# Computer Arithmetic

Wei Wang

## Optional Readings from Textbooks

- “Computer Organization and Design,” Chapter 3 “Arithmetic for Computers.”
- “Computer Architecture: A Quantitative Approach,” Appendix A “Instruction Set Principles.”

# Road Map

- Binary and hexadecimal representations
- Binary Math
- Two’s Complement Encoding
- Floating-point Numbers
- Basic ALU Design

## Binary and hexadecimal representations

## Numbering Systems

- We use a decimal numbering system
- Base is `10`
- There are ten different digits, starting with zero (`0 – 9`).
- Expressing a decimal number, we write it with `n` digits as $i_n i_{n-1} i_{n-2} ... i_1 i_0$. The actual value of this number is $i_n*10^n + i_{n-1}*10^{n-1} + i_{n-2}*10^{n-2} + … + i_1*10^1 + i_1 * 10^0$
  - E.g. 256 express the value of $2*10^2 + 5*10^1 + 6*10^0 = 256_{base=10}$.

# Binary Numbers

- Similar to decimals, we can express any number in binary with 2 as the base.
- In binary numbers, there are only two different digits, which are 0 and 1.
- For a binary number $i_n i_{n-1} i_{n-2} ... i_1 i_0$,
  - it's value is $i_n * \mathbf{2}^n + i_{n-1} * \mathbf{2}^{n-1} + i_{n-2} * \mathbf{2}^{n-2} + \ldots + i_1 * \mathbf{2}^1 + i_1 * \mathbf{2}^0$
    - Notice that 10s are replaced with 2s.
  - For example, binary number 1011 has a value of $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11_{base=10}$

# Hexadecimal Numbers

- We can also express any number in hexadecimal (hex) with `16` as the base.
- In hex numbers, there are 16 different digits,
  - `0-9` as in decimal
  - The letters `A-F` for numbers 10 to 16
    - `A = 10, B = 11, C = 12, D = 13, E = 14, F = 15`
- For a hex number $i_n i_{n-1} i_{n-2} ... i_1 i_0$,
  - it's value is $i_n * \mathbf{16}^{n} + i_{n-1} * \mathbf{16}^{n-1} + i_{n-2} * \mathbf{16}^{n-2} + \ldots + i_1 * \mathbf{16}^{1} + i_1 * \mathbf{16}^{0}$
    - Notice that 10s are replaced with 16s.
  - For example, binary number A1B7 has a value of $A*16^3 + 1*16^2 + B*16^1 + 7*16^0 = (10*16^3 + 1*16^2 + 11*16^1 + 7*16^0)_{base=10} = 41399_{base=10}$

# Some Conventions

- We can write a number with its base in subscript
  - E.g., $1001_{10}$ is a decimal number, $1001_2$ is a binary number and $1001_{16}$ is a hexadecimal number.
- In most programming languages, you can write
  - `0bDDDD`, as binary numbers. E.g., `0b0001010`
  - `0xDDDD`, as hex numbers, E.g., `0x10AB`
- I may use any of these formats in the slides.

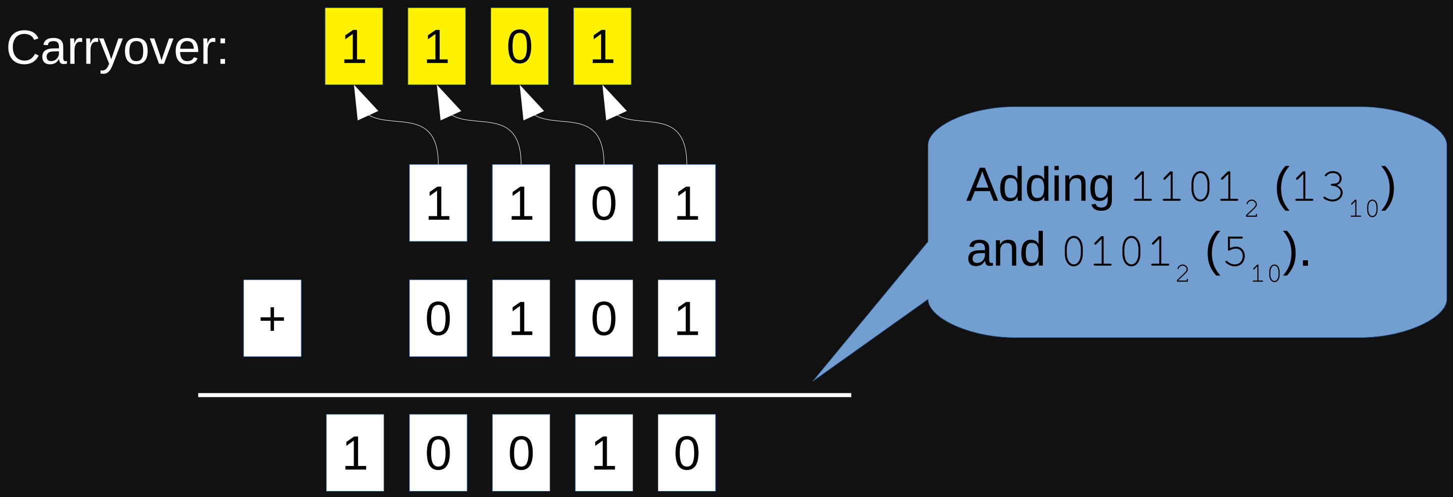
## Binary Math

## Single Digit Sums

- $0_2 + 1_2 = 1_2$
- $1_2 + 0_2 = 1_2$
- $0_2 + 0_2 = 0_2$
- $1_2 + 1_2 = 10_2$

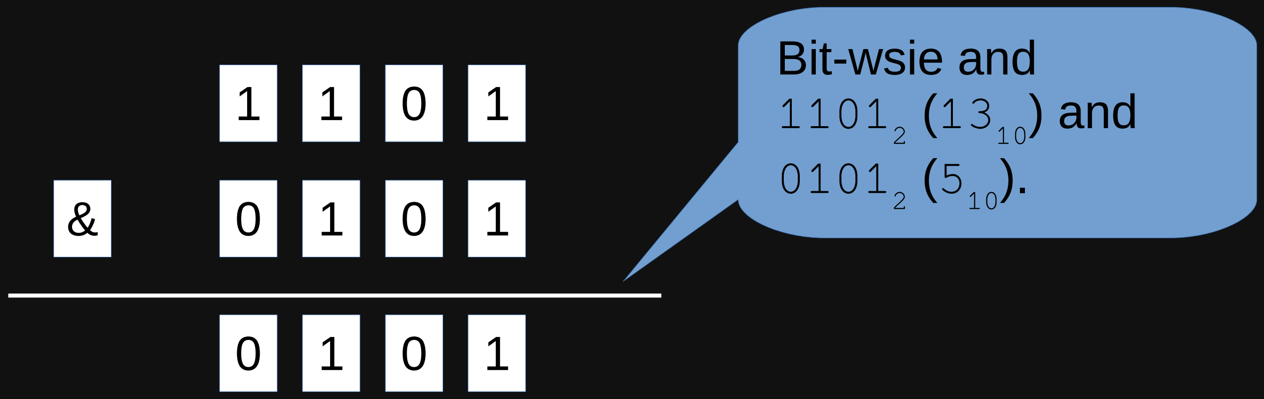
## Multiple Digits Sums

- Adding two binary numbers is the same as adding two decimal numbers
  - Add from the right-most digit
  - Keep track of carryovers



## Other Math Operations

- Subtractions, multiplications and divisions are carried out similarly to decimal numbers as well.
- Bit-wise logical operations are straightforward.



## Two’s Complement Encoding

## Numbers inside Computers

- Since we only have 0s and 1s (charged/uncharged transistors) in our computers, all numbers are represented in binary.
  - A charged bit represents `1`, and an uncharged bit represents `0`.
- For unsigned and positive integers, they are represented with bits correspond their binary representations.
- For negative integers, they are represented using a format slightly different from their absolute values, as there are no negative signs in transistors
  - This format is called **Two’s complement** encoding.

## Two’s Complement

- Let’s consider a simple case:
  - We want to express $-2_{10}$ or $-10_2$, with 3 binary bits.
  - With 3 bits, they can represent $2^3=8_{10}=1000_2$ numbers.
  - We then define the representation of $-2_{10}$ or $-10_2$ as: $1000_2-10_2 = 110_2$
- In general, computers express a negative number $-x$ with $N$ bits using the binary representation of its complement, $2^N-x$.
  - Note that $2^N$ must be larger than $|-x|$, otherwise you need more bits.

## Generalizing Two’s Complement Encoding

- In two’s complement encoding with N bits,
  - For a positive number `x`, it is encoded with its binary representation.
  - For a negative number `−x`, it is encoded with the binary representation of its complement, $2^N - x$.
- Note that, for in two’s complement encoding, the highest bit for a negative number is always `1`.

## Example: 3-bit Two’s Complement Encoding

| Decimal Value | Two’s Complement Binary Encoding |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| -4 | 100 |
| -3 | 101 |
| -2 | 110 |
| -1 | 111 |

# A Faster Algorithm of Two’s Complement

- A faster algorithm to find the two’s complement encoding for `−x` is,
  - Get the encoding for `x`. For example, for `−1`, we starts with the 3-bit encoding of `1`, which is `001`
  - Invert the digits of `x`’s encoding, i.e., `0` becomes `1`, `1` becomes `0`. E.g., for `001`, inverting the digits gives `110`.
  - Add `1` to the inverted encoding gives the encoding of `−x`. E.g., add `1` to `110` gives `111`, which is the two’s complement encoding for `−1`.
- I am not going to discuss why this algorithm works, but you are welcome to formally prove its correctness.

# Why Two’s Complement?

- Two’s complement encoding makes add every easy.
  - No need to convert adding a negative number into subtracting.
  - For example, adding `−1` with `3`. With two’s complement, we are adding `111` with `011`.
    - Simply adding them as two positive binary integers, we have `111 + 011 = 1010`.
    - Removing the left-most bit, we have `010`, with is `2`, the correct result.
  - There are still corner cases to handle, but much easier in general.
- Subtraction is also fairly easy with two’s complement.
- Multiplication and division are slight more difficult than adds and subtractions, but no more difficult than multiplying and dividing in other encoding.

## Floating-Point Numbers

# Scientific Notation for Real Numbers

- Computers are also used to do computations on real numbers.
- Real numbers are written in the following form in scientific notation

  $m * 10^n$

- For example,

| Decimal Notation | Scientific Notation |
|---|---|
| 2 | $2.0 * 10^0$ |
| 300 | $3.0 * 10^2$ |
| 42421.3232 | $4.24213232 * 10^4$ |
| −0.2 | $-2.0 * 10^{-1}$ |

## Reals in Binary Format with Scientific Notation

- Similarly, we can express reals in binary format with scientific notation:
  $\texttt{1.xxxxxxx} * 2^{\texttt{eeee}}$
- For example:
  - $\texttt{1.001010} * 2^{\texttt{10010}}$
- The `xxxxx` is the fraction of the real number, while `eeee` is the exponent in base 2.
- The dot before the fraction part is called binary point (analogous to decimal point).
- Note that there is only one digit before the binary point, and that digit is always `1`.
  - Since the digits can only be `1` or `0`, we can always remove leading `0`s and leaving only one `1` before the decimal point.

# Floating Point Numbers

- Computer arithmetic that supports binary real numbers in their scientific format is called floating point, as the position of the binary point is moving depending on the value of the exponent.
  - An integer is fixed point, as the binary point is always at the end of the integer.
- In C language, real numbers are simply call floats.

## Encoding Floats with Bits

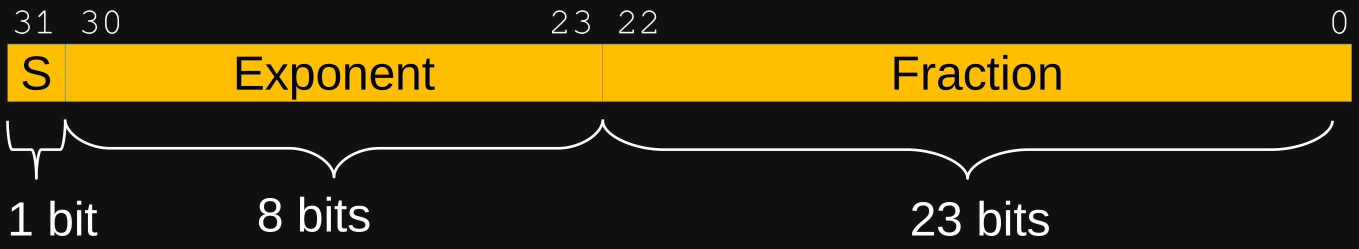
- Derived from the scientific notation, a binary floating point number is uniquely identified by its sign (s), fraction (F) and exponent (E).

  $(-1)^{s} * 1.F * 2^{E}$

- Therefore, computers encode floats with only sign, fraction and exponent.
  - The fraction part is also called “mantissa.”

# Encoding Floats with Bits cont'd

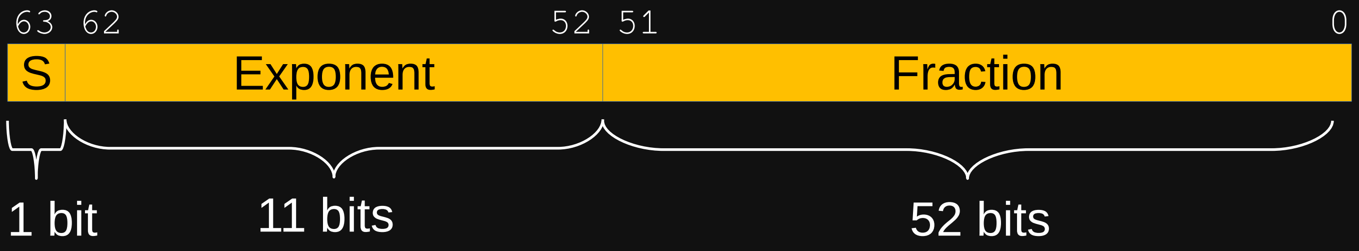
- For 32-bit floats



  - 1 bit for the sign, 8 bits for exponent and 23 bits for fractions
  - The maximum positive float with 32-bit encoding is $(1+2^{-23})*2^{127}$, roughly $10^{38}$
  - The minimum positive float with 32-bit encoding is $(2-2^{-32}) * 2^{-126}$, roughly $10^{-38}$
  - Any number larger than the maximum causes an overflow, while any number smaller than the minimum causes an underflow.

## Encoding Floats with Bits cont’d

- For 64-bit doubles



  - 1 bit for the sign, 11 bits for exponent and 32 bits for fractions

# IEEE 754 Encoding

- IEEE 754 is a standard encoding for floating point numbers.
- The encoding is roughly the same as the 32-bit and 64-bit encoding we have seen.
- Exponent part is encoded with a bias, instead of two's complement
  - If the exponent is E (positive or negative), it is encoded as the binary number of (E + Bias).
  - For 32-bit single-precision floats, the bias is 127.
    - E.g., if the exponent is -5, it is encoded as (-5+127=122) or 0b01111010.
  - For 64-bit double-precision floats, the bias is 1023.

## IEEE 754 Encoding cont’d

- Specially values of the exponent and fraction are reserved to represent special cases

| Exponent | Fraction | Value |
|---|---|---|
| 0 | 0 | 0 |
| 0b11111111 | 0 | Infinity |
| 0 | Not 0 | Denormalized (no leading 1 before binary point) |
| 0b11111111 | Not 0 | Not a number (NAN) |

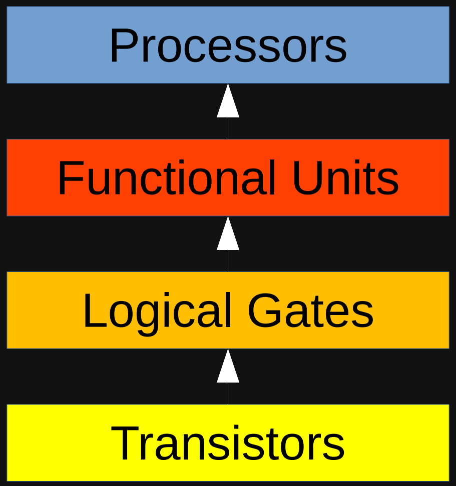
## Floating Point Computation

- There are standard algorithms for
  - Converting between binary floats and decimal floats
  - Doing sum, subtraction, multiplication and division.
- I will not cover these algorithms, but you can easily find these algorithms online and in the text book.

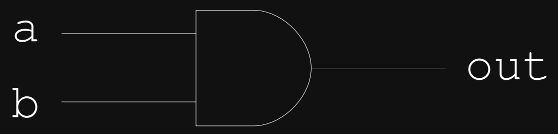
## Basic ALU Design

## Logical Gates

- Processors are built from transistors.
- More accurately, transistors constitute logical gates, logical gates constitute functions units, and functional units constitute processors.

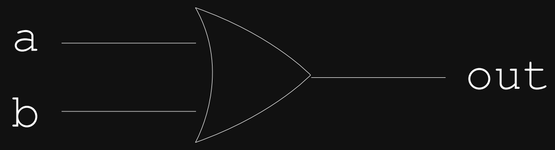


## Basic Logic Gates: AND Gate



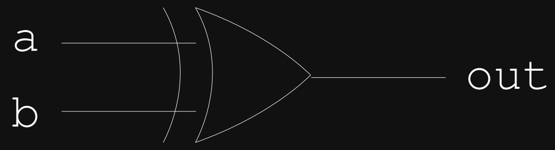
| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Basic Logic Gates: OR Gate



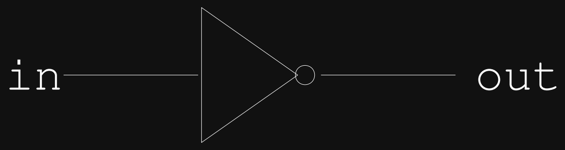
| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

## Basic Logic Gates: XOR Gate



| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

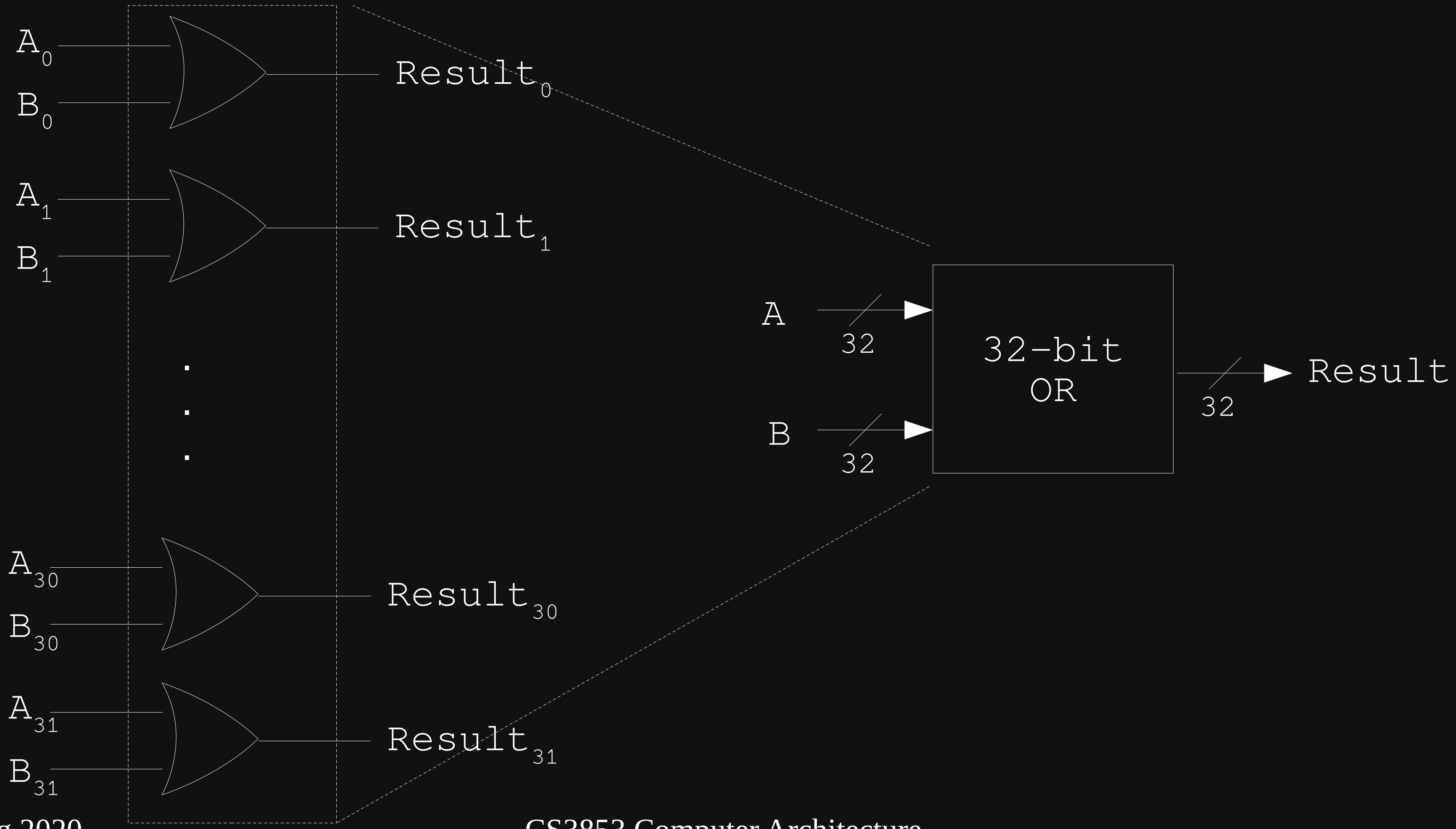
## Basic Logic Gates: Not Gate



| in | out |
|---|---|
| 0 | 1 |
| 1 | 0 |

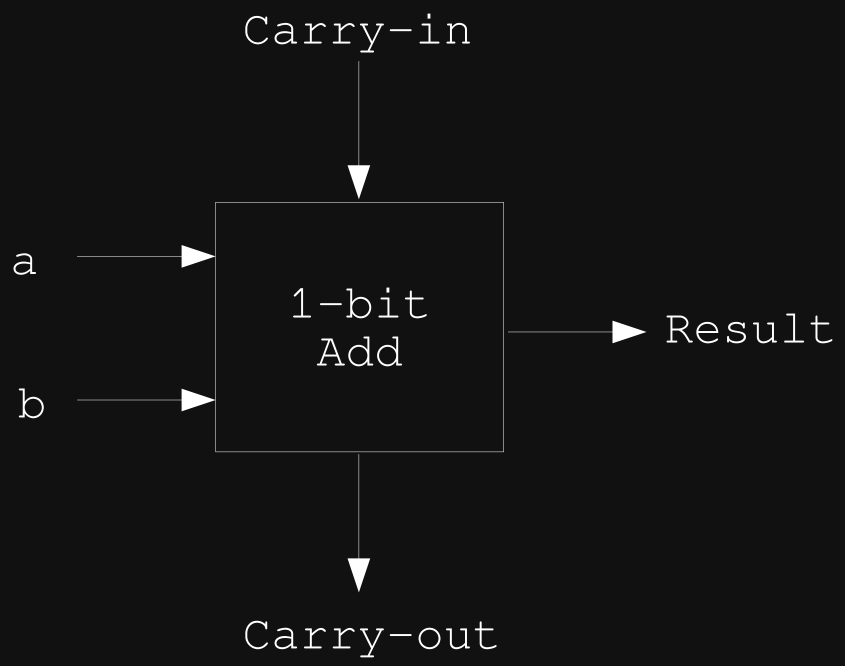
# ALU: 32-Bit Bitwise OR

- Connect 32 OR gates in parallel.



# ALU Unit: 1-Bit Adder

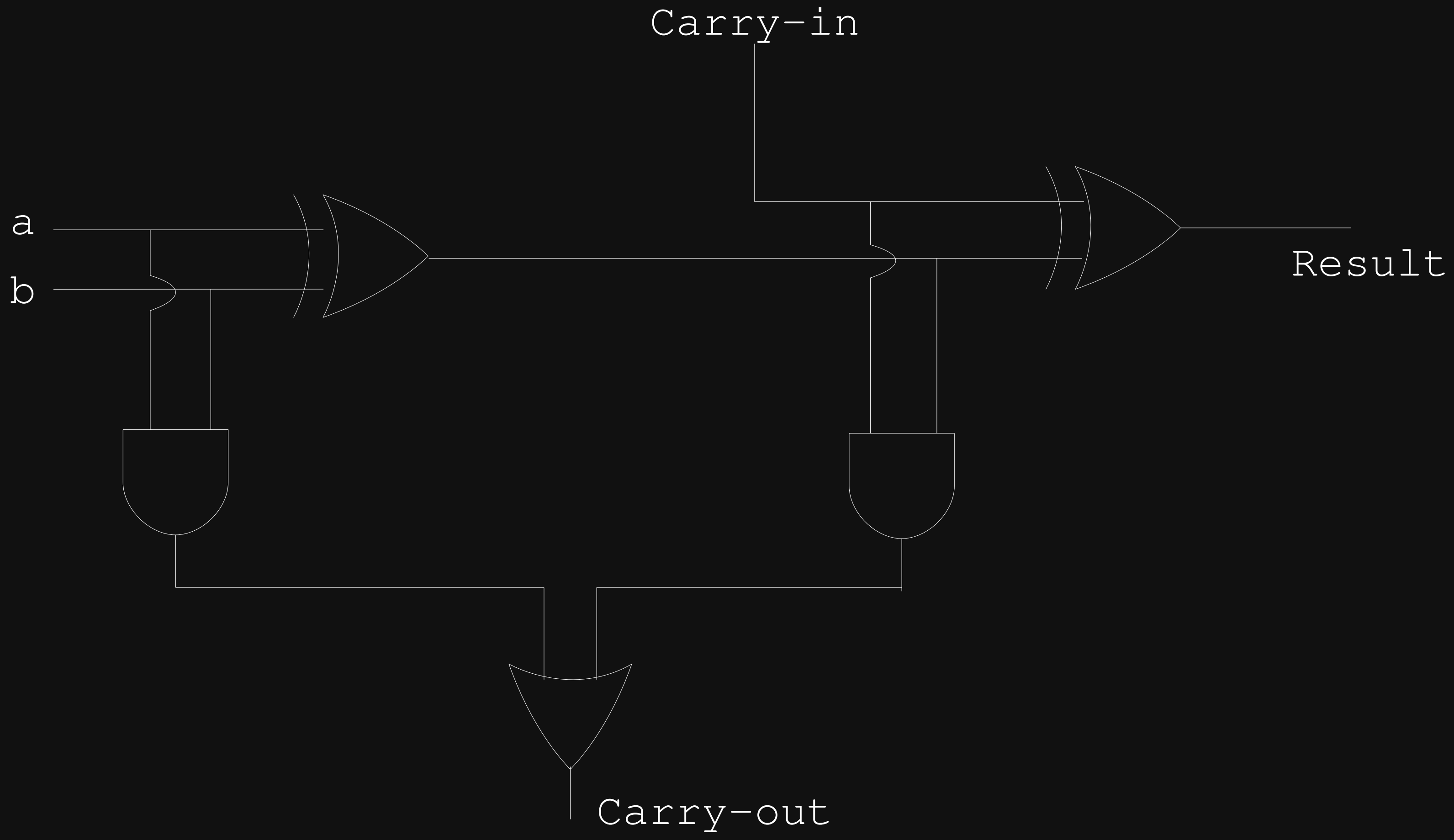
- 1-bit add has two 1-bit inputs and one carry-in.
- It returns one 1-bit results with a carry-out



| $C_{in}$ | a | b | Res. | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

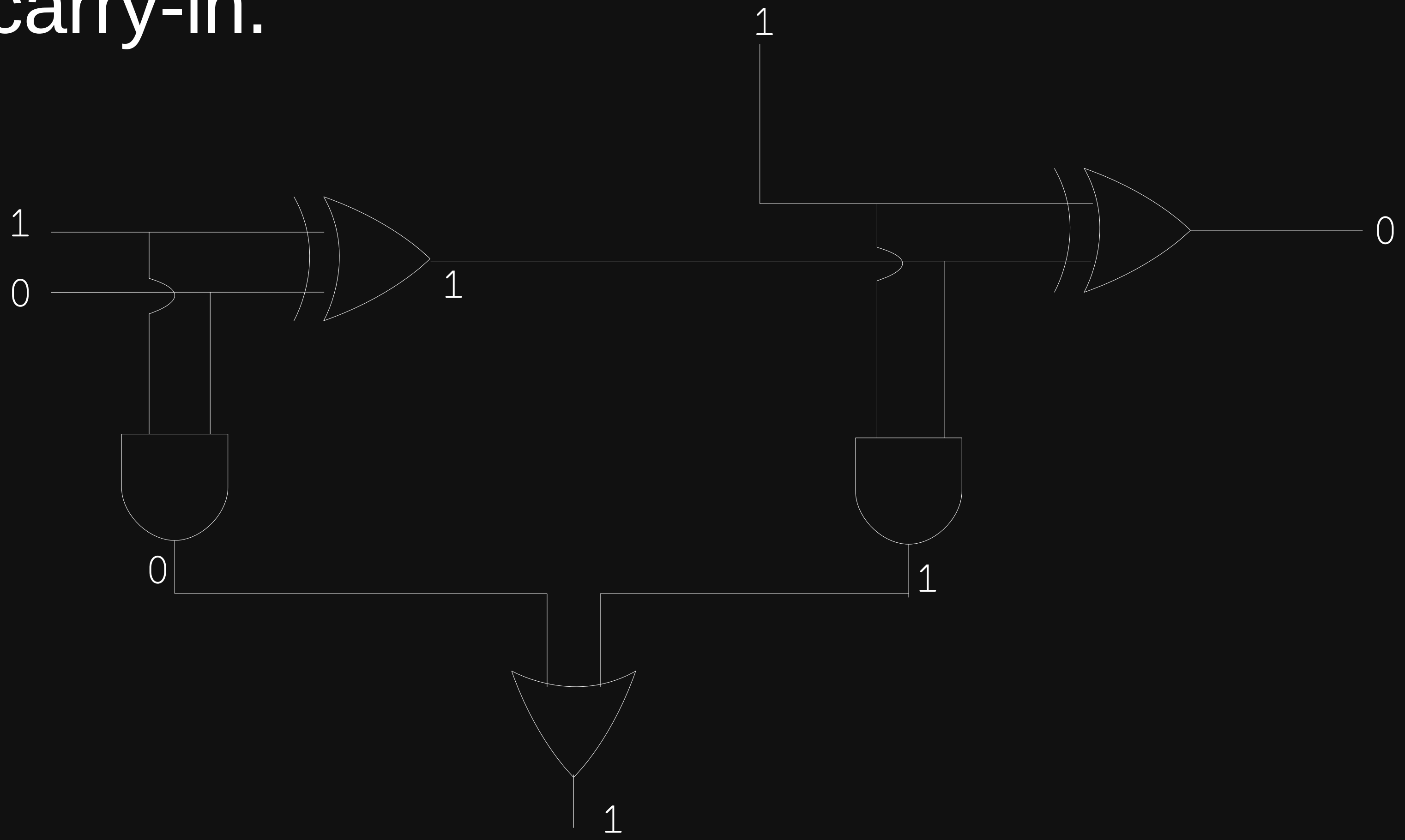
## ALU Unit: 1-Bit Adder cont’d

- Internal of 1-Bit add with 5 logic gates



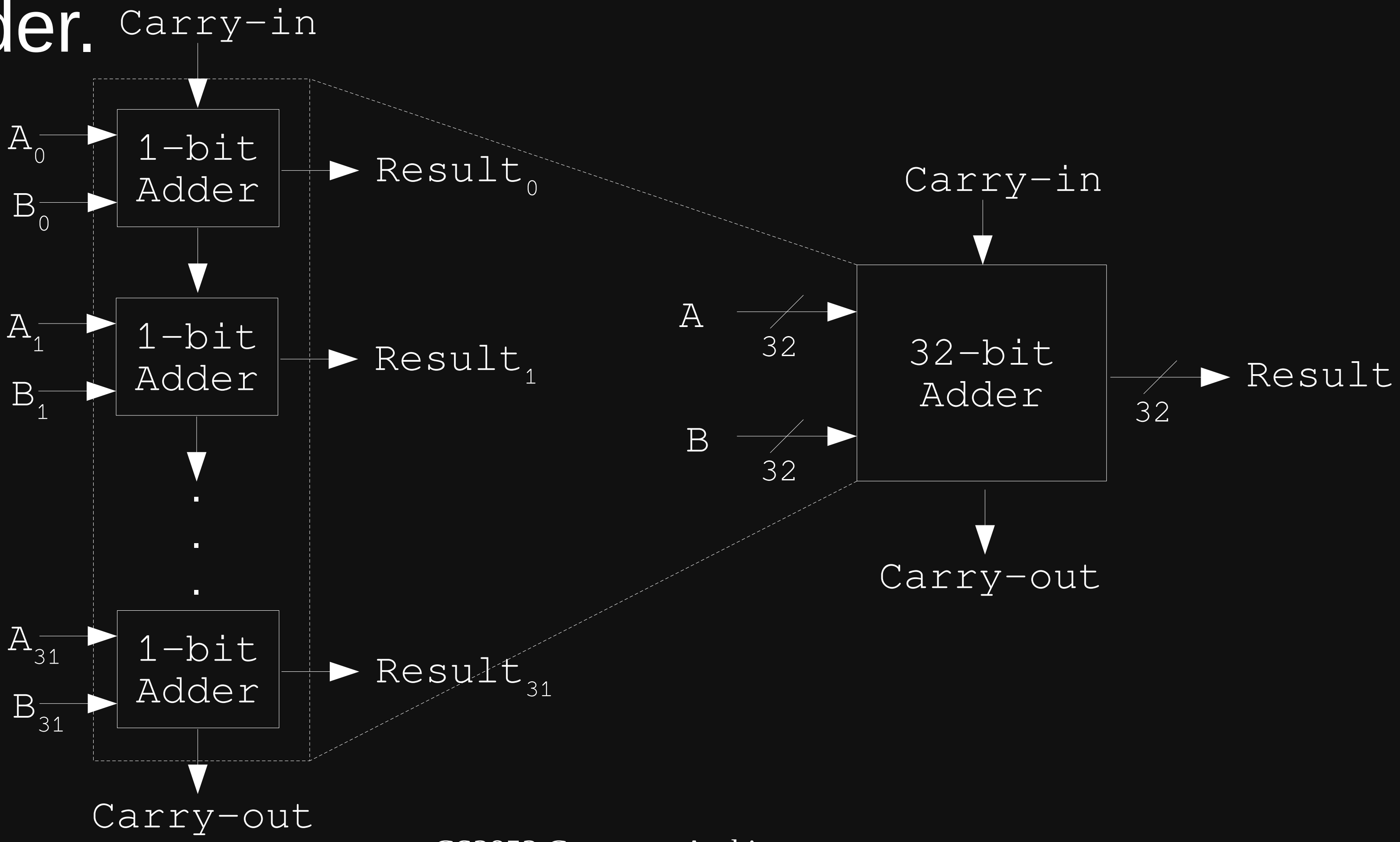
# ALU Unit: 1-Bit Adder cont’d

- A simple example of adding `1` and `0`, with `1` as carry-in.



# ALU Unit: 32-Bit Adder

- Serially connecting 32 1-bit adder gives a 32-bit adder.

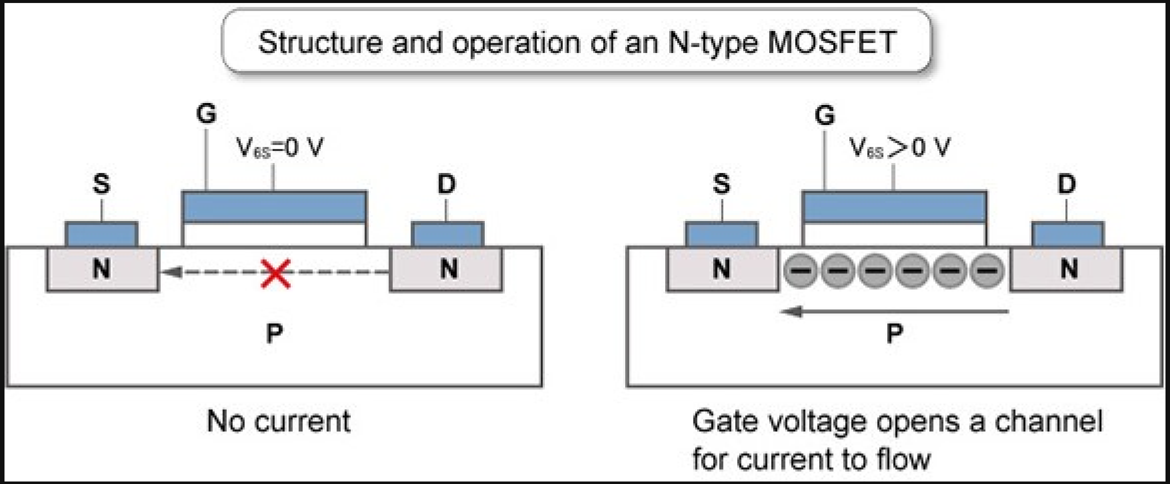


## Other Function Units

- Most functional units can be built from logical gates.
  - E.g., subtracter, multiplier and multiplexer
- Some functional units can be very complex, such as branch predictors.

# Transistor Basics

- Transistors used in processors are mostly Field-Effect Transistors (FETs).
  - Most common FET is MOSFET (metal-oxide-semiconductor FET).
  - For a MOSFET transistor, when it is given a gate voltage ($V_{GS}$) that is high enough, the transistor will be turned on and allows current to flow.

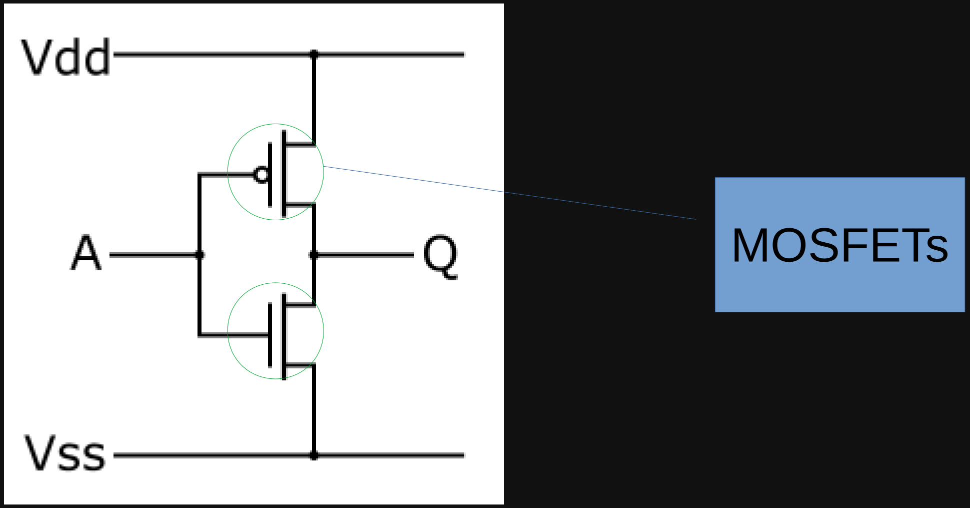

Structure and operation of an N-type MOSFET
G
$V_{GS}$=0 V
S
D
N
N
P
No current
G
$V_{GS}$>0 V
S
D
N
N
P
Gate voltage opens a channel for current to flow

A N-Type MOSFET as an ON/OFF Switch

* figure from https://www.renesas.com/us/en/support/technical-resources/engineer-school/electronic-circuits-02-diodes-transistors-fets.html

## Transistor Basics cont’d

- A NOT-gate (aka. CMOS inverter) can be built from P-type and N-Type MOSFETs:



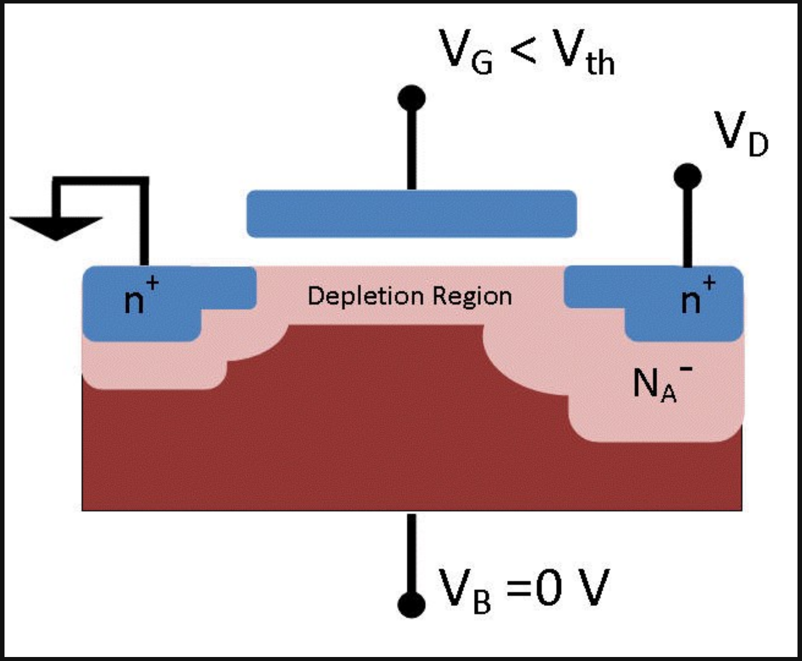
- Similarly, all other gates can be built from transistors

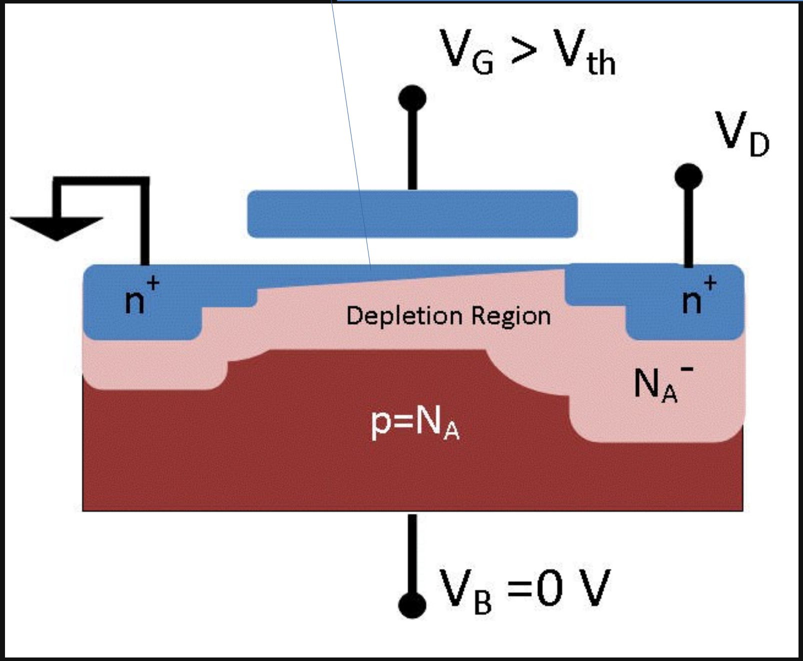
## Transistor Basics cont'd

- Computer architects generally ignore MOSFET and the internal design of logic gates. They focus more on the management of functional units.
- But the decline of Dennard Scaling drew computer architects' attention back to MOSFET.
  - For a MOSFET transistor, there is a minimum voltage that is required to turn on the transistor, which is call threshold voltage, $V_{th}$.
  - As the gate voltage ($V_{GS}$) keeps reducing, it finally approaches $V_{th}$. Once $V_{GS}$ is the same as $V_{th}$, we cannot reduce it anymore, and hence the failure of Dennard Scaling.

# Another Illustration of Threshold Voltage

Note the blue materials are connected





## Acknowledgment

- These slides are partially based on the lecture notes from Dr. Mirela Damian.