15-122: Principles of Imperative Computation

Recitation 2

Josh Zimmerman, Nivedita Chopra

Lecture Recap

The lecture talked about integers in C0, including base representations, bitwise and shift operators, two's complement and modular arithmetic. We'll go over some of these topics in detail in this recitation.

Integers can be represented in other bases besides the decimal (base 10) notation that we are used to in day-to-day life. Binary (base 2) notation uses only two digits, 0 and 1. This can represent the on-off states of computers.

Checkpoint 0

What is the binary representation of 42?

Hexadecimal notation

Base 16, or hexadecimal (often called hex), is another commonly used base. It has 16 digits: 0 – 9 & A – F. 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 has this property. In base 8 (called octal), 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.

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

This table is important and often, knowing the conversions (e.g. 1010 is A) will save you time on exams.

Formally, a hex number n that is d digits long is equal to $\sum_{i=0}^{d-1} n_i 16^i$. (If you're 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.)

Checkpoint 1

Convert the hex number 0x7f2c to binary.

ARGB images

We can represent colors using bits, too. In lab 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 four 8-bit sections and these sections correspond to alpha, red, green and blue in order. Here's a helpful visualization from Wikipedia:

(for a color version, you can look at `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/rec/03.pdf`, which you can get to from the course calendar).

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. An array of these pixels, along with width and height data, allows us to construct an image.

Bit manipulation

Bitwise operators operate on integers one bit at a time. They treat all bits in a number as independent units that don't have anything to do with each other. Here are some tables illustrating the bitwise operators in C0:

and

&	1	0
1	1	0
0	0	0

or

|	1	0
1	1	1
0	1	0

xor (exclusive or)

^	1	0
1	0	1
0	1	0

complement

~	1	0
	0	1

There are also shift operators. They take a number and shift it left (or right) by the specified number of bits. In C0, right shifts *sign extend*. This means that if the first digit was a 1, then 1s will be copied in as we shift. For shifts, note that it doesn't really make sense to shift by more bits than there are in the number. If you do so, C0 will give a division by 0 error.

Note that left-shifting by k is equivalent to multiplying by 2^k and that right-shifting by k is equivalent to dividing by 2^k and rounding towards $-\infty$.

Here are some examples (assuming we're using a 4-bit two's complement system):

$$\begin{array}{r} 0101 \\ \& \ 1100 \\ \hline 0100 \end{array} \qquad \begin{array}{r} 0101 \\ \mid \ 1100 \\ \hline 1101 \end{array} \qquad \begin{array}{r} 0101 \\ \hat{} \ 1100 \\ \hline 1001 \end{array} \qquad \begin{array}{r} \sim \ 0101 \\ \hline 1010 \end{array} \qquad \begin{array}{r} 0101 \\ >> \quad 1 \\ \hline 0010 \end{array} \qquad \begin{array}{r} 1101 \\ >> \quad 1 \\ \hline 1110 \end{array} \qquad \begin{array}{r} 0101 \\ << \quad 1 \\ \hline 1010 \end{array} \qquad \begin{array}{r} 1101 \\ << \quad 1 \\ \hline 1010 \end{array}$$

Bitwise operations can also be thought of in another way. Assume that a bit is represented by the variable X (which is either 0 or 1)

- Bitwise AND clears the bit
  X & 0 = 0 X & 1 = X
- Bitwise OR sets the bit
  X | 1 = 1 X | 0 = X
- Bitwise XOR toggles the bit
  X ^ 1 = ~X X ^ 0 = X

This easily translates into the idea of a *mask*, which we'll see in labs 1 and 2. Masks are used to set, get, invert or do other operations on certain bits of a number with one bitwise operation.

Let's look at an example of masking so you can get a better idea of how it's used. Here is function that, given a pixel in the ARGB format, returns the green and blue components of it.

```
1  typedef int pixel;
2  int greenAndBlue(pixel p)
3  //@ensures 0 <= \result && \result <= 0xffff;
4  {
5    return p&0xFFFF;
6  }
```

`typedef` is a command that is used to define a new data type. In this example, we defined the new type `pixel` to be the same as an `int`. An easy way to remember the order of the arguments to a `typedef` statement, is to think of it as a simple variable declaration without the `typedef`.
e.g. `int pixel;`
This is a valid C0 statement (treating pixel as a variable name), hence our order of arguments is correct.

Checkpoint 2

Write a function that gets the alpha and red pixels of a pixel in the ARGB format. Your solution can use any of the bitwise operators, but will not need all of them.

```
1  typedef int pixel;
2  int alphaAndRed(pixel p)
3  //@ensures 0 <= \result && \result <= 0xffff;
4  {
5
6  }
```

Two's Complement

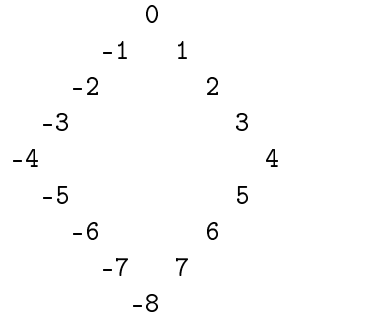
To represent negative numbers, we use the two's complement system.

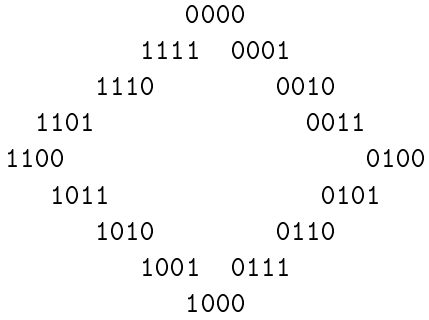
In unsigned (binary) arithmetic, a number n with k bits $d_{k-1} \dots d_0$ would be: $n = \sum_{i=0}^{k-1} d_i * 2^i$ (d_i is the ith digit of n, where the 0th digit is the rightmost one)

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: $5 + -5 = 0$.

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 $\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$.

Checkpoint 3

Calculate $5 + -1$ in two's complement arithmetic.

Checkpoint 4

(Optional) Formally prove that flipping the bits and adding 1 does, in fact, produce the negation. Work by yourself or with other people to prove this.