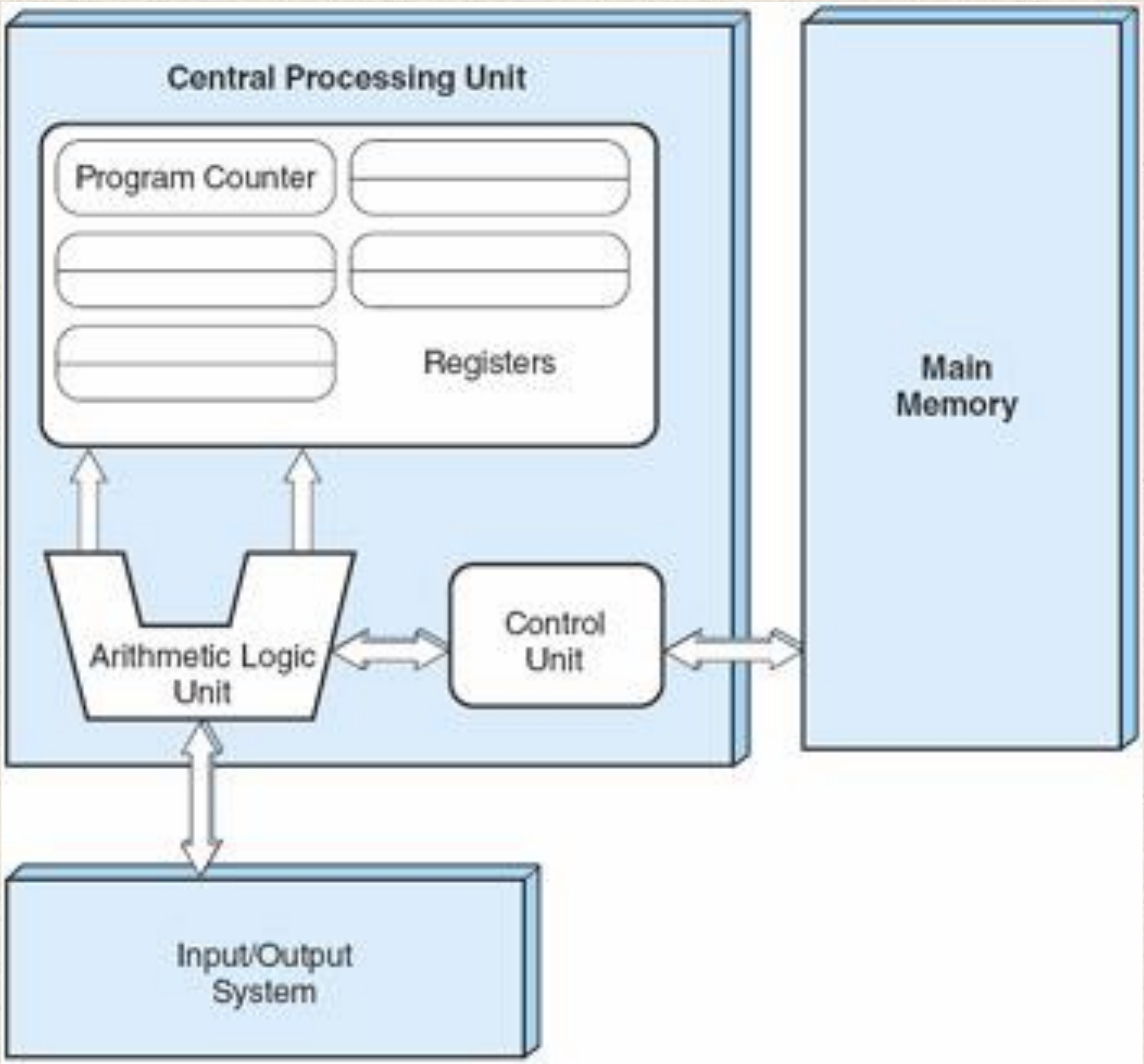
# Chapter 2: Computer Arithmetic

# Topics

- Review: The Von Neumann Model
  - Chapter 1.8 by Null
- Number system
  - Chapter 1 by Berger
  - Chapter 2.1 by Null
- Negative binary number
  - Chapter 2.4, 2.5 (Null)
- IEEE floating point
  - Chapter 2.4, 2.5 (Null)

# The Von Neumann Model



# The Von Neumann Model

- Consists of three hardware systems: A central processing unit (CPU) with a control unit, an arithmetic logic unit (ALU), registers (small storage areas), and a program counter; a main memory system, which holds programs that control the computer's operation; and an I/O system.
- Capacity to carry out sequential instruction processing.
- Contains a single path, either physically or logically, between the main memory system and the control unit of the CPU, forcing alternation of instruction and execution cycles.

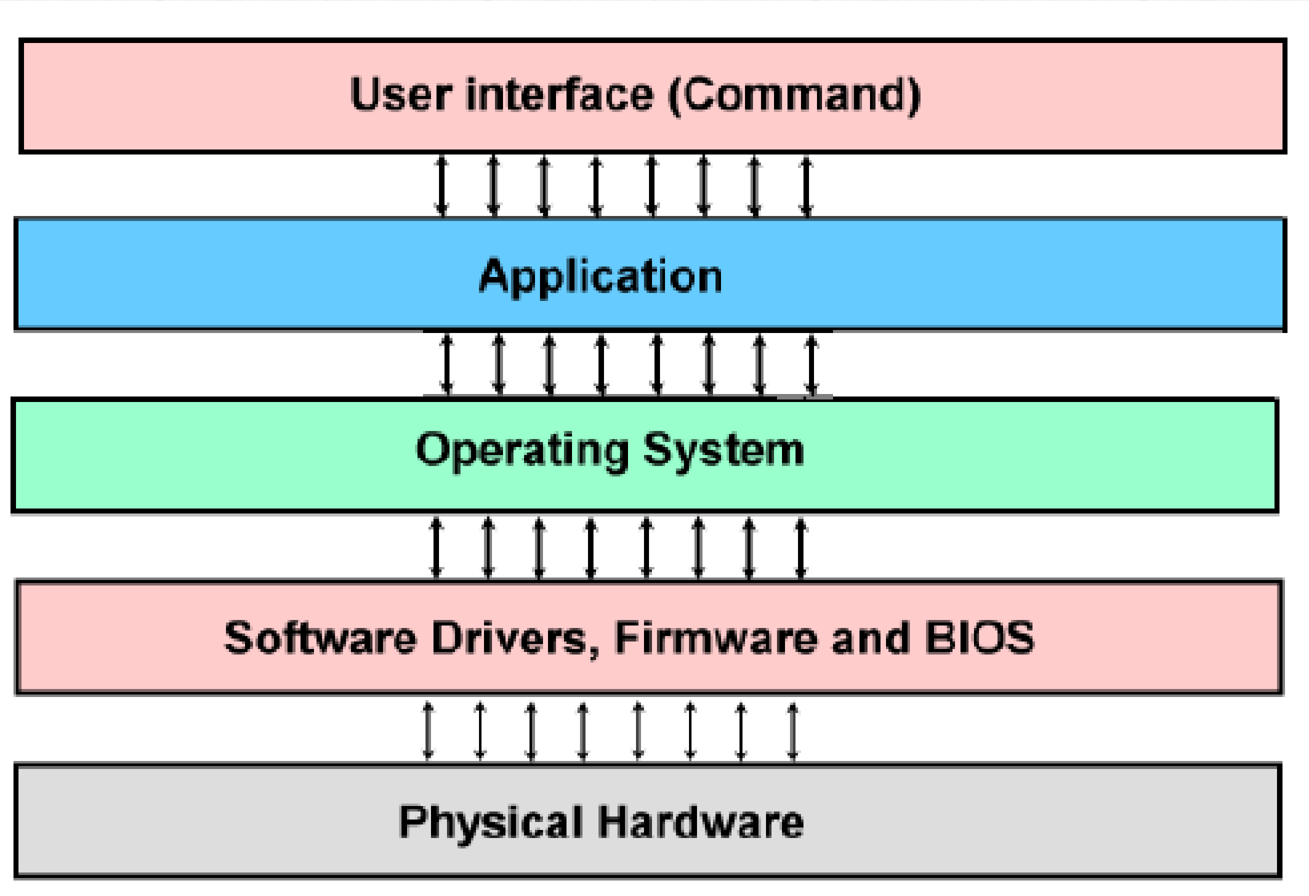
# The Digital Computer

- Machine to carry out instructions
  - A program

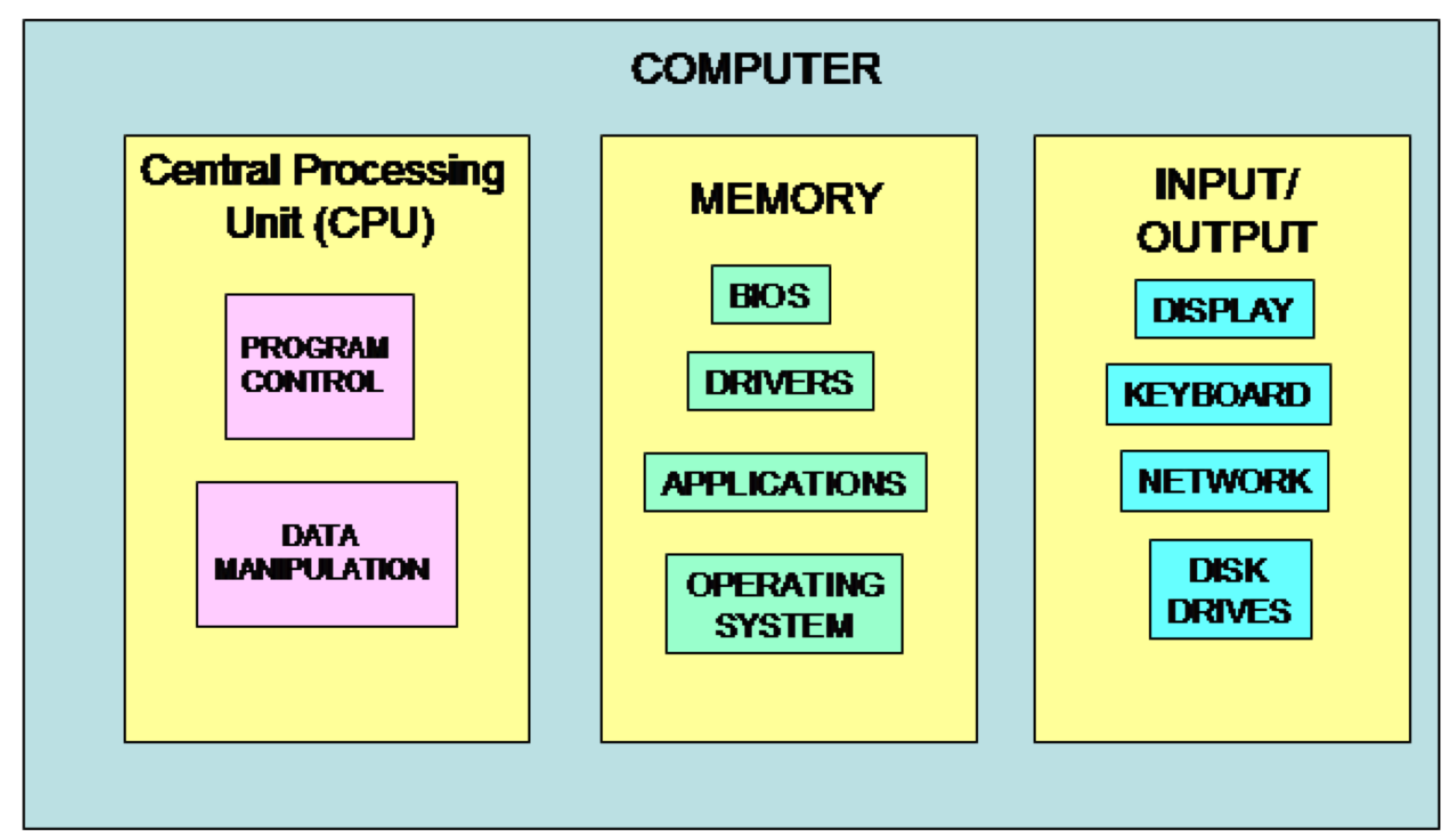
- Instructions are simple
  - Add numbers (*no subtraction actually*)
  - Check if a number is zero
  - Copy data between memory locations

- Primitive instructions in machine language

# Software Designer's View of Today's Computer



# Hardware Designer's View of Today's Computer



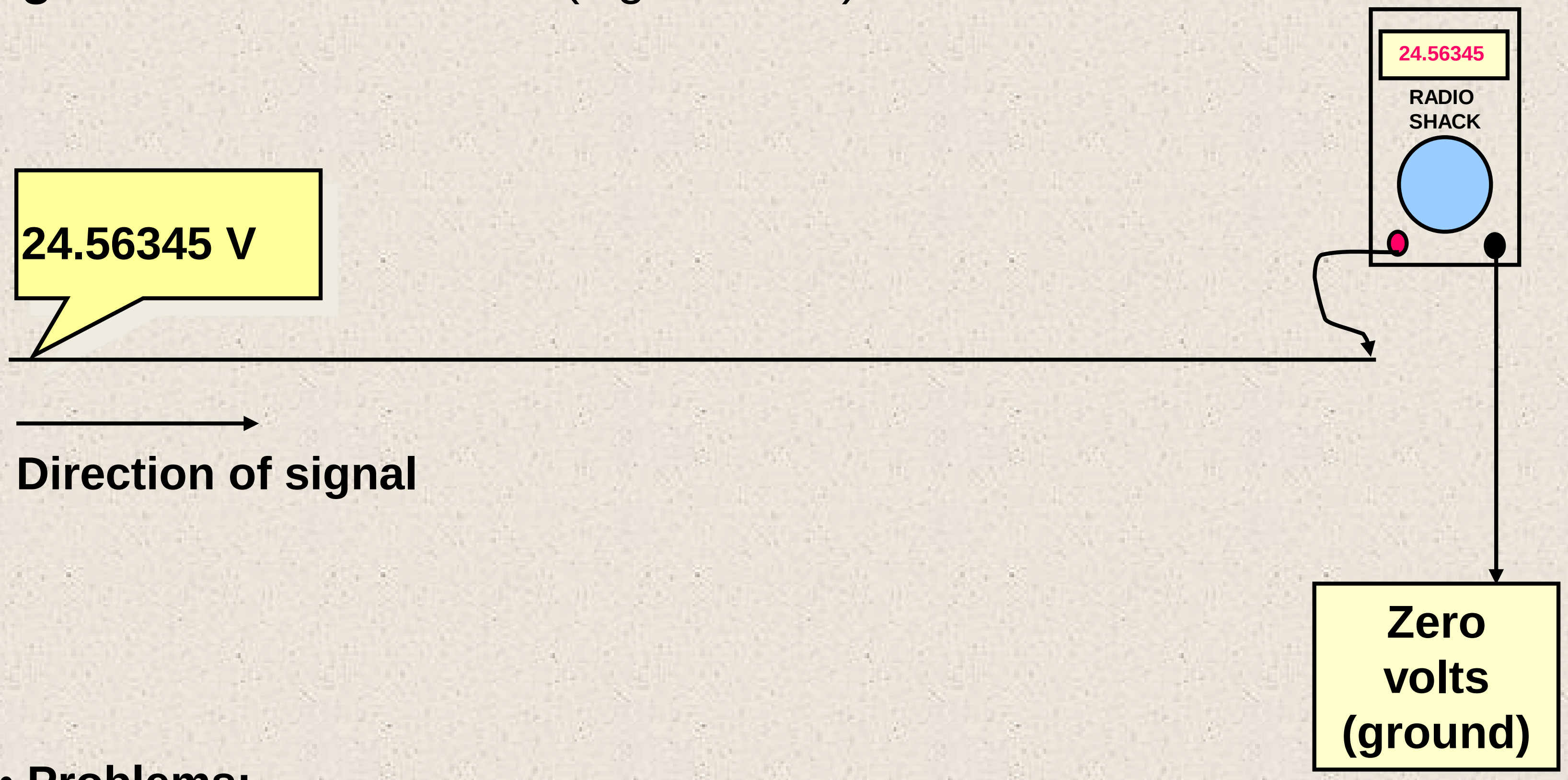
# Data Representation in Computers

**Question:**

**How can we quickly, cost effectively and accurately transmit, receive, store and manipulate numbers in a computer?**

# Possible Approach #1

• Represent the data value **as a voltage or current** along a **single** electrical conductor (signal trace) or wire




- **Problems:**
  - **Measuring large numbers is difficult, slow and expensive!**
  - **How do you represent +/- 32,673,102,093?**

# Possible Approach #2

- Represent the data value **as a voltage or current** along **multiple** electrical conductors
- Let **each wire represent one decade of the number**
- Only need to **divide up the voltage on each wire into 10 steps**
  - 0 V to 9 volts

# Comments on Approach #2

- **This is better than the first approach**
  - Only need to worry about 10 discrete signal levels
- However, **modern electronics are still not sufficiently fast enough to make this a viable solution**
- It can have **considerable "slop" between values** before it causes problems
  - What if the second wire gives 4.2 V, or 4.5 V?

- **Better approach?**
- Hint: Electronics are ***really good*** at switching things on and off very fast
  - Modern transistors (electronic switches and amplifiers) can switch a signal on or off in 10's of picoseconds (trillionths of a second)

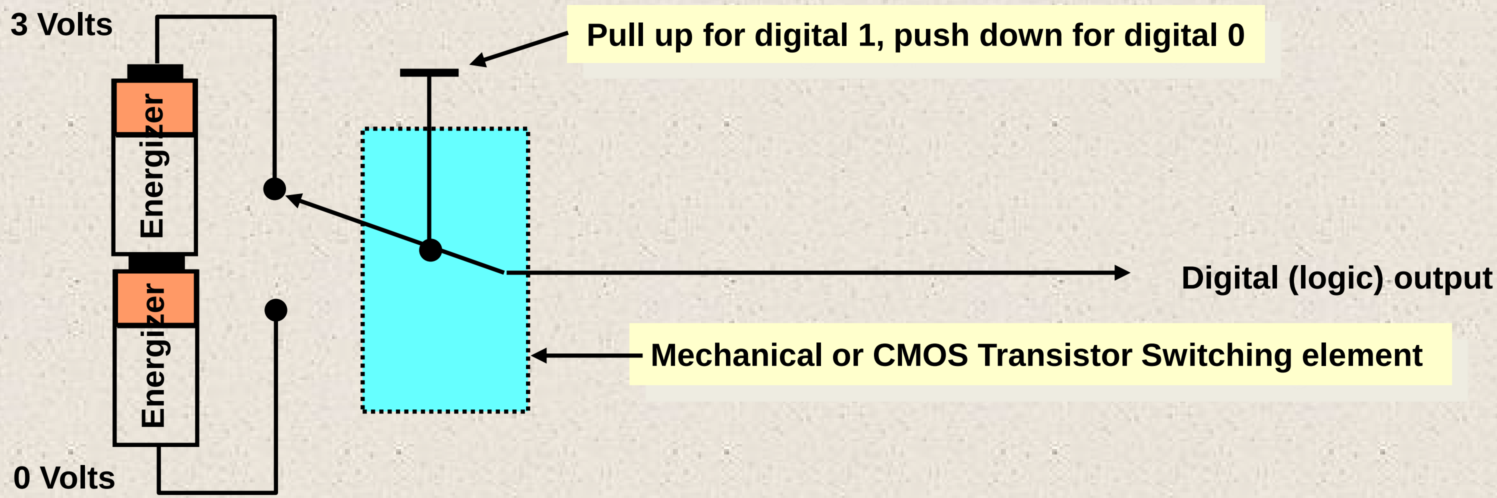
# Possible Approach #3

- Represent the data value **as a voltage or current** along **multiple, parallel,** electrical conductors
- Let **each wire represent one power of 2 of the number ( $2^0$ ~ $2^N$ )**
- Only need to **divide the voltage on each wire into 2 possible steps**
    - 0 V “no volts” or “some volts” greater than zero (on or off )
- **Can have lots of “slop” between values**

| | | | | |
|---|---|---|---|---|
| $2^0$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^1$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |
| $2^2$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^3$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |
| $2^4$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |
| $2^5$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^6$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^7$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^8$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |
| $2^9$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |
| $2^{10}$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |
| $2^{11}$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^{12}$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^{13}$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^{14}$ | $\longrightarrow$ | on | $\longrightarrow$ | 1 |
| $2^{15}$ | $\longrightarrow$ | off | $\longrightarrow$ | 0 |

# Comments on Approach #3

- Using transistors as electronic, high-speed on/off switches is a very efficient way to accurately send signals at high speed
- Each signal on a wire is either “on” or “off”
  - An “on” signal means that **some voltage is present** ( ~3 volts or greater )
  - An “off” signal means that the **voltage is mostly absent** ( < 0.4 volts )
- Each wire or signal trace represents either the number 0 ( no voltage ) or the number 1 ( some voltage )
- Imagine that each electronic device is like a mechanical switch that can **quickly switch the voltage on a wire** between 0 volts and 3 volts



# Binary Number System

- Since we are switching between two voltage levels, our number system has only 2 digits, 1 or 0: a ***binary*** number system
- The arithmetic and logical operations on a set of binary number is called ***Boolean Algebra***
- From approach #3, what is the number:
  - off on on on on off off off on on on off off on off on ?
  - 0 1 1 1 1 0 0 0 1 1 1 0 0 1 0 1 ?
  - Answer: 30949

- How did I get this?
  - $0 \times 2^{15} = 0$ $1 \times 2^{14} = 16{,}384$ $1 \times 2^{13} = 8{,}192$
  - $1 \times 2^{12} = 4{,}096$ $1 \times 2^{11} = 2{,}048$ $0 \times 2^{10} = 0$
  - $0 \times 2^{9} = 0$ $0 \times 2^{8} = 0$ $1 \times 2^{7} = 128$
  - $1 \times 2^{6} = 64$ $1 \times 2^{5} = 32$ $0 \times 2^{4} = 0$
  - $0 \times 2^{3} = 0$ $1 \times 2^{2} = 4$ $0 \times 2^{1} = 0$
  - $1 \times 2^{0} = 1$

16,384 + 8,192 + 4,096 + 2,048 + 128 + 64 +32 + 4 + 1 = 30949

# Number Systems

- We count in the decimal system because we have 10 fingers
  - There is nothing unique about counting in decimal
  - We would count in octal (base 8) if we had 8 fingers
- The ***BASE (Radix)*** of a number system is just the number of distinct digits in that system
  - Computer systems are naturally binary (base 2)
  - Common number systems used with computational devices:
  - Base 2: 0,1 : Binary
  - Base 8: 0,1,2,3,4,5,6,7 : Octal
  - Base 10: 0,1,2,3,4,5,6,7,8,9 : Decimal
  - Base 16: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F : Hexadecimal

# Binary, Octal and Hexadecimal

- We use the binary number system to represent numbers and logical operations in a computer
- Reading and writing binary numbers is tedious and error-prone because the numbers can be very long
- Octal and hexadecimal are ways to simplify the representation of numbers to make them easier to understand and manipulate
- For example:
  - 0 1 1 1 1 0 0 0 1 1 1 0 0 1 0 1 = 30,949 in decimal
  - 0 111 100 011 100 101 = 074345 in octal
  - 0111 1000 1110 0101 = 78E5 in hexadecimal
- Notice how hexadecimal is the most compact way to represent the number
- Notice how the binary numbers are grouped together in octal (by 3) and hexadecimal (by 4)
- As you'll see, we convert between binary, octal and hexadecimal be **changing how the binary numbers are grouped** together

# Converting from Decimal to Other Bases

- Algorithm:
  - Example: Convert 73,503 to base 17

| | quotient | | remainder |
|---|---|---|---|
| 17 | 73,503 | | |
| 17 | 4,323 | ……… | 12 (C) |
| 17 | 254 | ……… | 5 |
| 17 | 14 | ……… | 16 (G) |
| 17 | 0 | ……… | 14 (E) |

Therefore, the answer is $EG5C_{17}$.

Why this algorithm?

EG5C = { E * 17^2 + G * 17 + 5}*17 + C

= {{ E * 17 + G} * 17 +5} * 17 + C

# Converting from Decimal to Other Bases

- Algorithm:
  - Example: Convert $EG5C_{17}$ To Decimal

EG5C = E * 17^3 + G * 17^2 + 5 * 17^1 + C *17^0

= {{ E * 17 + G} * 17 +5} * 17 + C

.

.

.

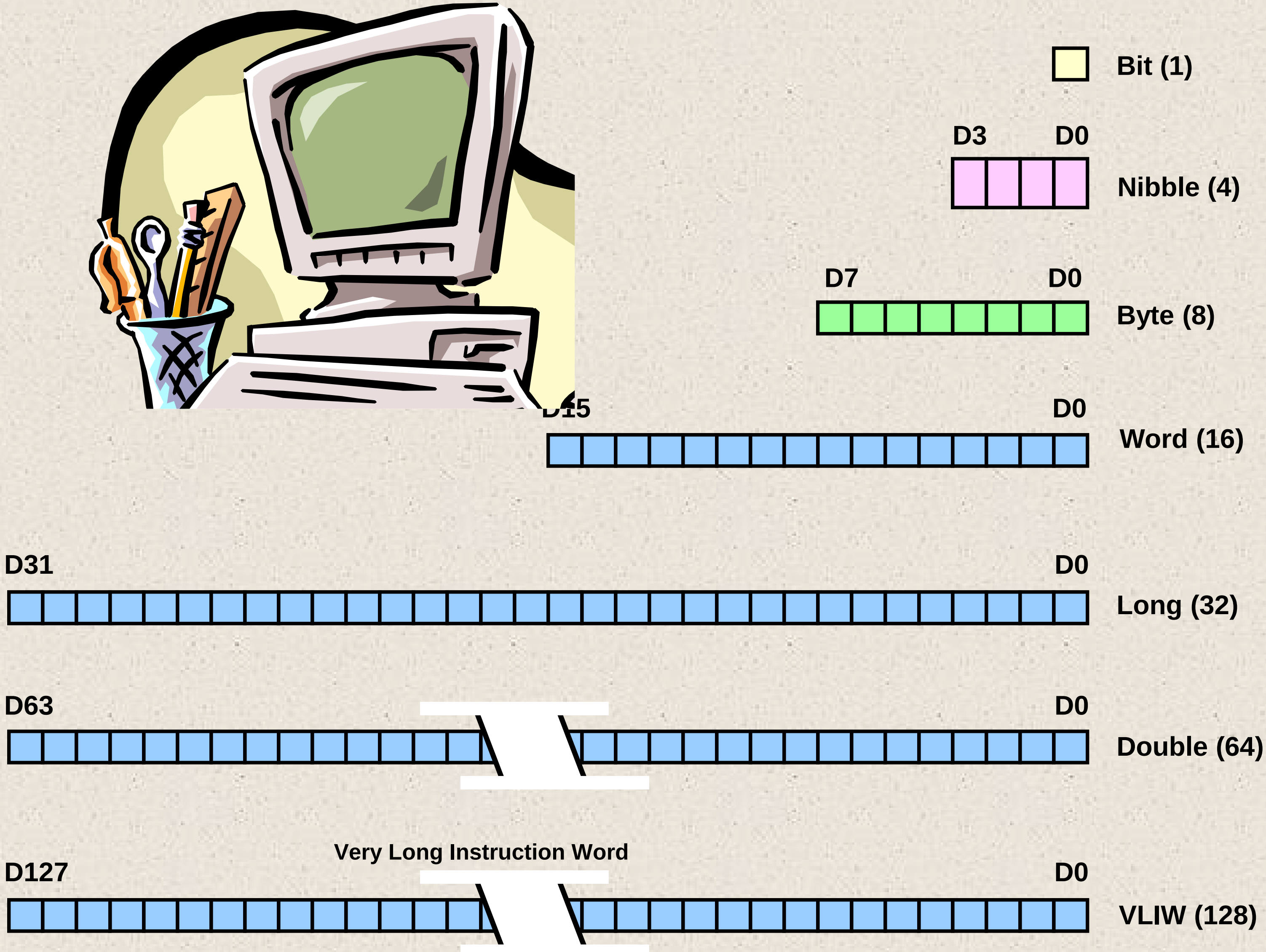
= 73,503

# Let’s do a 16-bit number

- Binary: 0101111111010111
- Octal: 0 101 111 111 010 111 = 057727 (group by threes )
- Hex: 0101 1111 1101 0111 = 5FD7 (group by fours)
- Decimal: ???

- Exercise: Convert $5DE37A05_{16}$ to Octal

# Bits, Bytes, Nibbles, Words, etc.



# Size of Numbers and C++

| Binary Digits | Architectural | C++ | Possible unsigned number range |
|---|---|---|---|
| 1 | bit | Boolean | 0, 1 |
| 4 | nibble | N/A | 0 ~ 15 |
| 8 | byte | char | 0 ~ 255 |
| 16 | word | short | ? |
| 32 | long | int | ? |
| 64 | double | double | ? |

# Engineering Notation

- In order to represent very large or very small numbers, we usually resort to ***scientific notation:***
    - For example: *Avogadro's Number = 6.022 x 10* $^{23}$
        - Mantissa = 6.022, exponent = 23
- It is common in engineering to use a shorthand version of scientific notation
- Replacement values

| | |
|---|---|
| **TERA = $10^{12}$ (T)** | **PICO = $10^{-12}$ (p)** |
| **GIGA = $10^{9}$ (G)** | **NANO = $10^{-9}$ (n)** |
| **MEGA = $10^{6}$ (M)** | **MICRO = $10^{-6}$ (µ)** |
| **KILO = $10^{3}$ (K)** | **MILLI = $10^{-3}$ (m)** |

# Computers and Numbers

- In the digital world
  - 1K means 1,024, or $2^{10}$
  - 1M means 1,048576, or $2^{20}$
  - 1G means 1,073,741,824, or $2^{30}$
- Example
  - 512 megabytes of memory really means 512 x ($2^{20}$ ) bytes, or $2^{29}$ bytes of memory
- In general, ***anything to do with the size in bytes*** uses ***computer-speak*** K,M,G
  - Anything else, such as clock speed or time, use standard units

# Negative Integer in Binary

- Subtraction in a processor is done by **changing positive numbers to negative number** and then **adding** them

- A processor **always** assumes "**an addition is on signed numbers**"

How to represent a negative integer in a computer system?

# Negative Integer – Two's Complement

- **Define the negative number as a complement number**
- 10's complement of 1718 in a 4-decimal system

  $10^4 - 1718 = 10000 - 1718 = 8282.$

  8282 is 10's complement of 1718 in a 4-decimal system

- 2's complement of 7 in an 8-bit system

  $2^8 - 7 = 1\,0\,0\,0\,0\,0\,0\,0\,0_2 - 0\,0\,0\,0\,0\,1\,1\,1_2 = 1\,1\,1\,1\,1\,0\,0\,1_2$

  Define $1\,1\,1\,1\,1\,0\,0\,1_2$ as a negative integer of 7 ($0\,0\,0\,0\,0\,1\,1\,1_2$ )

- Subtraction is confusing? Then,

  $2^8 - 7 = (2^8 - 1) - 7 + 1$

  $= 11111111_2 - 111_2 + 1_2 = 11111000_2 + 1_2 = 11111001_2$

- **It's the same as "flip bits** (1's to zero, zero's to 1) **and add 1 at the end"**

# Negative Integer – Two's Complement

- How to get the **2's complement of K** in **n-bit system**
  - $2^n$ – K, or
  - ($2^n$ -1) – K + 1, or
  - **Flip** (Complement ) all bits of K, **then add 1**

- Example: In an 8-bit system, compute the 2's complement of 0x5E
  - Step 1: Convert to binary: 0x5E = 0101 1110
  - Step 2: Flip bits → 1010 0001
  - Step 3: Add 1 → = 1010 0001 + 1 = 1010 0010
  - Step 4: Convert to hex again: $1010\ 0010_2$ → 0xA2

# Signed Number Range

- Signed number in a computer system
  - A negative number is in the format of 2’s complement
- Two’s complement negative numbers imply
  - All arithmetic operations are converted to addition
  - The **MSB** is ***always*** 1 if it is a ***negative number*** (zero is positive)
  - Range of an n-bit number is $\mathbf{-2^{n-1}}$ **to +** $\mathbf{(2^{n-1}-1)}$
  - E.g., range of 4-bit numbers is $-2^3$ to $+(2^3-1)$ ➔ -8 to 7

- Exercise: in a 4-bit system
  - What is the signed number $1000_2$ in decimal?
    - The MSB is 1, so it is a negative number
    - 2’s complement of $1000_2$ is $0111_2 + 1 = 1000_2 = 8$
    - So it is -8

    We cannot have a positive 8 in 4-bit signed system.

    Similarly, we cannot have a positive 128 in 8-bit signed system.

# Repeat the question with 2's complement

- Question (in a 4-bit system)

1. How to represent zero? 0000 or 1000 ?

   0000 is zero, and 1000 is -8
2. How many unsigned numbers you can have in a 4-bit system?

   unsigned: 0000 to 1111 (0 to 15)
3. How many signed numbers (in 2's complement) you can have in a 4-bit system?

   Positive: 0000 to 0111 (0 to 7)

   Negative: 1000 to 1111 (-8 to -1): So, total 16 numbers
4. With this representation in a 4-bit system,
   1) What is 7 – 5 in binary?

   $0111_2 - 0101_2 = 0010_2$

   2) What is 7+ (-5) in binary?

   $0111_2 + 1011_2 = 1\ 0010_2$

   (Since it is a 4-bit system, the carry bit won't appear, so $0010_2 = 2_{10}$)

# Two's Complement Arithmetic

- Arithmetic ***overflow***
  - Adding **two positive** number **results in a negative** number
  - Adding **two negative** number **results in a positive** number
  - ***How can this be possible***?
- If the result is **out of range**, the computer results in an **incorrect answer**.

- For example, in a 4-bit system (range is -8 to 7)

  7 + 7 = **0** 1 1 1 + **0** 1 1 1 = **1** 1 1 0 (It is not 14 but -2) → incorrect, negative

  (-7) + (-8) = **1** 0 0 1 + **1** 0 0 0 = **1** 0 0 0 1 (it has a carry bit and the result is 1) → incorrect, positive

# Two's Complement Arithmetic

- In a 4-bit system (range -8 to 7)

1. 7 + (-3) = 4

|   |   |
|---|---|
|   | **0** 111 |
| + | **1** 101 |
| **carry = 1** | **0** 100 |

No overflow

carry-out bit is invisible.
Then the answer is 4 (correct)

2. (-7) + ( -8)

|   |   |
|---|---|
|   | **1** 001 |
| + | **1** 000 |
| **carry = 1** | **0** 001 |

Overflow (sign bit changed to 0)

***Incorrect result, error.***

# How to represent a real number in binary?

# From Real to Binary Numbers

- Let's convert decimal number **3.8125** to a binary number
  - **Integer** part: the same as the integer binary $11_2 = 3$
  - **Fractional** part:

1. **Multiply the fraction by two**
2. Write down the **integer part on right**
3. Repeat 1 and 2 **until there is no fractional part on left**
4. **Read the integer** part on right, from **top to bottom**

**0.8125 * 2 = 0.625 + 1**

**0.625*2 = 0.25 + 1**

**0.25 *2 = 0.5 + 0**

**0.5*2 = 0.0 + 1**

**0.0 STOP HERE**

$\mathbf{3.8125_{10} = 0011.1101_2}$

# From Binary to Real Numbers

Binary $I_m I_{m-1} \ldots I_1 I_0 . F_1 F_2 F_3 \ldots F_{n-1} F_n =$

Decimal $I*2^m + I*2^{m-1} + \ldots + I*2^0 + F*2^{-1} + F*2^{-2}$

$+ \ldots + F*2^{-(n-1)} + F*2^{-n}$

$10.101_2 = 1*2^1 + 0*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$

$= 2 + 0.5 + 0.125 = 2.625$

# Real Numbers and Errors

- Many fractions are **repeating infinitely**

  E.g., convert 0.6 to binary

| | Integer |
|---|---|
| 0.6 * 2 = 0.2 ----- | 1 |
| 0.2 * 2 = 0.4 ----- | 0 |
| 0.4 * 2 = 0.8 ----- | 0 |
| 0.8 * 2 = 0.6 ----- | 1 |
| **0.6 * 2 = 0.2 -----** | **1** |
| **0.2 * 2 = 0.4 -----** | **0** |
| **0.4 * 2 = 0.8 -----** | **0** |
| **0.8 * 2 = 0.6 -----** | **1** |

So, 0.6 → 0.1001100110011001….. (will be repeated infinitely)

# Rounding and Truncation

- **Keep the number of bits finite**
  - **Truncation**: The simplest technique – just drop unwanted bits

    E.g., 0.1101101 → 0.1101

  - **Rounding**: Better technique, but a bit complicated

  If the **value of the lost digits** is **greater than half of the least-significant bit of the retained digits**, add 1 to the LSB; otherwise drop.

  E.g., 0.1101101: If I want to lose the last three bits, what shall I do?

  0.1101101 = 0.1101 + 0.0000101

  → 0.1101 + 0.0001

  = 0.11**10**

LSB of retained digits: 0.0001 = $2^{-4}$
Lost digits: 0.0000101 = $(2^{-5} + 2^{-7}) > 2^{-4} / 2$

How to represent a real number in a computer system?

# Real Numbers in a Computer System

- Two main approaches: Fixed-point vs. Floating-point
- Fixed-point representation (**NOT used** now)
  - Divide the bits into **integer** part and **fraction** part
  - The “Point” is fixed
  - Easier but less flexible
- Floating-point representation (**IEEE standard**)
  - Divide the bits into **sign**, **exponent** and **mantissa**
  - The “Point” is floating
  - Match with scientific notation
  - Flexible but more complex

# Fixed-Point Representation

- Fixed-point representation
  - Divide the bits for integer part and fraction part
  - For example, $3.625_{10} = 11.101_2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |

**Integer part** **Fractional part**

- Not flexible
  - What if you really need to represent $1.984 * 10^{(-123)}$ in computer?
  - How many bits will be needed? ( more than 372 bits)

# Floating-Point Representation

- Floating-point representation
  - Divide the bits into **sign**, **exponent** and **mantissa**

- IEEE floating-point format
  1. IEEE **short real** or **single precision**: **32** bits

| Sign (1) | Exponent (8) | Mantissa(23) |
| --- | --- | --- |

  2. IEEE **long real** or **double precision**: **64** bits

| Sign (1) | Exponent (11) | Mantissa(52) |
| --- | --- | --- |

# Floating-Point Representation - Single Precision

Steps to convert a real number to IEEE **Single Precision** floating-point representation

1. Convert decimal to binary
2. Normalize: moving the point left or right
3. Add 127 to the exponent
4. Mantissa is the one **after the floating point** in the normalized form
   – If the mantissa part is less than 23 bits, add zeros at the end
5. Put the corresponding numbers into each field

IEEE Floating point converter:

http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

# Floating-Point Representation - Single Precision

- **32-bit (single precision)** format

| Sign (1) | Exponent (8) | Mantissa(23) |
|---|---|---|

- Let's represent a real number to floating-point format

  E.g., $\mathbf{-3.8125_{10}}$

  $= \mathbf{-11.1101_2}$ (note that the integer part is **not** 2's complement)

  $= \mathbf{-1}.11101 * 2^{\mathbf{1}}$ (**normalize**: scientific notation)

- **Sign** bit = 1, because this is a negative number
- **Exponent** bits = 1 + **127** (biased) = 128 = $10000000_2$
- **Mantissa** bits: 111 0100 0000 0000 0000 0000

  Therefore, the real number in floating-point representation is:

  1 100 0000 0 111 01 00 0000 0000 0000 $0000_2$ = $C0740000

# Floating-Point Representation - Single Precision

- **Normalization**
  - **"1" shall always appear as an integer part**
  - No need to represent this bit in the format -> save one bit

- **Biased exponent**
  - The exponent has 8 bits, meaning it can range from -127 to 127 (Here we assume that -128 will never appear).
  - Therefore, if we add 127 to the exponent, it will always be a non-negative number.
  - Assuming such a representation, 0~254 is then available for the exponent filed. How about 255?

# Floating-Point Representation - Double Precision

Steps to convert a real number to IEEE **Double Precision** floating-point representation

1. Convert decimal to binary
2. Normalize: moving the point left or right
3. Add **1023** to the exponent
4. Mantissa is the one **after the floating point** in the normalized form
   – If the mantissa part is less than 52 bits, add zeros at the end
5. Put the corresponding numbers into each field

IEEE Floating point converter:

http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

# Floating-Point Representation - Double Precision

- **64-bit (double precision)** format

| Sign (1) | Exponent (11) | Mantissa(52) |
|---|---|---|

- Let's represent a real number to floating-point format

  E.g., **$-3.8125_{10}$**

  = **$-11.1101_2$** (note that the integer part is **not** 2's complement)

  = **-1**.11101*$2^1$ (normalize: scientific notation)

- **Sign** bit = 1, since negative
- **Exponent** = 1 + **1023** (biased) = 1024 = $100\ 0000\ 0000_2$
- **Mantissa**: 1110 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

  Therefore in floating-point representation,

  1 100 0000 0000 1110 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 $0000_2$
  = $C00E800000000000

# More about real numbers

- Why using biased exponent?
  - **Effect**: changing **negative** exponent value **to positive** value
  - **Motivation**: for quick comparison (bit-by-bit) of two real numbers

- Why adding 127 for single-precision floating numbers?
  - **Effect**: positive numbers in the rage of **0 to 254**
  - **Motivation**: reserve 255 for **special number usage**

| Sign | Exponent ($e$) | Fraction ($f$) | Value |
|---|---|---|---|
| 0 | 00⋯00 | 00⋯00 | +0 |
| 0 | 00⋯00 | 00⋯01<br>⋮<br>11⋯11 | Positive Denormalized Real<br>$0.f \times 2^{(-b+1)}$ |
| 0 | 00⋯01<br>⋮<br>11⋯10 | XX⋯XX | Positive Normalized Real<br>$1.f \times 2^{(e-b)}$ |
| 0 | 11⋯11 | 00⋯00 | $+\infty$ |
| 0 | 11⋯11 | 00⋯01<br>⋮<br>01⋯11 | SNaN |
| 0 | 11⋯11 | 1X⋯XX | QNaN |
| 1 | 00⋯00 | 00⋯00 | −0 |
| 1 | 00⋯00 | 00⋯01<br>⋮<br>11⋯11 | Negative Denormalized Real<br>$-0.f \times 2^{(-b+1)}$ |
| 1 | 00⋯01<br>⋮<br>11⋯10 | XX⋯XX | Negative Normalized Real<br>$-1.f \times 2^{(e-b)}$ |
| 1 | 11⋯11 | 00⋯00 | $-\infty$ |
| 1 | 11⋯11 | 00⋯01<br>⋮<br>01⋯11 | SNaN |
| 1 | 11⋯11 | 1X⋯XX | QNaN |

- **NaN**: Not A Number
- **QNaN**: **Quiet** NaN
  - generated from an operation when the result is not mathematically defined
  - denote ***indeterminate*** operations
- **SNaN**: **Signaling** NaN
  - used to signal an exception when used in operations
  - can be to assign to uninitialized variables to trap premature usage
  - denote ***invalid*** operations