

CSCI-2500: Computer Organization

Boolean Logic & Arithmetic for
Computers
(Chapter 3 and App. B)

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
 - sometimes we call the values TRUE / FALSE
- Three operators:
 - OR written as +, as in $A + B$
 - AND written as \cdot , 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 like our old pal *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 like our old nemesis *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:

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

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 *good old 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

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

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

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

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

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

Other Operators

- Boolean Algebra is defined over the 3 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

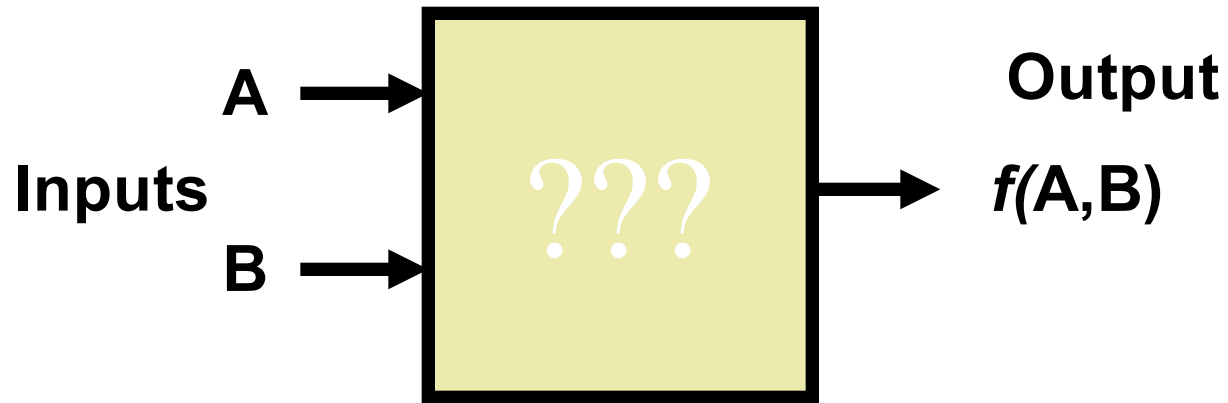
Truth Table for $(X+Y) \cdot Z$

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.

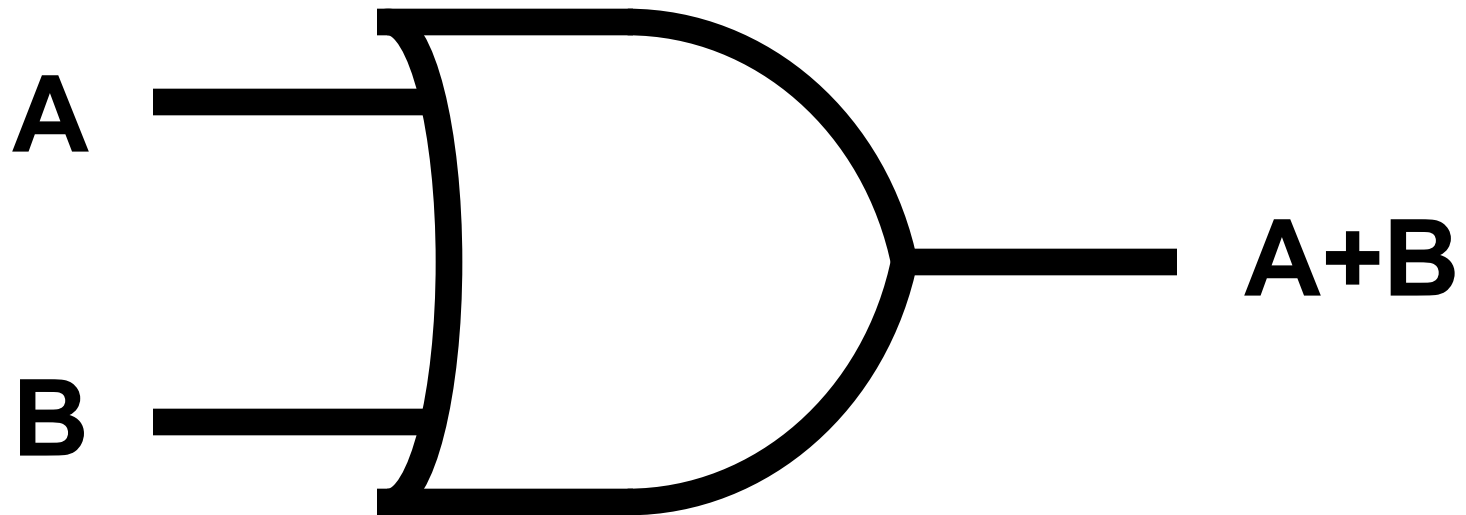
A Gate



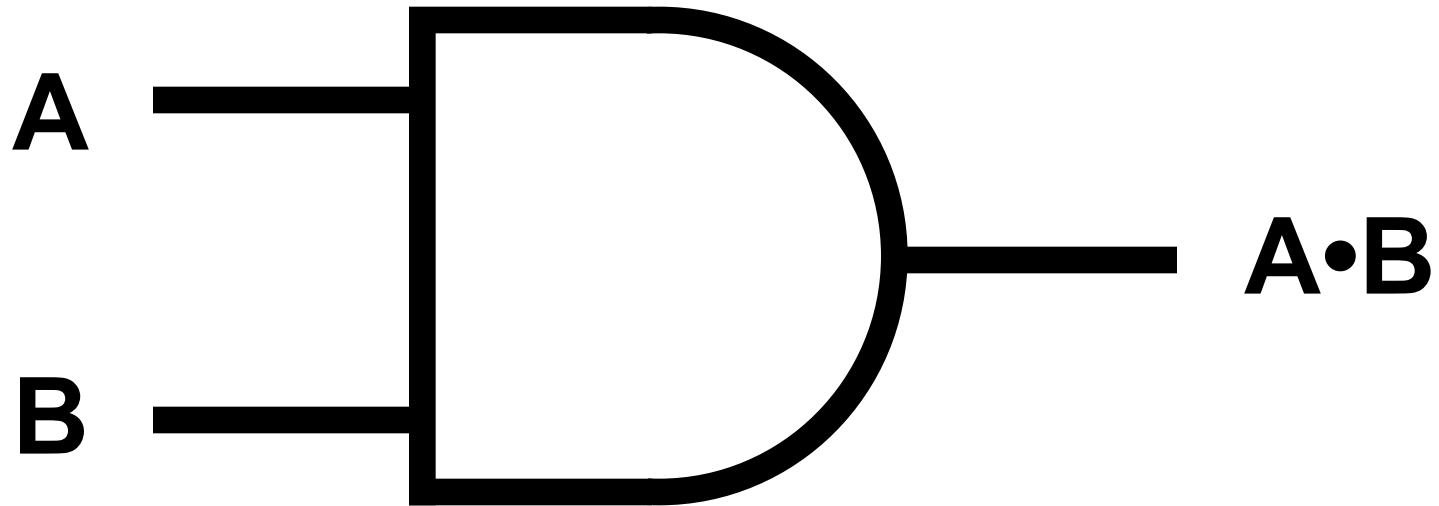
Gates compute something!

- 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.

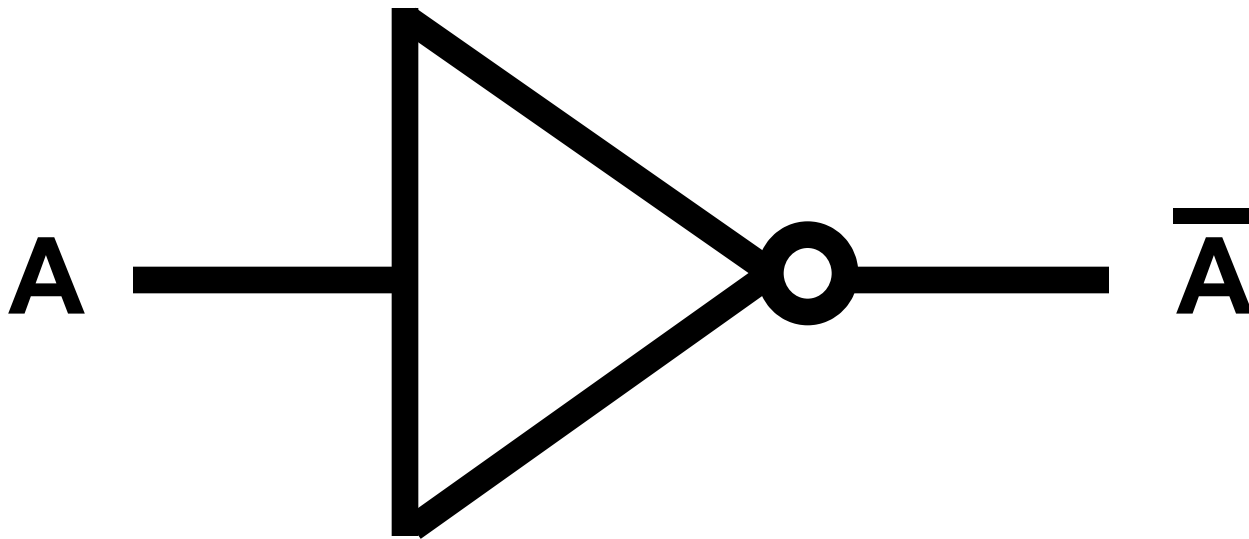
An OR gate



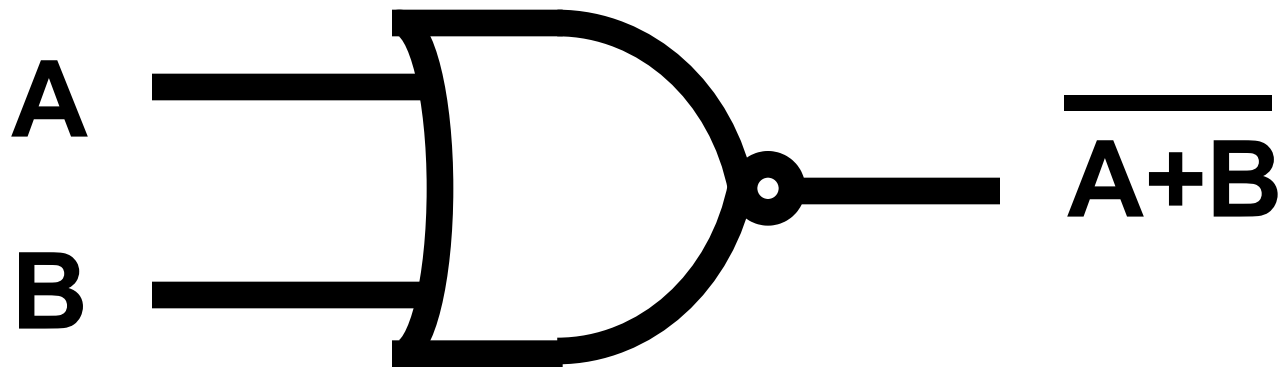
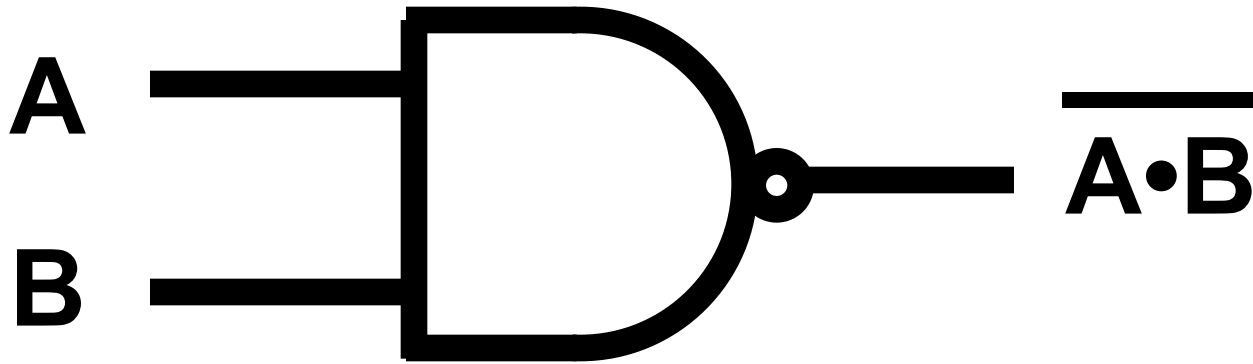
An AND gate



A NOT gate



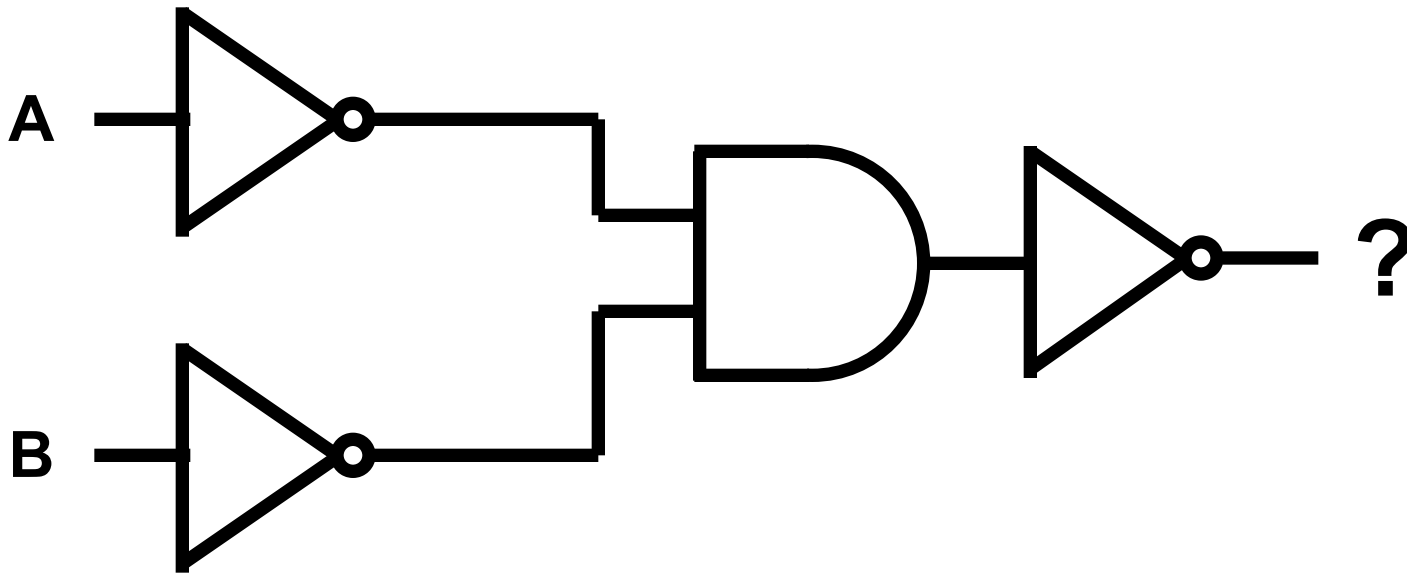
NAND and NOR gates



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



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)

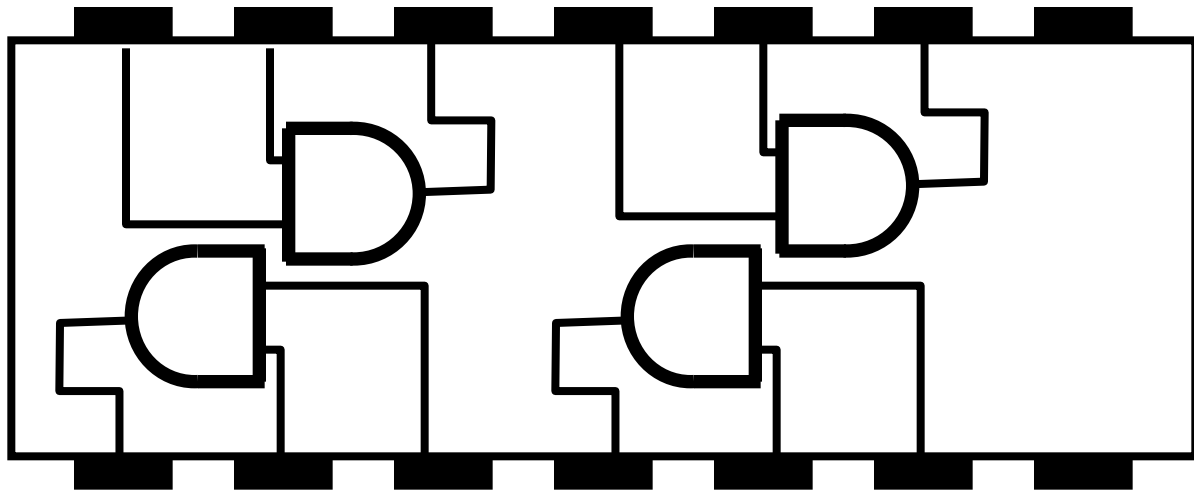
Truth Table

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

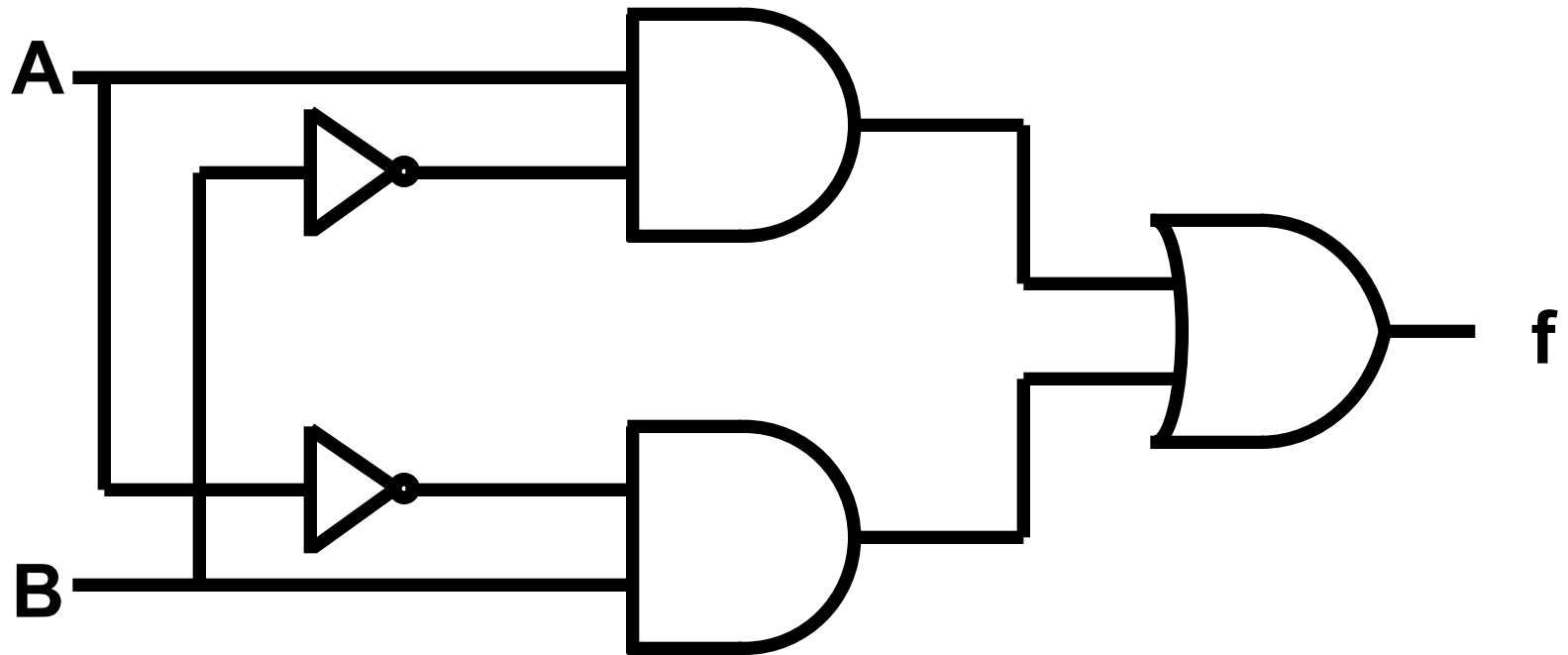
- Given a Boolean function expressed as a truth table or Boolean Equation, there are many possible implementations.
- The actual implementation depends on what kind of gates are available.
- In general we want to minimize the number of gates.

Example: $f = A \cdot \bar{B} + \bar{A} \cdot B$

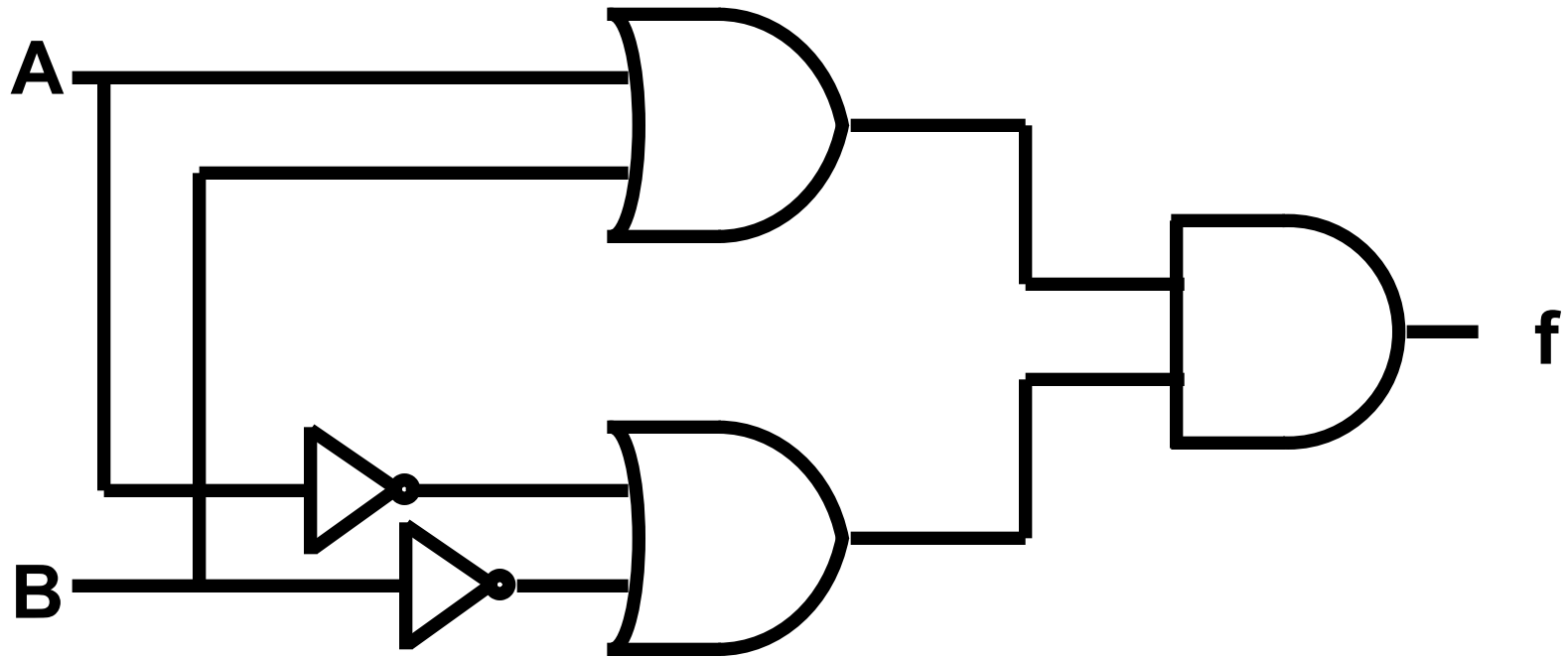
A	B	$A \cdot \bar{B}$	$\bar{A} \cdot B$	f
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

One Implementation

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



Another Implementation



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

Proof it's the same function

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

DeMorgan's Law

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

DeMorgan's Laws

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

Distributive

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

Distributive

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

Inverse, Identity


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

DeMorgan's Law

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

DeMorgan's Laws

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



Logic Design

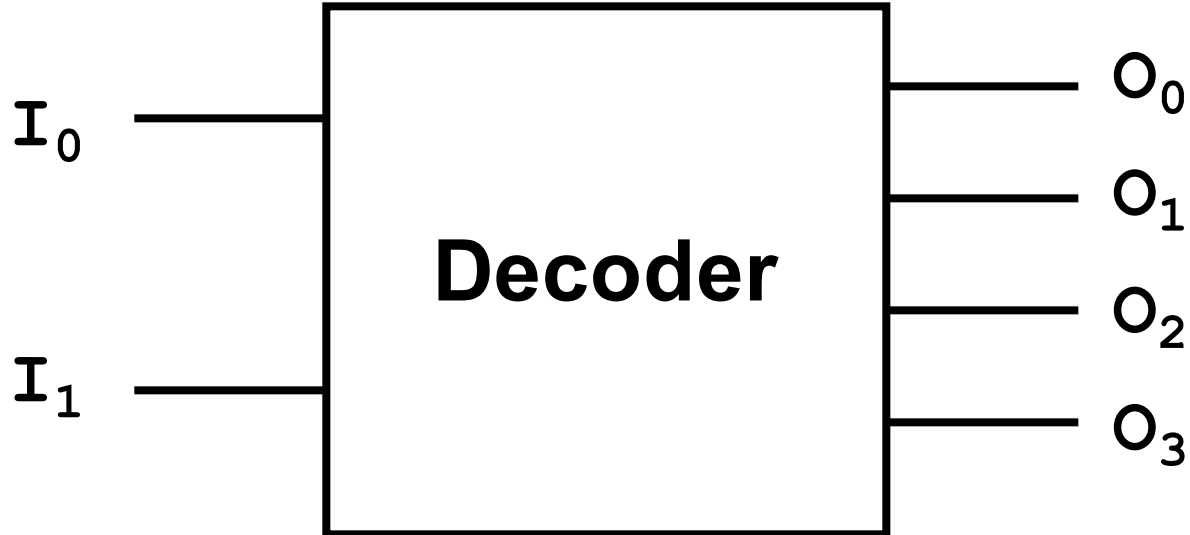
Common Components

- 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_0	I_1	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

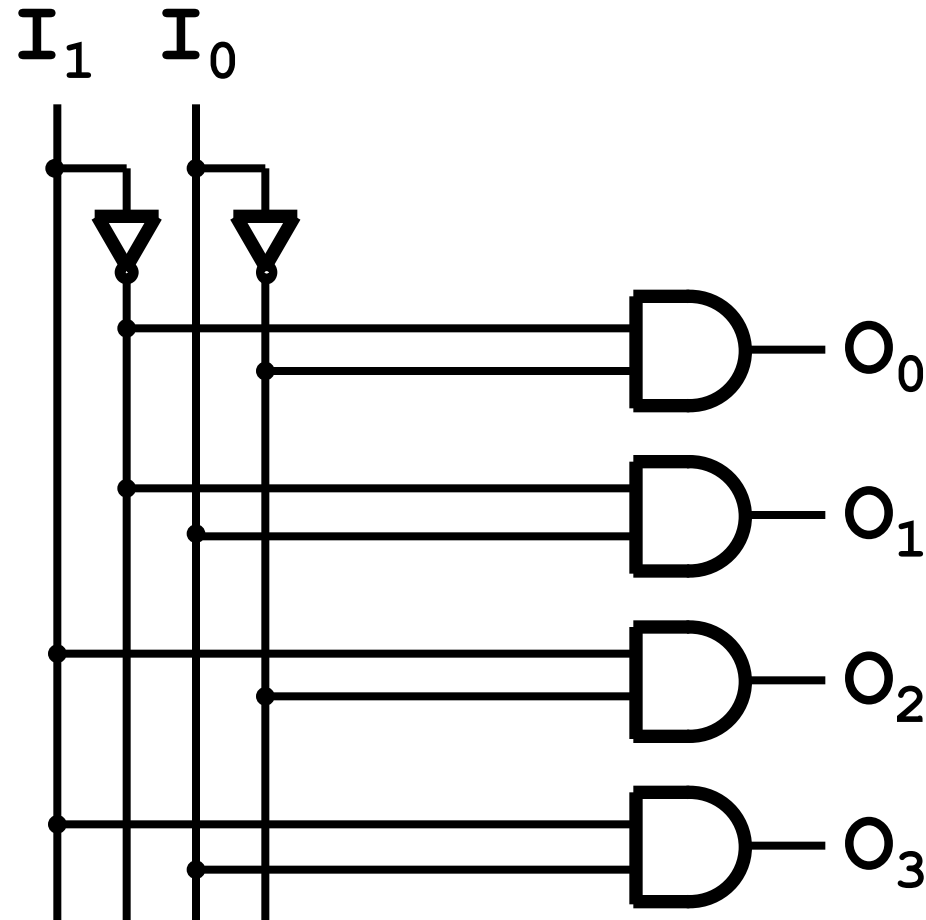
$$O_0 = \overline{I_0} \cdot \overline{I_1}$$

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

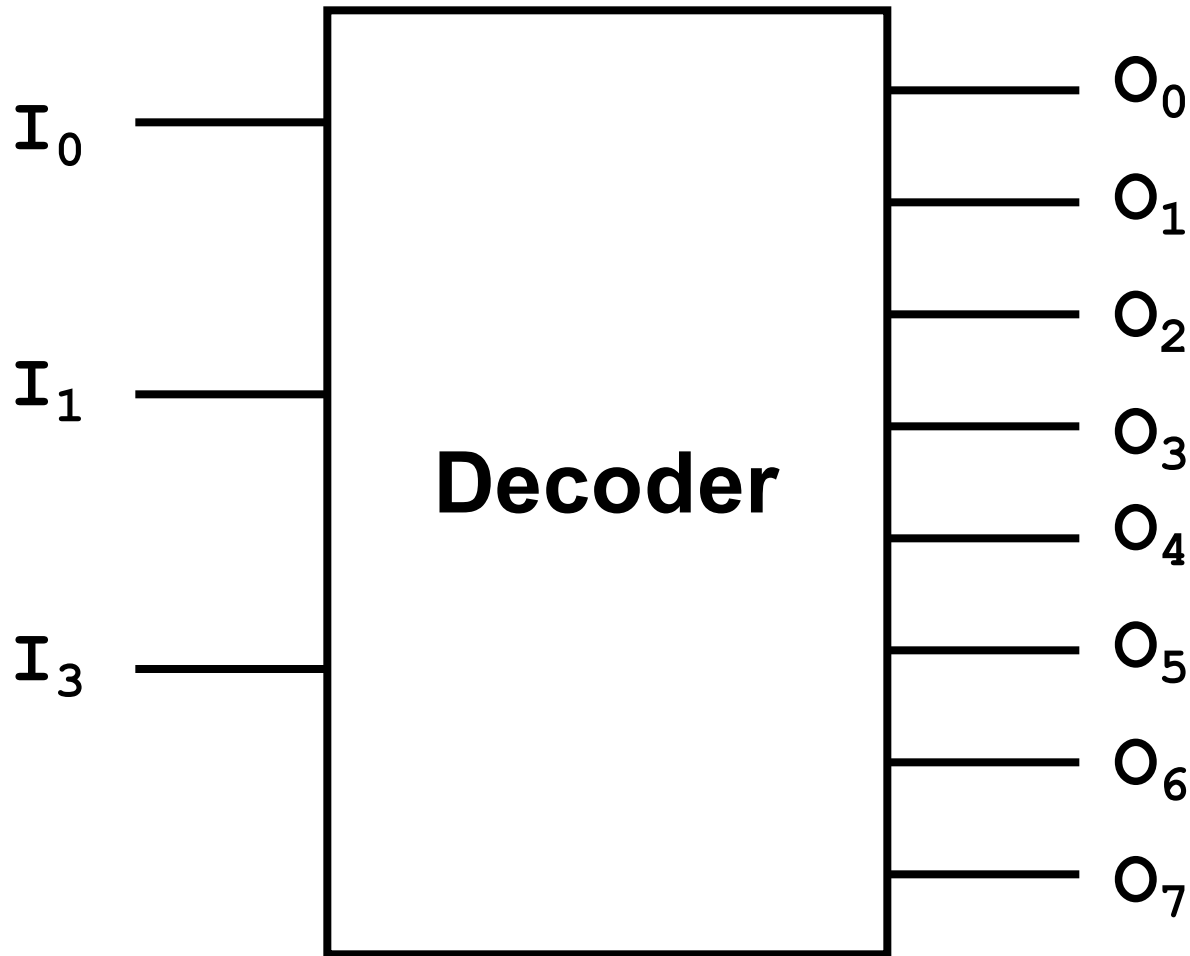
$$O_2 = I_0 \cdot \overline{I_1}$$

$$O_3 = I_0 \cdot I_1$$

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

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

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

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

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

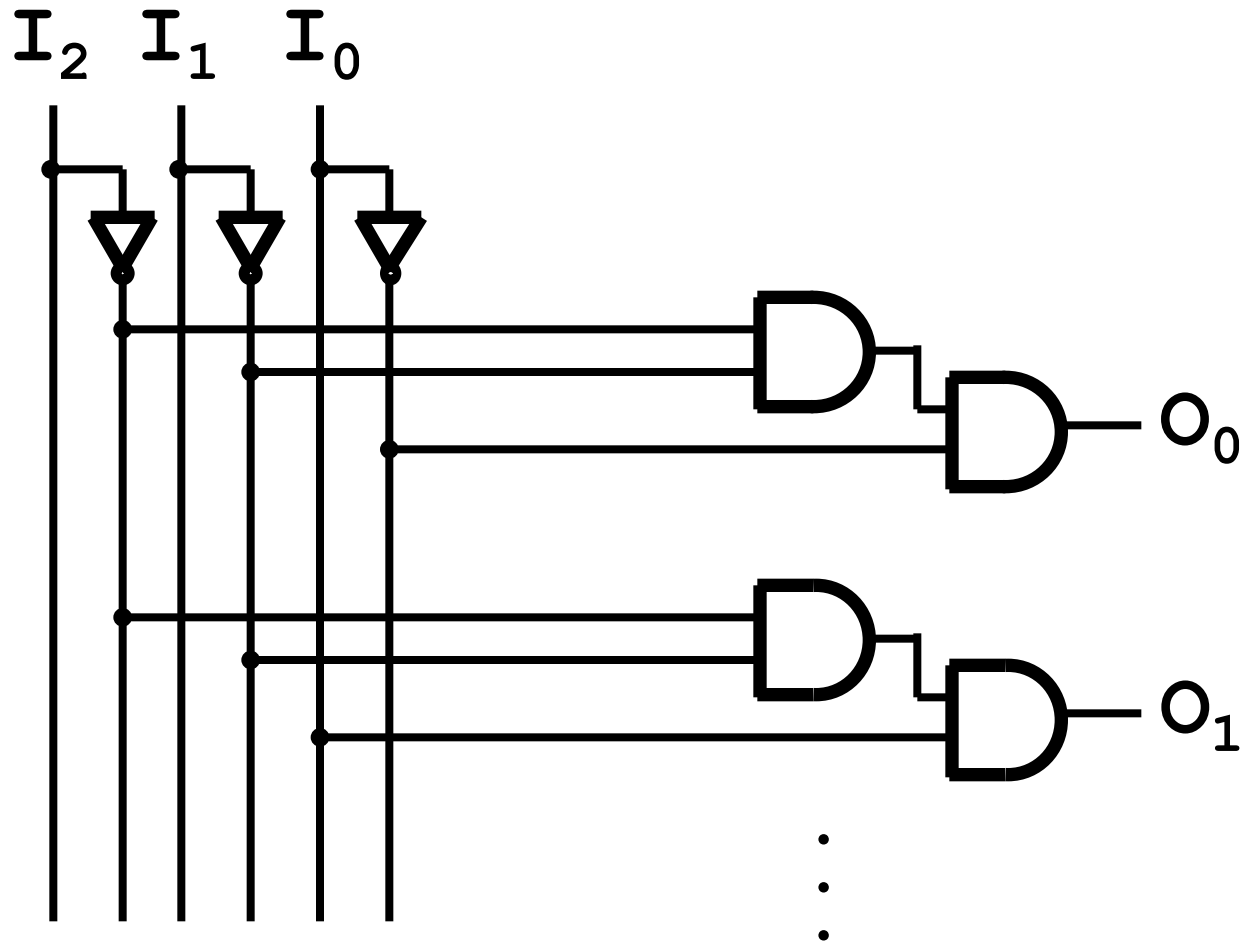
$$O_4 = I_2 \cdot \overline{I_1} \cdot \overline{I_0}$$

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

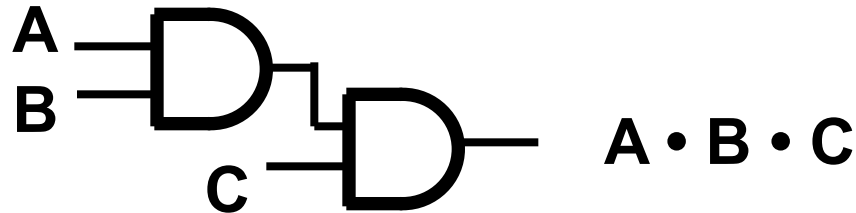
$$O_6 = I_2 \cdot I_1 \cdot \overline{I_0}$$

$$O_7 = I_2 \cdot I_1 \cdot I_0$$

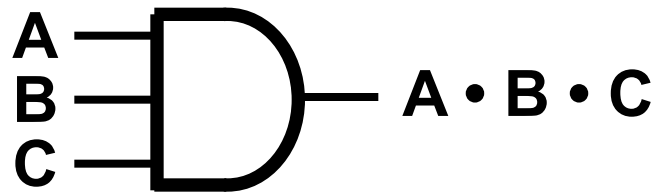
3-Decoder Partial Implementation



A Useful Simplification

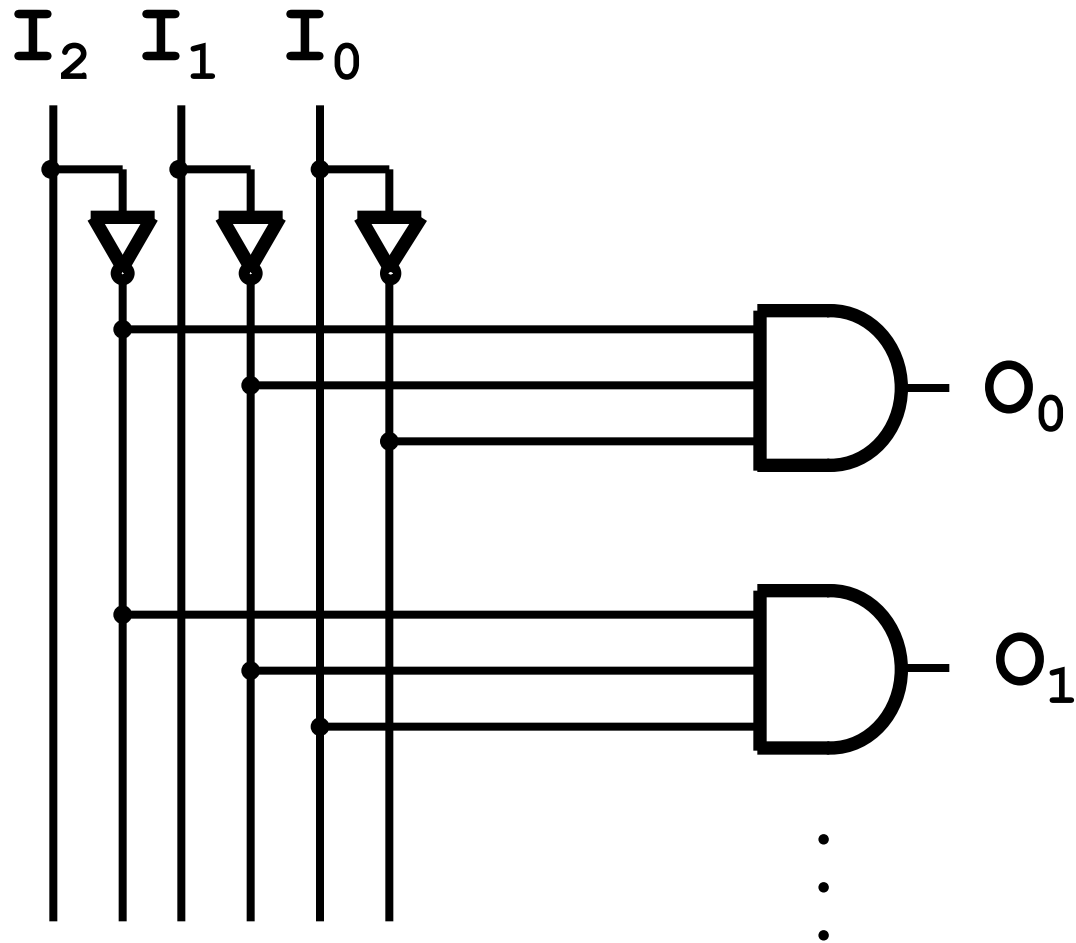


The above logic diagram is often abbreviated as shown below:

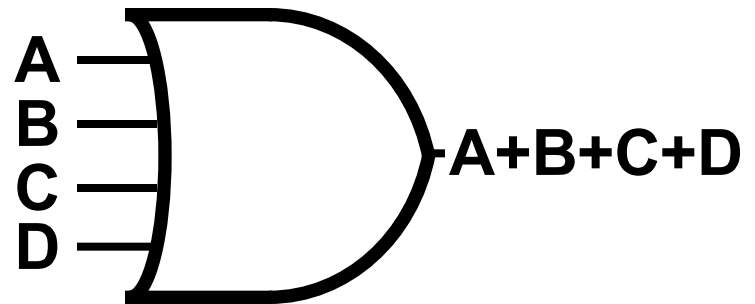
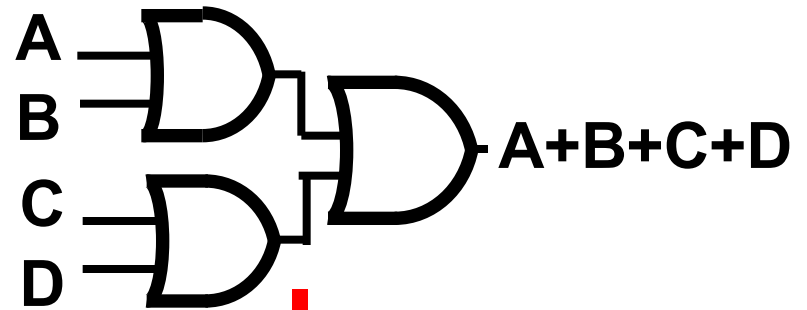
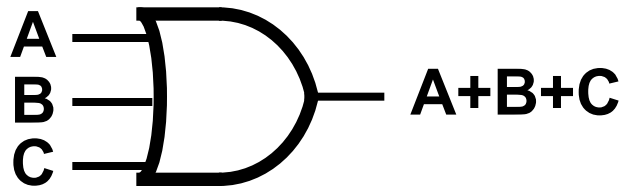
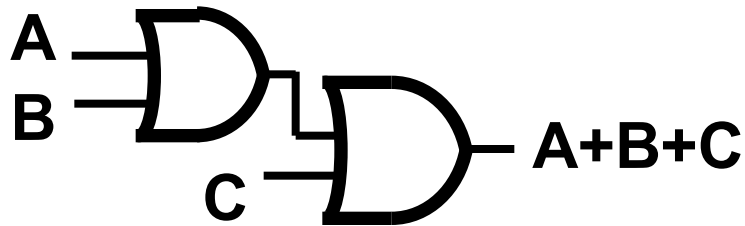


We can do this (without possible confusion) because of the associative property.

Revised Partial 3-Decoder



Multiple Input Or Gates



2 Input Multiplexor

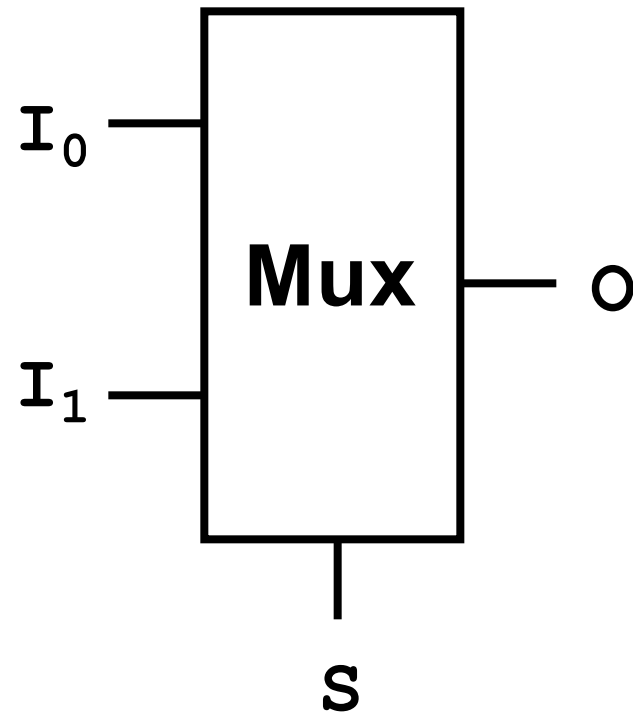
Inputs: I_0 and I_1

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

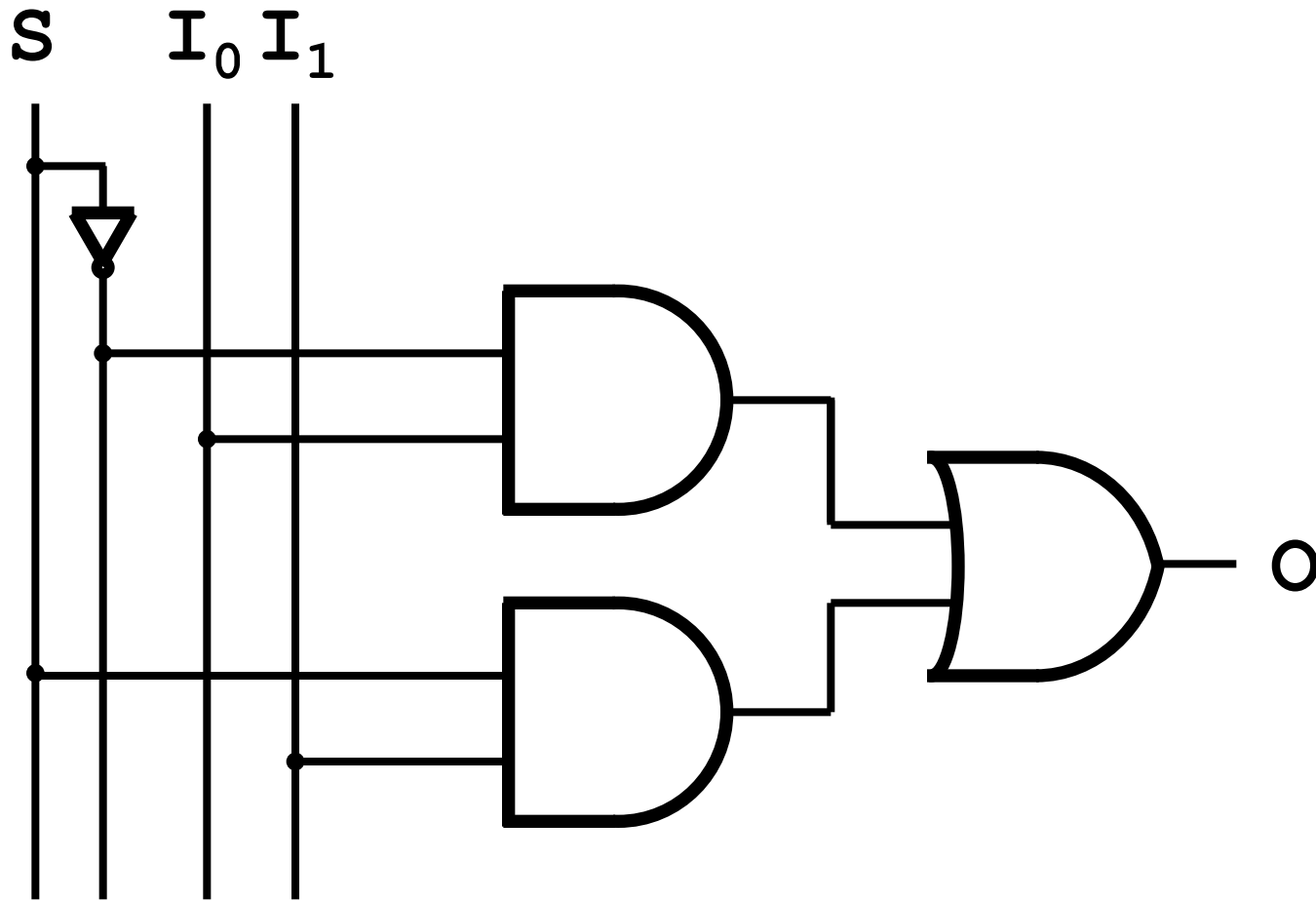
2-Mux Boolean Expression

$$O = (I_0 \cdot \bar{S}) + (I_1 \cdot S)$$

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



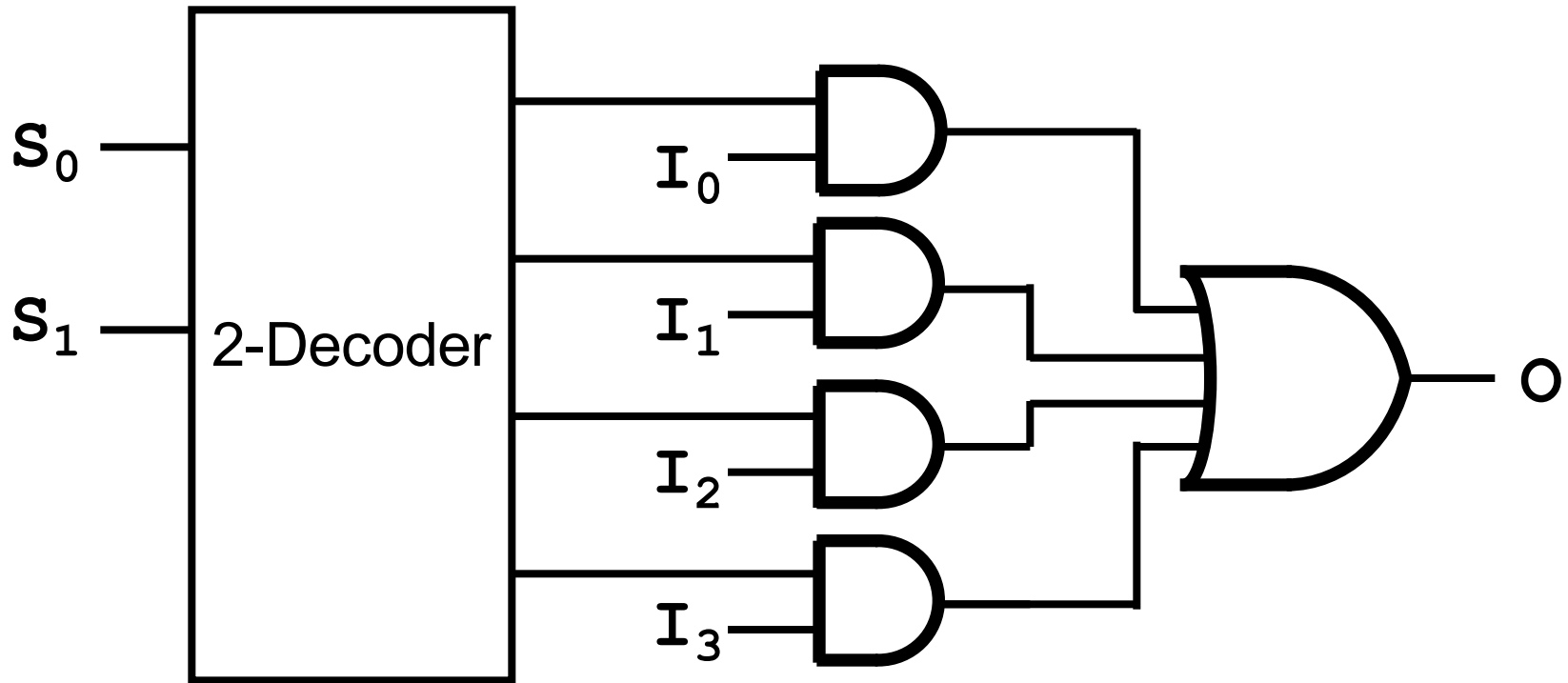
4 Input Multiplexor

- 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 S .

—

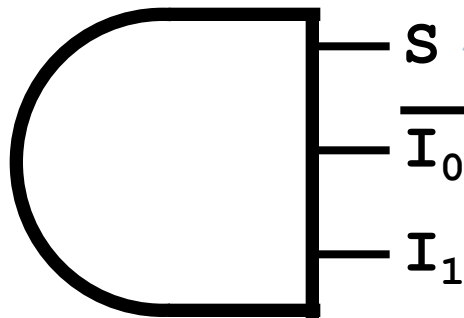
Connect I_0 and I_1 in the same way.

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



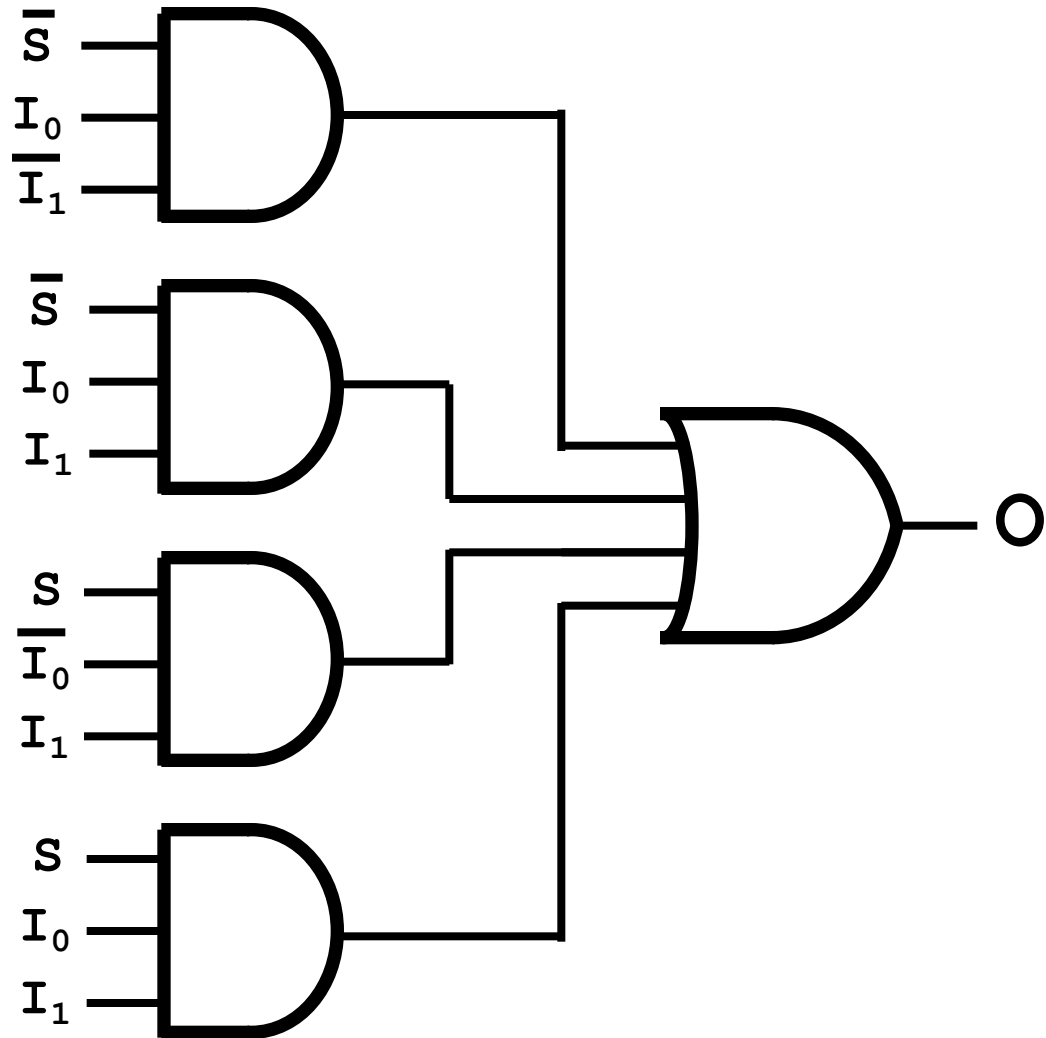
SOP Construction

- 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.

4-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



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 \bar{S} to a 3-input OR gate, otherwise connect S .

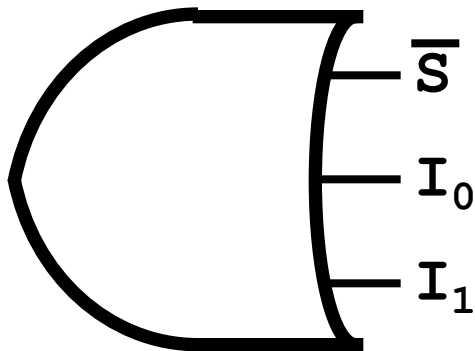
Connect I_0 and I_1 in the same way.

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



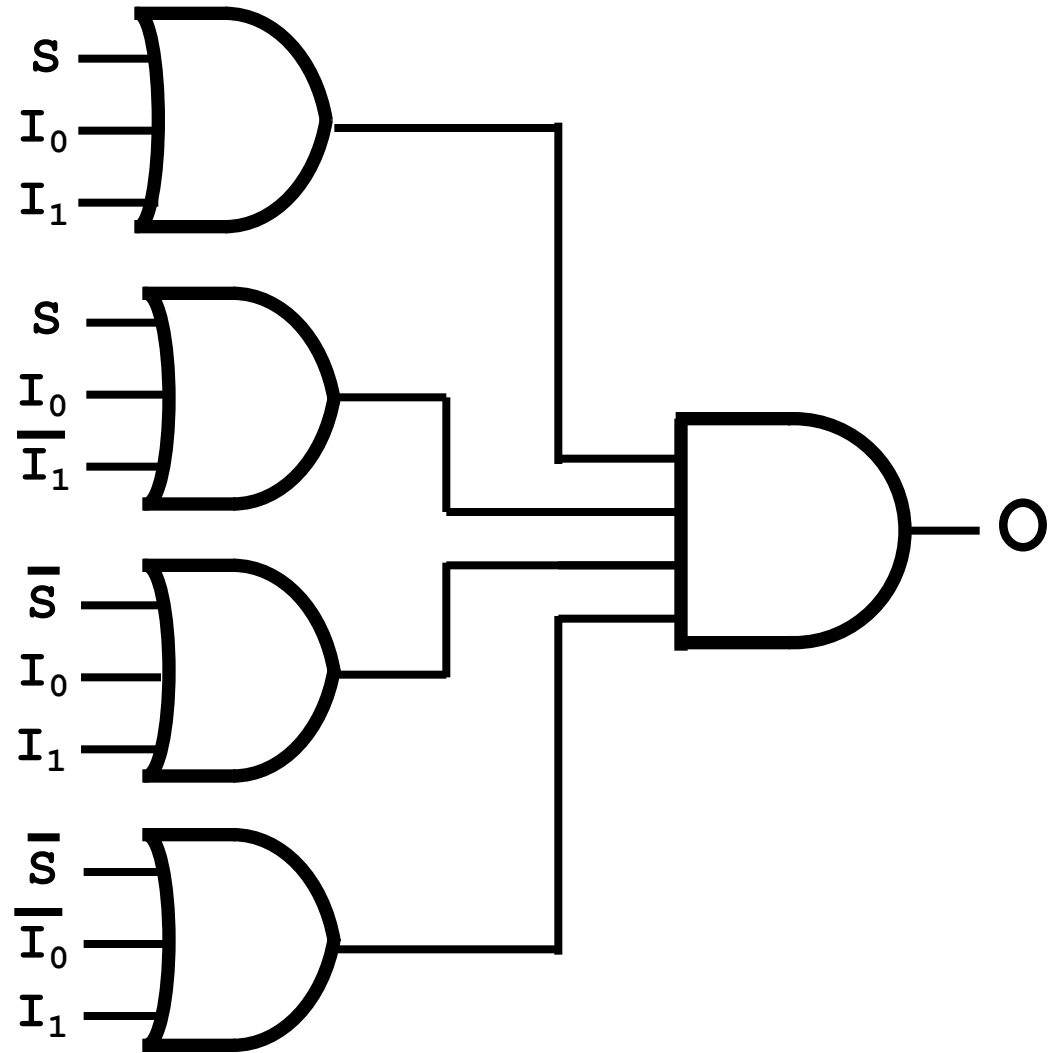
POS Construction

- 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



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.

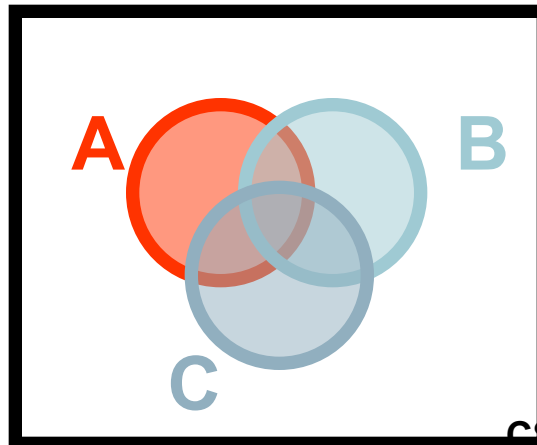
Karnaugh Map (K-map)

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

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

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

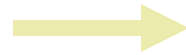


K-Map Example

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

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:

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

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 2 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 1 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 3 possible *mergings* 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

K-Map

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

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

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

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

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

K-Map Concept

- A professional *Logic Designer* would need to use minimization techniques every day.
- We are just amateurs, so all we need to know is the general idea.
 - that there are systematic procedures for minimizing SOP and POS form Boolean equations.

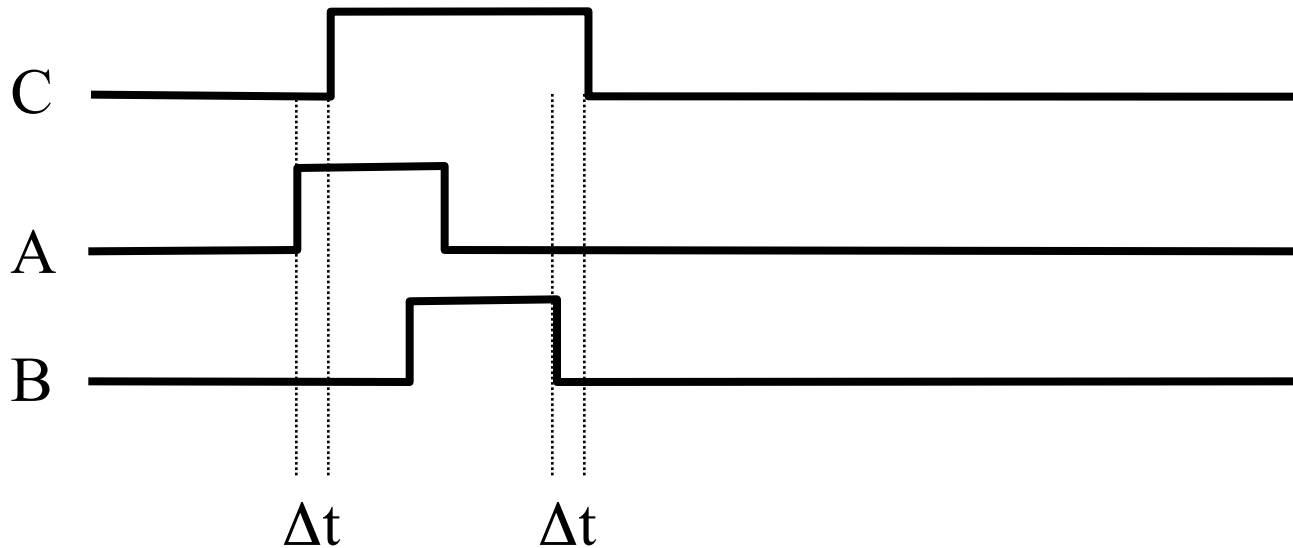
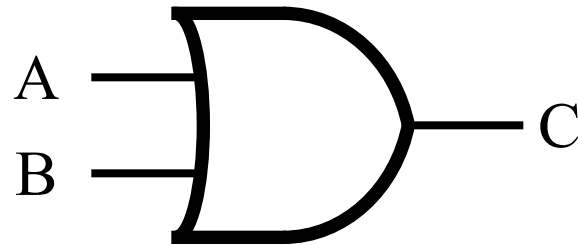
Combinational vs. Sequential

- Combinational: output depends completely on the value of the inputs.
 - time doesn't matter.
- Sequential: output also depends on the *state a little while ago*.
 - can depend on the value of the output some time in the past.

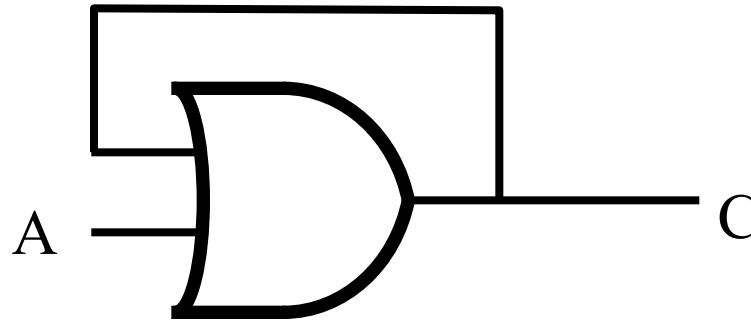
Memory

- Think about how you might design a combinational circuit that could be used as a single bit *memory*.
- Use your *memory* to recall that the output of a gate can change whenever the inputs change.

Gate Timing



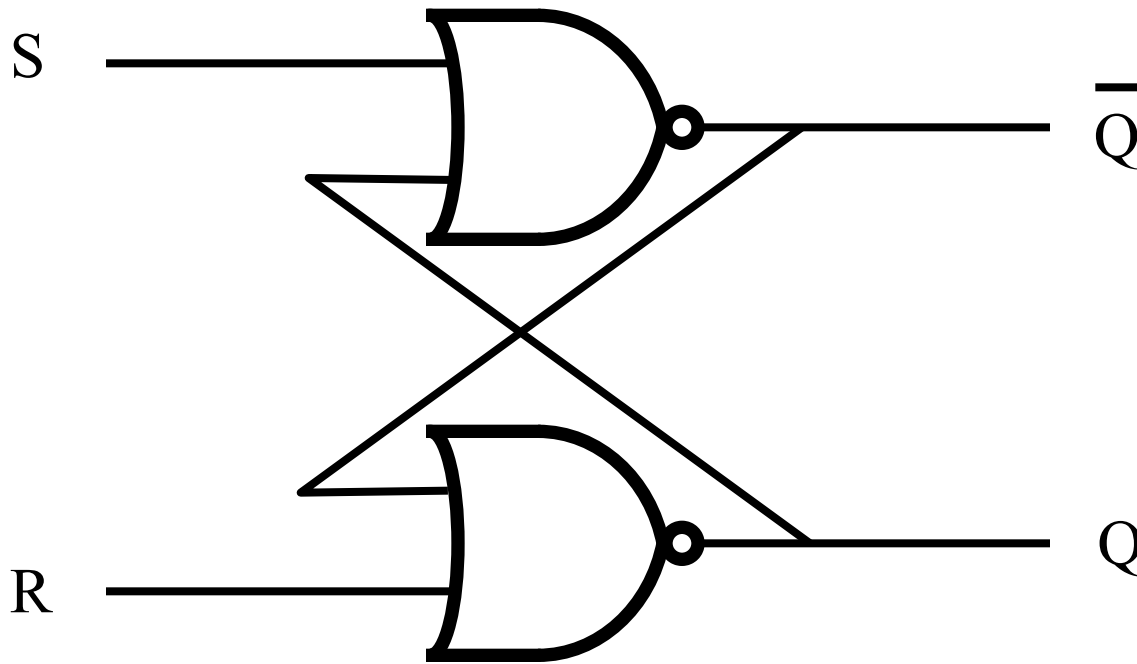
Feedback



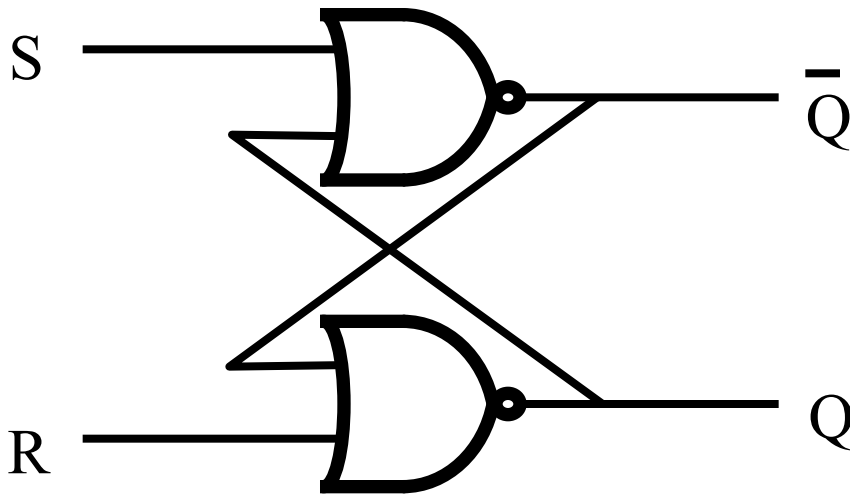
- What happens when A changes from 1 to 0?

S-R latch

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0



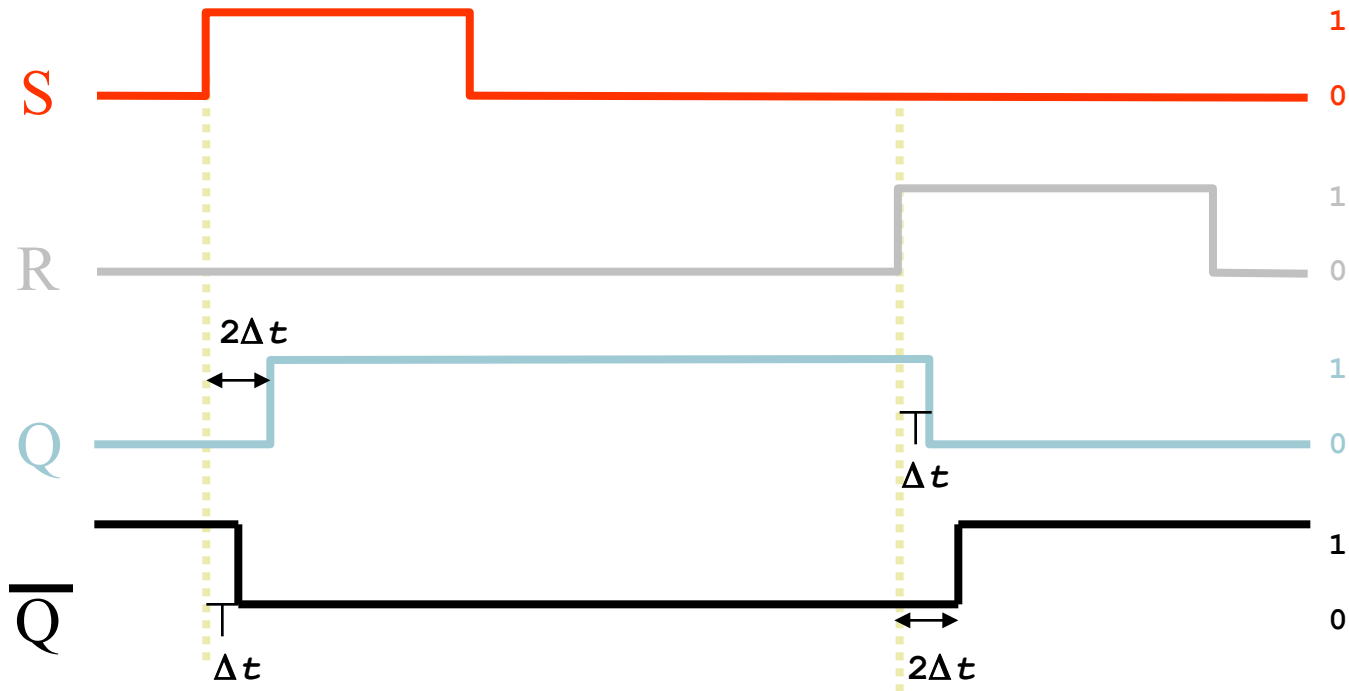
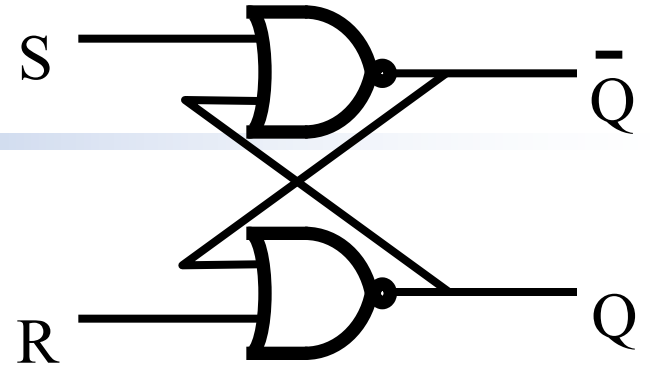
S-R latch Truth Table



If S and R = 1, then Q's output is undefined

Q_t	S_t	R_t	Q_{t+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0?
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0?

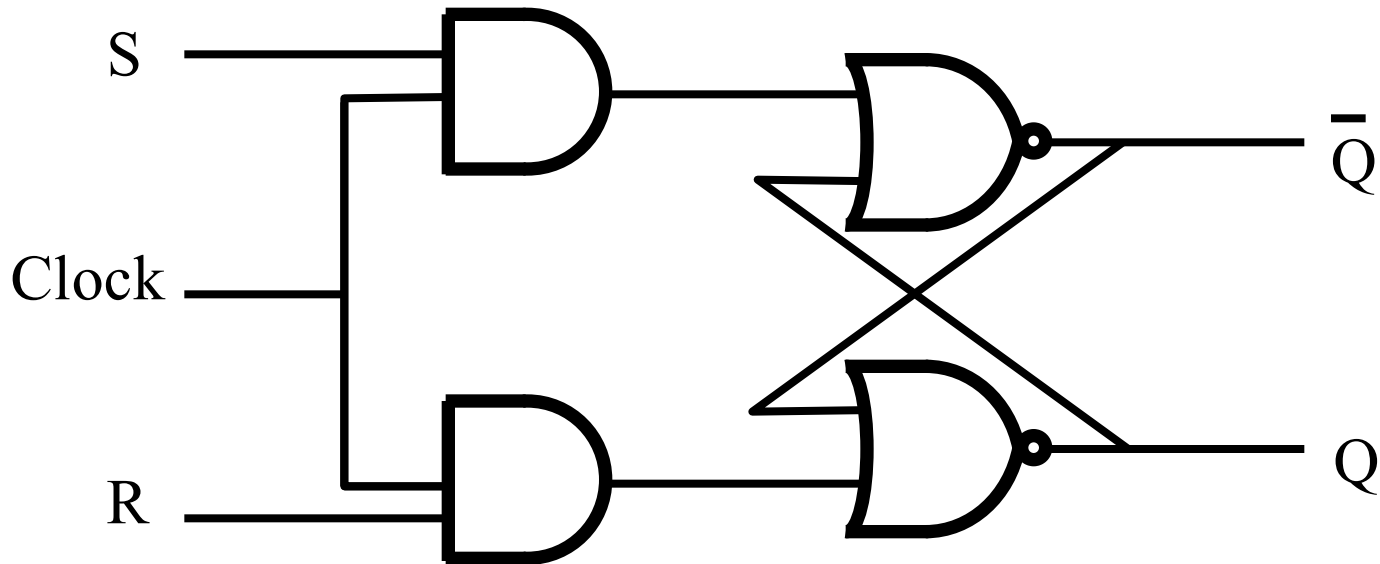
S-R latch Timing



Clocked S-R Latch

- Inside a computer we want the output of gates to change only at specific times.
- We can add some circuitry to make sure that changes occur only when a *clock* changes (when the clock changes from 0 to 1).

Clocked S-R Latch



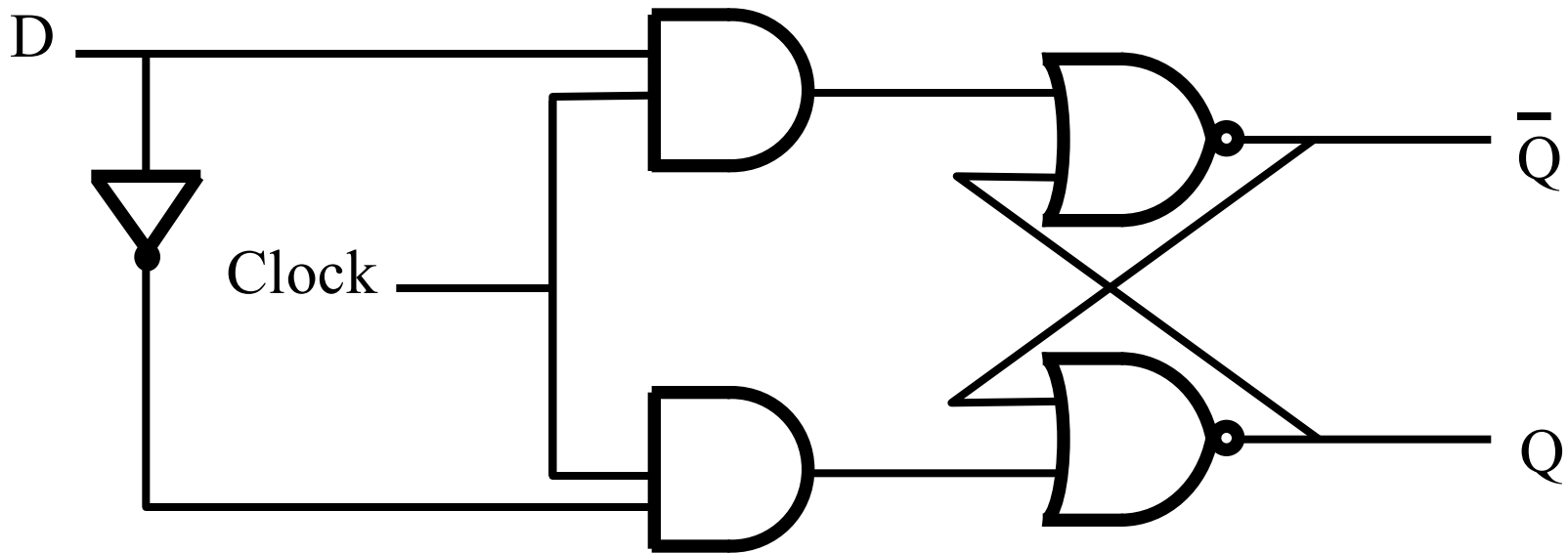
- Q only changes when the Clock is a 1.
- If Clock is 0, neither S or R *reach* the NOR gates.

What if $S=R=1$?

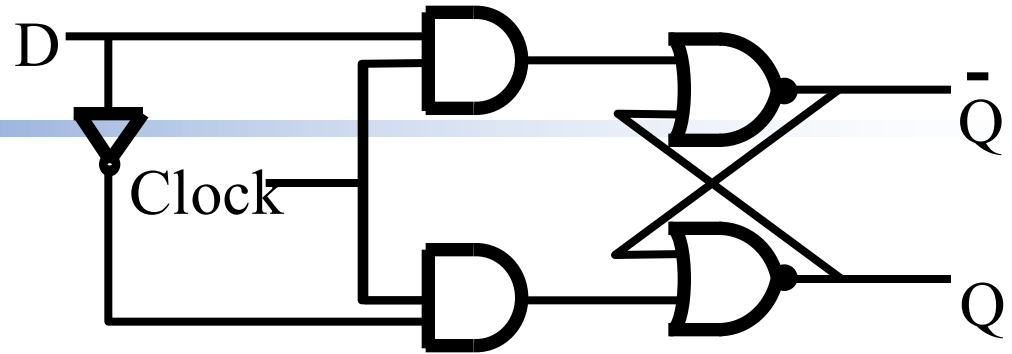
- The truth table shows ? when $S=R=1$.
- The value of Q is undetermined.
 - The circuit is not *stable*.
- We can make sure that $S=R \neq 1$ now that we have a clock.

Avoiding $S=R=1$:

D Flip-Flop

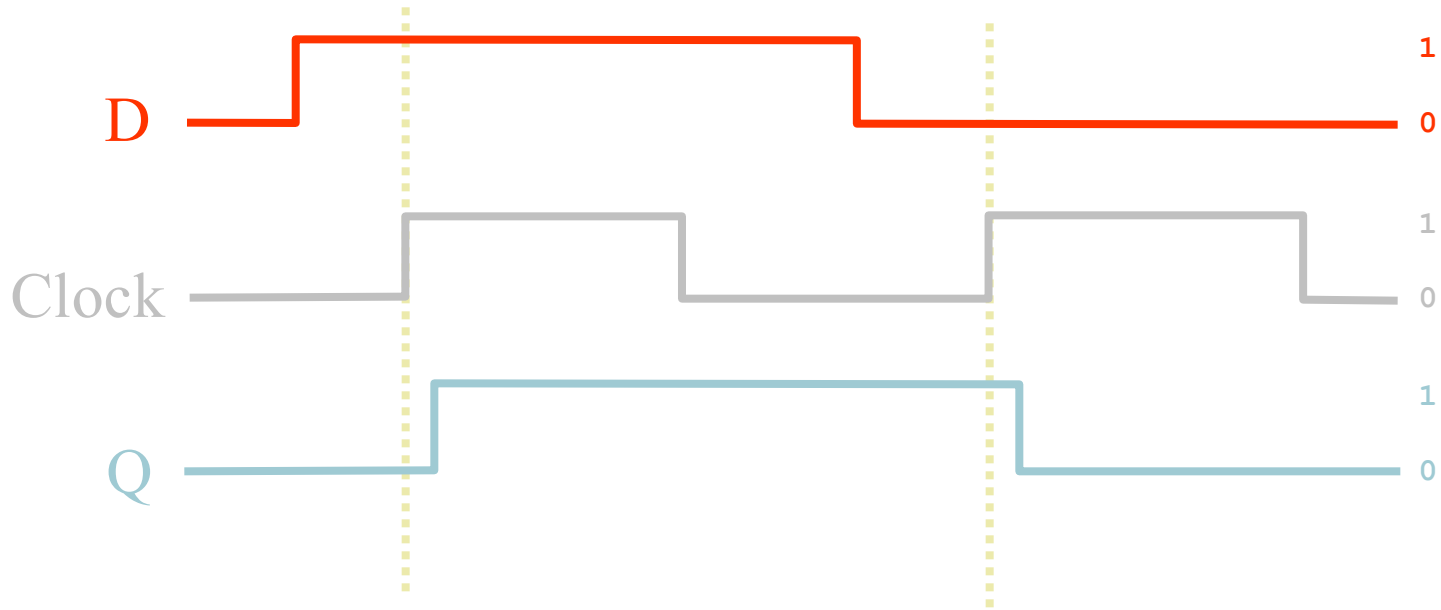


D Flip-Flop



- Now have only one input: D.
- If D is a 1 when the clock becomes 1, the circuit will *remember* the value 1 ($Q=1$).
- If D is a 0 when the clock becomes 1, the circuit will *remember* the value 0 ($Q=0$).

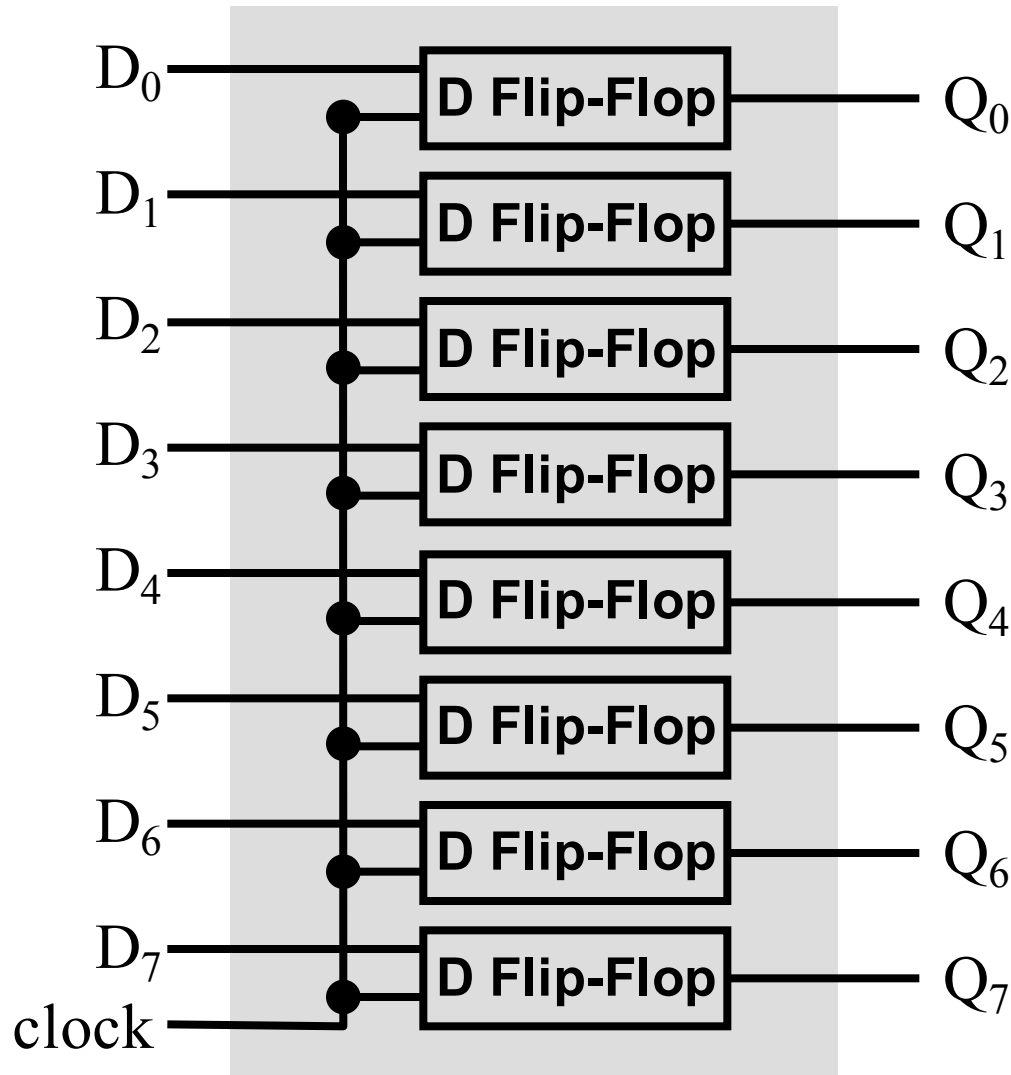
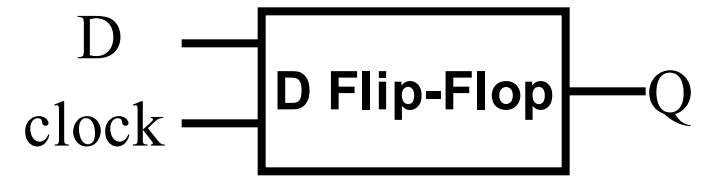
D Flip-Flop Timing



8 Bit Memory

- We can use 8 D Flip-Flops to create an 8 bit memory.
- We have 8 inputs that we want to *store*, all are *written* at the same time.
 - all 8 flip-flops use the same clock.

8 Bit Memory



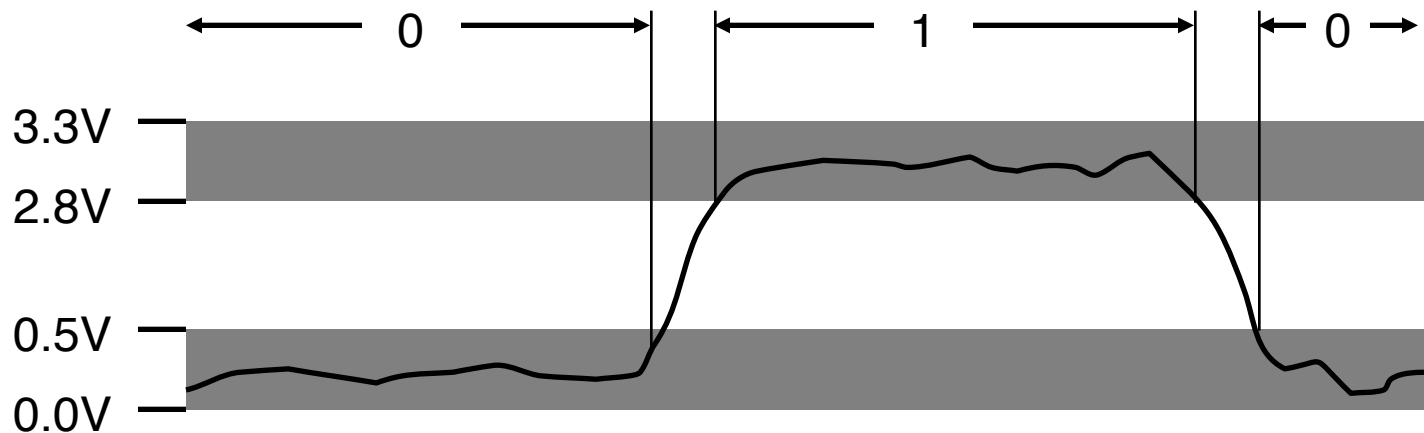
Bits, Bytes & Words

Why Don't Computers Use Base 10?

- Base 10 Number Representation
 - That's why fingers are known as "digits"
 - Natural representation for financial transactions
 - Floating point number cannot exactly represent \$1.20
 - Even carries through in scientific notation
 - 1.5213×10^4
- Implementing Electronically
 - Hard to store
 - ENIAC (First electronic computer) used 10 vacuum tubes / digit
 - Hard to transmit
 - Need high precision to encode 10 signal levels on single wire
 - Messy to implement digital logic functions
 - Addition, multiplication, etc.

Binary Representations

- Base 2 Number Representation
 - Represent 15213_{10} as 11101101101101_2
 - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
 - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$
- Electronic Implementation
 - Easy to store with bistable elements
 - Reliably transmitted on noisy and inaccurate wires



- Straightforward implementation of arithmetic functions

Byte-Oriented Memory Organization

- Programs Refer to Virtual Addresses
 - Conceptually very large array of bytes
 - Actually implemented with hierarchy of different memory types
 - SRAM, DRAM, disk
 - Only allocate for regions actually used by program
 - In Unix and Windows NT, address space private to particular "process"
 - Program being executed
 - Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control Allocation
 - Where different program objects should be stored
 - Multiple mechanisms: static, stack, and heap
 - In any case, all allocation within single virtual address space

Encoding Byte Values

- Byte = 8 bits
 - Binary 00000000_2 to 11111111_2
 - Decimal: 0_{10} to 255_{10}
 - Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write $FA1D37B_{16}$ in C as $0xFA1D37B$
 - Or $0xfa1d37b$

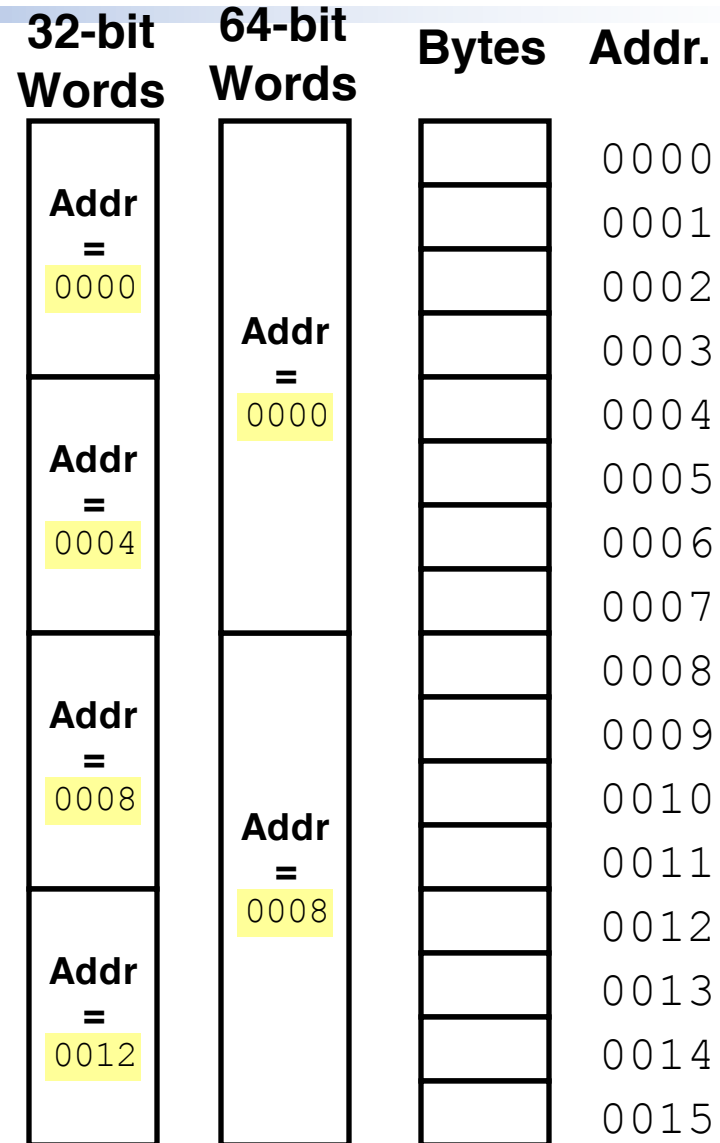
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Machine Words

- Machine Has "Word Size"
 - Nominal size of integer-valued data
 - Including addresses
 - Most current machines are 32 bits (4 bytes)
 - Limits addresses to 4GB
 - Becoming too small for memory-intensive applications
 - High-end systems are 64 bits (8 bytes)
 - Potentially address $\approx 1.8 \times 10^{19}$ bytes
 - Machines support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
 - Address of first byte in word
 - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Data Representations

- Sizes of C Objects (in Bytes)

- C Data Type Compaq Alpha Typical 32-bit Intel IA32

• int	4		4	4
• long int	8		4	4
• char	1		1	1
• short	2		2	2
• float	4		4	4
• double	8		8	8
• long double	8		8	10/12
• char *	8		4	4

- Or any other pointer

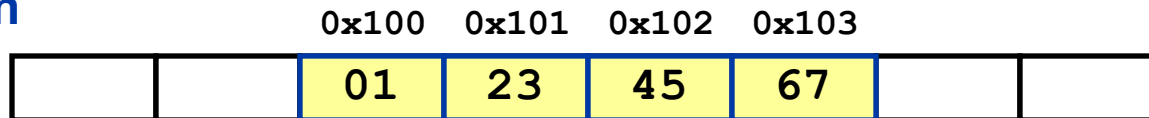
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Conventions
 - Sun's, Mac's are "Big Endian" machines
 - Least significant byte has highest address
 - Alphas, PC's are "Little Endian" machines
 - Least significant byte has lowest address

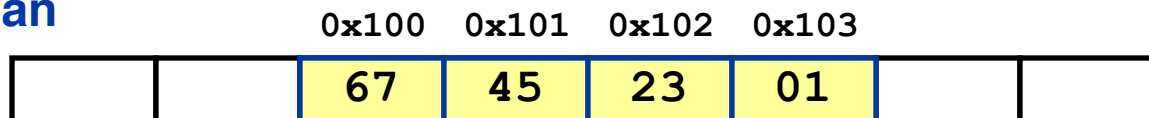
Byte Ordering Example

- Big Endian
 - Least significant byte has highest address
- Little Endian
 - Least significant byte has lowest address
- Example
 - Variable `x` has 4-byte representation `0x01234567`
 - Address given by `&x` is `0x100`

Big Endian



Little Endian



Reading Byte-Reversed Listings

- Disassembly
 - Text representation of binary machine code
 - Generated by program that reads the machine code
- Example Fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab, %ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0, 0x28(%ebx)

- Deciphering Numbers

- Value: 0x12ab
- Pad to 4 bytes: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

Examining Data Representations

- Code to Print Byte Representation of Data
 - Casting pointer to `unsigned char *` creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p\t0x%.2x\n",
               start+i, start[i]);
    printf("\n");
}
```

Printf directives:

%p: Print pointer

%x: Print Hexadecimal

show_bytes Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```
int a = 15213;
0x11ffffcb8  0x6d
0x11ffffcb9  0x3b
0x11ffffcba  0x00
0x11ffffcbb  0x00
```

Representing Integers

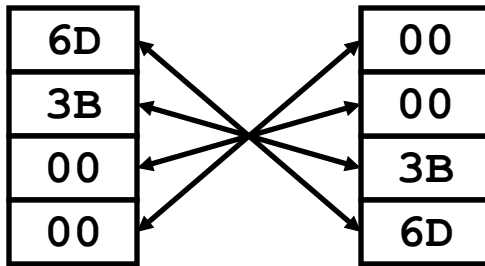
- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`

Decimal: 15213

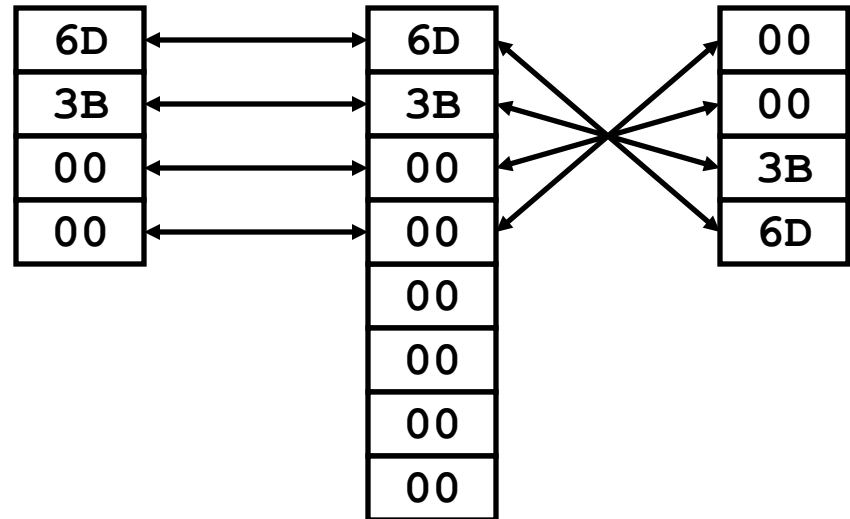
Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

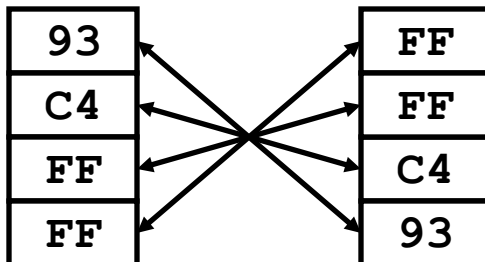
Linux/Alpha A Sun A



Linux c Alpha c Sun c



Linux/Alpha B Sun B



**Two's complement representation
(Covered in future)**

Representing Pointers

Alpha P

- `int B = -15213;`
- `int *P = &B;`

Alpha Address

Hex: 1 F F F F F C A 0
Binary: 0001 1111 1111 1111 1111 1111 1100 1010 0000

A0

FC

FF

FF

01

00

00

00

Sun P

Sun Address

Hex: E F F F F B 2 C
Binary: 1110 1111 1111 1111 1111 1011 0010 1100

EF

FF

FB

2C

Linux P

Linux Address

Hex: B F F F F 8 D 4
Binary: 1011 1111 1111 1111 1111 1000 1101 0100

D4

F8

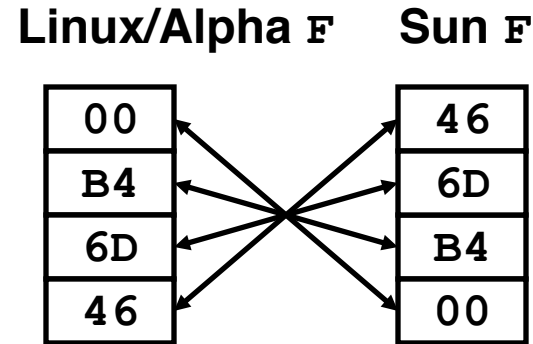
FF

BF

Different compilers & machines assign different locations to objects

Representing Floats

- Float $F = 15213.0;$



IEEE Single Precision Floating Point Representation

Hex: 4 6 6 D B 4 0 0
Binary: 0100 0110 0110 1101 1011 0100 0000 0000
15213: 1110 1101 1011 01



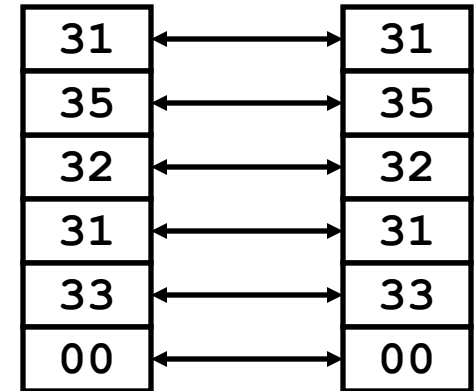
Not same as integer representation, but consistent across machines

Can see some relation to integer representation, but not obvious

Representing Strings

- Strings in C
 - Represented by array of characters
 - Each character encoded in ASCII format
 - Standard 7-bit encoding of character set
 - Other encodings exist, but uncommon
 - Character "0" has code $0x30$
 - Digit i has code $0x30+i$
 - String should be null-terminated
 - Final character = 0
- Compatibility
 - Byte ordering not an issue
 - Data are single byte quantities
 - Text files generally platform independent
 - Except for different conventions of line termination character(s)!

```
char S[6] = "15213";  
Linux/Alpha s Sun s
```



Machine-Level Code Representation

- Encode Program as Sequence of Instructions
 - Each simple operation
 - Arithmetic operation
 - Read or write memory
 - Conditional branch
 - Instructions encoded as bytes
 - Alpha's, Sun's, Mac's use 4 byte instructions
 - Reduced Instruction Set Computer (RISC)
 - PC's use variable length instructions
 - Complex Instruction Set Computer (CISC)
 - Different instruction types and encodings for different machines
 - Most code not binary compatible
- Programs are Byte Sequences Too!

Representing Instructions

- `int sum(int x, int y)`
- `{`
- `return x+y;`
- `}`
- For this example, Alpha & Sun use two 4-byte instructions
 - Use differing numbers of instructions in other cases
- PC uses 7 instructions with lengths 1, 2, and 3 bytes
 - Same for NT and for Linux
 - NT / Linux not fully binary compatible

Alpha sum

00
00
30
42
01
80
FA
6B

Sun sum

81
C3
E0
08
90
02
00
09

PC sum

55
89
E5
8B
45
0C
03
45
08
89
EC
5D
C3

Different machines use totally different instructions and encodings

Bit-Level Operations in C

- Operations $\&$, $|$, \sim , \wedge Available in C
 - Apply to any "integral" data type
 - long, int, short, char
 - View arguments as bit vectors
 - Arguments applied bit-wise
- Examples (Char data type)
 - $\sim 0x41 \rightarrow 0xBE$
 $\sim 01000001_2 \rightarrow 10111110_2$
 - $\sim 0x00 \rightarrow 0xFF$
 $\sim 00000000_2 \rightarrow 11111111_2$
 - $0x69 \ \& \ 0x55 \rightarrow 0x41$
 $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
 - $0x69 \ | \ 0x55 \rightarrow 0x7D$
 $01101001_2 \ | \ 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
 - `&&`, `||`, `!`
 - View 0 as "False"
 - Anything nonzero as "True"
 - Always return 0 or 1
 - Early termination
- Examples (char data type)
 - `!0x41 --> 0x00`
 - `!0x00 --> 0x01`
 - `!!0x41 --> 0x01`

 - `0x69 && 0x55 --> 0x01`
 - `0x69 || 0x55 --> 0x01`
 - `p && *p` (avoids null pointer access)

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - Shift bit-vector x right y positions
 - Throw away extra bits on right
 - Logical shift
 - Fill with 0's on left
 - Arithmetic shift
 - Replicate most significant bit on right
 - Useful with two's complement integer representation

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse

$$A \oplus A = 0$$

```
void swap(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```

	*x	*y
Begin	A	B
1	$A \oplus B$	B
2	$A \oplus B$	$(A \oplus B) \oplus B = A$
3	$(A \oplus B) \oplus A = B$	A
End	B	A

Two's Complement

Range of integers

- A mathematical integer ranges from $-\infty$ to $+\infty$
- Consequently, a mathematical integer consists of an unbounded number of bits.
- No computer can store all the integers in this range (would require infinite storage).
- To use computer memory more efficiently, two broad categories of integer representation have been developed: unsigned integers and signed integers.

Unsigned & signed integer arithmetic

- An unsigned integer ranges from 0 to $+\infty$.
- The maximum unsigned integer that a computer can store depends on the number of bits the computer allocates to store an unsigned integer.

Range of unsigned integers

<i># of Bits</i>	<i>Range</i>
8	0 .. 255
16	0 .. 65,535
32	0 .. 4,294,967,296

Range of unsigned integers

- Let's add 19 and 23

$$\begin{array}{r} 1 111 \leftarrow \text{carry} \\ 00010011 \quad 19 \\ \underline{00010111} \quad \underline{23} \\ 00101010 \quad 42 \end{array}$$

Range of unsigned integers

- Given an 8-bit allocation, what happens when we add 250 and 8

$$\begin{array}{r} 11111010 \quad 250 \\ + 00010000 \quad 8 \\ \hline 0000010 \quad 2 \end{array}$$

- The 1 bit that carries out of the left end of the operation will be discarded. The answer we compute will be 2, which is $(250 + 8) \bmod 256$

Range of unsigned integers

- The previous problem arises when you try to store a number that is not within the range defined by the allocation.
- With an 8-bit allocation, the largest number that can be stored is 255; however, the result of the addition is 258.
- **Overflow** is the term used for the condition that results when there are insufficient bits to represent a number in binary.

Signed 8-bit arithmetic

- So far we have concentrated on positive numbers.
- There is no negative sign inside the computer; therefore, we have to devise a scheme for representing negative numbers.
- We will consider One's complement and two's complement.
- For simplicity, we will use an 8-bit representation.

Signed 8-bit arithmetic

- One's complement format of a number
 - Change the number to binary, ignoring the sign.
 - Add 0s to the left of the binary number to make a total of 8 bits
 - If the sign is positive, no more action is needed.
 - If the sign is negative, complement every bit (i.e. change from 0 to 1 or from 1 to 0)

Signed 8-bit arithmetic

- Write 25 in one's complement format

$$00011001 \quad 25 = (2^4 + 2^3 + 2^0)$$

- Write -25 in one's complement format

- Since the number is negative, complement each bit

$$11100110 \quad -25$$

Signed 8-bit arithmetic

- One's complement
 - Negation is easy.
 - Addition / subtraction is relatively easy...
 - Range: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
 - Drawback: Two values for 0
 - +0 00000000
 - 0 11111111

One's Complement to Decimal

- If the sign bit (the leftmost bit) is 0, convert from binary to decimal.
- If the sign bit is 1 (negative number)
 - complement the number
 - convert the number to decimal
 - put a negative sign in front of the number.

One's Complement to Decimal

- Convert the following 1's complement representation to decimal:
 - a) 11110001:
 - Since the sign bit is 1, complement the number: 00001110
 - Convert to decimal: $00001110_2 = 14_{10}$
 - Put a negative sign in front: -14
 - b) 00011010
 - Since the sign bit is 0, do not complement the number, just do the direct binary to decimal conversion.
 - $2^4 + 2^3 + 2^1 = 26$

Signed Arithmetic in 2's complement

- Most computers today use 2's complement representation for negative numbers.
- The 2's complement of a negative number is obtained by adding 1 to the 1's complement.

For -13:

00001101	base integer
11110010	1's complement
+1	
11110011	2's complement

Signed Arithmetic in 2's complement

- Write -25 in two's complement format.
- $+25 = 2^4 + 2^3 + 2^0 = 00011001$
- Formats for -25 are:
 - 1 1 1 0 0 1 1 0 one's complement
 - 1 1 1 0 0 1 1 1 two's complement

Signed Arithmetic in 2's complement

- To add two integers in two's complement, add two bits and propagate the carry to the next column. If there is a final carry after the leftmost column addition, discard it.

Add -25 and 20:

1 1 1 0 0 1 1 1 (-25)

0 0 0 1 0 1 0 0 (20)

1 1 1 1 1 0 1 1

Signed Arithmetic in 2's complement

- Since the negative of any number is its two's complement, the sum of a number and its two's complement is always 0
- The difference, $a - b$, is computed as $a + \text{twos_complement}(b)$ (i.e., flip bits and add 1)

Signed Arithmetic in 2's complement

- Add +12 and -12

$$+12 = 00001100_2$$

$$\underline{-12 = 1110100_2}$$

$$0 \quad 00000000_2$$

Summary: 2's complement

- Two's complement
 - Negation is easy
 - Addition / subtraction is easy
 - One value for zero.
 - Range: $-(2^{n-1})$ to $+(2^{n-1} - 1)$
 - Conversion:
 - If the sign bit is 0, convert the binary number to decimal.
 - If the sign bit is 1 subtract 1 from the binary number
 - complement each bit
 - convert the binary number to decimal
 - put a minus sign in front

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
 - can add 32 bit numbers, etc.

Starting Small

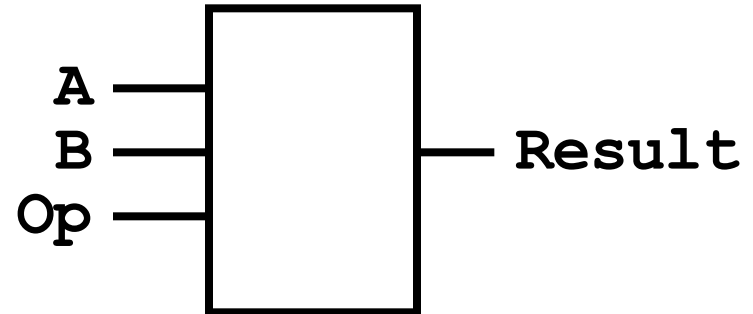
- We can start by designing a 1 bit ALU.
- Put a bunch of them together to make larger ALUs.
 - building a larger unit from a 1 bit unit is simple for some operations, can be tricky for others.
- Bottom-Up approach:
 - build small units of functionality and put them together to build larger units.

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



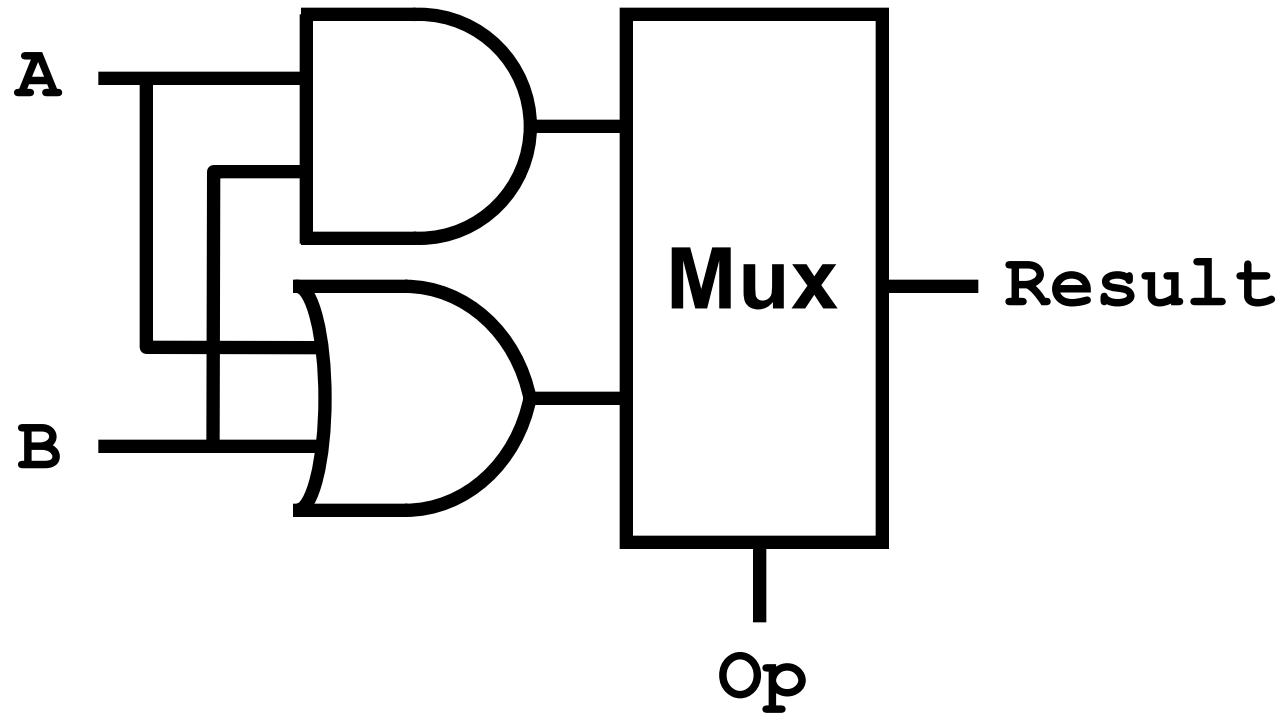
Op=0: Result is $A \cdot B$

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 *A painful reminder of the test*

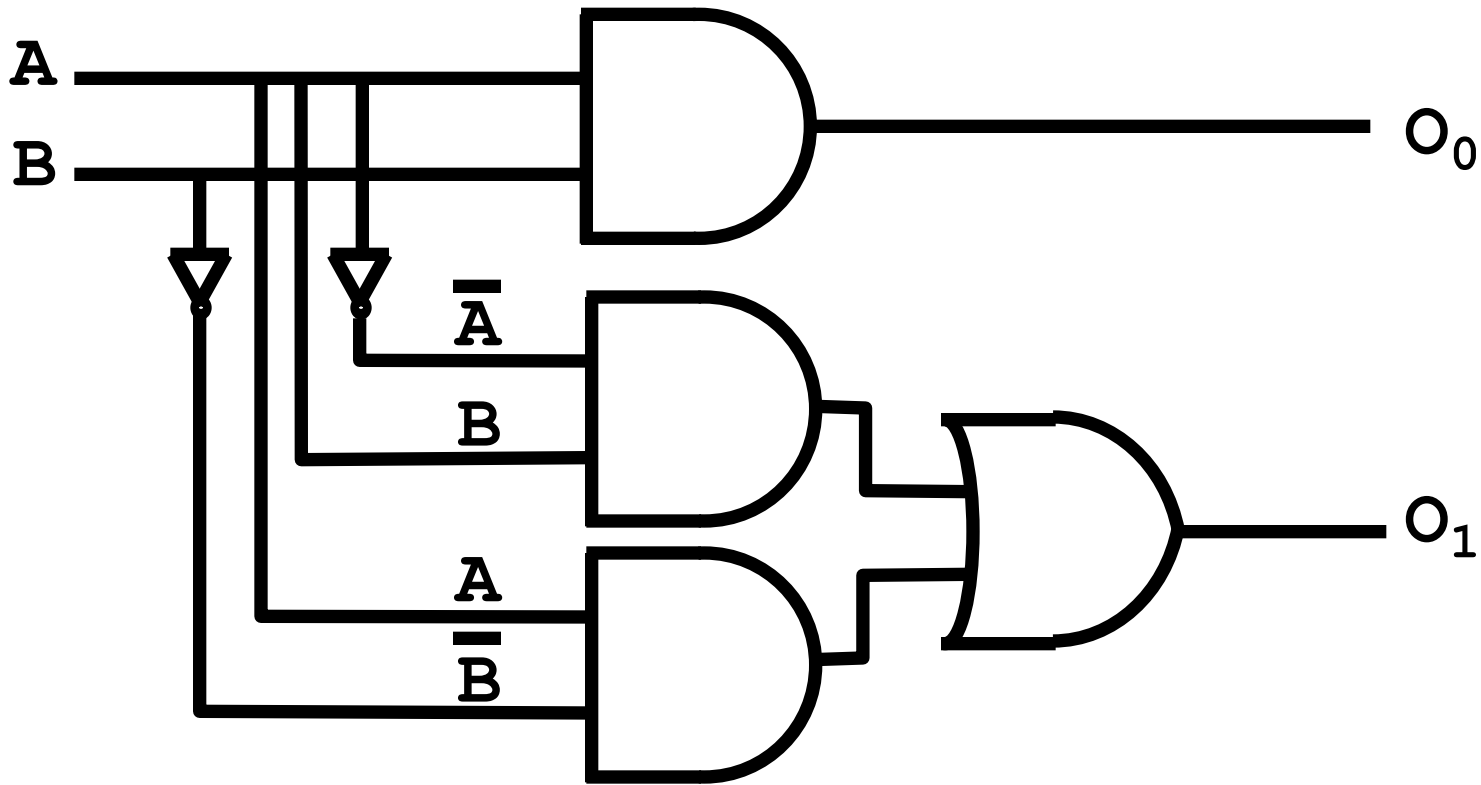
- We need to build a 1 bit *adder*
 - compute binary addition of 2 bits.
- We already know that the result is 2 bits.

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?

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

- We are adding 3 bits
 - new bit is the *carry* from the first column.
 - The output is still 2 bits, a *sum* and a *carry*

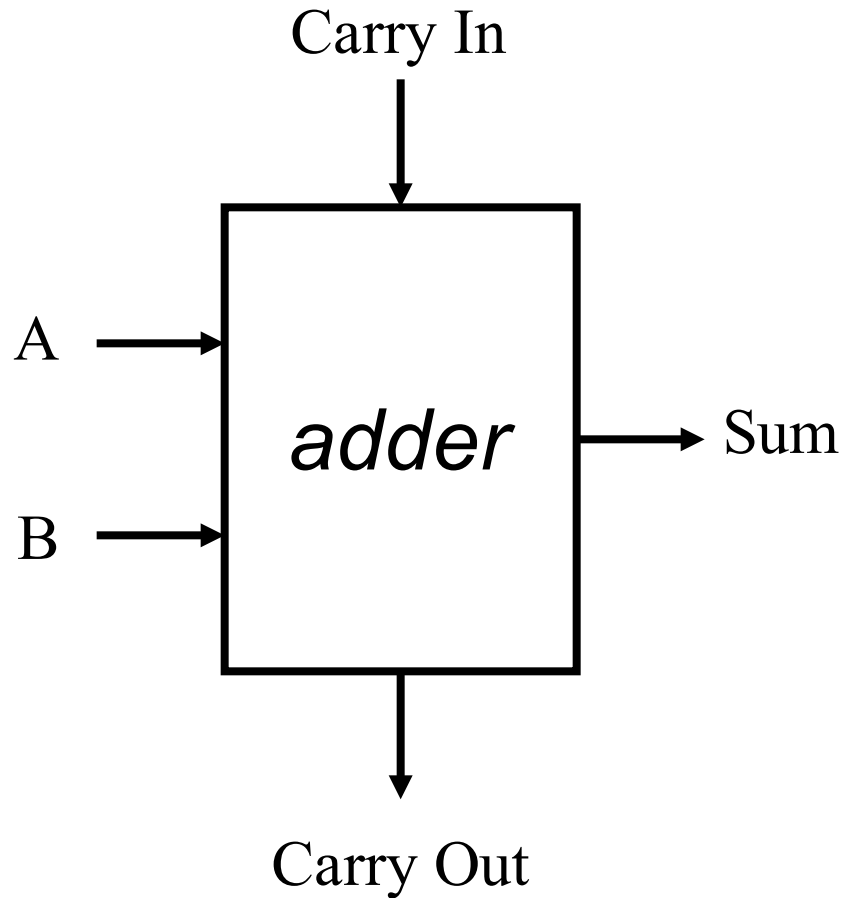
Revised Truth Table for Addition

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



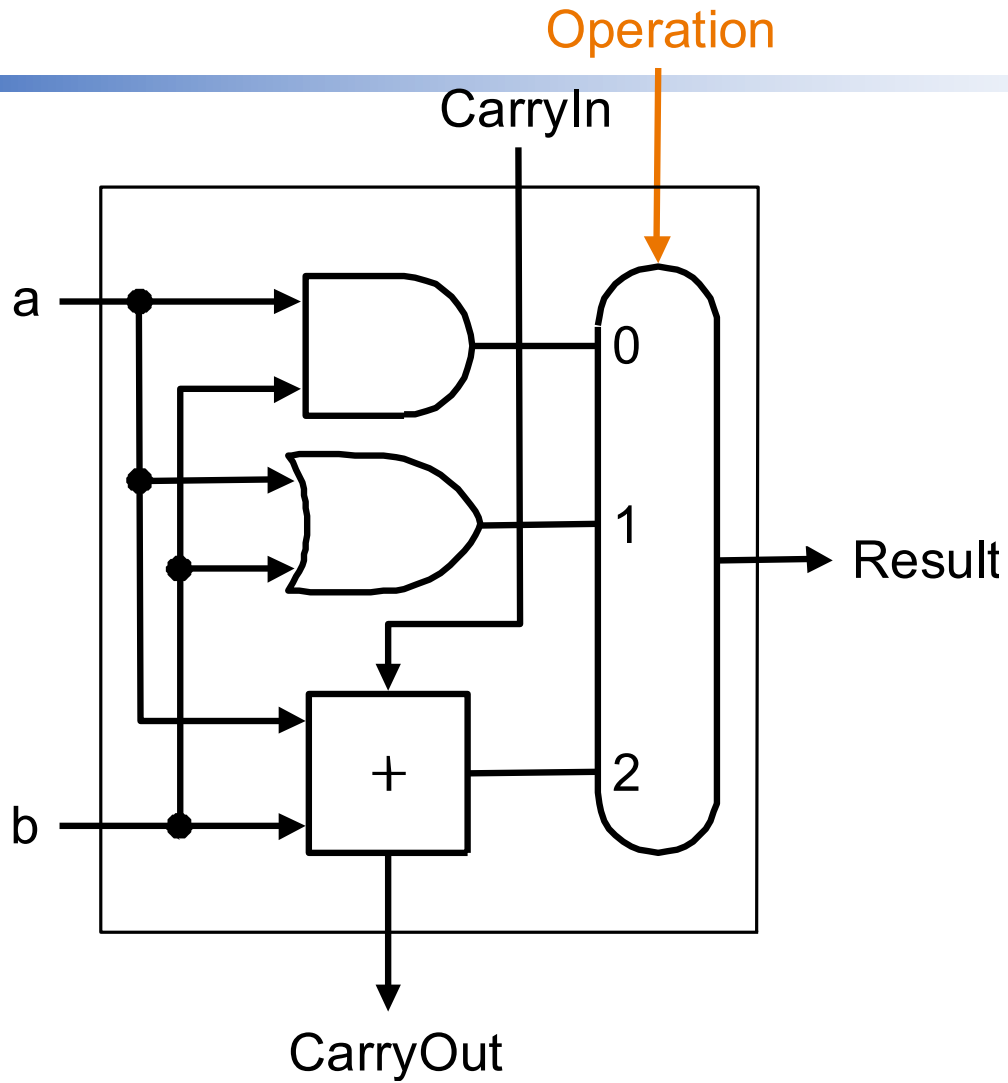
1 Bit ALU

- Combine the AND/OR with the adder.
- We must now use a 4-input MUX with 2 selection inputs.

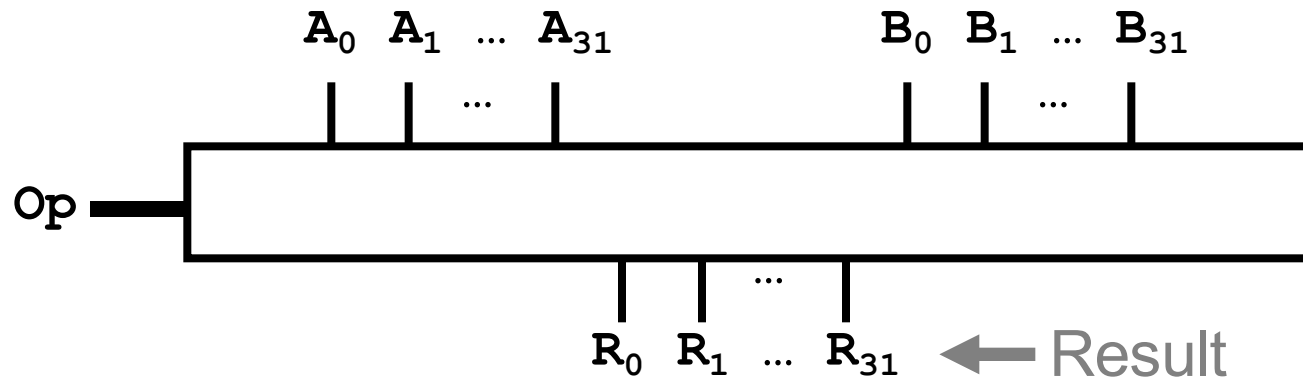
AND

OR

add

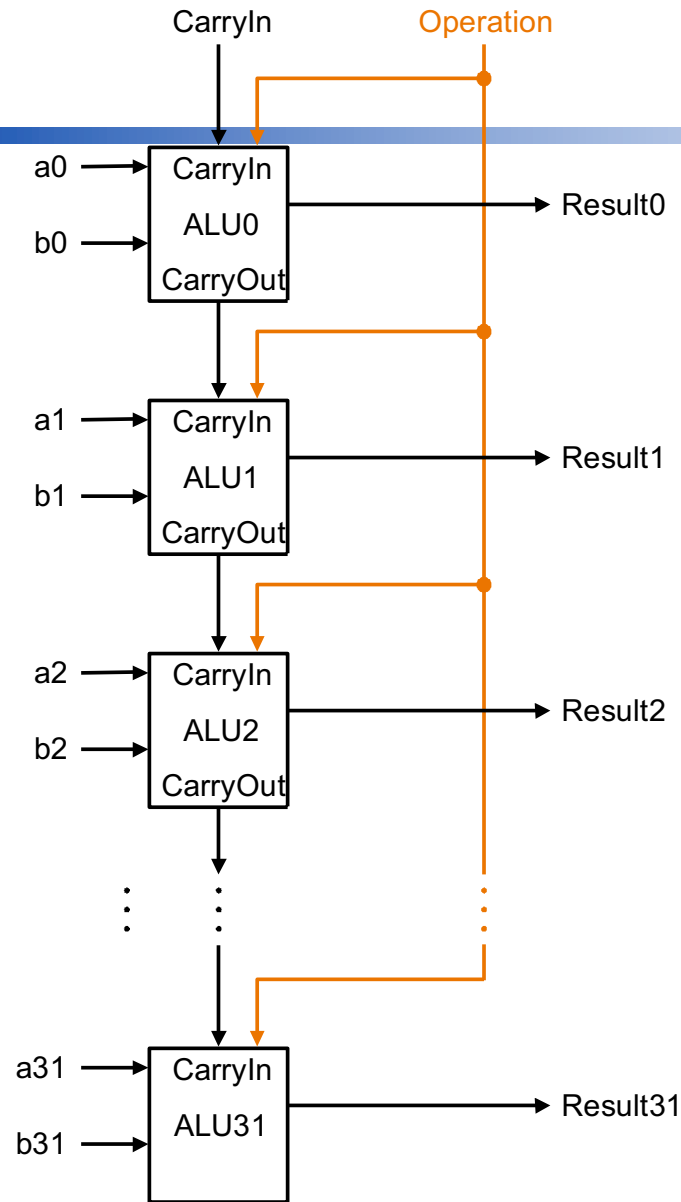


Building a 32 bit ALU



- 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.

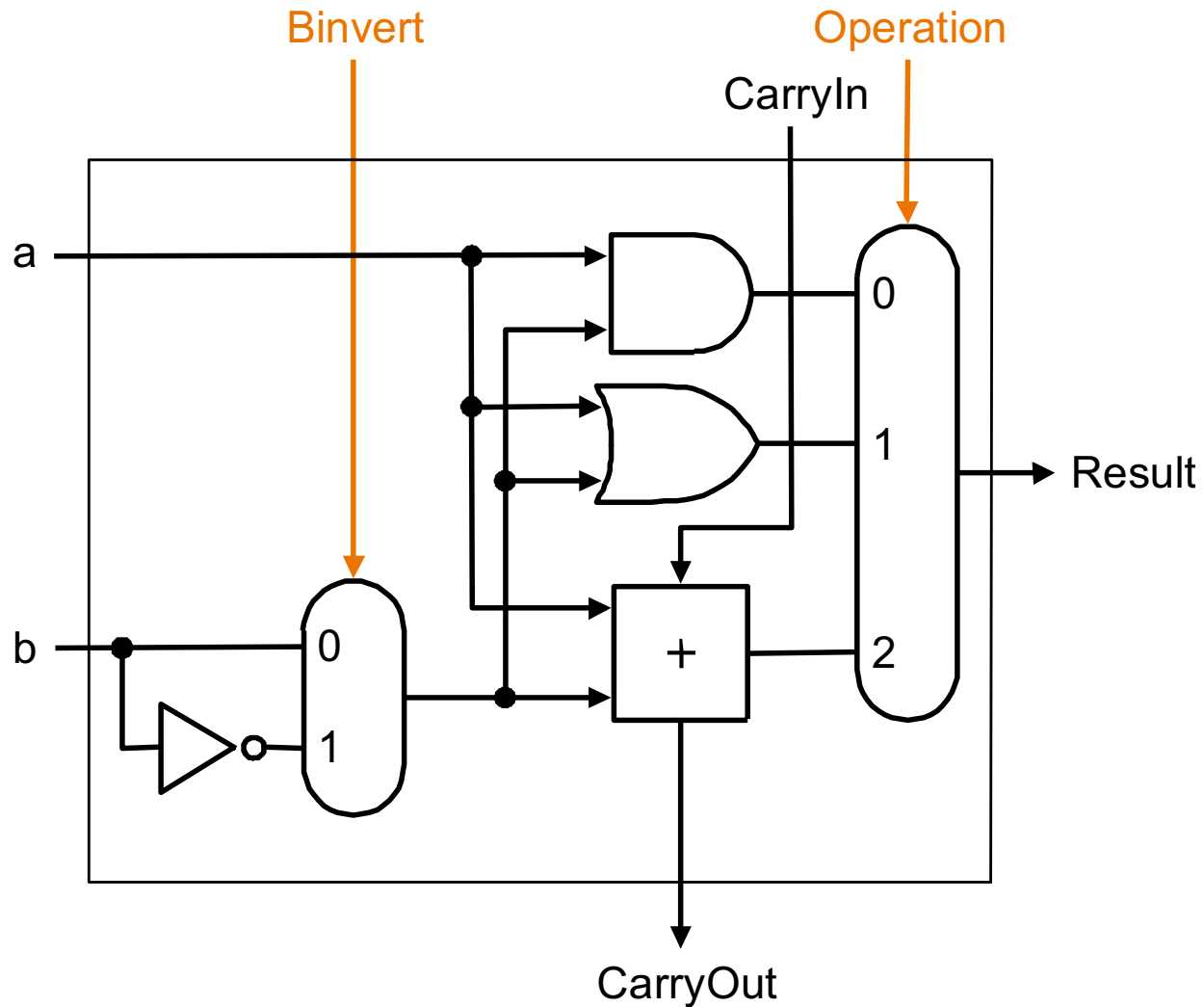
New Operation: Subtraction

- 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

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 2 numbers.

- comparison requires a subtraction.

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

True: output should be 0000000...001

False: output should be 0000000...000

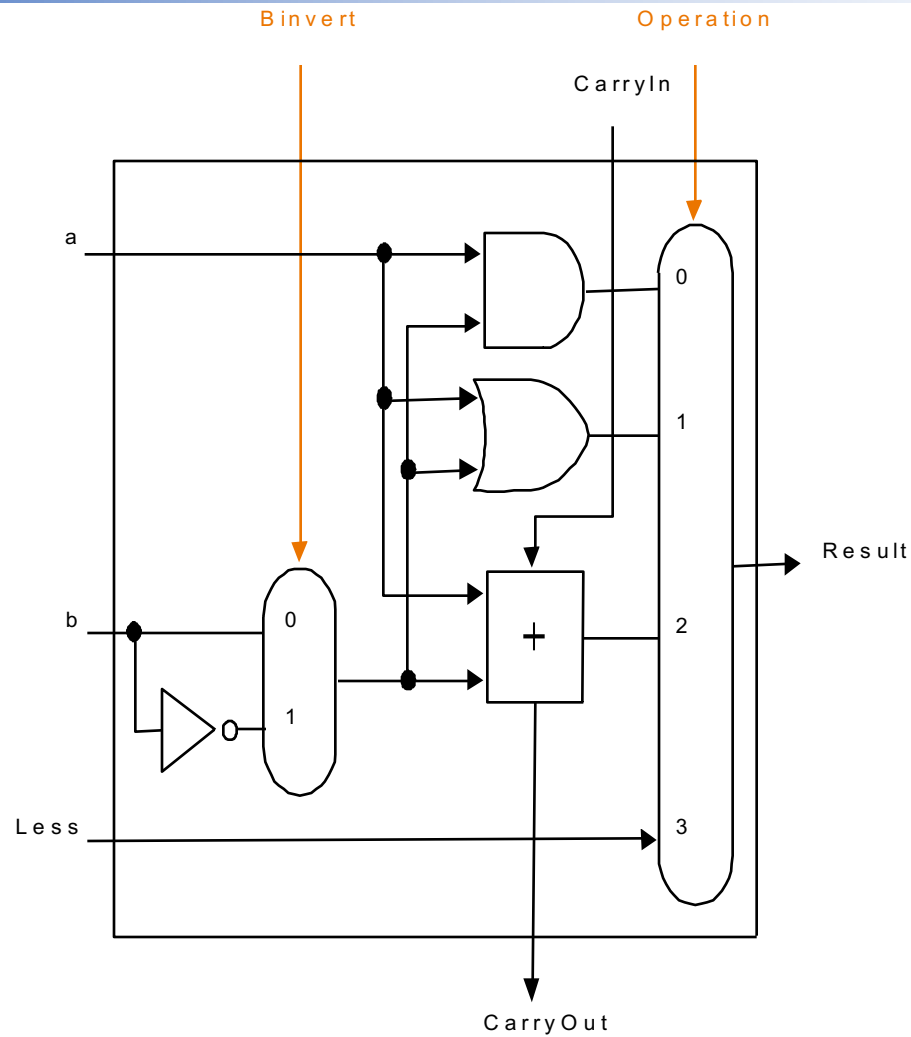
slt Strategy

- To compute `slt A B`:
 - subtract B from A (set `binvert` and the L.S. `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!

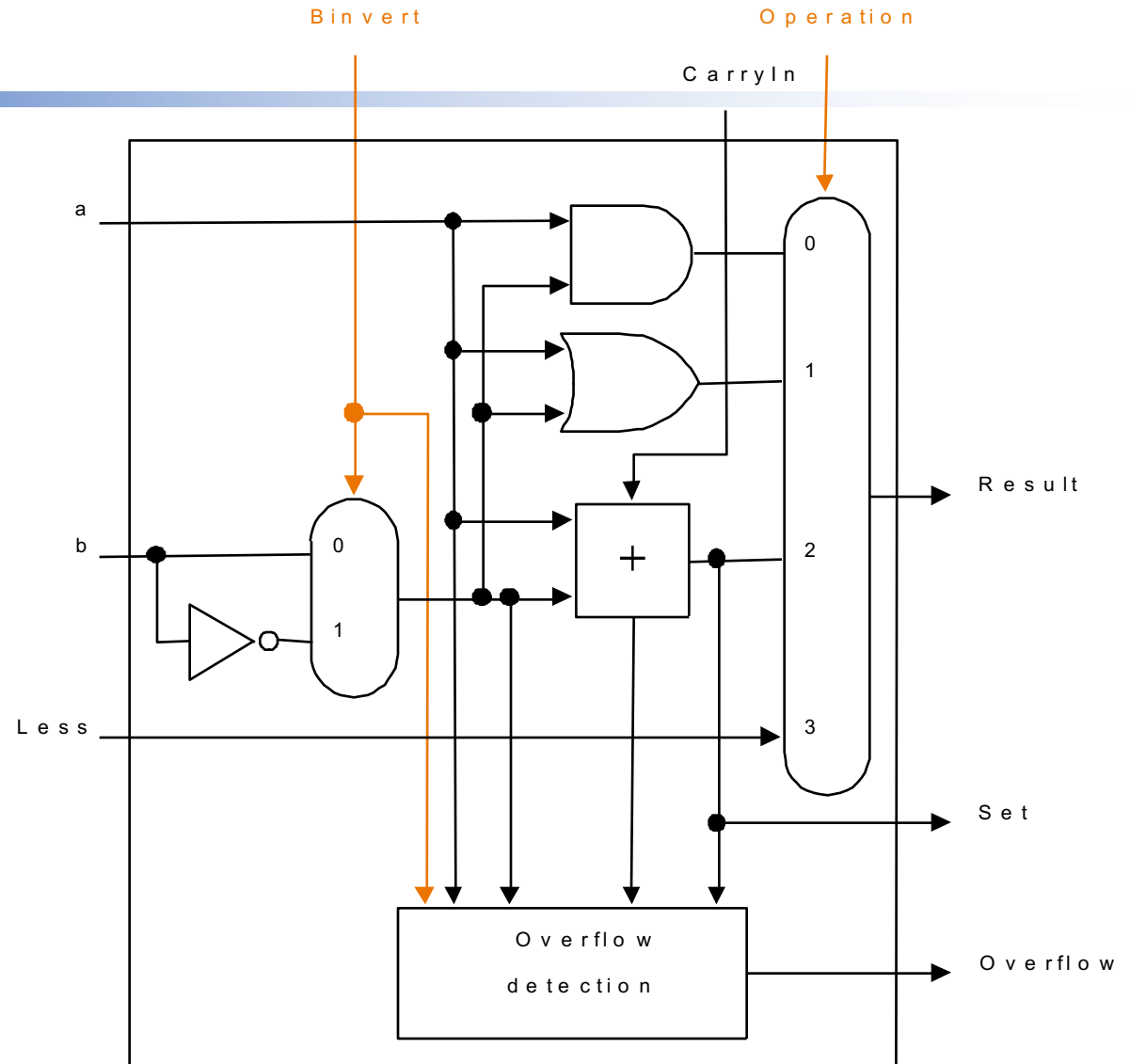
LS: Least significant (rightmost)

MS: Most significant (leftmost)

New 1-bit ALU

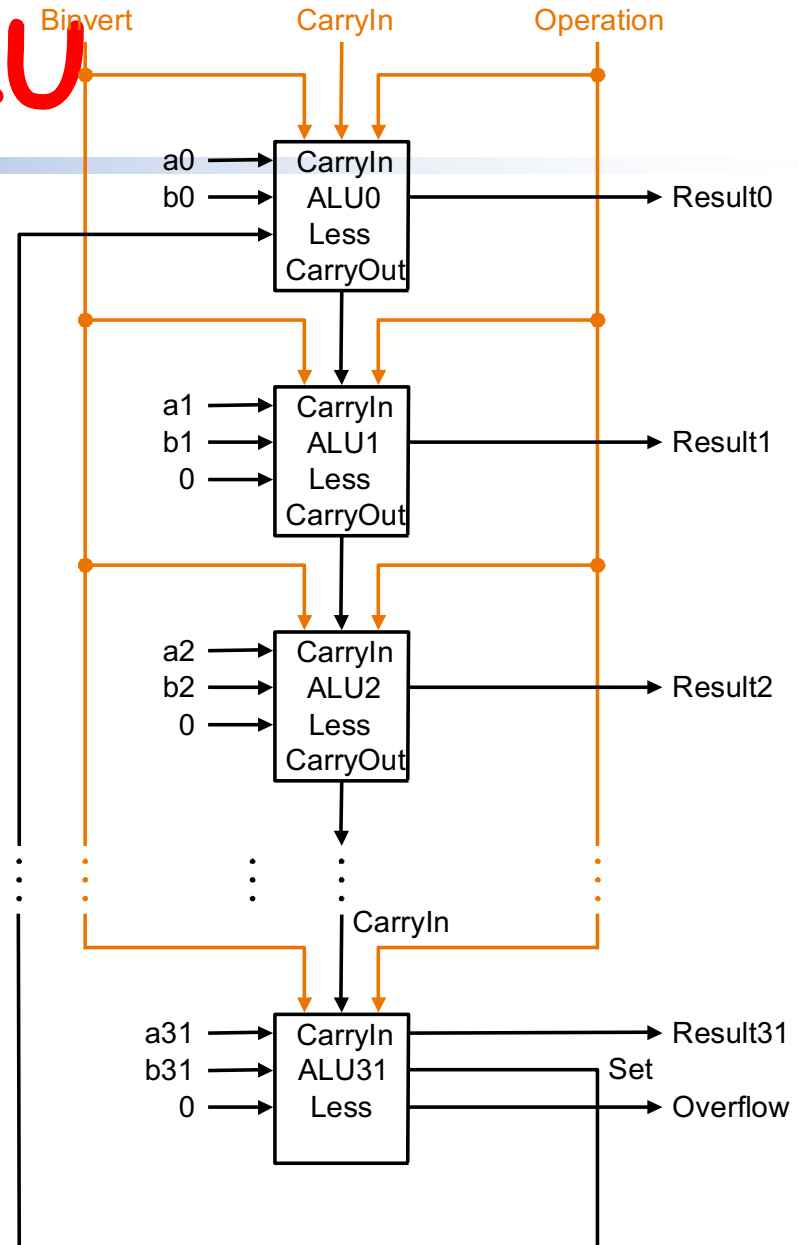


MSB ALU

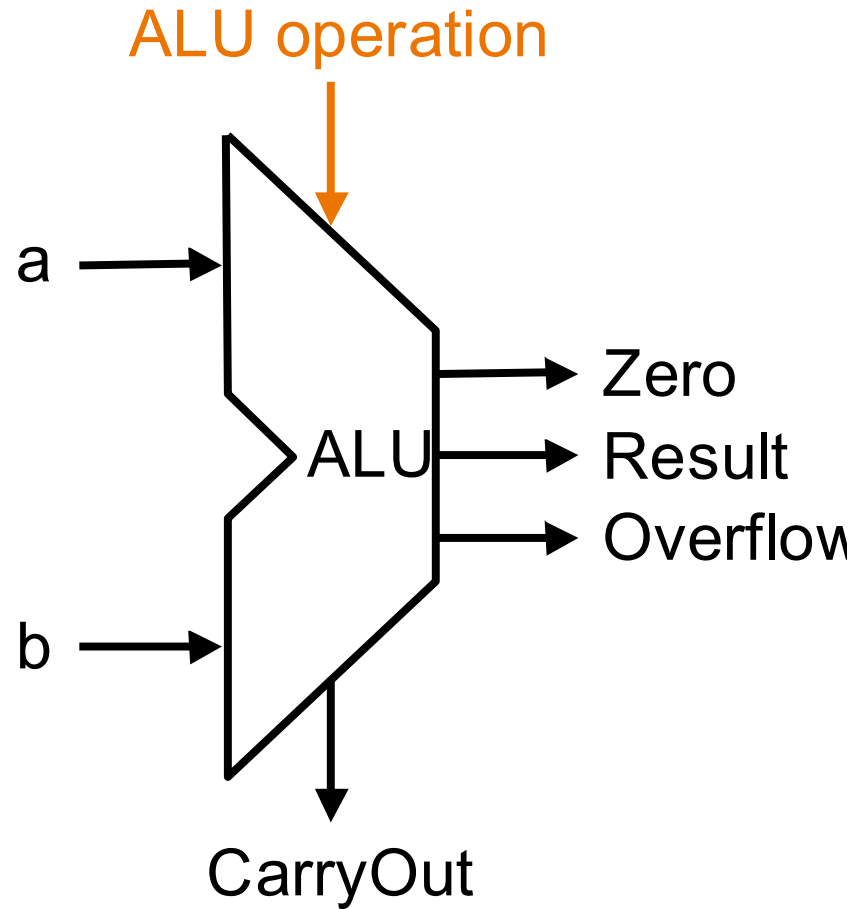


New 32-bit ALU

- **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



Speed is important.

- Using a *ripple carry adder* the time it takes to do an addition is too long.
 - each 1-bit ALU has something like 2 *levels* of gates.
 - The input to the i^{th} ALU includes an output from the $i-1^{\text{th}}$ ALU.
 - For 32 bits we have something like 64 gate delays before the addition is complete.

Strategies for speeding things up.

- We could derive the truth table for each of the 32 result bits as a function of 64 inputs.
- We know we can build SOP expressions for each and implement using 2 levels of gates.
- This might be a good test question!
 - don't worry, you would need so much paper I couldn't carry the tests to class...

A more realistic approach

- The problem is the *ripple*
 - The last carry-in is takes a long time to compute.
- We can try to compute the carry-in bits as fast as possible
 - this is called *carry lookahead*
 - It turns out we can easily compute the carry-in bits much faster (but not in constant time).

Carry In Analysis

- CarryIn_i is an input to the i^{th} 1 bit adder.
- CarryOut_{i-1} is connected to CarryIn_i
- We know about how to compute the CarryOuts

A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	0	1	0	0
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

Computing Carry Bits

- CarryIn_0 is an input to the adder.
 - we don't compute this - it's an input.
- CarryIn_1 depends on A_0 , B_0 and CarryIn_0 :

$$\text{CarryIn}_1 = (B_0 \cdot \text{CarryIn}_0) + (A_0 \cdot \text{CarryIn}_0) + (A_0 \cdot B_0)$$

↑
SOP: Requires 2 levels of gates

CarryIn₂

$$\text{CarryIn}_2 = (B_1 \cdot \text{CarryIn}_1) + (A_1 \cdot \text{CarryIn}_1) + (A_1 \cdot B_1)$$

We can substitute for CarryIn_1 and get this mess:

$$\begin{aligned} \text{CarryIn}_2 = & (B_1 \cdot B_0 \cdot \text{CarryIn}_0) + (B_1 \cdot A_0 \cdot \\ & \text{CarryIn}_0) + (B_1 \cdot A_0 \cdot B_0) + (A_1 \cdot B_0 \cdot \text{CarryIn}_0) + (A_1 \\ & \cdot A_0 \cdot \text{CarryIn}_0) + (A_1 \cdot A_0 \cdot B_0) + (A_1 \cdot B_1) \end{aligned}$$

The size of these expressions will get too big (that's the whole problem!).

Another way to describe CarryIn

$$\begin{aligned}C_{i+1} &= (B_i \cdot C_i) + (A_i \cdot C_i) + (A_i \cdot B_i) \\ &= (A_i \cdot B_i) + (A_i + B_i) \cdot C_i\end{aligned}$$

$A_i \cdot B_i$: Call this *Generate* (G_i)

$A_i + B_i$: Call this *Propagate* (P_i)

$$C_{i+1} = G_i + P_i \cdot C_i$$

Generate and Propagate

$$C_{i+1} = G_i + P_i \cdot C_i$$

$$G_i = A_i \cdot B_i$$

$$P_i = A_i + B_i$$

- When A_i and B_i are both 1, G_i becomes a 1.
 - a CarryOut is *generated*.
- If P_i is a 1, any Carry in is *propagated* to Carry Out.

Using G_i and P_i

$$C_1 = G_0 + P_0 \cdot C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 \cdot C_1 \\ &= G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0) \\ &= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \end{aligned}$$

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Implementation

- Expression still get too big to handle (for 32 bits).
- We can minimize the time needed to compute all the CarryIn bits for a 4 bit adder.
- Connect a bunch of 4 bit adders together and treat CarryIns to these adders in the same manner.

