# A Short Introduction to Digital Logic Design

Z. Jerry Shi
Department of Computer Science and Engineering
University of Connecticut

Based on materials from Computer Organization: the Hardware/Software Interface by D. Patterson and J. Hennessy

## Outline

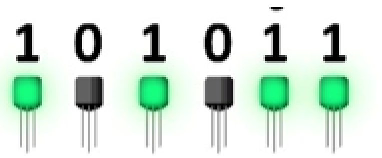
- Gates and truth table (Appendix A.2)
- Build a combinational circuit with gates (Appendix A.3)
  - Common combinational circuit modules
  - Decoder and multiplexor
- Basic arithmetic logic unit ALU (Appendix A.5)

| Operator | Digital logic design | Python (bitwise) | Python (logical) |
|---|---|---|---|
| AND | $X \cdot Y$ | X & Y | and |
| OR | $X + Y$ | X \| Y | or |
| NOT | $\overline{X}$ | ~X | not |
| XOR | $X \oplus Y$ | X ^ Y | |

## How is a bit handled in digital circuit?

- The value of a bit is represented by the states of circuit elements

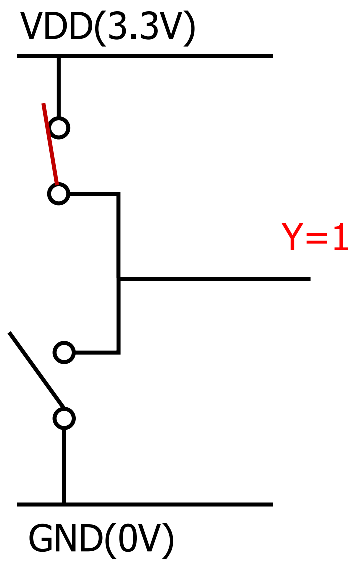
For example,

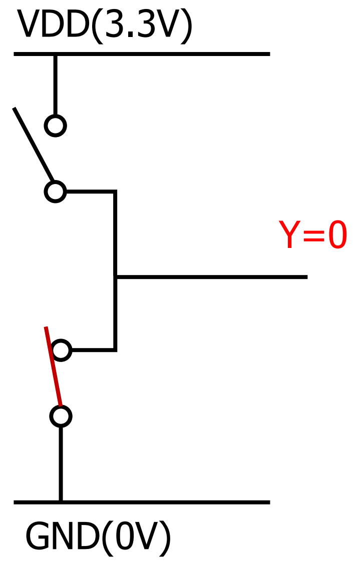


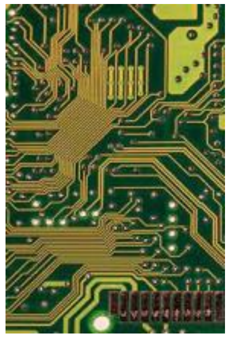
High voltage (e.g., 3.3V) indicates 1 and low voltage (e.g., 0V) indicates 0

Controlling the switches,
we can set a signal/bit to 0 or 1

We build gates to compute on bits

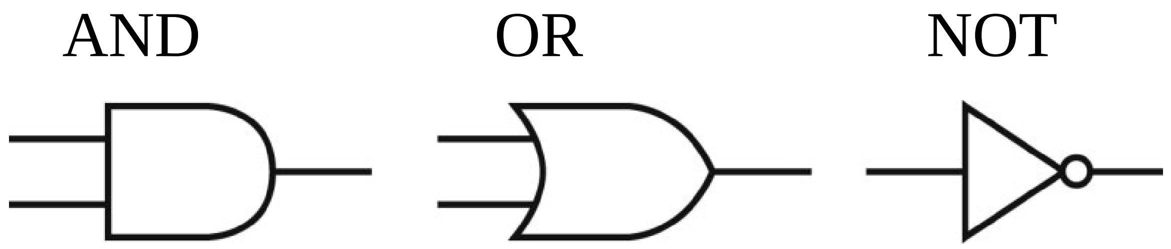






## Gates

- Gates are small circuit that computes on bits
  - A network of switches in each gate sets the output to 0 or 1



We can build any digital circuit with these three kinds of gates!

## Combinational versus sequential

Two types of circuit:

- Combinational circuit: the outputs depend on the current input values

- Sequential circuit: the outputs also depend on the history of inputs
  - Two identical sequential circuits may produce different outputs even if their current inputs are the same

## Truth table: NOT, AND, OR

- AND, OR, and NOT gates are combinational circuit
  - We can use truth tables to describe their functions
  - Pay attention to the new notation for NOT, AND, and OR operations

NOT: $\overline{X}$

| X | NOT X |
|---|---|
| 0 | 1 |
| 1 | 0 |

AND: $X \cdot Y$

| X | Y | X AND Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR: $X + Y$

| X | Y | X OR Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

## Logic expression and equations

- Logic function can be defined with logic equations

  Variable = logic expression

Logic expression is also called Boolean expression.

Literal: a variable or its complement, for example, $X, \overline{X}, \mathrm{Sel}$

Expression: literals combined by AND, OR, parentheses, and NOT

$$P \cdot \overline{Q} \cdot R \cdot (X + Y) \cdot \left(A + \overline{B \cdot C}\right) + \overline{A} \cdot D$$

Logic Equation:

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

## Boolean algebra

Logic expressions can be transformed with laws in Boolean algebra

| | | |
|---|---|---|
| Identity laws | $A + 0 = A$ | $A \cdot 1 = A$ |
| Zero and One laws | $A + 1 = 1$ | $A \cdot 0 = 0$ |
| Inverse laws | $A + \overline{A} = 1$ | $A \cdot \overline{A} = 0$ |
| Commutative laws | $A + B = B + A$ | $A \cdot B = B \cdot A$ |
| Associative laws | $A + ( B + C ) = (A + B) + C$ | $A \cdot ( B \cdot C ) = (A \cdot B ) \cdot C$ |
| Distributive laws | $A\cdot(B + C) = (A\cdot B) + (A\cdot C)$ | $A+(B\cdot C) = (A + B )\cdot(A+C)$ |
| DeMorgan's laws | $\overline{A + B} = \overline{A} \cdot \overline{B}$ | $\overline{A \cdot B} = \overline{A} + \overline{B}$ |

## Product term and sum term

Product term : A single literal or a product (AND) of two or more literals

$$A, \qquad \overline{B}, \qquad A \cdot B, \qquad A \cdot B \cdot C, \qquad D \cdot E \cdot \overline{F}$$

Sum term : A single literal or a logical sum (OR) of two or more literals

$$A, \qquad \overline{B}, \qquad A + B, \qquad X + Y + \overline{Z}$$

## Assignments

- 0 or 1 can be assigned to a variable

Only one assignment makes a product term evaluated to 1

For each product term, find the assignment that makes it equal to 1

$$A \cdot B \qquad A \cdot \overline{B} \cdot \overline{C} \qquad D \cdot E \cdot \overline{F} \cdot \overline{G}$$

## Question

What is the assignment that makes the following product term to be 1?

Write three bits, starting from A.

$$A \cdot B \cdot \overline{C}$$

## Sum of product

Sum-of-product : A logical sum of product terms

$$A \cdot B \cdot C + C \cdot D + E$$

All logic expressions can be represented as a sum of product

If we have a logic expression, we can transform it into a sum of product

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

$$\overline{X + Y} = \overline{X} \cdot \overline{Y}$$

## Write logic equation from truth table

We can write a logic equation from a truth table

1. Write a product term for each row where the function outputs 1
2. Write a sum of products by ORing all the product terms

Example: Write the logic equation for function G.

$X \oplus Y$

| X | Y | G |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$\overline{X} \cdot Y$

$X \cdot \overline{Y}$

Step 1

$$G = X \cdot \overline{Y} + \overline{X} \cdot Y$$

Step 2

## Example

- Write the logic equation for function F.

| | X | Y | Z | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | 0 |
| 7 | 1 | 1 | 1 | 1 |

We can number the rows with the binary number formed by bits X, Y, and Z

## Answer

- Write the logic equation for function F.

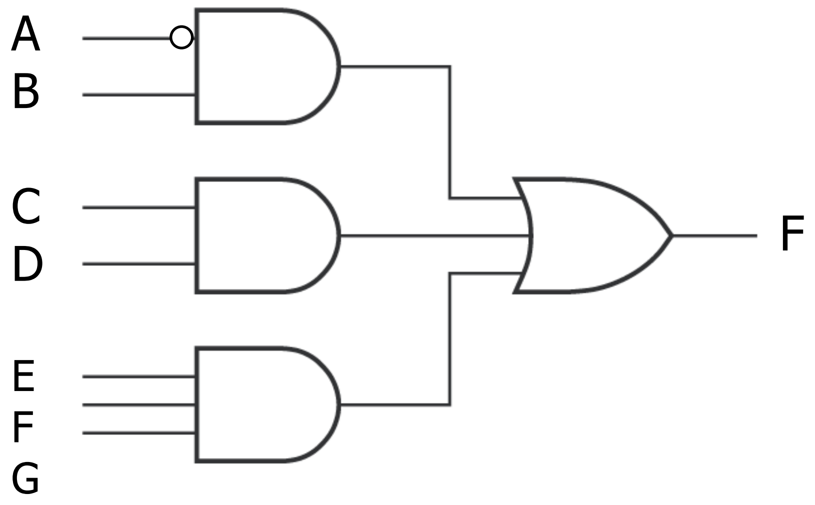
$$F = \overline{X} \cdot \overline{Y} \cdot Z + \overline{X} \cdot Y \cdot \overline{Z} + X \cdot \overline{Y} \cdot \overline{Z} + X \cdot Y \cdot Z$$

| | X | Y | Z | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 0 |
| 7 | 1 | 1 | 1 | 1 |

## Implementation of Logic Circuit

- A sum of product can be implemented with two-level circuit.

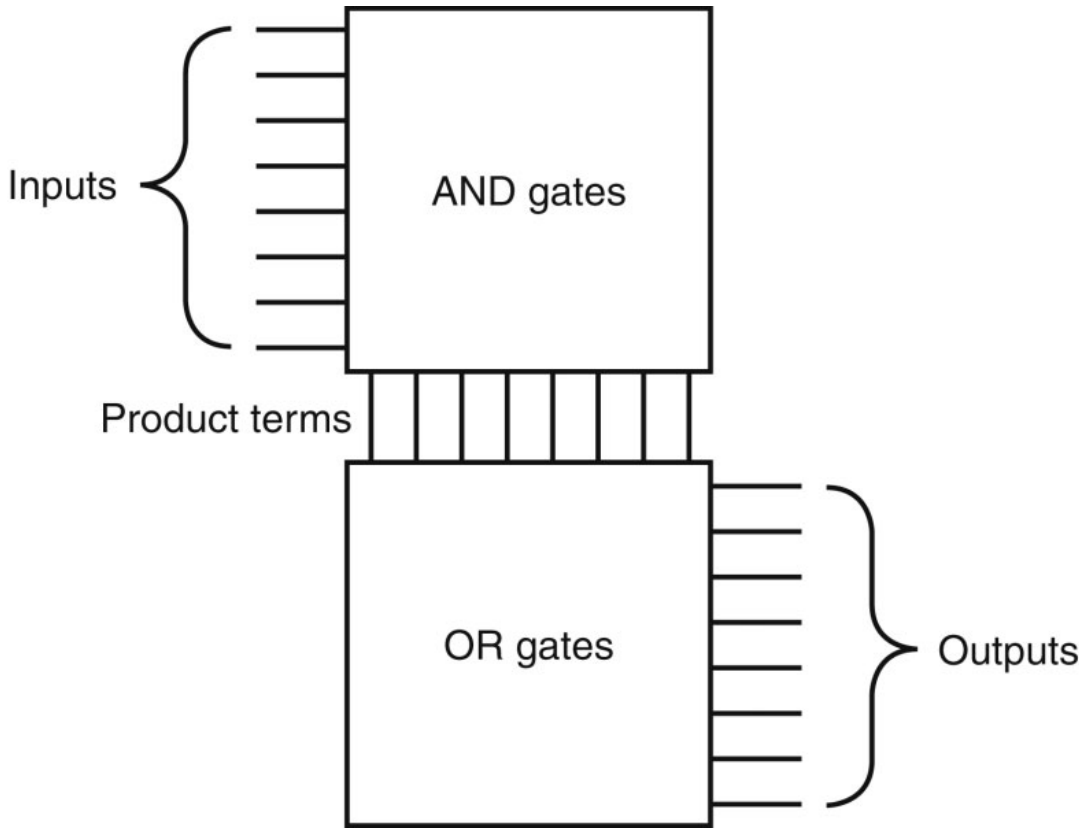
$$F = \overline{A} \cdot B + C \cdot D + E \cdot F \cdot G$$



Both $A$ and $\overline{A}$ are literals. This is still a two-level circuit.
NOT is only needed at input

## Two-level circuit

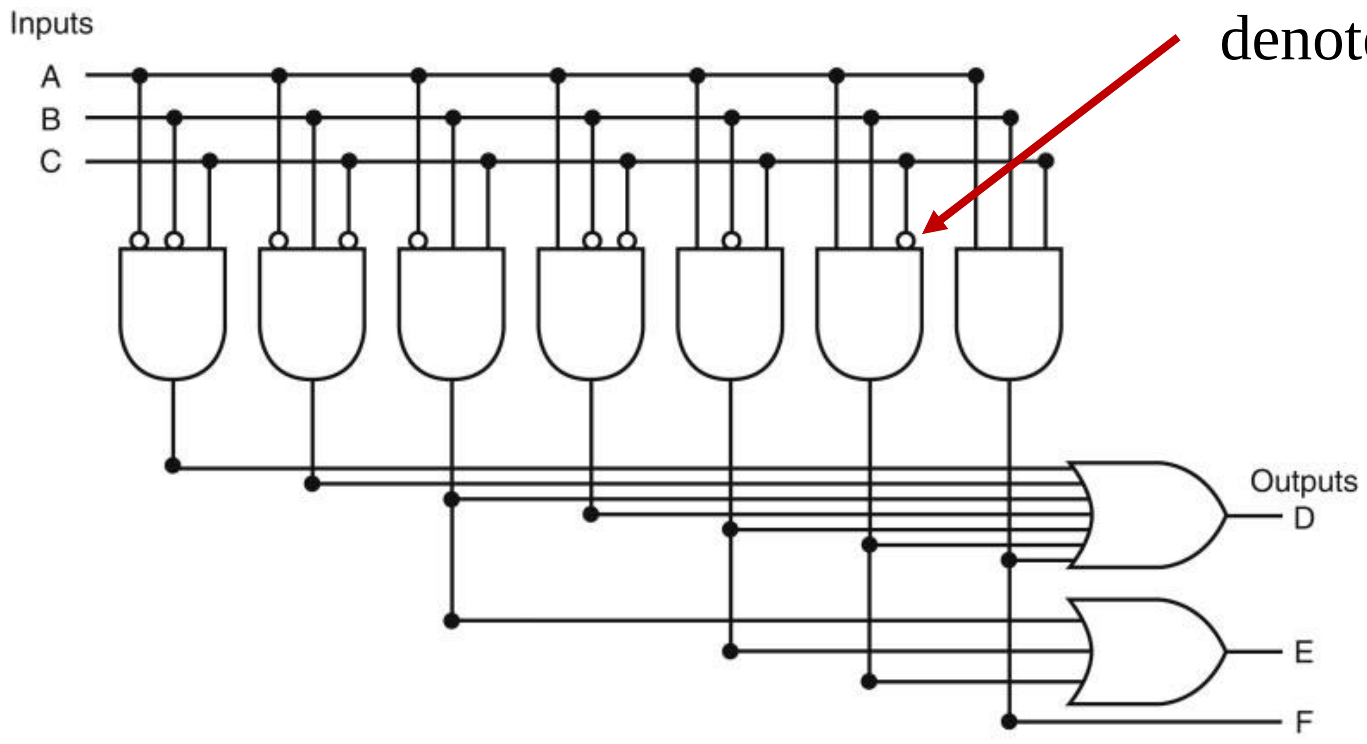
- All the combinational circuit can be implemented as two-level AND and OR gates
  - Direct implementation of sum of product



## Example of two-level circuit

- Can you write the logic expressions for D, E, and F?

A bubble denotes NOT



## X in Truth Table

- X at output: Don’t care
  - Designer can set the output to 0 or 1. Either is correct.
  - Designer picks 0 or 1, e.g., for simpler implementation
- X at input: The signal can be 0 or 1. It does not affect the output

| A | B | C | F | G |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | X |
| 1 | 0 | 1 | 0 | X |
| 1 | 1 | X | 1 | X |

G can be either 0 or 1

C can be either 0 or 1.
ABC = 110 and 111 have the same output

## Example: Implementation 1

Set don't cares at output to 001.

| A | B | C | G |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | X | 1 |

$A \cdot B$

Since C does not affect the output in this row, we do not include C in the product term

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

## Example: Implementation 2

Set don't cares at output to 000.

| A | B | C | G |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | X | 0 |

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

## Example: Implementation 3

Another way to set don't cares

Advantage:
G does not depend on A

| A | B | C | G |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$$G = \overline{B} \cdot \overline{C} + B \cdot C$$

All three implementations meet the spec, but they are different.

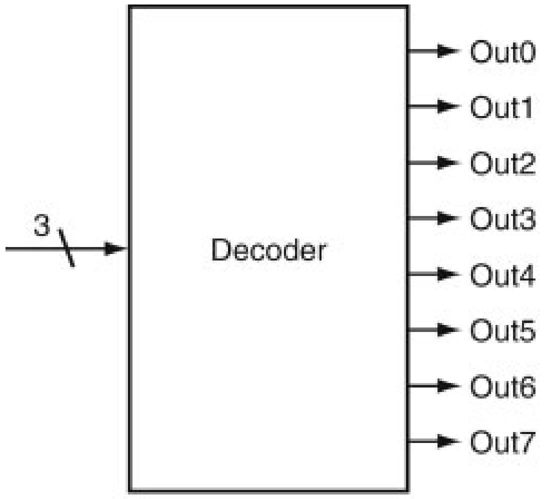
## Decoder

- *n*-bit input, $2^n$-bit output
- One and only one output specified by input is asserted (i.e., 1)

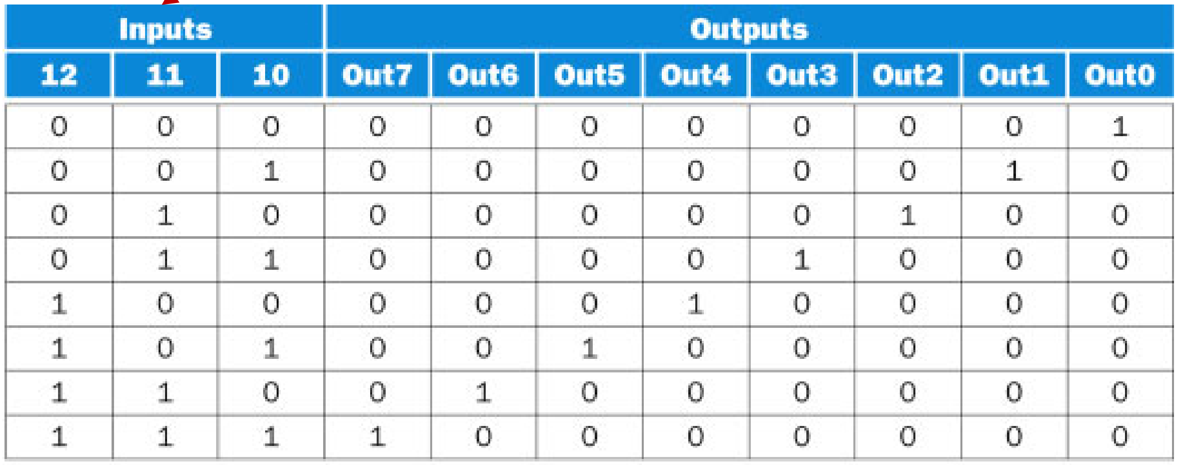
Example: A 3 to 8 decoder.

Write a logic equation for Out6.

Three bits in the input: I2, I1, and I0



a. A 3-bit decoder



| Inputs | | | Outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| I2 | I1 | I0 | Out7 | Out6 | Out5 | Out4 | Out3 | Out2 | Out1 | Out0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

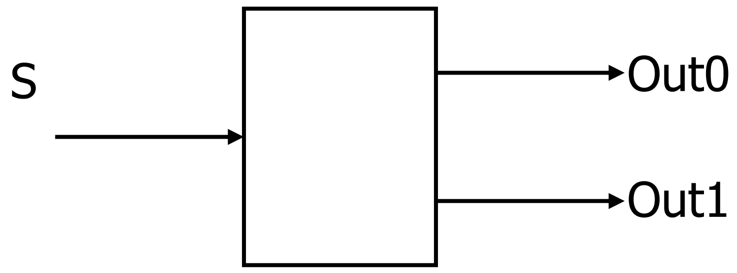
b. The truth table for a 3-bit decoder

## Example

- Design a 1-2 decoder
  - S activates one of the output signals: Out1 or Out0

- Write the logic equation for Out1 and Out0

| S | Out1 | Out0 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |



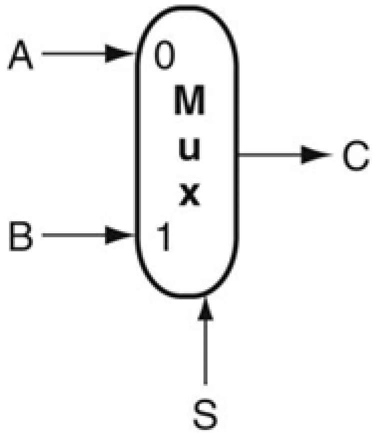
## Multiplexor (MUX)

- Select one out of multiple data sources

Example: 2-1 multiplexor

Use 1 bit to select one out of two input. (if-then-else)

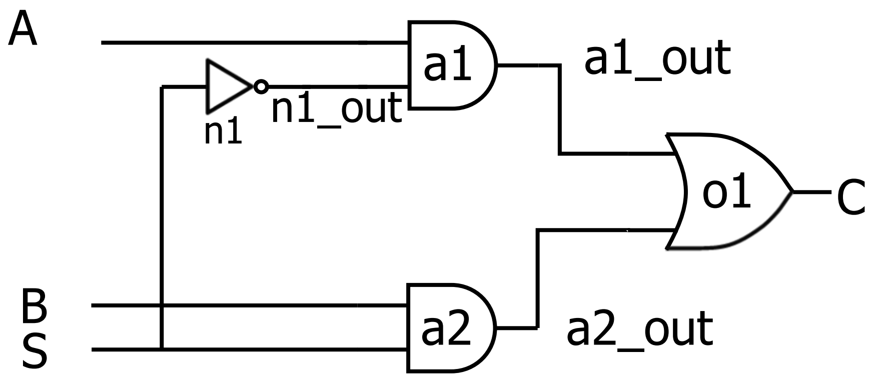
```
if s == 0:
        C = A
else:
        C = B
```



How can we implement the MUX with gates?

## Example: implementation of 2-1 MUX

Logic expression: $C = \overline{S} \cdot A + S \cdot B$



Idea:

Use AND gates as switch

A 1-to-2 decoder decides which switch is on

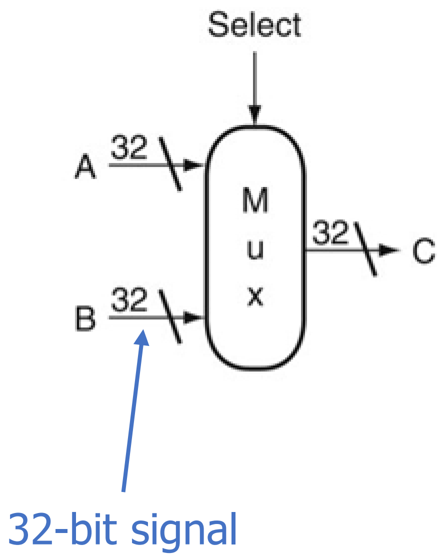
OR gate combines the output of AND gates

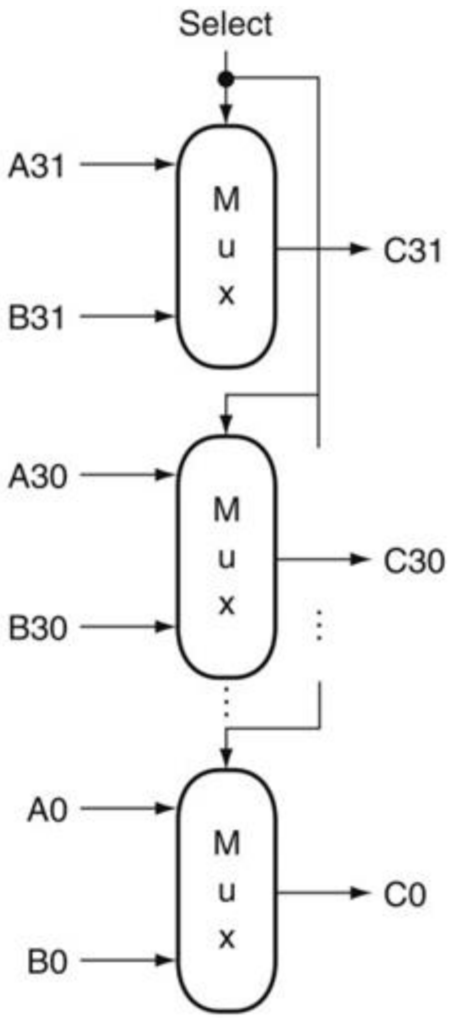
This is a 1-bit 2-1 Mux.
How do we design a 32-bit Mux?

MyHDL implementation:
[cse3666/mux.py at master · zhijieshi/cse3666 (github.com)](cse3666/mux.py at master · zhijieshi/cse3666 (github.com))

## 32-bit 2-1 Multiplexor

- Just make 32 copies and get an array of 32 1-bit MUX
- All controlled by the same select signal

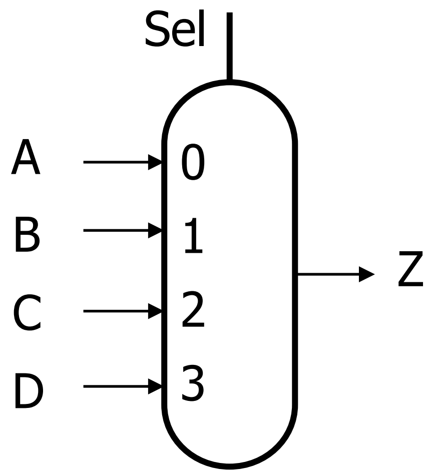




## Example: 4-1 multiplexor

Use 2-bit Sel to select one out of four. (if-then-else)

```
# MyHDL
@always_comb
def mux_logic():
    if sel == 0:
        z.next = a
    elif sel == 1:
        z.next = b
    elif sel == 2:
        z.next = c
    else:
        z.next = d
```



How can we implement it with basic gates?

## Example: 4-1 multiplexor

Assume S1 and S0 are bits 1 and 0 in Sel.

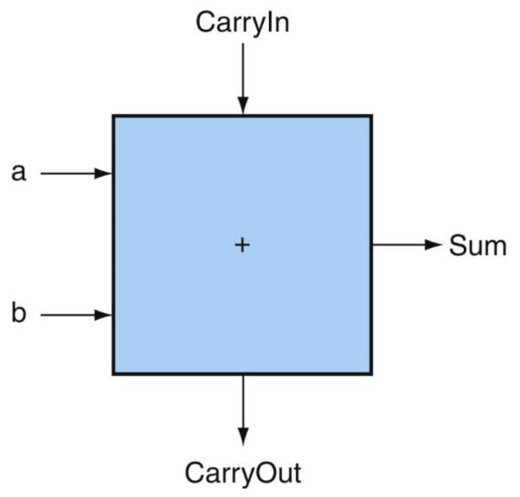
Let us use E0, E1, E2 and E3 to indicate which branch is enabled.

$$E0 = \overline{S1} \cdot \overline{S0} \quad E1 = \overline{S1} \cdot S0 \quad E1 = S0 \cdot \overline{S0} \quad E2 = S1 \cdot S0$$

$$\begin{aligned} Z &= E0 \cdot A + E1 \cdot B + E2 \cdot C + E3 \cdot D \\ &= \overline{S1} \cdot \overline{S0} \cdot A + \overline{S1} \cdot S0 \cdot B + S1 \cdot \overline{S0} \cdot C + S1 \cdot S0 \cdot D \end{aligned}$$

```
# MyHDL
z.next = ((~S1 & ~S0 & A) | (~S1 & S0 & B) |
          ( S1 & ~S0 & C) | ( S1 & S0 & D)) & 1
# "& 1" Keeps only the lowest bit
# because ~1 is not 0
```

## 1-bit full adder



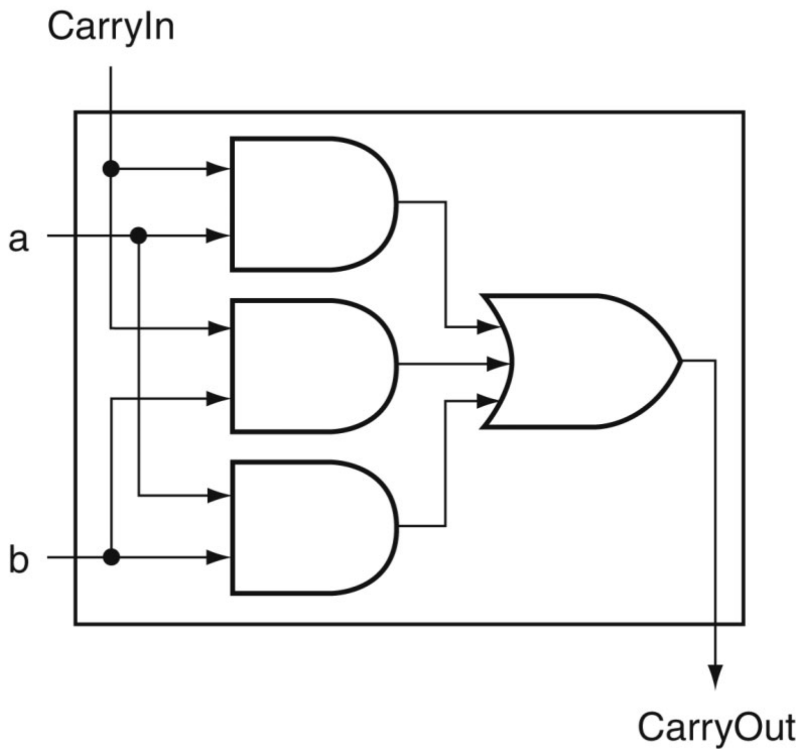
| Inputs | | | Outputs | | Comments |
|---|---|---|---|---|---|
| a | b | CarryIn | CarryOut | Sum | |
| 0 | 0 | 0 | 0 | 0 | $0 + 0 + 0 = 00_{two}$ |
| 0 | 0 | 1 | 0 | 1 | $0 + 0 + 1 = 01_{two}$ |
| 0 | 1 | 0 | 0 | 1 | $0 + 1 + 0 = 01_{two}$ |
| 0 | 1 | 1 | 1 | 0 | $0 + 1 + 1 = 10_{two}$ |
| 1 | 0 |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

## Generating CarryOut

CarryOut

$$= a \cdot b \cdot \overline{\text{CarryIn}} + a \cdot \overline{b} \cdot \text{CarryIn} + \overline{a} \cdot b \cdot \text{CarryIn} + a \cdot b \cdot \text{CarryIn}$$

$$= a \cdot b + a \cdot \text{CarryIn} + b \cdot \text{CarryIn}$$



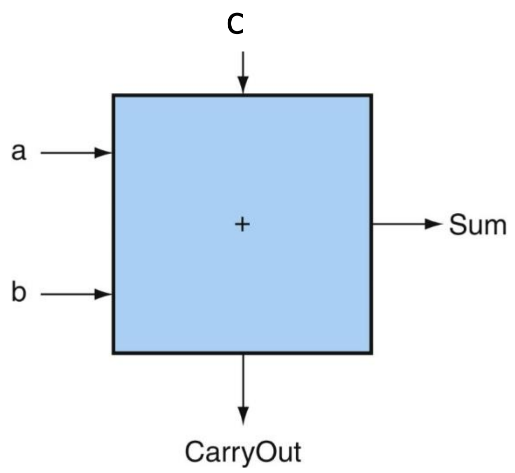
## Calculating sum

- From the truth table, we can write an equation for sum, using AND, OR, and NOT

$$\text{Sum} = \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot b \cdot c$$

- Or, we can use XOR gates:

$$\text{Sum} = a \oplus b \oplus c$$



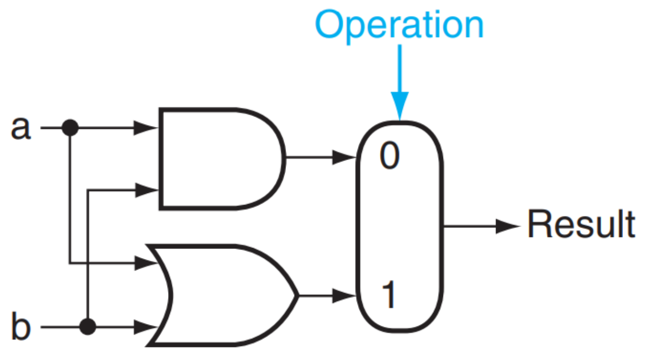
In the earlier example, the equation for F is:

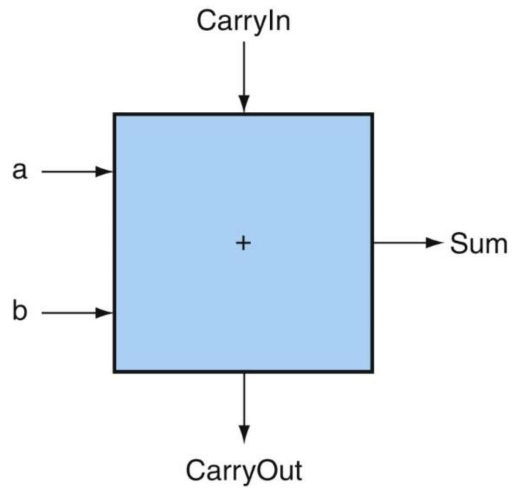
$$F = \bar{X} \cdot \bar{Y} \cdot Z + \bar{X} \cdot Y \cdot \bar{Z} + X \cdot \bar{Y} \cdot \bar{Z} + X \cdot Y \cdot Z$$

## Arithmetic and Logic Unit (ALU)

- ALU is the circuit that performs arithmetic and logic operations

- We now have an adder
- We can build a logic unit that supports AND and OR

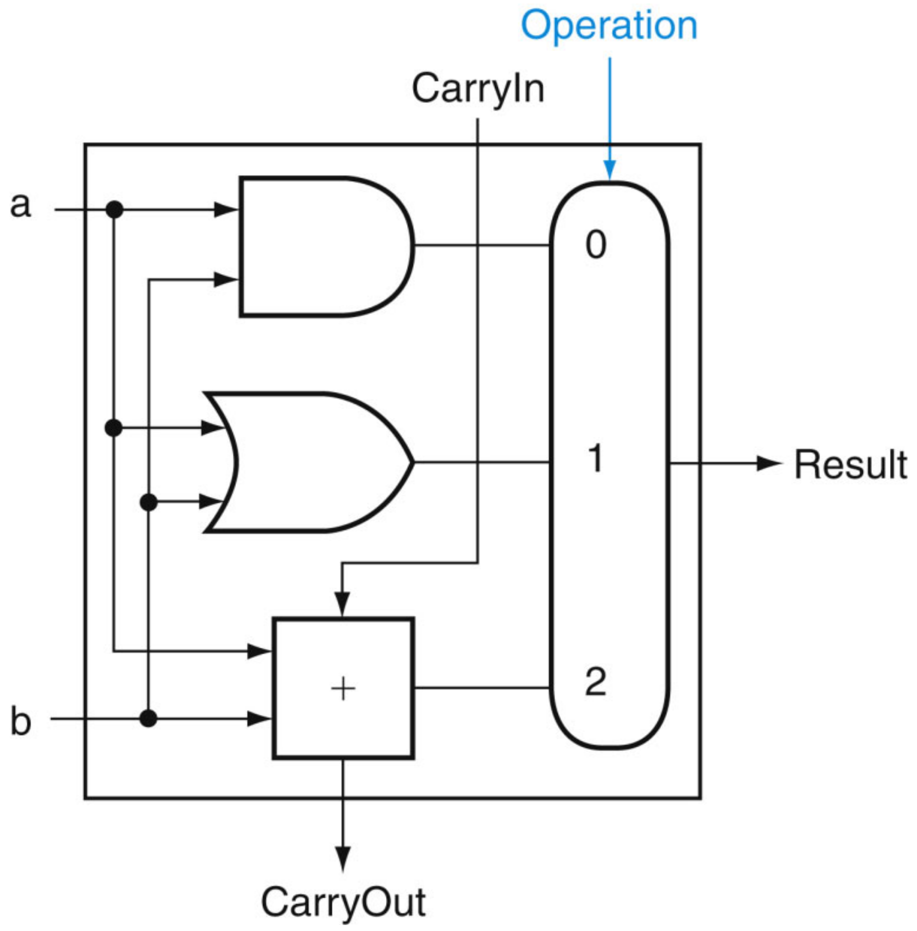




The logic unit supports AND and OR. A 2-1 MUX selects either AND or OR result.
How do we put them together?

## 1-bit ALU

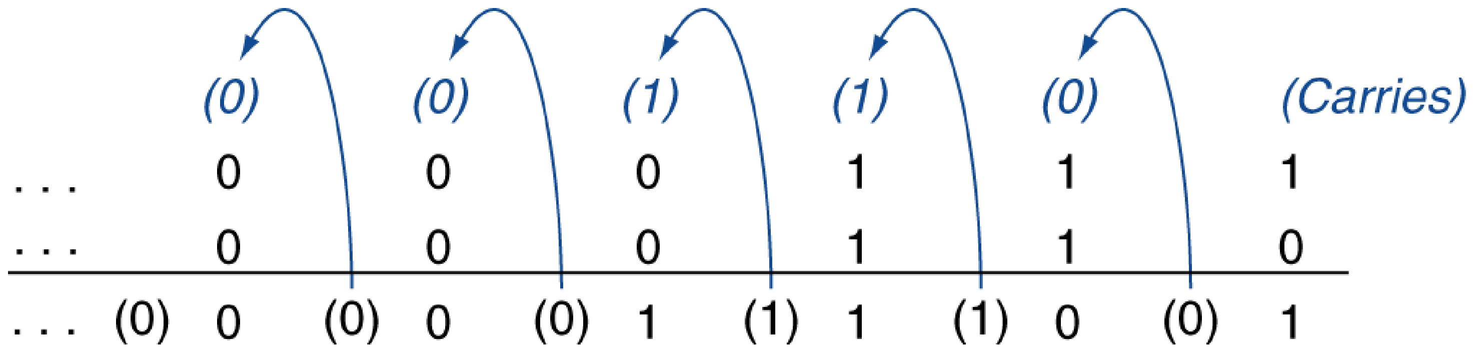
- Here is a 1-bit ALU that can perform AND, OR, and addition
  - All three results are generated, and only of them is selected

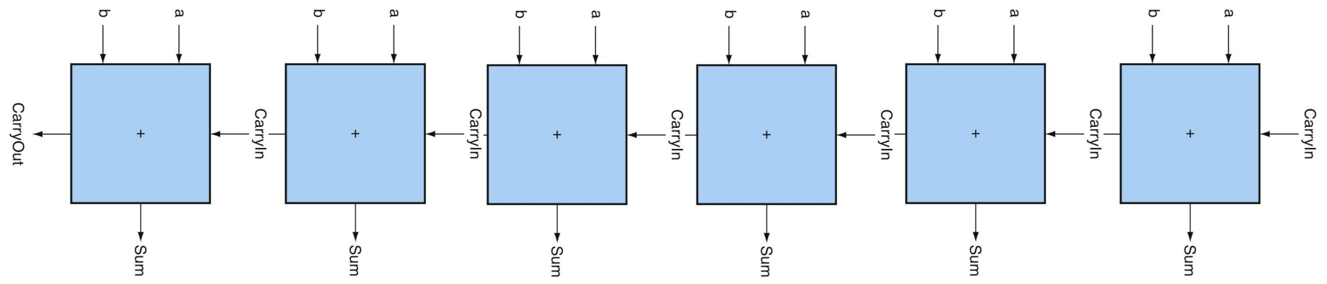


How do we build a 32-bit ALU?

# Integer Addition Example

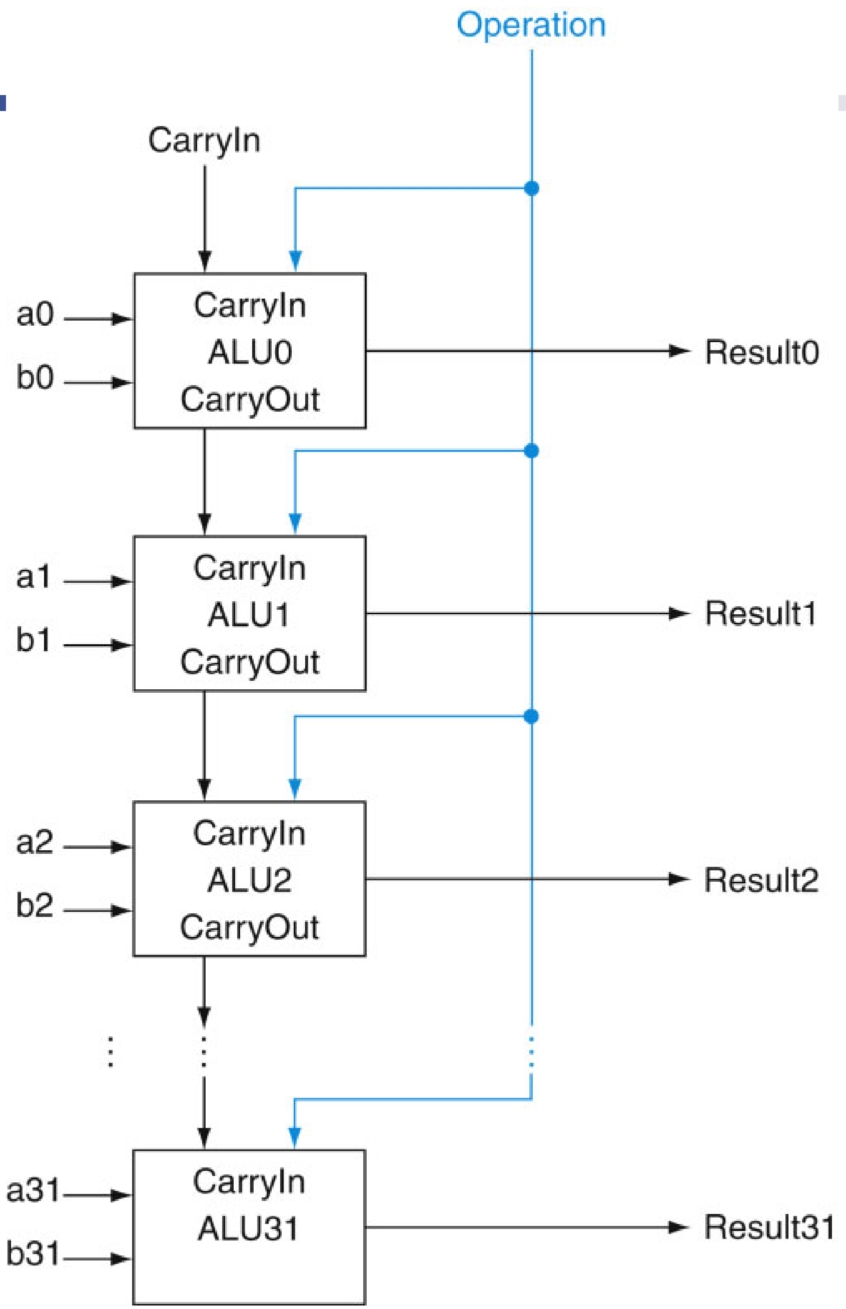
Example: 7 + 6





## 32-bit ALU

- All 1-bit ALUs performs the same operation, specified by the same Operation signal
- AND and OR are supported naturally
- Carry is chained for addition
  - ALU0 is the LSB



## How do we do subtraction?

## Integer Subtraction Example

To perform subtraction, we add the negation of the second operand

$$A - B = A + (- B)$$

Example: 7 – 6 = 7 + (–6)

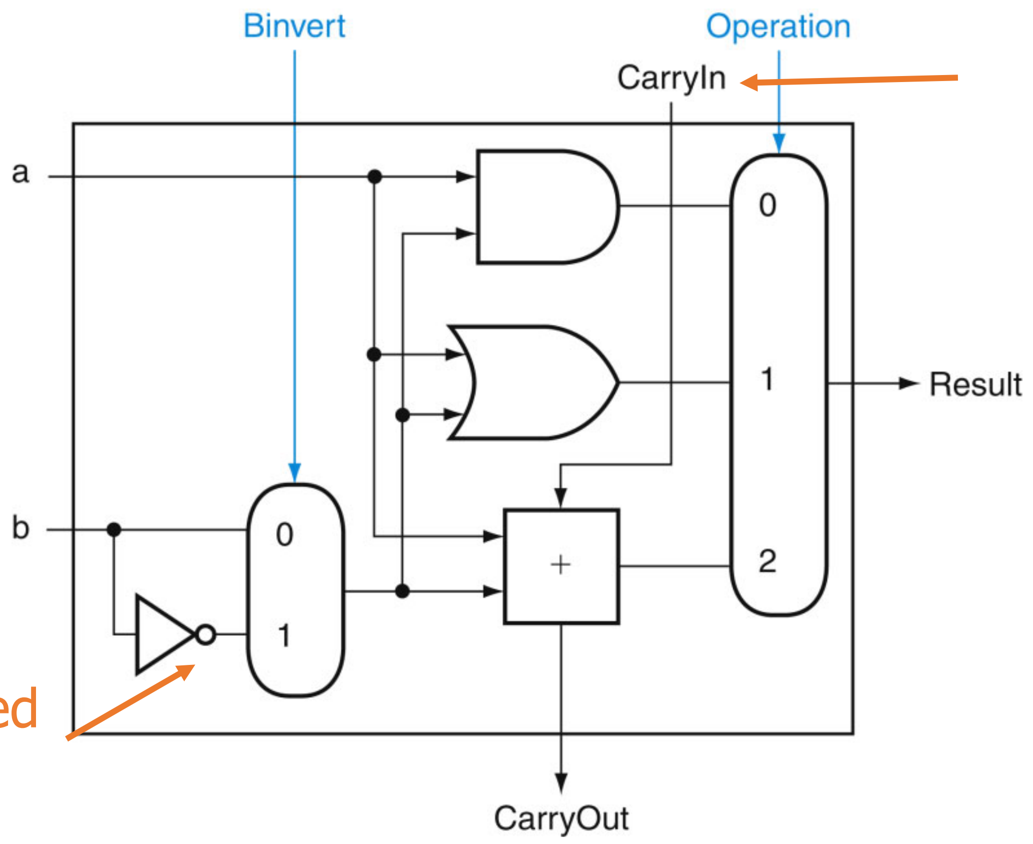
```
 7:    0000 0111
 6:    0000 0110          # bits for 6 and 7 are provided


       0000 0111
       1111 1001          # flip bits in 6
+    1 1111 1111          # add one by setting C0 to 1
-----------------         # green bits are not input
       0000 0001
```

## Subtraction

To negate a two's complement number, flip all the bits and add 1.

$$a - b = a + (-b) = a + \overline{b} + 1$$



ALU0:
CarryIn is set to 1 for subtraction

b is inverted for sub

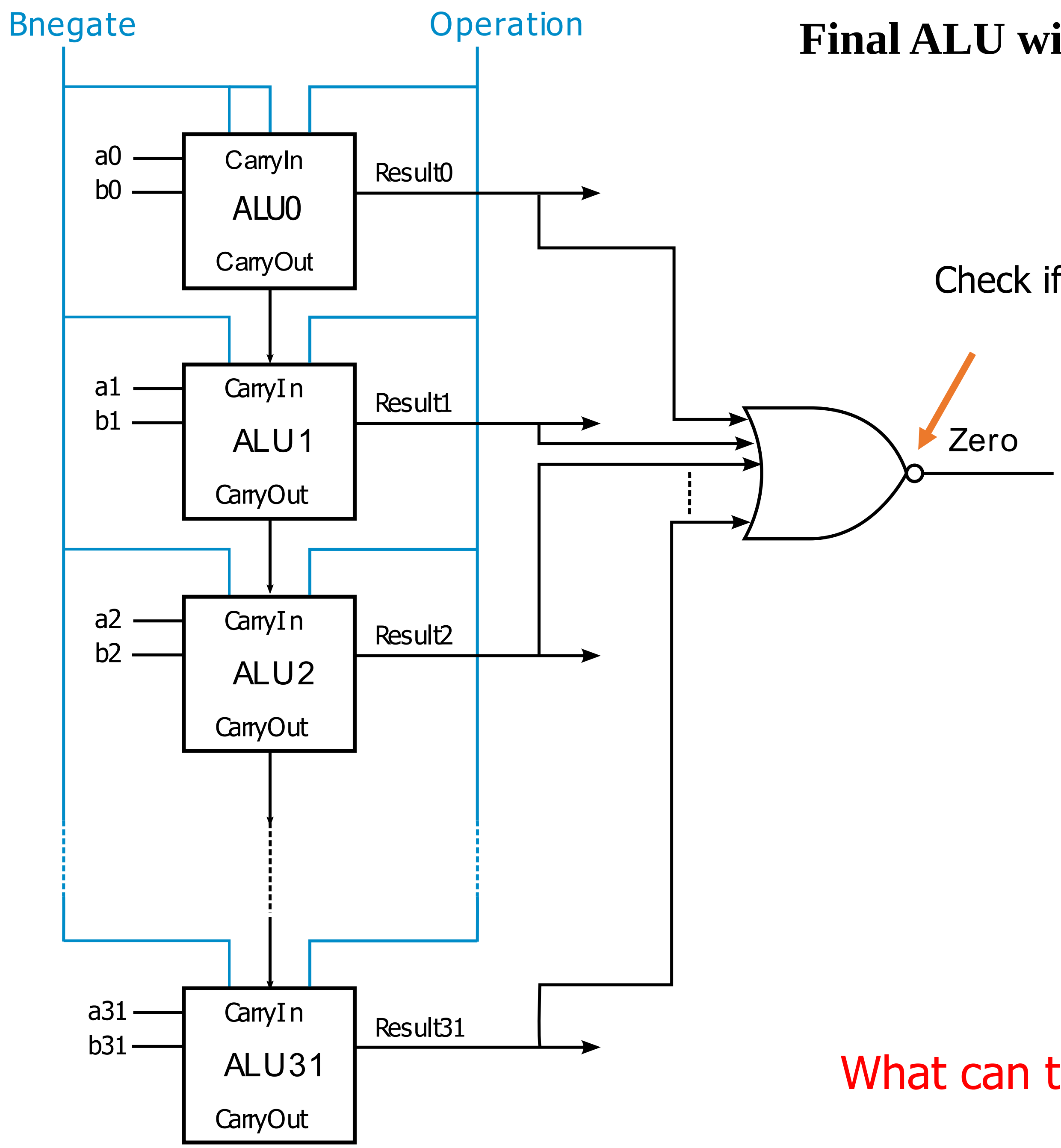
## Support for BEQ and BNE

- How do we check if two 32-bit values are the same?

BEQ, BNE

A == B

# Final ALU with zero detector



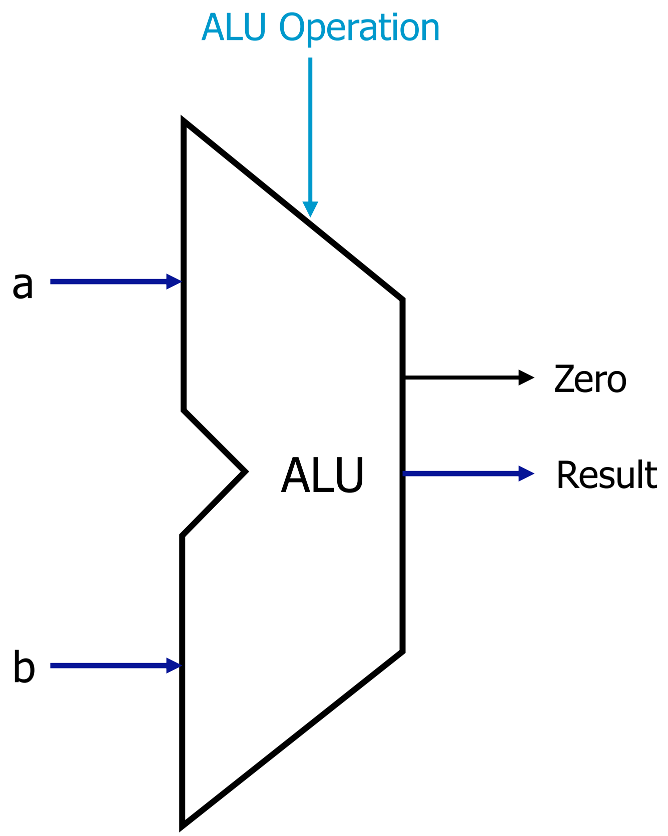
Check if all bits in result are 0

What can this ALU do?

## ALU we will use

- Support the following operations:
  - AND, OR, add, sub
- ALU operation has 4 bits, specifying the operation ALU performs
  - See any patterns in ALU operation code?

| ALU operation | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | ADD |
| 0110 | SUB |

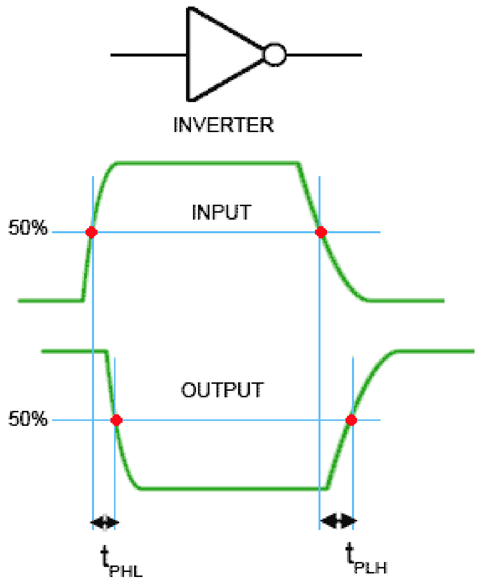


## Common Design Goals

- It works
- It is fast
- It is small
- It is energy efficient
- … and more

## Delay and Critical Path

- Signal takes time to propagate from input to output
  - Cannot go faster than light
  - Many other factors affect the delay

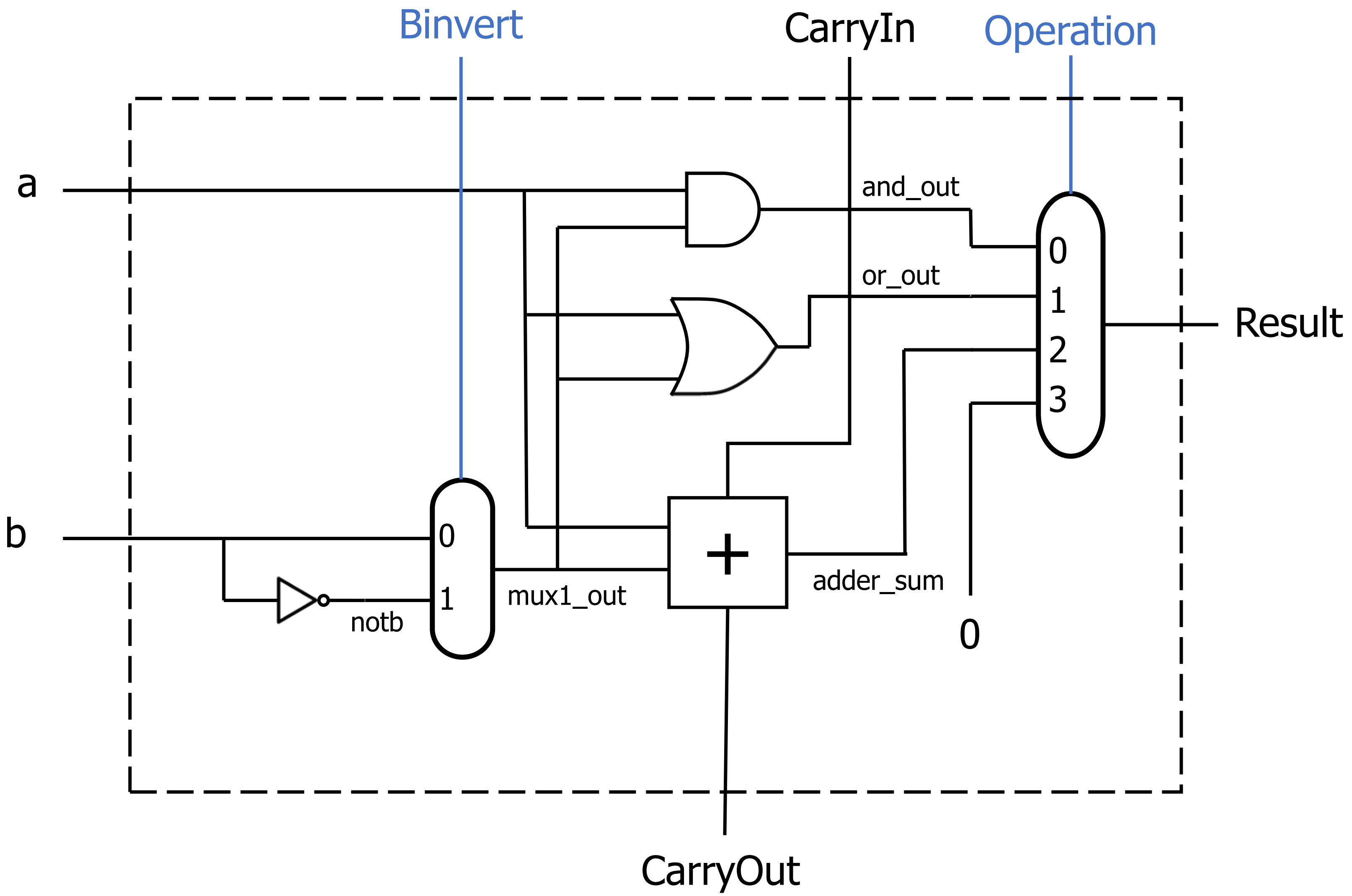


Credit: learnabout-electronics.org

- Critical path:
  - The path from input to output that has the longest delay

Can you identify the critical path of the 1-bit ALU?

# 1-bit ALU



We are going to build it in lab

## Question

- How can we modify ALU to support XOR?

## Question

There are 10 digital locks, each controlled by an UNLOCK signal.

Only one of them can be unlocked at any time.

Which of the following modules is preferred to generate the UNLOCK signals?

A. Decoder
B. Multiplexor
C. ALU

## Question

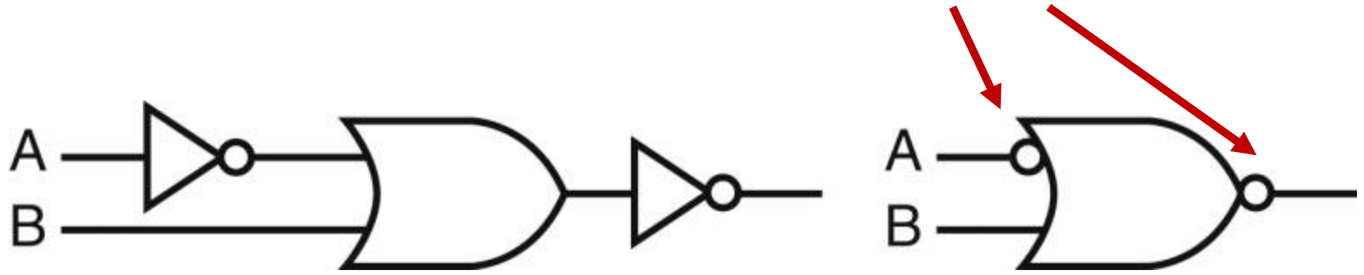
Any digital logic can be built with AND, OR, and NOT gates.

A. True
B. False

## Many ways to describe a logic function

Truth table, logic expressions, circuit diagram, and actual circuit

A bubble denotes NOT



Two diagrams for

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

which can be transformed into

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

## DeMorgan's laws

- DeMorgan’s laws on longer expressions
  - Invert all the terms
  - Change AND to OR (or OR to AND)

$$\overline{A + B + \mathrm{C} + \overline{D}} = \overline{A} \cdot \overline{B} \cdot \overline{\mathrm{C}} \cdot D$$

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

Exercise: Convert the following logic expression to a form that does not have a NOT operator.

$$\overline{\bar{W} \cdot \bar{X} \cdot \bar{Y}}$$

## Question

- Transform the following logic expression to a sum of product.

$$\mathrm{A} \cdot \overline{(\mathrm{B} \cdot \mathrm{C})} \cdot \overline{(\mathrm{C} + \mathrm{D})}$$

## Signal

- Typically, a physical signal is radio/electrical current/sound waves that carries information
- In digital circuit, we also use signal to refer to the path/wire/pin where a signal propagates

- A signal may correspond to one bit
  - The value is represented by the states of circuit elements (e.g., voltage)

A signal is asserted: signal is true or 1

A signal is deasserted: signal is false or 0

- A signal may carry multiple bits
  - For example, a bus is a collection of data lines