Notes for Computer Architecture/Computer Systems

Preface

These are notes I wrote for a Computer Architecture/Computer Systems course in 2025. It's a first pass at notes for a course I hadn't taught before and which was changing from what it used to be. As such, the notes are a little rough and could use expanding in some places, but I'm putting them on the internet in case they are helpful to people.

If you see anything wrong (including typos), please send me a note at `heinold@msmary.edu`.

Last updated: January 13, 2026.

Contents

Chapter 1

Data Representation

1.1 Binary

To really oversimplify things, computers are electronic devices consisting of many small on-off switches. We use 0 to represent a switch being off and 1 for it being on. In order to store data and communicate with a computer, we need a way to represent numbers, text, and other data in terms of these on/off switches. We use a cousin of our ordinary number system, something called the *binary number system*, to do so.

For a number like 658 in our ordinary decimal place value system, 8 is in the ones place, 5 is in the tens place, and 6 is in the hundreds place. We think of this as $6 \cdot 100 + 5 \cdot 10 + 8 \cdot 1$. We can also write it as $6 \cdot 10^2 + 5 \cdot 10^1 + 8 \cdot 10^0$. We can visualize it like below:

$$\underset{100}{6}\ \underset{10}{5}\ \underset{1}{8} \qquad \text{or} \qquad \underset{10^2}{6}\ \underset{10^1}{5}\ \underset{10^0}{8}$$

Our number system is called base-10 because it uses 10 digits (0 through 9) to represent numbers. Binary is a base-2 system since it uses only two digits (0 and 1). Instead of the ones (10^0), tens (10^1), hundreds (10^2), etc. places, binary uses ones (2^0), twos (2^1), fours (2^3), etc. For instance, the decimal number 6 can be represented in binary like below, where we have $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 4 + 2 = 6$.

$$\underset{4}{1}\ \underset{2}{1}\ \underset{1}{0} \qquad \text{or} \qquad \underset{2^2}{1}\ \underset{2^1}{1}\ \underset{2^0}{0}.$$

As another example, 53 is 110101 in binary, as shown below:

$$\underset{32}{1}\ \underset{16}{1}\ \underset{8}{0}\ \underset{4}{1}\ \underset{2}{0}\ \underset{1}{1} \qquad \text{or} \qquad \underset{2^5}{1}\ \underset{2^4}{1}\ \underset{2^3}{0}\ \underset{2^2}{1}\ \underset{2^1}{0}\ \underset{2^0}{1}.$$

Converting binary to decimal

To convert from binary to decimal, write it in the place value format with the powers of 2 under each place. Then add up all the powers of 2 corresponding to a 1 in the binary number. For instance, to convert 11010110, we write it like below. This tells us to add up $128 + 64 + 16 + 4 + 2 = 214$.

$$\underset{128}{1}\ \underset{64}{1}\ \underset{32}{0}\ \underset{16}{1}\ \underset{8}{0}\ \underset{4}{1}\ \underset{2}{1}\ \underset{1}{0}.$$

Converting decimal to binary

There are several methods for converting from decimal to binary. Here is the one I like: First, it helps to know some of the early powers of 2. They are 1, 2, 4, 8, 16, 32, 64, 128, and 256. The process starts by finding the

largest power of 2 that fits into the number. We get a 1 in that position in the place value system. Then we subtract that power of 2 and keep repeating the process until there is nothing left. All the positions that don't get 1s end up with 0s.

For example, to convert 102 into binary, the largest power of 2 that fits into it is $2^6 = 64$. So we'll get a 1 in the 2^6 place, and then we subtract $102 - 64 = 38$. The largest power of 2 that fits into 38 is $2^5 = 32$, so we get a 1 in that position. Then subtract $38 - 32 = 6$. The largest power of 2 that fits into 6 is $2^2 = 4$. Subtract $6 - 4 = 2$. Finally, $2^1 = 2$ fits into 2, and $2 - 2 = 0$. We stop once we get to 0. The final result is below.

$$\underset{64}{1}\ \underset{32}{1}\ \underset{16}{0}\ \underset{8}{0}\ \underset{4}{1}\ \underset{2}{1}\ \underset{1}{0}.$$

Larger conversions

For small numbers, and quick and convenient to convert by hand. For larger numbers, it's tedious. Many calculators and online tools can do conversions. In Python, `bin(x)` will convert the number `x` into binary. It puts `0b` before the binary representation to indicate it is in binary. To convert from binary to decimal, you can just put `0b` before the binary representation and Python will convert it to a number. You can also do `int(x, 2)` if `x` is a string containing the binary number.

Hexadecimal

Besides decimal and binary, there are number systems for every positive integer. For instance, the base 3 number system uses digits 0, 1, and 2 and the place values are 3^0, 3^1, 3^2, etc.

Most bases besides binary and decimal are rarely used, except for octal (base 8) and hexadecimal (base 16). Octal used to be more important. The main place you'll see it now is in the Unix chmod file permissions. However, hexadecimal (often abbreviated as "hex") is widely used. The main reason for using hexadecimal is that it's quick to convert back and forth to binary (because 16 is a power of 2) and unconverted binary is long and hard for humans to read. For instance, in many programming languages, integers are 32 bits in size. The number 1,234,567,890 is 01001001100101100000001011010010, while in hex it is 499602d2.

Hexadecimal is a base-16 system. Its 16 digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. A is 10, B is 11, etc., up to F being 15. Here is the hex number A4F expressed in place value notation.

$$\underset{256}{A}\ \underset{16}{4}\ \underset{1}{F} \qquad \text{or} \qquad \underset{16^2}{A}\ \underset{16^1}{4}\ \underset{16^0}{F}.$$

Converting hex to decimal

Converting back and forth between decimal and hex uses similar procedures to the conversions we used between decimal and binary.

For instance, to convert *A4F* to decimal, we write it in the place value format like below, and then multiply the values by their places, keeping in mind that $A = 10$, $B = 11$, etc.

$$\underset{256}{A}\ \underset{16}{4}\ \underset{1}{F}.$$

We get $10 \cdot 256 + 4 \cdot 16 + 15 \cdot 1 = 10 \cdot 256 + 4 \cdot 16 + 15 \cdot 1 = 2639$.

Converting decimal to hex

To convert 575 into hex, start by finding the largest power of 16 we can fit into it. This is $16^2 = 256$. We can fit 2 of them into it since $2 \cdot 256 = 512$. So there will be a 2 in the 256's place. Then subtract $575 - 512 = 63$.

Next, we can fit $16^1 = 16$ into 63. We can fit 3 of them in since $16 \cdot 3 = 48$, but $16 \cdot 4 = 64$. So there will be a 3 in the 16's place. Subtract $63 - 48 = 15$. The ones place will be a 15, which is F in hexadecimal, so the final result is 23F. See below.

$$\underset{256}{2}\ \underset{16}{3}\ \underset{1}{F}.$$

Converting between binary and hexadecimal

Converting between binary and hexadecimal is very quick, which is one of the main reasons hexadecimal is so important. The key idea is that every four bits corresponds to one hex digit. The table below will be helpful.

Dec	Hex	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

Dec	Hex	Bin
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

To convert from hex to binary, just replace each hex digit with its binary equivalent. For instance, to convert A14, we use 1010 for A, 0001 for 1, and 0100 for 4, and put them together to get 101000010100.

To covert from binary to hex, break things up into groups of 4 and find the corresponding hex for each group. For instance, 1011001101101111 becomes 1011 0011 0110 1111. From this we get B36F.

If you work with hex enough, you will eventually start to recognize the bit patterns without needing to refer back to the table. Also it's quick to work out the binary of the digits 0 to 15 if needed. For instance, 11 is 1011 (which comes from $8 + 2 + 1$). So it's not necessary to memorize the table.

You will typically see hex numbers preceded by `0x`. Python's `hex` function can be used to convert to hex. To convert from hex, you can put the `0x` in front of the number or use `int(x, 16)` if the number is in the string `x`.

Hex is used widely in computer science. You will often see it when looking at dumps of computer memory or network traces. It also shows up in character codes, IPv6 addresses, color codes in HTML/CSS, and hashes.

1.2 Character and Integer Data types

A single binary digit is called a *bit*. Bits are put into groups of 8 called *bytes*. A byte can store up to $2^8 = 256$ different values (running from 0=0b00000000 to 255=0b11111111). In general, the number of possible values you can store with n bits is 2^n.

Characters

Since computers store everything as numbers (in binary), how can they store text? The trick is that each character is associated with a numerical code. For instance, the capital letter A has a code of 65. In the early days of computing, there were many different systems of codes. In the 1960s, these were standardized into something called the ASCII system.

The original ASCII code system used 7 bits to store characters. The 8th bit was used as a parity check to detect errors in data transmission. A 7-bit system can store $2^7 = 128$ characters with codes from 0 to 127.

A few ASCII codes that are nice to know are capital letters (65 to 90), lowercase letters (97 to 122), numbers (48 to 57), and space (32). The codes from 0 to 31 were assigned to what are called "control codes." A primary

means of communicating with computers when ASCII was developed were teletype machines, which were like an electronic typewriter, and those control codes were associated with actions on those machines. Only a few of those early codes are still used, including the null character (code 0), tab (code 9), newline `\n` character (code 10), and the carriage return `\r` (code 13), which is often combined with `\n` in network packets.

As computers became more reliable, the parity bit wasn't needed, and various systems used it in different ways to extend to $2^8 = 256$ characters. But this wasn't standard, and also, way more than 256 characters are needed for all the various writing systems in the world. So in the 1990s, a new standard called *Unicode* was developed. There are actually several different Unicode standards, the most common of which is UTF-8. All Unicode standards agree with ASCII for the first 128 values.

UTF-8 uses a variable-length encoding, using 1 to 4 bytes to store characters. Common characters, like numbers, A-Z, etc. use only 1 byte, while more unusual ones use more. One thing to be careful of when looking up character codes online is sometimes they are given in decimal and sometimes in hexadecimal.

In Python, to see the Unicode character code associated with a character, use the `ord` function. For instance, `ord('A')` will return 65. To go the other way, use the `chr` function. For instance, `chr(65)` gives the letter A.

Unsigned integers

Computers store integers in a couple of different common sizes, depending on the system and programming language. These can be divided into *signed* and *unsigned* types, depending on if they allow negatives or not.

Below are the unsigned integer data types in C on Linux. Many programming languages and systems use something close to this, though there is some variation.

1. `unsigned char` — This is an 8-bit (1-byte) integer. The smallest 8-bit binary value is 00000000 (0 in decimal) and the largest is 11111111 (255 in decimal), so the possible values that can be stored run from 0 to 255. That is, since there are 8 bits, there are $2^8 = 256$ possible integers, running from 0 to $2^8 - 1$. This data type can be interpreted either as an integer or as a character, depending on the context.
2. `unsigned int` — This is a 32-bit (4-byte) integer, with values running from 0 to $2^{32} - 1$, which is a little over 4 billion.
3. `unsigned long` — This is a 64-bit (8-byte) integer, with values running from 0 to $2^{64} - 1$, which is around 18 quintillion.

In general, an n-bit unsigned integer stores values running from 0 to $2^n - 1$.

If you want larger integers than these standard types, some programming languages have special data types for arbitrarily large integers. But those tend to run a lot more slowly than the ones given above as the ones above translate directly to fast CPU instructions, whereas the arbitrarily large integers use higher-level programming to handle things, which takes longer.

Signed integers

When using signed integers, the leftmost bit is used to keep track of whether the number is negative or positive. This cuts down by roughly half on the maximum value. For instance, while an unsigned 8-bit integer data type holds values from 0 to 255, a signed 8-bit integer data type holds values from -128 to 127. In general, an n-bit signed integer will have values that range from -2^{n-1} to $2^{n-1} - 1$.

Let's look at how negative numbers are stored in binary. The most natural way to store negatives is to use the leftmost bit to store the sign (0 for positive, 1 for negative) and then use the other bits to store the numerical value. For instance, for an 8-bit integers, +5 might be stored as 00000101 and -5 as 10000101, the only difference being 0 versus 1 in the leftmost bit. However, this simple system isn't used because when adding a positive and a negative integer, you end up needing some if/then logic to determine if the final result should be

positive or negative. You can certainly build a system that does this, but there is a better way that avoids this. In fact, the method we will show below allows both addition and subtraction to be done using the same circuitry, saving valuable space on the CPU.

The method is called *two's complement*. To motivate the idea, let's first look at base-10 arithmetic. If we subtract $456-167$, we get 289. The algorithm to do this by hand is taught in grade school, and it's a little tedious. There is a trick called nine's complement that allows us to subtract by using addition. To do it on this example, for each digit of 167, we find its complement, what we have to add to it to get 9. The complement of 1 is 8, the complement of 6 is 3, and the complement of 7 is 2. This gives us the number 832. We add 1 to this to get 833. Then add $456+833$ to get 1289. Ignoring the 1, we see 289, which is the result of the subtraction. This always works.[1]

For binary numbers, the two's complement is gotten in a similar way to the nine's complement. Instead of seeing what we have to add to get to 9, we see what we have to add to get to 2. This is the same as just flipping the bits. So to get the two's complement of a binary number, we just flip the bits (0 becomes 1 and 1 becomes 0) and add 1. For instance, to find the two's complement of 11010011, first flip the bits to get 00101100 and then add 1 to get 00101101. To find the two's complement of 01101100, flip the bits to get 10010011 and add 1 to get 10010100.

To get a sense for how two's complement numbers are stored, the table below shows what things would look like for 4-bit integers.

Binary	Two's Complement
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Two's Complement
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

The leftmost bit is the sign bit, using a 1 to indicate if the number is negative. The number 0000 is 0, 0001 is 1, etc. up to 0111 is 7. The two's complement system means that the negative numbers end up going in reverse with -1 being at the end, -2 coming right before that, etc.

To check an example here, the two's complement of 1100 is 0100 (flip the bits and add 1). Converted to decimal, 0100 is 4, and we see that 1100 is how -4 is represented. If we want to go the other way, say to see how -7 is represented, first write 7 in binary as 0111. Then do the two's complement to get 1001, and we see in the table that 1001 is how -7 is stored.

Some 8-bit examples Let's look at what number the 8-bit value 10011100 stores. First, since the leftmost bit is 1, it is storing a negative. So we use the two's complement to find the value. Flip the bits to get 011000011. Convert this to decimal to get 99. Then add 1 to get 100. So the number it stores is -100.

What about the 8-bit value 00010001? The leftmost bit is 0, so this is storing a positive number. No two's complement is necessary. We just convert right to decimal to get 17.

How is the number -97 stored in binary? We'll do the process above in reverse. Start with 97, subtract 1 to get 96 and convert to binary to get 01100000. Then flip the bits to get 10011111.

Why use this system? The reason for this system is it makes it easier for computers to add two signed numbers. You can just add their binary representations, and things will work out perfectly. With other systems, it takes a little more work to keep track of the signs while adding, which slows things down.

[1] We can show why it works for three-digit number with a little algebra. If we're doing $a-b$, the nine's complement of b is the same as $999-b+1$. So when we add this to a, we get $a+(999-b+1)=1000+(a-b)$. If we ignore the initial 1 in our answer, that's like ignoring the 1000 in this expression, and we're just left with $a-b$.

For example, if we want to add $-7+3=-4$, we add 1001 and 0011. To do this by hand, use the grade-school arithmetic method, keeping in mind that $1+1=10$ in binary, so that we end up with a carry. See below. Notice that the answer, 1100 is exactly the representation of -4.

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

Overflow

An *integer overflow* happens if you try to store a value larger than the maximum value a data type can hold. This affects many programming languages, including C, C++, and Java. For instance, in C, an `unsigned char` data type can hold values from 0 to 255. If we have `c = 255` and do `c += 1`, instead of storing 256, it wraps back around to 0.

To see why, note that 255 is 11111111 in binary. To add binary numbers, we can use the same algorithm for addition that we learned in grade school, using carries. When we add 1 to 255, we do the following:

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

With the last carry, we get a ninth bit that is a 1. Since we only have 8 bits to store the number, this last bit is lost, and we end up storing 00000000. In a similar way, $255+2$ wraps around to 1, $255+3$ wraps around to 2 etc.

As another example, suppose a system has 4-bit unsigned integers. So the possible values run from 0 to 2^4-1, or 0 to 15. Suppose a variable `x` is holding the value 12. If we add 6 to it, what happens? We know $12+6=18$, but that is beyond the maximum 15, and things end up wrapping around to the start. Namely, 16 is the same as 0, 17 is the same as 1, and 18 is the same as 2. So $12+6$ wraps back around to 2. Another way to think of this is like arithmetic on a clock or with mods. For instance, when doing `12+6`, it's like $(12+6)$ mod 16, which is 2.

If we subtracted 14 from x, we would wrap around the other way. That $12-14=-2$, and -1 is the same as 15 and -2 is the same as 14, so we would get 14.

Overflows with signed numbers Looking at our table for 4-bit signed numbers, we see that we can store values from -8 to $+7$. If we have `x=7` and add 1, we would overflow. Since 7 is stored as 0111, when we add 1, we get 1000, which is used to store -8. So when you overflow a positive signed number instead of wrapping to 0 like for unsigned numbers, it instead goes to the most negative number.

Similarly, if you are at $x=-8$ and you subtract 1, you would be overflowing from the most negative to the most positive number since right before -8 in the table is $+7$.

Integer overflows are a source of some serious programming bugs. Here are a few examples.

1. A famous integer overflow occurred in the software of a rocket launched in 1996 that caused its destruction. It was a $370 million loss.
2. YouTube used a 32-bit signed integer to store the number of views for videos. The max value of a 32-bit signed integer is $2^{31}-1$, which is a little over 2 billion. In 2014, the video for Gangnam Style was going to go past that limit, so YouTube had to update things to use a 64-bit integer.
3. The 1980s video game Pacman could not be played past level 255. The level was stored with an 8-bit unsigned integer, and the game would crash if you were good enough to make it that far.
4. In 2015, people discovered a bug in Boeing 787 airplanes that would cause the electrical system to shut down (possibly mid-flight) if the plane's systems had been continuously running for 248 days. People

figured it was an integer overflow because 248 days translates to a little over 2 billion hundredths of a second, which is right at the limit of what can be stored in a 32-bit signed integer.

5. The Y2K and Year-2038 problem are examples of integer overflows. Y2K happened because systems programmed in the early days of computing stored dates as two digits in order to save memory (which was scarce). The year 1999 would be stored as 99 and the next year, 2000, would overflow and roll back over to 00, which corresponded to 1900. People were concerned this would destroy civilization as all the computer systems would crash. Nothing much ended up happening, though that may have been due to all the effort put into patching systems in the years leading up to it.

   The Year-2038 problem is similar. Many computer systems store time in a particular way – it's the number of seconds that have elapsed since January 1, 1970. For instance, as I type this, Python's `time()` function tells me the current time is 1755898107.70185, or a bit over 1.75 billion seconds. Times are often stored as 32-bit integers, and in 2038, things will hit that 32-bit-signed-integer limit of just over 2 billion and wrap back around to -2 billion, making unpatched systems think it is 1901.

6. Integer overflows are often used in conjunction with something called a buffer overflow. A buffer overflow, briefly, is where someone is able to read or write past a fixed region in memory. Buffer overflows are a key tool attackers and malware use to take over systems. The wrap-around from integer overflows is used to make the system think the buffer is larger than it really is, which then allows a buffer overflow. There is a lot of detail we are omitting here, but that is the basic idea.

1.3 Floating-point numbers

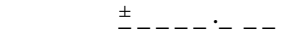
Here is a fun little exercise. Open up Python or your favorite programming language. Create variables `x = .2 + .1` and `y = .3`. Then check if `x == y`. It should come out false, which mathematically seems quite wrong. If you inspect the values of `x` and `y` out to about 17 decimal places, you might see that while `y` is 0.30000000000000000, `x` is actually 0.30000000000000004. This has to do with how real numbers are stored on computers, which is what we will look at now.

First, remember that decimal expansions of real numbers can go on forever. For instance, $1/3 = .3333\cdot$ with the 3s repeating forever. And $\pi = 3.14159\cdots$ goes on forever with the digits following no predictable pattern. In both cases, if we want to store these decimal expansions, we have to cut things off somewhere, and doing so means we lose some accuracy. This is part of what causes the problem in the paragraph above. The whole story is developed below.

Fixed-point numbers

There are two systems commonly used to store fractional numbers in computers: *fixed point* and *floating point*. Fixed point is the simpler approach, but it's only used in a few particular contexts.

To demonstrate the fixed-point system, here is a simple way to store 8-bit numbers.



The leftmost bit is for the sign, the next four are for the integer part, and the remaining three are for the fractional part. With four bits for the integer part, we can store integers from -15 to 15. The three bits for the fractional part allow decimals that are 0, .125, .25, .375, .5, .625, .750, and .875. This is a pretty limited system. As we add more bits, we can store larger numbers and get more decimal digits of accuracy.

However, if we want to store numbers needed for scientific calculations, then we'll end up needing a ton of bits. For instance, the mass of the earth in kilograms is 597200000000000000000000, while the mass of an electron in kilograms is .000000000000000000000000000000091. For a fixed point system to handle both of these, we would need over 60 decimal digits, which translates to over 200 bits, or 25 bytes to store a single number. And many calculations need even larger or smaller numbers than these. Scientific calculations need to use really large and really small numbers, so because of the size needed, fixed-point won't work very well for these.

The one big application for fixed point is storing money amounts, where there are usually exactly two digits after the decimal point, which is a natural fit for a fixed-point system. Also, the floating-point system we will talk about shortly has accuracy problems like the $.1 + .2 \neq .3$ problem discussed earlier, but a good fixed-point system won't have that problem. Thus, it is good for working with money amounts, where we need exactness.

Floating-point numbers

The "floating" in floating point means that the decimal point is allowed to float or move around. It's not always in the same place. The idea behind floating point is based on scientific notation. For instance, the mass of the earth, instead of writing it out as a huge number, can be written as 5.972×10^{24}. The number is stored in two parts. The 5.972 part is called the *significand* or *mantissa*, and the 24 part is called the *exponent*. Storing things this way allows us to represent a wide range of numbers with only 32 or 64 bits (4 or 8 bytes).

To understand how this works, first go back to our ordinary number system. In a number like 658.32, the 8 is in the one's place, the 5 is in the ten's place, and the 6 is in the hundred's. To the right of the decimal, the 3 is in tenth's place, and the 2 is in the hundredth's. See below.

$$\underset{100}{6}\ \underset{10}{5}\ \underset{1}{8}.\underset{1/10}{3}\ \underset{1/100}{2} \qquad \text{or} \qquad \underset{10^2}{6}\ \underset{10^1}{5}\ \underset{10^0}{8}.\underset{10^{-1}}{3}\ \underset{10^{-2}}{2}$$

Something similar works in binary. For instance, the binary number 101.1101 is $4 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 5.8125$ in decimal. See below.

$$\underset{4}{1}\ \underset{2}{0}\ \underset{1}{1}.\underset{1/2}{1}\ \underset{1/4}{1}\ \underset{1/8}{0}\ \underset{1/16}{1} \qquad \text{or} \qquad \underset{2^2}{1}\ \underset{2^1}{0}\ \underset{2^0}{1}.\underset{2^{-1}}{1}\ \underset{2^{-2}}{1}\ \underset{2^{-3}}{0}\ \underset{2^{-4}}{1}$$

Almost all hardware uses the IEEE-754 floating-point standard. IEEE is an engineering organization, and they put out a lot of standards for computer technology. It's important that for something as foundational as arithmetic that everyone do things the same way so that you don't get different results depending on where you run something.

IEEE-754 defines a 32-bit floating-point data type that is typically called a `float` and a 64-bit floating-point data type that is typically called a `double`. These are sometimes called single-precision and double-precision. Here are the details on each.

1. The 32-bit, single-precision types can hold values as small as around 10^{-38} to as large in magnitude as $\pm 10^{38}$. You get about 7 to 8 digits of precision.
2. The 64-bit, double-precision types can hold values as small as around 10^{-308} to as large in magnitude as $\pm 10^{308}$. You get around 15 to 17 digits of precision.

Let's look in detail at the 32-bit single-precision float. The leftmost bit is used to store the sign. The next 8 bits store the exponent. The remaining 23 bits store the significand.

With 8 bits to store the exponent, there are $2^8 = 256$ possible exponents we can store. Some are used for negative exponents (small numbers close to 0) and some are used for positive exponents. The system centers things around 127, so that the value 127 stored in the exponent (01111111 in binary) corresponds to an actual exponent of 0. 128 is an exponent of 1, 129 is an exponent of 2, etc., and 126 is an exponent of -1, 125 is an exponent of -2, etc. The values 0 and 255 are special cases that will be covered below.

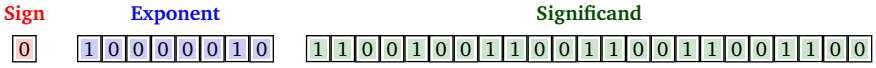
Example To understand how the IEEE-754 system works, let's look at the exact way the fractional number 14.3 is stored. First, 14 is 1110 in binary. Next, you can either work out by hand or using an online tool (Wolfram Alpha is good for this) that 0.3 in binary is 0.01001100110011..., with the 0011 repeating forever. So 14.3 in binary looks like below:

1110.01001100110011001100110011...

We then write this in binary scientific notation. Scientific notation always has just a single digit before the decimal place, so we have to move the binary decimal place over 3 places to the left in our number. This gives us our exponent, which is 3. So our number is

$$1.1100100110011001100110011001100110011\cdots \times 2^3$$

Since this is binary scientific notation, we have 2 to a power instead of 10 to a power. To store this in 32-bit floating point, we first use a sign bit of 0 as the leftmost bit, since this is a positive number. For the exponent, remember that everything is centered around 127, so we do $127+3=130$. So the exponent is 130, which is 10000010 in binary. In general, whatever the exponent is from the scientific notation form, just add 127 to it to get what is actually stored. Finally, take the first 23 digits *after* the moved decimal point to get the significand. The initial 1 before the decimal place is implied. That is, we assume it is always there. This is a neat trick that allows us to store an extra bit of precision for free. The end result is below.



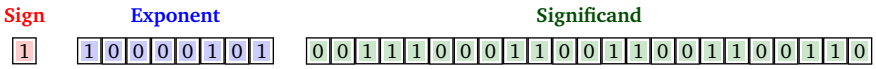
We can see here where inaccuracy in floating point numbers creeps in. The binary expansion continues on forever, but we have to cut it off after 23 bits.

Example As another example, let's look at storing -78.2.

First, we could look up or work out that .2 in binary is .00110011... with the 0011 repeating forever. We can write 78 in binary as 1001110. Our -78.2 in binary is below. Below that, we move the decimal place over 6 spots so the number starts with 1 point something. This is binary scientific notation.

$$-1001110.00110011\ldots$$
$$-1.00111000110011\cdots \times 2^6$$

The mathematical exponent is 6. But we have to add the bias of 127 to get the exponent IEEE-754 uses. So we get $127+6=133$, or 10000101 in binary. This is the exponent IEEE-754 stores. Since the number is negative, the sign bit is 1. The significand is the first 23 bits after the decimal point in the value above. So the final representation is this:



Some more details

We mentioned earlier that exponent values of 0 and 255 are special cases. Here are the details. Below e stands for the exponent value and f stands for the significand.

- If $e=0$ and $f=0$, then the value stored is 0.[1]
- If $e=0$ and f is not 0, then the value stores an *denormalized number*. Without going into too much detail, the 1 that comes before the decimal place in the scientific notation format is assumed to be there for most floating point numbers. Numbers where the 1 is assumed to be there are called *normalized*. Denormalized numbers are a way to store values really close to 0.
- If $e=255$ and $f=0$, then the number is either `+inf` or `-inf`. These are ways to store infinity.
- If $e=255$ and f is not 0, then the number stores `NaN`. This stands for "not a number". It is a special value used to indicate some type of problem. In some programming languages, you'll get if you try to do the undefined operation 0/0. Another time you'll see it is if you try to convert a string to a floating point number (like with JavaScript's `parseFloat` function), and the string doesn't make sense as a number, like if you do `parseFloat("xyz")`.

[1] Because of the sign bit, the system actually stores both +0 and −0, but both are treated as 0.

How 64-bit floating-point numbers are stored For 64-bit double-precision numbers, we have 1 bit for the sign, 11 for the exponent, and the remaining 52 for the significand.

When to use each type Personally I usually use 64-bit doubles instead of 32-bit floats for most things. Doubles are more accurate, and the extra memory use is usually not a problem. However, floats are useful in a couple of places. Their main advantages are that they take up less space than doubles, and operations on them are faster since they are half the size. Applications like video games and generative AI can really benefit from both the space-saving and the speed-up of floats.

What if you need more precision If for some reason you need more than the 15-17 digits you get from doubles, most programming languages have libraries that allow you to get more precision. Python's `Decimal` class is one example. However, these libraries will be much slower than using the native `float` and `double` data types of most programming languages. Those data types use fast CPU instructions for doing math with floating point numbers. Programming language libraries for higher-precision have to do their operations in software, which is much slower.

Floating-point problems

When working with floating-point numbers, you have to be careful. The key issue is that most numbers can't be stored exactly. The error due to this is called *roundoff error*.

For example, .3 in binary is 0.01001100110011..., and when we cut this off after a certain number of decimal places, we introduce a small error. To see this, if we do `print('{:.60f}'.format(0.3))` in Python, we get `0.299999999999999988897769753748434595763683319091796875000000`. This value is the closest double-precision floating-point number to 0.3. Note that the first 16 digits round right up to 0.3, but after that things start going weird. Again, that weirdness is due to cutting off the infinite decimal representation. Here are some consequences of roundoff error.

1. Sometimes when you print a floating-point number, it will print out something like above, like `0.2999999999999999`, depending on the programming language. That can look bad, so it's sometimes necessary to use the programming language's rounding or formatting functions to avoid that.

2. Comparing floating point values can be a problem. For instance, we saw earlier that `if .2+.1 == .3` will actually return false. This is due to the roundoff errors in .2 + .1 combining in a different way than the roundoff error in .3. To deal with this, you can use a trick like this:

   ```
   x = .2 + .1
   y = .3
   if abs(x - y) < .000000001:
       print('Equal')
   ```

   The code above is Python, but the same idea works in most other languages. The idea is that the two variables are probably equal if the difference between them is really small. We chose .000000001 here because it is hopefully a lot smaller than whatever is stored in `x` and `y`, and it is also bigger than the 15-17-digit precision limit before we run into roundoff error. Depending on the application, you might need to change that value around a little, like if `x` and `y` themselves can be really small.

3. Precision matters. In Python, if you do 200 + .008, you'll get 200.008. But if you do 200 + .000000000000008, the result will be 200, which is not mathematically correct. The reason is that Python uses 64-bit IEEE-754 doubles, which only give 15-17 digits of precision. In the first example 200.008 only uses 6 digits of precision, so we are okay. but in the second example, 200.000000000000008 needs 18 digits of precision to store, which is more precision than we are allowed. So the last few digits are essentially dropped, and we end up with 200.

4. Most fractional numbers have roundoff error, but not all. Any number that can be written as a finite and small enough number of powers of 2 won't have any roundoff error. For instance, .5 in binary is just 0.1.

There is no infinite expansion, it's just all 0s. That is because to the right of the decimal point are the one half's place, the one quarter's place, etc., and .5 can be represented exactly with a 1 in the one half's place and 0s everywhere.

Similarly, .25 is 0.01 in binary, .75 is 0.11 in binary, and .625 is 0.101 in binary. All of their decimal expansions stop, so they can be represented exactly. But something like .8, which is 0.1100... repeating forever, will have to get cut off after awhile and thus won't have an exact representation.

5. Roundoff error accumulates. For instance, the roundoff error in 0.1 happens around the 17th decimal place, but the code below produces a value of `100000.00000133288` for `x`, which is actually off in just the 6th decimal place (or 12 places from the initial 1). Adding 0.1 to itself a million times causes the roundoff error to propagate, or move forward in the calculation.

```
x = 0
for i in range(1000000):
    x = x + .1
print(x)
```

6. Be careful about subtracting nearly equal numbers. For instance, try computing the following:

```
x = (1.000000000000001 - 1) * 100000000000000
```

The exact answer should be .1, but the program gives 0.11102230246251565, only correct to one decimal place. We can see where this comes from by doing `print('{:.25f}'.format(1.000000000000001))`. This gives `1.0000000000000011102230246`. The part towards the end is "floating-point junk" that comes from roundoff error. Remember that we only get around 15 to 17 digits of accuracy. Anything past that is an artifact of the floating-point system cutting things off. When we do $1.000000000000001 - 1$ we are dropping the 1 off the end of this value and when we multiply by that huge number, we are essentially moving the decimal point over so that the floating-point junk comes right into the key part of the number. So just be careful about subtracting nearly equal numbers.

Chapter 2

Digital Logic

If you wanted to build a computer from scratch, from first principles, what could you do? A typical laptop has a keyboard, monitor, motherboard, RAM, hard drive, various ports, a network card, power supply, fans, and much more. While all of those are important, they are not the heart of the computer. The heart is the CPU. Without a CPU, you don't have a computer, so the CPU is what we will focus on. It seems silly to say, but a computer *computes*. It does things like addition and multiplication, it has if/then logic, and it has a capacity to repeat instructions. We will focus on how it does some of these basic operations.

Computers are based on *boolean logic*. There are certain fundamental operations – AND, OR, NOT, XOR, NAND, NOR, and a few others – that we can use to build everything else. Computers work in binary, and we think of 0 as being false and 1 as being true. In the physical world, 0 corresponds to low voltage and 1 corresponds to high voltage, sort of like a switch that is either off or on.

2.1 Logical Operations and Gates

Basic logical operations are described by *truth tables*. Below is the truth table for some common logical operations.

AND The AND operation is true only when both operands are true. Otherwise, it is false. This is just like real-world logic – when you say you will do something if *a and b* are true, you mean that both have to be true. If only one is true, or none at all, then you won't do it. Here is the truth table.

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

We see that the only 1 (true) is in the last row, corresponding to both a and b being 1 (true).

OR The OR operation is true if one, the other, or both of its operands are true. This, again, is like real-world logic where if you might say you'll do something if either of a or b is true. The OR operation includes the possibility that both are true. Here is the truth table.

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

XOR The XOR operation is the *exclusive or*. It is for when you want the result to be true if one or the other, *but not both*, of the operands are true. In the real world, this might be where if you have two friends that don't get along and you want to hang out with one or the other, but never with both at the same time. Here is the truth table:

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

NOT The NOT operation is pretty simple – it just negates its operand. That is, NOT true is false and NOT false is true. Here is the truth table:

a	NOT a
0	1
1	0

NAND The NAND operation is short for NOT AND. It is what we get by negating the AND operation. Here is its truth table:

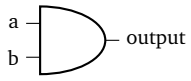
a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

There is also a NOR (not or) operation that you sometimes see, but we won't be working with it much.

Combining operations We can combine logical operations. For instance, suppose we want the truth table for a XOR (b OR c). There are 8 possibilities for all the values of a, b, and c. A good way to approach this is to break it into parts. First work out the b OR c part, and use the XOR operation on that with a. See below:

a	b	c	a	b OR c	a XOR (b OR c)
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	0	1	1	0
0	0	1	0	1	1
0	1	1	0	1	1
1	0	1	1	1	0
1	1	1	1	1	0

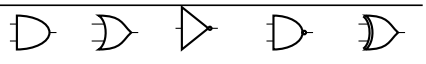
Logic Gates Logical operations can be chained together like this to do everything that a computer does. Boolean logic operations are built into physical components called *logic gates*. Below is an example of an AND gate.



The two inputs, a and b, can be either 0 or 1. Some circuitry inside, which we will discuss shortly, does the AND operation, and the output is either 0 or 1. Or more properly, since this is an electronic device, there are either low or high voltages at the input wires going into the AND gate, and the gate's circuitry controls whether the voltage at the output is low or high.

All of the operations of a CPU are built by combining logic gates together in clever ways. Also, some electronics

that don't have CPUs just have logic gates wired together in interesting ways to do things like control a microwave or change a TV station. Below are the symbols for various types of gates. We won't be using those here, but you might see them in other places, so it's good to know about them.

Gate	AND	OR	NOT	NAND	XOR
Symbol					

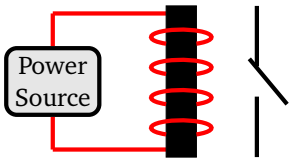
Of all the gates, the most important to modern computing is the NAND gate. It is possible to build all the other gates purely out of NAND gates, and that is how modern computers are built.

2.2 Nandgame

We will soon look at how to build some basic operations out of NAND gates. First, we will look at how to build a NAND gate out of fundamental physical components. The discussion here is inspired by Charles Petzold's excellent book *Code* and the online game Nandgame (https://www.nandgame.com).

Relays

One way to build a NAND gate is to use a physical device called a relay. One is pictured below. To build one, start with an iron bar. Wrap a copper wire around it several hundred times and connect the wire to a power source, such as a battery. Once it is connected and current is flowing in the wire, the iron bar becomes a magnet (an electromagnet). Next to the iron bar is a switch. This is made of something magnetic so that when the iron bar becomes magnetized, it pulls the switch shut. The switch also is spring-loaded, so once power is cut to the electromagnet, it swings back open.

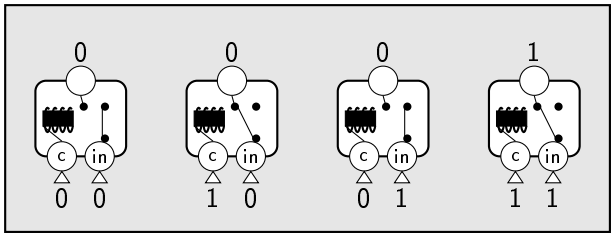


The purpose of a relay is that it acts a little like a light switch, except that instead of being controlled by a person flipping the switch, it is controlled by electric current, basically whether or not the coiled wire is connected to the battery or not.

A relay can be built in two different configurations: default on, or default off. The one shown above is default off, which is to say that the switch is open initially, and when current is flowing in the coiled wire, it closes the switch. The default-on relay is built so that the switch is initially closed, and when current is flowing, it pulls the switch toward the iron bar, opening it. We use both types of relay to build a nand gate.

From here and for the next few pages, we will be building various gates and other components of a basic computer. We are following the levels of Nandgame, and what follows are spoilers for it. It would be best to try it yourself before reading any further.

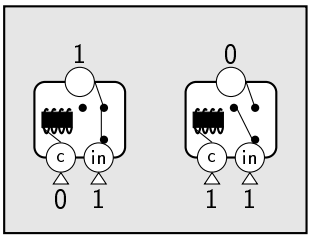
Below are simulated screenshots from the online Nandgame, showing the four possible states of the default-off relay. In these, 0 stands for there being no current present at that point in the device and 1 stands for there being current.



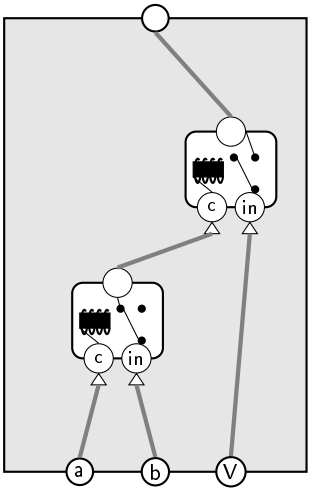
In the leftmost case, both inputs have 0s. There is no current flowing to the coil, so the switch stay open. Thus there ends up no current at the output, which we mark with a 0. In the second from the left here power is going to the coiled wire. That pulls the switch closed. However, since the other input is 0, there is no power in the line connected to the switch, so the overall output is 0. The next image over has power in the switch line, but the switch is open since there is no power going to the coil, so the overall output is still 0. Finally, on the right we have power to the coil, so the switch is closed, and because there is power in the switch line, we get power at the output.

Putting all this together, the output is 1 when both inputs are 1 and 0 otherwise. That is, we have just built an AND gate. This is nice, but we are looking for a NAND gate.

To get a NAND gate we also need a default-on relay. Below is what a default-on relay acts like if we connect the right input to a power source that is always 1. We see it acts like a negation operation: if the left input is 0, then this relay flips it to 1, and if the left input is 1, it gets flipped to 0. This acts like a NOT gate or inverter.



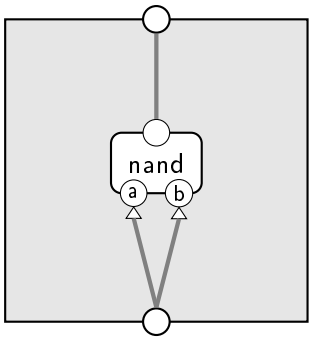
Since NAND is NOT AND, we can connect the first relay to the second one like below to build a NAND gate.



Once we've built the NAND gate, we can imagine packaging it into a little black box labelled NAND and not worry about the relays inside. We can then wire various NAND gates together to do interesting things.

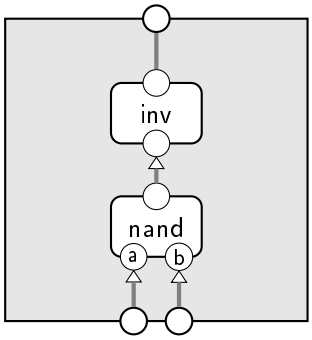
Note: Modern computers are not built out of relays. They are built using *transistors* instead. Some of the early computers of the 1940s were in fact built from relays, and some hobbyists still build computers from relays. It's nice to know that we can build a computer from basic parts (iron bar, wire, battery) that we could find lying around. However, relays are kind of big and slow. Thousands of them are needed to build all the components of a computer. Transistors accomplish similar things to what we would get from a relay, but they are much smaller, like a few dozen nanometers in size, less than 100 atoms. The massive increase in computing power and storage since the 1960s has been made possible by people being able to make transistors smaller and smaller, allowing more to fit onto a chip. Transistors also use much less power and are more reliable than relays.

NOT/Invert This is about building a NOT gate, or inverter, from a NAND gate. We saw that we can build one from a relay directly, but let's look at how to use a NAND gate to do it. It's pretty simple: if we are inverting a, just connect a to both inputs of the NAND gate. Since NAND is only 0 when both inputs are equal to 1, this will be 1 when the input is 0 and 0 when the input is 1.



Now that we've built an inverter, just like with the NAND gate, we can put our inverter circuitry into a little box called "inv", and whenever we want to flip a bit, we can plug it into our inverter.

AND Again, we can use a relay to do this directly, but let's look at using NAND gates. This is not too complicated. Since NAND is "NOT AND", if we do NOT NOT AND, the double-negative essentially goes away and we're just left with AND. That is, we connect an inverter to a NAND gate, like below.

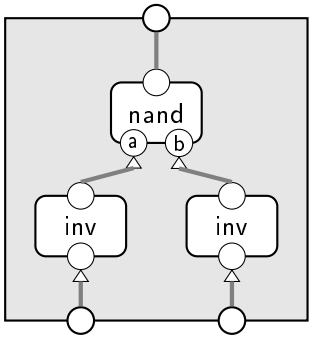


OR Building an OR gate is a little trickier than the previous two. One way is to notice that the truth table for OR is like NAND, but in reverse order. See below.

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

This suggests an approach: invert the a and b inputs and feed them into a NAND gate. The idea is that NAND is only 0 if $a = b = 1$, while OR is only 0 if $a = b = 0$, so if we flip things before feeding them into the NAND gate, then it will behave like an OR gate.



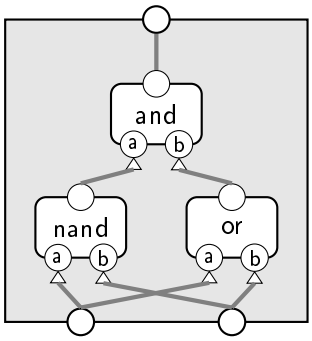
XOR There are a few different approaches to this. One of them uses 4 NAND gates. Another approach, shown below, is to notice that the two places the truth tables for NAND and OR both agree is when a and b are different.

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

So we can build an XOR gate by using an AND gate to see where the outputs of OR and NAND gates agree, like below.

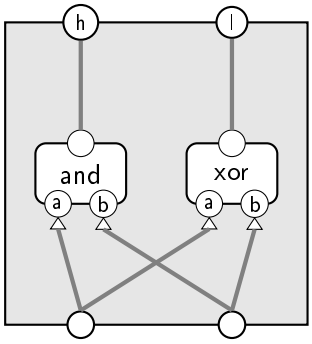


With this, we have now built enough fundamental logic gates to handle basic logic. Now, we look at arithmetic.

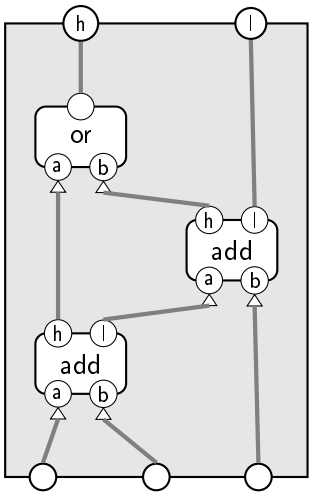
Half Adder A half-adder is a circuit that adds two bits. The possible values are below.

```
  0     0     1     1
+ 0   + 1   + 0   + 1
 00    01    01    11
```

The output is shown as two bits. The low bit follows the pattern 0 1 1 0. This looks just like the truth table for XOR, so that tells us what to use for the low bit. The high bit follows the pattern 0 0 0 1, which looks like the truth table for AND. Thus, we use XOR to get the low bit and AND to get the high bit, like below.



Full Adder A full adder takes inputs a and b along with a carry c that we assume comes from some other part of a computation. The trick to building the full adder is that we are doing $a + b + c$, and we can think of that as $(a + b) + c$. That is, we add a and b first, and add the result of c to that. The solution is below.

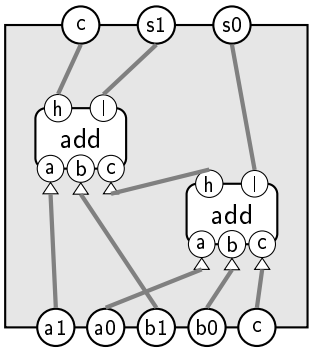


The half-adder on the left adds a and b. The low bit of the result gets added with c in the right adder. The low bit of that is the low bit of the addition $a + b + c$. For the high bit, we might get a carry out from either adder. If one or the other or both has a carry, then there will be a carry in the final result, so we use an OR gate to handle that.

Multibit adder This is about adding two-bit numbers. We have something like below:

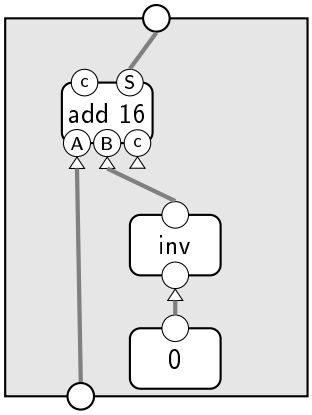
	carry out	bit 1	bit 0
carry in			1
a		1	0
b		1	1
result	1	1	0

We are adding a with a0 and a1 representing its two bits, and b with bits b0 and b1, along with a carry c that maybe came from some other step in a computation. The solution is below.



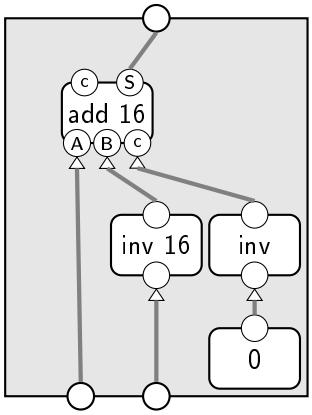
The idea is that we add the right bits and carry first with an adder. This gives us the low bit of the answer. It also might give us a carry. We then add that carry along with the high bits in another adder. Basically, we just add bit-by-bit, taking into account the carries.

Increment *Incrementing* means to add 1 to a value. The solution is below.

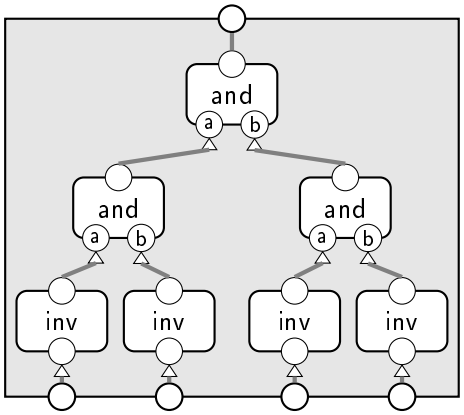


The idea is simply that we need to add the value a with 1. To get 1, we can use the 0 component (that is always 0) followed by an invert component to make it 1.

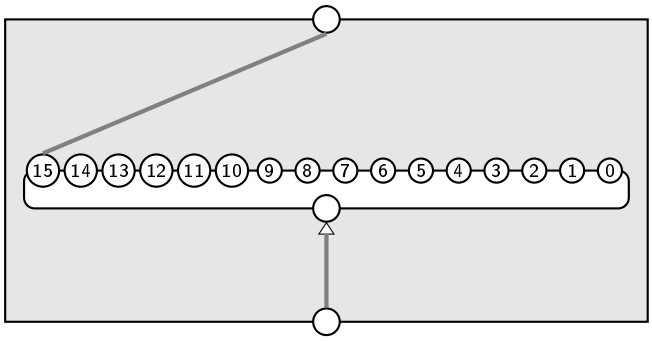
Subtract Remember that subtraction works via the two's complement. In ordinary math, $a - b$ is $a + -b$, and in binary $-b$ is the two's complement. We get the two's complement by flipping the bits and adding 1. We can get that by using the `inv16` component, followed by the `inc16` component. The solution is below.



Equal to 0 We have a 4-bit number and we want to check if it's 0. That is, all four bits – a0, a1, a2, and a3 – need to be 0. We want a0 == 0 AND a1 == 0 AND a2 == 0 AND a3 == 0. The AND gate only takes two inputs, so we need to chain a few AND gates together to do this. Also, since we want to check if things are 0, we have to invert first. The solution is below.



Less than 0 We are checking if a number is less than 0. The solution to this is quick. Recall that the leftmost bit is the sign bit. So we just connect that to the output. See below.



We'll stop here with the Nandgame. It continues with many more levels, and it's worth your time to work through it.

2.3 Bitwise Operations

While we are on the subject of logical operations, it's worth looking at bitwise operations. These are where we apply the same logical operation to all the bits of a number. For instance, below is an example.

	1	1	0	0	1	0	1	0
XOR	0	0	1	0	0	0	1	1
	1	1	1	0	1	0	0	1

To do this, we just XOR each individual bit of the first number with each individual bit of the second number, using the rules from the XOR truth table for each one.

Bitwise operations are used a lot in low-level programming, especially when working with hardware. They are often used to check or change the values of particular bits of a number. We will look at how this works below and later look at specific applications.

Bitwise operations are built into many programming languages. Python, Java, and C all use the following symbols for them:

AND	OR	XOR	NOT	Left shift	Right shift
`&`	`|`	`^`	`~`	`<<`	`>>`

The left shift operation shifts all the bits left by a specified amount. For instance, `x << 2` will shift all the bits of `x` left two positions. The two leftmost bits will essentially fall off the end of the number and be lost. At the right two new zeros will be inserted. Note in particular, it is a shift and not a rotation. Right shifts work similarly, except that bits are shifted to the right. Bits at the right will fall off the end and be lost, and zeros will come in on the left.

As an example, if `x` is the 8-bit number 00001100, here are the results of some shifts:

`x << 1` = 00011000
`x << 2` = 00110000
`x << 3` = 01100000
`x << 4` = 11000000
`x << 5` = 10000000
`x << 6` = 00000000

Until bits start falling off the end, a left shift behaves like multiplying a number by 2. For instance, the binary value of `x` used above is 12, and the results of the shifts above are 24, 48, 96, 192, 128, and 0, assuming that `x` is an unsigned character. A right shift behaves like dividing by 2 and rounding down.

Checking and setting specific bits

Setting bits to 0 To set a bit to 0, we AND with a binary number that is 1 everywhere, except with a 0 in the position to be set to 0. For instance, To set the third bit from the left to 0, we can AND with `0b11011111`. The ones will leave everything alone, and the 0 in the desired bit will force that bit to become 0. Here is an example:

	1	1	0	1	0	0	1	0
&	1	1	0	1	1	1	1	1
	1	1	0	1	0	0	1	0

	1	1	1	1	0	0	1	0
&	1	1	0	1	1	1	1	1
	1	1	0	1	0	0	1	0

Sometimes you'll see people write this as `x = x & 0xDF`. The hex version is a little shorter to write. You'll also sometimes see it written as `x & ~(1 << 5)`, where `1 << 5` gives a binary value with a 1 in the desired bit and 0s elsewhere, and then the `~` operator flips all the bits so it becomes the same as `0b11011111`.

Setting bits to 1 To set a bit to 1, we OR with a binary number that is 0 everywhere, except with a 1 in the position to be set to 1. For instance, to set the third bit from the left to 1, we would use the `|` operation, specifically `x = x | 0b00100000`. The zeroes in all the other positions will have no effect on the other bits, and the 1 in the desired position will have the effect of turning that bit to 1 if it is 0 and leaving it set at 1 if it is already 1. Here is an example:

	1	1	0	1	0	0	1	0
|	0	0	1	0	0	0	0	0
	1	1	1	1	0	0	1	0

	1	1	1	1	0	0	1	0
|	0	0	1	0	0	0	0	0
	1	1	1	1	0	0	1	0

Flipping a bit Flipping a bit is where if it is 1, then it gets set to 0 and if it is 0, then it gets set to 1. The XOR operation is used for this. Specifically, to flip a bit, XOR with a value that has 0 everywhere, except a 1 in the position we want flipped. For instance, to flip the third bit from the left, we do `x ^ 0b00100000`. The zeroes have no effect on the number, and XOR-ing with 1 has the effect of flipping things. Here is an example:

	1	1	0	1	0	0	1	0
^	0	0	1	0	0	0	0	0
	1	1	1	1	0	0	1	0

	1	1	1	1	0	0	1	0
^	0	0	1	0	0	0	0	0
	1	1	0	1	0	0	1	0

Checking the value of a bit To test a single bit, the `&` operator can be used to isolate it. Specifically, we AND with a binary number that is 0 everywhere, except with a 1 in the bit we are interested in. We then check to see if the result is 0 or not, which will tell us if the desired bit is 0 or 1. For instance, if we want to check if the third bit from the left in an 8-bit variable `x` is set to 1, we would use the following if statement:

```
if ((x & 0b00100000) != 0)
```

The idea is if we AND with a binary number that is 0 everywhere except with a 1 in the bit we are interested in, then the zeros will have the effect of wiping out everything else, and the 1 will isolate the desired bit. Note the extra set of parenthesis around the bitwise operation. In some languages (like C, C++, Java, and Javascript, though not Python), they are necessary due to order of operations. The `&` operation has a lower precedence than the `!=` operation.

Here is an example showing the operation in action:

	1	1	0	1	0	0	1	0
&	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0

	1	1	1	1	0	0	1	0
&	0	0	1	0	0	0	0	0
	0	0	1	0	0	0	0	0

Notice how everything is zeroed out except for possibly the desired bit.

Extracting multiple bits Sometimes we want to extract multiple bits at a time. Maybe we have an 8-bit variable `x` and we want to extract the lower 3 bits (bits #0 to #2 from the right). We can use the `&` operation

like this: `x & 0b00000111`. Alternately, we could AND with the hexadecimal value `0x1F`. The value we `&` with to extract the desired bits is sometimes called a *bitmask* or just a *mask*. Here is an example:

```
    1 1 0 1 0 0 1 0
&   0 0 0 0 0 1 1 1
-----------------
    0 0 0 0 0 0 1 0
```

If we want the upper 3 bits, we could try a mask by `0b11100000`, but thinking of those 3 bits as representing a number from 0 to 7, we wouldn't get the right result. Instead, use a shift right like this: `x >> 5`.

Applications

Here are a few places bitwise operations are used.

- Bitwise operations are used a lot in low-level and embedded-systems programming. There it is important to be able to access individual bits of a number. In embedded systems, memory is often scarce, and in networking, every wasted bit adds overhead that slows down the connection. If we have some values acting as flags, we could make separate variables for each, but a more efficient approach is to use individual bits of a variable to act as those flags. If we have an 8-bit number, each of those 8 bits can act as a separate flag. If we used 8 different variables instead, the total space usage would be 64 bits, which is much less efficient than the 8 bits if each bit is its own flag.
- The bitwise XOR operation is widely used in cryptography, in encryption algorithms and hashing algorithms. It is also useful in checksum operations for catching errors in data transmission.
- Bitwise operations are used in subnet masking of IP addresses. A subnet mask of 255.255.255.0 corresponds to a bit mask of 11111111111111111111111100000000 (24 ones and 8 zeroes). Networking software uses a bitwise AND of this with the IP address to isolate the two parts of the address, called the network and host portion.
- Bitwise operations are used to speed up certain computations. For instance, doing `x >> 1` to divide by 2 is much faster on most CPUs than `x / 2`, since the division operation is a particularly slow one.
- Colors of pixels in images are typically stored in an RGBA format. These are 32-bit numbers, where the highest 8 bits store the red component of the pixel, the next 8 store the green component, the next 8 store the blue component, and the rightmost 8 store the alpha or transparency information. If want to read or change just one component, bitwise operations are useful.

  For example, assume below that the symbols `r`, `g`, `b`, and `a` stand in place of the actual values for the red, green, blue, and alpha components. If we AND with the bit sequence below, that will isolate the green component.

```
    r r r r r r r r g g g g g g g g b b b b b b b b a a a a a a a a
&   0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-----------------------------------------------------------------
    0 0 0 0 0 0 0 0 g g g g g g g g 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

  However, the green component is currently shifted up 16 spots in the number. To get the actual 8-bit value of it (which should be a number between 0 and 255), we need to shift it right by 16. So someone would typically write this operation as `(x & 0xFF0000) >> 16`, where we've used hexadecimal instead of binary (and dropped off the leading zeros) since it's much shorter.

  An alternate approach would be to do the shifting first. This would give `(x >> 16) & 0xFF`. The AND operation at the end is to clear out the values from the blue component, which we don't want.

  As another example, suppose we want to set the green component to some new value stored in a variable `y`, while leaving the other components unchanged. We could do that as `(x & 0xFF00FFFF) | (g << 16)`. That is, we first zero out just the green component of the original number. Then we use the OR operation to add in the green component, and we make sure to shift it by 16 bits so it's in the right place.

- In scenarios where you really need to save storage space, you can pack several values into a single variable and then use bitwise operations to extract the individual parts. Suppose, for instance, that we are storing information from some weather sensors that read temperature, humidity, UV index, and rain amount. Assume all are integers with the temperature from -50 to 50 Celsius, humidity from 0 to 100, UV from 0 to 14, and rain amount from 0 to 10,000. If we use 8-bit integers (the smallest data type available in most programming languages) for the first three and a 16-bit integer for the rain amount, then we will need 5 total bytes to store the information. If the sensor reads once a second, we will have to store around 30 million times over the course of a year, which comes out to about 150 MB of data.

  However, the temperature and humidity ranges cover around 100 values, so we only need 7 bits to store them. This is because $2^6 = 64$ and $2^7 = 128$, so 6 bits could store up to 64 total values and 7 bits could store up to 128 total values. Likewise, we only need 4 bits for the 15 possible UV index (since $2^4 = 16$), and 14 bits for the rain amount (since $2^{14} = 16384$). Adding these bit amounts, we get $7 + 7 + 4 + 14 = 32$, so these can fit exactly into a single 32-bit (4-byte) integer. Using this bit-packing approach thus only needs 4 bytes instead of 5, saving us 30 MB of space of the course of a year. This isn't that much, but if we were storing a bunch more sensor values, the savings could be enough to save us from having to buy an additional hard drive.

  Bitwise operations can be used, like in the color example, to pack the values and then extract them when needed. To pack them, we would do the following:

  ```
  packed = ((temp + 50) & 0x7F) | ((humidity & 0x7F) << 7)
         | ((uv & 0x0F) << 14) | ((rain & 0x3FFF) << 18);
  ```

  To unpack them, we use the reverse of these. For instance, the humidity would be gotten via `(packed >> 7) & 0x7F`.

Chapter 3

The CPU

3.1 Metric prefixes

Metric prefixes are used all over computing, so it is helpful to be familiar with some common ones. For large values, especially for numbers of bytes and processor speeds, we use *kilo*, *mega*, *giga*, and *tera*. Here are the conversions:

1	base unit
1,000	kilo (K)
1,000,000	mega (M)
1,000,000,000	giga (G)
1,000,000,000,000	tera (T)

That is, *kilo* is for thousands, *mega* is for millions, *giga* is for billions, and *tera* is for trillions. These are all very important to know. The next two prefixes after these, *peta* and *exa*, are starting to become more useful for measuring network traffic and storage in data centers.

Technical note: We are defining a kilobyte (kB) as 1000 bytes, but sometimes you'll see people define a kilobyte as 2^{10} or 1024 bytes. To distinguish the two, people sometimes refer to 2^{10} bytes as a kibibyte (kiB). There are similar issues with higher prefixes. For instance, $2^{20} = 1048576$ sometimes means a megabyte, and some people use the term mebibyte for 2^{20} bytes.

For small numbers, we have the prefixes *milli*, *micro*, and *nano*.

1	base unit
0. 001	milli (m)
0. 000 001	micro (μ)
0. 000 000 001	nano (n)

In computing, these are most often used for times. A millisecond (ms) is 1/1000 of a second, a microsecond (μs) is 1 millionth of a second, and a nanosecond (ns) is 1 billionth of a second. There are 1000 milliseconds in a second, 1 million microseconds in a second, and 1 billion nanoseconds in a second. Some of these numbers are unfathomably small. For instance, human reaction time is around 1/10 of a second at best. A nanosecond is 100 million times smaller than that. A CPU with a 3 GHz clock speed does 3 billion cycles in a second, so each cycle is around 0.33 nanoseconds.

Example conversions

There are several ways to do conversions, and if you have a preferred way of doing it, go with that. If not, one way to do these is by moving the decimal place over. Each time you go from a smaller to a larger prefix, move the decimal point left. Move right if going from a larger to a smaller prefix. All of the prefixes given above correspond to moves of 3 decimal places from the nearest one. There are other ways that we will show in the examples below.

Example 1: *Convert 4000 megabytes to gigabytes.*

Using the decimal point approach, going from mega to giga is a move of 3 decimal places left, so we go from 4000 to 4, giving us 4 gigabytes.

Using the other approach, since *mega* is million, 4000 megabytes is 4000 million bytes, or 4,000,000,000 bytes. Written like this, we see we have 4 billion bytes, and since billion is *giga*, this is 4 gigabytes.

Example 2: *Convert 50 milliseconds to nanoseconds.*

Using the decimal point approach, since we are going from a larger prefix *milli* to a smaller prefix *nano*, we move the decimal point right. We are going two prefixes over, so that is $3+3=6$ decimal places in total, So 50 ms turns into 50,000,000 ns.

Another way is since a millisecond is a thousandth of a second and a nanosecond is a billionth of a second, there are 1,000,000 nanoseconds in a millisecond. So multiply 50 by 1,000,000 to get 50,000,000 nanoseconds.

3.2 Machine and Assembly Language

We will now look at how the Central Processing Unit (CPU) works. Let's start with a little code. On the left is a short C program. A program called a *compiler* translates that code into the *machine language* code in the middle. The *assembly language* code on the right is a human-readable version of the machine language.[1]

```
#include <stdio.h>
int main() {
    int x = 2;
    int y = 3;
    printf("%d", x + y);
    return 0;
}
```

```
1139:    55                      push   rbp
113a:    48 89 e5                mov    rbp,rsp
113d:    48 83 ec 10             sub    rsp,0x10
1141:    c7 45 fc 02 00 00 00    mov    DWORD PTR [rbp-0x4],0x2
1148:    c7 45 f8 03 00 00 00    mov    DWORD PTR [rbp-0x8],0x3
114f:    8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
1152:    8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
1155:    01 d0                   add    eax,edx
1157:    89 c6                   mov    esi,eax
1159:    48 8d 05 a4 0e 00 00    lea    rax,[rip+0xea4]
1160:    48 89 c7                mov    rdi,rax
1163:    b8 00 00 00 00          mov    eax,0x0
1168:    e8 c3 fe ff ff          call   1030 <printf@plt>
116d:    b8 00 00 00 00          mov    eax,0x0
1172:    c9                      leave
1173:    c3                      ret
```

Each line in the display after the C code contains an *instruction*, something that tells the CPU what to do. The first part of each line contains the relative memory addresses of the instructions, where each one is located. After that are the machine language instructions, followed by the assembly language instructions. Each line consists of a single instruction.

The machine language codes are purely numerical. They are written in hex to make them a bit more readable to humans, though to the CPU they are purely binary values. These numerical codes are the precise instructions that the CPU runs, one after the other, more or less. As we can see, the machine language instructions vary in size.

The assembly language on the right is a human-readable version of those machine language instructions. They

[1] This comes from the Linux command-line tool `objdump`.

are composed of a few parts. The first part is the operation that is being done, like `push`, `mov`, `sub`, etc. The `push` operation pushes a value on a part of memory called the stack. The `mov` operation stores a value. The `sub` operation subtracts two values. Instructions are often simple operations like this, and they are implemented inside the CPU using logic gates, like we covered earlier. The other parts of the instructions are the operands or arguments. Some of these, like `rbp`, `rsp`, and `edx`, are *registers*, which are small storage units inside the CPU. Most instructions involve a register.

Each type of CPU has its own *instruction set architecture* (ISA). The ISA is the CPU's design of what instructions are available, the sizes and numbers of registers available, the format of the instructions, what data types are supported, and several other factors. The code above is for the x86 ISA, which is what Intel and AMD CPUs use.

Later in these notes, we will look at how to write assembly language in order to program the CPU directly. For now, the main thing to remember is that the CPU understands machine language, which are numerical instructions for the CPU. Assembly language is the human-readable equivalent of machine language. Most of the time, people write in a higher-level programming language. That code is translated into machine language code via programs called *compilers* and *interpreters*. People do sometimes write assembly language directly, especially when doing low-level hardware and systems programming, and when extremely efficient code is needed, though compilers are better at generating efficient code than most programmers. Almost no one ever writes pure (numerical) machine language.

3.3 Parts of a CPU

In the early days of computing, in the 1930s and 1940s, people had different visions of how computers might work. Some of the early machines were programmed purely by moving wires around. Eventually, the concept of a *stored-program computer* won out. This is the idea that we write code, store it somewhere, load it into the computer, and then the computer automatically runs it. We are all so familiar with this idea now that it seems strange that things might be otherwise, but this was not so obvious to early computer scientists.

One of the key early designers of computers was John von Neumann. The design he described in an influential paper came to be known as the *Von Neumann Architecture*. Most common processors follow this design. We will start by giving a simplified view of CPUs that use this design. Later we will look at some of the improvements that modern processors use.

The key parts of a CPU are below:

- *Registers* — These are small memory units. Each one typically holds a single 32- or 64-bit value. Most CPUs have a few dozen registers. Many CPU operations involve working with values in registers. In the assembly code given earlier, `rbp`, `rsp`, `eax`, `edx`, `esi`, `rax`, and `rip` were registers. Notice that almost every instruction uses one of them. Accessing registers is much faster than accessing data stored in RAM.
- *Arithmetic Logic Unit (ALU)* — The ALU is the brains of the whole operation. It is where operations like addition, multiplication, etc. as well as logical operations like comparisons are run.
- *Control Unit* — This is the manager of the CPU. Its job is to coordinate everything. It reads the instructions from memory, figures out what they are saying, retrieves values from memory, and arranges for the ALU to have everything it needs to do its work.

In addition to these, there are *buses*. These are communication lines that data and commands flow through. There are multiple buses connecting everything together. A simple system might have a data bus for moving data between memory and the CPU components, a command bus for commands from the control unit, an address bus for keeping track of which memory addresses are being used, and an I/O bus for getting data from external devices. Though this is a simplification of what a real computer would look like.

The fetch-decode-execute cycle

In a simplified CPU, each instruction goes through a three-step process called the *fetch-decode-execute cycle*. Here is a brief description of each stage.

- *Fetch* — The control unit has to retrieve the instruction from memory. It uses a register called the *program counter* to keep track of the memory location of the current instruction.
- *Decode* — The control unit has to break the instruction into its component parts to figure out what operation is to be done and what registers or memory locations are involved. The control unit also has to set everything up for the execute phase, to get the values from memory/registers ready for the ALU.
- *Execute* — The ALU runs the instruction. The result is written to the appropriate registers or memory locations, depending on the instruction.

You will often see textbooks that talk about this as a five-stage process, with the execute stage being broken into execute, memory, and write-back steps. These last two steps involve the control unit handling any memory operations that are part of the instructions and then writing any output values to registers.

In a simple CPU, each of the three or five stages takes one or more CPU cycles to do, with only one stage active at a time. Modern CPUs have a more complicated scheme that we will look at soon.

The clock

To keep everything synchronized, there is a clock. For instance, we wouldn't want the decode stage starting before the fetch stage has finished, so everything is synced to a clock. The clock is an electronic circuit that continually oscillates back and forth between 0 and 1.

Clock speeds are measured in hertz, or cycles per second. Clock speeds of early PCs of the 1970s were a couple of megahertz, meaning the clock alternates between 0 and 1 a few million times per second. Speeds increased exponentially from the 1970s through the early 2000s, where they leveled off around 3-6 gigahertz (GHz). Clock speed is a measure of how many instructions per second the CPU can do. Very simple operations, like adding two integer register values will take a single cycle on average. Some more complex instructions, such as division, will take considerably longer. All other things being equal, a 3 GHz processor is 1000 times faster than a 3 MHz processor.

As the clock speed gets higher, power usage increases rapidly, so it's likely that processor speed won't increase much beyond where it is currently. A 1 terahertz processor, for example, would require so much power that it would melt. Since this leveling off of clock speeds in the early 2000s, computers have been sped up by adding more *cores*. A core is essentially an entire mini-CPU, and a *multicore* processor is one that has several CPUs. Computers now typically have 2 to 16 cores, with high-end machines having many more. Roughly speaking, a computer with n cores can be about n times as fast as one with only a single core.

Propagation delay

The speed of light turns out to be an important factor in CPU design. Nothing can move faster than the speed of light, which is around 186,000 miles per second. This works out to about 1 foot per nanosecond. For a typical 3 GHz CPU, each cycle takes 1/3000000000 seconds, or 1/3 of a nanosecond. Thus during a single CPU cycle, light can travel around 4 inches. Electricity moves at about 2/3 the speed of light, so we can't reasonably expect to send data more than a couple of inches in a CPU cycle. Thus, if we want to be able to access things, like register values, in a single CPU cycle, the CPU must be kept small.

Von Neumann bottleneck

Von Neumann's design has program instructions and data both stored in memory, and the same data bus is used for both. The bottleneck here is that since they share a bus, we can either retrieve data or instructions, but not both at the same time. This slows things down somewhat. On the other hand, having the program instructions and data in the same place does make computers easier to build, and it allows the program instructions to be treated as data, which makes them easy to modify while the program is running. It is also more space efficient, as whatever space is not used by the program instructions can be used for data.

There are other architectures that do not have the bottleneck. One is the Harvard architecture, which is used in some places, though not in any common laptops or phones. Because computers built with the Von Neumann architecture became so successful, it has become the standard way of doing things, and this not likely to change anytime soon. The bottleneck has less of an effect now than it used to due to various improvements in processor design that we'll cover next.

3.4 Improvements to Modern Processors

The CPU design covered in the previous section is a simplified view of things, a decent model for what early CPUs looked like. Here we will look at some things modern processors do for performance gains.

Multicore processors As mentioned earlier, most CPUs nowadays have 2 or more independent CPUs that can each be running their own programs.

Hyperthreading Threading is an important programming concept that we will cover later. CPUs do their own version of threading, called *hyperthreading*. The basic idea is that the CPU will have two programs both ready to execute at the same time. Both share the CPU resources. When one of them stalls for some reason, like if it is waiting for data to arrive from somewhere, then the CPU can make sure resources are used by the other program. The operating system also does this with ordinary threads, but hyperthreading happens at a lower level.

Pipelining In the simple CPU example of the previous section, we looked at the fetch-decode-execute cycle. Early CPUs would run each of those three stages for each instruction. When they were done, then the three stages would run for the next instruction. Modern CPUs allow each of these to be running at the same time with different instructions. This idea of multiple stages running at the same time with different instructions is called a *pipeline*. Below on the left is what things look like for a non-pipelined CPU and on the right is what they look like for one using pipelining.

No pipelining

Step	Fetch	Decode	Execute
1.	Instr 1	–	–
2.	–	Instr 1	–
3.	–	–	Instr 1
4.	Instr 2	–	–

With pipelining

Step	Fetch	Decode	Execute
1.	Instr 1	–	–
2.	Instr 2	Instr 1	–
3.	Instr 3	Instr 2	Instr 1
4.	Instr 4	Instr 3	Instr 2

Modern CPUs have pipelines that break each of the three stages into more fine-grained steps, up to a few dozen steps in some cases.

Out-of-order execution Modern CPUs will often reorganize a program's instructions to speed things up. For instance, reading from a register is much faster than reading from memory. So if instruction 1 involves reading from memory while instructions 2 and 3 are simple instructions adding register values, then the CPU might run

2 and 3 while waiting for instruction 1's memory access to finish. Of course, it can only do this if those instructions don't depend on the result from the first instruction. The important point to remember is that you might code something in one way, even if writing directly in assembly, but there is no guarantee that the CPU will run them in that order.

Branch prediction A lot of programs have if/else blocks, like below

```
if (x != 0)
    y = 3 / x;
else
    y = 0;
```

Often what happens is one of the two branches, either the if branch or the else branch, ends up being taken most of the time. The CPU takes advantage of this to get a pretty big speedup in some cases. It keeps track of how often each branch is taken, and next time the if/else condition comes up, it chooses the more likely branch, without waiting for the result of the condition. If it turns out correct, then we get a little speed boost. If it turns out wrong, then the instructions it was running turn out to be wasted, and there is a bit of a performance penalty. This is called *branch prediction*.

GPUs

A *graphics processing unit* GPU is an alternative to CPUs for certain kinds of work. GPUs were initially for speeding up the processing of on-screen graphics. They were built to do the same computationally intense operations to the many pixels on the screen. Because of this, they are really good at doing the same computation in parallel to many different things at the same time. Unlike a CPU that typically only has a few cores, GPUs have hundreds or even thousands of cores. But those cores are limited in that they are good for doing computation, but not general operations like a CPU would do. A simple, predictable task with a lot of computation that can be done in separate, independent portions is a perfect task for a GPU. An unpredictable task with a lot of branching is something a CPU handles better. GPUs are widely used for crypto mining and for the massive processing power needed by modern AI tools.

RISC vs. CISC Two competing ideas in CPU design are *Reduced Instruction Set Computer* (*RISC*) and *Complex Instruction Set Computer* (*CISC*).

A RISC processor has a relatively small set of simple instructions. With a RISC processor, you build more complex operations by combining those couple of instructions together. For instance, some RISC processors don't include a division operation since the circuitry to implement division is a bit complicated, and the operation ends up taking dozens of CPU cycles to run. Instead, it's up to the programmer or compiler to use the operations the CPU does provide to put together a division routine. A CISC processor, on the other hand, has a larger set of instructions, some of which can be complicated and slow.

Since CISC designs involve building hardware for a lot of different operations, those operations will be faster since they are running in special hardware rather than in software. On the other hand, RISC designs lead to a simpler CPU where most instructions complete in one cycle, which allows for simpler and faster pipelining.

Because RISC has fewer CPU instructions, programs written for RISC CPUs tend to be larger than CISC programs. For instance, dividing two numbers on a CISC CPU would be just one instruction, whereas it would eventually translate to dozens on a RISC CPU. The more complicated circuitry of a CISC CPU uses more power than a RISC computer, however.

As we see, each design has its own pros and cons, and there are popular CPUs that use each type. Early CPU design tended to be more CISC, but RISC has become more popular over the years. Intel CPUs have a CISC design, but under the hood they have become more RISC-like. Specifically, the instruction set the programmer sees is CISC-like, but the CPU translates those into a smaller RISC-like set of operations.

Common processors

In the 1970s, Intel put out a processor called the 8086. It was hugely influential. Over the coming decades, they put out new versions, each one backwards-compatible with 8086, so that the instruction set from that old processor is still the standard, though with many updates. It is called *x86*. AMD started out making cheap processors that followed the x86 standard, but now they are an important competitor to Intel. AMD and Intel chips are widely used in laptops, desktops, and servers.

A different architecture is the ARM processors. These are RISC chips. Their simpler design and lower power usage make them better for use in smartphones. Most smartphones use ARM chips or ARM-compatible chips made by Apple. Apple also uses them in its laptops.

Chapter 4

Memory

Computers have many different types of memory, including RAM, registers, cache, hard drives, removable storage (like SD cards and USB drives), and cloud storage. As we move farther from the CPU, memory gets slower and larger. As a brief overview, the fastest and smallest storage are the CPU's registers, just a few but very fast. Next comes cache, which holds commonly-accessed values. After that is main memory or RAM, which holds the bulk of the memory used by programs. Next comes permanent storage on hard drives and various removable storage devices. These are often considerably larger than RAM and slower. The slowest and largest memory is cloud storage, which is storage on servers in data centers somewhere.

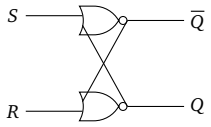
4.1 RAM, Registers, and Cache

RAM

RAM stands for *random access memory*. The name comes from the fact that you can access any item in RAM in about the same amount of time as any other item. This is to distinguish it from sequential memory, where things are arranged in a line of sorts, and you have to go through everything ahead of what you want before getting to it. An example of sequential memory is a cassette tape, where you would have to fast-forward through the tape to get to the point you want. RAM is more like a CD or record player, where you can skip to whatever point you want right away.

RAM is *volatile*, which means that it needs power in order to store data, and when the power is turned off, all the data goes away.[2] Registers and cache are also volatile.

There are two types of RAM: SRAM and DRAM, standing for static and dynamic RAM. They differ in how they store data. SRAM uses electronic components called latches and flip-flops to store data. A simple example of an SR latch is below. It is built from two NOR gates that feed back into each other. This feedback idea is a key for using logic gates to store values. The latches used by most SRAM can be built with 6 transistors.



DRAM, on the other hand, stores data using a single transistor and a capacitor. A capacitor is an electronic device that stores electricity and slowly discharges.

The differences in design of each type lead to different use cases for each. SRAM uses more power to store its values, which also generates more heat. The reason is that SRAM needs current in its circuits continuously,

[2] In digital forensics, if you super-cool RAM, it slows down the process by which the RAM cells lose their data, allowing recovery of data for a longer time after the power is turned off.

while DRAM only needs a boost of electricity about once every 64 ms to maintain the charge in the capacitors. SRAM requires more space, about 6 times the space of DRAM for the same amount of storage since it has more transistors. SRAM is also more expensive than DRAM. However, SRAM has one major benefit, which is that it is faster. Because of all of this, SRAM is used for registers and cache memory, and DRAM is used for main memory (what we typically call RAM).

It typically takes on the order of about 100 ns to access items from RAM.

Registers

As mentioned earlier, registers are small storage units directly on the CPU. Some serve very specific purposes, like the program counter that keeps track of the location of the current program instruction. Others are general purpose and used for storage of values the program is using. Most CPUs have a few dozen to maybe a few hundred registers, depending on the architecture. Register access is usually a single CPU cycle, around 1/3 of a nanosecond on a 3 GHz processor.

Cache memory

Cache memory sits in between the core of the CPU and RAM. It is used for commonly-accessed items to save the time needed to pull values from RAM. Cache access times can be 10 to 100 times faster than RAM. There are three levels of cache:

L1 cache This is the smallest and fastest cache. It sits on the CPU itself, with each core having its own. L1 cache is typically around 32 to 64 KB. About half of that is used to store program instructions, and the rest is for data. Program instructions are the core of what the CPU does, so it's reasonable to have them in the fastest cache. Access times are typically 1 to 4 cycles, which corresponds to a few tenths of a nanosecond to around 1 ns.

L2 cache This is the medium level of cache. It sits on the CPU and is shared by the CPU cores. It is typically around 256 KB to 1 MB. Access times are around 3 to 6 ns.

L3 cache This is the largest cache, typically 8 to 64 MB, and sometimes larger. Access times are around 10 to 20 ns.

A natural question is why not make cache larger if it is so fast? There are a couple of reasons. First, part of the speed of cache comes from the very fact that it is *small*. The larger the cache is, the longer it takes to search through it. As a simple analogy, think of a library. A library has shelves and shelves of books and it takes awhile to find what you are looking for. A library might have an area out front with the 100 most commonly requested books, where it is quick to find what you want. That is like a cache. But if the common books area had 10,000 books, then it wouldn't be so quick anymore.

Another reason cache needs to be small has to do with propagation delay. The speed of light, as we saw earlier, limits how far things can be from each other if we want to keep access times within a few cycles. If the cache is too large, the outer edges of it would be too far from the ALU and control unit to have fast access times.

Cache policies When we put something into the cache, we usually have to make room for it by kicking something else out. How do we choose what to kick out? There are a lot of interesting algorithms, but to keep things short, the most important is *least recently used* (LRU). That is, we kick out things that haven't been used in a while and try to keep things that have been used recently. The cache doesn't have very long to make this decision, so it approximates LRU by using a few bits of data to track accesses.

On a related note, usually when we put something into the cache, we also put in some of the stuff around it, typically 32 or 64 bytes worth. This is because if we need something from memory, there is a good chance we will also need things stored nearby. This is especially true for array operations.

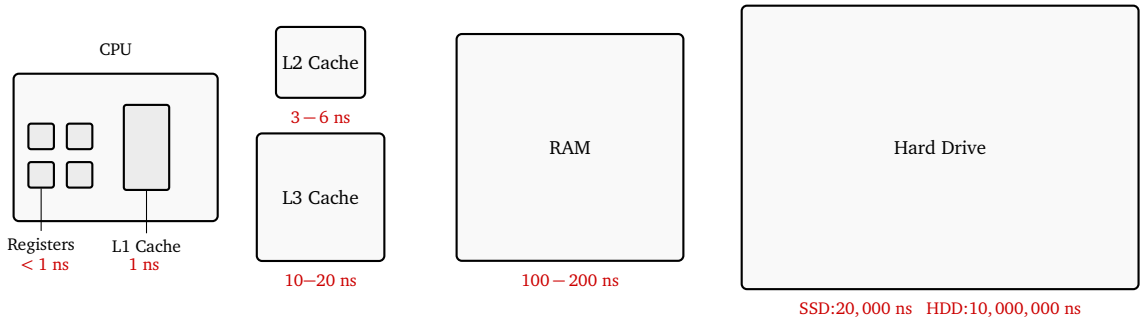
Hard drives

There are two main types of hard drives – hard disk drives (HDDs) and solid-state drives (SSDs). HDDs are the older technology of the two. They consist of spinning disks with magnetic material on them and a read/write head that is positioned over the disk. They look a lot like a record player. SSDs, unlike HDDs, are purely electronic. There are no moving parts in an SSD.

Because HDDs consist of moving physical parts, they are slow. It takes time to move the read/write head to the correct spot on the disk and to wait for the disk to spin around to the right spot. All this typically takes on the order of around 10 ms. SSDs are much faster, with times typically around 20 μs. These are seek times, the time to get to the right spot on the drive. Once you are there, it is considerably faster to read or write a chunk of data.

Because of the speed difference, SSDs are usually preferable to HDDs. The one downside is SSDs are more expensive. SSDs are good as the main drive on which the OS is located, and they are good for gaming. HDDs are good for bulk storage, especially when you need a lot of it. HDDs, because they are physical devices, don't like being dropped. They also don't work quite right at high altitudes, and they can lose data if a strong magnet gets too close to it. SSDs don't have any of these problems, but the technology used to store the data has a limited number of writes and rewrites, so they will eventually wear out.

Below is a diagram showing roughly what memory looks like, along with typical approximate latency times.



Notice how much slower things get as you move to the right. An HDD is more than 10 million times slower than a register access. On the other hand, as you move right, space increases. We only have a few kilobytes worth of registers, several megabytes of cache, several gigabytes of RAM, and maybe a few terabytes of hard drive. Not shown here is cloud storage, which is essentially unlimited. Latency for cloud storage is on the order of 10 to 100 ms (10 to 100 million nanoseconds) or higher since we are limited by the speed of light in how long it takes to transfer data back and forth to remote locations.

Chapter 5

The Operating System

The operating system (OS) is what makes the computer actually usable. It handles loading and running programs, managing memory, and handling the details of the various devices like the keyboard or hard drive.

The most important current operating systems are Windows, MacOS, Linux, Unix, Android, and iOS. Except for Windows, all of these derive from the original Unix operating system of the early 1970s. Many operating systems derive from Linux as well, most notably Android.

5.1 Processes

A *process* is a running program. Most computers typically have dozens or even hundreds of processes running at any point in time. But they only have one CPU with maybe a few cores. How do so many processes get to run? The answer is called *multiprogramming*. This is where the operating system rapidly switches back and forth between the various processes, giving each a small amount of time to use the CPU.

A typical time slice might be 20 ms, with important programs sometimes getting more. The operating system might run A for 20 ms, then B for 20 ms, then C for 20 ms, eventually coming back to A again. For someone using a program, this gives the illusion that their program has full control of the CPU, but in fact it just uses it for short bursts at a time. It's a little like those flip book kid toys that, give the illusion of motion when you flip them fast enough, even though they are just a set of static pictures.

Another important concept is that most programs spend a lot of time doing I/O (input/output). That is, they are often waiting for input, such as a keypress, mouse motion, file read, or download; or they are waiting for output, like a writing to a file or to the screen. The CPU does not need to be involved during I/O. So when a program pauses to wait for I/O, the OS can let another program use the CPU.

The changeover from one process using the CPU to another is called a *context switch*. A context switch involves the OS copying the current process's CPU register values to memory to put the process in a state of suspended animation. For the incoming process, the OS copies its saved register values from memory into the CPU registers.

5.2 Virtual Memory

In really old computer systems, every process had full access to all of memory. Full access like this is bad news from a security perspective. If you're running a game from a sketchy company and your banking app, you would not want the game to be able to access anything to do with the banking app. So part of the OS's job is to keep each process's memory separate from other processes.

Modern operating systems use a system of *virtual memory*. This is where memory looks simple to processes and the people that program them, but under the hood things are actually different. The OS handles all those details.

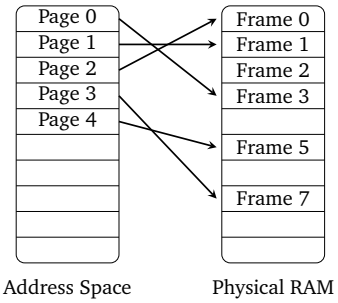
Each process is given an *address space*. This is a set of 2^{32} memory addresses on a 32-bit machine or a set of 2^{64} addresses on a 64-bit machine. This address space is *virtual*. It is not real. Instead, the OS translates addresses in the address space into actual locations in RAM. As we'll see, a process's memory might be spread out all around RAM. But the programmer doesn't have to worry about that. The address space looks just like a really long array of memory locations.

In a 32-bit system, there are $2^{32} \approx 4$ billion possible addresses. This is 4 GB of space. So 32-bit computers are limited to 4 GB of RAM. You could put more RAM into the system than that, but it wouldn't be able to work with it. In a 64-bit system, there are 2^{64} addresses. This corresponds to around 18 exabytes of space. However, the hardware of most systems currently doesn't go past 48 bits, so you're limited to 2^{48} addresses, which is around 280 TB.

Virtual addresses are usually displayed in hex. Each hex digit is 4 bits, so a 32-bit address will be 8 hex digits long. The addresses just increase by 1 from `0x00000000`, `0x00000001`, `0x00000002`, all the way up to `0xFFFFFFFF`. In a 64-bit system, addresses are 16 hex digits long.

Paging

Most operating systems use a system called *paging* for virtual memory. Each process's address space is broken into equal-sized chunks called *pages*. These are usually 4 KB in size. Physical RAM is also broken into 4 KB chunks called *frames*. Each page is mapped to a specific frame in RAM, like in the figure below.



Notice in particular, that the frame for a page can be anywhere. Things don't have to go in any particular order. Frame 4 in the figure above doesn't have a page corresponding to it, but it might correspond to some other process's page. In general, each process has its own set of mappings.

A data structure called a *page table* is used to track which pages are in which frames. For reasons we won't get into, this page table is usually multiple levels deep.

Each 4 KB page has 4096 addresses in it. For a virtual address like `0xFF12049A`, the higher bits of the address tell what the page number is and the lower bits tell where in the page we are at. This is called the *offset*. Bitwise operations are used to extract the page number and offset.

TLB

The page table is stored in RAM, and since the table has multiple levels, multiple RAM accesses are needed when we want to find the address. This is slow. A special cache called a *translation lookaside buffer* (TLB) is used to store the results of recent page table lookups. Finding a result in the TLB takes only a few nanoseconds, versus the several hundred nanoseconds that would be needed for a full lookup in the page table.

It's called a *TLB miss* when a page is needed but is not in the TLB. TLB misses cause a big performance loss since page table walks are so much slower. If you're a programmer and your code is running unexpectedly slow, it's possible that something in your algorithm is causing the TLB not to be used in an efficient way. It could also be that something about your algorithm is not making good use of the caches. Chances are that the problem is something higher level than this, but the TLB and caches are something you should consider.

Why use this system?

It's natural to wonder why the OS spreads a process's memory all around RAM instead of storing it all in one place. This reason has to do with what is called memory fragmentation. If the OS stores a process's memory all in the same place in RAM, we run into issues when a process needs more memory. There might be another process's memory in RAM immediately after it in RAM, and there might not be any contiguous chunks of RAM available that are big enough for the additional memory the process needs. We could "defragment" RAM by moving everything around so that all the free space is at the end, but this takes a long time and would have to be done all the time. Instead, with paging, anytime a process needs more memory, we just have to find free 4 KB pages somewhere and give them to the process.

Swapping

A typical system might have 16 GB of RAM and 2 TB (2000 GB) of hard drive space. It might be nice if we could use some of that hard drive space to expand RAM. The problem is hard drives are so much slower than RAM. However, most systems do use an idea called *swapping* to do this.

The idea is that a large file on the hard drive (maybe around 16 GB in size) is used as an overflow for RAM. The OS stores infrequently used items from RAM there. When one of those items is needed, the system copies it into RAM and boots something else out of RAM onto the hard drive to make room.

When an item is needed that is not in RAM but has been swapped out to the hard drive, that is called a *page fault*. A page fault is undesirable because we will need a slow hard drive operation to read the item back in from memory. When this happens, usually the OS will do a context switch to another process while the hard drive read is happening in the background. A system that has too many page faults will really start to slow down. This often happens if we are using too much memory.

5.3 Operating System Functions

In the previous sections we looked a little at how the OS manages processes and memory. Here we will look at a few other things it does.

The heart of the OS is called the *kernel*. It is the part responsible for actually running the system. The kernel runs in *kernel mode*, and everything else runs in *user mode*. Anything running in kernel mode has full access to the system. Processes running in user mode are much more limited.

A nice example of this limitation is writing to the hard drive. It would not be a good idea to let any process other than OS kernel read or write to the hard drive directly. A mistake in programming could easily lead to key operating system files being overwritten or damage to the drive. A rogue program could also read private user data from things like banking apps.

On the other hand, many ordinary programs need to be able to save things on the hard drive. So what is done is a *system call*. This is where a user-mode program asks the OS to do a privileged operation for it. System calls look just like ordinary function calls to a programmer. When one is made, the OS will perform a context switch over to its code for the system call, run it, and then context switch back to the program. Besides disk I/O, some other common system calls include device I/O, creating and destroying processes, and memory allocation.

Firmware

Firmware is the software that runs a device. For instance, an HDD's firmware controls the spinning of the disk and the motion of the read head. Sometimes, if a device is not working right, it could be due to a bug in the firmware. Updating the firmware might help. People sometimes write their own versions of firmware for devices to get them to do things the manufacturer didn't intend.

BIOS/UEFI

Computers themselves have firmware. This is called BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface). One of their jobs is to load the operating system when the computer is booted. They also have code that the operating system uses to interact with hardware, such as a few simple routines for keyboard and mouse input. These are also where you can find some important system settings. To get into them, you usually have to hit a certain key when booting up the computer. It's often one of the function keys, though that varies by manufacturer.

BIOS is the older of the two. It has some limitations, such as not working on large hard drives, so UEFI was developed as an improved version of BIOS. People often refer to UEFI as BIOS, however.

Device Drivers

Many devices connected to computers are incredibly complicated pieces of electronics that require very careful programming to interact with. For instance, some SSDs have manuals over 1000 pages in length, covering things such as error-correcting codes, dealing with bad blocks, voltages and timing sequences, wear leveling, and so much more. On the other hand, someone writing a program just wants to call a file-writing function to save a file.

To allow everyone to not have to worry about all the low-level details, we have *device drivers*. This is software that connects the device to the operating system. Basically, someone goes through once and deals with all the intricacies of programming the device directly, and the operating system and everyone using it can rely on that code. Device driver code makes up a large portion of an operating system's code.

If a device is malfunctioning, it could be because of a problem with the device driver. One way to potentially fix the problem would be to reinstall the device drivers.

Programmed I/O and Direct Memory Access

Programmed I/O is where the CPU does the reading/writing to the device. The alternative is *Direct Memory Access*. This is where there is a dedicated hardware device whose job is to do the I/O. A benefit of this is that the CPU is free to do other things while the I/O is happening, which is good because device I/O can be slow. Writing a large file to an HDD could take several seconds, and in that time, a CPU could execute billions of other instructions.

Both approaches are used in modern operating systems. If the amount of device I/O is minimal, it's not worth the setup time needed to start the DMA device, so programmed I/O is used in that case. DMA is better for longer I/O jobs.

5.4 Files

Most people have an idea of what a file is, but have you ever thought about what a file really is? It's not really a physical thing. It's more of an abstract concept. To an operating system, a file is a virtual object that has a name, some properties, and something to indicate where on the drive its contents are stored. You can usually view most of these by right-clicking and selecting "properties" or "get-info."

The properties mentioned above are called *metadata*. These include things like the file size, creation date, last modified date, owner, permissions, etc.[1]

[1] One interesting piece of metadata associated with some image files is the GPS coordinates of where the photo was taken. If you post the photo online, people can use that to figure out where you were, though many social media sites remove that data.

File types

Files usually have a file type. This is often, but not always, indicated by its extension. Some examples are .txt, .jpg, .exe, .py, .java, .docx, and .pdf. The data of each file is just a bunch of 0s and 1s. Different file types organize those 0s and 1s differently.

One of the simplest is an ASCII text file. Every byte (8 bits) corresponds to a character. Unicode-based text files are similar except that multiple bytes can correspond to characters. A lot of file types are actually text files, especially programming language files, like .py, .java, .html, etc.

Image files are stored differently. A simple image format is .bmp. The start of the file has information about the image, such as its dimensions and how the colors are stored. After that, each pixel's RGB values are stored. A more complicated format is .jpg. The start of the file also has image info, but after that the pixel data is in a compressed format that makes for a smaller file size.

Most file types are like this in that they have their own specific format for what is stored where. It's fun to try to open non-text files in a text editor. You'll usually see garbage, but with a few human-readable strings. The garbage comes from the text editor trying to interpret the bytes of the file as characters, when in fact they are storing the file's data in some other format. The few human-readable strings can contain interesting data. For instance, executable files often contain strings hardcoded into the program. These might be things like passwords or IP addresses that the programmer did not want to be visible. But they are visible by opening the file in a text editor or using a tool, such as the Linux `strings` command.

The first few bytes of a file often tell you what type of a file it is. For instance, BMP files always start with the ASCII characters BM (`0x42` and `0x4D`) in hex. JPG files always start with `0xFF`, `0xD8`, `0xFF` in hex. One nice way to see these is to open up the file in a hex editor.

The file system

Operating systems have a *file system*. Each OS maker has its own, often more than one. For instance, Windows has NTFS, FAT32, and exFAT. The file system's job is to store files on disk, keep track of where they are located, manage the free space on the disk, and control various details about files.

Most file systems treat hard drives as broken into equal-sized pieces called *blocks*, which are usually 4096 bytes (4 KB) in size. The block size is the smallest unit of storage. A file with only 100 bytes of data will use up an entire 4 KB block. So if you had 10,000 100-byte files, though there is only $10000 \times 100 = 1$ MB of data, they take up $10000 \times 4096 \approx 41$ MB, which is a huge difference.

Most file systems use a *bitmap* to keep track of which blocks on a drive are free and which are in use. It's basically just an array of 0s and 1s, with 0 meaning the block is free and 1 meaning it's in use.

Each file's data occupies certain blocks on the disk. Often, file systems track these as a list, like `[45, 46, 47, 104, 105, 188, 189, 200]`. Notice that the blocks need not be contiguous, or all in one piece. You might create a file and add some data and then add more data later. The blocks right after where the early file data is stored might have been filled by other files. So the OS uses the bitmap to find space somewhere else on the drive. The important lesson here is that files are often stored with pieces all over the drive.

Summary The lesson to remember here is this: The hard drive is broken into equal-size blocks that are usually around 4 KB each. The OS tracks which blocks are free and which are in use, and for each file, it tracks which blocks are part of that file.

Defragmenting Ideally, we would have the file all in one contiguous piece, since hard drives, especially HDDs, are faster reading bytes sequentially, rather than jumping all around the disk. For an HDD moving around the disk requires repositioning the read head, which is slow. But because files are constantly being created, added to, and deleted, it's not really feasible to store them contiguously. One option is to *defragment* the drive. This involves moving everything around so all the files are in one piece. For HDDs, this can give a good performance

boost to reading files, but it can take multiple hours to do. It's not recommended for SSDs because the memory cells in SSDs wear out with repeated use, and all the writing that happens in a defrag makes them wear out sooner. The performance gain we get from contiguous files isn't really that much anyway for SSDs.

Deleting a file When you delete a file, its contents are usually still there on the drive until something overwrites them. When a file is deleted (and removed from the recycle bin), the OS will mark the file's blocks as free in the bitmap. The data from the file will sit there in those blocks undisturbed until used to store another file's data.

If you are getting a rid of a hard drive and don't want people to be able to recover data from it, there are a few things you can do:

- One option is physical destruction. If you have really sensitive data on the drive, and you want to guarantee no one gets it, then the best option is destruction. This can involve smashing it with a hammer, degaussing an HDD with a magnet, or, in really serious cases, dissolving the drive in acid.
- If you're planning on giving the drive to someone else or selling it, then there are other options. It's not enough to delete all the files or reformat the drive since the data won't be overwritten. Instead, there are programs out there that overwrite all the data on the drive with garbage. These can work well, but sometimes they are not enough. As mentioned earlier, SSD memory cells can wear out with repeated use. To prevent this, hard drive firmware implements what is called wear leveling, where it avoids writing to certain blocks to keep them from getting worn out. So the software that writes all the blocks of the disk might not actually write to every block of the disk.
- Some SSD manufacturers have a feature built in to their drives that will reset all of the memory cells on the drive.
- One other option is to encrypt the entire drive and then throw away the key.

Moving vs. copying Let's look at moving a file from one folder to another on the same drive versus making a copy of that file in another folder. Which takes longer? The answer is copying. When you move a file, it is not actually moved to another place on the hard drive. All that happens is the OS's file system updates its information on what folder the file belongs to. On the other hand, to make a copy means that we have to write the file's bytes to the drive somewhere else.

System crashes When a computer crashes, data can be lost from the file system if we are not careful. If a file was being written to when the crash happens, the bitmap and the part of the file system tracking the blocks that are part of each file can get out of sync. Operating systems run various utilities to fix the file system when this happens.

Chapter 6

Threads

6.1 Introduction

Threads are a way for a process to split itself into several "mini-processes" that are each scheduled by the operating system's process scheduler. Threads can be useful for allowing the program to do other things while it is waiting for an I/O operation to finish. On a multicore system, threads can allow a program to use multiple cores at once. Below is a really simple Python program with two threads.

```
from threading import *

def f():
    print('hello from thread 1')

def g():
    print('hello from thread 2')

t1 = Thread(target=f)
t2 = Thread(target=g)

t1.start()
t2.start()
```

The code creates two threads, tells them what code to run, and then starts them. These threads are now separate entities that run on their own without having to wait for one or the other to run.

One place threads are useful is if we are getting user input and we want to be able to do other things while waiting for that input. Without threads, the program would pause at the input statement, and nothing else could happen until after that.

Threads can often give a speedup to programs. How much depends on if it is doing CPU-bound tasks or I/O-bound tasks. CPU-bound tasks are ones that use the CPU a lot. For example, finding all the prime numbers less than 1 million is a CPU-bound task. An I/O-bound task is one that does I/O, like waiting for a keypress, waiting for a network download, or reading a file from a hard drive. Many tasks are a mix of CPU-bound and I/O bound.

On a single-core processor, CPU-bound tasks won't get much of a speedup from using threads. This is because they all need the CPU, and there is only one core to go around. In fact, because the OS will have to context-switch between our threads, we might actually get a bit of a slowdown. However, if we have multiple cores, then threads can give a speedup to CPU-bound threads as each thread can be running on a separate core. The speedup maxes out once the number of threads matches the number of cores. [2]

[2] CPUs do have something called hyperthreading that can allow for a bit of a speedup beyond the number of cores, but it won't be that big.

Python note: Python has something called the Global Interpreter Lock (GIL) that only allows one thread to be running at a time, regardless of how many cores you have. It is fundamental to the way Python is implemented, though there has been work on eliminating it. If you want to use threading to speed up something that is CPU-intensive, Python threads won't help. You'll need to use a different language or look into multiprocessing.

For I/O-bound tasks, we can get a speedup on both single-core and multicore processors. I/O-bound tasks tend to not use the CPU very much, so even if there is only one core, that is not much of a limitation. A nice example is a program that downloads several webpages, each in its own thread. Threads allow us to start up several different downloads that can all be going at the same time. Starting up and processing each download only uses the CPU for a few milliseconds, so the CPU doesn't act as a bottleneck, like it does for CPU-bound tasks. Instead, most of the time is spent waiting for the data to transfer over the network.

6.2 A Couple of Demonstrations

The code below downloads several different Wikipedia pages. We time it two ways: first with each download happening one after the other, and second with each download in its own thread.

```
from threading import *
import requests
from time import time

pages = ['https://en.wikipedia.org/wiki/france',
'https://en.wikipedia.org/wiki/italy',
'https://en.wikipedia.org/wiki/belgium',
'https://en.wikipedia.org/wiki/germany',
'https://en.wikipedia.org/wiki/poland',
'https://en.wikipedia.org/wiki/russia',
'https://en.wikipedia.org/wiki/norway',
'https://en.wikipedia.org/wiki/sweden',
'https://en.wikipedia.org/wiki/canada',
'https://en.wikipedia.org/wiki/congo',
'https://en.wikipedia.org/wiki/algeria',
'https://en.wikipedia.org/wiki/morocco',
'https://en.wikipedia.org/wiki/malta',
'https://en.wikipedia.org/wiki/japan',
'https://en.wikipedia.org/wiki/china',
'https://en.wikipedia.org/wiki/mongolia',
'https://en.wikipedia.org/wiki/india']

def get(page):
    text = requests.get(page)

start = time()
for page in pages:
    get(page)
print('Non-threaded version:', time()-start)

T = []
start = time()
for page in pages:
    t = Thread(target=get, args=[page]))
    T.append(t)
    t.start()

for t in T:
    t.join()    # wait for each thread to finish
print('Threaded version:', time()-start)
```

On my computer, the non-threaded version took 2.4 seconds to run, and the threaded version took 0.37 seconds. Why the speedup? The running time of the non-threaded version is the sum of all the download times. We have to wait for the first download to finish before we can start the second, we have to wait for the second to finish before we can do the third, etc. On the other hand, with the threaded code, all the downloads are happening at the same time, so the total time is just whatever the time of the longest download is.

Below is a different program, this time in Java. It counts how many primes are between 1,000,000 and 2,000,000. The non-threaded version just repeatedly counts this 8 times. The threaded version does the same count in 8 simultaneous threads.

```
import java.util.ArrayList;
import java.util.List;

public class PrimeThreadingExample {

    public static void primes(int m, int n) {
        int found = 0;
        for (int i=m; i<=n; i++) {
            boolean prime = true;
            for (int d=2; d <= Math.floor(Math.sqrt(i)); d++) {
                if (i % d == 0) {
                    prime = false;
                    break;
                }
            }
            if (prime)
                found++;
        }
        System.out.println("I found " + found + " primes.");
    }

    public static class PrimeThread implements Runnable {

        @Override
        public void run() {
            primes(1000000, 2000000);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        int numThreads = 8;

        long start = System.currentTimeMillis();
        System.out.println("Without threads");
        for (int i=0; i<numThreads; i++)
            primes(1000000, 2000000);
        System.out.println((System.currentTimeMillis()-start)/1000.0);

        start = System.currentTimeMillis();
        System.out.println("With threads");
        List<Thread> list = new ArrayList<Thread>();
        for (int i=0; i<numThreads; i++) {
            Thread t = new Thread(new PrimeThread());
            list.add(t);
            t.start();
        }
        for (Thread t: list)
            t.join();

        System.out.println((System.currentTimeMillis()-start)/1000.0);
    }
}
```

I ran this on my 8-core processor using a number of threads running from 1 to 10. The times are below.

# threads	unthreaded	threaded	ratio
1	573	554	1
2	1173	555	2.1
3	1729	555	3.1
4	2291	590	3.9
5	2833	644	4.4
6	3382	703	4.8
7	3918	759	5.1
8	4492	825	5.4
9	5019	887	5.7
10	5584	955	5.9

Notice we get a 2 times speedup with 2 cores, a 3 times speedup with 3 cores and a 4 times speedup with 4 cores. So we can see the benefit of threads. Beyond 4 threads, the speedup isn't quite as good. This has to do with something CPUs do when you are using all the cores at once. It slows down the overall clock speed when they are all in use to avoid overheating. So even though it's an 8-core CPU, we only reach around a 6 times speedup, rather than an 8 times speedup. Nevertheless, threads do give a nice speedup.

6.3 Locks

When working with threads, there are various problems that can happen with the threads interfering with each other. This can happen especially when both have access to a shared resource. Here is one way this can happen. The code below has two threads updating a shared count variable. Each adds 1,000,000 to the count, so overall the count should end up at 2,000,000 when the program finishes. But it doesn't, at least if run on Python 3.11 or earlier. The output is usually a seemingly random value between 1,000,000 and 2,000,000.

```
from threading import *

def f():
    global count
    for i in range(1000000):
        count += 1

def g():
    global count
    for i in range(1000000):
        count += 1

count = 0

t1 = Thread(target=f)
t2 = Thread(target=g)

t1.start()
t2.start()

t1.join()
t2.join()

print(count)
```

What happens is the `count +=` operation is actually done in three parts: (1) load the variable from memory, (2) add 1 to it, (3) store the new value. If a context switch happens in between these steps, bad things can happen. For instance, suppose the first thread runs and advances the count to 10,000. Then suppose the second thread runs, advances the count to 20,000, and gets interrupted when adding 1 to make it 20,001. Specifically, suppose a context switch happens after the addition operation, but before the store operation. Thread 1 then runs, and maybe it advances the count to 30,000. When the next context switch to Thread 2 happens, that

thread picks up right where it left off, which was that it was about to store 20,001. When it does that, it wipes out all the progress Thread 1 had made.

This is why we get seemingly random values. Sometimes the context switch will happen in the middle of the `count += 1` operation, and we lose progress. Other times, the context switch will happen after the operation and nothing bad happens. But the exact sequence of context switches will be pretty random, depending on how long of a time slice the thread has to run and where it gets to when that time slice is up. This is an example of a *race condition*, which is a problem that happens with some threaded programs, where the output will depend on the exact state of the computer at the time the program is run, meaning the outputs can vary from run to run, even if there is no randomness in the algorithm.

We can also get a problem like this on a multicore system. If two threads are running the `count += 1` code at the same time, you can have a situation where they both hit the `load` part of that operation at more or less the same time. So they might both load the value 20,000, each add 1, and each store 20,001. But really 20,002 is what we should get since there are two separate additions.

One solution to this problem is to use a *lock*. A lock is a way to restrict access to a shared resource to just one thread at a time. Any time a thread wants to use the resource, it first has to acquire the lock. If the lock is currently in use, then the thread is forced to wait for the lock to become free before it can get the resource. To use a lock in Python, we can create it via `lock = Lock()`. The for loop in the counting code could use the lock like below, where we acquire the lock before using the shared count variable and release it when done.

```
for i in range(1000000):
    lock.acquire()
    count += 1
    lock.release()
```

The resulting code will always give the correct value of 2,000,000. On the other hand, the overhead of continually acquiring and releasing the lock, as well as threads having to wait for the lock to be free, means the code runs considerably more slowly.

An interesting note about this race condition is it doesn't happen if the for loop is short, like only running to 1000 instead of 1,000,000. The reason for this is that the loop is so short that it finishes before a context switch can ever happen, so no weirdness is possible.

Another interesting note is that the race condition does not typically happen in Python 3.10 or later. Changes were made to the Python Global Interpreter Lock that prevent it. However, the issue still occurs in earlier versions of Python and in many other languages.

6.4 Threading in Python

Getting started First, make sure to import the threading module like below:

```
from threading import *
```

To create a thread called `t` with a target function `f`, use the line below:

```
t = Thread(target=f)
```

The target function contains the code that the thread will run. To start the thread, use its start method. Here is a program showing how to create and run a thread:

```
from threading import *

def f():
    print('hello from thread 1')

t1 = Thread(target=f)
t1.start()
```

A program with multiple threads Here is an example of a program that creates and runs two threads.

```
from threading import *

def f():
    print('hello from thread 1')

def g():
    print('hello from thread 2')

t1 = Thread(target=f)
t2 = Thread(target=g)
t1.start()
t2.start()
```

Note that each thread has a different target function. If both threads were to run the exact same code, it would be okay to give them both the same target function. Each thread would essentially get its own copy of that function.

Creating a list of threads Sometimes we want to create a bunch of threads. Rather than having a separate variable for each, we can create a list, like below:

```
from threading import *

def f(name):
    print(f'hello from thread {name}')

threads = []
for i in range(10):
    t = Thread(target=f, args=[i])
    t.start()
    threads.append(t)
```

The one tricky thing in the code above is the `args` keyword argument. For this program, to give each thread its own "name", the target function has a name parameter. In order to set that value when we create the thread, we use the `args` argument. The arguments are sent in a list. Note that if you run this, the printing will be all jumbled up. This could be fixed with locks.

The join method The join method is used to wait for a thread to finish. The code after the join method is called will not be run until after the thread is finished. Here is an example:

```
from threading import *

def f():
    print('hello from thread 1')

t1 = Thread(target=f)
t1.start()
t1.join()
print("This won't print until after the thread is done.")
```

Locks

To create a lock called `lock`, use the line below:

```
lock = Lock()
```

Locks have two methods we will use: acquire and release. Surround the code you want to protect with acquire

and release. if the lock is free when a thread calls acquire, then the thread can run the code the code being protected by the lock. If the lock is in use, the thread has to wait to run that code until the lock is free. The lock is freed by the release method.

In the example below, we take the list of threads program from the previous section and add a function called `safe_print`. We only ever want one thread at a time printing to the screen. To do this, we create a lock and surround the print statement with calls to acquire and release.

```
from threading import *

def safe_print(string):
    lock.acquire()
    print(string)
    lock.release()

def f(name):
    safe_print(f'hello from thread {name}')

lock = Lock()

threads = []
for i in range(10):
    t = Thread(target=f, args=[i])
    t.start()
    threads.append(t)
```

6.5 Threading in Java

Getting started Here is a how to create a thread in Java.

```
public class ThreadHelloWorld {
    public static class MyThread implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello world");
        }
    }

    public static void main(String[] args) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
}
```

We first create a static class that implements the Runnable interface. Anything that implements that interface must have a `run` method. That method contains the code we want the thread to run. This is analogous to the target function in Python. Then to create and run the thread, we make an object from our class and call its `start` method.

A program with multiple threads Here is an example of a program that creates and runs two threads.

```
public class TwoThreads {
    public static class MyThread1 implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello from thread 1.");
        }
    }

    public static class MyThread2 implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello from thread 2.");
        }
```

```
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread1());
        Thread t2 = new Thread(new MyThread2());

        t1.start();
        t2.start();
    }
}
```

Note that each thread has a different class. If we want both threads to run the exact same code, it would be okay to make them both objects from the same class.

Creating a list of threads Sometimes we want to create a bunch of threads. Rather than having a separate variable for each, we can create a list, like below:

```
public class ListOfThreads {

    public static class MyThread implements Runnable {
        private String name;

        public MyThread(String name) {
            this.name = name;
        }
        @Override
        public void run() {
            System.out.println("Hello from thread " + name);
        }
    }

    public static void main(String[] args) {
        List<Thread> threads = new ArrayList<Thread>();
        for (int i=0; i<10; i++) {
            Thread t = new Thread(new MyThread(""+i));
            threads.add(t);
            t.start();
        }
    }
}
```

For this program, to give each thread its own "name", we introduce a class variable called name into the `MyThread` class and initialize it in a constructor. If you run this, notice that things don't always print out in the same order. This is a fact of life of threads—you're at the mercy of the OS scheduler in terms of when things will run.

The join method The `join` method is used to wait for a thread to finish. The code after the `join` method is called will not be run until after the thread is finished. Below is an example. Java requires us to do something about a potential exception, which is why there is a `throws` statement in the `main` method.

```
public class ThreadJoin {
    public static class MyThread implements Runnable {
        @Override
        public void run() {
            System.out.println("Hello world");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new MyThread());
        t.start();
        t.join();
        System.out.println("This won't print until after the thread is done.");
    }
}
```

Locks

To create a lock called `lock`, use the line below:

```
Lock lock = new ReentrantLock();
```

A reentrant lock is like an ordinary lock except that it allows the same thread to reacquire a lock it already has. This is useful if the thread's code is recursive, and it also avoids us having to check to make sure we haven't acquired the lock already, which can get tricky. Note that Python also has a reentrant lock, though we never talked about it.

Locks have two methods we will use: `lock` and `unlock`. Surround the code you want to protect with `lock` and `unlock`. When a thread calls the `lock` method and the lock is free, then the thread can run the protected code. If the lock is in use, the thread has to wait to run that code until the lock is free. The lock is freed by the `unlock` method.

In the example below, we use locks to protect a shared counter variable to avoid a well-known race condition. There are two threads each running the same code that adds 1 to a counter 10 million times. The code should print out 20000000. Try removing the lock and notice that you won't necessarily get 20000000.

```
public class ThreadLock {
    static int count = 0;
    static Lock lock;

    public static class MyThread implements Runnable {
        @Override
        public void run() {
            for (int i=0; i<10000000; i++) {
                lock.lock();
                count++;
                lock.unlock();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        lock = new ReentrantLock();
        Thread t1 = new Thread(new MyThread());
        Thread t2 = new Thread(new MyThread());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(count);
    }
}
```

A different approach Java provides a feature called the `synchronized` keyword that allows us to do the above program a little differently. The `synchronized` keyword sort of acts like a lock that doesn't allow more than one thread to run the `increment` function at a time.

```
public class ThreadLock2
{
    static int count = 0;

    public static class MyThread implements Runnable {
        @Override
        public void run() {
            for (int i=0; i<10000000; i++)
                increment();
        }
    }

    public static synchronized void increment() {
        count++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new MyThread());
        Thread t2 = new Thread(new MyThread());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(count);
    }
}
```

Chapter 7

C Programming

This chapter is an introduction to C programming. It assumes you are already familiar with programming, especially in a related language, like Java.

C is worth learning because it is one of the closest languages to the system itself. It gives you access to memory in ways that higher-level languages like Java and Python don't. Being so close to the system also allows C to be one of the fastest languages around. Operating systems, especially Linux, are mostly programmed in C. C is also used a lot for embedded systems programming, which is where you are writing to code to run appliances, IoT devices, and various types of equipment. C gives easier low-level hardware access than many other languages.

7.1 Basic C Programming

Java based its syntax off of C, so if you know Java, you'll find a lot of things work the same way in C. There are some differences that we'll note below. Here is a sample C program that asks the user to enter two numbers and whether to add or subtract them. Then it does the appropriate operation.

```c
#include <stdio.h>

int main()
{
    int num1, num2;
    char type;

    printf("Enter the first number:\n");
    scanf("%d", &num1);

    printf("Enter the second number:\n");
    scanf("%d", &num2);

    printf("Enter a for add, s for subtract:\n");
    scanf(" %c", &type);

    if (type == 'a')
        printf("%d", num1+num2);
    else if (type == 's')
        printf("%d", num1-num2);
    else
        printf("I don't understand.");

    return 0;
}
```

Notice that the basic syntax of braces, semicolons, operators, loops, and if statements is the same as in Java. Some differences to note:

- Importing things is done with `#include`.
- Instead of Java's `public static void main(String[] args)`, C uses `int main()`. You will also often

see `int main(int argc, char *argv[])` if the program takes command line arguments. The main function has a return type of int. This is the status code that the C program will send when it closes. You can use `return 0` at the end of `main` to return the status code 0, for success, though it is usually okay to leave it off.

- Input and output in C is different than in Java. Above we use `scanf` and `printf`, though there are other options. There is more on these a little later.

Data types Here are some similarities and differences between C and Java data types:

- Just like Java, C has `int`, `float`, `double`, and `char` data types.
- C has some data types, such as `unsigned char`, that are useful for low-level programming. Java doesn't have these.
- C originally didn't have a boolean data type, though newer versions did introduce one. Mostly, people use an `int` instead, with 0 = False and 1 = True.
- C does not have a string data type like Java. We'll look at strings a little later.
- The `char` data type can be interpreted as either a character or as an integer corresponding to the character's ASCII code. For instance, `char c='a'` and `char c=97` have the same effect.
- Arrays are similar to Java, but the declaration syntax is a bit different. Where Java would have `int[] a = new int[10]`, C would have `int a[10]`.
- C is not an object-oriented programming language. There are no classes or objects.

See https://introcs.cs.princeton.edu/java/faq/c2java.html for a nice side-by-side comparison of C and Java.

Functions Functions work similarly in C and in Java except that C functions usually have a prototype, which is basically just a repeat of the declaration line. Here is an example:

```
#include <stdio.h>

int add(int x, int y); // this is the prototype

int main() {
    printf("%d", add(2,2));
    return 0;
}

int add(int x, int y) {
    return x + y;
}
```

Since C is not object-oriented, there are no `public` or `static` keywords. The declaration just contains the return type, function name, and parameters.

7.2 The printf and scanf functions

A useful function for printing things is `printf`. It uses formatting codes to specify how things look. These formatting codes have been adopted in many other languages, so it's good to know them. Here is a basic example:

```
printf("x is %d", x);
```

The `%d` acts as a placeholder, and the value of `x` is inserted there. Here is an example with two variables:

```
printf("x is %d and y is %d", x, y);
```

There are a variety of different formatting codes. Here are some of the most common:

code	description
`%d`	for displaying integers in decimal
`%f`	for floating point numbers
`%g`	chooses between regular or scientific notation for floats
`%s`	for strings
`%c`	for single characters
`%x`	for displaying integers in hexadecimal
`%p`	for pointers (covered later)
`%u`	for unsigned integers

The codes can be customized in a few ways. For example, to allot 9 spots for a floating point number with 2 after the decimal point, use `%9.2f`. Any slots not used will be replaced with spaces, which means you can use the code to right-justify numbers. If you want an integer to have leading 0s, like 0034 instead of 34, you can use `%04d`. Here is a short program with some examples.

```
#include <stdio.h>

int main() {
    int x = 14;
    double y = 12.934;
    printf("%6.2f\n", y);
    printf("%18.10f\n", y);
    printf("%d\n", x);
    printf("%5d\n", x);
    printf("%05d\n", x);
    printf("x=%d and y=%f\n", x, y);
    printf("%s\n", "abcdefg");
    printf("%c %d\n", 'a', 'a');
    return 0;
}
```

Here is the output of the code above:

```
 12.93
     12.9340000000
14
   14
00014
e=439326816 and y=12.934000
abcdefg
a 97
```

A few notes: To use `printf`, include `stdio.h`. The `printf` function doesn't add a newline, so if you want things to appear on separate lines, you need to manually add a `\n` at the end.

scanf The `scanf` function can be used to get input from the keyboard. Here is an example that gets an integer.

```
int a;
char s[10]; % strings are just arrays of characters
printf("Enter a number");
scanf("%d", &a);
```

The function uses formatting codes to indicate the data type we're looking for. The variable name goes at the end, preceded by a `&` character. The `&` is the "address of" operator. It's needed here for `scanf`. There is more about it in the section on pointers. Like `printf`, the `scanf` function is in `stdio.h`.

7.3 Strings

C doesn't have a string data type. Instead, we use arrays of characters. Here are a few examples of declaring strings:

```
char a[] = {'a', 'b', 'c', '\0'};
char b[] = "abc";
char c[7];
```

Notice in the first example, the null character `'\0'`. Strings in C must be terminated with a null character. That is how C determines where the string ends. A very common error in C is to forget the null character, which can cause string functions to end up reading past the intended end of the string and into whatever is stored in memory right after the string. That can cause serious problems. The declaration in the second line is a shorthand for what was done in the first line, with the null character being automatically inserted by the C compiler. The third line declares a string long enough to hold 6 characters (leaving one for the null character). In the other examples, the C compiler infers the string length from the right side of the declaration.

Because of how pointers work (see below), if you want to use `scanf` to get a string, no `&` character is needed, like below.

```
char s[100];
printf("Enter a string");
scanf("%s", s);
```

In Java, if you want the length of a string `s`, you would use `s.length()`. But since C is not object-oriented, we can't do something like that. Instead, string operations are done with functions, which are mostly in `string.h`. For instance, here are a few examples:

```
#include <stdio.h>
#include <string.h>

int main()
{
    char a[10] = "abc", b[] = "defg";
    printf("Length of b: %d\n", strlen(b));
    strcat(a, b);  // concatenate b to a, like a = a + b in Java
    printf("%s", a);
    return 0;
}
```

7.4 Pointers

One of the trickiest things for people coming to C from Java are *pointers*. They are a way to directly access memory locations in a process's address space. A pointer variable is one that holds memory locations. Here are the basic ideas:

To get the memory location of a variable, use the `&` operator. For instance, `&x` gives the memory location of `x`. To store a memory location in a variable, use a pointer. Let's assume `x` and `y` are integer variables and `z` is a double. Here is how to declare and assign values to a pointer.

```
int *ptr = &x;      // create a pointer and use it to store the memory location of x
ptr = &y;           // now the variable stores the location of y
double *ptr2 = &z; // pointer's data type needs to match what it is pointing to
```

The `*` operator in the declaration is used to indicate that the variable being declared is a pointer. That `*` operator is also used to get the value stored at that memory location. The technical term for this is *dereferencing* the pointer. The example below will print out the value stored at the memory location pointed to by `ptr` (assuming it's an integer):

```
printf("%d", *ptr);
```

The `*` operator can be used to change the value stored at the memory location being pointed at, like below.

```
int x = 3;
int *ptr = &x;
*ptr = 9;  // x is now equal to 9.
```

When printing pointers and memory addresses with `printf`, use the `%p` formatting code, like in the examples below:

```
printf("%p", &x);
printf("%p", ptr);
```

It will give something like `0x7ffe796ce53c`, though the exact value and length of the address will depend on the system you are using and where exactly things are located in the process's address space, which will likely be different each time you run the program.

Here is a complete program showing the concepts we have just seen.

```
#include <stdio.h>

int main() {
    int x = 2;
    printf("Memory location of x: %p\n", &x);

    int *ptr = &x;
    printf("Address held by ptr: %p\n", ptr);
    *ptr = 12;
    printf("%d\n", x);
    return 0;
}
```

Pointers are used extensively in C. They give you low-level access to the machine, and that is especially helpful when doing low-level programming, like device drivers and working with hardware. Pointers also help to save memory since you can pass around a pointer to something rather than making copies of it.

Pointers and functions Pointers are often as arguments to functions so that the function can change values passed to it. Here is an example:

```
#include <stdio.h>

void func(int *x, int y);

int main() {
    int a=4, b=7;
    func(&a, b);
    printf("a=%d b=%d", a, b);
    return 0;
}

void func(int *x, int y) {
    *x = 78;
    y = 99;
}
```

The result of this program is that the variable `a` in main is changed to 78, while `b` is unchanged. This has to do with the parameters of the function. The `*x` parameter is a pointer, which means it will hold the memory location of the variable passed to it (note the `&a` in the function call). Thus we can use `*x` to change the value of things passed to it since we have direct access to the memory location. On the other hand, the `y` parameter is not a pointer. It is instead a local variable and it receives a copy of what's passed to it. Any changes to `y` will only affect that local copy and will have no effect on the calling variable (`b` in this case).

Below is another example that uses pointers to create a function that swaps two variables. It's not possible to write something like this in Java or Python.

```
#include <stdio.h>

void swap(int *x, int *y);

int main() {
    int a=4, b=7;
    printf("a=%d b=%d", a, b);
    swap(&a, &b);
    printf("a=%d b=%d", a, b);

    return 0;
}

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Pointer arithmetic Pointers hold memory addresses, and we can do math on them. For instance, if `p` is an `int` pointer holding the address `0x7ffe2000`, then `p+1` will be `0x7ffe2004` and `p+2` will be `0x7ffe2008`. Note that the addresses go up by 4 each time. This is because `p` is an `int` pointer, and the `int` data type is 4 bytes. If `p` had been a `char` pointer, the address would only go up by 1 since the `char` type is 1 byte in size. We can also subtract pointers to tell how far apart they are.

Let's look at a small example of basic pointer operations and pointer arithmetic. Suppose part of memory looks like below (specifying the memory location, value at that location, and variable referencing that location). The variables are all ints. Assume `ptr` is a pointer variable pointing at `w`.

address	value	variable
0xffff7720	40	w
0xffff7724	60	x
0xffff7728	10	y
0xffff772c	90	z

Below are some

- `&x` — This is `0xffff7724`, the address of `x`.
- `ptr` — This is `0xffff7720` since `ptr` holds the address of `w`.
- `*ptr` — This is 40, the value stored at the memory location held by `ptr`.
- `ptr + 1` — This is `0xffff7724` since we are going one integer's worth of memory past the address currently in `ptr`.
- `ptr + 2` — This is `0xffff7728` since we are going two integer's worth of memory past the address currently in `ptr`.
- `*(ptr + 1)` — This is the value stored at `0xffff7724`, which is 60.
- `*ptr + 1` — This is 41 since we are getting the value stored at the memory location held by `ptr` and adding 1 to it.

7.5 Memory allocation

In C, if you need a large chunk of memory, you have to allocate it yourself. The function that does this is called `malloc`. It is found in `stdlib.h`, so that file needs to be included. Here is an example of how malloc works:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *buffer = malloc(10000);
    buffer[0] = 'a';
    buffer[1] = 'b';
    buffer[2] = '\0';

    printf("%s\n", buffer);
    free(buffer);

    int *buffer2 = malloc(50000*sizeof(int));
    return 0;
}
```

The `malloc` function takes one argument, which is the number of bytes to allocate. It returns a pointer to where that memory is located, which will always be on the heap. Arrays, on the other hand, are local variables on the stack. The heap is much larger than the stack, so it's better to use `malloc` when you need a large block of space.

Notice that in the second call to `malloc` above, we used `sizeof(int)`, but not in the first call. To see why, note that the argument to `malloc` is the number of bytes to allocate. A `char` in C is one byte, while an `int` in C is

typically 4 bytes. In order to allocate enough space for 50,000 integers, we would need 50,000×4 = 200,000 bytes. In general, it's a good idea to use the `sizeof` function when using `malloc`, even for `char`.

Once the memory has been allocated, you can treat it just like an array, accessing elements using square brackets. When you are done with the memory, you can release it by calling the `free` function. If you're not using much memory, it's not worth worrying about freeing it since it will automatically be freed when your program exits. But if you are using lots of memory, especially in a long-running program, then it's good to call `free` to prevent your program from running out. Freed memory goes back to a pool of free memory that can be allocated for other things. Many higher level languages, like Java and Python, take care of this for you automatically, using a process called *garbage collection* that looks for things that are no longer being used and frees them . In C, you have to do this yourself. It's easy to accidentally lose track of the pointer to where the memory is allocated. This is a *memory leak*. Here is an example:

```
int *buf = malloc(10000*sizeof(int));
buf = malloc(2000*sizeof(int));
```

After the first line, `buf` stores the memory location of the start of the 10,000 integer allocation. But we lose this location when we run the second line. After that, we have no way of knowing what that first location was and so we can't free it. That memory is essentially lost and wasted until the program exits.

Arrays vs. malloc If we need space for 2000 ints, we could either create an array via something like `int a[2000];` or we could use `malloc`. What's the difference? A process's memory is typically broken up into a couple of areas for different purposes. One of those holds the program's code, another holds global variables. The other two are the *stack* and *heap*. The stack is used to hold information needed for function calls. In particular, local variables for functions are stored on the stack. The stack generally isn't all that big, so if we need a large enough array, it won't be able to hold it. That's when to use `malloc`. An array defined on the stack will also be destroyed when the function ends, whereas memory on the heap persists. So if you want something accessible by multiple functions, heap memory allocated by `malloc` is a better choice.

7.6 Other Topics

Structs

C is not object-oriented, but it does have a way to group variables together into something a little like an object. This is called a *structure*. Here is an example:

```
#include <stdio.h>
#include <string.h>

struct Person {
    int age;
    char name[32];
};

int main() {
    struct Person bob;
    bob.age = 20;
    strcpy(bob.name, "what");
    printf("%d %s", bob.age, bob.name);
    return 0;
}
```

We access members of a structure using the dot operator just like in Java. If we have a pointer to a structure, however, then we need to use the `->` operator to access the members. Here is what that would look like for the `Person` structure above:

```
struct Person *ptr = &bob;
ptr->age = 30;
```

Buffer overflows

A very common error in C that can have disastrous security implications is a *buffer overflow*. C won't try to stop you if you try to read or write past the end of an array or string. Many other languages will crash your program with an index out of bounds error, but not C. Here is an example:

```
#include <stdio.h>
#include <string.h>

int main() {
    char a[10] = "abcdefg";
    char b[10] = "wxyz";

    strcpy(b, "00011122233344455566");
    printf("a=%s\n", a);
    printf("b=%s\n", b);

    return 0;
}
```

The program creates a string `b` large enough to hold 9 characters (and a null), and then uses the `strcpy` function to copy a string much longer than 9 characters into it. This has the effect of writing the long string into the memory locations that are right after `b`, which will very likely be string `a`. When I ran it, it ended up printing that `a` is equal to `33444555666`.

Buffer overflows are arguably the most serious class of vulnerability out there since they can lead to total takeover a machine if an attacker can overwrite the right parts of memory. In C, a stack frame is generated for each function call. It holds information related to the function call, like local variables and the place in the code segment to return to once the function is done. If an attacker can cause one of those local variables to overflow far enough to overwrite the return address, then they can control what code is run after the function. They often overwrite that return address with the address of code that opens up a shell prompt, allowing them to run commands on the system.

So many of the biggest vulnerabilities for the past few decades have been buffer overflows, often in very widely used software. It's unfortunately very easy for a programmer, even an experienced one, to accidentally introduce a buffer overflow vulnerability into their code. There are coding practices to help avoid them, but they can be a pain, and it's easy to accidentally forget to use them.

Working with pointers

When reading other peoples' C code, you'll often find that they use pointers a lot, especially when working with strings. There are various shortcuts also in use. Here is a typical way someone might code a string copy function from scratch[1]

```
char* string_copy(char *dest, const char *src) {
    char *p = dest;

    while (*src != '\0') {
        *p = *src;
        p++;
        src++;
    }
    *p = '\0';

    return dest;
}
```

If you plan on doing a lot of C, you'll want to get good at reading this kind of code. Notice that there is no index variable here, like would be common in Java. Instead, everything is done with pointers. In the loop, the lines `p++` and `src++` change the addresses stored in those pointers, moving up 1 address each time. The line `*p = *src` takes the value currently pointed to by `src` and stores it where `p` is currently pointing to. The while loop condition says to keep going until the `src` pointer is at a location with the null character, which indicates the end of the string. Below is a more compact version of this same function:

[1] Note that there is a function `strcpy` in `string.h` that does this already. This example is just for demonstration.

```
char* string_copy(char *dest, const char *src) {
    char *original_dest = dest;

    while ((*dest++ = *src++) != '\0');

    return original_dest;
}
```

At this point, we are not yet interested in writing this code ourselves, just being able to read it. The entire while loop has been compressed to a single line. C allows you to put increment statements inside loop and if conditions, which is what happens here.

Another example Here is a function that returns the location of the first occurrence of character `c` in a string `s` and −1 if it isn't found.

```
int index(char *s, char c) {
    int i = 0;
    while (s[i] != 0) {
        if (s[i] == c)
            return i;
        i++;
    }
    return -1;
}
```

This version treats the string as a character array and doesn't really use much to do with pointers. Below is a rewrite of this to use pointers.

```
int index2(char *s, char c) {
    char *t = s;
    while (*t) {
        if (*t == c)
            return t - s;
        t++;
    }
    return -1;
}
```

The loop continues until the character pointed at by `t` is null (ASCII value 0). This value is interpreted by the while loop as false (0 is always false and anything else is true). Inside the loop, we use `*t` to get the current character and we use the pointer arithmetic `t - s` to compute how many characters away `t` currently is from the start of the string. At the end of the loop, `t++` uses pointer arithmetic to move forward in memory one character at a time.

Chapter 8

Assembly Language

8.1 Introduction

In this chapter we will look at programming in assembly language. In assembly, you are directly programming the CPU without anything getting in the way. Assembly is not widely used, but it does have its niches. It is used especially when you need direct access to the underlying hardware. OS programmers use it in places. People writing firmware for various devices often write some of their code in assembly. Game engines and other applications where raw speed is needed have critical parts of the code written in assembly. Assembly is also really important for reverse engineering malware and other programs. Besides these applications, understanding a little about assembly language is an important part of being educated about computers.

Each family of processors has its own assembly language. The two most popular are x86 assembly and ARM assembly. The former is used primarily in Windows and Linux laptops, desktops, and servers, while the latter is used especially in smartphones and IoT devices. We will look at x86 assembly in these notes. Once you understand the concepts of assembly language, it's not too hard to move from one type to another.

8.2 x86 Assembly Language

The x86 assembly language got its start in the 1970s with the Intel 8086 chip, one of the first microprocessors. The 8086 processor evolved over the years to new processors that kept the same basic assembly syntax, though adding new features and changing things a bit. The original 8086 was a 16-bit processor. Its successors were 32-bit and eventually 64-bit processors.

Below is an example of a C program and the assembly language it translates to. A *compiler* is a program that turns a high-level language, such as C, into machine language code. The C code here was compiled into an executable file, which contains the machine-language instructions of the program. That executable was then opened in a program called a *debugger* that translates the machine-language instructions into their assembly language equivalents. A nice online site where you can test this out is `https://www.onlinegdb.com`. If you are using this site or the GNU debugger in Linux, start by typing the command `set disassembly intel` to use the Intel syntax, which is what these notes use. Then type `disas main` to "diassemble" the machine-language code into human-readable assembly.

```
int main() {
    int x = 1;
    int y = 2;
    y = y + x;
    return 0;
}
```

```
0x1129 <+0>:     push   rbp
0x112a <+1>:     mov    rbp,rsp
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x1
0x1134 <+11>:    mov    DWORD PTR [rbp-0x8],0x2
0x113b <+18>:    mov    eax,DWORD PTR [rbp-0x4]
0x113e <+21>:    add    DWORD PTR [rbp-0x8],eax
0x1141 <+24>:    mov    eax,0x0
0x1146 <+29>:    pop    rbp
0x1147 <+30>:    ret
```

At the start of each line of the diassembly, we see the memory address of the instruction. After that, the number in slanted brackets just indicates how many bytes that instruction is from the start of the `main` function. After that we have the actual assembly language. Each instruction consists of the command – like `mov`, `add`,`pop`, etc. – followed by its operands, which can be registers, memory locations, or constants. Some of the registers we see above are `rbp`, `rsp`, and `eax`.

Registers

Recall that registers are small storage units on the CPU. In 64-bit x86 assembly, there are several 64-bit general-purpose registers, which can be used for many purposes. They are called `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, and `r15`. The first six used to have more specialized uses, which is why their names are so weird, but nowadays they are used for whatever. Those are the ones you'll see most commonly in disassembly. Below are a couple of other important registers to know:

- `rbp` and `rsp` – These are used to indicate the start and end of the stack frame, which is where the local variables are stored.
- `rip` – This is the instruction pointer, which keeps track of the memory location of the next instruction to run.
- `rflags` – This contains various status and control flags, especially ones that are used in various comparison operations.

The original version of x86 assembly from the late 1970s was 16 bits. In the 80s, it was extended to 32 bits, and in the 2000s to 64 bits. They kept the same registers, but extended them. For instance, in the original one of the registers was `ax`. When the architecture was extended to 32 bits, the `ax` register became `eax`. And when it was extended further to 64 bits, the `eax` register became `rax`. But we still use `eax` to refer to the lower 32 bits and `ax` to refer to the lower 16 bits of the `rax` register. You will also sometimes see `al` to refer to the lowest 8 bits, and `ah` to refer to the next 8 bits.

Many of the registers follow a similar pattern. For instance, `rbx` is the full 64-bit register, `ebx` is its lower 32 bits, `bx` is its lower 16 bits, with `bl` and `bh` referring to the lower and upper 8 bit portions of `bx`.

Instructions

Below are a few basic instructions to know:

- `mov` – This is for storing values in registers or in memory. The instruction `mov a, b` is essentially like doing `a = b`.
- `add` – This is for adding two integers. The instruction `add a, b` adds `a` and `b` and stores the result in `a`
- `sub` – This is for subtracting two integers. The instruction `sub a, b` does $a - b$ and stores the result in `a`.

- `imul` – This multiplies two signed integers. The instruction `imul a, b` does $a \times b$ and stores the result in `a`. It can often be used for unsigned integers as well, though there is an instruction `mul` that is specifically for that.
- `idiv` — This does division and also modulo. In the simplest case, it takes a single operand, divides whatever is in the `rax` register by it, and stores the quotient in `rax` and remainder (modulo) in `rdx`.

In the instructions above, `a` can be either a register or a memory address, and `b` could be a register, memory address, or a constant. There are some limitations to this. One of those is that you can't instructions where both operands are memory addresses. This is a design choice Intel made when developing their processors because it is more difficult to build a CPU where both of the operands are allowed to be memory addresses.

Memory addresses

One of the instructions in the disassembly above is this:

```
mov     DWORD PTR [rbp-0x4],0x1
```

First, note that the disassembly usually specifies numbers in hex. That's what the `0x` indicates. Next, look at the first operation. It's a memory location. Memory locations are indicated by square brackets. For instance, we could do `mov [0x1234], 1` to store the value 1 at the memory location `0x1234`. Usually, it's a pain to directly keep track of memory locations like this. When we start writing our own assembly, we will be able to define variable names to use in places like this. When reading disassembly from compiled code, we usually see memory addresses which are given like `[rbp-0x4]`.

Variables for a program are stored in the stack portion of its address space. Each function has its own part of the stack, called a stack frame. The `rbp` and `rsp` registers track the start and end of the stack frame, and the variables are referenced by their location relative to the base. We might have three integer variables stored at `rbp-0x4`, `rbp-0x8`, and `rbp-0xc`, for instance.[1]

The `DWORD PTR` part is a size specifier to say how much space the variable stored at `rbp-0x4` takes up. The possible size specifiers are these:

- `BYTE` – 1 byte (8 bits). It is for `char` variables
- `WORD` – 2 bytes (16 bits). It is for the uncommonly used `short` data type.
- `DWORD` — 4 bytes (32 bits). This is a "double word". It typically is for the `int` data type and the 32-bit `float` data type.
- `QWORD` — 8 bytes (64 bits). This is a "quad word". It typically is for the `long` or `long long` data type[2] and the 64-bit `double` data type.

8.3 Reading Assembly Code

Example 1 Let's look at the example from earlier and see if we use the assembly to work out what the original C code is.

[1] The `rbp` register is the base pointer, which is actually at the highest address, since the stack grows from higher addresses down toward lower addresses. The `rsp` register is the stack pointer, which is at the lowest address.

[2] On Linux and MacOS, a `long` is 64 bits, but on Windows it is only 32 bits, and `long long` is used for the 64-bit integer type there.

```
int main() {
    int x = 1;
    int y = 2;
    y = y + x;
    return 0;
}
```

```
0x1129 <+0>:     push   rbp
0x112a <+1>:     mov    rbp,rsp
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x1
0x1134 <+11>:    mov    DWORD PTR [rbp-0x8],0x2
0x113b <+18>:    mov    eax,DWORD PTR [rbp-0x4]
0x113e <+21>:    add    DWORD PTR [rbp-0x8],eax
0x1141 <+24>:    mov    eax,0x0
0x1146 <+29>:    pop    rbp
0x1147 <+30>:    ret
```

When disassembling a C program, the first couple of lines of the disassembly (first two in this case) usually involve setting up the stack frame, which is where the program will store its variables. The last couple of lines tear down the stack frame and return from the `main` function (the last three lines here). We won't worry too much about them. Let's look at the rest of the lines.

The third and fourth lines (at offsets +4 and +11) correspond to the `int x = 1` and `int y = 2` lines of the C code. The first one stores the value 1 at the memory location that is 4 bytes from the location stored in the base pointer `rbp`. We can tell it is an `int` datatype by the `DWORD PTR`. The next instruction is similar, except that the location is at `rbp-0x8`. As we read further lines of the assembly, whenever we see `[rbp-0x4]`, we know it is referring to the `x` variable, and likewise `[rbp-0x8]` is referring to the `y` variable.

The next two lines correspond to the `y = y + x` line from the C code. Because x86 assembly can't have both operands be memory locations, it has to move one of the values into a register first. So the instruction at line +18 is essentially doing `eax = x`, and the instruction at line +21 is doing `y = y + eax`. Putting the effects of these two lines together gives the original C code `y = y + x`.

Example 2 Here is a short example without the C code. Let's try to work backward to figure out what the C code must have been.

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0xc
0x1134 <+11>:    mov    BYTE PTR [rbp-0x5],0x61
0x1138 <+15>:    mov    QWORD PTR [rbp-0x10],0x0
```

These are three variable declarations. The first is something like `int x = 12`. We know it's an `int` because of the `DWORD PTR`. The value it is being set to is `0xc` (in hex), which translates to 12 in decimal. It wouldn't be wrong to write `int x = 0xc`, but most likely the original programmer used 12 since people tend to use decimal. We don't know what the original programmer chose for their variable name, so `x` is as good a name as any.

The second line is something like `char c = 0x61`, which would probably have been `char c = 'a'`, since `0x61` (97 in decimal) is the ASCII code for 'a'. We know this is a `char` variable because of `BYTE PTR`. The last line is `long y = 0`, with `long` being indicated by `QWORD PTR`. Note that it is possible that this could be storing a `double` variable, but it most likely isn't. Compilers tend to use the registers `xmm0`, `xmm1`, etc. for those. We won't be focusing much on floating-point here, however.

Example 3 Here is a more involved example:

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x3
0x1134 <+11>:    mov    DWORD PTR [rbp-0x8],0x4
0x113b <+18>:    mov    edx,DWORD PTR [rbp-0x4]
0x113e <+21>:    mov    eax,DWORD PTR [rbp-0x8]
0x1141 <+24>:    add    edx,eax
0x1143 <+26>:    mov    eax,DWORD PTR [rbp-0x4]
0x1146 <+29>:    sub    eax,DWORD PTR [rbp-0x8]
0x1149 <+32>:    imul   eax,edx
0x114c <+35>:    mov    DWORD PTR [rbp-0xc],eax
```

The first two lines assign two integer variables. Let's call them `x = 3` and `y = 4`. The next three lines move `x` into the `edx` register, `y` into the `eax` register, and then add the two registers. At this point, `edx` holds the result of

`x + y`. The next line moves `x` into the `eax` register, and the line after that essentially does `eax = eax - y`, so `eax` holds `x - y`. The second-to-last line multiplies `eax` and `edx`, which corresponds to doing `(x + y) * (x - y)`. It stores the result in `eax`. The last line moves that into another memory location. So these several lines of assembly correspond to the following C code:

```
int x = 3;
int y = 4;
int z = (x + y) * (x- y);
```

Notice how much longer the assembly code is than the C code. This is usually the case. With assembly, you are directly programming the CPU using instructions it knows (like `mov`, `add`, `sub`, etc.), and you don't have access to nice things like math notation, so things take more work.

Example 4 Below is an example with an if statement.

```
int main() {
    int x = 3;
    int y;
    if (x < 1)
        y = 5;
    return 0;
}
```

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x3
0x1134 <+11>:    cmp    DWORD PTR [rbp-0x4],0x0
0x1138 <+15>:    jg     0x1141 <main+24>
0x113a <+17>:    mov    DWORD PTR [rbp-0x8],0x5
0x1141 <+24>:    mov    eax,0x0
```

We see two new operations `cmp` and `jg`. The `cmp` instruction is short for "compare." The next instruction `jg` uses the result of the comparison to make a decision. This instruction is short for "jump if greater than." The "jump" part is that it jumps to a different part of the code.

Looking at the code as a whole, the first line creates the variable `x` and sets it to 3. The next line compares `x` to 0. If `x` is greater than 0, then the code jumps past the next instruction and runs the one at line +24, which corresponds to the start of the `return 0` line of the C code. If `x` is not greater than 0, then it runs the `mov` instruction, which corresponds to setting `y` to 5.

Conditionals and loops are implemented in assembly using comparisons and jumps. Again, assembly is directly programming the CPU, and we don't have any access to high-level syntax like for loops or if/else blocks. Reading the code is a little like a choose-your-own adventure book.

How the instructions work In general, `cmp a, b` compares `a` and `b`, and sets some values in the `rflags` register indicating the result of the comparison. It does this actually by computing `a - b` and noting whether the result is negative or zero.[1]

Jump instructions read the flags set by the `cmp` operation and use them to make their decisions. Under the hood, jump instructions correspond simply to changing the value of the instruction pointer `rip` to the address of the instruction to jump to. Besides `jg`, there are `jge`, `jl`, `jle`, `je`, `jne`, and several others. These correspond, respectively, to jumping if greater than or equal to, less than, less than or equal to, equal to, and not equal to. There is also `jmp`, which we'll see below, that jumps to a given line automatically without checking any conditions.

Example 5 Let's modify the previous example to add an else block.

```
int main() {
    int x = 3;
    int y;
    if (x < 1)
        y = 5;
    else
        y = 6;
    return 0;
}
```

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x3
0x1134 <+11>:    cmp    DWORD PTR [rbp-0x4],0x0
0x1138 <+15>:    jg     0x1143 <main+26>
0x113a <+17>:    mov    DWORD PTR [rbp-0x8],0x5
0x1141 <+24>:    jmp    0x114a <main+33>
0x1143 <+26>:    mov    DWORD PTR [rbp-0x8],0x6
0x114a <+33>:    mov    eax,0x0
```

[1] It also notes whether there was a carry and an overflow, but we won't worry about that.

The big change is the `jmp` instruction in line +24. The flow of the code is that it compares `x` to 0. If it is greater than 0, it jumps to line +26 and sets `y` to 6. Otherwise, it sets `y` to 5 at line 17. However, after that, we do not want to run the code at line +26 that sets `y` to 6. So we jump over it, going right to line +33, which is the start of the `return 0` line of the C code.

Note also that the C compiler tends to produce assembly that is the opposite of the C code. The C code checked if `x` is less than 1, but the compiler turns it into essentially checking if `x` is greater than 0. When writing your own assembly, you can do things either way. The compiler tends to do things this way because it's more efficient.

Example 6 Here is an example with a loop.

```
int main() {
    int x = 0;
    for (int i=1; i<=10; i++)
        x += i;
    return 0;
}
```

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x0
0x1134 <+11>:    mov    DWORD PTR [rbp-0x8],0x1
0x113b <+18>:    jmp    0x1147 <main+30>
0x113d <+20>:    mov    eax,DWORD PTR [rbp-0x8]
0x1140 <+23>:    add    DWORD PTR [rbp-0x4],eax
0x1143 <+26>:    add    DWORD PTR [rbp-0x8],0x1
0x1147 <+30>:    cmp    DWORD PTR [rbp-0x8],0xa
0x114b <+34>:    jle    0x113d <main+20>
0x114d <+36>:    mov    eax,0x0
```

The C code adds up the numbers from 1 to 10. The first line of the assembly sets `x` to 0. The next line is the `i=1` part of the for loop. After that comes the loop. Compilers tend to do loops by putting the loop condition at the end. This is a bit like a do/while loop that you might be familiar with from Java or C.

The code jumps down to line +30, which is the comparison operation. It checks if `i` is less than or equal to 10, and jumps back up to line +20 if it is. Line +20 through line +26 form the body of the loop. Line +20 puts the value of `i` into the `eax` register. The next line adds that to `x` and stores the result in `x` (remember that we can't do an addition operation with both parts being memory locations, so we need to bring in a register). Line +26 corresponds to the `i++` part of the loop.

In general, you can recognize something is a loop and not a simple if statement because a loop in assembly will usually have a jump at the start to a comparison farther down, followed by a backward jump to the instruction right after the initial jump. This causes a cycle, which is basically what looping is.

Example 7 Here is a slightly more complicated example.

```
0x112d <+4>:     mov    DWORD PTR [rbp-0xc],0x2
0x1134 <+11>:    mov    DWORD PTR [rbp-0x10],0x7
0x113b <+18>:    mov    DWORD PTR [rbp-0x4],0x0
0x1142 <+25>:    mov    eax,DWORD PTR [rbp-0xc]
0x1145 <+28>:    mov    DWORD PTR [rbp-0x8],eax
0x1148 <+31>:    jmp    0x1163 <main+58>
0x114a <+33>:    mov    eax,DWORD PTR [rbp-0x8]
0x114d <+36>:    imul   eax,eax
0x1150 <+39>:    cmp    DWORD PTR [rbp-0xc],eax
0x1153 <+42>:    jne    0x115b <main+50>
0x1155 <+44>:    add    DWORD PTR [rbp-0x4],0x1
0x1159 <+48>:    jmp    0x115f <main+54>
0x115b <+50>:    add    DWORD PTR [rbp-0x4],0x2
0x115f <+54>:    add    DWORD PTR [rbp-0x8],0x1
0x1163 <+58>:    mov    eax,DWORD PTR [rbp-0x8]
0x1166 <+61>:    cmp    eax,DWORD PTR [rbp-0x10]
0x1169 <+64>:    jl     0x114a <main+33>
```

The first three lines set up some variables. We'll say these are `int a = 2`, `int b = 7`, and `int c = 0`. After that, the next two lines look like they set up one more variable declaration, `int d = a`.

The code then jumps down to line +58, where a comparison and then a jump backwards to line +33 happen. So there is a loop. The comparison statement compares the variable at `rbp-0x8` (which we called `d`) with the variable at `rbp-0x10`, which we called `b` and jumps back up if `d` is less than `b`. Further, line +54, right before the comparison starts, adds 1 to `d`, so `d` is our loop variable. Putting this all together, we could write this with a loop like `for (int d=a; d<b; d++)`. We could also use a while loop if we wanted.

The inside of the loop corresponds to the lines from +33 to +50. We see at line `39` a comparison followed by a `jne` and a few lines later a `jmp`, so this is an if/else statement. Based on lines +33 through +39, the comparison is checking if `d*d` is equal to `a`. If it is not, it adds 2 to `c`. If it is equal, it adds 1 to `c`. So overall, the original C code must look something like this:

```
int a = 2;
int b = 7;
int c = 0;
for (int d=a; d<b; d++) {
    if (d*d == a)
        c += 1;
    else
        c += 2;
}
```

8.4 Arrays and Function Calls

Arrays

In programming, arrays are data stored in a contiguous chunk of memory. If we compile something like `int a[] = {1, 2, 3, 4}`, it creates the following:

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x10],0x1
0x1134 <+11>:    mov    DWORD PTR [rbp-0xc],0x2
0x113b <+18>:    mov    DWORD PTR [rbp-0x8],0x3
0x1142 <+25>:    mov    DWORD PTR [rbp-0x4],0x4
```

Notice that it looks just like a bunch of variables. So under the hood, an array really isn't all that different than a bunch of variables. It's just that they are all next to each other in memory. Here they are all 4 bytes apart since the `int` data type takes up 4 bytes.

Let's now look at this C code.

```
int a[10];
for (int i=0; i<10; i++)
    a[i] = 5;
```

When we compile it, we get the following assembly:

```
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],0x0
0x1134 <+11>:    jmp    0x1147 <main+30>
0x1136 <+13>:    mov    eax,DWORD PTR [rbp-0x4]
0x1139 <+16>:    cdqe
0x113b <+18>:    mov    DWORD PTR [rbp+rax*4-0x30],0x5
0x1143 <+26>:    add    DWORD PTR [rbp-0x4],0x1
0x1147 <+30>:    cmp    DWORD PTR [rbp-0x4],0x9
0x114b <+34>:    jle    0x1136 <main+13>
```

The variable stored at `rbp-0x4` is the loop variable `i`. The instructions at lines +11, +26, +30, and +34 all correspond to the loop, similar to what we have seen before. The new parts are the inside of the loop from line +13 to line +18. The +13 line moves the loop variable into the `eax` register. In the next line, the `cdqe` instruction extends the value in `eax` into the full `rax` register. It has to do with the fact that the loop variable is stored as a 32-bit `int` in the C code, and the array operations require the full `rax` register. If we had used a 64-bit `long` for the loop variable, we wouldn't see the `cdqe`. The `cdqe` instruction is something that's needed at the assembly level, but if you're trying to read the assembly code, it doesn't actually translate into anything

meaningful at the C level, so you can mostly ignore it.

The most important line is below:

```
0x113b <+18>:    mov    DWORD PTR [rbp+rax*4-0x30],0x5
```

When you see something like this, you know an array is involved. It's helpful to think of the stuff in the square brackets as `[rbp-0x30 + rax*4]`. The location `rbp-0x30` is the base of the array, where the first item is stored. To it, we add `rax*4`, which is the index into the array. We are doing looking at `a[i]`, and remember that `i` is currently in the `eax`/`rax` register. We multiply by 4 because each integer takes up 4 bytes. So at index `i` (with `i` in `rax`), we are `4*rax` bytes from the base of the array.

Function calls

Below is a simple C program that calls a function:

```
int f(int a, int b, int c) {
    return a + b + c;
}

int main() {
    int x = f(1, 2, 3);
}
```

Here is what a C compiler generates for the `main` function:

```
0x1145 <+0>:     push   rbp
0x1146 <+1>:     mov    rbp,rsp
0x1149 <+4>:     sub    rsp,0x10
0x114d <+8>:     mov    edx,0x3
0x1152 <+13>:    mov    esi,0x2
0x1157 <+18>:    mov    edi,0x1
0x115c <+23>:    call   0x1129 <f>
0x1161 <+28>:    mov    DWORD PTR [rbp-0x4],eax
0x1164 <+31>:    mov    eax,0x0
0x1169 <+36>:    leave
0x116a <+37>:    ret
```

The `call` instruction calls the function. Right before it, we see the values 1, 2, and 3 being put into the `edi`, `esi`, and `edx` registers. In 64-bit assembly on Linux, the convention is that the first six integer or pointer parameters are put into `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. Anything more is pushed onto the stack using the `push` instruction (which we aren't covering here). On Windows, the conventions are a little different. There are also conventions for floating-point variables, which we won't cover here.

The other important convention is that the return value is in the `rax` register. We see that in line +28, where right after the function call, we store the `eax` value (the lower 32 bits of `rax`) into memory. This corresponds to the part of the C code that stores the result of the function call in the variable `x`.

Here is the code for the function itself:

```
0x1129 <+0>:     push   rbp
0x112a <+1>:     mov    rbp,rsp
0x112d <+4>:     mov    DWORD PTR [rbp-0x4],edi
0x1130 <+7>:     mov    DWORD PTR [rbp-0x8],esi
0x1133 <+10>:    mov    DWORD PTR [rbp-0xc],edx
0x1136 <+13>:    mov    edx,DWORD PTR [rbp-0x4]
0x1139 <+16>:    mov    eax,DWORD PTR [rbp-0x8]
0x113c <+19>:    add    edx,eax
0x113e <+21>:    mov    eax,DWORD PTR [rbp-0xc]
0x1141 <+24>:    add    eax,edx
0x1143 <+26>:    pop    rbp
0x1144 <+27>:    ret
```

We see that after setting up the stack, the first thing the code does is copy the register values into memory. This is taking the 1, 2, and 3 that the function passed to it and storing them in the `a`, `b`, and `c` parameter variables of the function. The assembly code then adds all the values and puts the final result in `eax` since that is where the return value is stored.[1]

8.5 Writing Assembly

In this section we will cover how to write some basic assembly, focusing on formulas, loops, and conditionals. One way to test out our code is to use an online tool. The one at https://onecompiler.com/assembly is pretty good. Whatever tool you are using, if you want to follow the examples here, make sure your tool is using 64-bit x86 assembly. A lot of sites use 32-bit x86, and the examples below won't work with those.

One issue with programming assembly is that it is so low-level that even printing things to the screen is painful. Printing a single character can be done in a few lines, but printing a number takes considerably longer. We have included some code in the appendix of these notes that you can use to print numbers to test out the code below.

To keep things simple, we will just be working with registers, not memory locations. We will be mainly working with `rax`, `rbx`, `rcx`, `rdx`, `rdi`, and `rsi`. If you need more, there are also `r8`, `r9`, ..., `r15`.

Simple arithmetic

Example 1 Here is a little assembly to add 2 and 3:

```
mov     rax, 2
add     rax, 3
```

The `mov` instruction stores 2 in `rax`, and then the `add` instruction adds 3 to that value and stores the result in `rax`. If you are using the template in the appendix, add the code below to print out the value in `rax` to make sure it comes out to 5.

```
mov     rdi, rax
call    print_int
call    print_newline
```

A few notes:

- Spacing and indentation for instructions doesn't matter. It's a good idea to space things nicely since assembly can be hard to read.
- You might be wondering why we didn't just do `add 2, 3`. The reason is that the `add` instruction needs somewhere to store the result, and it always stores the result in the first operand, so we need that to be a register or a memory address.

Example 2 Let's add $2+3+5$:

```
mov     rax, 2
add     rax, 3
add     rax, 5
```

This is a lot like the previous example. Note that we just keep adding to `rax`.

Example 3 Let's do $(2\cdot 3)+(4\cdot 5)$:

[1] Note that if we were writing this assembly ourselves, we could probably skip all the local variables and just do everything with registers. C compilers often by default will translate the C code very literally into assembly, which is what we see here.

```
    mov     rax, 2
    imul    rax, 3    ; 2*3

    mov     rbx, 4
    imul    rbx, 5    ; 4*5

    add     rax, rbx  ; add the two (stores in rax)
```

We first do the $2 \cdot 3$ part and store that in `rax`. Then we do the $4 \cdot 5$ part and store that in a different register, `rbx`. Then we add the two together and store it in `rax`. For the multiplication, we use `imul` instead of `mul` since `imul` is a little easier to work with.

Note the semicolon indicates a comment. Because assembly can be tricky to read, people tend to use a lot of comments.

Example 4 Let's do 42/5.

```
    mov     rax, 42
    mov     rdx, 0
    mov     rbx, 5
    idiv    rbx
```

Division is a little weird. First, note that the division instruction only takes one operand. The reason is that it assumes the numerator is in `rax`. The one operand is the denominator. We also need to zero out `rdx`.[1] The quotient ends up in `rax` and the remainder or mod ends up in `rdx`.

The reason division is weird is that the division operation was designed this way back in the 1970s, and x86 has kept backward compatibility with it all these years, so we are sort of stuck with it. Some newer assembly languages, like ARM, have much more reasonable division operations. Division is also one of the slowest operations. Addition and subtraction typically take a single cycle or less, multiplication takes a little longer, but division takes much longer, anywhere from 20 to 90 cycles. Compilers and assembly programmers often look for tricks to avoid division (such as using a bit shift to the right if dividing by a power of 2).

Loops

Things start to get a little trickier now. Below is a loop that adds up the numbers 1 to 10.

```
    mov     rcx, 0      ; summing variable x
    mov     rax, 1      ; loop variable i
loop_cond:
    cmp     rax, 10
    jg      end_loop
    add     rcx, rax    ; x = x + i
    add     rax, 1      ; i++
    jmp     loop_cond
end_loop:
```

The first thing to notice are the labels `loop_cond` and `end_loop`. These are names we make up. We can call them whatever we want, subject to similar rules to naming variables in programming languages. Note that labels must come at the beginning of the line, with no indentation. The labels are used for the jump statements. Earlier, when we were reading compiler-generated assembly, jumps went to specific line numbers. When you are writing your own assembly, jumps go to labels you create.

The first part of the loop is the comparison statement. We compare the loop variable, which is the register `rax`, to 10. If it is greater than 10, we jump out of the loop by jumping to the label we called `end_loop`. Otherwise, we run the code in the body of the loop, which adds to the total and then increments the loop variable. Finally,

[1] Actually, the numerator is a 128-bit value with the lower 64 bits in `rax` and the higher 64 bits in `rdx`. But if you not dealing with really huge values, you can just zero out `rdx`.

at the end of the loop, we jump back to the start of the loop. This jump back is where a loop gets its name from. We loop back to the start.

This is the basic structure of how to create a loop. Note that it is a bit backwards from the compiler-generated code we looked at earlier that always puts the loop condition at the end. That code can be a little more efficient, but it is also a little trickier to understand.

As a side note, x86 has an instruction `inc` that can be used to add 1 to a variable, so you can use `inc rax` instead of `add rax, 1`, if you like.

Conditionals

Example 1 below is a simple if statement that checks if the value in the `rax` register is less than or equal to 5. If it is, then it sets the `rbx` register to 5.

```
    cmp     rax, 5
    jg      end_if
    mov     rbx, 10
end_if:
```

This is the basic structure we can use to do a simple if statement. Notice that though we phrased the problem as "if `rax <= 5` then `rbx=10`, the actual code uses a `jg` instead of `jle`. We could use `jle`, but that would make the code a little messier. This code compares `rax` to 5 and if it is not less than or equal to 5 (i.e., is greater than 5), then it jumps over the part of code that sets `rbx` to 10. And if it doesn't jump, then it does the setting.

Example 2 Let's try an if/else type of condition. If the value in `rax` is less than or equal to 5, set `rbx` to 10 and otherwise set it to 20.

```
    cmp     rax, 5
    jle     le_code
    mov     rbx, 20         ; this is the else portion
    jmp     end_if
le_code:                    ; this is the if portion
    mov     rbx, 10
end_if:
```

One really important note is the `jmp`. Without it, the else portion of the code would set `rbx` to 20 and then keep going and overwrite it with 10. We need to make sure to jump over that part of the code to avoid this problem.

Example 3 Finally, let's try and if/else if/else condition. If `rax` is less than 5, then set `rbx` to 10. If `rax` equals 5, then set `rbx` to 15, and otherwise set it to 20.

```
    cmp     rax, 5
    jl      le_code
    je      equal_code

    mov     rbx, 20         ; this is the else portion
    jmp     end_if

le_code:                    ; this is the if portion
    mov     rbx, 10
    jmp     end_if

equal_code:                 ; this is the else if portion
    mov     rbx, 15
end_if:
```

This isn't the only way to write it, but it is one of the simpler ways. Note that after a comparison we can have

multiple jumps. The way the `cmp` operation works is it subtracts its operands and then bases its jumps on whether the result was negative, positive, or zero. These values (and a few others) are set in the `rflags` register, and the jumps all rely on them to make their decisions. As long as no other flag-setting code intervene, the jumps will reference the first comparison we made. Note again the `jmp` calls we have to jump over other blocks of code so that we don't accidentally run the code for multiple conditions.

Working with memory on the stack

If we want to store data in memory, instead of just using registers, we can use the program's stack. To do that, we first have to allocate some space on the stack. If you're familiar with the stack data structure, this stack works similarly. The two key operations are *push* and *pop*. The push operation pushes something onto the stack, and the pop operation pops off the stack the most recently added item (top of the stack). It's a little like a stack of dishes, where you add and remove dishes from the top of the stack.

The `rbp` and `rsp` registers are used for managing the stack. The `rsp` register is the "stack pointer" that tracks where the top of the stack is. The `rbp` is the "base pointer" that usually is used to store a previous location of the top of the stack. It is used as a reference point for locating variables we store on the stack. Typically, we set up an area of the stack to store variables by doing something like the following:

```
push rbp
mov rbp, rsp
sub rsp, 24
```

The first line saves the old value of `rbp` because we will be overwriting it. The second line stores the current stack pointer in `rbp`. The last line moves the stack pointer to make room for the variables. Here we have subtracted 24, which makes room for 24 bytes of memory. When we are done with the variables, we can "free up" the space we are using by the following two lines:

```
mov rsp, rbp
pop rbp
```

In between the setup and teardown, access our variables via a line such as the following:

```
mov qword [rbp-8], 4
```

This is a little different from the syntax we saw earlier when we were reading assembly. We just use `qword` and not `qword ptr`. Whether or not you need the `ptr` depends on the assembler you are using. The one are using does not use `ptr`. The `qword` is important here because it tells the operation how much memory to use. This line stores 64 bits (8 bytes), which is a quad word. If we were storing 32 bits, we would use `dword`, and we would use `byte` if we were just storing a single 8-bit byte's worth of data. You often need these specifiers when accessing memory like this, but if the size can be inferred by the operation, then it can be left off.

Below we put everything together to add two numbers that are stored in memory and store them in another memory location.

```
mov qword [rbp-8], 4
mov qword [rbp-16], 5

mov rax, [rbp-8]
add rax, [rbp-16]
mov qword [rbp-24], rax

mov     rdi, [rbp-24]
call    print_int
call    print_newline
```

When writing longer programs, it can be a pain trying to remember where everything is stored, so you can define names for things. For instance, we could use `%define total rbp-24` and then use `total` everywhere we have `rbp-0x24` in the code.

As a slightly more complicated example, here we use memory on the stack like an array and use it to store the first 20 Fibonacci numbers.

```
        push rbp
        mov rbp, rsp
        sub rsp, 160

        mov rax, 1
        mov rbx, 1
        mov rcx, 0

        mov qword [rbp-160], 1
        mov qword [rbp-152], 1

    loop_check:
        cmp rcx, 18
        jge loop_end

        mov rdx, rbx
        add rbx, rax
        mov rax, rdx
        mov [rbp + 8*rcx - 144], rbx

        add rcx, 1
        jmp loop_check

    loop_end:
        mov     rdi, [rbp - 88]
        call    print_int
        call    print_newline

        mov rsp, rbp
        pop rbp
```

The Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, . . . , where the first two are both 1 and each number thereafter is the sum of the previous two. For instance, the fourth number in the sequence, 3, is the sum of 1 and 2, the two numbers right before it in the sequence. After that, 5 is $2+3$, 8 is $3+5$, etc.

The code starts by making room for the 20 Fibonacci numbers on the stack. We allocated 160 bytes, which is 20 times 8 bytes for each integer. To keep things simple, since this is 64-bit assembly, we are just using 64-bit integers. We put the first two Fibonacci numbers into the registers `rax` and `rbx` as well as into memory at `rbp-160` and `rbp-152`.

Then we have a loop to generate the other 18 numbers. The two most recent Fibonacci numbers are in `rax` and `rbx`, and we use those to generate the next ones. The code moves the value in `rbx` into `rax`, and `rbx` gets the sum of the previous `rax` and `rbx`. We then store the newest value in memory. The line `mov [rbp + 8*rcx - 144]` does that by using the loop variable `rcx` as the "index" in memory, basically saying how far off from `rbp - 144` to store it. At the end, we print out the value at `rbp - 88`, which is the 10th number in the sequence.

We'll stop here with our brief tour of assembly language. The goal here was just to get enough exposure to it to understand how it works.

Appendix

Below is a template for testing out simple assembly. Toward the bottom is the place to insert your code. The code here was generated by ChatGPT since printing out a number in x86 assembly is kind of painful.

```
section .data
    newline db 10           ; Newline character

section .bss
    buffer  resb 20         ; Buffer for ASCII conversion (enough for 64-bit int)

section .text
    global _start

; ========================================
; Procedure: print_int
; Input: rdi = integer to print
; Modifies: rax, rbx, rcx, rdx, rsi, r8
; ========================================
print_int:
    push    rbp             ; Save base pointer
    mov     rbp, rsp        ; Set up stack frame
    push    rdi             ; Save the original number
    push    rax
    push    rbx
    push    rcx
    push    rdx
    push    rsi
    push    r8

    ; Handle negative numbers
    mov     r8, 0           ; Flag for negative (0 = positive)
    test    rdi, rdi        ; Check if negative
    jns     .positive       ; Jump if not negative
    neg     rdi             ; Make positive
    mov     r8, 1           ; Set negative flag

.positive:
    mov     rax, rdi        ; Number to convert
    mov     rcx, buffer+19  ; Point to end of buffer
    mov     rbx, 10         ; Divisor
    mov     byte [rcx], 0   ; Null terminator
    dec     rcx             ; Move back one position

    ; Handle special case of 0
    test    rax, rax
    jnz     .convert_loop
    mov     byte [rcx], '0'
    dec     rcx
```

```
    jmp     .done_convert

.convert_loop:
    xor     rdx, rdx        ; Clear remainder
    div     rbx             ; Divide by 10
    add     dl, '0'         ; Convert to ASCII
    mov     [rcx], dl       ; Store digit
    dec     rcx             ; Move back in buffer
    test    rax, rax        ; Check if done
    jnz     .convert_loop

.done_convert:
    ; Add minus sign if negative
    test    r8, r8          ; Check negative flag
    jz      .print
    mov     byte [rcx], '-'
    dec     rcx

.print:
    ; Print the string
    inc     rcx             ; Point to first character
    mov     rax, 1          ; sys_write
    mov     rdi, 1          ; stdout
    mov     rsi, rcx        ; String address
    mov     rdx, buffer+19  ; Calculate length
    sub     rdx, rcx
    syscall

    pop     r8
    pop     rsi
    pop     rdx
    pop     rcx
    pop     rbx
    pop     rax
    pop     rdi             ; Restore original number
    pop     rbp             ; Restore base pointer
    ret                     ; Return to caller

; ========================================
; Procedure: print_newline
; Prints a newline character
; ========================================
print_newline:
    push    rax
    push    rdi
    push    rsi
    push    rdx
    push    rcx

    mov     rax, 1          ; sys_write
    mov     rdi, 1          ; stdout
    mov     rsi, newline    ; Address of newline
    mov     rdx, 1          ; Length
    syscall

    pop     rcx
    pop     rdx
    pop     rsi
    pop     rdi
    pop     rax
```

```
    ret

; ========================================
; Main program
; ========================================
_start:
    ; Put your code here.  Whatever you want to print goes in rdi.

    mov     rdi, 15
    call    print_int
    call    print_newline


    ; Exit program
    mov     rax, 60         ; sys_exit
    xor     rdi, rdi        ; Exit code 0
    syscall
```