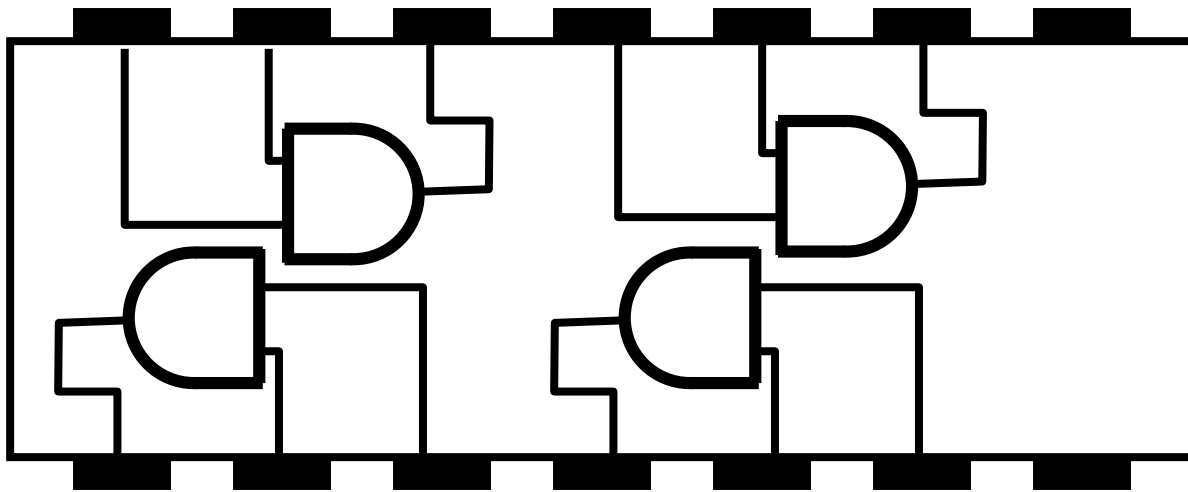
Boolean Equation
Circuit (Logic Diagram)
Truth Table

Implementation

- A logic diagram is used to design an *implementation* of a function
- The implementation is the specific gates and the way they are connected
- We can buy a bunch of gates, put them together (along with a power source) and build a machine

Integrated Circuits

- You can buy an *AND gate chip*:



Function Implementation

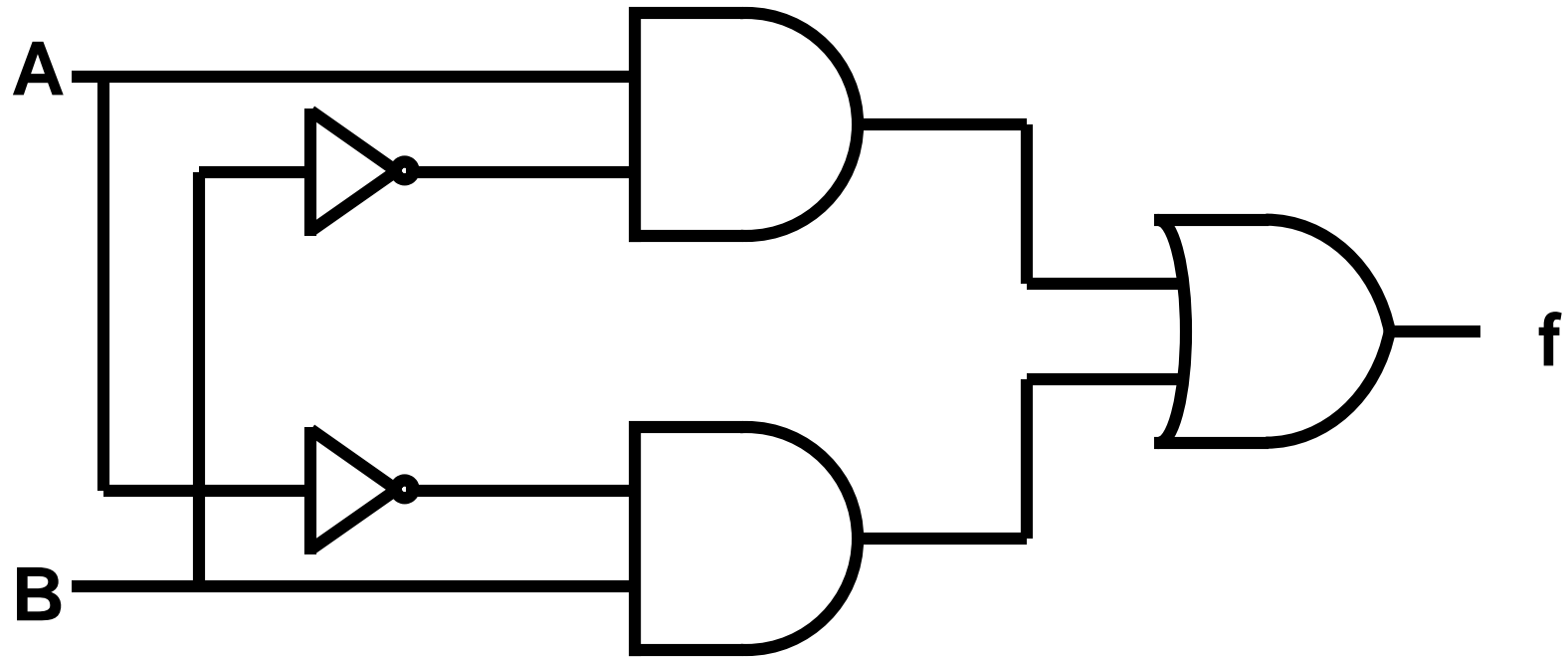
- Given a Boolean function expressed as a truth table or Boolean Equation, there are many possible implementations
- The actual implementation depends on what kind of gates are available
- In general we want to minimize the number of gates (why?)

Example: $f = A \bullet \overline{B} + \overline{A} \bullet B$

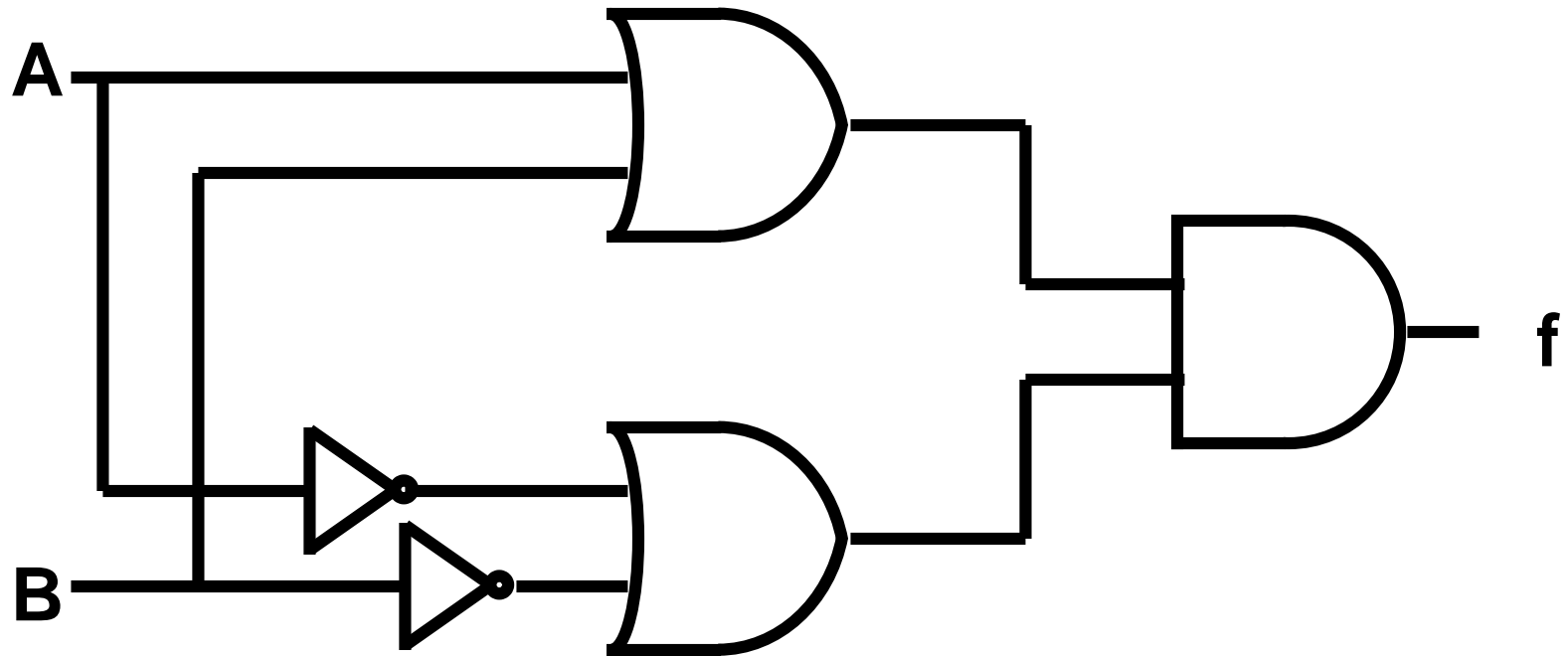
A	B	$A \bullet \overline{B}$	$\overline{A} \bullet B$	f
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

One Implementation

$$f = A \bullet \overline{B} + \overline{A} \bullet B$$



Another Implementation



$$f = A \bullet \bar{B} + \bar{A} \bullet B = (A + B) \bullet (\bar{A} + \bar{B})$$

Prove it's the same function

$$A \bullet \bar{B} + \bar{A} \bullet B =$$

DeMorgan's Law

$$\overline{(A \bullet \bar{B}) \bullet (\bar{A} \bullet B)} =$$

DeMorgan's Laws

$$\overline{(\bar{A} + B) \bullet (A + \bar{B})} =$$

Distributive

$$\overline{((\bar{A} + B) \bullet A) + ((\bar{A} + B) \bullet \bar{B})} =$$

Distributive

$$\overline{(\bar{A} \bullet A + B \bullet A) + (\bar{A} \bullet \bar{B} + B \bullet \bar{B})} =$$

Inverse, Identity

$$\overline{(B \bullet A) + (\bar{A} \bullet \bar{B})} =$$

DeMorgan's Law

$$\overline{(B \bullet A) \bullet (\bar{A} \bullet \bar{B})} =$$

DeMorgan's Laws

$$(\bar{B} + \bar{A}) \bullet (A + B)$$

Logic Design

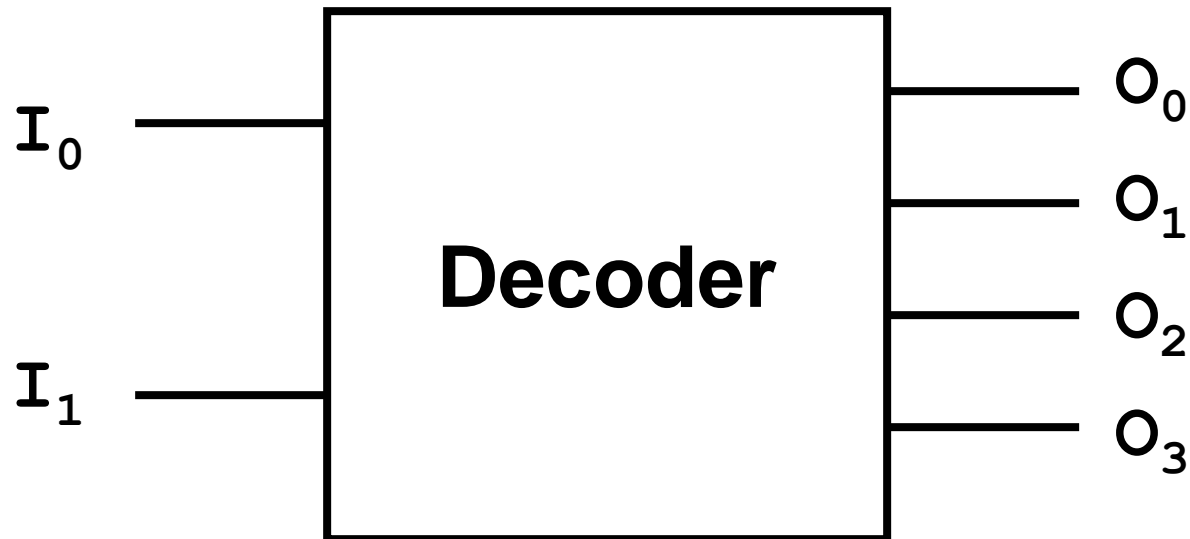
Common Components

- There are many commonly used components in processor design.
- We will use these components when we design control systems (later).
- We will look at the functionality and design of some of these components now.

Some commonly used components

- Decoders: n inputs, 2^n outputs.
 - *the inputs are used to select which output is turned on.*
- Multiplexors: 2^n inputs, n selection bits, 1 output.
 - *the selection bits determine which input will become the output.*

2 input Decoder



Decoder Truth Table

I_1	I_0	O_0	O_1	O_2	O_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Decoder Boolean Expressions

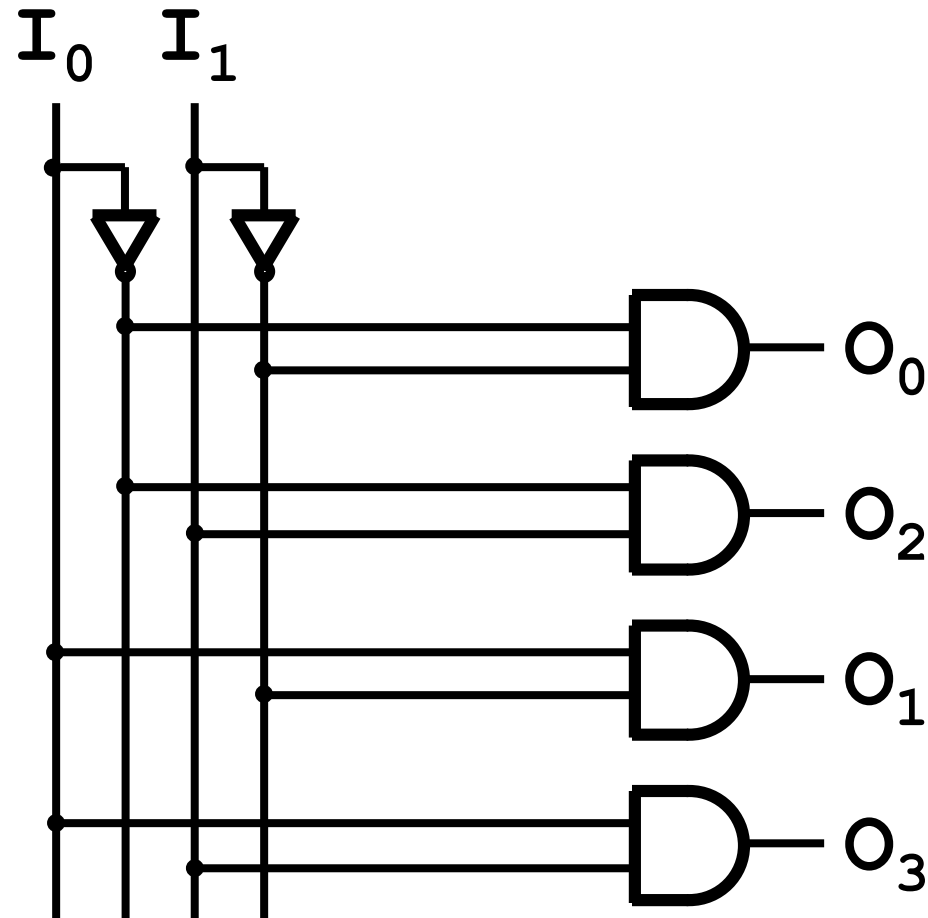
$$O_0 = \overline{I_0} \bullet \overline{I_1}$$

$$O_1 = I_0 \bullet \overline{I_1}$$

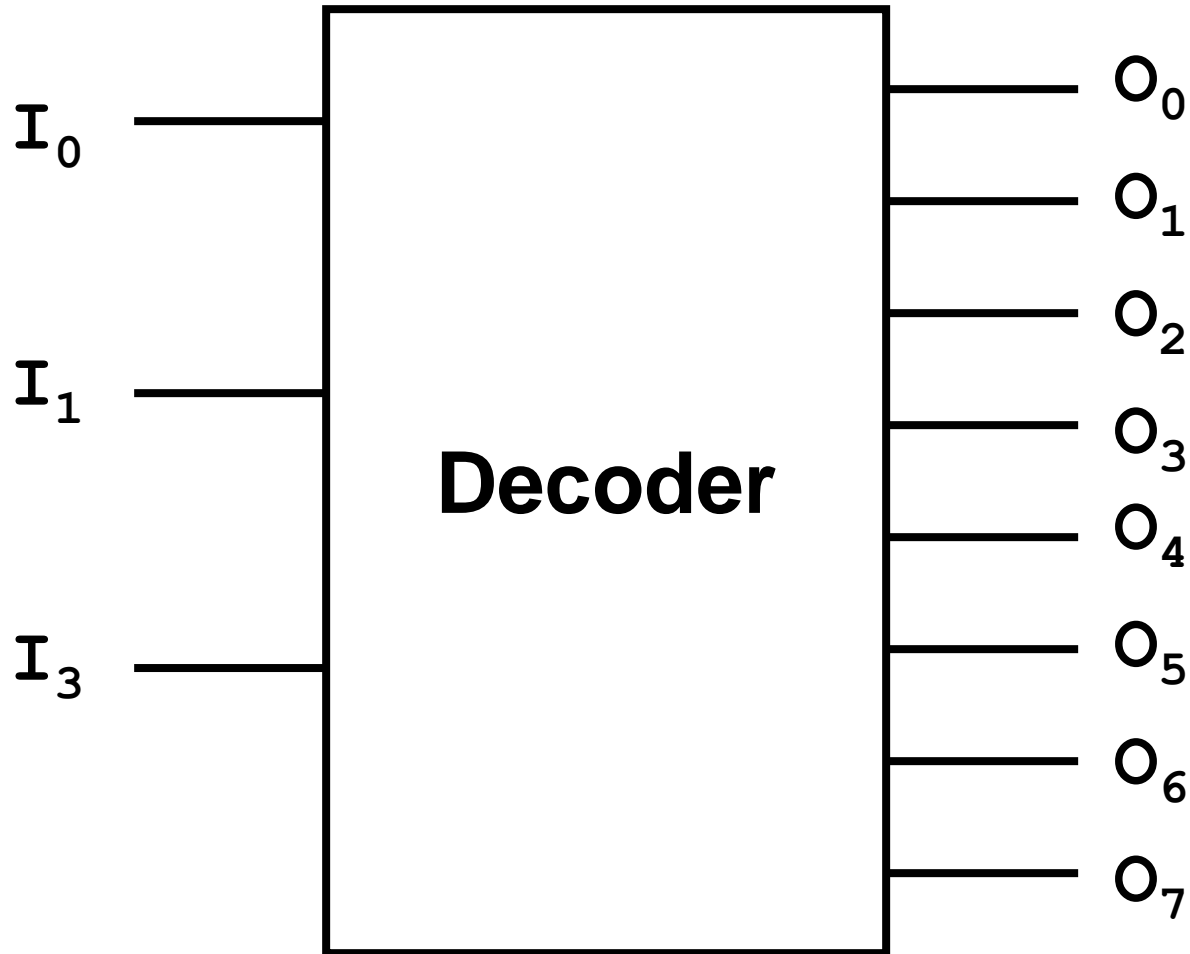
$$O_2 = \overline{I_0} \bullet I_1$$

$$O_3 = I_0 \bullet I_1$$

Decoder Implementation



3 Input Decoder



3 Input Decoder Truth Table

I_2	I_1	I_0	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

3-Decoder Boolean Expressions

$$O_0 = \overline{I_0} \bullet \overline{I_1} \bullet \overline{I_2}$$

$$O_1 = \overline{I_2} \bullet \overline{I_1} \bullet I_0$$

$$O_2 = \overline{I_2} \bullet I_1 \bullet \overline{I_0}$$

$$O_3 = \overline{I_2} \bullet I_1 \bullet I_0$$

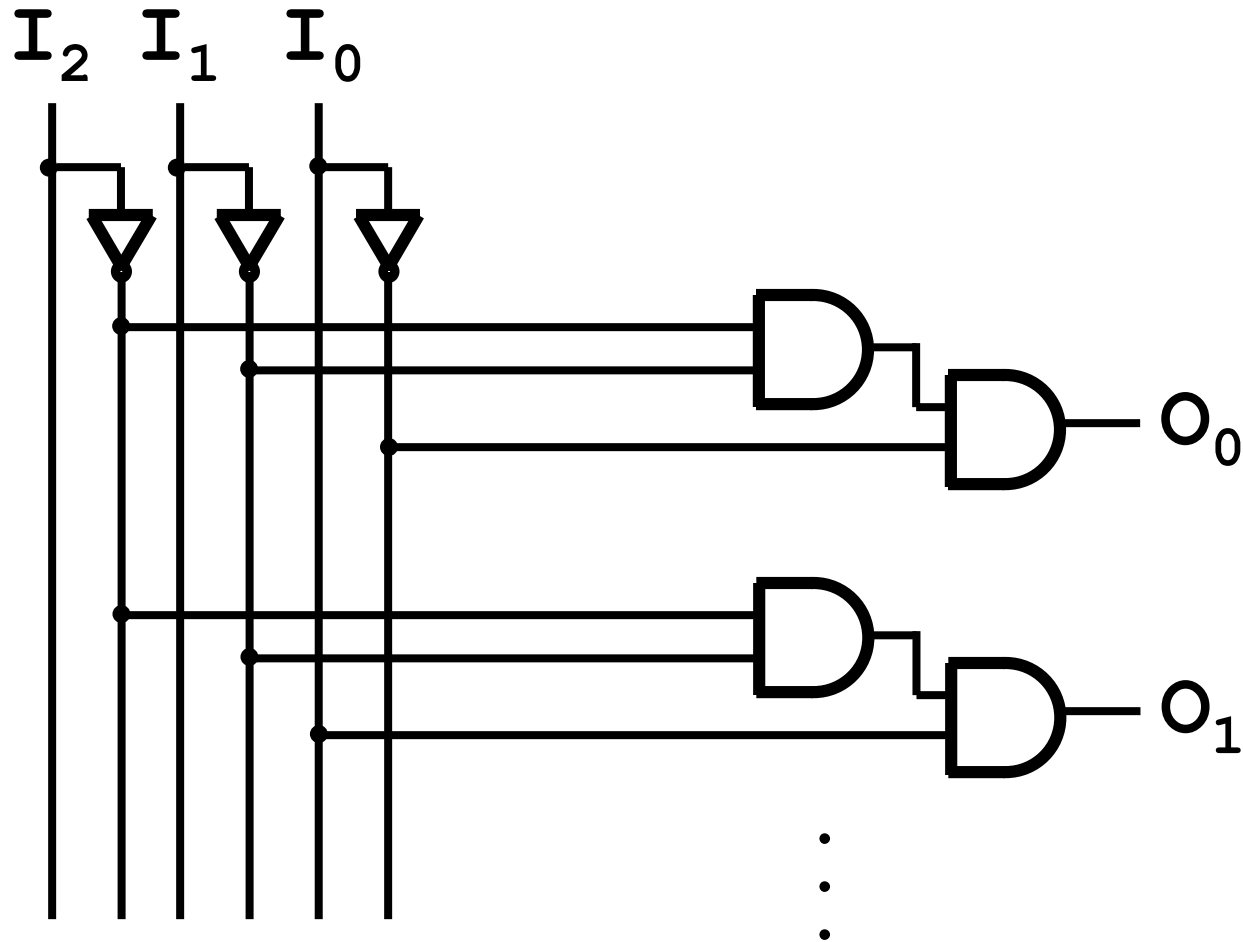
$$O_4 = I_2 \bullet \overline{I_1} \bullet \overline{I_0}$$

$$O_5 = I_2 \bullet \overline{I_1} \bullet I_0$$

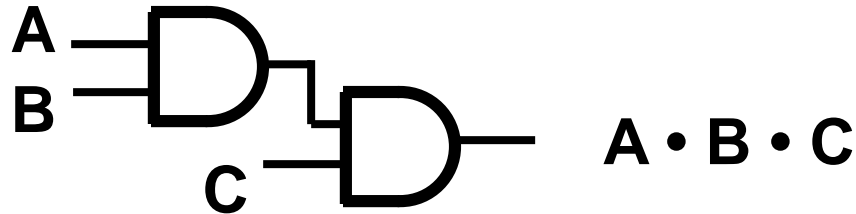
$$O_6 = I_2 \bullet I_1 \bullet \overline{I_0}$$

$$O_7 = I_2 \bullet I_1 \bullet I_0$$

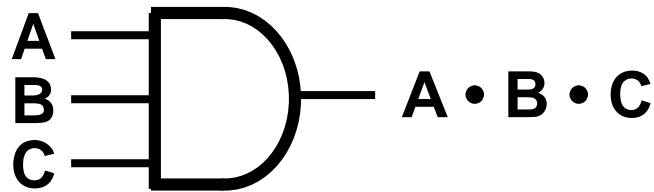
3-Decoder Partial Implementation



A Useful Simplification

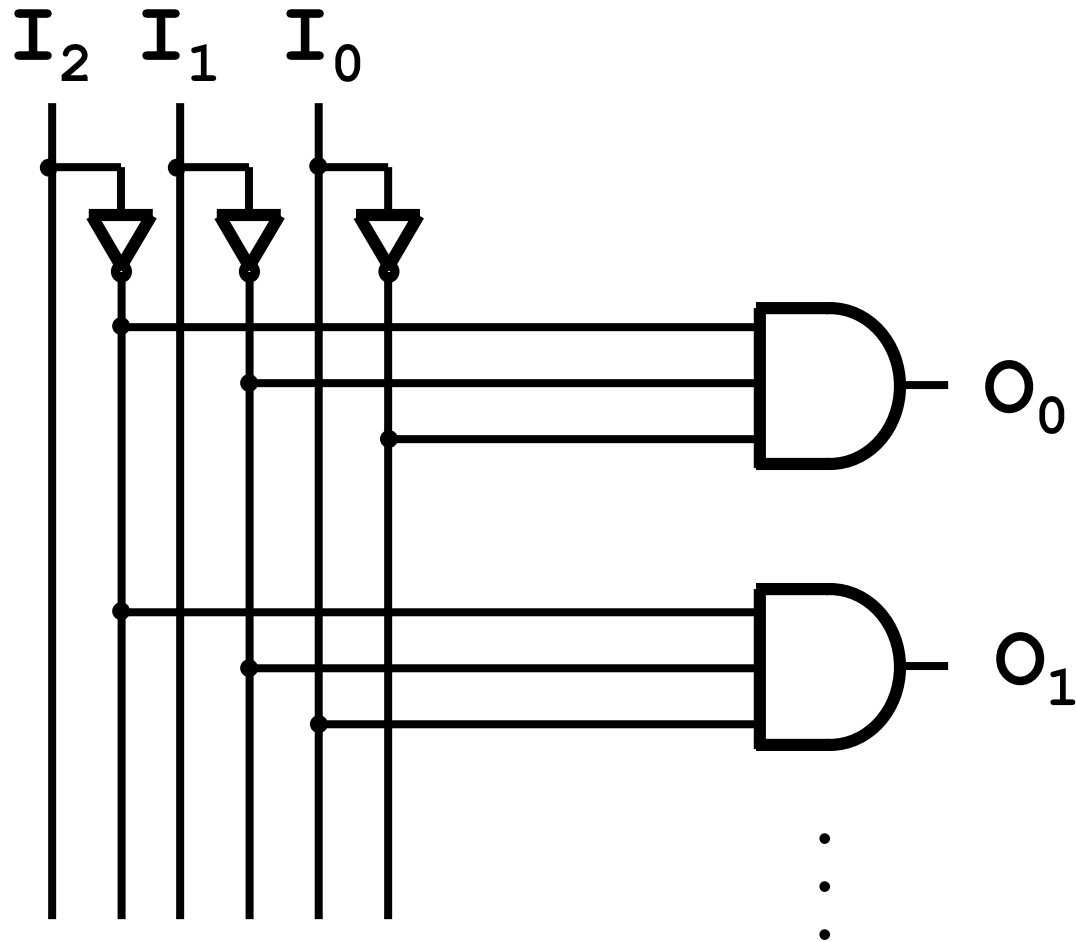


The above logic diagram is often abbreviated as shown below:

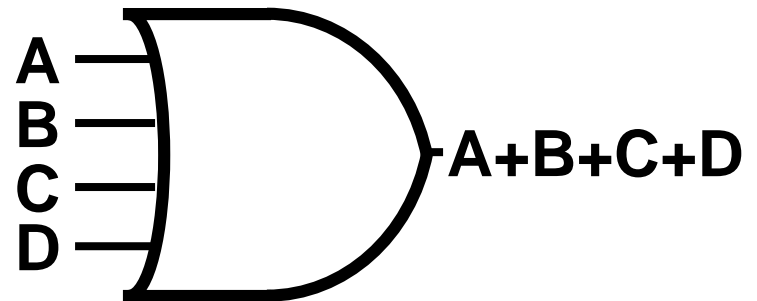
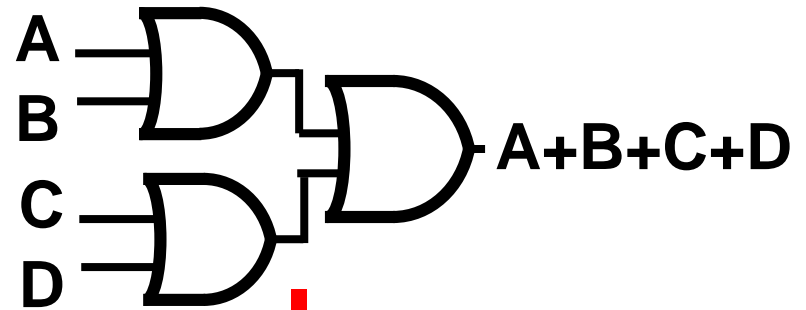
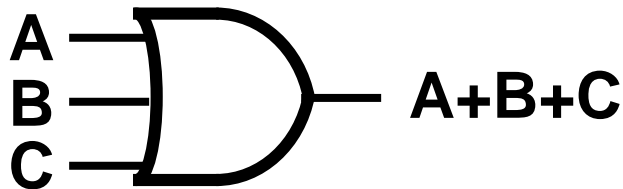
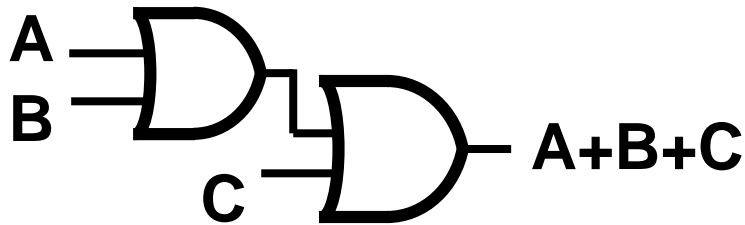


We can do this (without possible confusion) because of the associative property.

Revised Partial 3-Decoder



Multiple Input OR Gates



Pop Quiz

- For the function $f = I_1 \cdot I_2 \cdot \dots \cdot I_{n-1} \cdot I_n$, $n \geq 2$ what is the **minimum** number of layers of 2-input AND gates required?
- A: n
- B: 2^n
- C: $\lceil \log_2 n \rceil$
- D: $\lfloor \log_2 n \rfloor$

2 Input Multiplexor

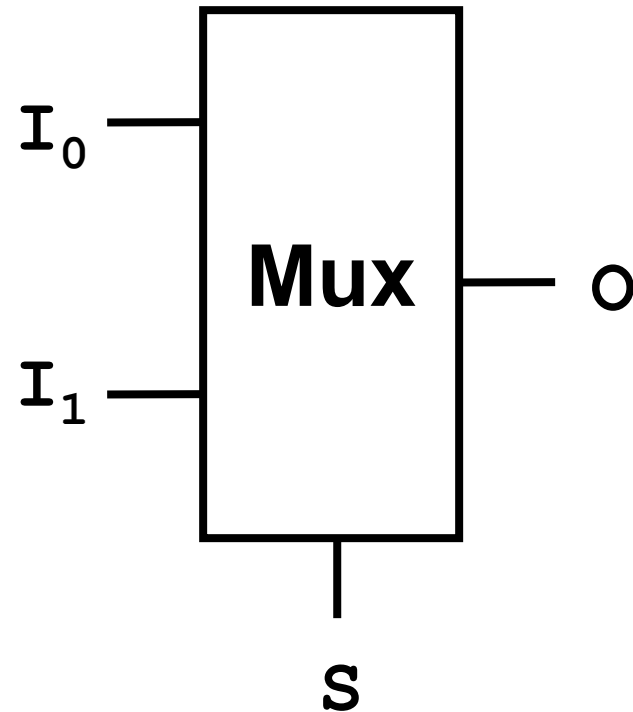
Inputs: I_0 and I_1

Selector: s

Output: o

If s is a 0: $O=I_0$

If s is a 1: $O=I_1$



2-Mux Boolean Function

- The output depends on I_0 and I_1
- The output also depends on s !!!
- We must treat s as an input

$$O = f(I_0, I_1, s)$$

2-Mux Truth Table


Abbreviated
Truth Table

S	O
0	I_0
1	I_1

S	I_0	I_1	O_0
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

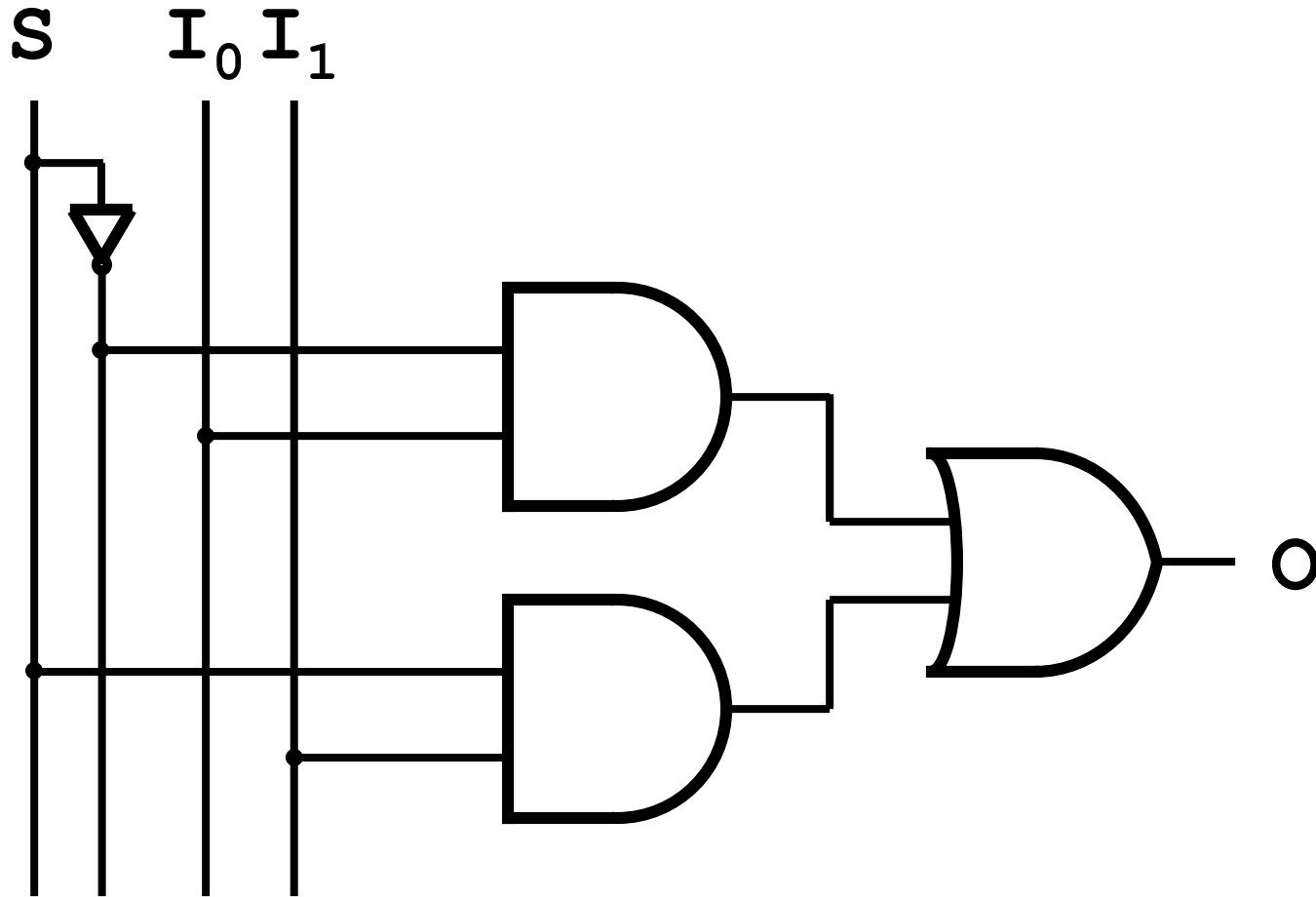
2-Mux Boolean Expression

$$O = \left(I_0 \bullet \bar{S} \right) + \left(I_1 \bullet S \right)$$


terms

Since S can't be both a 1 and a 0, only one of the *terms* can be a 1.

2-Mux Logic Design



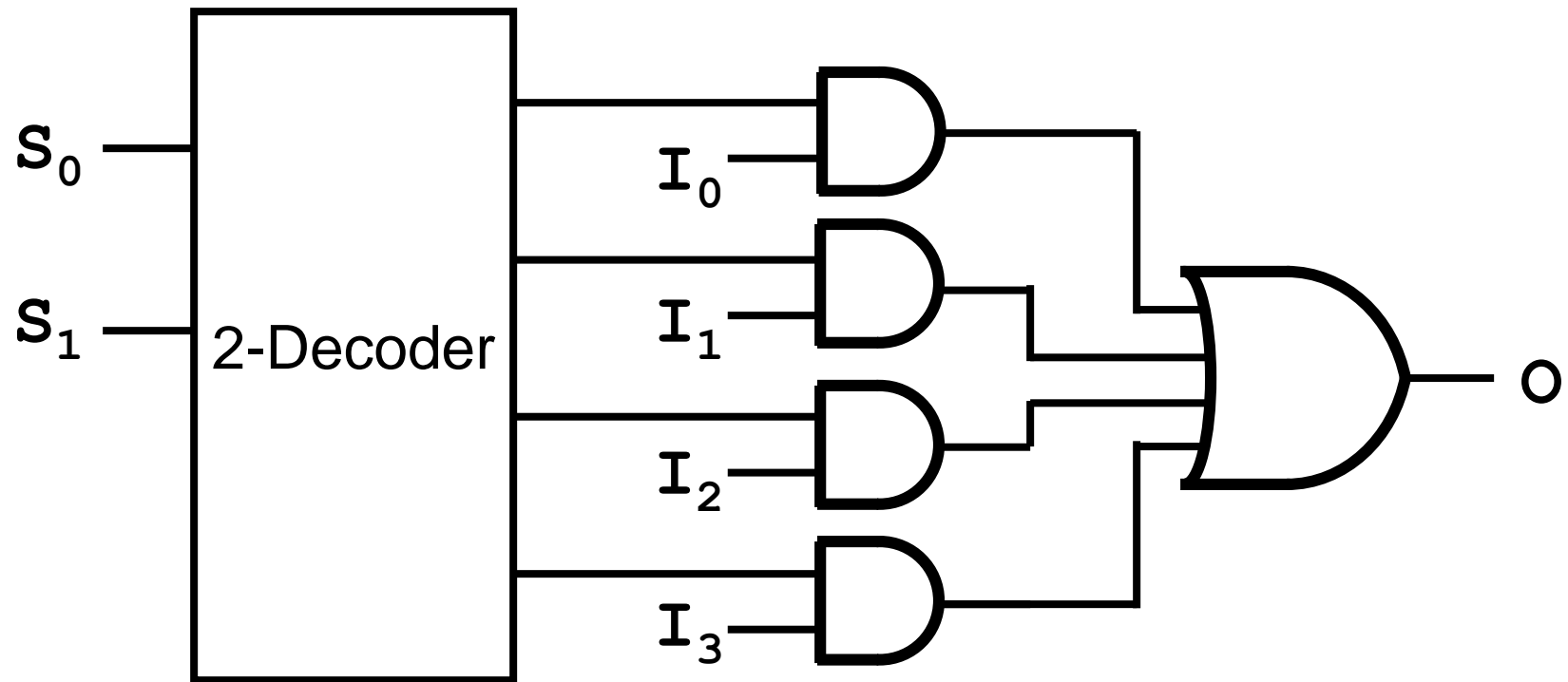
4 Input Multiplexor

- If we have 4 inputs, we need to have 2 selection bits: S_0 S_1

Abbreviated
Truth Table

S_0	S_1	O
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

One Possible 4-Mux



Common Implementations

- There are two general forms that are used in many circuit implementations:
 - Product of Sums
 - A bunch of ORs leading to a big AND gate
 - Sum of Products
 - A bunch of ANDs leading to a big OR gate

Sum of Products

- Express the function by listing all the combinations of inputs for which the output should be a 1
- These combinations are rows in the truth table where the function has the value 1
- Represent each combination with an AND gate
- OR all the AND gates to generate the output

SOP Example: 2-Mux

Find rows in truth table where the output is 1

If S is 1 in that row, connect S to a 3-input AND gate, otherwise connect \bar{S}

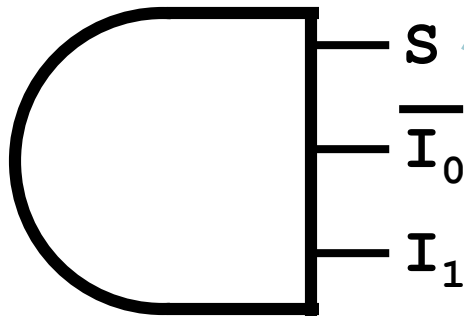
Connect I_0 and I_1 in the same way

The AND gate corresponds to the row in the truth table

S	I_0	I_1	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

SOP Example: 2-Mux (cont).

If the output of this AND gate is a 1, the value of the function is a 1!



S	I_0	I_1	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



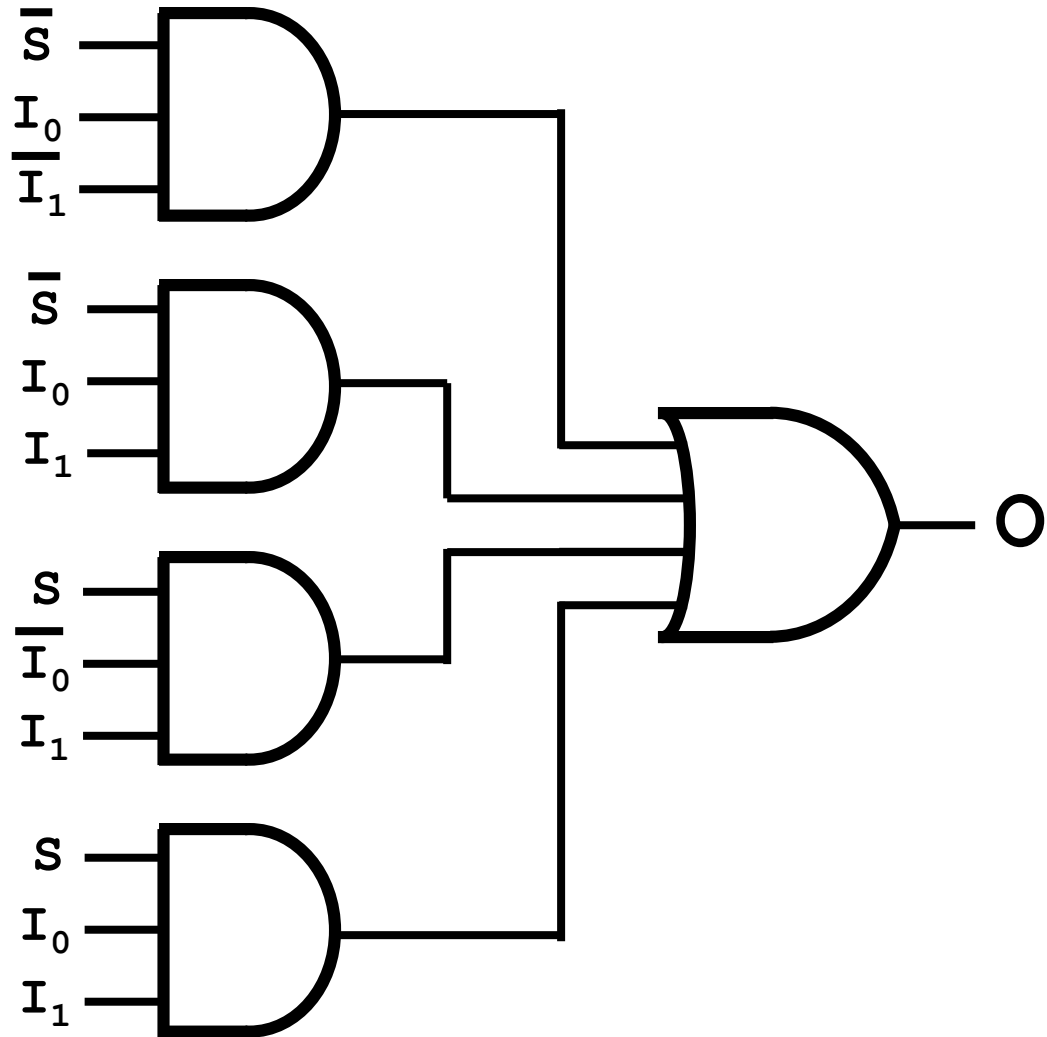
SOP Construction

- For each row on the truth table that has the value 1 (the function has the value 1) build the corresponding AND gate
- Ignore all rows where the function has the value 0
- Connect the output of all the AND gates to one big OR gate

2-Mux Sum Of Products

Truth Table

S	I ₀	I ₁	O ₀
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Pop Quiz

- For the truth table below, the circuit on the previous slide (four 3-AND gates, one 4-OR gate) uses the **minimum** number of gates to make an SOP solution.

- A: True
- B: False

Truth Table			
S	I ₀	I ₁	O ₀
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Product of Sums

- Express the function by listing all the combinations of inputs for which the output should be a 0
- These combinations are rows in the truth table where the function has the value 0
- Represent each combination with an OR gate
- AND all the OR gates to generate the output

POS Example: 2-Mux

Find rows in truth table where the output is 0

If S is 0 in that row, connect S to a 3-input OR gate, otherwise connect \bar{S}

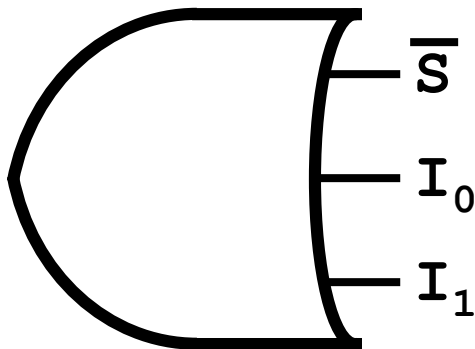
Connect I_0 and I_1 in the same way

The OR gate corresponds to the row in the truth table

S	I_0	I_1	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

POS Example: 2-Mux (cont).

If the output of this OR gate is a 0,
the value of the function is a 0



S	I_0	I_1	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



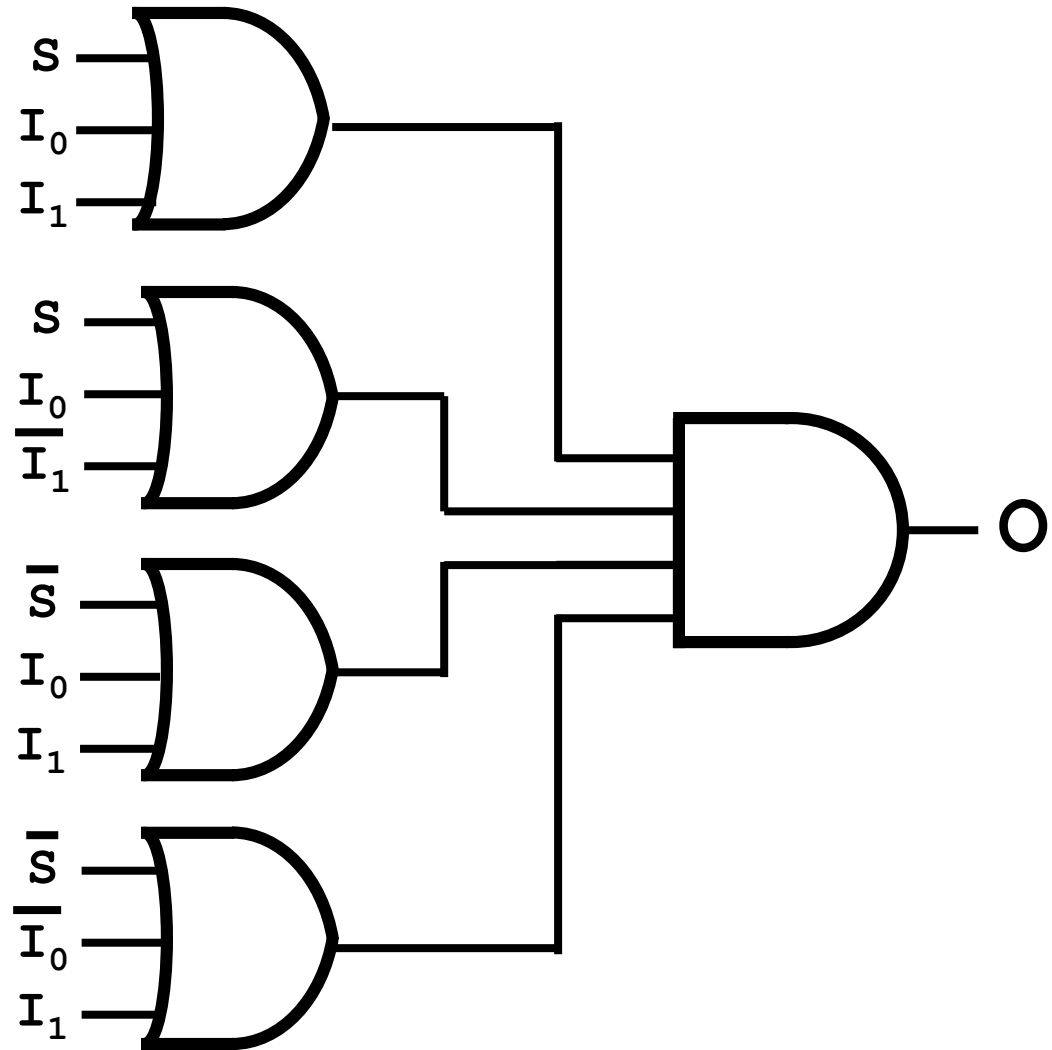
POS Construction

- For each row on the truth table that has the value 0 (the function has the value 0) build the corresponding OR gate
- Ignore all rows where the function has the value 1
- Connect the output of all the OR gates to one big AND gate

4-Mux Product of Sums

Truth Table

S	I ₀	I ₁	O
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Pop Quiz

- Which of these is the POS form of the truth table below?

- A: $(I_0 + \bar{I}_1) \cdot (I_0 + I_1)$
- B: $(\bar{I}_0 + I_1) \cdot (\bar{I}_0 + \bar{I}_1)$
- C: $(I_0 \cdot \bar{I}_1) + (I_0 \cdot I_1)$
- D: $(\bar{I}_0 \cdot I_1) + (\bar{I}_0 \cdot \bar{I}_1)$

Truth Table

I_0	I_1	O_0
0	0	0
0	1	0
1	0	1
1	1	1

Minimization

- SOP and POS forms provide a simple translation from truth table to circuit
- The resulting designs may involve more gates than are necessary
- There are a number of techniques used to minimize such circuits

Minimization Techniques

- Boolean Algebra
 - use postulates and identities to reduce expressions.
- Karnaugh Maps
 - graphical technique useful for small circuits (no more than 4 or 5 inputs)
- Tabular Methods
 - suitable for large functions - usually done by a computer program

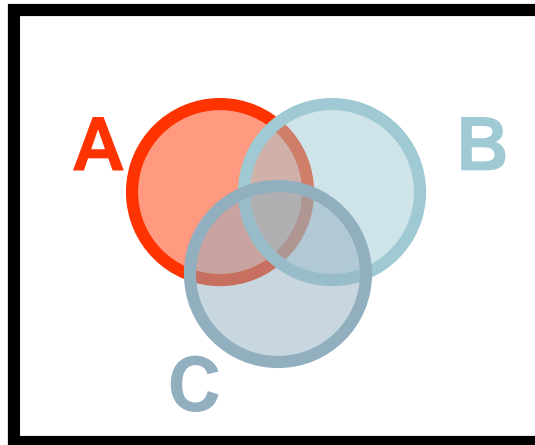
Karnaugh Map (K-map)

- Based on SOP form
- It may be possible to *merge* terms
- Example: $f = (A \bullet B \bullet C) + (\overline{A} \bullet B \bullet C)$
 - Close inspection reveals that it doesn't matter what the value of A is!
 - Here is a simpler version of the same function:

$$f = (B \bullet C)$$

Graphical Representation

- The idea is to draw a picture in which it will be easy to see when terms can be merged
- We draw the truth table in 2-D, the result is similar to a Venn Diagram

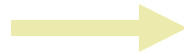


K-Map Example

$$f = A \bullet B + \bar{A} \bullet B$$

Truth Table

<i>A</i>	<i>B</i>	<i>f</i>
0	0	0
0	1	1
1	0	0
1	1	1



K-Map

	<i>B</i> =0	<i>B</i> =1
<i>A</i> =0	0	1
<i>A</i> =1	0	1

In the K-Map it's easy to see that the value of *A* doesn't matter

Ex 2: The Majority Function

- The majority function is 1 whenever the majority of the inputs are 1
- Here is an SOP Boolean equation for the 3-input majority function:

$$f = A \bullet B \bullet C + \bar{A} \bullet B \bullet C + A \bullet \bar{B} \bullet C + A \bullet B \bullet \bar{C}$$

K-Map for Majority Function

Truth Table

<i>A</i>	<i>B</i>	<i>C</i>	<i>f</i>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

K-Map

		<i>AB</i>			
		00	01	11	10
<i>C</i>	0	0	0	1	0
	1	0	1	1	1

K-Map Construction

AB		00	01	11	10
C	0	0	0	1	0
	1	0	1	1	1

- Notice that any two adjacent cells differ by exactly one bit in the input
 - either A is different, or B is different or C is different
 - Never more than one variable is different

How to use K-Map

K-Map

		AB			
		00	01	11	10
C	0	0	0	1	0
	1	0	1	1	1

Rectangular collections of cells that all have the value 1 indicate it is possible to *merge* the corresponding terms in SOP expression.

The number of cells in the rectangle must be a power of 2

Possible Mergings

- There are three possible *merges* of terms in this K-Map

K-Map

AB		00	01	11	10
C	0	0	0	1	0
	1	0	1	1	1

One of the merges

- The merge shown means "if C is 1 and B is 1, it doesn't matter what the value of A is"

K-Map

AB /		00	01	11	10
		0	0	1	0
C	0	0	0	1	0
	1	0	1	1	1

$$\overline{A} \bullet B \bullet C + A \bullet B \bullet C = B \bullet C$$

All 3 reductions

K-Map

		<i>AB</i>			
		00	01	11	10
<i>C</i>	0	0	0	1	0
	1	0	1	1	1

Original: $f = A \bullet B \bullet C + \bar{A} \bullet B \bullet C + A \bullet \bar{B} \bullet C + A \bullet B \bullet \bar{C}$

Reduced: $f = B \bullet C + A \bullet C + A \bullet B$

Another Example

K-Map

		AB			
		00	01	11	10
C	0	0	1	1	0
	1	0	1	1	0

- Here we could make two 2x1 or two 1x2 groups
- Since we have four 1s that can fit in a rectangle, we could simplify further!
- This function is really just $f = B$

Yet Another Example

K-Map

		AB			
		00	01	11	10
C	0	0	0	0	0
	1	1	1	1	1

- Here we could make several different 1x2 groups
- Since we have four 1s that can fit in a rectangle, we could simplify further!
- This function is really just $f = C$

And Yet Another Example

K-Map

		AB			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

- Looks like we can only make two 2x1 groups
- We can wrap around edges thanks to adjacency coding
- This function is really just $f = \bar{B}$

K-Map Concept

- A professional *Logic Designer* would need to use minimization techniques every day
- There are systematic procedures for minimizing SOP and POS form Boolean equations

Constructing an ALU

Arithmetic Logic Unit

- The device that performs the arithmetic operations and logic operations
 - arithmetic ops: addition, subtraction
 - logic operations: AND, OR
- For MIPS we need a 32-bit ALU
 - so we can add 32-bit numbers, etc.

Starting Small

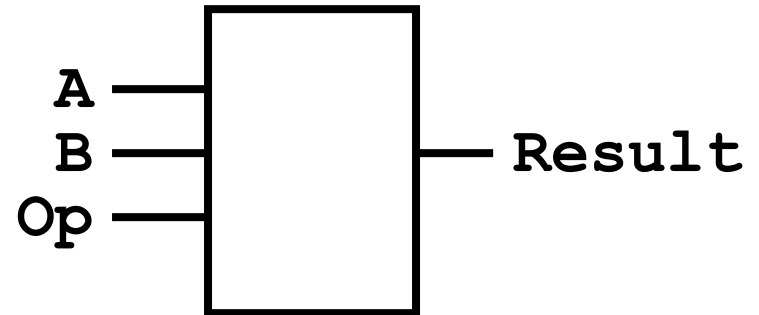
- We can start by designing a 1-bit ALU
- Put a bunch of them together to make larger ALUs
 - building a larger unit from a 1-bit unit is simple for some operations, can be tricky for others
- Bottom-Up approach:
 - build small units of functionality and put them together to build larger units

1-bit AND/OR machine

- We want to design a single box that can compute either AND or OR
- We will use a *control input* to determine which operation is performed
 - Name the control “ op ”
 - if $op == 0$ do an AND
 - if $op == 1$ do an OR

Truth Table For 1-bit AND/OR

Op	A	B	Result
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



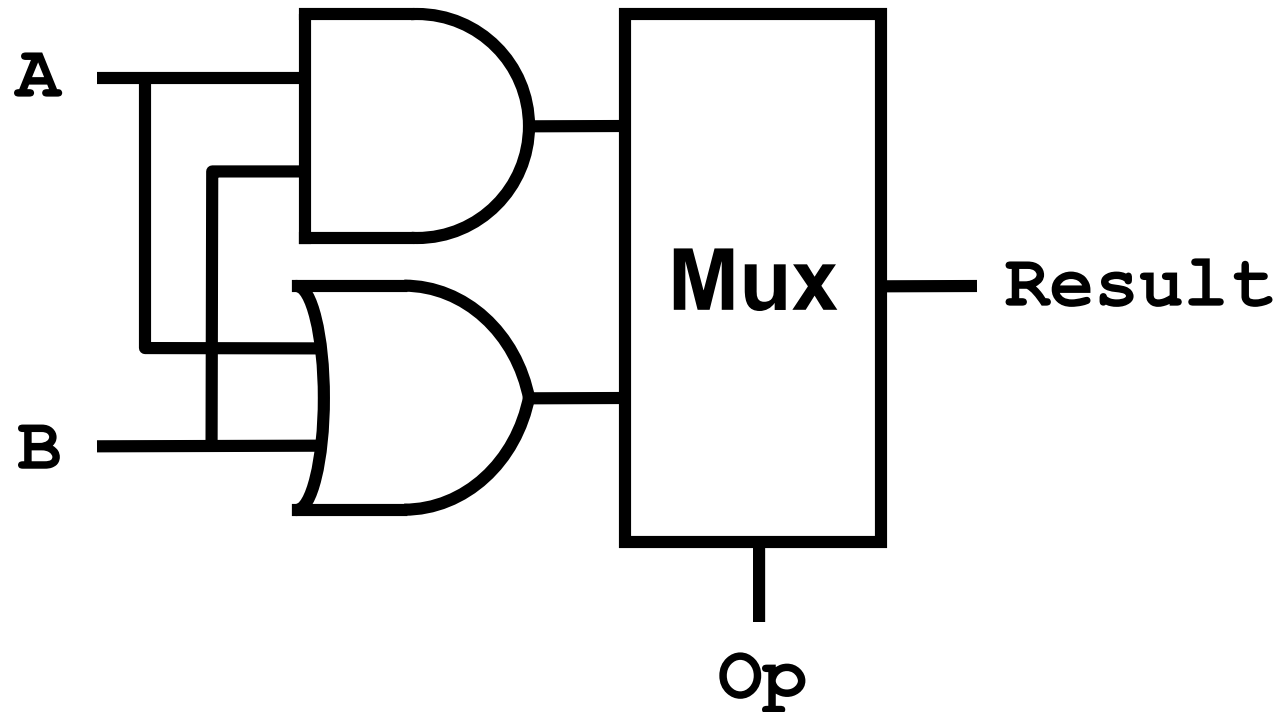
Op=0: Result is $A \cdot B$

Op=1: Result is $A + B$

Logic for 1-Bit AND/OR

- We could derive SOP or POS and build the corresponding logic
- We could also just do this:
 - Feed both A and B to an OR gate
 - Feed A and B to an AND gate
 - Use a 2-input MUX to pick which one will be used
 - Op is the selection input to the MUX

Logic Design for 1-Bit AND/OR




Addition

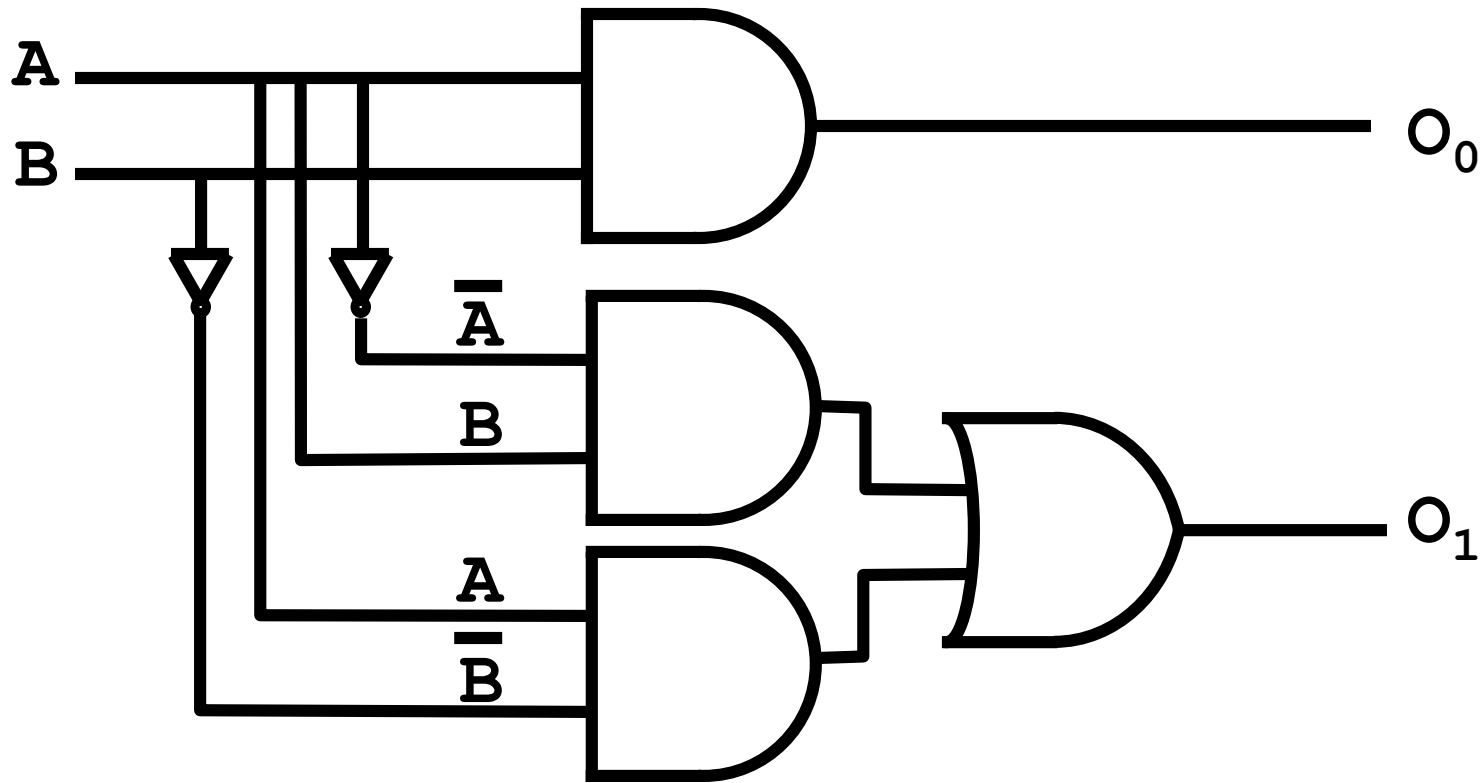
- We need to build a 1 bit adder
 - compute binary addition of 2 bits
- We already know that the result is two bits

A	B	O_0	O_1
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

This is addition,
not logical OR!


$$\begin{array}{r} \text{A} \\ + \text{B} \\ \hline O_0 \quad O_1 \end{array}$$

One Implementation



Binary addition and our adder

$$\begin{array}{r} 1 1 \leftarrow \text{Carry} \\ 01001 \\ + 01101 \\ \hline 10110 \end{array}$$

What we really want is something that can be used to implement the binary addition algorithm

- O_0 is the *carry*
- O_1 is the *sum*

What about the second column?

$$\begin{array}{r} 1 1 \leftarrow \text{Carry} \\ 01001 \\ + 01101 \\ \hline 10110 \end{array}$$

- We are adding three bits
 - new bit is the *carry* from the first column
 - The output is still two bits, i.e., a *sum* and a *carry*

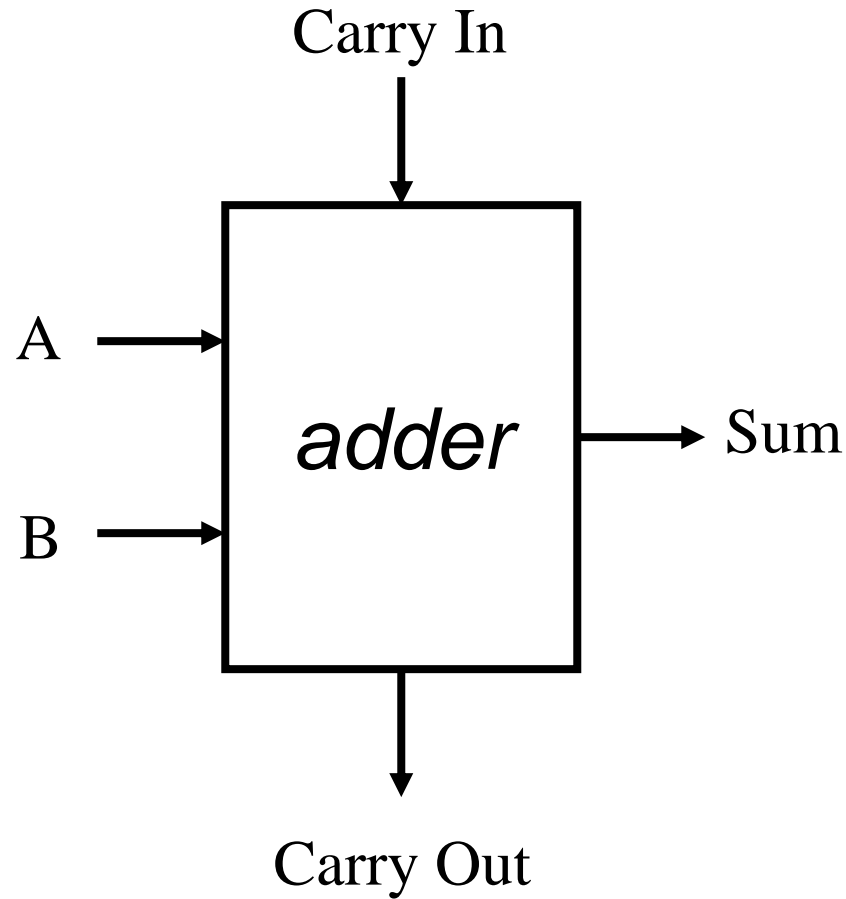
Revised Truth Table for Addition

A	B	Carry In	Carry Out	Sum
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

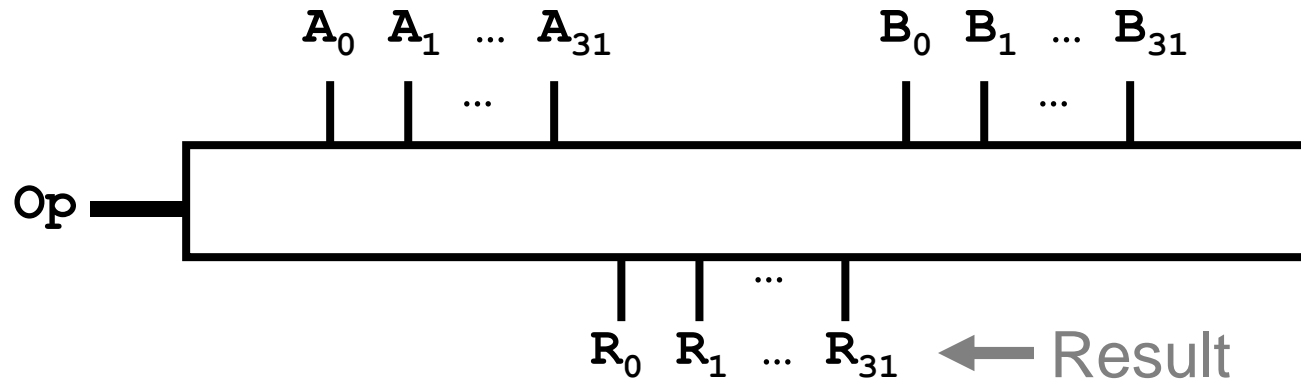
Logic Design for new adder

- We can derive SOP expressions from the truth table
- We can build a combinational circuit that implements the SOP expressions
- We can put it in a box and give it a name

New Component: Adder

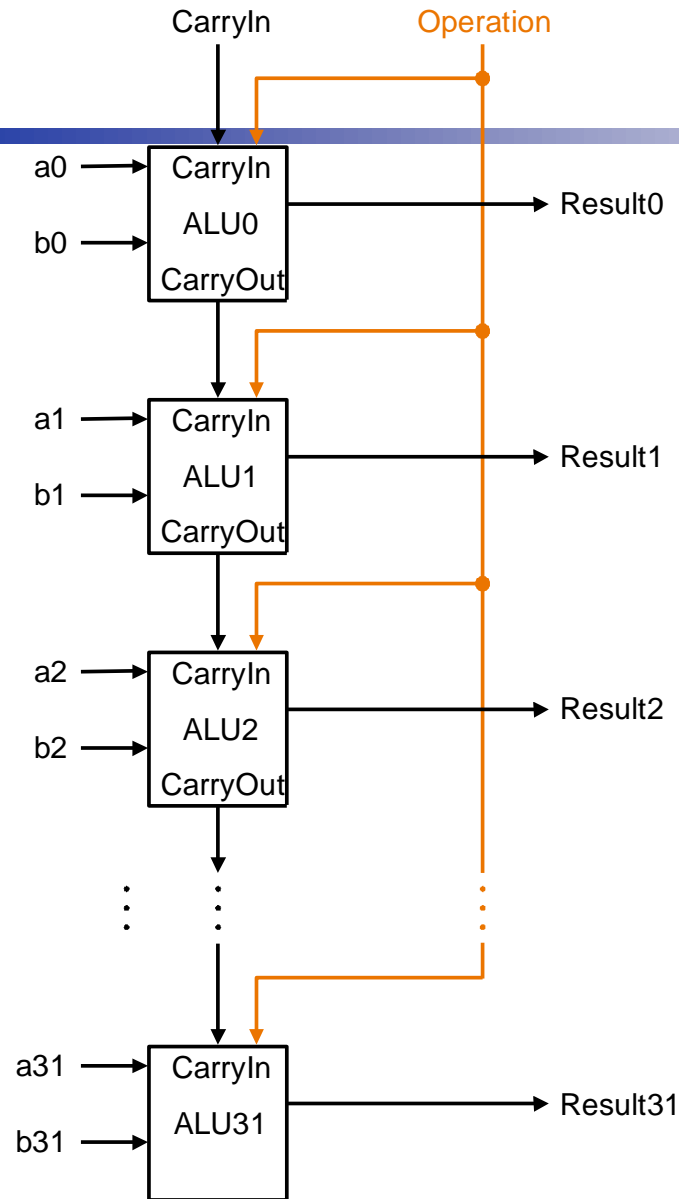


Building a 32-bit ALU



- 64 inputs
- 3 different Operations (AND, OR, add)
- 32 bit output

Ripple Carry Adder



- Carry out from ALU_0 is sent to carry in of ALU_1
- How long will it take for the result to become available?
 - the CarryOuts must propagate through all 32 1-Bit ALUs

Pop Quiz

- Claim: To support subtraction, we will need to make a separate “subtractor” block instead of reusing the adder
- A: True
- B: False
- C: Only true if $A > 0$, $B > 0$
- D: Only true if $A < 0$, $B < 0$
- E: Only true if $A > B$

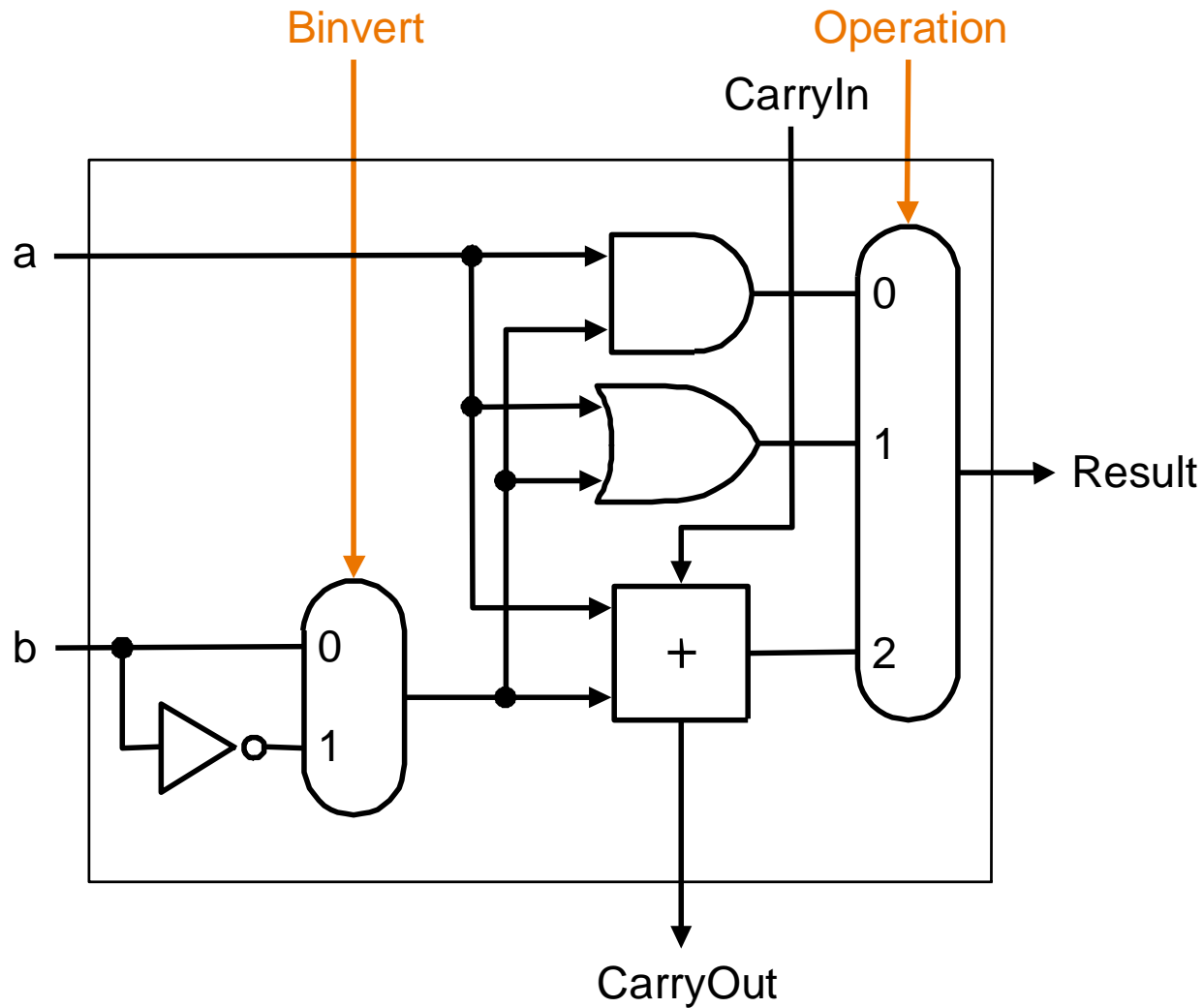
New Operation: Subtraction

- Subtraction can be done with an adder:
 $A - B$ can be computed as $A + -B$
- To negate B we need to:
 - invert the bits
 - add 1
 - (remember, 2's complement)

Negating B in the ALU

- We can negate B by in the ALU by:
 - providing \overline{B} to the adder
 - need a selection bit to do this
- *To add 1, just set the initial carry in to 1!*

Revised 1-Bit ALU



Uses for our ALU

- addition, subtraction, OR and AND instructions can be implemented with our ALU
 - we still need to get the right values to the ALU and set control lines
- We can also support the `slt` instruction
 - need to add a little more to the 1-bit ALU

Supporting `slt`

`slt` needs to compare two numbers

- comparison requires a subtraction

if $A - B$ is negative, then $A < B$ is true;
otherwise $A < B$ is false

True: output should be 00000000...001

False: output should be 00000000...000

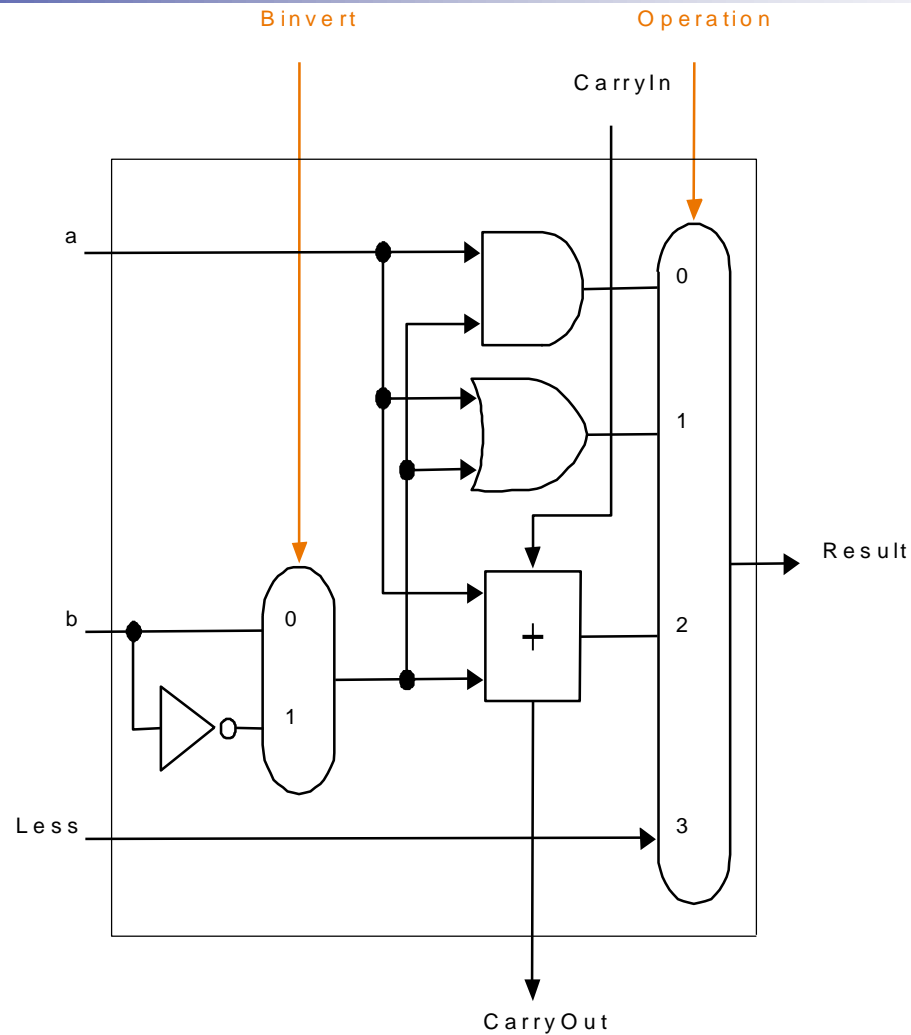
slt Strategy

- To compute `slt A B`:
 - subtract B from A (set `binvert` and the `LS Carry In` to 1)
 - Result for all 1-bit ALUs except the LS should always be 0
 - Result for the LS 1-bit ALU should be the result bit from the MS 1-bit ALU

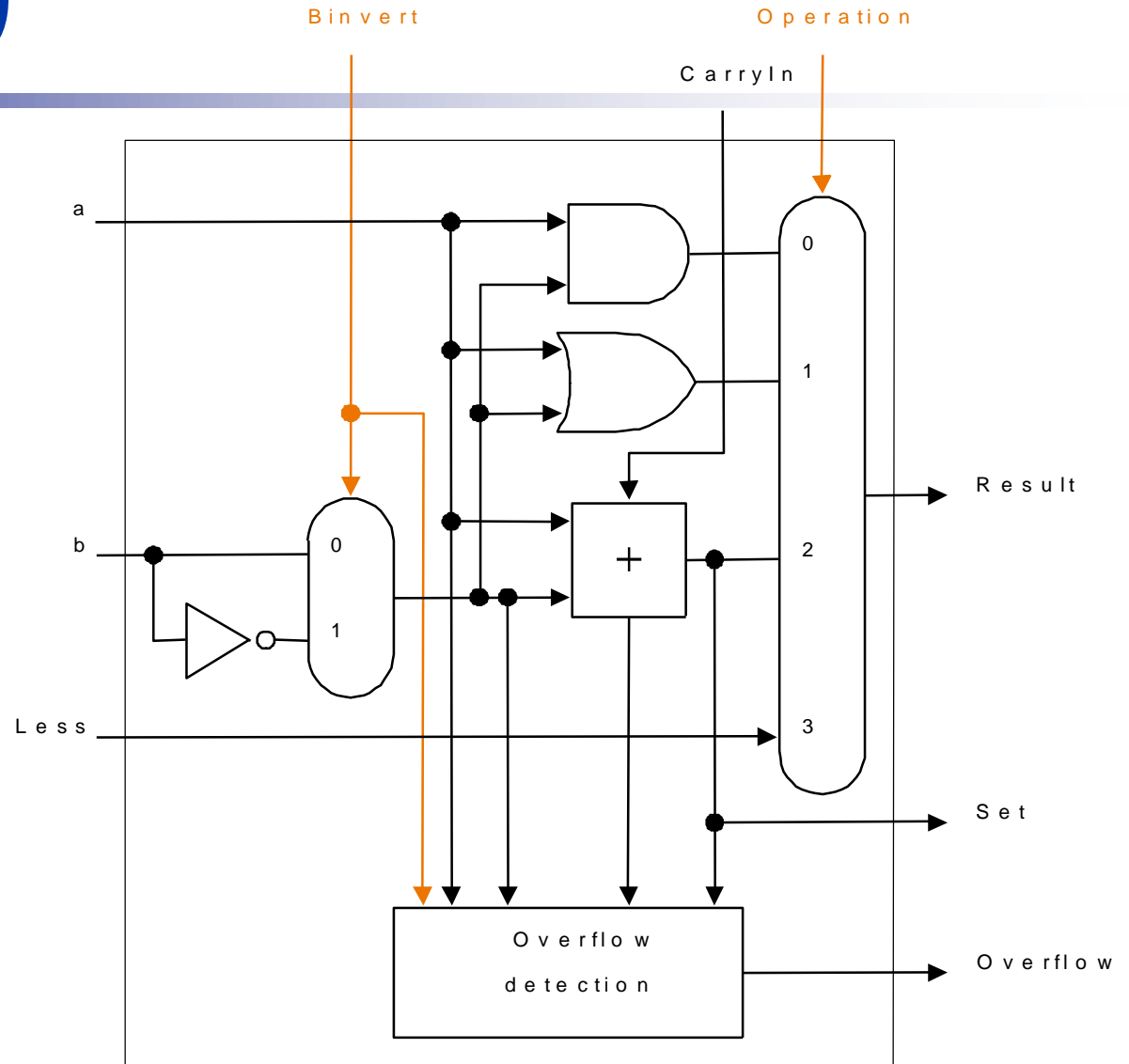
LS: Least significant (rightmost)

MS: Most significant (leftmost)

New 1-bit ALU

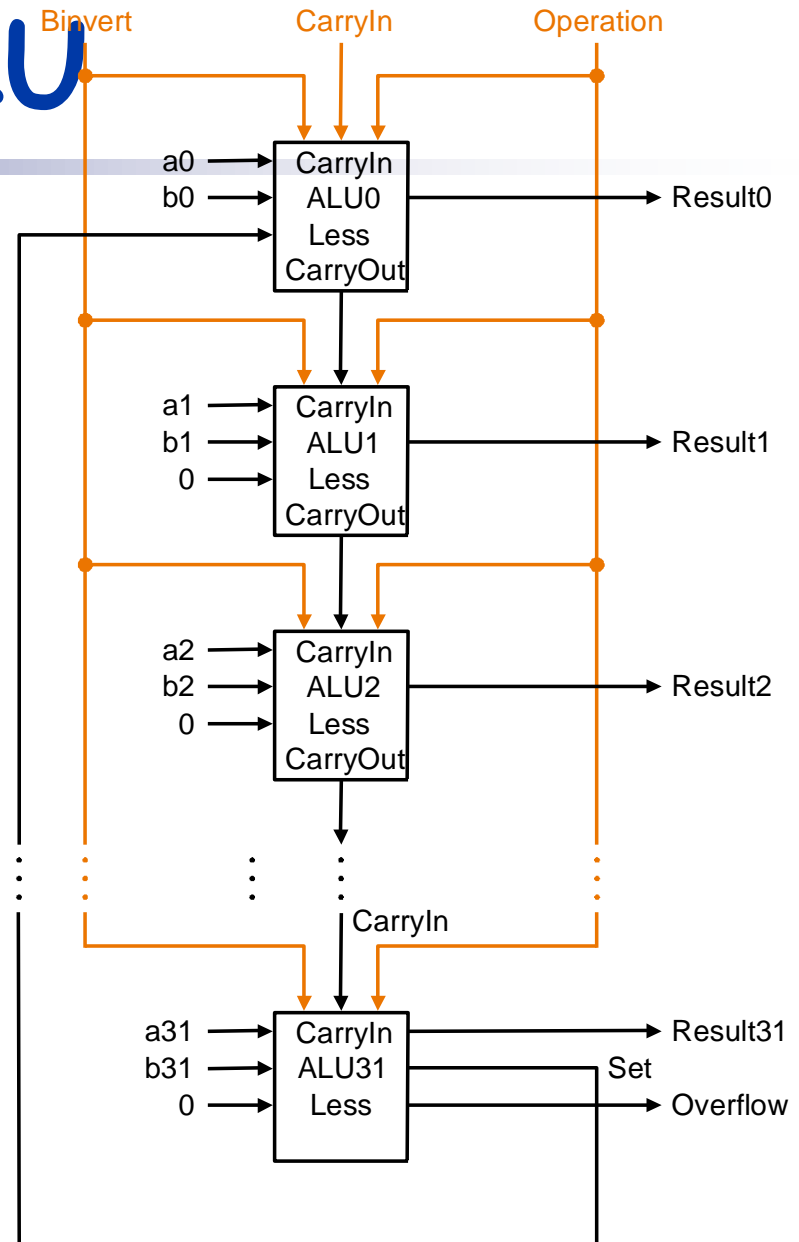


MSB ALU



New 32-bit ALU

- **Less** input is 0 for all but the LS.
- Result of addition in the MS ALU is fed back to the **Less** input of the LS ALU



Put it in a box and give it a name

