CS107, Lecture 2

Unix, C, Bits and Bytes, Integer Representations

Reading: Bryant & O'Hallaron, Ch. 2.2-23 (skim)

Ed Discussion: https://edstem.org/us/courses/28214/discussion/1842418

The C Language

C was created around 1970 to make writing Unix and Unix tools easier.

- Part of the C/C++/Java family of languages (C is by far the oldest of the three)
- Design principles:
  - Small, simple abstractions layered over hardware
  - Minimalist, WYSIWYG
  - Prioritizes efficiency and simplicity over safety, high-level abstractions

C vs. C++ and Java

They all share:

- Syntax
- Basic data types
- Arithmetic, relational, and logical operators

C limitations:

- No advanced features like operator overloading, default arguments, pass by reference, classes, etc.
- No elaborate libraries (graphics, networking, etc.) – small language means less to learn ☺
- Forgiving compiler and virtually no runtime checks — lack of runtime support means carelessly written code can be easily exploited

Programming Language Philosophies

C is procedural and imperative: you implement functions, rather than define classes and invoke methods on objects. **C is small, fast and efficient.**

C++ is procedural, with objects: you write functions, define new variable types as classes, and invoke methods on objects.

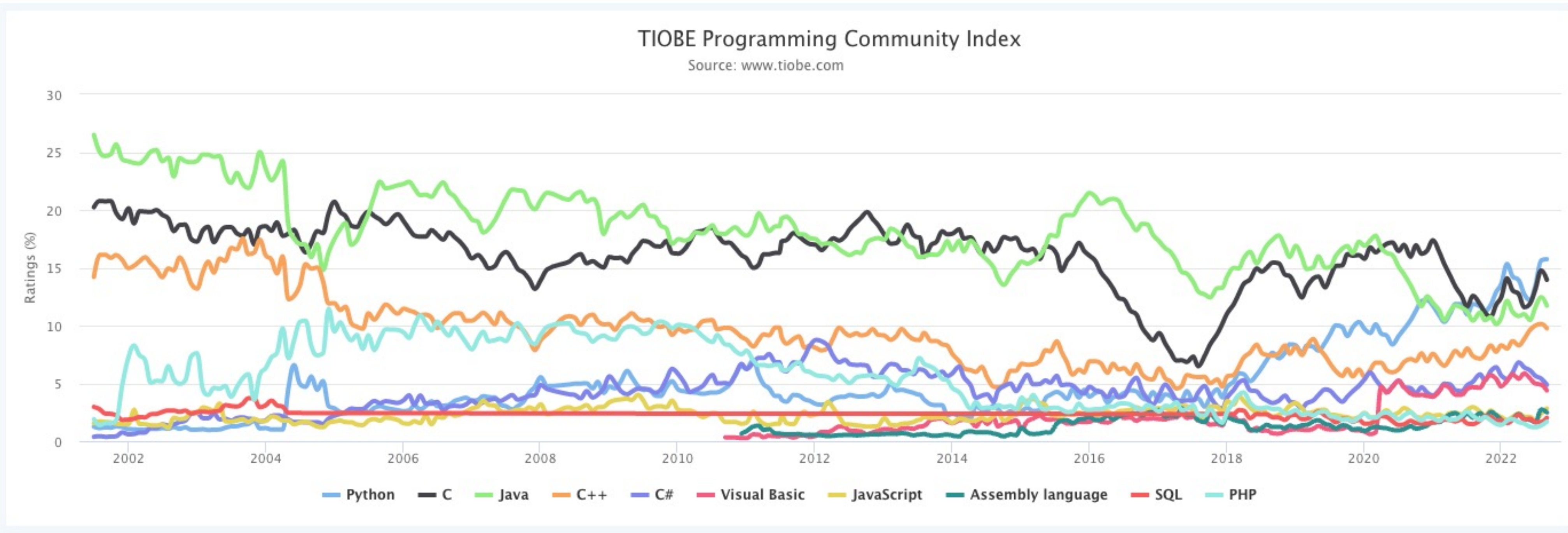
Python is procedural, but dynamically typed: you still write functions and invoke methods on objects, but type checking occurs during runtime.

Java is truly object-oriented: nearly everything is an object, and everything you write conforms to the object-oriented paradigm.

Why C?

- Many tools (and even other languages, e.g., Python) are implemented using C.
- C is the language of choice for fast, highly efficient programs.
- C is popular for systems programming (operating systems, networking, etc.).
- C lets you examine and manipulate the underlying system.
- Modern alternatives to C as a systems programming language are emerging, but they’re more complicated.

Programming Language Popularity



https://www.tiobe.com/tiobe-index/

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Our First C Program

```
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>  // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Program comments
You can write block or inline comments.

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Import statements
C libraries are written with angle brackets.
Local libraries have quotes:
`#include "wordle-utils.h"`

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Main function – entry point for the program
Should always return an integer (0 = success)

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

Main parameters – **main** takes two parameters, both relating to the *command line arguments* used to execute the program.

argc is the *number* of arguments in **argv**
argv is an *array of arguments* ***(char * is C string)***

Our First C Program

```c
/*
 * hello.c
 * This program prints a welcome message
 * to the user.
 */
#include <stdio.h>   // for printf

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

printf – prints text to the screen

Console Output: printf

`printf(text, arg1, arg2, arg3,...);`

`printf` makes it easy to print out the values of variables or expressions.

If you include *placeholders* in your printed text, `printf` will replace each placeholder *in order* with the values of the parameters passed after the text.

`%s (string)`   `%d (integer)`   `%f (double)`

```
// Example
char *prefix = "CS";
int number = 107;
printf("You are in %s%d", prefix, number);      // You are in CS107
```

Familiar Syntax

```
int x = 42 + 7 * -5;                  // variables, types
double pi = 3.14159;
char c = 'Q';                         /* two comment styles */

for (int i = 0; i < 10; i++) {        // for loops
    if (i % 2 == 0) {                 // if statements
        x += i;
    }
}

while (x > 0 && c == 'Q' || DEBUG) {  // while loops, logic, DEBUG global bool
    x = x / 2;
    if (x == 42) {
        return 0;
    }
}

binky(x, 107, c);                     // function call
```

Boolean Variables

To declare Booleans, (e.g. `bool b = ____`), you must include `stdbool.h`:

```
#include <stdio.h>      // for printf
#include <stdbool.h>    // for bool

int main(int argc, char *argv[]) {
    bool test = argc > 2 && binky(argc) > 0;
    if (test) {
        printf("Hello, world!\n");
    } else {
        printf("Howdy, world!\n");
    }
    return 0;
}
```

Question Break

Writing, Debugging and Compiling

We will use:

- the **emacs** text editor to write our C programs
- the **make** tool to compile our C programs
- the **gdb** debugger to debug our programs
- the **valgrind** tools to debug memory errors and measure program efficiency

Now

Next week

Working On C Programs

- **ssh** – remotely log in to Myth computers
- **Emacs** – text editor to write and edit C programs
  - Use the mouse to position cursor, scroll, and highlight text
  - CTRL-x CTRL-s to save, CTRL-x CTRL-c to quit
- **make** – compile program using provided Makefile
- **./myprogram** – run executable program (optionally with arguments)
- **make clean** – remove executables and other compiler files
- Lecture code is accessible at **/afs/ir/class/cs107/lecture-code/lect[N]**
  - Make your own copy: **cp -r /afs/ir/class/cs107/lecture-code/lect[N] lect[N]**
  - See the website for even more commands, and a complete reference.

Demo: Compiling And Running A C Program

Get up and running with our guide:

http://cs107.stanford.edu/resources/getting-started.html

assign0

Assignment 0 (Intro to Unix and C) is due in a week from today on **10/5 at 11:59PM PDT**.

There are **5** parts to the assignment, which is meant to get you comfortable using the command line, and editing/compiling/running C programs:

- Navigate website to become familiar with common Unix commands
- **Clone** the assign0 starter project
- **Answer** several questions in `readme.txt`
- **Compile** a provided C program and **modify** it
- **Submit** the assignment

Question Break

CS107 Topic 1

How can a computer represent integer numbers?

Why is answering this question important?

- Helps us understand the limitations of computer arithmetic (today and Friday)
- Shows us how to more efficiently perform arithmetic (Friday and Monday)
- Shows us how we can encode data more compactly and efficiently (Monday)

assign1: implement 3 programs that manipulate binary representations to (1) work around the limitations of arithmetic with addition, (2) simulate an evolving colony of cells, and (3) print Unicode text to the terminal.

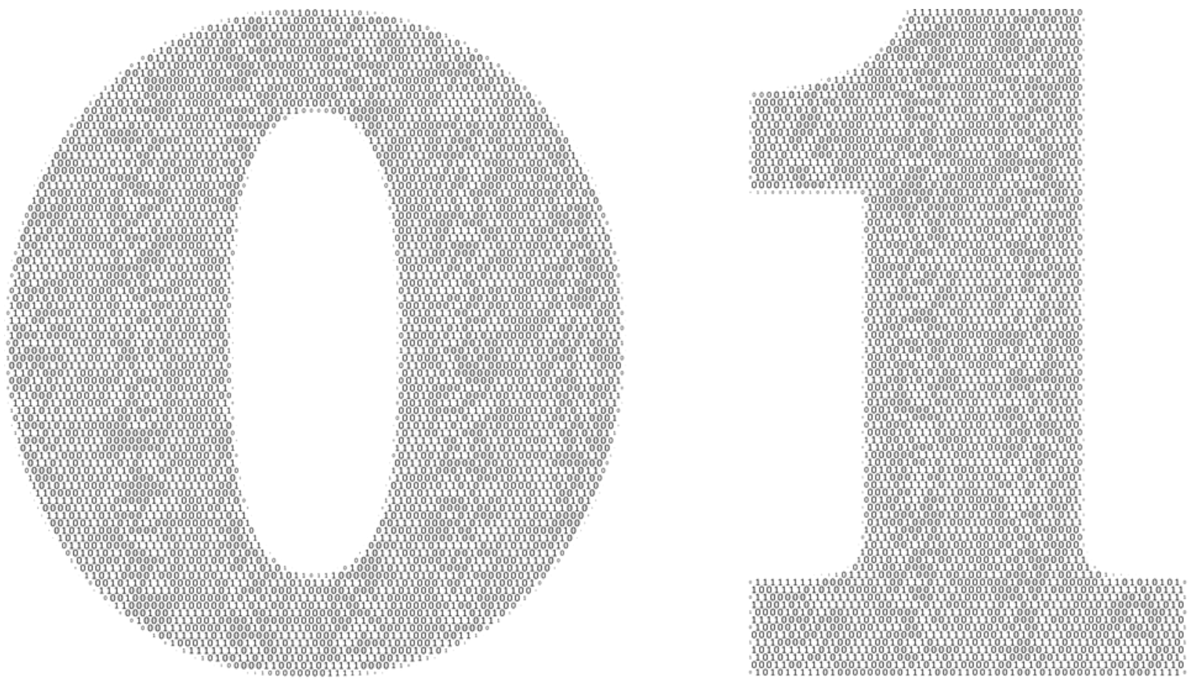
Learning Goals

- Learn about the binary and hexadecimal number systems and how to convert between number systems
- Understand how positive and negative numbers are represented in binary
- Learn about overflow, why it occurs, and its impacts

Demo: Unexpected Behavior

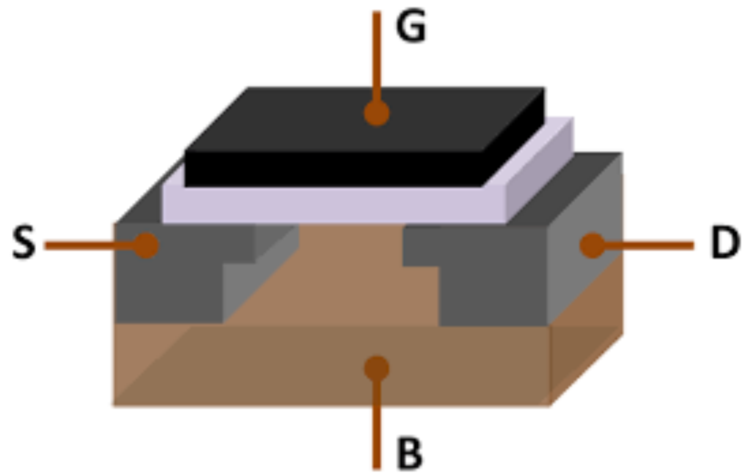
```
cp -r /afs/ir/class/cs107/lecture-code/lect2 .
```

Bits



Bits

Computers are built around the idea of two states: "on" and "off". Transistors represent this in hardware, and bits represent this in software!



One Bit At A Time

- We can combine bits, as with base-10 numbers, to represent more data.
  8 bits = 1 byte.
- Computer memory is just a large array of bytes. It is **byte-addressable**; you can't address a bit in isolation, only a full byte.
- Computers still fundamentally operate with bits; we have just gotten more creative about how to represent data using bits!
  - Images
  - Audio
  - Video
  - Text
  - And more…

Base 10

5 9 3 4

digits 0 – 9

(or rather, 0 through base – 1)

Base 10

5 9 3 4

thousands hundreds tens ones

$= \mathbf{5} * 1000 + \mathbf{9} * 100 + \mathbf{3} * 10 + \mathbf{4} * 1$

Base 10

$$5\quad 9\quad 3\quad 4$$

$$10^3\quad 10^2\quad 10^1\quad 10^0$$

Base 10

5 9 3 4

10^X: 3 2 1 0

Base 2

1 0 1 1

2^X: 3   2   1   0

digits 0 – 1
(or rather, 0 through base – 1)

Base 2

1 0 1 1

2^3 2^2 2^1 2^0

Base 2

Most significant bit (MSB)

Least significant bit (LSB)

1 0 1 1

eights fours twos ones

$$= \mathbf{1} * 8 + \mathbf{0} * 4 + \mathbf{1} * 2 + \mathbf{1} * 1 = 11_{10}$$

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **2^2=4**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **2^1=2**
  - 6 – 2^2 – 2^1 = 0!

$$\underset{2^3}{\underline{0}}\quad \underset{2^2}{\underline{1}}\quad \underset{2^1}{\underline{1}}\quad \underset{2^0}{\underline{0}}$$

= **0***8 + **1***4 + **1***2 + **0***1 = 6

Practice: Base 2 to Base 10

What is the base-2 value 1010 in base-10?

a) **20**
b) **101**
c) **10**
d) **5**
e) **Other**

Practice: Base 10 to Base 2

What is the base-10 value 14 in base 2?

a) **1111**

b) **1110**

c) **1010**

d) **Other**

Byte Values

What are the minimum and maximum base-10 values a single byte (8 bits) can represent?

minimum = 0 **maximum = 255**

11111111

2^x: 7 6 5 4 3 2 1 0

- **Strategy 1:** $1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 255$
- **Strategy 2:** $2^8 - 1 = 255$

Multiplying by Base

$$1450 \times 10 = 1450\underline{\mathbf{0}}$$

$$1100_2 \times 2 = 1100\underline{\mathbf{0}}$$

Key Idea: appending 0 to the end effectively multiplies by the base!

Dividing by Base

$$1450 / 10 = 145$$

$$1100_2 / 2 = 110$$

Key Idea: chomping off 0 from the end divides by the base!

Question Break

Hexadecimal

When working with bits, oftentimes we have large numbers with 32 or 64 bits.

- Instead, we'll generally encode numbers in **base-16**, or **hexadecimal**, instead.

$$\underbrace{0110}_{0\text{-}15}\ \underbrace{1010}_{0\text{-}15}\ \underbrace{0011}_{0\text{-}15}$$

Hexadecimal

When working with bits, oftentimes we have large numbers with 32 or 64 bits.

- Instead, we'll generally encode numbers in **base-16**, or **hexadecimal**, instead.

0-15 0-15 0-15

Each quartet of bits can be rewritten as a single digit, in **base-16**!

Hexadecimal

Hexadecimal is **base-16**, so we need digits for 1-15. How?

0 1 2 3 4 5 6 7 8 9

Hexadecimal

- If it's not clear from context, we can explicitly identify numbers as hexadecimal by prefixing them with **`0x`** and identify numbers as binary by prefixing with **`0b`**.
- e.g., **`0xf5`** is **`0b11110101`**

$$0x\,\underbrace{f}_{1111}\;\underbrace{5}_{0101}$$

Hexadecimal

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

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

Practice: Hexadecimal to Binary

What is **`0x173A`** in binary?

Hexadecimal	**1**	**7**	**3**	**A**
Binary	**0001**	**0111**	**0011**	**1010**

Practice: Hexadecimal to Binary

What is **`0b1111001010`** in hexadecimal? (*Hint: start from the right)*

Binary	**11**	**1100**	**1010**
Hexadecimal	**3**	**C**	**A**

Hexadecimal: Quirky but concise

- Let's take a single byte (8 bits):

`165`	base-10: Human-readable, but cannot easily interpret on/off bits
`0b10100101`	base-2: Yes, computers love this, but most humans do not.
`0xa5`	base-16: Easy to convert to base-2, More easily digested format (fun fact: a half-byte is called a nibble)

Number Representations

- **Unsigned Integers**: positive integers and 0. (e.g. 0, 1, 2, … 99999…)
- **Signed Integers:** negative, positive integers and 0. (e.g. …-2, -1, 0, 1,… 9999…)

- **Floating Point Numbers:** real numbers. (e,g. 0.1, -12.2, 1.5×10^{12})

Look up IEEE floating point if you're interested!

Number Representations

C Declaration	Size (Bytes)
`int`	4
`double`	8
`float`	4
`char`	1
`char *`	8
`short`	2
`long`	8

Back When Jerry Learned C

C Declaration	Size (Bytes)
`int`	4
`double`	8
`float`	4
`char`	1
`char *`	4
`short`	2
`long`	4

Transitioning To Larger Data Types

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32} - 1, equaling **2^{32} bytes of addressable memory**. This equals **4 gigabytes**, meaning that 32-bit computers could address *at most* **4GB** of memory!
- Because of this, most computers now are to **64-bit.** This means that data types were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers can distinguish between addresses 0 to 2^{64} - 1, equaling **2^{64} bytes of addressable memory.** This equals **16 exabytes**, meaning that 64-bit computers could address up to **16 * 1024 * 1024 * 1024 GB** of memory!

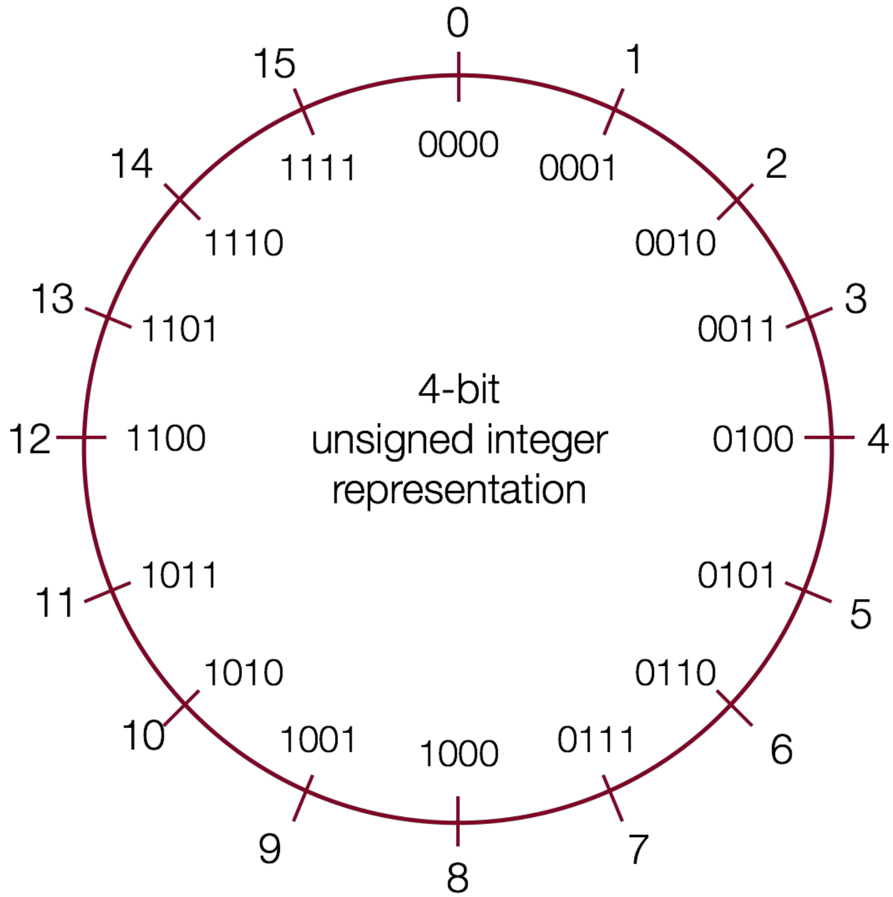
Unsigned Integers

- An **unsigned** integer is either 0 or some positive integer (no negatives).
- We have already discussed the conversion between decimal and binary. Examples:

```
0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15
```

- The range of an unsigned number is $0 \rightarrow 2^{w} - 1$, where w is the number of bits. e.g., a 32-bit integer can represent 0 to $2^{32} - 1$ (4,294,967,295).

Unsigned Integers



Question Break

Signed Integers

A **signed** integer is a negative integer, 0, or a positive integer.

- *Problem:* How can we represent negative *and* positive numbers in binary?

Idea: let the **most** significant bit represent sign and let the others represent magnitude.

Sign Magnitude Representation: 4-bit

$$\underbrace{0}_{\text{positive}}\underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\underbrace{011}_{3}$$

Sign Magnitude Representation: 4-bit

$$\underbrace{0}_{\text{positive}}\underbrace{000}_{0}$$

$$\underbrace{1}_{\text{negative}}\underbrace{000}_{0}$$

Sign Magnitude Representation: 4-bit

1 000 = -0	0 000 = 0
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

We're only representing 15 different values via 16 different patterns.
#sadness

Sign Magnitude Representation

- **Pro:** easy to represent, and easy to convert to and from decimal.
- **Con:** +/-0 is 🤯
- **Con:** we lose a bit that could be used to represent more numbers
- **Con:** arithmetic is tricky: we need to find the sign, perhaps subtract (borrow and carry, etc.), maybe change the sign, maybe not. This complicates the hardware support for something as fundamental as addition.

Can we do better?

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$
\begin{array}{r}
0101 \\
+\;{\color{red}????} \\
\hline
0000
\end{array}
$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$\begin{array}{r} 0101 \\ +\ 1011 \\ \hline 0000 \end{array}$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$
\begin{array}{r}
0011 \\
+\ ???? \\
\hline
0000
\end{array}
$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$\begin{array}{r} 0011 \\ +\,1101 \\ \hline 0000 \end{array}$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$
\begin{array}{r}
0000 \\
+\ ???? \\
\hline
0000
\end{array}
$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$\begin{array}{r} 0000 \\ +\ {\color{red}0000} \\ \hline 0000 \end{array}$$

There Seems To Be A Pattern

0101	0011	0000
+ 1011	+ 1101	+ 0000
0000	0000	0000

The negative number is the positive number **inverted, plus one!**

There Seems To Be A Pattern

A binary number plus its inverse is all 1s.

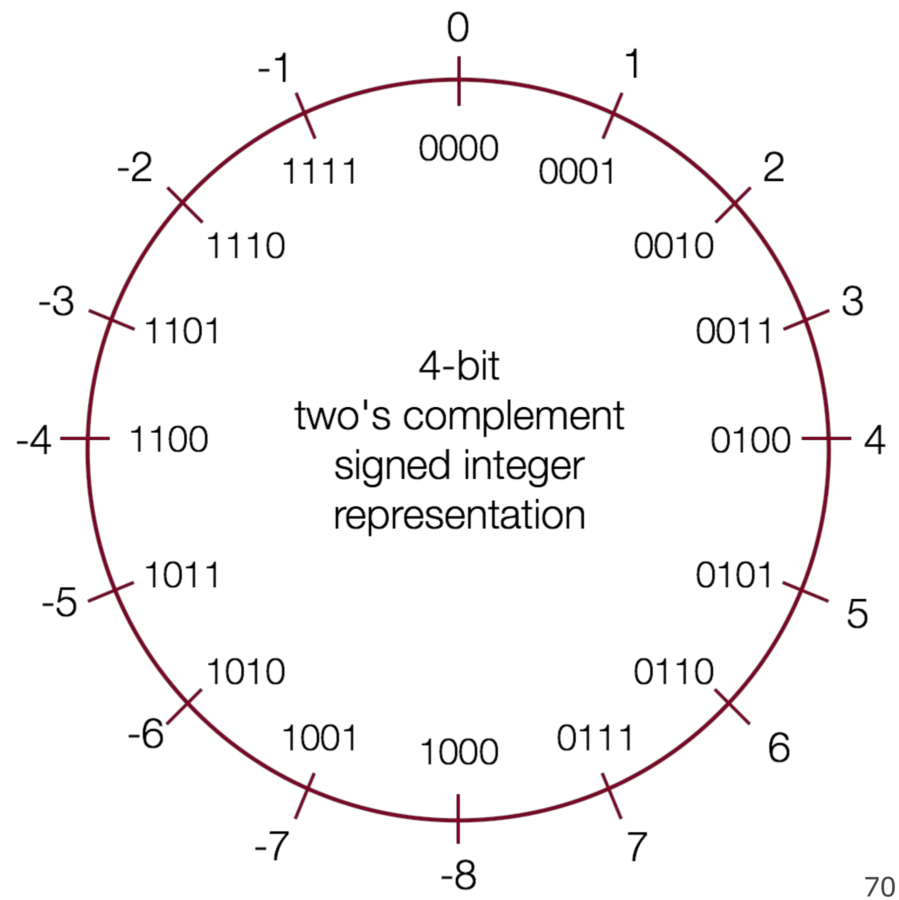
$$\begin{array}{r} 0101 \\ +\,1010 \\ \hline 1111 \end{array}$$

Add 1 to this to carry over all 1s and get 0!

$$\begin{array}{r} 1111 \\ +\,0001 \\ \hline 0000 \end{array}$$

Two's Complement

- With **two's complement**, we represent a positive number as **itself**, and its negative equivalent as the **two's complement of itself.**
- The **two's complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, **and** back from negative to positive!



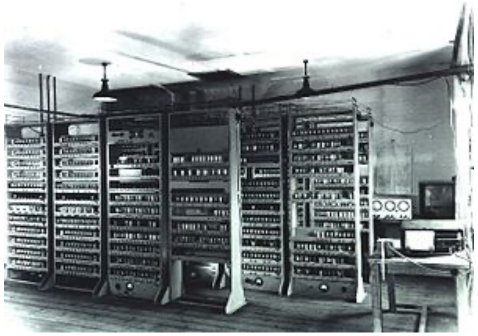
History: Two's complement

- Binary representation was first proposed by John von Neumann in *First Draft of a Report on the EDVAC* (1945).
  - That same year, he also invented the merge sort algorithm
- Many early computers used either sign-magnitude or one's complement.

```
+7    0b0000 0111
-7    0b1111 1000
```

8-bit one's complement

- The System/360, developed by IBM in 1964, was widely popular (it had 1024KB memory!) and established two's complement as the dominant binary representation of integers.

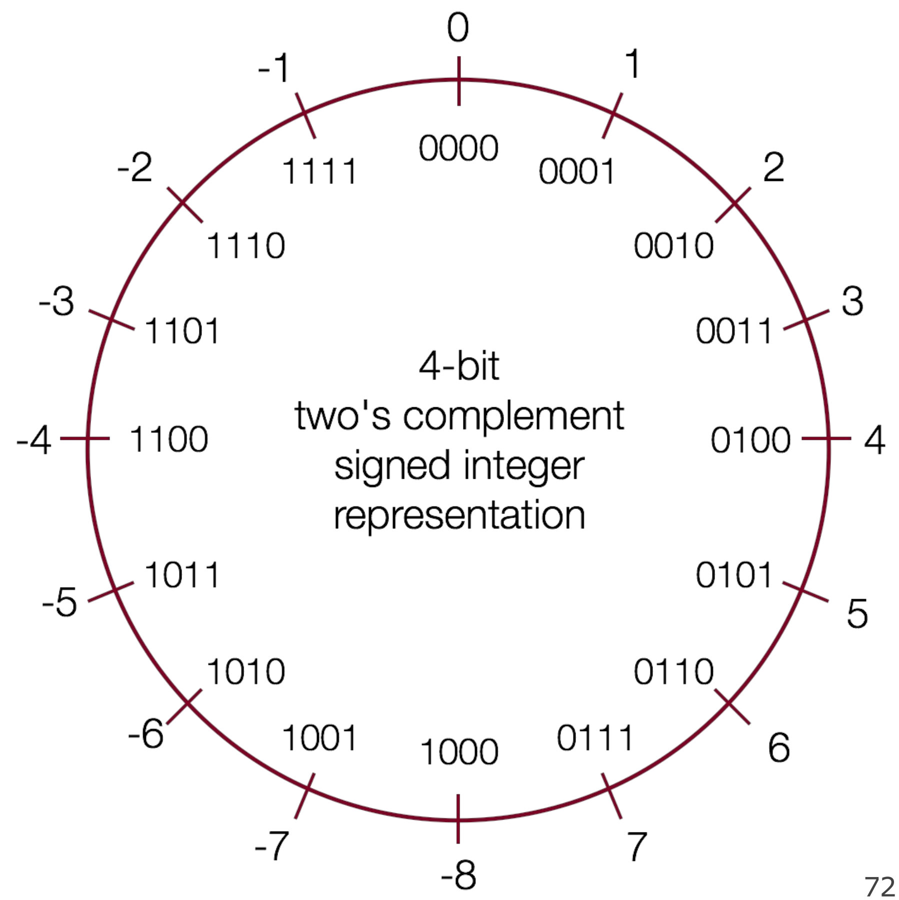


EDSAC (1949)

System/360 (1964)

Two’s Complement

- **Con:** more difficult to represent, and difficult to convert to and from decimal, between positive and negative.
- **Pro:** only 1 representation for 0! 😍
- **Pro:** the most significant bit still indicates the sign of a number.
- **Pro:** addition works for any combination of positive and negative!



Two's Complement

Adding two numbers is just that: adding! There is no special case needed for negative numbers. e.g., what is 2 + -5?

0010	2
+ 1011	-5
1101	-3

Two’s Complement

Subtracting two numbers is just performing the two’s complement on the second of them and then adding instead of subtracting, e.g., 4 – 5 = -1.

$$
\begin{array}{rr}
 & 0100 \\
- & 0101 \\
\hline
\end{array}
\qquad
\begin{array}{r}
4 \\
5
\end{array}
\qquad \longrightarrow \qquad
\begin{array}{rr}
 & 0100 \\
+ & {\color{red}1011} \\
\hline
 & 1111
\end{array}
\qquad
\begin{array}{r}
4 \\
-5 \\
-1
\end{array}
$$

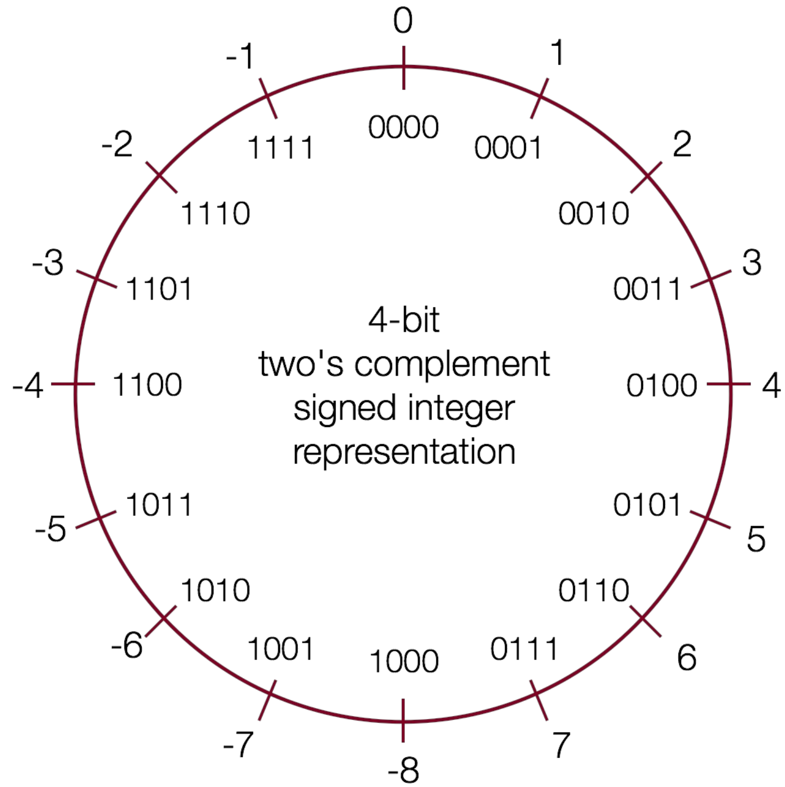
Practice: Two's Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)



Question Break