中国科学技术大学
University of Science and Technology of China

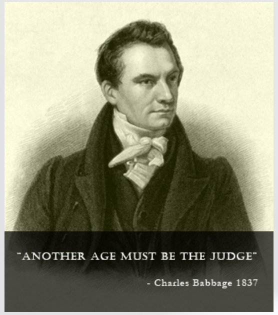
计算系统概论A
Introduction to Computing Systems
( CS1002A.03)

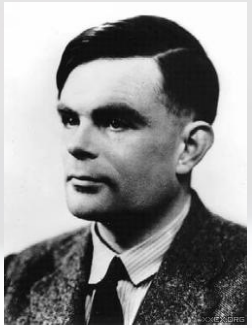
# Chapter 2
# Bits, Data Types, and Operations

陈俊仕
cjuns@ustc.edu.cn
2023 Fall

计算机科学与技术学院
School of Computer Science and Technology

# Previously: from mechanical computer to electronic computer

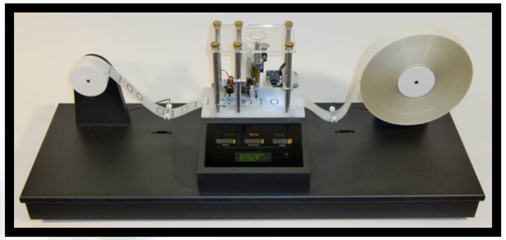


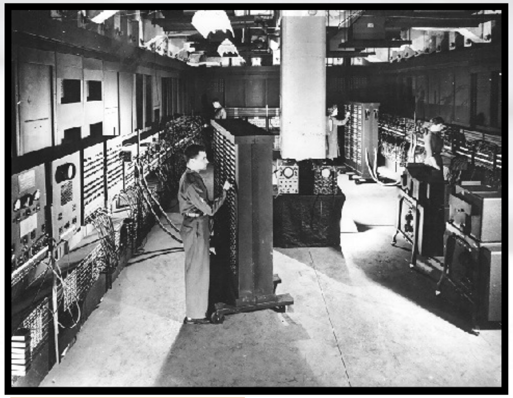



Eckert(24) and Mauchly(36)

**Charles Babbage,**
**1791 – 1871,England**

**Alan Turing(24)**





**1832,2002,2008**
**The Babbage Difference Engine, 17 years, 25,000 parts, 5ton, cost: £17,470**

**Turing Machine, 1936**

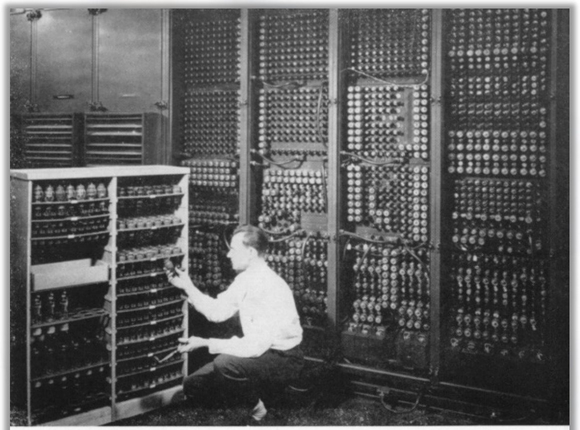
**ENIAC**

**1946**

## Previously： First computer vs. First microprocessor chip

### 1946，ENIAC(Electrical Numerical Integrator And Calculator )

- 18000 vacuum tubes
- 1500 relays
- 174 KW
- 30 tons
- 1800 sq. ft. footprint
- Clock: 100kHz
- RAM: ~230bytes
- IO: punched card

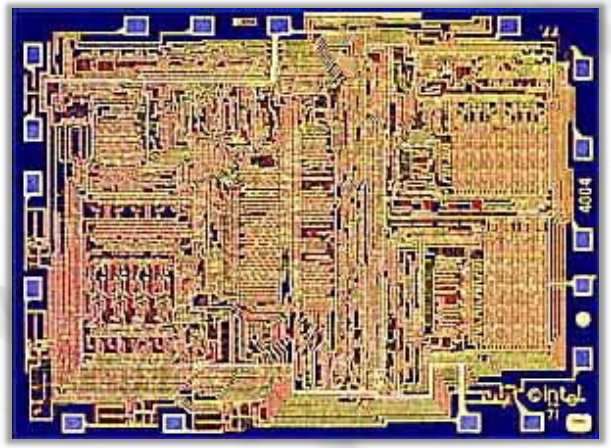


Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

After 25 years

### 1971, Intel 4004

- 10 micron process, NMOS-Only Logic
- 2,250 transistors
- 3cmx4cm die
- 4-bit bus
- Performance < 0.1 MIPS
- 640 bytes of addressable Memory
- 740 KHz



# Previously： Thirty years after the first microprocessor chip was born

## 1971, Intel 4004

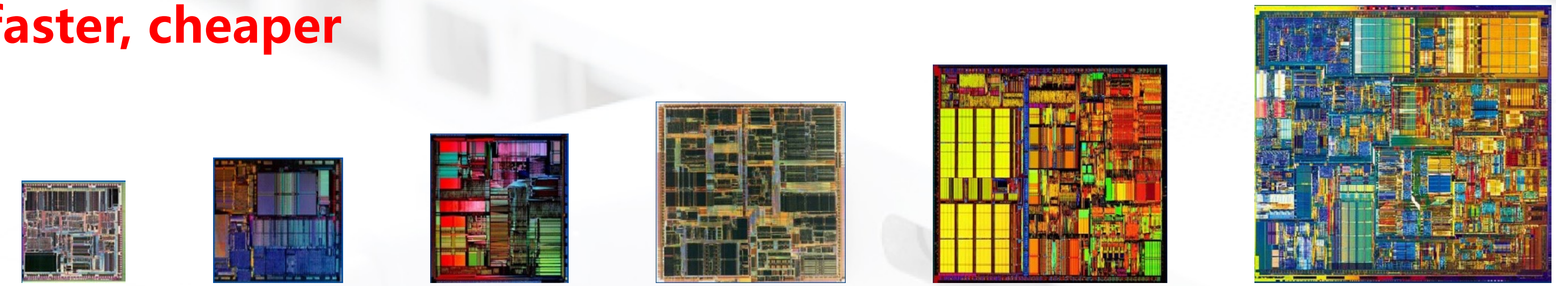
- 10 micron process
- 2,300 transistors
- 3x4 mm die
- 4-bit bus
- 640 bytes of addressable Memory
- 750 KHz

After 30 years

## 2000, Intel Pentium IV

- Issues up to 5 uOPs per cycle
- MMX, SSE, and SSE2
- 0.18 micron process
- 42 million transistors
- 217 mm die
- 64-bit bus
- 8KB D-cache, 12KB op trace cache (I-cache), 256KB L2 cache
- 1.4 GHz

**Performance improved 5000x: smaller, faster, cheaper**



# Previously： 人类如何实现从物理设备到问题求解的？

Application

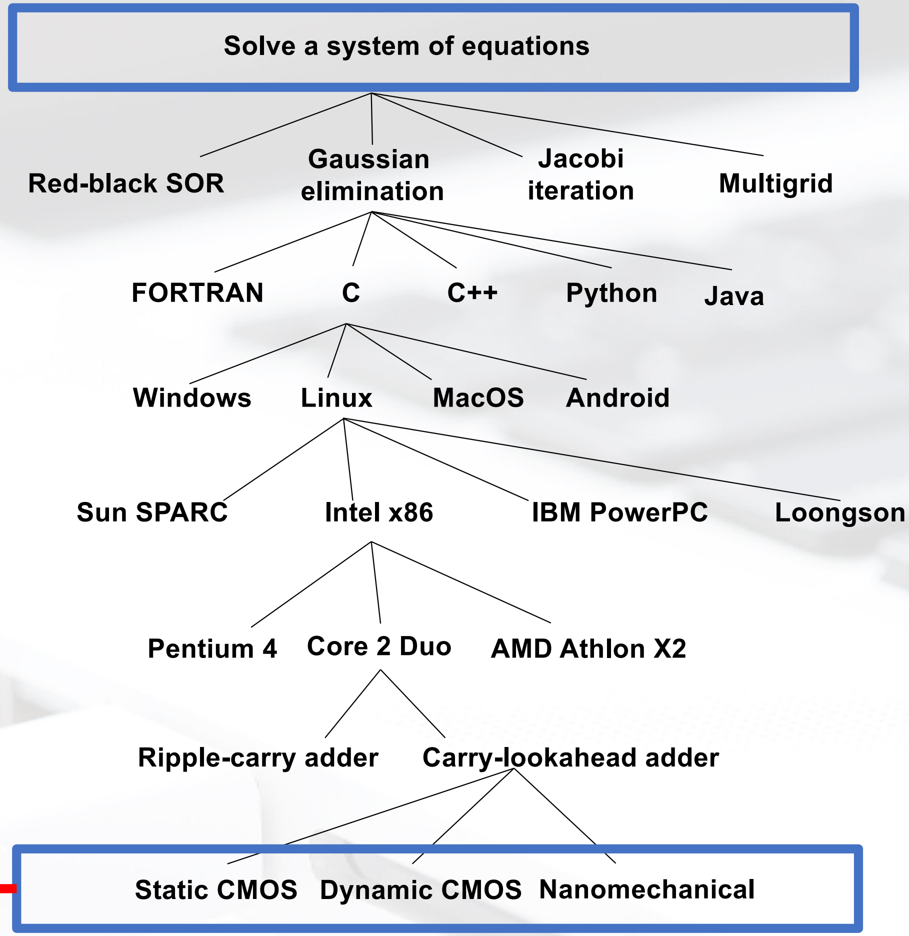

差距太大，
一步无法跨越！

也有例外，
例如罗盘

Physics

# Previously : Many Choices at Each Level

- Application
- Algorithm and Data Structure
- Programming Language/Compiler
- Operating System/Virtual Machines
- Instruction Set Architecture (ISA)
- Microarchitecture
- Gates/Register-Transfer Level (RTL)
- Analog/Digital Circuits
- Electronic Devices
- Physics



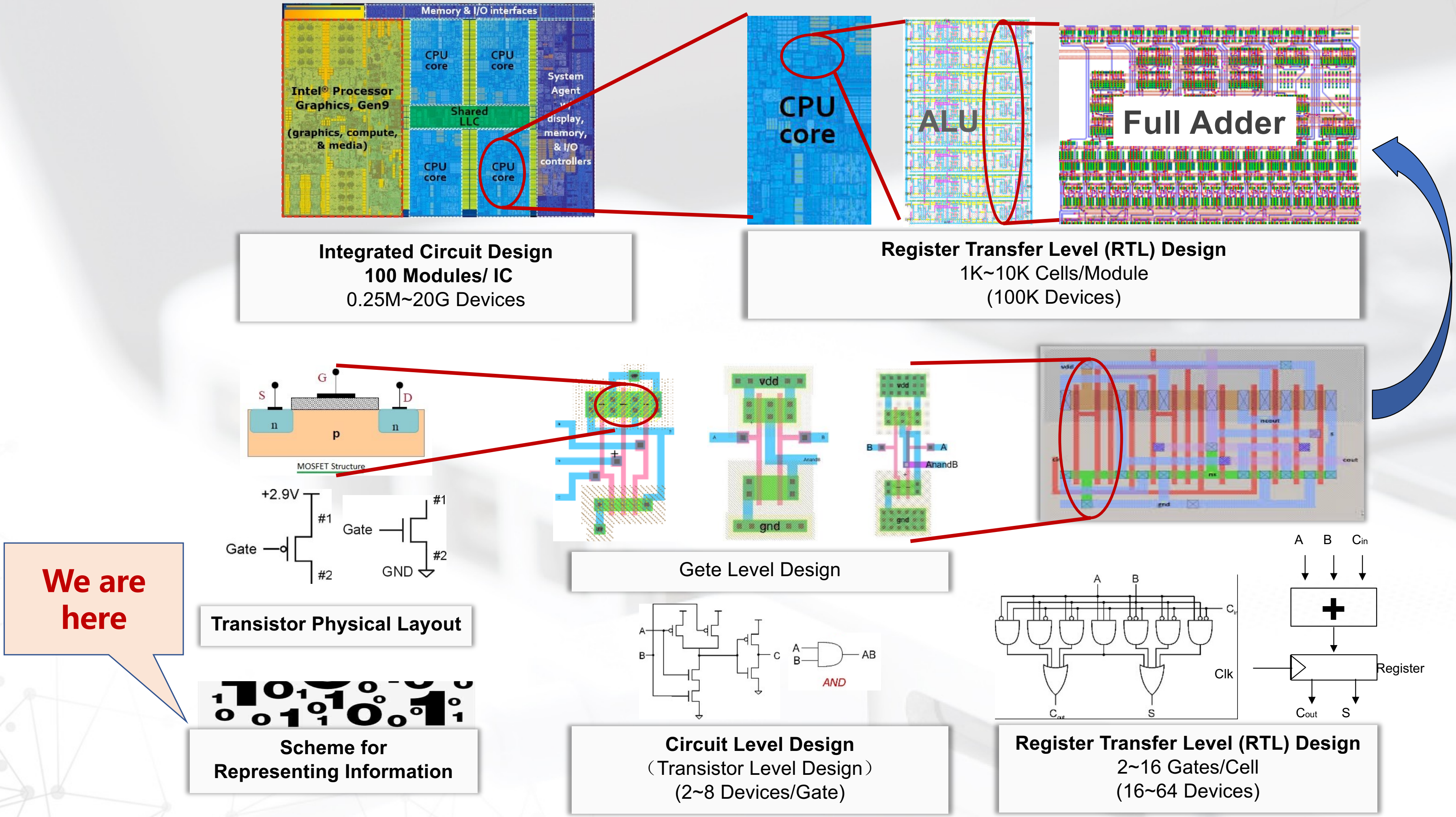
# Previously: Abstraction helps us Manage Complexity

| | USTC Courses |
|---|---|
| **Application** | |
| Algorithm and Data Structure | 算法基础/数据结构 |
| Programming Language/Compiler | 程序设计/编译技术 |
| Operating System/Virtual Machines | 操作系统/虚拟机 |
| **Instruction Set Architecture (ISA)** | 计算机组成 |
| Microarchitecture | 计算机组成 |
| Gates/Register-Transfer Level (RTL) | 数字逻辑 |
| Analog/Digital Circuits | 集成电路 |
| Electronic Devices | 微电子 |
| **Physics** | |

需要一门**贯通课程**，帮助学生从底层物理到高层应用，整体上理解计算机系统。

**从广义上讲，计算机系统结构是抽象层次的设计，它允许我们使用可用的制造技术有效地实现信息处理应用程序。**

# Today



Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
Shared LLC
System Agent
display, memory, & I/O controllers
CPU core
ALU
Full Adder
Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices
Register Transfer Level (RTL) Design
1K~10K Cells/Module
(100K Devices)
G
S
D
n
p
n
MOSFET Structure
+2.9V
#1
Gate
#2
Gate
#1
#2
GND
vdd
gnd
A
AandB
B
A
AandB
Gete Level Design
We are here
Transistor Physical Layout
Scheme for Representing Information
A
B
C
A
B
AB
AND
Circuit Level Design
（Transistor Level Design）
(2~8 Devices/Gate)
A
B
S
Clk
A B Cin
+
Register
Cout S
Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)

# Outline

1. How do we represent information in a computer?
2. Integer Data Types
3. 2’ Complement Integers
4. Binary-Decimal Conversion
5. Operations on Bits: Arithmetic and Logical
6. Other Representation

# Outline

# A Computer looks like......

**Hardware and Software**

**3 + 5 = ?**
**3 > 5 ?**

人类大脑与处理器芯片的比较

| | 器件数量 | 功耗 | 频率 | 通信 | 体积 | 能力 |
|---|---|---|---|---|---|---|
| 人类大脑 | $10^{10}\sim10^{11}$个神经元 | 20W | 100Hz | $10^{14}$个突触 | $1.4\text{x}10^{3}\text{cm}^{3}$ | 可处理数学上无法严格定义的问题 |
| 处理器芯片 | $10^{11}$个晶体管 | 40W | 109Hz | 稀疏互连网络 | $3.2\text{cm}^{3}$ | 只能处理数学上严格定义的问题 |

刘宇航, ***冯·诺伊曼《计算机与人脑》要点归纳及启发***, 中国计算机学会通讯, 2018

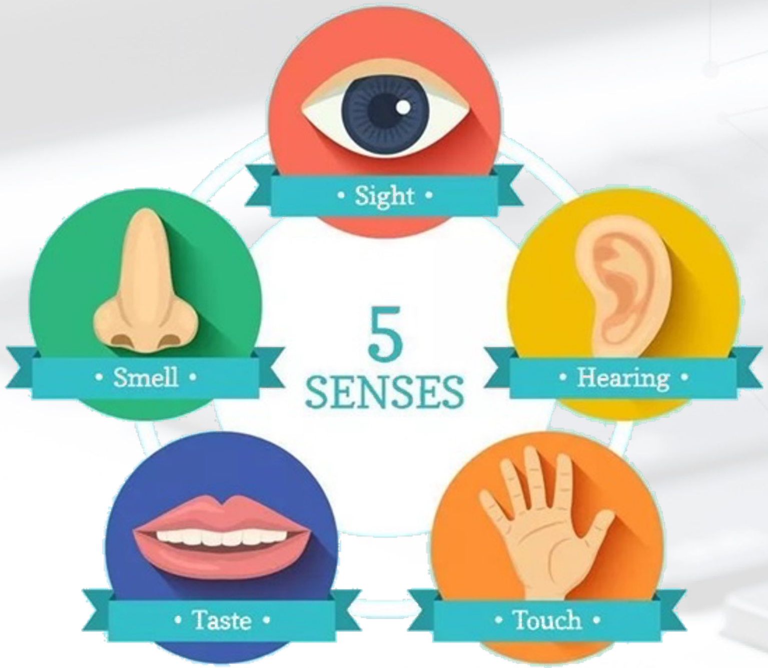
# *The Computer and the Brain* – *by Neumann J V.*


| **《计算机与人脑》** | **第二部分 人脑** |
|---|---|
| **引言** | 第八章 神经元功能简述 |
| **第一部分 计算机** | 第九章 神经脉冲的本质 |
| 第一章 模拟方法 | 第十章 刺激的判据 |
| 第二章 数字方法 | 第十一章 神经系统内的记忆问题 |
| 第三章 逻辑控制 | 第十二章 神经系统的数字部分和模拟部分 |
| 第四章 混合数字方法 | 第十三章 代码及其在机器功能的控制中之作用 |
| 第五章 准确度 | 第十四章 神经系统的逻辑结构 |
| 第六章 现代模拟计算机的特征 | 第十五章 使用的记数系统之本质：它不是数字的而是统计的 |
| 第七章 现代数字计算机的特征 | 第十六章 人脑的语言不是数学的语言 |


刘宇航, ***冯·诺伊曼《计算机与人脑》要点归纳及启发***, 中国计算机学会通讯, 2018

# 5 Senses of Human

- Sight
  - Image, picture, photo, vedio, …
- Hearing
  - Sound, voice, speech, music, …
- Touch
  - Shape, soft, hard, hurt, numb, …
- Taste
  - Sour, sweet, bitter, spicy, salty, …
- Smell
  - Sweet, smelly, …




to record by number, data, words, symbols, text, language, ……

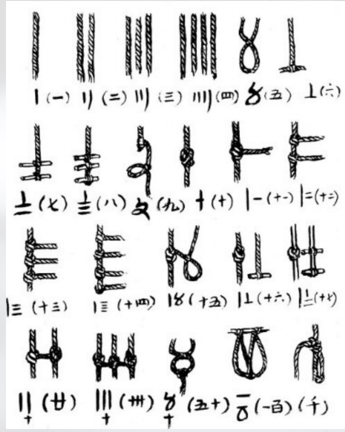
# What kinds of information do we need to represent?

- Kinds of Information
  - Numbers – natural number, integers, positive/negative integers, integers/decimals, real, complex, rational, irrational, signed, unsigned, floating point, …
  - Text – characters, strings, …
  - Logical – true, false
  - Images – pixels, colors, shapes, …
  - Sound – sound of talk, sound of sing, …
  - Video – a series of images
  - Instructions – plus(+), minus(-), times (*) ,divided by(/) , …
  - …
- Data type: *representation* and *operations* within the computer

We’ ll start with numbers…

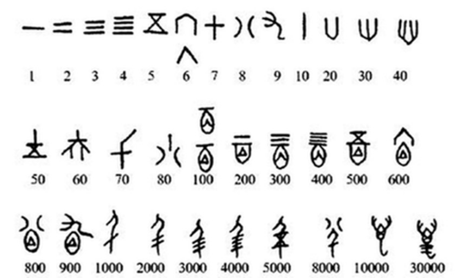
# Number Notation

Counting stone(石头 )



Counting rod(算筹)

Knotting(结绳 )




Inscriptions on oracle bones
（甲骨文上刻字）

# Number Notation

## Non-positional notation(like to counting rod)

- Could represent a number ("5") with a string of ones ("11111")

problems?

5

V

11111

# Number Notation

## ■ Weighted positional notation

- decimal numbers(denary numbers): "329"
- "3" is worth 300, because of its position(with place value 100),
- while "9" is only worth 9, because of its position(with place value 1)

**329**

$10^2$ $10^1$ $10^0$

$\boxed{3x100 + 2x10 + 9x1 = 329}$

Denary numbers
**base** is 10,
**place value** according its position

# Denary numbers - base ten

- $(5346)_{10}$

**5346**

| Available digit | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 | | | |
|---|---|---|---|---|
| Place value | $10^3=1000$ | $10^2=100$ | $10^1=10$ | $10^0=1$ |
| Digit | 5 | 3 | 4 | 6 |
| Product of digit and place value | 5x1000=5000 | 3x100=300 | 4x10=40 | 6x1=6 |

$(5346)_{10}$ = 5x1000+3x100+4x10+6x1

# Octonary numbers - base eight

■ $(5346)_8$

**5346**

| Available digit | 0, 1, 2, 3, 4, 5, 6, 7 | | | |
|---|---|---|---|---|
| **Place value** | $8^3$=512 | $8^2$=64 | $8^1$=8 | $8^0$=1 |
| **Digit** | 5 | 3 | 4 | 6 |
| **Product of digit and place value** | 5x512 | 3x64 | 4x8 | 6x1 |

$(5346)_8$ = 5x512+3x64+4x8+6x1=$(2790)_{10}$

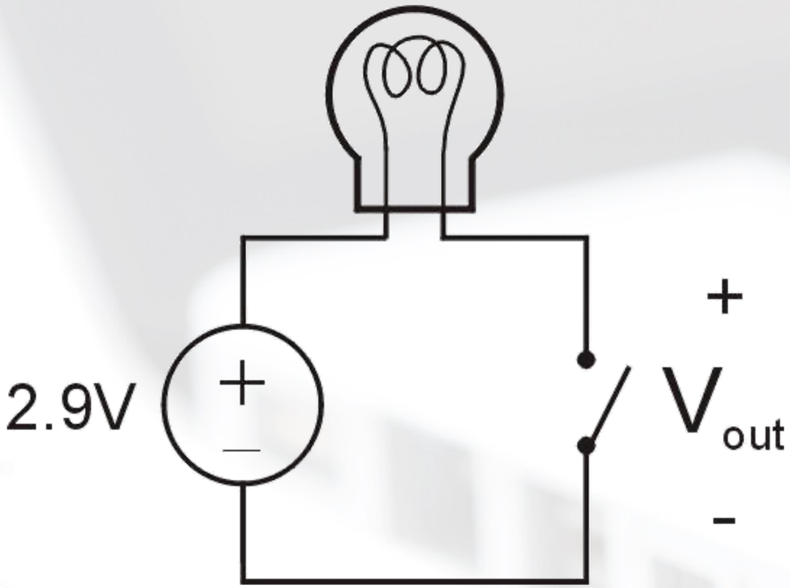
$(75)_8$ = 7x8+5x1=$(61)_{10}$

$(31276)_8$ = 3x4096+1x512+2x64+7x8+6x1=$(12990)_{10}$

## How do we represent data in a computer?

- At the lowest level, a computer is an electronic machine.
  - works by controlling *the flow of electrons*
- Easy to recognize two conditions:
  - presence of a voltage – we'll call this state "1"
  - absence of a voltage – we'll call this state "0"
- Could base state on *value* of voltage, but control and detection circuits more complex.
  - compare turning on a light switch to measuring or regulating voltage
- We' ll see examples of these circuits in the next chapter.

# Simple Switch Circuit




## Switch open:

- No current through circuit
- Light is off
- $V_{out}$ is +2.9V

## Switch closed:

- Short circuit across switch
- Current flows
- Light is on
- $V_{out}$ is 0V

***Switch-based circuits*** can easily represent two states:
on/off, open/closed, voltage/no voltage.

# Computer is a binary digital system

**Digital system:**

- finite number of symbols

**Binary (base two) system:**

- has two states: 0 and 1

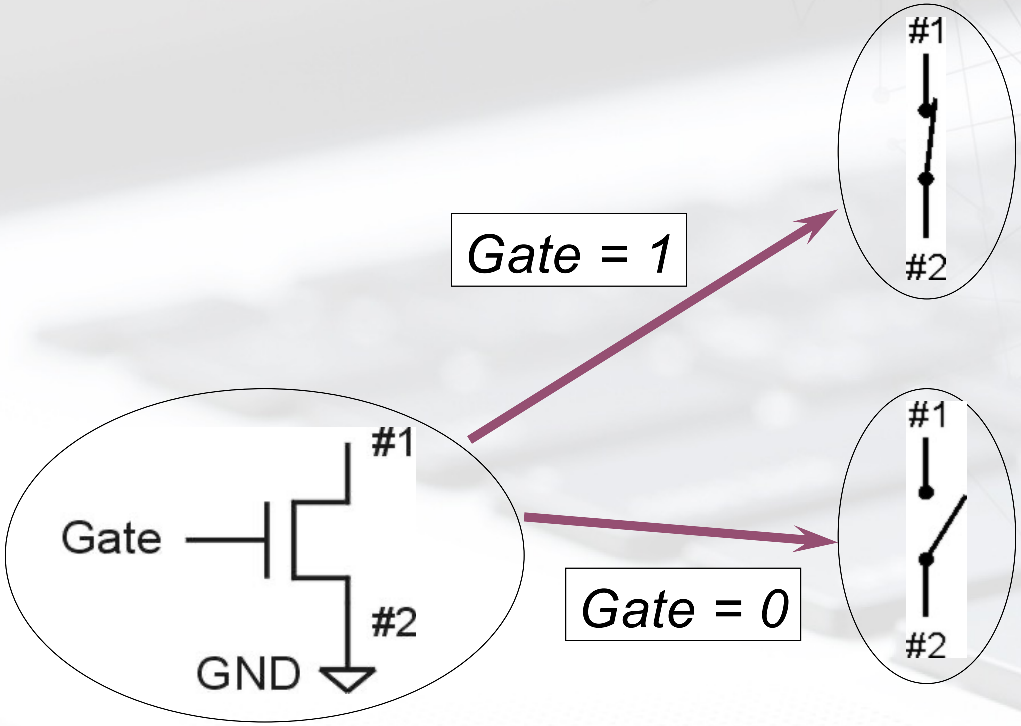

**Basic unit of information is the *binary digit*, or *bit*.**

**Values with more than two states require multiple bits.**

- **A collection of two bits has four possible states:** **00, 01, 10, 11**
- **A collection of three bits has eight possible states:** **000, 001, 010, 011, 100, 101, 110, 111**
- ***A collection of n bits has $2^n$ possible states.***

# N-type MOS Transistor

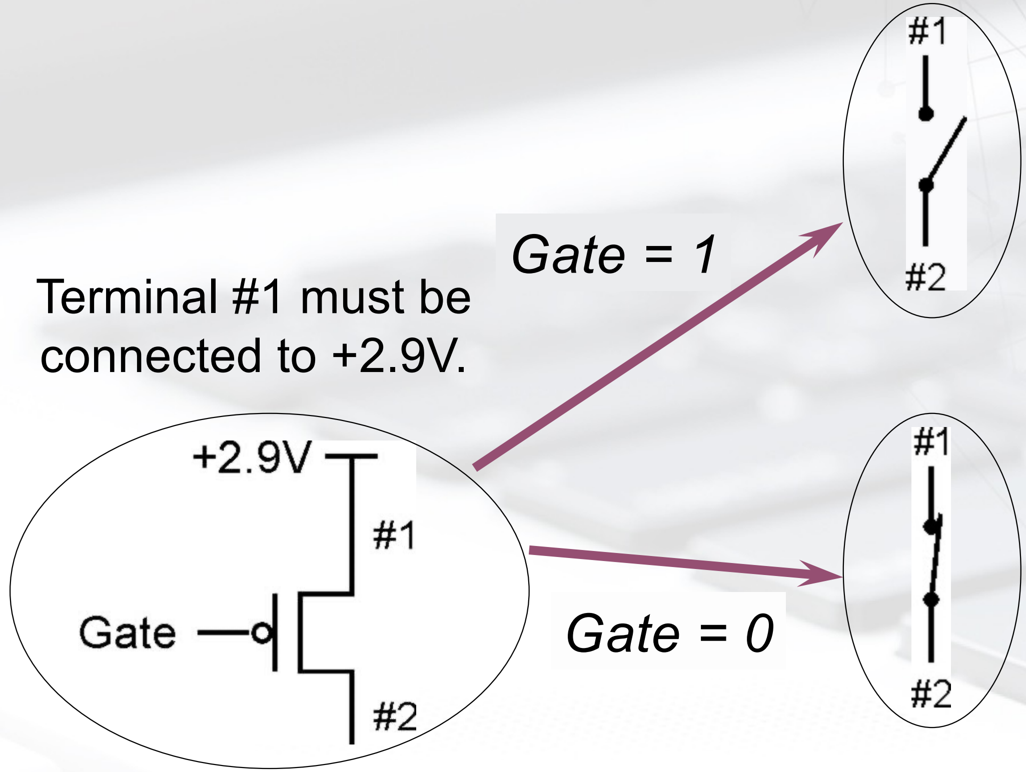
- **MOS = Metal Oxide Semiconductor**
  - two types: N-type and P-type
- **N-type**
  - when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
  - when Gate has zero voltage, open circuit between #1 and #2 (switch open)




Terminal #2 must be connected to GND (0V).

# P-type MOS Transistor

- **P-type** is *complementary* to N-type
  - when Gate has positive voltage, open circuit between #1 and #2 (switch open)
  - when Gate has zero voltage, short circuit between #1 and #2 (switch closed)



# Logic Gates

- Use switch behavior of MOS transistors to implement logical functions: **AND, OR, NOT.**
- Digital symbols:
  - recall that we assign a range of analog voltages to each digital (logic) symbol


  - assignment of voltage ranges depends on electrical properties of transistors being used
  - typical values for "1": +5V, +3.3V, +2.9V, +1.1V for purposes of illustration, we'll use +2.9V

# Binary numbers - base two

- $(101110)_2$

**10 1110**

| Available digit | 0, 1 | | | | | |
|---|---|---|---|---|---|---|
| **Place value** | $2^5=32$ | $2^4=16$ | $2^3=8$ | $2^2=4$ | $2^1=2$ | $2^0=1$ |
| **Digit** | 1 | 0 | 1 | 1 | 1 | 0 |
| **Product of digit and place value** | 32 | 0 | 8 | 4 | 2 | 0 |

$(101110)_2 = 1\text{x}32+0\text{x}16+1\text{x}8+1\text{x}4+1\text{x}2+0\text{x}1 = (46)_{10}$

$(11110100)_2 = 1\text{x}128+1\text{x}64+1\text{x}32+1\text{x}16+0\text{x}8+1\text{x}4+0\text{x}2+0\text{x}1=(244)_{10}$

$(2790)_{10} = ($ ? $)_2$

$(5346)_{10} = ($ ? $)_2$

# Outline

# Unsigned Integers

- An $n$-bit unsigned integer represents $2^n$ values: from 0 to $2^n$-1.

| $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

# Unsigned Binary Arithmetic

- **Base-2 addition – just like base-10!**
  - add from right to left, propagating carry

carry

```
  10010        10010        1111
+  1001      +  1011      +    1
  11011        11101       10000

               10111
             +   111
```

- Subtraction, multiplication, division,…

# Signed Integers

- **With n bits, we have $2^n$ distinct values.**
  - assign about half to positive integers (1 through $2^{n-1}$) and about half to negative ($-2^{n-1}$ through -1)
  - that leaves two values: one for 0, and one extra
- **Positive integers**
  - just like unsigned – zero in Most Significant (MS) bit
    00101 = 5
- **Negative integers**
  - sign-magnitude – set top bit to show negative, other bits are the same as unsigned
    10101 = -5
  - one's complement – flip every bit to represent negative
    11010 = -5
  - in either case, MS bit indicates sign: 0=positive, 1=negative

# Three representations of signed integers

| Representation | | | | | Value Represented | | |
|---|---|---|---|---|---|---|---|
| | | | | | Signed Magnitude | 1's Complement | 2's Complement |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 2 | 2 | 2 |
| 0 | 0 | 0 | 1 | 1 | 3 | 3 | 3 |
| 0 | 0 | 1 | 0 | 0 | 4 | 4 | 4 |
| 0 | 0 | 1 | 0 | 1 | 5 | 5 | 5 |
| 0 | 0 | 1 | 1 | 0 | 6 | 6 | 6 |
| 0 | 0 | 1 | 1 | 1 | 7 | 7 | 7 |
| 0 | 1 | 0 | 0 | 0 | 8 | 8 | 8 |
| 0 | 1 | 0 | 0 | 1 | 9 | 9 | 9 |
| 0 | 1 | 0 | 1 | 0 | 10 | 10 | 10 |
| 0 | 1 | 0 | 1 | 1 | 11 | 11 | 11 |
| 0 | 1 | 1 | 0 | 0 | 12 | 12 | 12 |
| 0 | 1 | 1 | 0 | 1 | 13 | 13 | 13 |
| 0 | 1 | 1 | 1 | 0 | 14 | 14 | 14 |
| 0 | 1 | 1 | 1 | 1 | 15 | 15 | 15 |

**Signed Magnitude:**

**5 - 5 = 5 + (-5) =-10**

```
   00101    (5)
+  10101     (-5)
   11010  (-10)
```

**1's Complement:**

**5 - 5 = 5 + (-5) =- 0**

```
   00101    (5)
+  11010    (-5)
   11111    (-0)
```

# Three representations of signed integers

| Representation | | | | | Value Represented | | |
|---|---|---|---|---|---|---|---|
| | | | | | Signed Magnitude | 1's Complement | 2's Complement |
| 1 | 0 | 0 | 0 | 0 | −0 | −15 | −16 |
| 1 | 0 | 0 | 0 | 1 | −1 | −14 | −15 |
| 1 | 0 | 0 | 1 | 0 | −2 | −13 | −14 |
| 1 | 0 | 0 | 1 | 1 | −3 | −12 | −13 |
| 1 | 0 | 1 | 0 | 0 | −4 | −11 | −12 |
| 1 | 0 | 1 | 0 | 1 | −5 | −10 | −11 |
| 1 | 0 | 1 | 1 | 0 | −6 | −9 | −10 |
| 1 | 0 | 1 | 1 | 1 | −7 | −8 | −9 |
| 1 | 1 | 0 | 0 | 0 | −8 | −7 | −8 |
| 1 | 1 | 0 | 0 | 1 | −9 | −6 | −7 |
| 1 | 1 | 0 | 1 | 0 | −10 | −5 | −6 |
| 1 | 1 | 0 | 1 | 1 | −11 | −4 | −5 |
| 1 | 1 | 1 | 0 | 0 | −12 | −3 | −4 |
| 1 | 1 | 1 | 0 | 1 | −13 | −2 | −3 |
| 1 | 1 | 1 | 1 | 0 | −14 | −1 | −2 |
| 1 | 1 | 1 | 1 | 1 | −15 | −0 | −1 |

**Signed Magnitude:**

5 - 5 = 5 + (-5) =-10

```
   00101     (5)
+  10101      (-5)
   11010  (-10)
```

**1's Complement:**

5 - 5 = 5 + (-5) =- 0

```
   00101     (5)
+  11010    (-5)
   11111    (-0)
```

**2's Complement:**

5 - 5 = 5 + (-5) = 0

```
   00101      (5)
+  11011       (-5)
   00000      (0)
```

# Outline

# Two’ s Complement Representation

- If number is positive or zero,
  - normal binary representation, zeroes in upper bit(s)
- If number is negative,
  - start with positive number
  - flip every bit (i.e., take the one’s complement)
  - then add one

| | | | | | |
|---|---|---|---|---|---|
| | 00101 | (5) | | 01001 | (9) |
| | 11010 | (1's comp) | | | (1's comp) |
| + | 1 | | + | 1 | |
| | 11011 | (-5) | | | (-9) |

# Two’ s Complement

## Problems with sign-magnitude and 1’ s complement

- two representations of zero (+0 and −0)
- arithmetic circuits are complex
  - How to add two sign-magnitude numbers?
    - e.g., try 2 + (-3)
  - How to add two one’s complement numbers?
    - e.g., try 4 + (-3)

## *Two’ s complement* representation developed to make circuits easy for arithmetic.

- for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with “normal” addition, ignoring carry out

| | | | | | |
|---|---|---|---|---|---|
| | 00101 | (5) | | 01001 | (9) |
| + | 11011 | (-5) | + | _____ | (-9) |
| | 00000 | (0) | | 00000 | (0) |

# Two's Complement Shortcut

## To take the two's complement of a number:

- copy bits from right to left until (and including) the first "1"
- flip remaining bits to the left

```
  011010000
  100101111   (1's comp)
+         1
  ---------
  100110000
```

```
0110 10000
(flip)  (copy)
1001 10000
```

# Two's Complement Signed Integers

- MS bit is sign bit – it has weight $-2^{n-1}$.
- Range of an n-bit number: $-2^{n-1}$ through $2^{n-1} - 1$.
  - The most negative number ($-2^{n-1}$) has no positive counterpart.

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |

## Question

- Why is 1's complement called 1's complement?

- Why is 2's complement called 2's complement?

# Outline

# Converting Binary (2’ s C) to Decimal

1. If leading bit is one, take two’s complement to get a positive number.
2. Add powers of 2 that have “1” in the corresponding bit positions.
3. If original number was negative, add a minus sign.

$$X = 01101000_{two}$$
$$= 2^6+2^5+2^3 = 64+32+8$$
$$= 104_{ten}$$

*Assuming 8-bit 2’s complement numbers.*

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

# More Examples

$$X = 00100111_{two}$$
$$= 2^5+2^2+2^1+2^0 = 32+4+2+1$$
$$= 39_{ten}$$

$$X = 11100110_{two}$$
$$-X = 00011010$$
$$= 2^4+2^3+2^1 = 16+8+2$$
$$= 26_{ten}$$
$$X = -26_{ten}$$

*Assuming 8-bit 2's complement numbers.*

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

# Converting Decimal to Binary (2’s C)

## First Method: *Division*

1. Divide by two – remainder is least significant bit.
2. Keep dividing by two until answer is zero, writing remainders from right to left.
3. Append a zero as the MS bit; if original number negative, take two’s complement.

$X = 104_{ten}$

| | | | |
|---|---|---|---|
| 104/2 | = | 52 r0 | *bit 0* |
| 52/2 | = | 26 r0 | *bit 1* |
| 26/2 | = | 13 r0 | *bit 2* |
| 13/2 | = | 6 r1 | *bit 3* |
| 6/2 | = | 3 r0 | *bit 4* |
| 3/2 | = | 1 r1 | *bit 5* |
| 1/2 | = | 0 r1 | *bit 6* |

$X = 01101000_{two}$

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

## Converting Decimal to Binary (2's C)

### Second Method: *Subtract Powers of Two*

1. Change to positive decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as MS bit; if original was negative, take two's complement.

| | | |
|---|---|---|
| $X = 104_{ten}$ | $104 - 64 = 40$ | *bit 6* |
| | $40 - 32 = 8$ | *bit 5* |
| | $8 - 8 = 0$ | *bit 3* |
| $X = 01101000_{two}$ | | |

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

# Outline

# Operations: Arithmetic and Logical

- Recall: a data type includes ***representation*** and ***operations***. We now have a good representation for signed integers, so let's look at some arithmetic operations:
  - **`Addition`**
  - **`Subtraction`**
  - **`Sign Extension`**
- We'll also look at overflow conditions for addition, multiplication, division, etc., can be built from these basic operations.
- Logical operations are also useful:
  - **`AND`**
  - **`OR`**
  - **`NOT`**

# Addition

- As we' ve discussed, 2' s comp. addition is just binary addition.
  - assume all integers have the same number of bits
  - ignore carry out
  - for now, assume that sum fits in n-bit 2's comp. representation

```
    01101000 (104)            11110110 (-10)
  + 11110000 (-16)         +           (-9)
    01011000 (88)                      (-19)
```

*Assuming 8-bit 2's complement numbers.*

# Subtraction

- **Negate subtrahend (2nd no.) and add.**
  - assume all integers have the same number of bits
  - ignore carry out
  - for now, assume that difference fits in n-bit 2's comp. representation

```
   01101000  (104)            11110110 (-10)
 - 00010000   (16)         -            (-9)

   01101000  (104)            11110110 (-10)
 + 11110000  (-16)         +             (9)
   01011000   (88)                     (-1)
```

*Assuming 8-bit 2's complement numbers.*

# Sign Extension

- To add two numbers, we must represent them with the same number of bits.
- If we just pad with zeroes on the left:

| **4-bit** | | **8-bit** | |
|---|---|---|---|
| `0100` | (4) | `00000100` | (still 4) |
| `1100` | (-4) | `00001100` | (12, not -4) |

- Instead, replicate the MS bit -- the sign bit:

| **4-bit** | | **8-bit** | |
|---|---|---|---|
| `0100` | (4) | `00000100` | (still 4) |
| `1100` | (-4) | `11111100` | (still -4) |

# What if too big/small?

- Integer and floating-point operations can lead to results too big/small to store within their representations: **overflow/underflow**
- Binary bit patterns are simply representatives of numbers. Abstraction!
  - Strictly speaking they are called "numerals".
- Numerals really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, *, / ) cannot be represented by these rightmost HW bits, we say **overflow** occurred

# Overflow

- If operands are too big, then sum cannot be represented as an *n*-bit 2’ s comp number.

| | | | |
|---|---|---|---|
| 01000 | (8) | 11000 | (-8) |
| + 01001 | (9) | + 10111 | (-9) |
| 10001 | (-15) | 01111 | (+15) |

- We have overflow if:
  - signs of both operands are the same, and
  - sign of sum is different.

- Another test -- easy for hardware:
  - carry into MS bit does not equal carry out

# Logical Operations

- Operations on logical TRUE or FALSE
  - `two states -- takes one bit to represent: TRUE=1, FALSE=0`
- View *n*-bit number as a collection of *n* logical values
  - `operation applied to each bit independently`

| A | B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| A | NOT A |
|---|---|
| 0 | 1 |
| 1 | 0 |

# Examples of Logical Operations

## AND

- **useful for clearing bits**
  - —AND with zero = 0
  - —AND with one = no change

```
     11000101
AND  00001111
     --------
     00000101
```

## OR

- **useful for setting bits**
  - —OR with zero = no change
  - —OR with one = 1

```
     11000101
OR   00001111
     --------
     11001111
```

## NOT

- **unary operation -- one argument**
- **flips every bit**

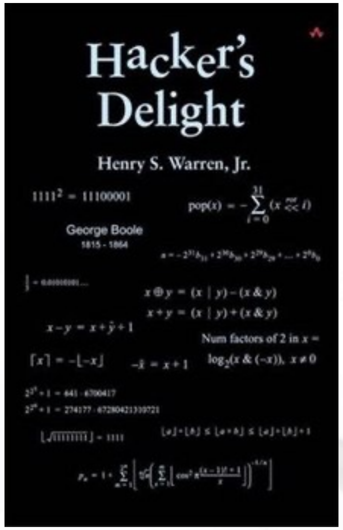
```
NOT  11000101
     --------
     00111010
```

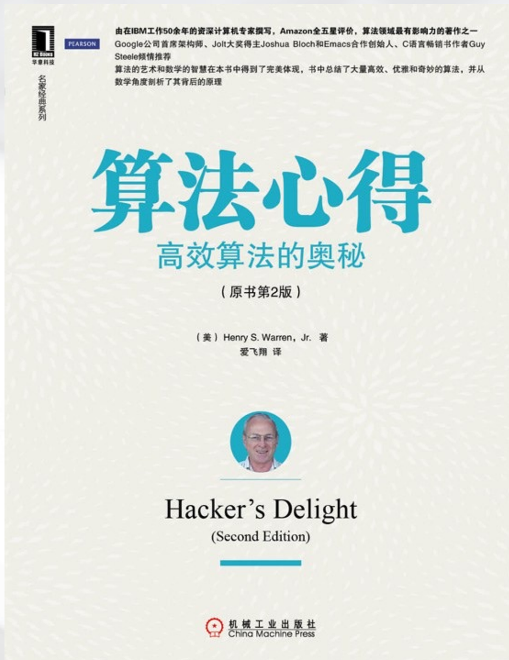
## ■ *Hacker's Delight*

- by Henry S. Warren, Jr.
- first published in 2002
- fast bit-level and low-level arithmetic algorithms

## ■ Bit Twiddling Hacks

- By Sean Eron Anderson
- https://graphics.stanford.edu/~seander/bithacks.html



# Outline

# Fractions: Fixed-Point

■ **How can we represent fractions?**

- Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."
- 2's comp addition and subtraction still work.
  - —if binary points are aligned

$2^{-1} = 0.5$

$2^{-2} = 0.25$

$2^{-3} = 0.125$

```
  00101000.101 (40.625)
+ 11111110.110 (-1.25)
  00100111.011 (39.375)
```

*No new operations -- same as integer arithmetic.*

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

# Fractions: Fixed-Point

- **How can we represent fractions?**
  - **Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."**
  - **2's comp addition and subtraction still work.**
    - **—if binary points are aligned**
- **Example    5-bit fraction**

| fraction | | | Integer | |
|---|---|---|---|---|
| 010.10 | 2.5 | $(10/2^2)$ | 01010 | 10 |
| +101.11 | -2.25 | $(-9/2^2)$ | + 10111 | -9 |
| 000.01 | 0.25 | (1/4) | 00001 | 1 |

- **A *n*-bit binary fraction with *k* fraction bits is equivalent to the n-bit binary integer divided by $2^k$**

# Very Large and Very Small Data

- The LC-3 use the 16bit 2's complement data type,
- One bit to identify positive or negative, 15bits to represent the magnitude of the value. We can express values:

$$-2^{15} \text{ through } 2^{15}-1$$

(– 32768 through 32767)

**How can we represent very large and very small data?**

# Very Large and Very Small Data

**Large values: 6.023 x $10^{23}$ — requires 79 bits**

**Small values: 6.626 x $10^{-34}$ — requires >110 bits**

**How can we represent very large and very small data?**

# Very Large and Very Small: Floating-Point

**Large values: $6.023 \times 10^{23}$ — requires 79 bits**

**Small values: $6.626 \times 10^{-34}$ — requires >110 bits**

**Use equivalent of "scientific notation" : $F \times 2^E$**

**Need to represent F (*fraction*), E (*exponent*), and sign.**

**IEEE 754 Floating-Point Standard (32-bits):**

| *1b* | *8b* | *23b* |
|---|---|---|
| S | Exponent | Fraction |

$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \le \text{exponent} \le 254$$

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

# Floating Point Example

## Single-precision IEEE floating point number:

$$\underbrace{1}_{\text{sign}}\ \underbrace{01111110}_{\text{exponent}}\ \underbrace{10000000000000000000000}_{\text{fraction}}$$

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.100000000000… = 0.5 (decimal).

Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.

# Floating Point Example

- **Example 2.12**

0 01111011 000 0000 0000 0000 0000 0000

\+ 123 0

$$1.0\times 2^{123-127} = 2^{-4} = \frac{1}{16}$$

- **Example 2.13**

$-\ 6\frac{5}{8}$

$-\ (6+\frac{4}{8}+\frac{1}{8}) = -110.101$

$-\ 1.10101\times 2^{2}$

$-\ 1.10101\times 2^{129-127}$

1 10000001 101 0100 0000 0000 0000 0000

# Floating Point Example

- **Example 2.14**

0 10000011 001 0100 0000 0000 0000 0000

$+\ 1.00101 \times 2^{131-127} = 10010.1 = 18.5$

1 10000010 001 0100 0000 0000 0000 0000

$-\ 1.00101 \times 2^{130-127} = -1001.01 = -9.25$

0 11111110 111 1111 1111 1111 1111 1111

$+\ 1.1111\ldots \times 2^{254-127} = 1.1111\ldots \times 2^{127} \approx 2^{128}$

## Floating-Point Operations

- **Will regular 2's complement arithmetic work for Floating Point numbers?**

**(*Hint*: In decimal, how do we compute $3.07 \times 10^{12} + 9.11 \times 10^{8}$?)**

# Other Data Types

- **Text strings**
  - sequence of characters, terminated with NULL (0)
  - typically, no hardware support
- **Image**
  - array of pixels
    - — monochrome: one bit (1/0 = black/white)
    - — color: red, green, blue (RGB) components (e.g., 8 bits each)
    - — other properties: transparency
  - hardware support:
    - — typically none, in general-purpose processors
    - — MMX -- multiple 8-bit operations on 32-bit word
- **Sound**
  - sequence of fixed-point numbers

**Within the Computer: Everything is a Number.**

# How do computers represent text ? -- ASCII Characters

## ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, …) characters

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | nul | 10 | dle | 20 | sp | 30 | 0 | 40 | @ | 50 | P | 60 | ` | 70 | p |
| 01 | soh | 11 | dc1 | 21 | ! | 31 | 1 | 41 | A | 51 | Q | 61 | a | 71 | q |
| 02 | stx | 12 | dc2 | 22 | " | 32 | 2 | 42 | B | 52 | R | 62 | b | 72 | r |
| 03 | etx | 13 | dc3 | 23 | # | 33 | 3 | 43 | C | 53 | S | 63 | c | 73 | s |
| 04 | eot | 14 | dc4 | 24 | $ | 34 | 4 | 44 | D | 54 | T | 64 | d | 74 | t |
| 05 | enq | 15 | nak | 25 | % | 35 | 5 | 45 | E | 55 | U | 65 | e | 75 | u |
| 06 | ack | 16 | syn | 26 | & | 36 | 6 | 46 | F | 56 | V | 66 | f | 76 | v |
| 07 | bel | 17 | etb | 27 | ' | 37 | 7 | 47 | G | 57 | W | 67 | g | 77 | w |
| 08 | bs | 18 | can | 28 | ( | 38 | 8 | 48 | H | 58 | X | 68 | h | 78 | x |
| 09 | ht | 19 | em | 29 | ) | 39 | 9 | 49 | I | 59 | Y | 69 | i | 79 | y |
| 0a | nl | 1a | sub | 2a | * | 3a | : | 4a | J | 5a | Z | 6a | j | 7a | z |
| 0b | vt | 1b | esc | 2b | + | 3b | ; | 4b | K | 5b | [ | 6b | k | 7b | { |
| 0c | np | 1c | fs | 2c | , | 3c | < | 4c | L | 5c | \ | 6c | l | 7c | \| |
| 0d | cr | 1d | gs | 2d | - | 3d | = | 4d | M | 5d | ] | 6d | m | 7d | } |
| 0e | so | 1e | rs | 2e | . | 3e | > | 4e | N | 5e | ^ | 6e | n | 7e | ~ |
| 0f | si | 1f | us | 2f | / | 3f | ? | 4f | O | 5f | _ | 6f | o | 7f | del |

# ASCII ( American Standard Code for Information Interchange )

# ASCII表

( American Standard Code for Information Interchange 美国标准信息交换代码 )

| 高四位 | | ASCII控制字符 | | | | | | | | | | | | ASCII打印字符 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0000 | | | | | | 0001 | | | | | | 0010 | | 0011 | | 0100 | | 0101 | | 0110 | | 0111 | | |
| | | 0 | | | | | | 1 | | | | | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | |
| 低四位 | | 十进制 | 字符 | Ctrl | 代码 | 转义字符 | 字符解释 | 十进制 | 字符 | Ctrl | 代码 | 转义字符 | 字符解释 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | Ctrl |
| 0000 | 0 | 0 | | ^@ | NUL | \0 | 空字符 | 16 | ► | ^P | DLE | | 数据链路转义 | 32 | | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p | |
| 0001 | 1 | 1 | ☺ | ^A | SOH | | 标题开始 | 17 | ◄ | ^Q | DC1 | | 设备控制 1 | 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q | |
| 0010 | 2 | 2 | ☻ | ^B | STX | | 正文开始 | 18 | ↕ | ^R | DC2 | | 设备控制 2 | 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r | |
| 0011 | 3 | 3 | ♥ | ^C | ETX | | 正文结束 | 19 | ‼ | ^S | DC3 | | 设备控制 3 | 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s | |
| 0100 | 4 | 4 | ♦ | ^D | EOT | | 传输结束 | 20 | ¶ | ^T | DC4 | | 设备控制 4 | 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t | |
| 0101 | 5 | 5 | ♣ | ^E | ENQ | | 查询 | 21 | § | ^U | NAK | | 否定应答 | 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u | |
| 0110 | 6 | 6 | ♠ | ^F | ACK | | 肯定应答 | 22 | ▬ | ^V | SYN | | 同步空闲 | 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v | |
| 0111 | 7 | 7 | • | ^G | BEL | \a | 响铃 | 23 | ↨ | ^W | ETB | | 传输块结束 | 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w | |
| 1000 | 8 | 8 | ◘ | ^H | BS | \b | 退格 | 24 | ↑ | ^X | CAN | | 取消 | 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x | |
| 1001 | 9 | 9 | ○ | ^I | HT | \t | 横向制表 | 25 | ↓ | ^Y | EM | | 介质结束 | 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y | |
| 1010 | A | 10 | ◙ | ^J | LF | \n | 换行 | 26 | → | ^Z | SUB | | 替代 | 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z | |
| 1011 | B | 11 | ♂ | ^K | VT | \v | 纵向制表 | 27 | ← | ^[ | ESC | \e | 溢出 | 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { | |
| 1100 | C | 12 | ♀ | ^L | FF | \f | 换页 | 28 | ∟ | ^\ | FS | | 文件分隔符 | 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| | |
| 1101 | D | 13 | ♪ | ^M | CR | \r | 回车 | 29 | ↔ | ^] | GS | | 组分隔符 | 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } | |
| 1110 | E | 14 | ♫ | ^N | SO | | 移出 | 30 | ▲ | ^^ | RS | | 记录分隔符 | 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ | |
| 1111 | F | 15 | ☼ | ^O | SI | | 移入 | 31 | ▼ | ^- | US | | 单元分隔符 | 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | ⌂ | ^Backspace 代码：DEL |

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键 ”方法输入。

# Example：how do computers represent text ?

- Using the ASCII table, try to decode this message:

010101000110100001100101001000000110110101101111011101000110100001100101

01110010011011000110000101101110011001000000101100010010010010000001100011

0110111101101101011001010010000000110001001100001011000110110110100100001

# Example：how do computers represent text ?

- Using the ASCII table, try to decode this message:

01010100 01101000 01100101 00100000 01101101 01101111 01110100 01101000 01100101

01110010 01101100 01100001 01101110 01100100 00101100 01001001 00100000 01100011

01101111 01101101 01100101 001000000 01100010 01100001 01100011 01101101 00100001

## Converting binary to text

- Using the ASCII table, try to decode this message:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 01010100 | 01101000 | 01100101 | 00100000 | 01101101 | 01101111 | 01110100 | 01101000 | 01100101 |
| 54 | 68 | 65 | 20 | 6d | 6f | 74 | 68 | 65 |
| 01110010 | 01101100 | 01100001 | 01101110 | 01100100 | 00101100 | 01001001 | 00100000 | 01100011 |
| 72 | 6c | 61 | 6e | 64 | 2c | 49 | 20 | 63 |
| 01101111 | 01101101 | 01100101 | 001000000 | 01100010 | 01100001 | 01100011 | 01101101 | 00100001 |
| 6f | 6d | 65 | 20 | 62 | 61 | 63 | 6d | 21 |

# Converting text to binary

- Using the ASCII table, try to decode this message:

**The motherland, I come back!**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| T | h | e | (sp) | m | o | t | h | e |
| 54 | 68 | 65 | 20 | 6d | 6f | 74 | 68 | 65 |
| 01010100 | 01101000 | 01100101 | 00100000 | 01101101 | 01101111 | 01110100 | 01101000 | 01100101 |
| r | l | a | n | d | , | I | (sp) | c |
| 72 | 6c | 61 | 6e | 64 | 2c | 49 | 20 | 63 |
| 01110010 | 01101100 | 01100001 | 01101110 | 01100100 | 00101100 | 01001001 | 00100000 | 01100011 |
| o | m | e | (sp) | b | a | c | k | ! |
| 6f | 6d | 65 | 20 | 62 | 61 | 63 | 6d | 21 |
| 01101111 | 01101101 | 01100101 | 001000000 | 01100010 | 01100001 | 01100011 | 01101101 | 00100001 |

# Converting binary to text

**The motherland, I come back!**

# Interesting Properties of ASCII Code

- What is relationship between a decimal digit ('0', '1', ...)and its ASCII code?
- What is the difference between an upper-case letter ('A', 'B', ...) and its lower-case equivalent ('a', 'b', ...)?
- Given two ASCII characters, how do we tell which comes first in alphabetical order?
- Are 128 characters enough? (http://www.unicode.org/)

*No new operations – integer arithmetic and logic.*

# How do computers represent image ?

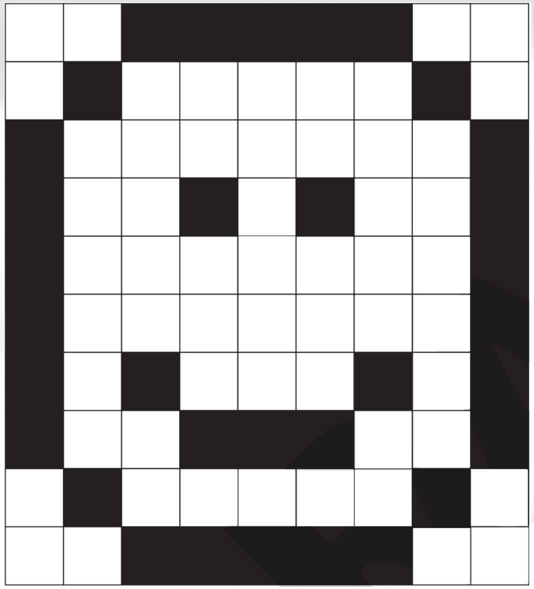
- Each image has a **resolution** and a **color depth**.
  - The **resolution** is the number of pixels wide and the number of pixels high that are used to create the image.
  - The **color depth** is the number of bits that are used to represent each color.

**Figure 1.3:** 8-bit colour 16-bit colour 32-bit colour

- For example, each color could be represented using 8-bit, 16-bit or 32-bit binary numbers.
- The greater the number of bits, the greater the range of colors that can be represented.

# Converting images to binary

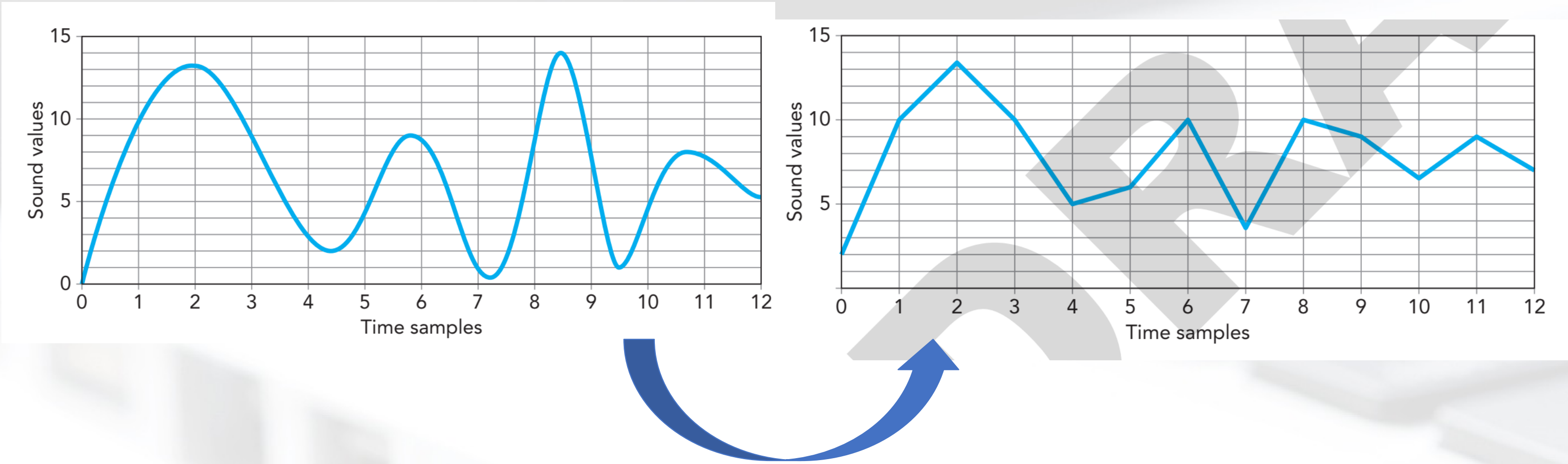
- If each pixel is converted to its binary value, a data set such as the following could be created:



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

# How do computers represent sound ?

- Sound is made up of sound waves. When sound is recorded, this is done at set time intervals. This process is known as **sound sampling** :




| Time sample | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sound value | 9 | 13 | 9 | 3.5 | 4 | 9 | 1.5 | 9 | 8 | 5 | 8 | 5.5 |

# Hexadecimal Notation

011101010001111010011010111

## Hexadecimal Notation

- It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.
  - fewer digits -- four bits per hex digit
  - less error prone -- easy to corrupt long string of 1's and 0's

| Binary | Hex | Decimal |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |

| Binary | Hex | Decimal |
|---|---|---|
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | A | 10 |
| 1011 | B | 11 |
| 1100 | C | 12 |
| 1101 | D | 13 |
| 1110 | E | 14 |
| 1111 | F | 15 |

# Converting from Binary to Hexadecimal

- **Every four bits is a hex digit.**
  - `start grouping from right-hand side`

**011 1010 1000 1111 0100 1101 0111**

| 011 | 1010 | 1000 | 1111 | 0100 | 1101 | 0111 |
|---|---|---|---|---|---|---|
| ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| **3** | **A** | **8** | **F** | **4** | **D** | **7** |

***This is not a new machine representation, just a convenient way to write the number.***

# LC-3 Data Types

- Some data types are supported directly by the instruction set architecture.
- For LC-3, there is only one supported data type:
  - `16-bit 2's complement signed integer`
  - `Operations: ADD, AND, NOT`
- Other data types are supported by <u>interpreting</u> 16-bit values as logical, text, fixed-point, etc., in the software that we write.