CS 241

Foundations of Sequential Programs

Course Website and Piazza

- The course website is: https://student.cs.uwaterloo.ca/~cs241/
  - You can find it by searching for CS 241 in your favourite search engine.
- Read the course logistics page and the course outline.
  - The outline is available on https://outline.uwaterloo.ca.
- Two assignment questions have been posted already.
  - You can start on them if you want. They are due next Friday.
- Piazza is where important announcements will be made: https://piazza.com/uwaterloo.ca/fall2023/cs241
  - Make sure you get enrolled in Piazza as soon as possible.

Course Staff and Office Hours

- **Sylvie Davies** (me)
  - Mon 3:30-4:30pm in-person (DC 3133), Fri 3:30-4:30pm online (Teams)
- **Edward Lee** (instructor for the other two sections)
  - Tue/Thu 4-5pm, in-person (DC 3548)
- Instructional Support Assistant: **Evan Girardin**
  - Thu 1-2pm and Fri 11am-12pm, in-person (MC 4065)
- Instructional Support Tutor (part time): **Deven Wolff**
  - Tue 1-2pm in-person (MC 4065), Wed 11am-12pm online (Teams)
- Instructional Apprentice: **Kris Frasheri**
- Instructional Support Coordinator: **Gang Lu**

Course Notes

- Course notes are available on the course website.
- They are being updated for this term and it is a work in progress. New chapters will be released periodically.
- Right now, the first 2 chapters are available.
- The course notes should be a useful reference, but are not meant to be a substitute for lectures.
- We may cover topics in lectures that are not in the notes.
- You are expected to know all material covered in lectures, unless we specifically say it's optional.

Evaluation Structure

- Assignments are worth 30%:
  - 10 shorter “questions”, worth 1% each
  - 5 larger “projects”, worth 4% each
- Both questions and projects will be submitted to Marmoset and automatically tested and graded (no hand-marking).
- Midterm exam worth 30%, scheduled for Wed Oct 25, 7:00-8:50pm.
- Final exam worth 40%, not yet scheduled.
- **You must pass the weighted average of the midterm and final exams to pass the course.**

Late Policy

- Assignments can be submitted late up until the last day of classes.
- Your mark will be the average of your best on-time mark and your best overall (on-time or late) mark.
- Assignments often build on each other, so we want to encourage you to try to complete all of them.
- Particularly, try to complete all the projects because they come together to produce a compiler for a simple high-level language! Building your own complete compiler can be very satisfying.

Bonus Marks

- A bonus of up to 5% on your final grade can be earned as follows:
  - Up to 3.5% for attending lectures and completing feedback surveys.
  - Up to 1.5% for answering bonus assignment questions, and for “linguistic diversity” when implementing the projects (using a mix of C++ and Racket).
- The course website contains the exact breakdown of bonus marks.
- To earn the survey bonus marks, you need to actually attend lectures. We will check attendance using iClicker Cloud.
  - We’re still working out the details of this; we’ll make a Piazza announcement soon. For attendance checking, iClicker Cloud does not cost money for students and a mobile device or laptop can be used for check-ins.

Tutorials

- There are tutorials on Wednesdays where the ISA and IA will cover additional material.
- Tutorials are optional, but it’s recommended you attend unless you are doing very well in the course and don’t need extra help.
- Tutorials will generally be focused on working through examples or practice problems related to course content.
- Tutorial-exclusive material is not tested on assignments or exams.
- The first tutorial will be next week. There was no tutorial yesterday.

Goal of the Course

- How does code in a high-level language like C actually get executed?
- **Compilation:** A program called a **compiler** translates the code into an executable file containing low-level instructions for the computer.
    - C and C++ are compiled languages.
- **Interpretation:** A program called an **interpreter** reads the code and performs the actions specified by the code.
    - Racket and Python are interpreted languages.
- You can write interpreters in interpreted languages, but at some point the computer needs those low-level instructions to know what to do.
- The goal of this course is to learn about the **process of compilation**.

Data Representations

Abstraction

- To abstract something is to remove details.
- In computer science, the goal is usually to remove irrelevant or unimportant details and focus on what is essential.
- It’s often said that computers store data as 0s and 1s (binary).
- Computers actually store data using two-state electronic circuits.
- But the construction and physical properties of these circuits are generally irrelevant to computer scientists.
- By abstracting the details away, we can focus on how patterns of these two states (0 and 1) let us store different kinds of data.

Bits

- Through abstraction, a computer’s memory can be thought of as a huge collection of 0s and 1s.
- To do anything useful with computers, we need to build more abstractions.
- An individual **bit** (binary digit, 0 or 1) can only represent two distinct pieces of data.
- By grouping bits together, we can use patterns of bits to represent larger collections of data.
- Numbers, text, programs…

Common Groupings of Bits

- A **byte** is a sequence of 8 bits, e.g., 01101001.
- A **nibble** is a sequence of 4 bits.
- A **word** is a sequence whose length is context-dependent, based on the computer architecture being used.
- Most modern computer architectures use 64-bit words, but in this course, **a word will be a sequence of 32 bits**.

  00000011111000000000000000001000

- Game consoles used to use word size as a marketing point. The Nintendo 64 even included the word size in the console name.

Representing Numbers

- If we want to do calculations on a computer, we need a way to represent numbers using groupings of bits.
- A grouping of *n* bits can represent 2^n different values.
- We'll focus on representations of *integers.*
- A simple one is **base 2 representation** for non-negative integers.
- Base 10 (decimal) : $1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$
- Base 2 (binary): $10101 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- With *n* bits, we can represent integers from 0 to $2^n – 1$.
- This is also called **unsigned representation** in a computing context.

Hexadecimal Notation

- Base 16, **hexadecimal**, is also widely used in computer science. Why?
- Since $16 = 2^4$, each hexadecimal digit corresponds to exactly four bits.

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hex	0	1	2	3	4	5	6	7
Binary	1000	1001	1010	1011	1100	1101	1110	1111
Hex	8	9	A	B	C	D	E	F

- This means hexadecimal is useful as a **shorthand** for writing down long sequences of binary data.
- The prefix 0x is often used for hexadecimal, e.g., 11110001 = 0xF1.

Hexadecimal Notation: Example

- Here we have a 32 bit binary sequence:

```
00011011101011011111000000001101
```

- Break it into nibbles, and convert each nibble to hex:

```
0001 1011 1010 1101 1111 0000 0000 1101
   1    B    A    D    F    0    0    D
```

```
00011011101011011111000000001101 = 0x1BADF00D
```

- We can convert from hex to binary in the same way:

```
0xDEADBEEF → D    E    A    D    B    E    E    F
           → 1101 1110 1010 1101 1011 1110 1110 1111
```

Fixed-Width Representations

- In unsigned representation, larger numbers require more bits to represent them.
- The number of bits used to represent a value is often called the **width** of the value.
- In practice, computer architectures prefer to operate on groupings of bits with the same width.
- It is usually faster and often mixed-width operations are not even provided.
- We will focus on *fixed-width representations* of values in this course.

Fixed-Width Unsigned Representation

- We specify a fixed-width unsigned representations by stating the number of bits, e.g., “8-bit unsigned”.
- Recall: With *n* bits, we can represent integers from 0 to $2^n - 1$.
- In C++, the “unsigned int” type is 32 bits on most systems, so the maximum representable value is $2^{32} - 1 = 4294967295$.
- What happens if you type this on a system with 32-bit unsigned ints?

```
unsigned int uh_oh = 4294967296;
```

- You could declare that this is an error, or undefined behaviour.
- A common solution is to instead use *modular arithmetic*.

Modular Arithmetic Review

- *Arithmetic modulo n* can be thought of as integer arithmetic that “loops around” every *n* values.
- “24-hour clock arithmetic” is a simple example: 23:00 + 02:00 = 01:00
- More formally, arithmetic modulo *n* is performed on *equivalence classes,* which are sets of integers, instead of plain integers.
- $[m] = \{m + nk : k \in \mathbb{Z}\}$ is the set of integers equivalent to *m* modulo *n.*
- So $[0] = [n] = [2n] = \ldots$ and $[1] = [n+1] = [2n+1] = \ldots$
- We can define addition, subtraction and multiplication (not division) on these classes and it makes sense, e.g., $[a] + [b] = [a + b]$.

Fixed-Width and Modular Arithmetic

- We can interpret *n*-bit unsigned representation in two ways:
  - As representing *integers* from 0 to $2^n - 1$.
  - As representing the *equivalence classes* of these integers modulo 2^n.
- We are flexible, and switch our interpretation depending on context.

unsigned int uh_oh = 4294967295 + 1;

- Here we do the arithmetic modulo 2^{32}:

$$[4294967295] + [1] = [4294967295 + 1] = [4294967296] = [0]$$

- But if we wanted to *print* uh_oh, we would forget the equivalence class interpretation, and print a single integer in the range 0 to $2^n - 1$.

Unsigned Arithmetic and Overflow

- Let’s say we wanted to add the 8-bit unsigned integers 10010110 and 01110010, and express the result as an 8-bit unsigned integer.
- We could recast them as equivalence classes of decimal numbers:

  10010110 = [150] and 01110010 = [114]

  [150] + [114] = [264] = [8] (modulo $2^8 = 256$) which is 00001000.
- But we can also just do “grade school addition”:
- The result is 9 bits – **integer overflow**.

$$\begin{array}{cccccccccc}
 & 1 & 1 & 1 & 1 & & 1 & 1 & & \\
 & & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
+ & & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}$$

Unsigned Arithmetic and Overflow

- Let’s say we wanted to add the 8-bit unsigned integers 10010110 and 01110010, and express the result as an 8-bit unsigned integer.
- We could recast them as equivalence classes of decimal numbers:

  10010110 = [150] and 01110010 = [114]

  [150] + [114] = [264] = [8] (modulo 2^8 = 256) which is 00001000.
- But we can also just do “grade school addition”:
- The result is 9 bits – **integer overflow**.
- Discarding extra bits beyond the 8th has the same effect as reducing modulo 2^8.

$$
\begin{array}{cccccccccc}
 & 1 & 1 & 1 & 1 & & 1 & 1 & & \\
 & & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
+ & & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
\hline
 & \cancel{1} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

Binary and Decimal Conversions

- Binary to Decimal:
  - Write down the powers of 2 corresponding to each “1” bit and add them up.
- Decimal to Binary:
  - Greedy method: Take the largest power of 2 that fits in the decimal number, subtract it, and repeat until you’ve written the number as a sum of powers.
  - Division method: Repeatedly divide the decimal number by 2 and keep track of the remainders, which will always be 0 or 1. Once the decimal number is reduced to 0, read remainders from last to first to obtain the binary number.
- What is 11110001 in decimal?
  - $2^7 + 2^6 + 2^5 + 2^4 + 2^0 = 128 + 64 + 32 + 16 + 1 = 160 + 80 + 1 = 241$ ☺

Binary and Decimal Conversions: Examples

- Convert 170 to binary using the greedy method.

  170 → 128 + 42 → 128 + 32 + 10 → 128 + 32 + 8 + 2 → $2^7 + 2^5 + 2^3 + 2^1$

- Result: 10101010
- Convert 203 to binary using the division method.

```
203 / 2 = 101 r1
101 / 2 = 50  r1
 50 / 2 = 25  r0
 25 / 2 = 12  r1
 12 / 2 = 6   r0
  6 / 2 = 3   r0
  3 / 2 = 1   r1
  1 / 2 = 0   r1
```

- Result: 11001011

Abstraction, Again

- We abstracted away the physical details of the electronic circuits in a computer to view everything as 0s and 1s.
- Unsigned representation is a further abstraction: We view groupings of 0s and 1s as representing numbers (or equivalence classes).
- It is important to remember in this course that we are building abstractions, not truths or universal laws.
- The statement "11110001 in binary is 241 in decimal" is only true in the context of the *unsigned representation* abstraction that we built.
- We can interpret 11110001 in other ways – like as a negative number.

Representing Negative Integers

- In base 10 notation, we represent negative integers by just appending a negative sign: -1234
- As mathematicians, we can do the same thing for base 2: -10101
- As computer scientists, we cannot!
- Computers use two-state electronic circuits.
- We would need three states (0, 1, and negative sign) to represent negative base 2 numbers in the mathematical way.
- We need to represent negation using just two states, somehow.

The Obvious Way: Sign-Magnitude

- Probably the first solution most people would think of is to use a fixed-size representation, but reserve one bit to mean “negative sign”.
- This is called **sign-magnitude representation**.
- In 8-bit sign-magnitude representation:
  - 00001000 is 8 and 10001000 is -8
  - 01111111 is 127 and 11111111 is -127
  - 00000000 is 0 and 10000000 is also 0 (negative zero?)
- We can no longer use “grade school arithmetic”:
  - 8 + (-8) should be 0, but 00001000 + 10001000 gives 10010000 = -16.
- Sign-magnitude is flawed and only saw use in early computers.

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

Below: Adding machine keypad with **nines' complements** as subscripts. The nines' complement of a number is given by replacing each digit with 9 minus the digit.

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

To perform addition, enter the numbers normally.

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	1
+	2	4	0
	4	8	1

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

To perform subtraction, enter the **nines' complement** of the first number, then read off the **nines' complement** of the result.

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	1
+	2	4	0
	4	8	1

	7_2	5_4	8_1
+	2	4	0
	9_0	9_0	8_1

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

Alternatively, enter the **ten's complement** (nines' complement plus 1) of the second number, and ignore the final carry (if any).

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	1
+	2	4	0
	4	8	1

	7_2	5_4	8_1
+	2	4	0
	9_0	9_0	8_1

	2	4	1
+	7_2	5_4	9_0

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

Alternatively, enter the **ten's complement** (nines' complement plus 1) of the second number, and ignore the final carry (if any).

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	1
+	2	4	0
	4	8	1

	7_2	5_4	8_1
+	2	4	0
	9_0	9_0	8_1

	2	4	1	
+	7	6	0	+1

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

Alternatively, enter the **ten's complement** (nines' complement plus 1) of the second number, and ignore the final carry (if any).

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	1
+	2	4	0
	4	8	1

	7_2	5_4	8_1
+	2	4	0
	9_0	9_0	8_1

	2	4	1
+	7	6	0
1	0	0	1

The Method of Complements

- Centuries before the invention of computers and digital calculators, many intricate mechanical calculators were developed.
- It was (and still is!) useful to be able to use the same hardware to perform addition and subtraction.

Alternatively, enter the **ten’s complement** (nines’ complement plus 1) of the second number, and ignore the final carry (if any).

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	1
+	2	4	0
	4	8	1

	7_2	5_4	8_1
+	2	4	0
	9_0	9_0	8_1

	2	4	1
+	7	6	0
$\cancel{1}$	0	0	1

Complements and Negative Numbers

- What happens if the result is negative? Nine’s complement method:

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	0
+	2	4	1
	4	8	1

	7_2	5_4	9_0
+	2	4	1
1_8	0_9	0_9	0_9

Ignore the carry?

- Ten’s complement method:

1_8	2_7	3_6	
4_5	5_4	6_3	0_9
7_2	8_1	9_0	

	2	4	0
+	7_2	5_4	8_1

	2	4	0
+	7	5	9

+1

	2	4	0
+	7	5	9
			9	9	9

Representing Negative Numbers

- For an n-digit number m, the nines' complement is $10^n - 1 - m$.
- Therefore, the ten's complement is $10^n - m$.
- Working in a system with a maximum of n decimal digits, and ignoring overflowing digits, is essentially the same as working **modulo 10^n**.
- So in the context of a fixed number of decimal digits, **ten's complement behaves exactly like (modular arithmetic) negation**.
- The n-digit tens' complement of 1 is 999…99 (n nines).
- If we treat this as a **representation of -1**, the ten's complement addition on the previous slide becomes correct.

Two’s Complement

- Nines’ complement and ten’s complement are tied to base 10.
- We can apply the same ideas to base 2 to obtain **ones’ complement** and **two’s complement** (ones’ complement plus 1).
- When working with *n*-bit numbers:
  - Taking the two’s complement behaves exactly like negation modulo 2^n.
  - Negative numbers can be represented by taking the two’s complement of the corresponding positive number.
- Notice that ones’ complement is just “flipping the bits”, since 1 – 1 = 0 and 1 – 0 = 1. This makes ones’ complement and two’s complement easy to compute.

Representing Integers in Two’s Complement

- What does the 8-bit sequence 11111111 represent?
- In 8-bit unsigned, it is 255.
- Let’s take the two’s complement (flip the bits and add one):

  11111111 → 00000000 → 00000001
- So the representation of -255 is 00000001.
- …This is also the representation of positive 1.
- That’s okay, because [1] = [-255] modulo 2^8 = 256.
- But if we were to print out this integer, we would have to decide whether it’s more reasonable to print out 1 or -255.

Relationship to Unsigned Representation

- Recall that we can interpret *n*-bit unsigned in two ways:
  - Bit patterns are *integers* from 0 to $2^n - 1$.
  - Bit patterns are *equivalence classes* of these integers modulo 2^n.
- In *n*-bit two's complement, the interpretation as equivalence classes works the same as for *n*-bit unsigned.
- But should we do when interpreting bit patterns as integers?
- We want the range of representable integers to be balanced (each positive number is matched with a negative number).
- Sadly, this is impossible. There are 2^n numbers and one of them is 0.

The Range of Representable Integers

- In *n*-bit unsigned representation, we chose 0 to $2^n - 1$ as the range of representable integers.
- For *n*-bit two's complement, we will use $\mathbf{-2^{n-1}}$ to $\mathbf{2^{n-1} - 1}$.
- There are 2^n integers available, so we shift the range of representable integers to the left by 2^{n-1} (half of the available integers).
- This ensures every positive integer has a negative counterpart.
- However, one negative number (-2^{n-1}) has no positive counterpart!
- In unsigned, 2^{n-1} is 1000…000. If we take the two's complement:

1000…000 → 0111…111 → 0111…111 + 1 = 1000…000 (no change!)

Two's Complement: Additional Notes

- "Flipping the bits and adding 1" is equivalent to "flipping the bits to the left of the rightmost 1".
  - Flip the bits and add 1: 00101000 → 11010111 → 11011000
  - Flip the bits to the left of the rightmost 1: 00101000 → 11011000
- Two's complement can be thought of as a variant of fixed-size unsigned representation where the leftmost bit has *negative weight*.
  - *n*-bit unsigned: $2^{n-1} \cdot b_{n-1} + 2^{n-2} \cdot b_{n-2} + \ldots + 2^{1} \cdot b_{1} + 2^{0} \cdot b_{0}$
  - *n*-bit two's complement: $-2^{n-1} \cdot b_{n-1} + 2^{n-2} \cdot b_{n-2} + \ldots + 2^{1} \cdot b_{1} + 2^{0} \cdot b_{0}$

  $11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$
- The leftmost bit indicates sign! (1 for negative, 0 for non-negative)

Two's Complement: Summary

- A fixed-size integer representation that includes negative integers.
- Negative integers are represented by taking the "two's complement" of the corresponding positive integer (flip the bits and add 1).
- We can reuse hardware used for unsigned arithmetic.
  - Addition and subtraction are identical between unsigned / two's complement.
  - Multiplication is almost identical (more on this later). Division is different.
- The range of *n*-bit two's complement is -2^{n-1} to $2^{n-1} - 1$.
  - The smallest number, -2^{n-1}, has no positive counterpart. Be careful.
- Like unsigned but the leftmost bit represents a negative power of 2.

Abstractions So Far

- Computers store everything as binary data (0s and 1s, bits).
- We can view groupings of 0s and 1s as numbers.
- Using *unsigned representation* we can represent non-negative integers in the range 0 to $2^n - 1$ as sequences of *n* bits.
- Using *two's complement representation* we can represent integers in the range -2^{n-1} to $2^{n-1} - 1$ as sequences of *n* bits.
- But how do we represent **the text that you're reading right now?**
- That's outside the scope of this course but we can talk about a simple (yet still relevant) representation of text called **ASCII**.

ASCII Representation for Text

- The *American Standard Code for Information Interchange* (ASCII) was developed in the 1960s based on telegraph codes.
- ASCII originally used *7-bit* codes to represent characters, giving 2^7 = 128 different characters, enough for English text but not much else.
- On modern computers, it’s more convenient to group things into bytes, so we use 8 bits (one byte) per ASCII character.
- The modern standard for text is Unicode.
- Unicode is backwards-compatible with ASCII, but also allows multi-byte characters, to support other languages and special symbols.

ASCII Overview: Letters, Digits, Punctuation

- **Uppercase A to Z:** 01000001 (0x41) to 01011010 (0x5A)
- **Lowercase a to z:** 01100001 (0x61) to 01111010 (0x7A)

If you add 00100000 (0x20) to the code of an uppercase letter, you get its lowercase counterpart!

- **Digits 0 to 9:** 00110000 (0x30) to 00111001 (0x39)

To get the code for digit *n*, take the code for digit 0 and add *n*.

- **Space character:** 00100000 (0x20)
- The following punctuation characters are available:

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

ASCII Overview: Control Characters

- When ASCII was developed, electromechanical devices like *teleprinters* were still in common use.
- ASCII included a number of *control characters* which were not printed, but could be used to manipulate these devices.
- Only a handful of these still do anything on modern computers.
- When a program is asked to display a control character that no longer has a function, various things can happen:
  - It could display nothing.
  - It could display some kind of code that represents the control character.
  - It could display a special symbol representing the control character.

Table of Control Characters

Character	Code	Character	Code	Character	Code	Character	Code
Null	00000000 (0x00)	Backspace	00001000 (0x08)	Data Link Escape	00010000 (0x10)	Cancel	00011000 (0x18)
Start of Heading	00000001 (0x01)	Horizontal Tab	00001001 (0x09)	Device Control 1	00010001 (0x11)	End of Medium	00011001 (0x19)
Start of Text	00000010 (0x02)	Line Feed	00001010 (0x0A)	Device Control 2	00010010 (0x12)	Substitute	00011010 (0x1A)
End of Text	00000011 (0x03)	Vertical Tab	00001011 (0x0B)	Device Control 3	00010011 (0x13)	Escape	00011011 (0x1B)
End of Transmission	00000100 (0x04)	Form Feed	00001100 (0x0C)	Device Control 4	00010100 (0x14)	File Separator	00011100 (0x1C)
Enquiry	00000101 (0x05)	Carriage Return	00001101 (0x0D)	Negative Acknowledgement	00010101 (0x15)	Group Separator	00011101 (0x1D)
Acknowledgement	00000110 (0x06)	Shift Out	00001110 (0x0E)	Synchronous Idle	00010110 (0x16)	Record Separator	00011110 (0x1E)
Bell	00000111 (0x07)	Shift In	00001111 (0x0F)	End of Transmission Block	00010111 (0x17)	Unit Separator	00011111 (0x1F)
						Delete	01111111 (0x7F)

Most of these are no longer useful, but some still retain functionality in modern computers.

Summary

- What does the byte 10000000 (or 0x80 in hexadecimal) represent?
- It could be the 8-bit unsigned value 128.
- It could be the 8-bit two's complement value -128, the smallest 8-bit two's complement number, which has no positive counterpart.
- We could also think of it as representing the equivalence class [128] modulo $2^8 = 256$. This equivalence class contains both 128 and -128.
- It's not a valid ASCII character, but maybe it represents something in another text encoding system.
- **It is a grouping of eight electrical signals that can represent anything. It all depends on the abstractions we build.**