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-decimaly) below.

____	× 2 +	____	=	____	____	× 2 +	____	=	____	____	× 2 +	____	=	____
____	× 2 +	____	=	____	____	× 2 +	____	=	____	____	× 2 +	____	=	____
____	× 2 +	1	=	1	____	× 2 +	____	=	____	____	× 2 +	____	=	____
1	× 2 +	0	=	2	____	× 2 +	____	=	____	____	× 2 +	____	=	____
2	× 2 +	1	=	5	____	× 2 +	____	=	____	____	× 2 +	____	=	____
5	× 2 +	0	=	10	____	× 2 +	____	=	____	____	× 2 +	____	=	____
10	× 2 +	0	=	20	____	× 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 bits. 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 hexadecimal number `0x20` 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.

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

Checkpoint 1

What does ($0001\ 0101_{[2]}$ & $0011\ 0101_{[2]}$) | ($1010\ 1010_{[2]}$ ^ $0001\ 1110_{[2]}$) evaluate to? ____________

What does ($5_{[10]}$ | $13_{[10]}$) ^ ($28_{[10]}$ & $10_{[10]}$) evaluate to? ____________________

What is the difference between logical and bitwise operators?

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

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.

```
int getSignBit(int x)
//@ensures \result == 0 || \result == 1;
{
    return ____________________________________________;
}
```

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).

```
int safe_add(int a, int b)
/*@requires



@*/
{ return a + b; }
```

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

```
int safe_mult(int a, int b)
/*@requires a >= 0 && b >= 0 &&



@*/
{ return a * b; }
```

ARGB representation of color

In C0 we use 32-bit **int**s 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 an **int**:

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