15-122: Principles of Imperative Computation

Recitation 1: Bits and Bites on Bits and Bytes — Nivedita Chopra

Converting between binary and decimal

In class, we saw an algorithm for converting between our familiar decimal notation and binary notation. The example to the left describes the conversion between the decimal representation of 20 (which we could write as $20_{[10]}$ to emphasize its decimal-ness) and the binary representation $10100_{[2]}$.

$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$
$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$
$\_\_\_\_\_ \times 2 + 1 = 1$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$
$1 \times 2 + 0 = 2$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$
$2 \times 2 + 1 = 5$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$
$5 \times 2 + 0 = 10$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$
$10 \times 2 + 0 = 20$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$	$\_\_\_\_\_ \times 2 + \_\_\_\_\_ = \_\_\_\_\_$

Checkpoint 0

What is the decimal representation of $1111010_{[2]}$? _____

What is the binary representation of $49_{[10]}$? _____

Hexadecimal notation

Hex is useful because every hex digit corresponds to exactly 4 binary digits (bits). Base 8 (octal) is similarly useful: each octal digit corresponds to exactly 3 binary digits. However, hex more evenly divides up a 32-bit integer.

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

In C0 we indicate we are using base 16 with an `0x` prefix, so we write $7f2c_{[16]}$ as `0x7f2c`.

Convert the binary number $1011111010101101_{[2]}$ to hex. _____

Convert the hex number $20_{[16]}$ to decimal. _____

Why wouldn't it make sense to write a C0 function that converts hex numbers to decimal numbers?

Bit manipulation

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

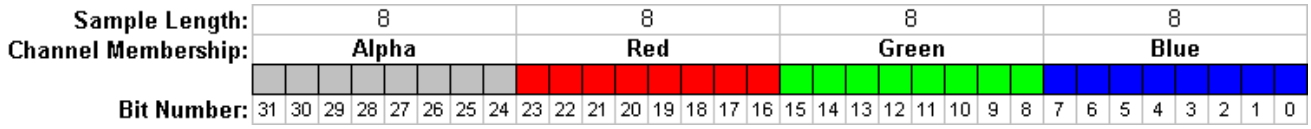
complement

~	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.

ARGB representation of color

We usually use 32-bit integers in C0 to represent a single integer. However, it's possible to use the bits in other ways: as 32 separate Boolean values or as 4 separate 8-bit numbers in the range $[0, 255)$. This lets us represent a color (red, green, and blue intensities, plus transparency or "alpha"), as 32-bit C0 integer.



Checkpoint 1

Write a function that gets the alpha and red pixels of a pixel in the ARGB format, moving them from bits 31-16 to bits 15-0. 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     return ____________________________________________________;
6  }
```

Two's complement

Because C0's `int` type only represents integers in the range $[-2^{31}, 2^{31})$, addition and multiplication are defined in terms of modular arithmetic. As a result, adding two positive numbers may give you a negative number!

Checkpoint 2

What assertion would you need to write to ensure that an addition would give a result without overflowing (in other words, to ensure that the result you get in C0 is the same as the result you get with true integer arithmetic).

```
1 int safe_add(int a, int b)
2 /*@requires
3
4
5
6
7 @*/
8 { return a + b; }
```

What about multiplication? For the sake of simplicity, you can assume both numbers are non-negative.

```
1 int safe_mult(int a, int b)
2 /*@requires a >= 0 && b >= 0 &&
3
4
5
6
7 @*/
8 { return a * b; }
```