15-122: Principles of Imperative Computation

Recitation Week 1 — Josh Zimmerman, Nivedita Chopra

Checkpoint 0

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

What is the binary representation of $42_{[10]}$? _____

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`.

Checkpoint 1

Convert the hex number $7f2c_{[16]}$ to binary. _____

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

Bit manipulation

and

&	1	0
1	1	0
0	0	0

or

|	1	0
1	1	1
0	1	0

xor (exclusive or)

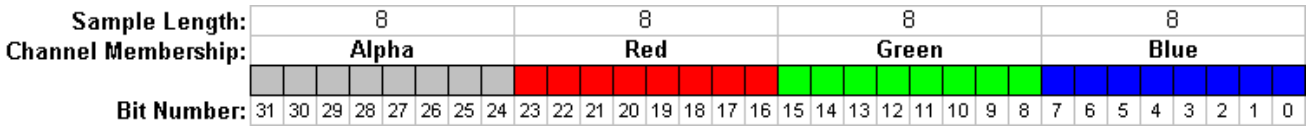
^	1	0
1	0	1
0	1	0

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.

Checkpoint 2



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
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 3

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).

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

Checkpoint 4

Fill in the blanks in the code to show that there are no out of bounds array accesses. Assume you are given the function `slow_fib` that can be used as a reference function.

```
1  int slow_fib(int n) /*@requires n >= 0; @*/ ;
2
3  int fib(int n)
4  //@requires ____________;
5  //@ensures \result == slow_fib(n);
6  {
7    int[] F = alloc_array(int, n);
8    if (n > __) {
9       F[0] = 0;
10   }
11   else {
12      return 0;
13   }
14   if (n > __) {
15      F[1] = 1;
16   }
17   else {
18    return 1;
19   }
20   for (int i = 2; i < n; i++)
21    //@loop_invariant ________________________________;
22    /*@loop_invariant F[i - 1] == slow_fib(i - 1)
23                && F[i - 2] == slow_fib(i - 2); @*/
24    {
25      F[i] = F[i - 1] + F[i - 2];
26    }
27   return F[n - 1] + F[n - 2];
28 }
```

Checkpoint 5

Are the loop invariants strong enough to allow you to prove the postcondition? Show that the loop invariant is true initially, and is preserved by every iteration of the loop; the loop invariant and the negation of the loop guard imply the postcondition; and the loop terminates.