

# Primer for Intro to Programming Courses at Foothill

Anand Venkataraman

© 2016 Computer Science Department, Foothill College, Los Altos, CA.

This document is free to share unmodified for educational purposes. Please try to be green and not print it - peruse it electronically as much as possible. It's also available for download at [my departmental home page](https://fgamedia.org/faculty/anand/references/Module0-InformationandDataRepresentation.pdf):

<https://fgamedia.org/faculty/anand/references/Module0-InformationandDataRepresentation.pdf>

Regardless of whether you're in my section or not, you'll benefit from reading this document before you embark on Computer Programming courses at Foothill. Even though we'll review this content in my intro to programming classes, you should strive to master all of this before you begin. It's quite simple actually, and helps to establish a firm baseline off which I can teach. Further, if you're enrolled in one of my sections, this document gives you a taste of my (informal) teaching style during lectures (digressions and all).

Ideally, you have already read this before the first lecture (f2f students) or before you begin reading any of the remaining modular material (online students).

## Data is King

Gordon Gecko said it best in the movie Wall Street: Information is the most valuable commodity. As humans we process information all the time. Using computers, we can process certain kinds of information a whole lot faster today. One important aspect of Computer Science is the study of how to create programs that can process



information to solve interesting problems. So let's start at the very beginning. I want to give you a quick primer on *Information* before we start looking at how we can manipulate it using machines.

You can quantify many commodities. For example, you could say you bought a hundred grams of gold, or 2 gallons of gas. But can you quantify information? Can you say something like, "I'd like to buy X units of information?" How would you do it if you can? Think about how you might pay for tips at a race course.

In this module we will use the words *information* and *data* interchangeably, although there is a subtle and crucial difference between the two (What is it? Could you consider data to be *encoded information*?)

To process information, we need to:

1. Know (or rather, have an agreement on) how to represent it
2. Store the data using the above representation(s)
3. Manipulate (i.e. change) the data

(2) and (3) are straightforward once we know (1). In fact, this course is mostly concerned with (3). So the bulk of this module will focus on how we represent information and store data.

## How many letters do you need in a language's alphabet?

Once I was at a dinner where I was party to an interesting conversation that went like this (names have been changed):



**Marcus:** I already know English, Spanish and Italian. I want to learn many more languages, especially the non-European ones. Wouldn't it be easy if they all used the Roman script? That way a big hurdle in learning these languages would be removed.

**Moshe** (*interjects*): That's ridiculous and impossible. There are so many sounds in my native language that English doesn't even have letters for.

**Jiang** (*nodding her head*): I agree, and in my language you'll even find tones. Seven of them. I can't even begin to imagine how you can use the Roman script for those.

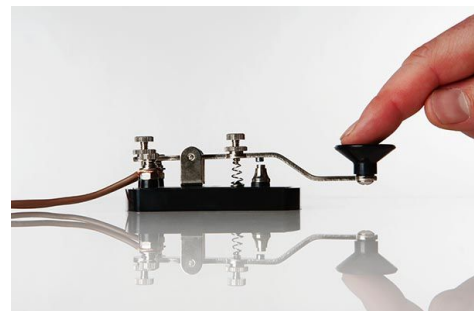
**Efua**: (*also nodding*): Yeah, and clicks too. (Clicks his tongue as if to demonstrate a sound that can't be transcribed using the Roman alphabet).

**Marcus**: Well, I don't see any problems. If we all agree on a way to transcribe these things in a particular way, I'm sure it could be done using the English alphabet.

Who is right? Would the English alphabet suffice to transcribe every other spoken language in the world? What about every other language in the whole entire infinite universe? Some, like Moshe, may object saying "Well, there are only 26 letters in English, but there are so many more sounds in language X, not to mention tones and clicks (and eye-rolls in teenspeak). Clearly the Roman alphabet with its paltry 26 letters is insufficient for the purpose."

But if you think about it, 26 is plenty. Plenty more than we actually need. In fact we only ever need two letters in an alphabet to transcribe anything as long as there is common ground and shared experience between two communicating parties on what is denoted by some *particular sequence* of English letters.

Conversely, the existence of a common script, no matter how big its alphabet, does nothing to



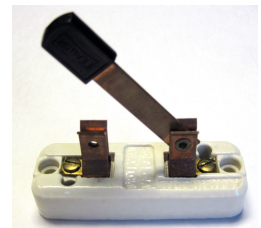
solve the problem of representing/communicating information between parties that have no common experiential ground. For instance, you can never successfully communicate the color red to a person who has been severely color blind from birth even though you may both converse quite fluently in English!

So that's the key to successful data representation. Rather than use single symbols to communicate a datum, we always use some combination of symbols.

Look up Morse Code, which used to be quite common in sending Telegrams in days of yore! How many symbols does Morse Code use?

We only use two symbols in our alphabet in the study of most of today's digital machines. This is because these symbols lend themselves to easy representation in an electrical circuit.

The key idea here is that of a switch. A switch can be either in an on state or an off state (no middle state allowed) - Either current is flowing through a wire or it is not. Both of these situations are easily observable and recordable.



Thus we denote the on state by the symbol 1 and the off state by the symbol 0 - the only other symbol in our CS alphabet.

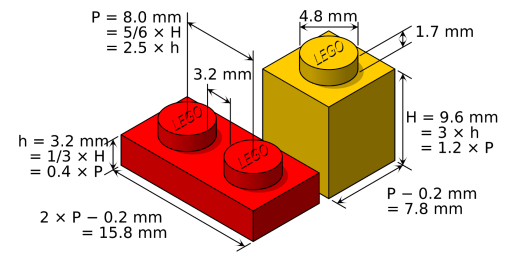
Everything else we see, hear, change or record will be made of some combination of 1s and 0s.

We call these basic symbols *bits* which stands for **B**inary **I**tems.

Wouldn't it be funny if the early Computer Scientists had instead created a meaningful 3-way switch (Three possible stable states) and used that as the basis of information representation? I wonder what programmers would be thinking of all day long if that were the case... Hmmm!

Bits are like the bare-bones lego blocks out of which you can build all other rather complex shapes. But it gets cumbersome and tiring pretty quickly if you always

have to build everything from the absolute bare-bones primitives, yes? So what we'll do is make it slightly easier on ourselves by manufacturing some slightly more sophisticated *primitives* from these bits so that whenever necessary we can build our things from these larger building blocks. The analogy to keep in mind here is that even though you could technically build an entire city out of atoms, we choose to build it using bricks and mortar, while being mindful that our bricks and mortar are time-saving constructs that were mass produced for us from the atoms.



Now for the purposes of this course, we'll assume we want to represent, store and manipulate the following kinds of primitive data.

1. Letters of an alphabet (e.g. English)
2. Numeric quantities (integers and fractional quantities)
3. Logical data (true/false)
4. Are there any others you can think of?



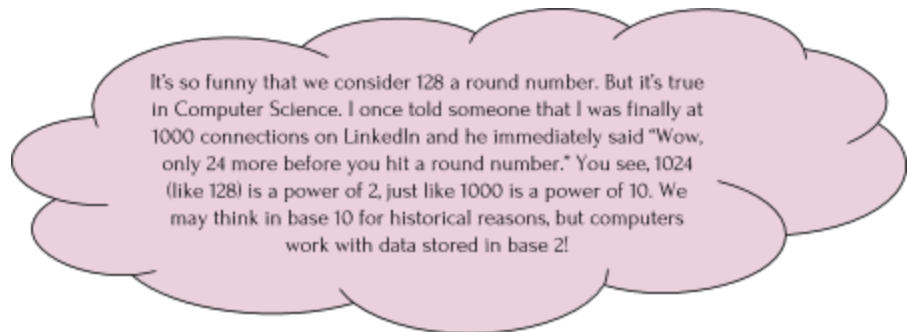
For historical reasons, we group our switches (bits) in 8-packs. These 8 bit packs are collectively referred to as *bytes*. (Sometimes we need to deal with half a byte, which is commonly referred to as a *nybble* - Yes, Computer Scientists do have a wonderfully quirky sense of humor. Seek them out at parties.)

At this point, ask yourself how many distinct bit sequences you could possibly have in one byte. To guide your thinking, you may want to first ask yourself how many bit sequences you could have in just one bit (duh... 2, right?) Then ask

yourself how many using 2 bits, 3 bits and so on until 8.<sup>1</sup> The answer, which you have hopefully arrived at is  $2^8$  which is 256. Go on... Enumerate it and convince yourself. I always say "*When in doubt... Try it out.*" Now there's a nice rhyme for you to memorize! We'll come back to this ditty many times during this course.

Ah... You're back. Good. Let's now think of how to represent our higher level data. There's 26 letters in English. Considering both uppercase and lowercase, that's 52. Throw in all the punctuation symbols you can think of, and add symbols for the various Hindu-Arabic numerals 0-9, we still end up at fewer than about a 100. Let's round things off and say 128.

We can get 128 or  $2^7$  distinct bit patterns using 7 bits, but since our byte is made of 8 bits, we throw in an additional bit and say that we'll represent all



the letters of the English Alphabet (and more, like box-drawing characters, arrows, selected math symbols, etc.) using various 8-bit sequences of 1s and 0s. Such a code was once designed and adopted as a standard (just like Morse code). Those who were party to this convention called it the American Standard Code for Information Interchange, or ASCII for short. [Here is a full list of the ASCII codes](#) for various printable and non-printable characters on Wikipedia.

So the next time someone at a party asks you "What is ASCII?", you won't have to respond "ASCII stupid question GETTY stupid answer".

---

<sup>1</sup> Cultivating this way of thinking about problems is especially useful in Computer Science where you will need to find solutions to large problems by solving them in the small first, a key skill to acquire as you go on to design Recursive or Dynamic Programming algorithms later.

## Representing Numbers

Now that we know how to represent single characters (A-Z, a-z, punctuations, numerals, etc.), you may be thinking that's all you need to manipulate information. For example, you could store numbers in ASCII form using individual ASCII characters for each of the digits (and the minus sign and radix too). However, such a representation is kludgy and doesn't lend itself to easy manipulation. Suppose we want to add two numbers, for instance, and these numbers were stored as ASCII letters, you'll find that the addition algorithm is rather convoluted and messy. Even worse when you want to do more sophisticated things like multiply and divide.

What we want, ideally, is some way of representing numeric quantities separately from their ASCII representations that lends itself easily to standard arithmetic operations. For example, we want to apply our trusted techniques that use carries and borrows for addition and subtraction. How do we do this? Before you read ahead to see what we actually do, I'd like you stop at this point and think for a few minutes (or an hour if needed) about a strategy that might work.



Whirrrr... That's the sound of your mental gears crunching ...

## Binary Number System

Ok, here's how we do it. See if it is similar to the strategy you came up with. The key insight to storing numbers is really quite similar to that for storing text. It all boils down to the fact that instead of the ten different numeric symbols we are accustomed to (the digits), we have to make do with only 2, with one important



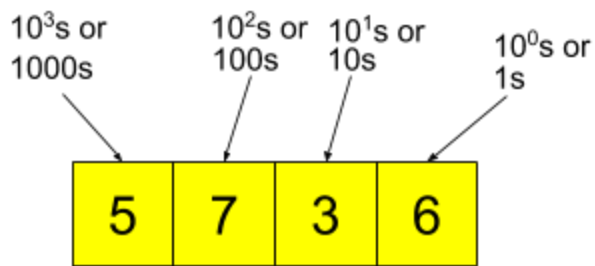
caveat - we'll continue to use that incredibly ingenious technique called the place value system even in our reduced base (i.e. base-2 instead of base-10). You can think of this as the system of numbers invented by aliens who only have one finger on each hand.

Believe it or not, 99% of the work in teaching you how this binary number system works has already been done for you by your teachers in elementary school. Yes! You read that right. Today you look at numbers and use the place value system unthinkingly as though it was second nature to you. But in reality, a great deal of subtle transformations had to happen in your brain for you to truly get it. You have your 2nd and 3rd grade teachers to thank for it. The marvelous thing about it is that once that knowledge is in place, you essentially have everything it takes to understand numbers in any base, whatsoever.

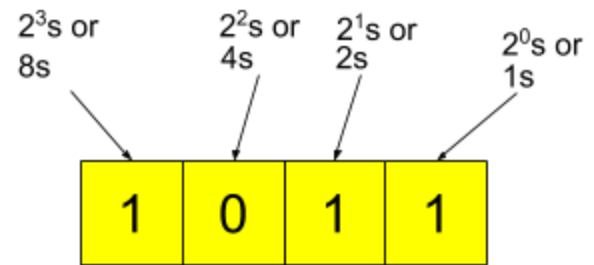
In fact, whenever you're at a loss to manipulate numbers in a base you're not used to, always think back to decimal, and think of what you'd do when faced with an identical problem in base-10. And then, hey presto! You'll have your answer.

So let's first take a new look at decimal numbers. The number that follows 9 is 10. The way we've been taught to interpret that "10" is that there is one (the left-most digit) ten and 0 ones. Likewise, a decimal number written as "5736" tells us it's a quantity in which there are 5 thousands, 7 hundreds, 3 tens and 6 ones. The procedure to convert a number to a quantity is identical in binary. See the picture below:

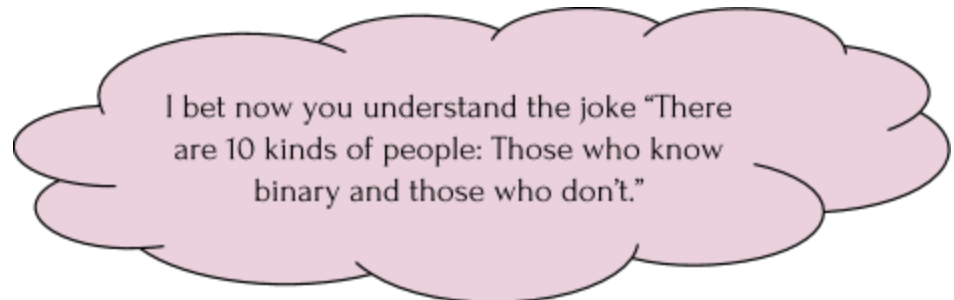




Interpreting a Decimal Number

Interpreting a Binary Number ( $8+2+1 = 11$ )

Since we don't have the luxury of using 9 different symbols in base-2, the symbol that follows 1 is not 2, but rather 10, whose value is the quantity two. And instead of using place to denote the number of powers of 10, we use place to denote the number of powers of 2.



So you know how to convert from binary to a quantity (which you conveniently represent using our friendly base-10). How do you convert the other way around? How do you figure out what the binary representation of a quantity, say seventy three, is?

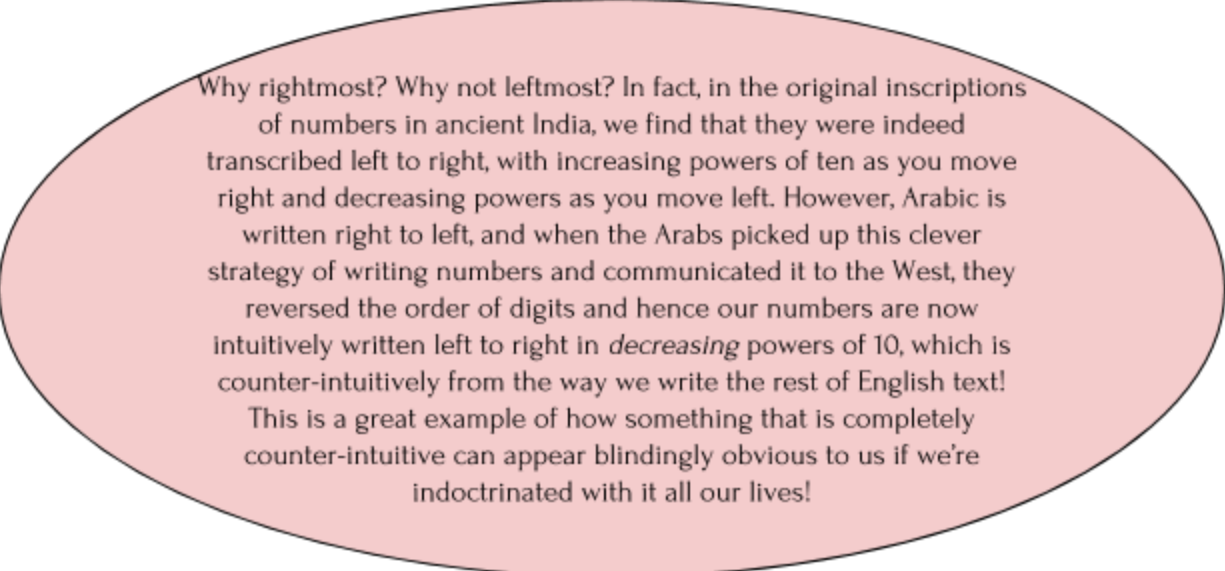
## From Decimal to Binary

We're going to use the advice I gave you before. When faced with a problem in binary, think back to how you would do it in decimal. Then the 2-finger alien counterpart of your elementary school teacher will guide you along the way. Pay special attention and understand this clearly because this is identical to the way you'd handle any base, be it 8, 16, or 27!

How do we convert a quantity such as seventy three into its decimal representation? We do it this way. First figure out the number of ones in it. We do

this by taking out as many tens as we can, and when we can't take out any more tens, the remainder is the number of ones - yes? Mathematically, we could state it as follows:

The units (ones) digit of a quantity is the remainder after you divide the quantity by 10. Since seventy three divided by 10 gives you a remainder of 3, it follows that the one's digit is 3. This is the rightmost digit of our decimal number.



Why rightmost? Why not leftmost? In fact, in the original inscriptions of numbers in ancient India, we find that they were indeed transcribed left to right, with increasing powers of ten as you move right and decreasing powers as you move left. However, Arabic is written right to left, and when the Arabs picked up this clever strategy of writing numbers and communicated it to the West, they reversed the order of digits and hence our numbers are now intuitively written left to right in *decreasing* powers of 10, which is counter-intuitively from the way we write the rest of English text! This is a great example of how something that is completely counter-intuitive can appear blindingly obvious to us if we're indoctrinated with it all our lives!

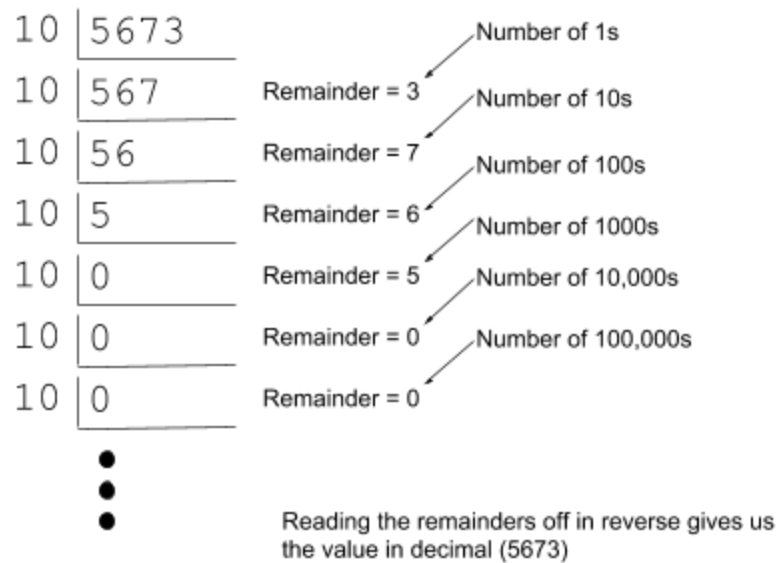
Once that's done, take the quotient (That's how many times 10 divides into the number reduced the remainder). If you divide that quantity by 10 again, that gives you the number of 100s in the quantity.

If we repeat this process until we get to a zero (and beyond), the remainders at each step tell you exactly the number of ones, tens, hundreds, etc. were used in making up your quantity. String these remainders together in the order in which you're used to making up your number and you have it.

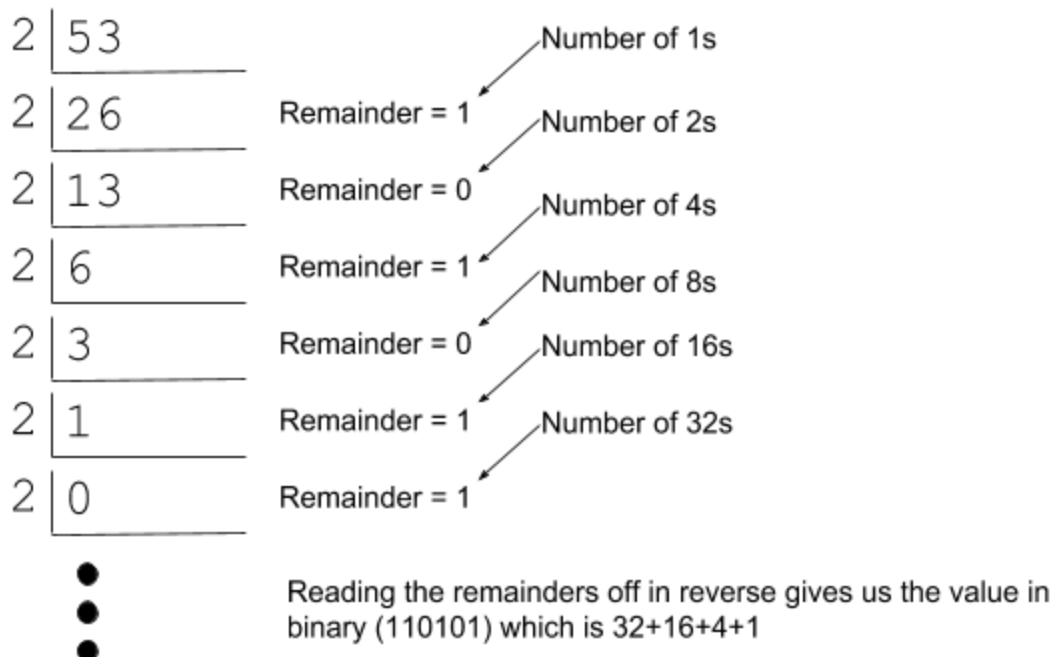
Let's look at a concrete example to nail down the concept. We're going to convert the quantity fifty six hundred seventy three into its decimal representation -

which kinda sounds silly, but it's instructive nonetheless because it gives you the exact procedure you're going to use for any other base.

Here's how it all comes down:



Now that you have an understanding of how that works, you'll see that the procedure is identical for binary (base-2), or indeed for any other base. Let's convert the quantity fifty three into binary this way.



Now in the case of binary only, you can exploit a clever fact. If you divide a number by 2 and have a remainder of 1, it means that the number is odd. otherwise, the number is even (a fact you'll leverage in some of your homework assignments in this course). This means that there is a short-cut to the above (tedious) long-division process. You could progressively halve the given quantity ignoring remainders. Then you scan all the quotients making note of which ones are even and which are odd. You write a 1 below the odd quotients and a 0 below the even ones and you have your resultant binary number. Below, we write down the 53 on the right, and keep halving it as we move to the left. Then we place a 1 below each odd quotient and a zero below each even quotient.

0	0	1	3	6	13	26	53
0	0	1	1	0	1	0	1

Our number is 110101. Of course, as shown above, you can add any number of zeroes in front (...000110101) without changing the quantity.

[This nice video](#) demonstrates how to do it.

## Octal Numbers

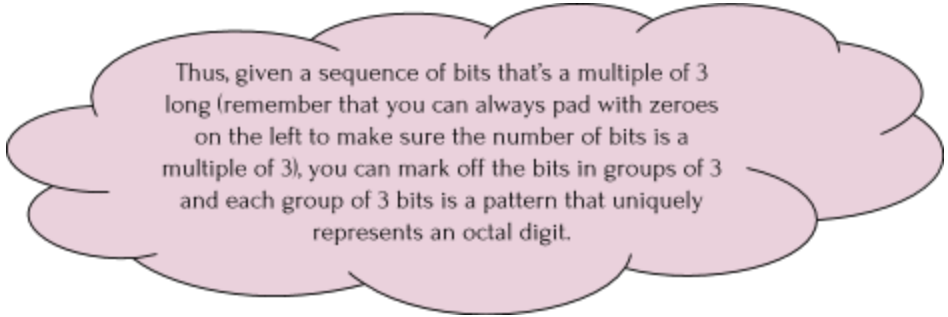
It's useful to know about octal (or base-8) numbers not only because it further solidifies your understanding of numeric bases, but also because you may (albeit infrequently) see

octal notation in  
computer programs.

Octal numbers are  
useful because each  
octal number

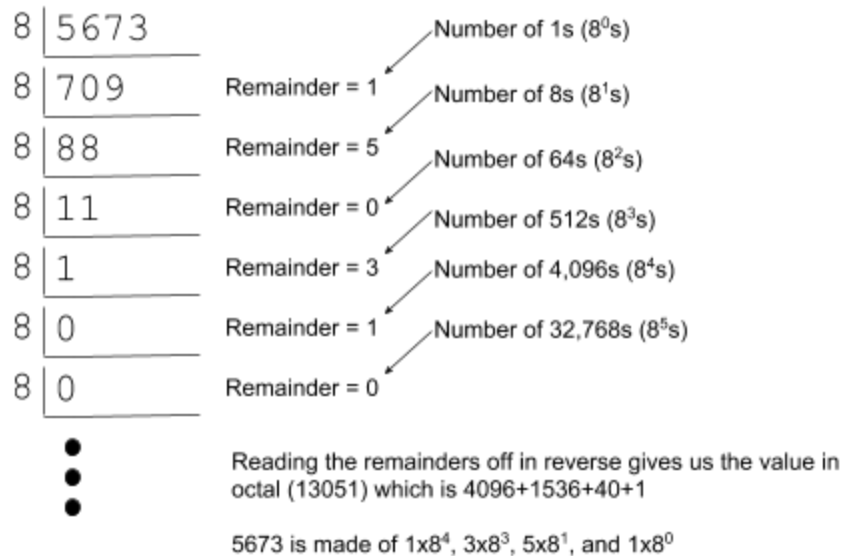
corresponds directly

with one of the 8 possible bit patterns you can generate from 3 bits.

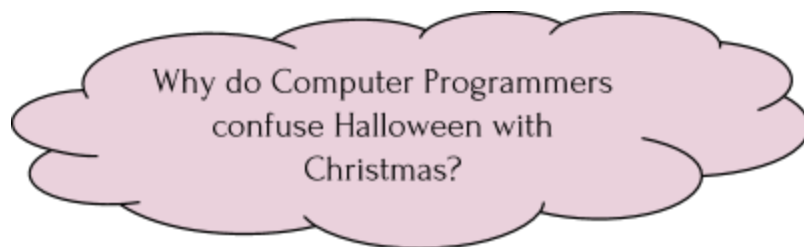


Thus, given a sequence of bits that's a multiple of 3 long (remember that you can always pad with zeroes on the left to make sure the number of bits is a multiple of 3), you can mark off the bits in groups of 3 and each group of 3 bits is a pattern that uniquely represents an octal digit.

Let's try the same tack with the number quantity fifty six hundred seventy three, and convert it to base-8 (or octal):



Conversely, if I gave you an octal number and asked you what quantity it



represents you'd do the reverse of the above procedure to discover the quantity. For example, to convert 347 octal into a

*grokkable* quantity, you'd say that it has 7 ones, 4 eights and 3 sixty fours, giving  $192 + 32 + 7 = 231$  in decimal.

## Hexadecimal Numbers

Like octal numbers, hexadecimal numbers (base 16) are also special because each hexadecimal digit corresponds to a unique bit pattern in a set of 4 bits.

However, 4 bits is exactly half a byte (which is sometimes called a nybble). And so you see it used a lot more frequently in computer code than octal numbers.

The way to tell if a numeric literal (a raw number) is hexadecimal is to see if it is prefixed with the letters 0x or 0X. If it is, then the number is to be interpreted as hexadecimal.

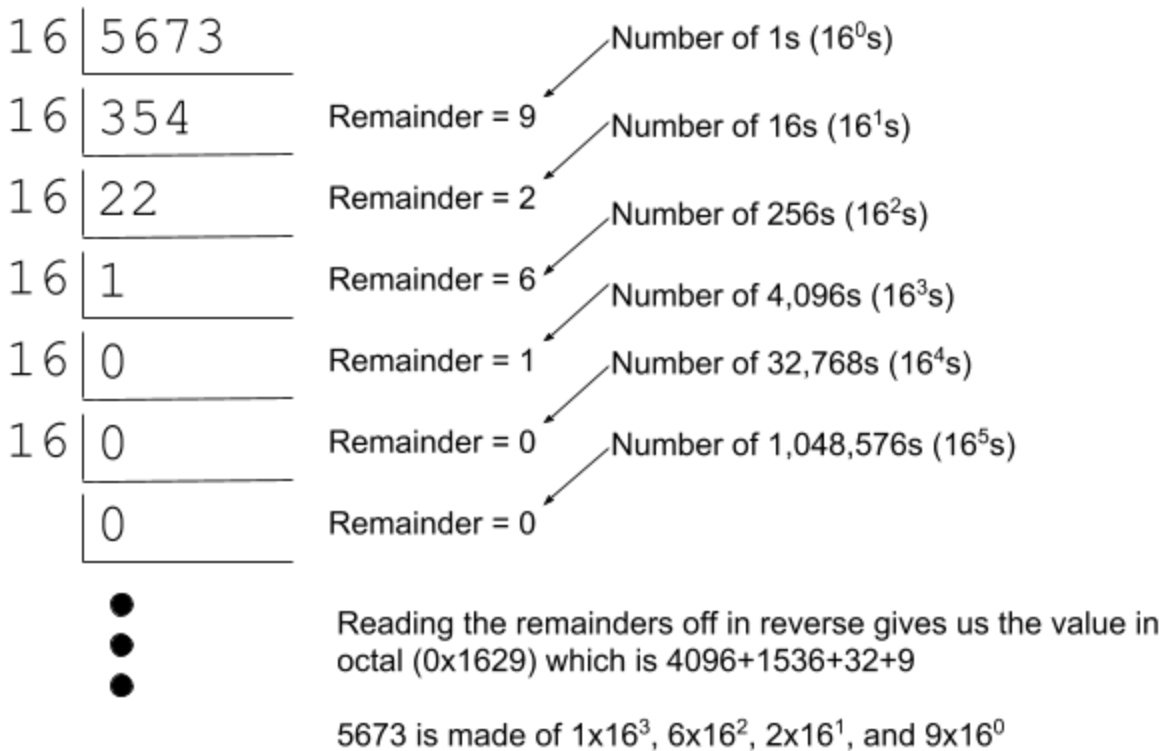
Another point of difference between hexadecimal and octal numbers is that while we were able to make use of our traditional Hindu-Arabic numerals for octals (since we only needed 8 symbols and we had 10), we have to manufacture 6 new symbols for hexadecimal notation. While we could use any symbols of our liking for the hexadecimal digits corresponding to the quantities 10 through 15 (remember - 16 will be written as 10 in hexadecimal), we chose to go with the option of borrowing from the English alphabet.

Here are possible candidates we might have used instead:

Quantity in Decimal	Hex Proposal 1	Hex Proposal 2	Hex Proposal 3 WINNER!
0	0	<i>0000</i>	0
1	1	<i>0001</i>	1
2	2	<i>0010</i>	2
3	3	<i>0011</i>	3
4	4	<i>0100</i>	4
5	5	<i>0101</i>	5
6	6	<i>0110</i>	6
7	7	<i>0111</i>	7
8	8	<i>1000</i>	8
9	9	<i>1001</i>	9
10	@	<i>1010</i>	A
11	\	<i>1011</i>	B
12	^	<i>1100</i>	C
13	△	<i>1101</i>	D
14	□	<i>1110</i>	E
15	◇	<i>1111</i>	F

Now that we're armed with a full inventory of symbols for our hexadecimal digits, we can use the same process as before to convert a quantity into hex.

Let's try that for the above number, fifty six hundred seventy three:



That's it. Now it's a real pity that none of the remainders in the conversion process above was above 9. Then we'd have been able to use our cool new hex digits for them. Can you think of a number for which you get these special hex-only digits in some of the cases? How about Ramanujan's number, 1729?

## Hexadecimal Nybbles and Bytes

In column 3 of the table of possible proposals for hexadecimal symbols, you saw that I listed a bunch of patterns made of strokes and ovals. That was intentional. You see, if you substitute the strokes by 1s and ovals by 0s, you effectively get the full set of bit patterns you can make using 4 bits (or a nybble). Thus you



could effectively mark off a byte into two half bytes of 4 bits each, and consider a byte to be a two digit hexadecimal number.

You better get used to thinking of a byte in this way because you'll be using this knowledge, especially if you advance into the B level of this course (CS2B or CS1B) and start performing bitwise manipulations. Furthermore, whenever a system prints out an internal quantity like a memory address, you'll notice that it prints it out in hexadecimal form (not that you'd ever have to decode it manually :-)

One cool aspect of this situation is that it lets you convert from binary to hexadecimal and vice-versa quite efficiently and quickly, without having to go through the tedious long-division or exponentiation and addition process. For example, to convert the value 0x02C9 into binary, simply convert each hexadecimal digit into its corresponding nybble. 2 is 0010, C (which is 12 decimal) is 1100, and 9 is 1001. So 0x02C9 must be 0000001011001001 in binary. Omitting the leading zeroes you could write it as 1011001001. How cool is that?



Now here is a nice challenge for you. Convert the decimal number 3,735,928,559 from decimal to hexadecimal and marinade in a bit of juicy CS history while pondering the result.

### Food for thought and another challenge: Base-27 anyone?

What is the quantity (in decimal) denoted by your first name(s) if you think of your name as a number in base-27 where the letters of the English alphabet represent your base-27 digits. That is, a space ( ' ') = 0, A = 1, B = 2, C = 3, ..., Z = 26.



This is how I'd calculate my special number. Since the most common English transcription of my name is ANAND, (A = 1, N = 14 and D = 4, the decimal equivalent of my special number

$$\begin{aligned}
 &= 4 \times 27^0 + 14 \times 27^1 + 27^2 + 14 \times 27^3 + 27^4 \\
 &= 4 + 378 + 729 + 275562 + 531441 \\
 &= 808,114
 \end{aligned}$$

It would be 1100,0101,0100,1011,0010 in binary

or C54B2 in hexadecimal

or 3052262 in octal

...

Calculate your own special number and post it in the discussion forums for a lively exchange with your classmates. It might be pretty large when converted to decimal, so you may want to look at its hexadecimal equivalent. And if you're lucky, maybe you can even pronounce it!

## Arithmetic in Binary

Now we're set up to do what we promised when we said we need a non-ASCII representation for quantities. We'd like to perform basic arithmetic operations like addition and subtraction on binary numbers. How?

Again your 3rd grade teacher springs to your rescue. You can use the exact same intuitions and algorithms for binary numbers as you've learned for decimal numbers long ago! Thus it helps to quickly review how you'd add two decimal numbers:

$$\begin{array}{r}
 \begin{array}{cccc}
 & & & \text{Carried over} \\
 & \swarrow & \swarrow & \swarrow \\
 1 & 7 & 2 & 9 \\
 + & 2 & 8 & 7 \\
 \hline
 2 & 0 & 1 & 6
 \end{array}
 \end{array}$$

Our tried and trusted addition algorithm for numbers  
In decimal notation

And here's how we do it in binary. The only difference is that we need to keep in mind that the largest single digit in binary is 1, not 9, and hence  $1+1$  in binary gives you 10, resulting in a 1 that carries over. Also keep in mind that  $1+1+1 = 11$  in binary.

$$\begin{array}{r}
 \begin{array}{cccc}
 & & & \text{Carried over} \\
 & \swarrow & \swarrow & \swarrow \\
 1 & 1 & 0 & 1 \\
 + & 1 & 1 & 1 \\
 \hline
 1 & 0 & 1 & 0 & 0
 \end{array}
 \begin{array}{l}
 = 13 \\
 = 7 \\
 \\
 = 20 \quad (\text{verify this by converting } 10100 \text{ binary})
 \end{array}
 \end{array}$$

Our tried and trusted addition algorithm for decimal  
numbers *ported* over to binary.

I'm not going to cover subtraction, multiplication or division here, but I urge you to think about how you'd do it. In particular, verify that the subtraction algorithm you're familiar with for decimal numbers (using borrows instead of carries) works equally well for binary numbers.

Do it now!



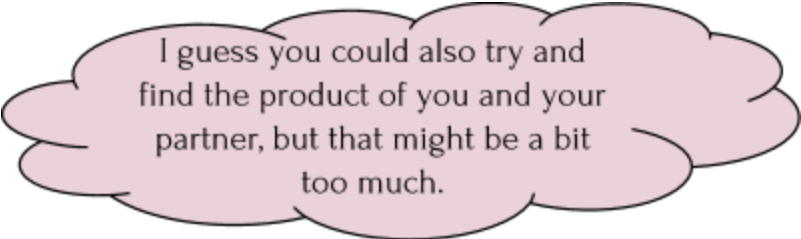
Why stop here? At this point you're equipped with enough knowledge to verify that these techniques work no matter what the base. Experiment with various numbers in different bases and make sure they work no matter what. Check that you can apply this algorithm to add, subtract or multiply numbers in octal or hexadecimal, why, even base-27 notation.



Here's yet another cool challenge problem for you.

I'd like you to partner up with someone else at home or in class and find what you get when you add yourself to your partner or subtract your partner from yourself (or vice-versa). To do this,

imagine, like before, that your (and your partner's) numeric equivalents are your first names in base-27 using the convention we discussed before (That is, a space ' ' = 0, A = 1, B = 2, ... Z = 26). You must, of course, also convert the result back to base-27.



I guess you could also try and find the product of you and your partner, but that might be a bit too much.

### Special Cases in Multiplying and Dividing Numbers

By now, you've probably experimented with your trusted multiplication and division algorithms and convinced yourself that they work just as well regardless of the base in which you operate. Kudos! But before we leave the topic of arithmetic in non-decimal bases, I want to touch upon two special cases because we'll come back to it later on as we learn more advanced concepts in programming. These are to do with multiplication and division of a number by its base. What do I mean by this? I mean multiplication (or division) of a number in decimal notation by 10, or an octal number by 8, a binary number by 2, or a base-27 number by 27, and so on.

Keep in mind that we haven't covered representation of fractional quantities yet, and so any remainder after a division is simply discarded. For example,  $3141/10 = 314$ , not  $314.1$  - Note that by *discard* I don't mean *round up or down*. I mean *truncate*. Thus  $314159/10$  ought to be  $31415$ , not  $31416$ .

Experiment with a few numbers in various bases to find out what happens.

Do it now!



What you would have found by now, hopefully, is that multiplying a number by its base has the net effect of shifting all the digits left by one position. Likewise, dividing by its base has the effect of shifting all the digits right by one position. The effect becomes more obvious if you consider numbers to be written into a fixed number of boxes, like the way you fill in your SSN into your tax returns. An example in base-10 is  $31415926 \times 10 = 314159260$ .

0	0	3	1	4	1	5	9	2	6
---	---	---	---	---	---	---	---	---	---

 $\times 10 =$ 

0	3	1	4	1	5	9	2	6	0
---	---	---	---	---	---	---	---	---	---

Likewise, in binary, we get

0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

 $\times 2 =$ 

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

$$13 \times 2 = 26 \text{ (Verify this)}$$

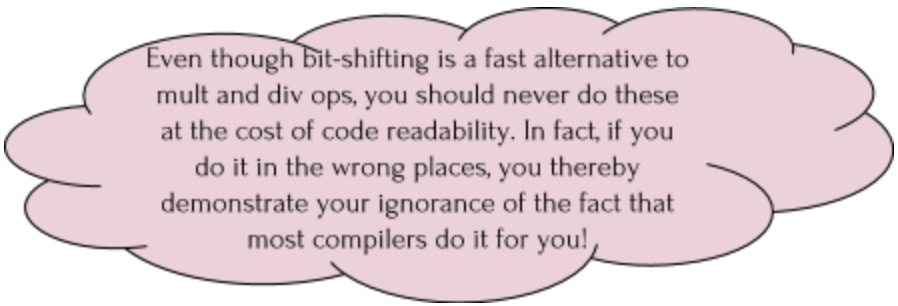
0	0	0	0	1	1	0	1
---	---	---	---	---	---	---	---

 $\times 4 =$ 

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

$$13 \times 2^2 = 52 \text{ (Verify this)}$$

This gives us two very important bitwise operations implemented as deeply as in the hardware of all computers. These are called the bitwise left-shift and right-shift operations. These are engineered to be so fast that many compilers identify multiplication and division by bases (2) and automatically transform these into shift operations for you so your program will run faster.



Even though bit-shifting is a fast alternative to mult and div ops, you should never do these at the cost of code readability. In fact, if you do it in the wrong places, you thereby demonstrate your ignorance of the fact that most compilers do it for you!

## Representing Negative Numbers

It's all well and good that we can represent quantities using bits and perform basic arithmetic on them. But we realize that numbers can be signed. That is, they could be negative. How does one go about representing a negative quantity using bits? After all, a bit can exist in only two states - 1 and 0. No intermediate state is possible, so how does one represent the minus sign we're so used to seeing before negative numbers? How would you store a -10, for instance, in a byte? Again, think about this and come up with a strategy of your own before reading on.



Whirrr... That's the sound of your mental gears crunching.

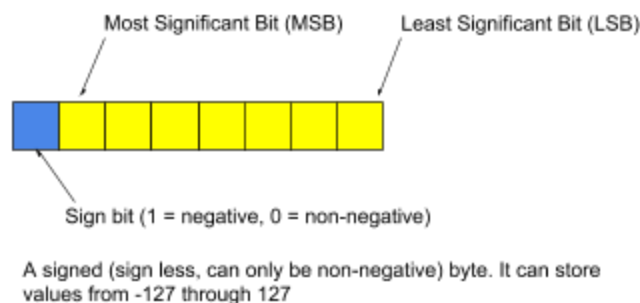
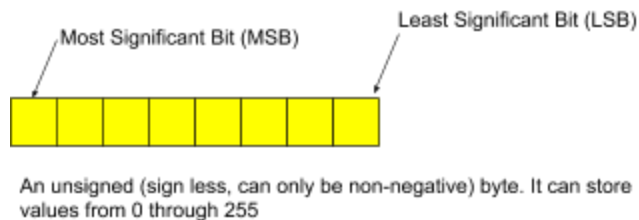
### Sign Magnitude Notation for Negative Numbers

Here's how we do it: Or more accurately, here's one (not the best) way to do it: We realize that a byte has 8 bits, and agree to use its left-most (or right-most) bit as the signature of a number stored in that byte. If the sign bit (signature) is

1, we take it that the represented quantity is negative. If it's 0, then it's non-negative.

The compromise here is that we've sacrificed a whole bit for the ability to represent negative numbers. This only leaves the 7 other bits to store the actual value of the number (its magnitude), which implies that the maximum value that can be stored in a signed byte is  $2^7-1$  which is only 127.

Note that 7 bits still gives us 128 distinct bit patterns. It's just that we use one of those to represent 0, leaving 127 patterns for the positive integers).



This way of storing numbers is called the Sign-Magnitude notation for the simple reason we store the signature and the magnitude in two explicitly distinct locations.

There is an obvious drawback with this technique. Can you think of what it is? There's a redundancy in that a certain number can be stored in two different ways. In other words, we end up wasting one bit pattern by making it the same value represented by another bit pattern - What value is this?



Chances are you guessed right - both 10000000 and 00000000 represent 0, since  $-0 = 0$ . Ideally, we'd like to make efficient use of the space of possible patterns available to us. And besides, arithmetic with numbers stored in sign magnitude notation is slightly more cumbersome than with another technique which we will now discuss.

## Two's Complement Notation

As with all life problems it always helps to step back from time to time, ask ourselves "Why?" and review what we hope to achieve from a particular endeavour. If we do that here, we find ourselves going back to question our basic notions of what makes a number negative in the first place. What is it that characterizes negativity? (In numbers, of course, not people :-)

I can short-circuit a potentially wrong track to answering that question by saying it's not the dash we put in front of negative numbers. That's just a notation. We're asking what the meaning of numeric negativity is.

One good answer to this question is that a negative number is any number that when added to its positive counterpart yields a zero.

Ponder that for a moment before moving forward...



So now we can ask a better question. What binary number can we add to a given binary number to get a zero? Remember, we're asking this question before we even know how to represent negative numbers, and hence all bit patterns are, in a sense, positive numbers up till now.



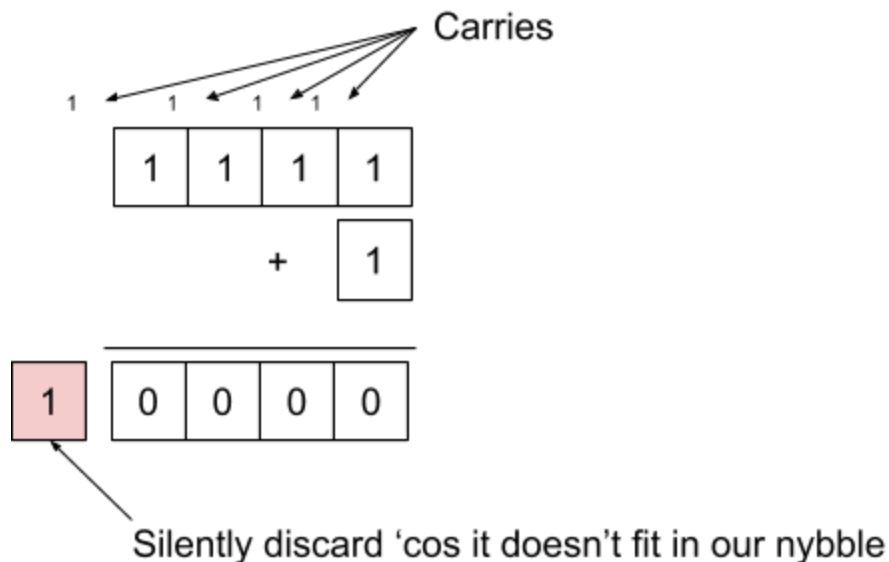
Clearly, no positive number yields a zero when added to another positive number. The problem, as just posed, appears futile. But herein lies a nice example of the resourcefulness and cleverness of Computer Scientists. The solution to this problem leverages not only what we know about the arithmetic of numbers, but also certain artifacts introduced by our very specific choices of how we choose to represent numbers in hardware.

The key insight here is that all concrete representations are finite, and at some point an increasing quantity will be so big as to overflow its container. When that happens, we accept that the overflow will be silently discarded. Given this property, we can now reframe the original problem: What binary number can we add to a given binary number such that the sum is just big enough to be zero **after** the overflow has been discarded? In other words, we need the sum to be a number whose (a) length is one bit greater than will fit in the allocated storage and (b) only non-zero bit is its MSB. The goal here (when adding 8 bit numbers) is to get a result that is 9 bits long with zeros in all of the original bits, so we can discard the finally carried over 1 (into the non-existent 9th bit) ending up with a zero, practically.

It's easy to come up with examples where this happens for smaller numbers that fit within a byte with plenty of space to spare. Consider 3 for instance. In binary 3 would be represented by 11 (two bits). Adding 1 to 3 gives us 4, which is 100. 4 is 3 bits long and has a 1 in its MSB with zeroes in all the other bits. Similarly 5 (101 in binary) + 3 (011) gives us 8 (1000).

Now we ask, given any 8 bit binary number,  $x$ , what other 8 bit number,  $y$ , can you add to  $x$ , such that  $x+y$  gives a 9 bit number with 0s everywhere except the MSB?

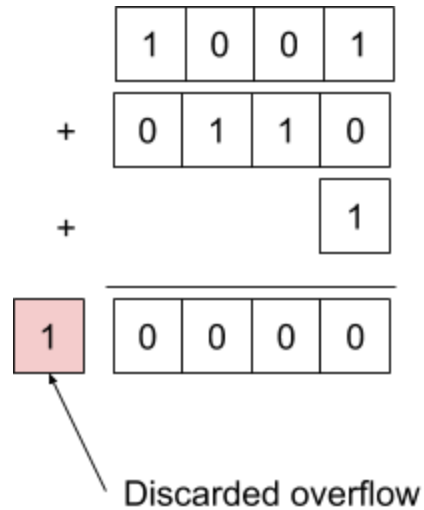
Here is an observation that helps us get the final answer: When we add 1 to any bit pattern that's all 1s, we end up with a bit pattern that's all zeroes and a 1 that carries beyond the leftmost digit (MSB). The picture below shows how this works with a single nybble (instead of an entire byte).



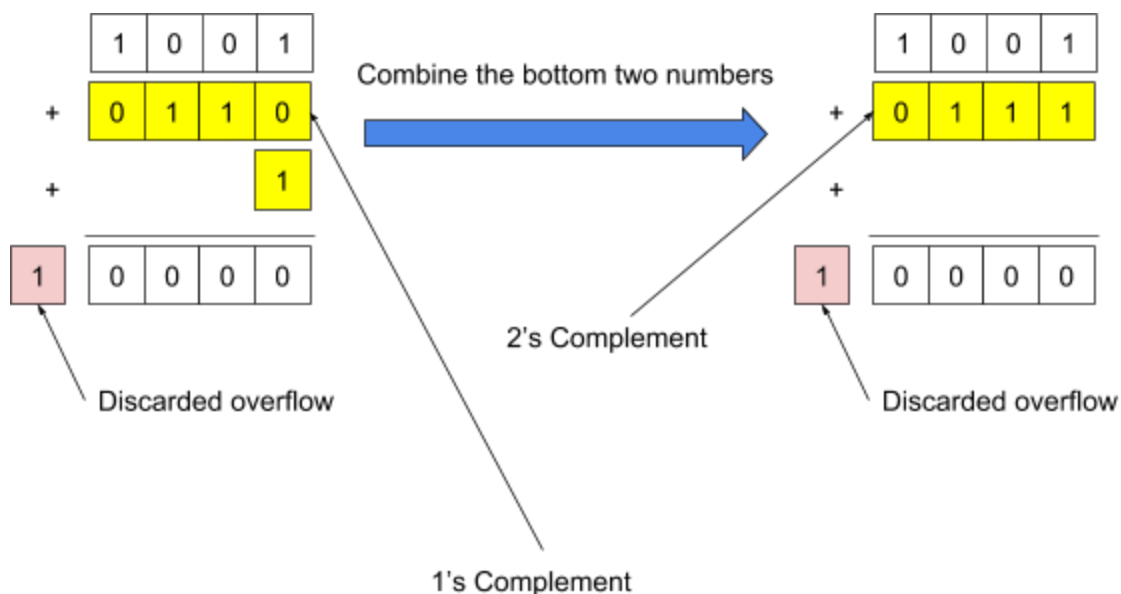
So only one thing remains at this point. How do we derive a byte (or any other fixed sized binary number) that's all 1s from a given binary number? It's quite simple, really. We add a 1 to every bit that's 0 and a 0 to every bit that's a 1. In other words, to every bit, you add the complement of that bit. When you apply the complement operation to every bit in a byte, what you effectively get is a complement of a byte. The resultant byte is called the 1's complement of the original. Adding a byte to its 1's complement always gives a byte that's all 1s.

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 1 \\ \hline \end{array} \\
 + \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline \end{array} \\
 \hline
 \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline \end{array}
 \end{array}$$

Here is the final observation that also leads us to the denouement. By adding 1 to the sum of a numeric byte and its 1's complement, we get all zeroes in the byte with a carry to discard.



Since addition is associative, by which I mean that  $(a+b) + c = a + (b+c) = a+b+c$ , we can prime our *annihilating* byte by adding 1 already to the one's complement of the original byte. The resultant bit pattern, which is one more than the 1's complement of the original, is now called - you guessed it - the 2's complement of the original byte. Below, we continue our example using the nybble.



Only one final thing remains. In our example above, where we get a 0 (effectively) by adding 1001 to 0111, do we consider the first bit pattern the positive and the second its negation or vice versa? Is the first one a 9 and the second a -9 or is the first one a -7 and the second a 7?

Think of the ramifications of either choice before reading further.



Here we simply resort to common sense. Since there are only  $2^n$  distinct bit patterns in  $n$  bits, and we want to use up half of them to store negative integers, it follows that the maximum number of representable non-negative integers is simply  $2^n/2$  which is  $2^{n-1}$ . For a nybble this would be  $2^3$  which is 8. Thus our range of representable non-negatives in one nybble is 0 through 7. In one byte, we would be able to represent non-negatives from 0 through 127. Note that this has the nice side-effect of giving us non-negatives which all have a 0 in their MSB. Thus by examining the MSB of a number we can tell if it's negative or not (like in Sign Magnitude Notation).

This is also the reason why when you're working with integers in your program and your quantity gets too big, it becomes negative all of a sudden. What just happened was an overflow into the MSB.

Ponder: What's the minimum (negative) number you can represent in one nybble (using 2's complement)



There are many more advantages and efficiencies we get out of storing integers in 2's complement form (other than the fact we now, rightly, only use one bit pattern for zero). We won't go into those topics just now. Hopefully this gives you a feeling for the subtleties involved in the various decisions that have shaped what we, as programmers, take for granted in our daily programming lives.

## Representing Fractional Quantities

One final topic before we conclude this primer on data representation - We need to understand how fractional quantities can be represented in binary. It's pretty straightforward to anyone who understand how fractional quantities are represented in decimal notation. (I don't mean the rational representation with one integer over another; I mean the more general decimal notation such as 3.14159)

Going back to our discussion of the place value system, we recall that a decimal number, according to our currently accepted English convention, has digits representing the number of increasing powers of 10 as you move left and decreasing powers of 10 as you move right. We also saw that the rightmost digit in an integer is the one's digit, which is the number of  $10^0$ s. What comes after it (to its right)? It only makes sense to think that the sequence doesn't end there, but continues endlessly. In fact, any digit after the rightmost digit of an integer part (traditionally marked off with a single period, otherwise called a *point* or *radix*) represents an increasingly negative power of 10. Thus in the number 3.1415, the first 1 after the radix represents the number of tenths  $10^{-1}$ s, the 4 represents the number of hundredths or  $10^{-2}$ s, and so on.

When it comes to binary (or any other base for that matter), the situation is identical, except that we keep in mind that the base is different. Thus, 1.5 in decimal is actually 1.1 in binary - Why? Because the 1 after the radix in binary represents  $2^{-1}$  or one half. By the same reasoning, 9.625 in binary would be written as 1001.101 ( $2^3 + 2^0 + 2^{-1} + 2^{-3} = 8 + 1 + .5 + .125$ ).

Here's a challenge that's not critical to your mastery of the material of this course, but it's fun and instructive in your overall understanding of the subject. Convince yourself (in any base) that you can denote any real number between 0 and 1 using some combination of digits after the radix. In particular, convince yourself that the values 0.111... in binary, 0.FFF... in hexadecimal, 0.999... in decimal and 0.777 in octal, or 0.ZZZ in your base-27 are not only all convergent, but also identical and evaluate exactly (not approximately) to the value 1.



### Storing Fractional Quantities

Now we consider the issue of storing our fractional quantities. The first thing to keep in mind is that for continuing fractions and irrationals it's not even possible to denote their accurate values using a numeric representation (I mean a literal representation, not computable ones or symbols like  $\sqrt{2}$ ,  $\pi$  or  $e$ ) We've never ever done that and won't either. So let's not pursue that fruitless path.

On the other hand, we can and do write down approximations all the time (like 3.1416 for  $\pi$ , 2.71828 for  $e$ , or 1.4142 for  $\sqrt{2}$ ). So let's focus our energies on how to store a possibly approximate fractional quantity using a finite length of paper or a finite number of bits. We already know how we do this for decimal fractions, so we won't discuss that too much except where it helps to discuss binary by analogy.



The challenge in storing a fractional quantity on a digital device is the same as the challenge we faced when having to store negative numbers. There simply isn't a switch state we could use to uniquely represent the radix. We only have *on* and *off*, 1 and 0 respectively. No dash for a minus or . for a radix. How do we get around this issue? Well, we solve this problem the same way we solved the problem of the missing minus. But before we get there, let's quickly review a topic from middle school - Scientific Notation.

**Scientific Notation** lets us write a fractional quantity in the form of a number multiplied by a base raised to an integral power. For example, an approximate value of  $\pi$  in Scientific Notation would be  $3.141 \times 10^0$ . Of course, you could also write that as  $0.3141 \times 10^1$  or  $31.41 \times 10^{-1}$ , but the normalized or standard form we're used to seeing typically has a single number whose magnitude is less than 10 but greater than 1 before the times sign.

In a similar way, you could write a number in binary scientific notation using a power of 2 instead of 10. Thus, for example, you'd write 7.5 (which, in binary would be 111.1) as  $1.111 \times 2^2$  (You can verify that's correct, of course, because  $1.111$  is nothing but  $1 + 0.5 + 0.25 + 0.125 = 1.875$  and  $1.875 \times 4 = 7.5$ ). Let's now take this notation a step further to get two numbers neither of which has a radix. Written in this form, 7.5 would be  $1111 \times 2^{-1}$ .

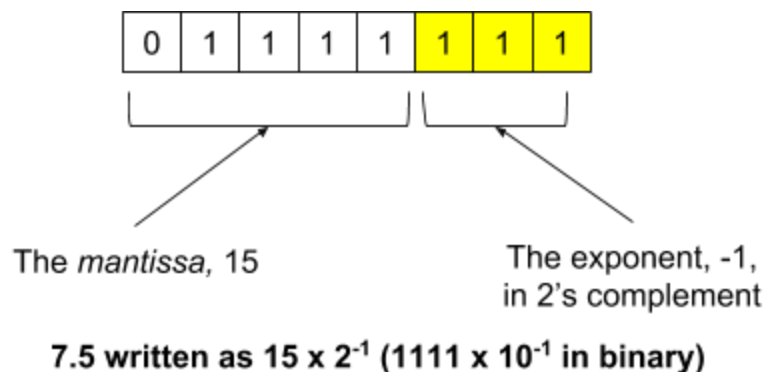
Many Computer Scientists call the part before the times sign the *Mantissa* and the exponent to which the base is raised as, well, the *exponent*. But as Don Knuth points out, there are good reasons to not use the word *mantissa* in this context because it has quite a different meaning in the context of logarithms.



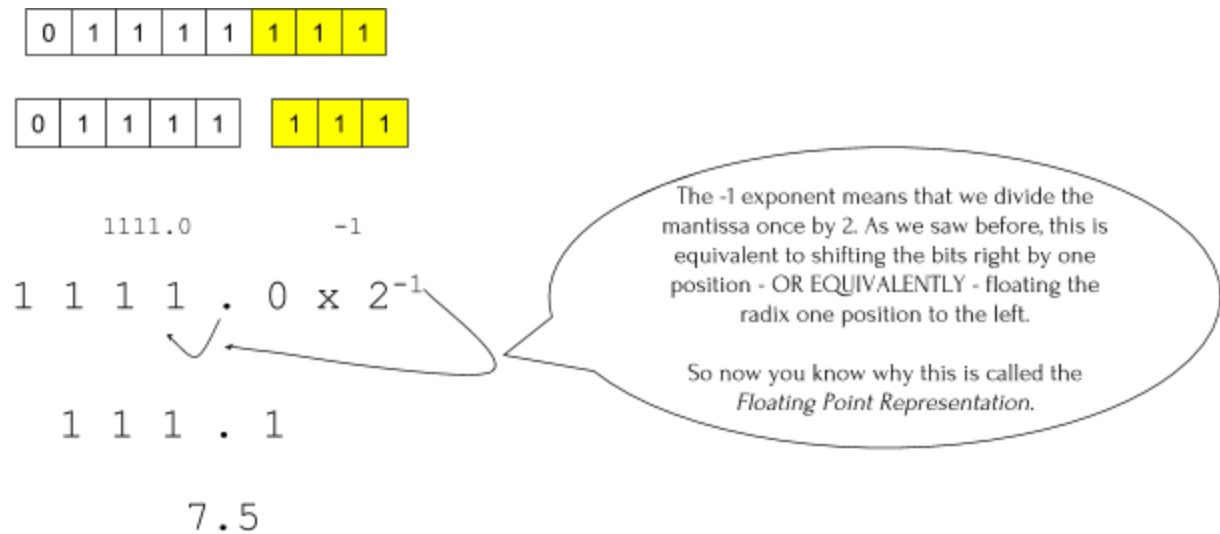
Nonetheless, for the purposes of this document, we'll refer to it as the *mantissa* if only to save space. We'll say that the mantissa is a non-fractional integral quantity preceding the times sign. If it turns out to be fractional, we'll assume that the exponent can be adjusted as necessary to make the mantissa whole.

Now we're in a position to propose a workable convention for storing fractional numbers. All we have to do is to divide up a byte into two parts, say the first 5 bits and the last 3 bits, store the mantissa of the binary scientific notation, the part before the times sign, in the first five bits, and the exponent in the last 3 bits.

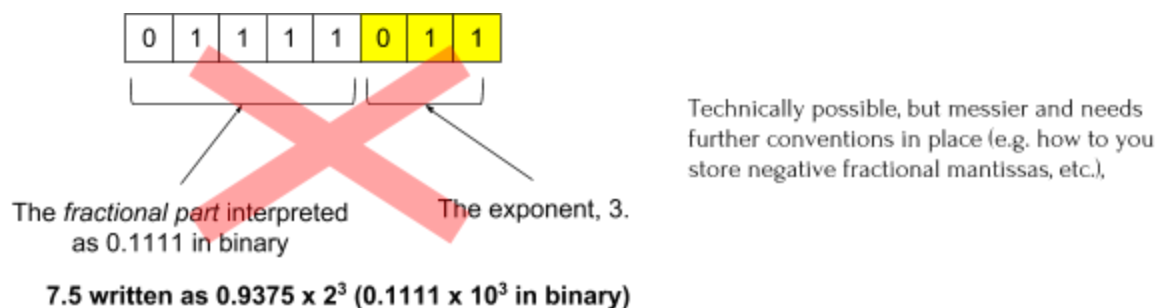
In the case of 7.5, our byte would now be composed of the two integers (15,-1) because 7.5 is  $15 \times 2^{-1}$ . The fifteen would be stored in the first five bits and the -1 in the last 3.



To decode this byte, which *must be* understood to be a fractional quantity, we would first split it up into its constituents per our initial agreement and then multiply the mantissa by 2 raised to the calculated exponent. Now this, as you already know, is quite simple and extremely fast in hardware - It's just a matter of shifting bits to the right or left (depending on the sign of the exponent). Equivalently to shifting bits right or left, we may move *or float* the point (radix) left or right, respectively. That's why this representation of fractional numbers is called - you guessed it - ...



Note that if we adopted the other convention that the radix always goes to the left of the mantissa (that is, the mantissa is between .1 and 1), then our 7.5 might be coded as  $0.9375 \times 2^3$ , which gives the same value after floating the radix to the right. But in that case, the bitwise representation of the mantissa gets trickier because we don't yet have an agreement on how to represent a negative fractional outside the framework we just built.



That concludes our preliminary discussion about information and data representation. The rest of the detail is, as they say, is just a matter of size - whether we use a nybble, a byte, or multiple bytes to store our various primitive data types. For the most part we used a single byte to demonstrate the techniques and conventions we have in place. But it must be clear by now that

the same techniques can be used to store integers and floating point numbers whether the underlying storage spans one byte (char), two bytes (a short int), 4 bytes (a long integer or float) or 8 bytes (a long long integer or a double). We won't go into that level of detail because what is short, long, single or double is more strongly a factor of the "now" of CS than are the underlying principles. A minor revolution in computer hardware tomorrow might cause us to say "double is the new single" or "long is the new short". We'll cover these issues and more in class.

I hope you enjoyed this little lesson. Onward to program now. Take the red pill.

Happy hacking.

&

