15-122: Principles of Imperative Computation

Recitation 3 — Josh Zimmerman

Hexadecimal notation

Just as we can represent numbers in base 2 (binary), we can represent numbers in other bases. Base 16, or hexadecimal (often called hex), is one such base. For clarity, hex numbers are written with "0x" at the beginning, so it's always possible to tell what base they are. Hex is useful because it allows us to compactly represent binary numbers: each hex digit corresponds to exactly 4 bits. For example, 0x4a = 01001010.

It's worth noting that any base that is a power of 2 would have this property: in base 8, each digit represents 3 bits and in base 32 each digit represents 5 bits. Hex is used partially because 4 evenly divides word sizes on almost all computers (almost all computers in use today are 8, 16, 32, or 64 bit systems) and partially because the number of digits needed grows slower than decimal while not needing to use many more characters.

Since hex is a higher base than base 10, we need more symbols for digits. Here's a table of hex values and their binary and decimal equivalents. (Note: the letters in hex numbers can be written in uppercase or lowercase).

Hex	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Bin.	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Dec.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Formally, a hex number n that is d digits long is equal to $\sum_{i=0}^{d-1} n_i 16^i$. (If working with that definition, keep in mind that a letter in a hex number corresponds to an integer between 10 and 15, as given in the above table.)

For an example, convert the hex number 0x7f2c to binary.

Two's Complement

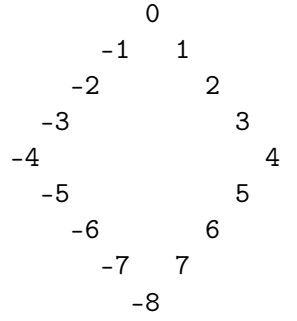
To represent negative numbers, we use the two's complement system.

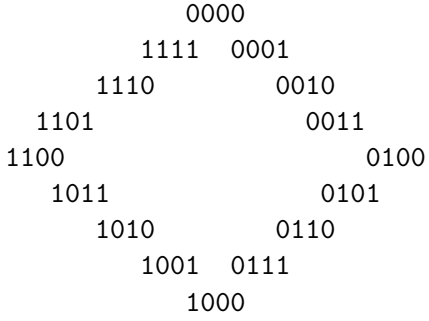
In unsigned (binary) arithmetic, a number n with k bits $d_{k-1} \dots d_0$ would be: $n = \sum_{i=0}^{k-1} d_i * 2^i$ (d_i is the ith digit of n, where the 0th digit is the rightmost one)

For example, in 4-bit unsigned arithmetic, $1011 = 1 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3 = 11$.

In two's complement, we modify this to allow us to have negative numbers. We make half of the numbers we represent nonnegative and half of them negative. (0 is neither positive nor negative. If you have an n-bit two's complement representation, there are $2^{n-1} - 1$ strictly positive numbers and 2^{n-1} negative numbers. This means that two's complement numbers that are n bits long must be at least -2^{n-1} and can be no more than $2^{n-1} - 1$.)

Two's complement arithmetic works like a clock (modularly), just like unsigned arithmetic does. Both of the below "clock" diagrams show 4-bit two's complement numbers. The left one shows the way they are interpreted and they way humans understand them and the right one shows the way they are represented as binary in the computer.





Except for 0 and -8, each number is across from its negative in both diagrams. Something else interesting to note here is that all negative numbers start with a 1 and all non-negative (positive and 0) numbers start with a 0. This is not just a coincidence, and we'll see why when we look at the formal definition of two's complement.

To find the negative of a two's complement number `x`, we simply flip all of the bits (so 1 becomes 0 and 0 becomes 1), and add one. The operator for flipping bits in C_0 is `~`. So, what this is saying is that `-x == (~x) + 1`. This is a very convenient property, since it lets us do addition without worrying about whether the number is in two's complement or not. (C_0 doesn't have any unsigned types but other languages, including C, do and this property allows for simpler hardware: we can use the same circuits to add unsigned and signed numbers.)

For example, let's calculate $5 + -5$. $5 = 4 + 1 = 1 * 2^2 + 1 * 2^0$, so its binary representation (if our integers are 4 bits long) is 0101. If we flip the bits, we get 1010, and if we then add 1 to that, we get 1011. Now let's do the addition:

```
  0101
+ 1011
------
 10000
```

However, since we're working with 4 bits, we have nowhere to store that leading 1, so the final answer that the CPU reports is 0000. So, in this case, two's complement worked out: $5 + -5 = 0$.

For another example, let's calculate $5 + -1$ in two's complement arithmetic.

We already know that $5 = 0101$. To find what -1 is, we take 0001, flip all of the bits, and add 1, giving us that -1 = 1110 + 1 = 1111.

Let's add to verify.

```
  0101
+ 1111
------
 10100
```

But again, this gets truncated to just the last 4 bits, which is 0100, which we can verify is 4.

To get a bit more formal, a k-digit two's complement number n is defined as follows: .

$$n = -d_{k-1}2^{k-1} + \sum_{i=0}^{k-2} d_i 2^i$$

So, in a 4-bit two's complement system, $-5 = 1011 = -1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = -8 + 2 + 1$.

The fact that we negate the most significant bit is what causes two's complement to work the way it does. If just the most significant bit is on, then the number is as small as it can be, because we're only subtracting and not adding. If every bit is on, we come close to canceling out the negative, but are just short of it, since $\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$.

Now, let's formally prove that flipping the bits and adding 1 does, in fact, produce the negation. Work on this by yourself or with other people to prove this.

Overflow

Overflow can cause potentially nasty bugs, so it's often advisable to check that your operations will not cause overflow before performing them. For example, let's add 7 + 1 using two's complement arithmetic:

```
  0111
+ 0001
------
  1000
```

But $1000 = -1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 = -8$. So 1 + 7 = -8 in four-bit two's complement arithmetic, which is not what we wanted to happen. (Similarly, -8 - 1 does strange things—you should play with that and see what happens.)

There's some more weird behavior when we negate the minimum int. (In a 4-bit system, that's 1000.) First, we flip the bits and get 0111. Next, we add 1 and get 1000, which is what we started with. That's an important oddity to be aware of: if you negate the minimum integer, you get the minimum integer back. This is related to the fact that there is one fewer strictly positive number than there are strictly negative numbers. (Which, in turn, is because of 0.)

So, clearly, overflow is bad. Write a precondition for a function `safe_add` that uses a precondition to ensure that overflow cannot happen.

```
1   int safe_add(int x, int y)
2   /*@requires _____________________________________
3     @       || _____________________________________
4     @       || _____________________________
5     @       || _____________________________;
6     @*/
7   {
8      return x + y;
9   }
```

ARGB images

We can represent colors using bits, too. In assignment 1, we represent images in an image format that uses the ARGB color scheme: Alpha (transparency level), red, green, blue.

We split a 32-bit integer into 8-bit sections, 1 each for each of those sections.

Here's a helpful visualization from Wikipedia:

(for a color version, you can look on `http://symbolaris.com/course/pic13-schedule.html`).

Sample Length:	8								8								8								8							
Channel Membership:	Alpha								Red								Green								Blue							
Bit Number:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Each of these "ints" (really, they're just sequences of 32 bits that we can pretend are ints) is one pixel in an image. We can store an array of these pixels which, along with width and height data, allows us to construct an image.

You'll take advantage of this fact on homework 1, and we'll talk more about ARGB images and manipulating bits in recitation on Friday.