Data Representation

CISC 3310 — CUNY Brooklyn College
Lecture Notes

Data Representation

In this chapter, we will:

1. Learn how humans count vs. how computers count, and why it matters to our course.
2. Find out how we represent integers, floating-point numbers, and characters in a computer.
3. Practice with computer addition and subtraction of numbers.
4. Explain why the precision of representing floating-point numbers in a computer is limited.

How Does a Computer Count?

Before we dive deeper into how computer circuits and, in general, how hardware components are built up and work, we first need to grasp the thinking process of a computer. That is, how a computer could use the given hardware to calculate things, make decisions, and store data.

Hardware components are controlled by electric currents, which the computer will translate into numbers, so our first step will be to learn how a computer counts and works with numbers. Let's begin with a definition:

- A **base** (or **radix**) is an integer governing a counting system: each number is represented as powers of that base.

Example: Modern humans count using the base of $\mathbf{10}$, e.g.: $1029 = 1 \cdot 10^3 + 2 \cdot 10^1 + 9 \cdot 10^0$.

Fun question: Why do we use 10 as the base for counting? [***Hint***: something in our body relates to 10. What is it?]

How Does a Computer Count?

Now back to the computer, which is essentially a dumb pile of matter: the materials (chemical substances) from which the computer is built can, if correctly arranged and connected, distinguish between 2 events: a **relatively low electric current** and a **relatively high electric current** that flows through those materials. [The power/pressure of a current is called **voltage**.]

Engineers, who realized this rather simple ability of these substances, took advantage of this fact to build up complex connections/links using these chemical substances to represent, calculate, and store data.

Fun question: According to the above, what number does a computer use as a base for counting?

In Topic 3: Boolean Algebra & Logic Gates, when we discuss the Transistor device, we will explain what chemical substances it is built of, how much low is a 'low' current and how much high is a 'high' current, and how exactly a Transistor would behave in the case of being subject to a low electric current vs. a high electric current (and how we can use it to build a computer.)

How Does a Computer Count?

As an initial step, engineers agreed to represent the binary digit $\mathbf{0}$ using a low electric current, and the digit $\mathbf{1}$ using a high electric current.

- In other words, if a low electric current flows through the computer, it means we received the digit 0, and, on the other hand, if a high electric current flows through the computer, it means we got the digit 1.
- This is why the counting system of computers is built up of 0s and 1s: computers can only understand these two symbols.

Once this decision has been made, you can apply multiple currents in a row to represent a bunch of 0s and 1s, which, as we will see, could then represent numbers in other bases (e.g., base 10 and base 16,) letters, symbols, strings, objects like images, videos, and any other data structure that you can imagine (and implement.)

Now that we understood why the computer uses a **binary** counting system, let's define a few frequently-used computer science terms that we'll use throughout this course.

Binary System: Definitions

The word **bit** is a contraction of "binary digit", which is a data type that can be either 0 or 1. It was coined by Professor John W. Tukey of Princeton University in 1946.

A **byte** is a fixed number of bits attached to one another. The most common byte size is 8 bits, e.g., 01001110, although 7-bit bytes have certain meaningful applications in computing.

A **word** is a fixed number of bytes attached to one another. E.g., a `WORD` data type on Windows is of the size 16-bits (= 2 bytes), and a `DWORD` ("Double WORD") is 32-bits (= 4 bytes), twice as long as a `WORD`.

A **nibble** (sometimes called **nybble** or **nyble**) is 4 bits long: it's a half of an 8-bit byte.

Fun In-Class Activity: Assuming a byte is 8 bits long, how many bits long are (1) 64 KB? (2) 16 `DWORD`s? (3) 256 nibbles?

Binary System: Definitions

Any questions so far?

Converting between Bases

Suppose you see the binary number 01001110. What is its equivalent decimal (base 10) number? Conversely, how do you represent the decimal number 2619 in the binary system?

We will learn two methods: the **subtraction method** and the **division remainder method**. The division remainder method is the easiest and fastest of the two.

Before we see how these methods work, note that a **subscript** might be used sometimes to indicate in what base a number is written. For example, the number 101_{10} is the number 101 in the decimal system, while the number 101_2 is the binary sequence 101 written in base 2. [BTW, 101_2 is equal to the number 5_{10} in our decimal system.]

- Other examples of subscripts you will encounter are base 8 (= 'octal'), like 75_8, and base 16 (= 'hexa-decimal'), like $2\mathrm{B}_{16}$.
- The symbols of the octal system are: 0, 1, 2, 3, 4, 5, 6, and 7 (a total of 8 symbols,) and the symbols of the hexadecimal system are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F (a total of 16 symbols.)

Converting between Bases

The **subtraction method** works as follows: keep subtracting powers of the destination base from the number, until you get 0. Take note of which powers were used. Finally, write the multiples of all these subtracted powers in a string, which is your result!

Since we will frequently convert numbers to base 2, it is useful to keep a handy list of some common powers of 2:

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
1	2	4	8	16	32	64	128	256	512	$1,024$	$2,048$	$4,096$	$8,192$	$16,384$	$32,768$	$65,536$

Example: To convert 2619_{10} to base 2, we do the following:

Note that the biggest power of 2 that goes into 2619_{10} is $2^{11} = 2048$, so we subtract to get $2619 - 2048 = 571$. The biggest power that goes into 571 is $2^9 = 512$, so we get $571 - 512 = 59$. Keeping this way, we see that $59 - 32 = 27$, $27 - 16 = 11$, $11 - 8 = 3$, $3 - 2 = 1$, $1 - 1 = 0$. Next on, we will build up the resulting number.

Converting between Bases

In total, we saw that:

$$2619_{10} = 2048 + 512 + 32 + 16 + 8 + 2 + 1 = 2^{11} + 2^9 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0$$

Now that we know what powers of 2 'go' into 2619_{10}, we can write the string out.

If a power of 2 was 'present' in the calculation (e.g., 11), then we append the bit 1 to the string. If, conversely, a power of 2 was unused in the calculation (e.g., we didn't use 2^{10},) we add the bit 0 to the string. We keep adding these bits from left to right.

As such, since the bits we've seen are $11, 9, 5, 4, 3, 1$, and 0, these are the only bits in our string that will be 1: everything else will be 0. Our resulting string of bits is:

$$101000111011_2$$

Conclusion: 2619_{10} is 101000111011_2 in base 2.

Converting between Bases

Another example, with a concise solution:

$$1758_{10} = 1024 + 512 + 128 + 64 + 16 + 8 + 4 + 2 = 2^{10} + 2^9 + 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$$

The powers we've used are $10, 9, 7, 6, 4, 3, 2,$ and 1, so the result is:

$$11011011110_2$$

Conclusion: 1758_{10} is 11011011110_2 in base 2.

To convert from base 2 to base 10, you do the opposite actions, from right to left:

$$\begin{aligned} 110111010_2 &= 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 \\ &= 2 + 8 + 16 + 32 + 128 + 256 = 442_{10} \end{aligned}$$

Converting between Bases

Any questions so far?

Converting between Bases

Let's also practice converting to base 16. First, a handy list of common powers of 16:

16^0	16^1	16^2	16^3	16^4	16^5	16^6	16^7	16^8
1	16	256	$4,096$	$65,536$	$1,048,576$	$16,777,216$	$268,435,456$	$4,294,967,296$

Example: To convert $100,000_{10}$ to base 16, we do:

$$100000_{10} = 1 \cdot 65536 + 8 \cdot 4096 + 6 \cdot 256 + 10 \cdot 16 = 1 \cdot 16^4 + 8 \cdot 16^3 + 6 \cdot 16^2 + 10 \cdot 16^1 + 0 \cdot 16^0 = 186\text{A}0_{16}.$$

Note: Base 16 has 6 more symbols besides the digits $0-9$: A stands for 10, B stands for 11, C stands for 12, D stands for 13, E stands for 14, and F stands for 15. Examples demonstrating the use of these hexadecimal symbols:

- $13_{10} = 13 \cdot 1 = 13 \cdot 16^0 = \text{D}_{16}$.
- $242_{10} = 15 \cdot 16 + 2 \cdot 1 = 15 \cdot 16^1 + 2 \cdot 16^0 = \text{F}2_{16}$.

Converting between Bases

To convert from base 16 to base 10, you do the opposite actions, from right to left:

$$5\text{B}92\text{C}_{16} = 12 \cdot 16^0 + 2 \cdot 16^1 + 9 \cdot 16^2 + 11 \cdot 16^3 + 5 \cdot 16^4 = 12 + 32 + 2304 + 45056 + 327680 = 375,084_{16}.$$

The other base conversion method, the **division remainder method**, is faster and doesn't require you to look up powers of bases. You keep dividing the number by the base (e.g., 2, 16, etc.) and take note of the **remainders**: these will be the digits of the resulting number. This method is best for quickly converting numbers in base 10 to any base.

Examples: On the following slide, we convert 2619_{10} and 1758_{10} again to base 2, but this time using the division remainder method.

In these examples, note how we don't need to memorize any powers of 2: we just keep dividing by 2 and notice the remainders we get along the way.

Converting between Bases

The string of all the remainders, from bottom to top, when converting 2619_{10} to base 2 is 101000111011_2:

1. $2619/2$ gives 1309 with remainder 1
2. $1309/2$ gives 654 with remainder 1
3. $654/2$ gives 327 with remainder 0
4. $327/2$ gives 163 with remainder 1
5. $163/2$ gives 81 with remainder 1
6. $81/2$ gives 40 with remainder 1
7. $40/2$ gives 20 with remainder 0
8. $20/2$ gives 10 with remainder 0
9. $10/2$ gives 5 with remainder 0
10. $5/2$ gives 2 with remainder 1
11. $2/2$ gives 1 with remainder 0
12. $1/2$ gives 0 with remainder 1

The string of all the remainders, from bottom to top, when converting 1758_{10} to base 2 is 11011011110_2.

Notice the "R" short-hand writing for remainders.

1. $1758/2 = 879\mathrm{R}0$
2. $879/2 = 439\mathrm{R}1$
3. $439/2 = 219\mathrm{R}1$
4. $219/2 = 109\mathrm{R}1$
5. $109/2 = 54\mathrm{R}1$
6. $54/2 = 27\mathrm{R}0$
7. $27/2 = 13\mathrm{R}1$
8. $13/2 = 6\mathrm{R}1$
9. $6/2 = 3\mathrm{R}0$
10. $3/2 = 1\mathrm{R}1$
11. $1/2 = 0\mathrm{R}1$

Converting between Bases

Fun fact: Converting between bases of powers of 2 is even easier!

- To convert a number from base 2 to base 16, group every 4 bits together, and simply convert them to the respective hexadecimal symbol. Here is a shorthand table with all possible 4-bit configurations:

N_{16}	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	
N_2	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1

Example: $011111000010_2 = 0111\ 1100\ 0010_2 = 7\ C\ 2_{16} = 7C2_{16}$.

- Similarly, to convert from base 2 to base 8, group every 3 bits together and convert them to an octal digit:

N_8	0	1	2	3	4	5	6	7
N_2	000	001	010	011	100	101	110	111

Example: $011111000010_2 = 011\ 111\ 000\ 010_2 = 3\ 7\ 0\ 2_8 = 3702_8$.

Converting between Bases

Any questions so far?

How to Represent Signed Integers?

So far, we learned how to represent positive integers in different bases. However, how do we represent signed integers (including negative ones) using base 2?

We'll see 4 such ways: (1) sign & magnitude, (2) 1's complement, (3) 2's complement, and (4) biased system.

When using **sign & magnitude**, you use the leftmost bit to represent the sign (1 for 'negative' and 0 for 'positive',) while the rest of the bits contain the number itself.

Example: 101101_2 is negative 01101_2, which is -13_{10}. The leftmost 1 means it's a negative integer.

We can even **add** two integers together! We'll see a couple of examples on the following slide. Binary numbers are added just as decimal numbers are: we'll use column addition to show how everything works.

How to Represent Signed Integers?

Example: Adding the 6-bit numbers $7_{10} = 000111_2$ and $10_{10} = 001010_2$ (the result is, as we expected, $010001_2 = 17_{10}$:)

```
    111   <-- carry bits
0   00111
0 + 01010
    _____
0   10001
```

Note that the sign bits are listed to the left side of the addition. We don't touch them because both numbers are positive (= the sign bits are 0s,) so the result is itself positive.

Also, note that when we needed to add $1 + 1$, since there is no digit "2" in the binary system, we needed to 'carry' this bit over to the next bit addition, just like we carry numbers that are greater than 10 when adding base 10 numbers.

How to Represent Signed Integers?

Example of things getting wrong: Adding the 6-bit numbers $18_{10} = 010010_2$ and $23_{10} = 010111_2$:

```
1    11   <-- carry bits
0   10010
0 + 10111
   _____
1   01001
```

Look what happened: we got a negative number, which, obviously, isn't the correct sum.

The reason for this mishap is because we got a carry bit that was carried over to the sign bit, which mustn't happen: if this happens, we get an **overflow** situation, in which, the received result is incorrect. The 6-bit format is simply too small to represent the sum $18 + 23$: a format with more than 6 bits is needed to avoid the overflow.

How to Represent Signed Integers?

Any questions so far?

How to Represent Signed Integers?

If you need to add one positive and one negative number, you'll perform subtraction.

Example: Adding the 6-bit numbers $-6_{10} = 100110_2$ and $13_{10} = 001101_2$ requires to subtract 6_{10} from 13_{10}:

```
0   01101
1 - 00110
    _____
0   00111
```

The result we got is $-6 + (+13) = +7$.

The sign bit of the resulting number is always the sign bit of the number with the bigger magnitude from those we are adding: since $|+13| > |-6|$, the sign is positive (= 0.)

Problems with the Sign-Magnitude Method

Although the sign & magnitude method uses simple representation ideas (one bit for the sign and the rest for the number itself,) working with sign & magnitude numbers creates a few noticeable issues that we just can't ignore:

- Checking whether we are adding negative or positive numbers, and checking which one needs to be subtracted would take significant computational time (making the computer work slower.)
- When subtracting numbers, the decision of how many bits need to be 'borrowed' also takes significant time.
- The number 0 has **two** different representations, which shouldn't happen in a representation system: $00000...00_2$ and $10000...00_2$ (that is, 'positive zero': $+0$ and 'negative zero': -0.)

To prevent a computer from spending extra time on making decisions concerning adding or subtracting numbers, instead of doing these operations in a straightforward way, and to eliminate the ambiguity with the representation of 0, computers use **complement systems** to represent integers.

One's Complement Method

Deciding on whether you need to add or subtract numbers is nasty, cumbersome, and error-prone (if you do it by hand.)
Hence, a method called **one's complement** (or **1's complement**) was devised as works as follows:

- If your number is positive, its representation is the same as with the sign & magnitude method.
- Otherwise, if your number is negative, do the following:
  1. Find the sign & magnitude representation of its absolute value, and then
  2. Flip all the bits: change 0s to 1s, and 1s to 0s.

The action of flipping the bits is called 'applying 1's complement' because 1's complement is 0 (and vice versa.)

Example: The absolute value of -5_{10} is 5, so its sign & magnitude ("normal") representation in an 8-bit format is 00000101_2. The 1's complement format simply asks to flip the bits: 11111010_2. That's it!

One's Complement Method

Notes about the 1's complement method:

- Adding numbers became much simpler: regardless of whether you have positive or negative numbers that you need to add, you always do addition!
  - In case you get a carry bit of 1 beyond the leftmost bit, you don't discard it: just add this 1 back to the rightmost (smallest) bit. This is called "end-around carry".
- Unfortunately, we still have two representations of zero: positive zero (00000000) and negative zero (11111111). [We'll soon see another complement method that solves this problem!]

Fun Activity: Represent the following in 8-bit, 1's complement format: (1) 9_{10} (2) -9_{10} (3) -17_{10}.

Next, we'll check out some examples of (easily!) adding numbers in 1's complement format.

Adding Numbers with One's Complement Method

Example №1: Adding the 6-bit numbers $-6_{10} = 111001_2$ and $13_{10} = 001101_2$:

```
End-around carry:
|
v
1 11  1  <-- Carry Bits
  001101
+ 111001
  ______
  000110
+      1 <-- Adding the end-around carry
  ______
  000111 <-- What is this equal to in base 10?
```

Adding Numbers with One's Complement Method

Example №2: Adding the 6-bit numbers $7_{10} = 000111_2$ and $10_{10} = 001010_2$:

```
  111    <-- carry bits
  000111
+ 001010
  ______
  010001
```

Adding positive numbers is the same as with sign & magnitude (except that we add an end-around carry if there is such.)

Example №3: ***Fun Activity***: Let's add -3_{10} and -11_{10} in a 8-bit, 1's complement format!
[***Hint***: First, represent -3_{10} and -11_{10} in a 8-bit, 1's complement format, and then add them.]

Adding Numbers with One's Complement Method

Any questions so far?

Two's Complement Method

Recall that 1's complement still allows the existence of 2 forms for 0: $0000...000$ and $1111...111$. Another method called **two's complement** (or **2's complement**) solves this issue! 2's complement works in the same way as 1's complement except that you need to add 1 to every negative number you work with:

- If your number is positive, its representation is the same as with the sign & magnitude method.
- Otherwise, if your number is negative, do the following:
  1. Find the sign & magnitude representation of its absolute value, and then
  2. Flip all the bits: change 0s to 1s, and 1s to 0s.
  3. Add 1 to this number.

Example: The absolute value of -5_{10} is 5, so its sign & magnitude ("normal") representation in an 8-bit format is 00000101_2, the 1's complement representation is 11111010_2, and the 2's complement representation is: $11111010_2 + 00000001_2 = 11111011_2$. That's it!

Two's Complement Method

Notes about the 2's complement method:

- Adding numbers is even more simple: if you get an end-around carry, just discard it (you don't add it back to the result.) This improvement makes the computer run addition even faster!
- Zero (0) has only one representation: $0000...000$!
- 2's complement is what computers use nowadays to represent numbers!

Fun Activity №1: Represent the following in 8-bit, 2's complement format: (1) 9_{10} (2) -9_{10} (3) -17_{10}.

Fun Activity №2: Represent the following 2's complement numbers as decimal numbers: (1) 01001011_2 (2) 11010010_2 (3) 10000000_2.

Let's now see how to add numbers in 2's complement format.

Adding Numbers with Two's Complement Method

Example №1: Adding the 6-bit numbers $-6_{10} = 111010_2$ and $13_{10} = 001101_2$:

```
Discard the end-around carry:
|
v
1 11     <-- Carry Bits
  001101
+ 111010
  ______
  000111
```

We got $-6 + 13 = +7$! Yay!

Adding Numbers with Two's Complement Method

Example №2: Adding the 6-bit numbers $7_{10} = 000111_2$ and $10_{10} = 001010_2$:

```
  111    <-- carry bits
  000111
+ 001010
  ______
  010001
```

Adding positive numbers is the same as with sign & magnitude (except that you'd discard any end-around carry if there is such.)

Example №3: ***Fun Activity***: Let's add -3_{10} and -11_{10} in a 8-bit, 2's complement format!
[***Hint***: First, represent -3_{10} and -11_{10} in a 8-bit, 2's complement format, and then add them.]

Overflow

When the sum of 2 numbers is *too large* to contain, an **overflow occurs.** Here are 2 ways of detecting overflow:

- When adding 2 positive numbers, but the sum is negative, or when adding 2 negative numbers, but the sum is positive.
- When the carry bit into the sign bit is **different** from the carry bit out of the sign bit.

Example: Adding the 6-bit numbers $18_{10} = 010010_2$ and $23_{10} = 010111_2$:

```
0 1 11   <-- carry bits
  010010 <-- Positive
+ 010111 <-- Positive

  ______
  101001 <-- Negative?!?! We got overflow!
```

Overflow

Any questions so far?

Character Representation / Encoding

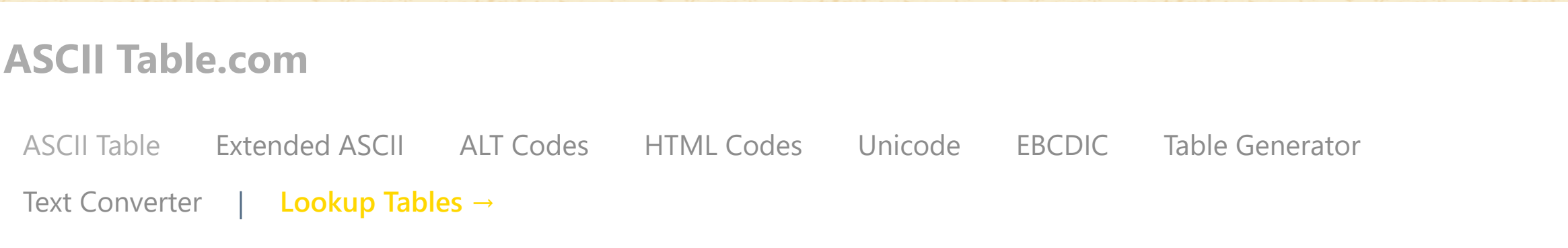
We now know how computers represent and store signed numbers. But what about characters and strings?

Several character representation systems (called **encoding systems**) have been developed and used since the middle of the 20th century, the two popular of which are ASCII and Unicode.

ASCII was developed in the early 1960s by a committee from the American National Standards Institute (ANSI). The first version of ASCII was published in 1963, and it was later updated in 1967 to include lowercase letters and more control characters. ASCII became widely adopted due to its simplicity and efficiency, particularly in early computers and communication systems. It was crucial in standardizing how devices exchanged text data, contributing to the growth of digital communication.

The final version of ASCII contains representations, in numeric format, for letters, numbers, punctuation marks, control characters, and certain other special characters. For example, the letter "A" is represented by the number 65 and "a" by 97.

Character Representation / Encoding



ASCII Table. Embedding https://www.lookuptables.com/text/ascii-table.

Character Representation / Encoding

As we've seen, the ASCII encoding contains representations for only 256 character types, mostly focusing on keyboard characters and Latin characters. How can letters in other languages be represented?

Unicode was created in 1991 to provide a single, consistent way to encode text from all languages, solving the problems of incompatibility and limited character representation that ASCII couldn't address:

- Before Unicode, there were many different character encodings used by different systems to represent non-English text, and these encodings could conflict. Text that is displayed correctly on one computer might appear garbled on another because they used different encoding standards.
- Unicode was developed to be a universal character encoding standard that could represent characters from all languages, along with symbols, emojis, and more. It assigns a unique code point to every character from every writing system, and it currently supports over **143,000** characters across multiple scripts and languages.
- Unicode uses different encoding forms like UTF-8, UTF-16, and UTF-32, allowing it to handle everything from simple ASCII characters (in 1 byte) to complex, multi-byte characters efficiently, depending on the use case.

Unicode Chart

Range	Decimal	Name
0x0000-0x007F	0-127	Basic Latin
0x0080-0x00FF	128-255	Latin-1 Supplement
0x0100-0x017F	256-383	Latin Extended-A
0x0180-0x024F	384-591	Latin Extended-B
0x0250-0x02AF	592-687	IPA Extensions
0x02B0-0x02FF	688-767	Spacing Modifier Letters

Unicode Table. Embedding https://www.ssec.wisc.edu/~tomw/java/unicode.html.

Character Representation / Encoding: Practice

Fun Class Activity:

1. Convert the following strings to binary ASCII:
    a. `Hello World!`
    b. `12345`
    c. `lol. LOL!`
2. Repeat the above for hexadecimal ASCII.

Note: ASCII characters have the same encoding in Unicode's UTF-8 format. That is, ASCII is a subset of UTF-8.

The C and C++ languages use ASCII for character encoding, while Java (and other) languages uses Unicode.

Character Representation / Encoding: Practice

Any questions so far?

Representation of Floating-Point Numbers

We learned how to represent integers (unsigned and signed) and characters in a computer. However, besides these data types, a computer must also work with fractions (also called **floating-point numbers** or **floats**.) Well, how does a computer represent floating-point numbers?

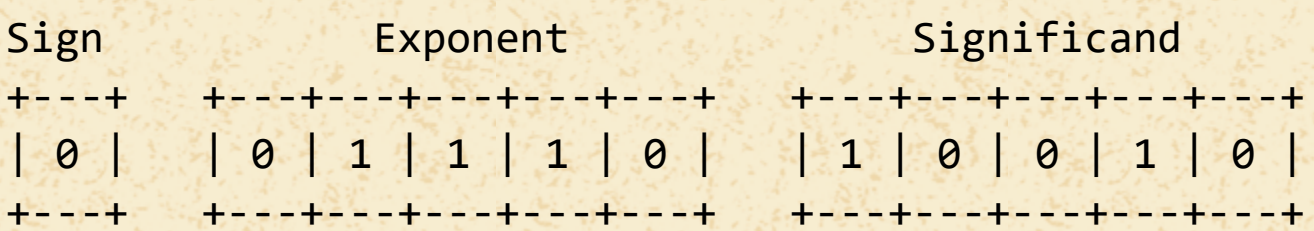
Every decimal number, whether it is an integer or a fraction, can be represented in **scientific notation**, e.g., $3400 = 3.4 \times 10^3$ and $-0.00012 = -1.2 \times 10^{-4}$. The 3.4 and -1.2 parts are called the **significand**.

In other words, in scientific notation, numbers are written as a product of a coefficient (between 1 and 10) and a power of 10. The exponent indicates how many places the decimal point has moved: a positive exponent for large numbers and a negative exponent for small ones.

Computers use a similar notation idea: we can represent floating-point numbers using a sign bit, an exponent, and a significand.

Representation of Floating-Point Numbers

The format of a floating point number in a computer will look similarly to the following:



Above, 1 bit (the leftmost) holds the sign bit, 5 bits store the exponent, and 5 bits store the significand.

The developers of a computer system decide how many bits are dedicated to the significand (which decides how precise numbers are) and to the exponent (which decides how large the range can get.) In other words, a different number of bits is dedicated to the exponent and significand on each computer system or language.

Representation of Floating-Point Numbers

We saw that the exponents of fractions are negative (e.g., $-0.00012 = -1.2 \times 10^{-4}$). Nonetheless, the sign bit we have is for the significand, not the exponent. So, how do we represent negative exponents?

To represent negative exponents, we use **bias** (also called **excess**,) which is a number that is added to each exponent to shift all the numbers forward. The bias will always be equal to $2^{\text{exponent bits}-1} - 1$, and will allow us to store negative numbers within the exponent bits, without the need to add an additional sign bit.

For example, in our 5-bit exponent format, the bias will be equal to: $2^{5-1} - 1 = 2^4 - 1 = 16 - 1 = 15$. This creates an **excess-15** exponent. Therefore, the exponent 1 will be represented as $1 + 15 = 16_{10} = 10000_2$, 2 as $17_{10} = 10001_2$, etc., while the exponent -5, for example, is represented as $-5 + 15 = 10_{10} = 001010_2$.

Note that we don't use 1's or 2's complement neither for the exponent nor for the significand. In fact, in our floating-point representation, both the exponent and significand are represented in a manner similar to the sign & magnitude format.

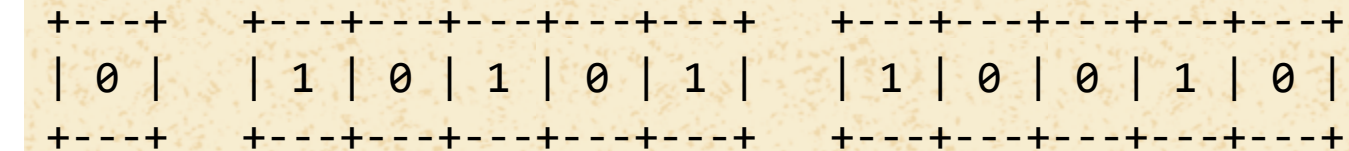
Representation of Floating-Point Numbers

Three possible representations for a 3-bit exponent. Miriam Briskman, CC BY-NC 4.0

Representation of Floating-Point Numbers

Final note before seeing examples: the significand contains the fraction part of a number (i.e., all the digits to the right of the binary point (.). For instance, a significand of 10001 represents the binary fraction 0.10001. The exponent will then magnify or diminish this number.

Example: The number represented by



can be understood as follows: (1) the number is positive since the sign bit is 0. (2) to figure out what the exponent is, 10101, subtract 15_{10} (the bias) from it. 10101 in binary is 21_{10}, so $21 - 15$ is 6, so the exponent is 6. (3) the significand is 10010, which is 0.10010 binary. Together, we get: $0.10010 \times 2^6 = 100100.0$, which is 36_{10}.

Representation of Floating-Point Numbers

Any questions so far?

Normalization

Some numbers can have 2 or more different representations.

Example: The number 36_{10} from the previous example can be represented by

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 1 | 0 | 1 | 0 | 1 |   | 1 | 0 | 0 | 1 | 0 | = 0.10010 x 2^6 = 36_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

or, equivalently, by

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 1 | 0 | 1 | 1 | 0 |   | 0 | 1 | 0 | 0 | 1 | = 0.01001 x 2^7 = 36_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

The representation in which the leftmost bit of the significand is 1 is called the **normalized** form. The process of expressing a given number in its normalized form is called **normalization**. In our example, 0.10010×2^6 is the normalized form of 36_{10}.

Floating-Point Numbers: Fractions

Let's now figure out what the number from slide 42 is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 0 | 1 | 1 | 1 | 0 |   | 1 | 0 | 0 | 1 | 0 |
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

Analysis: (1) the number is positive since the sign bit is 0. (2) to figure out what the exponent is, 01110, subtract 15_{10} (the bias) from it. 01110 in binary is 14_{10}, so $14 - 15$ is -1, so the exponent is -1. (3) the significand is 10010, which is 0.10010 binary. Together, we get: $0.10010 \times 2^{-1} = 0.010010 = 0.01 + 0.00001$, which is $\frac{1}{4} + \frac{1}{32} = 0.28125_{10}$.

Alternatively, note that the difference between the number 100100_2 (which we figured out is 36_{10}) and 0.010010_2 is that the latter is reached from the former by shifting the former 7 times to the right, which is the same as dividing the number by 2 seven times. Indeed: (1) $36/2 = 18$; (2) $18/2 = 9$; (3) $9/2 = 4.5$; (4) $4.5/2 = 2.25$; (5) $2.25/2 = 1.125$; (6) $1.125/2 = 0.5625$; and (7) $0.5625/2 = 0.28125_{10}$.

Representing Numbers in the Floating-Point Format

Let's convert the numbers (1) 5 (2) -18 (3) 0.5 (4) -0.25 (5) 0.375 to our floating-point number representation:

1. Analysis for 5: (1) it is positive, so the sign bit will be 0. (2) We know that $5_{10} = 101_2$, but since the significand represents a fraction form (i.e., $0.1xxx\ldots$,) we need to shift the digits of 101_2 several times to the right while increasing the exponent by 1 each time we shift: $101 \times 2^0 = 10.1 \times 2^1 = 1.01 \times 2^2 = 0.101 \times 2^3$. As such, the significand will be 10100. (3) Finally, for the exponent, we stopped at 2^3, whose exponent is 3. When we add the bias of 15, we get $3 + 15 = 18$, which makes our exponent equal to 10010. The number 5, therefore, in its floating-point form is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 1 | 0 | 0 | 1 | 0 |   | 1 | 0 | 1 | 0 | 0 | = 5_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

   Question: Is the floating-point representation for 5 above normalized or not? Explain why.

Representing Numbers in the Floating-Point Format

2. Analysis for -18: (1) it is negative, so the sign bit will be 1. (2) We know that $18_{10} = 10010_2$, but since the significand represents a fraction form (i.e., $0.1xxx\ldots$,) we need to shift the digits of 10010_2 several times to the right while increasing the exponent by 1 each time we shift: $10010 \times 2^0 = 1001.0 \times 2^1 = 100.10 \times 2^2 = 10.010 \times 2^3 = 1.0010 \times 2^4 = 0.10010 \times 2^5$. As such, the significand will be 10010. (3) Finally, for the exponent, we stopped at 2^5, whose exponent is 5. When we add the bias of 15, we get $5 + 15 = 20$, which makes our exponent equal to 10100. The number -18, therefore, in its floating-point form is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 1 |   | 1 | 0 | 1 | 0 | 0 |   | 1 | 0 | 0 | 1 | 0 | = -18_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

Representing Numbers in the Floating-Point Format

3. Analysis for 0.5: (1) it is positive, so the sign bit will be 0. (2) We know that $0.5_{10} = \frac{1}{2}_{10} = 2^{-1}_{10} = 0.1_2$. Thankfully, the significand is already in the form $0.1xxx\ldots$, so we just notice that it is equal to $0.1 \times 1 = 0.1 \times 2^0$. As such, the significand will be 10000. (3) Finally, for the exponent, we stopped at 2^0, whose exponent is 0. When we add the bias of 15, we get $0 + 15 = 15$, which makes our exponent equal to 01111. The number 0.5, therefore, in its floating-point form is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 0 | 1 | 1 | 1 | 1 |   | 1 | 0 | 0 | 0 | 0 | = 0.5_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

It's not always easy to recognize powers of 2 as we did with 0.5. As such, when we get a fraction in base 10 and want to find its binary representation, we could use a fraction multiplication trick. It goes as follows: (1) multiply the fraction by 2, and discard any integer parts (2) in each step, before discarding it, take note of the integer part of the number: whether it is 0 or 1, (3) repeat steps (1) and (2) until the fraction is empty (the fraction part is only zeros.) We will use this method for the next examples.

Representing Numbers in the Floating-Point Format

4. Analysis for -0.25: (1) it is negative, so the sign bit will be 1. (2) The binary form for 0.25 is computed as follows:
   - $0.25 \times 2 = 0.5$. The integer part in the result is 0, so use $0.5 - 0 = 0.5$ for the next step.
   - $0.5 \times 2 = 1.0$. The integer part in the result is 1, so use $1.0 - 1 = 0.0$ for the next step.
   - Since 0.0 is just 0, we stop here.

   The integer parts from the steps above make up the fraction part of the binary number we are looking for. As such, the binary form for 0.25, from top to bottom, is $0.01 = 0.01 \times 2^0 = 0.1 \times 2^{-1}$. The significand will, thus, be 10000. (3) Finally, for the exponent, we stopped at 2^{-1}, whose exponent is -1. When we add the bias of 15, we get $-1 + 15 = 14$, which makes our exponent equal to 01110. The number -0.25, therefore, in its floating-point form is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 1 |   | 0 | 1 | 1 | 1 | 0 |   | 1 | 0 | 0 | 0 | 0 | = -0.25_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

   [Note: we may have also noticed that $0.25 = \frac{1}{4} = \frac{1}{2^2} = 2^{-2} = 0.01_2$..]

Representing Numbers in the Floating-Point Format

5. Analysis for 0.375: (1) it is positive, so the sign bit will be 0. (2) The binary form for 0.375 is computed as follows:
   - $0.375 \times 2 = 0.75$. The integer part in the result is 0, so use $0.75 - 0 = 0.75$ for the next step.
   - $0.75 \times 2 = 1.5$. The integer part in the result is 1, so use $1.5 - 1 = 0.5$ for the next step.
   - $0.5 \times 2 = 1.0$. The integer part in the result is 1, so use $1.0 - 1 = 0.0$ for the next step.
   - Since 0.0 is just 0, we stop here.

   As such, the binary form for 0.375, from top to bottom, is $0.0\,1\,1 = 0.011 \times 2^0 = 0.11 \times 2^{-1}$. The significand will, thus, be 11000. (3) Finally, for the exponent, we stopped at 2^{-1}, whose exponent is -1. When we add the bias of 15, we get $-1 + 15 = 14$, which makes our exponent equal to 01110. The number 0.375, therefore, in its floating-point form is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 0 | 1 | 1 | 1 | 0 |   | 1 | 1 | 0 | 0 | 0 | = 0.375_{10}
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

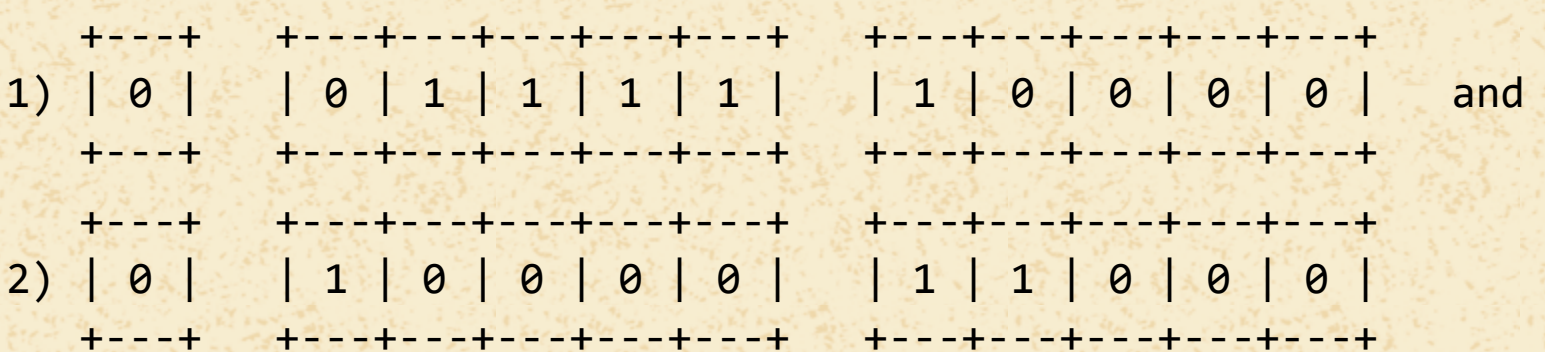
Question: 0.375 is the sum of which two negative powers of 2? In other words, what are m and n in $0.375 = \frac{1}{2^m} + \frac{1}{2^n}$?

Representing Numbers in the Floating-Point Format

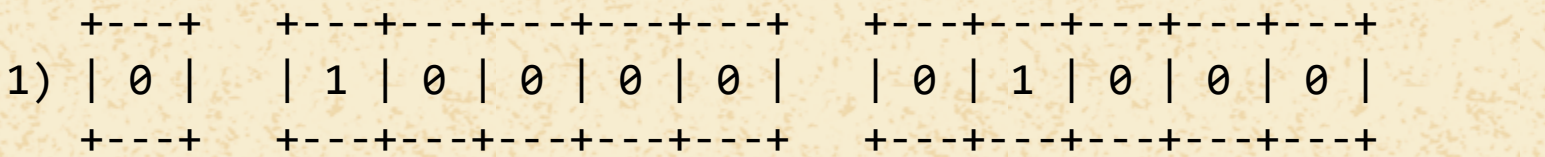
Any questions so far?

Adding and Multiplying Floating-Point Numbers

If you want to add two floating-point numbers, say:



you will first shift one of them such that its exponent becomes the same as the other number's exponent. In this case, you will take the 1st number and shift the 1 of its significand 10000 to the right and increase the exponent by 1 to get:



Adding and Multiplying Floating-Point Numbers

Now that both the exponents match, we will add the significands:

```
  1  1     <-- Carry bits
  0.01000
+ 0.11000
  _______
  1.00000
```

Finally, we need to normalize the resulting number and make it fit into the given floating-point format. We have 1.0 with the exponent 10000, but we need the significand to be of the form $0.1xxx\ldots$, so we divide 1.0 by 2 (= shift to the right once) to get 0.1, and, at the same time, increase the exponent 10000 by 1 (to keep the number balanced) and get 10001. The result is:

```
+---+   +---+---+---+---+---+   +---+---+---+---+---+
| 0 |   | 1 | 0 | 0 | 0 | 1 |   | 1 | 0 | 0 | 0 | 0 | --> What is this number in decimal?
+---+   +---+---+---+---+---+   +---+---+---+---+---+
```

Adding and Multiplying Floating-Point Numbers

No questions that ask to multiply floating-point numbers or integers will be given on assignments and exams. However, you should be aware of *how* multiplications of floating-point numbers are done:

When you are given two floating-point numbers (in a certain floating-point format), to multiply these numbers, you need to:

1. Multiply the two significands.
2. Add the two exponents.
3. Normalize the resulting number to make it fit into the given format.

Range and Accuracy

Range is defined as the difference between the largest and smallest numbers that can be represented in a given format.

- The largest (positive) number has 0 as its sign bit, while both the exponent and the significand are populated with just 1s.
- The smallest positive number has 0 as its sign bit, while the exponent is populated with all 0s, and the significand is populated with 0s, too, except for its leftmost bit, which must be 1 (in its normalized form.)
- The smallest negative number has 1 as its sign bit, while both the exponent and the significand are populated with just 1 s. (It's literally the negative of the largest positive number.)

Accuracy tells how close a number is to another number.

Unfortunately, because computers can store only a limited amount of data (our disks don't have infinite capacities,) this means that we can't represent each and every number out there in the universe. For example, you can't store the number $\pi \approx 3.14159265\ldots$, which has an infinite number of digits after the decimal point.

Range and Accuracy

When you are given a number and asked to store it in a certain floating-point format, but the number doesn't fit into the format, you'll, unfortunately, lose some data when you store it. You can, however, store the most accurate (close) number possible, which will reduce the loss of data.

Example: In the 1-bit sign, 3-bit exponent (with a bias of 3), and 3-bit significand format, we want to represent $11_{10} = 1011_2$, which we write as $1011 \times 2^0 = 101.1 \times 2^1 = 10.11 \times 2^2 = 1.011 \times 2^3 = 0.1011 \times 2^4$. Unfortunately, since we can use only 3 bits for the significand, we realize that the 4 bits of 1011_2 won't fit into the significand, so we will lose data.

To minimize the data loss, we should store the most significant bits of the number. In our case, these will be the 3 leftmost bits of 1011, which are 101. With a bias of 3, the exponent will be $4 + 3 = 7_{10} = 111$. We, hence, get:

```
+---+   +---+---+---+   +---+---+---+
| 0 |   | 1 | 1 | 1 |   | 1 | 0 | 1 | --> What number did we actually store here?
+---+   +---+---+---+   +---+---+---+
```

Range and Accuracy

Any questions so far?

Data Representation: Sources

Bergmann, Seth D. (2018). Chapter 2 and Appendix: ASCII Character Set. *Computer Organization with MIPS*. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

CS301: Computer Architecture. Unit 2.2 and Unit 4.1. *Saylor Academy*, 2010. License: CC BY: Attribution.

Tarnoff, David. (2020). Episodes 2.1, 2.2, 2.3, 2.4, 2.5, 3.01, 3.02, 3.03, 3.04, 3.05, 3.06, 3.07, 3.08, and 3.09. *Computer Organization and Design Fundamentals Series*. License: CC BY-NC-SA: Attribution-NonCommercial-ShareAlike.

Wienand, Ian. (2019). Chapter 2. *Computer Science from the Bottom Up*. License: CC BY-SA: Attribution-ShareAlike.