

15-122: Principles of Imperative Computation

Recitation 1: A Bit About Bytes Nivedita Chopra, Aaron Gutierrez

Converting between binary and decimal

To easily convert a number represented in binary notation, such as $10100_{[2]}$, we can employ Horner's algorithm. At each step, we multiply the previous result by 2, and add the next bit in the number. To convert in the other direction, we divide by 2 and write the remainder at each step from bottom to top. We can see the conversion between $10100_{[2]}$ and 20 (or $20_{[10]}$ to be extra-decimally) below.

_____ × 2 + _____ = _____	_____ × 2 + _____ = _____	_____ × 2 + _____ = _____
_____ × 2 + _____ = _____	_____ × 2 + _____ = _____	_____ × 2 + _____ = _____
_____ × 2 + <u>1</u> = <u>1</u>	_____ × 2 + _____ = _____	_____ × 2 + _____ = _____
<u>1</u> × 2 + <u>0</u> = <u>2</u>	_____ × 2 + _____ = _____	_____ × 2 + _____ = _____
<u>2</u> × 2 + <u>1</u> = <u>5</u>	_____ × 2 + _____ = _____	_____ × 2 + _____ = _____
<u>5</u> × 2 + <u>0</u> = <u>10</u>	_____ × 2 + _____ = _____	_____ × 2 + _____ = _____
<u>10</u> × 2 + <u>0</u> = <u>20</u>	_____ × 2 + _____ = _____	_____ × 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. In C0 we indicate we are using base 16 with an 0x prefix, so $7f2c_{[16]}$ is $0x7f2c$.

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

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

Convert the $0x20$ to decimal. _____

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

Bit manipulation

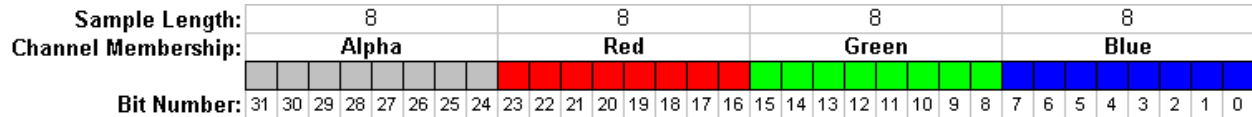
and	or	xor (exclusive or)	complement
& 1 0	1 0	~ 1 0	~ 1 0
1 1 0	1 1 1	1 0 1	0 1
0 0 0	0 1 0	0 1 0	

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.

$$1101\ 1111\ 0101\ 0010_{[2]} \gg 8 = \mathbf{1111\ 1111}\ 1101\ 1111_{[2]}$$

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.



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 1

Write a function that returns 1 if the sign bit is 1, and 0 otherwise. That is, write a function that returns the sign bit shifted to be the least significant bit. Your solution can use any of the bitwise operators, but will not need all of them.

```
1 int getSignBit(int x)
2 //@ensures |result == 0 || |result == 1
3 {
4     return _____;
5 }
```

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; }
```