15-122: Principles of Imperative Computation — Spring 2024

Recitation 02: A Bit About Bytes — Thursday January 18th

Binary and decimal representations

Although we typically write numbers in decimal (the base 10 number system), computers represent all values in binary, in base 2. So it is important to know how to understand numbers written in both of these representations. The way we interpret $106_{[10]}$ is as a sum of powers of 10. For example,

$$106_{[10]} = 1 \times 10^2 + 0 \times 10^1 + 6 \times 10^0$$

The position of the digit determines the power of 10 used for each term.

Similarly, we interpret a binary number as a sum of powers of 2. For example,

$$1101010_{[2]} = 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

If we carry out this calculation in decimal, we obtain 106. This provides a simple recipe to convert between the binary and decimal representations of a number.

To find the binary representation of a number written in decimal, we need to take a different route: we repeatedly divide by 2. The remainder of each step read from bottom to top is its binary representation. Let's convert 106 back to binary in this way:

106	/ 2 =	53	with a remainder of	0		____	/ 2 =	____	with a remainder of	____
53	/ 2 =	26	with a remainder of	1		____	/ 2 =	____	with a remainder of	____
26	/ 2 =	13	with a remainder of	0		____	/ 2 =	____	with a remainder of	____
13	/ 2 =	6	with a remainder of	1		____	/ 2 =	____	with a remainder of	____
6	/ 2 =	3	with a remainder of	0		____	/ 2 =	____	with a remainder of	____
3	/ 2 =	1	with a remainder of	1		____	/ 2 =	____	with a remainder of	____
1	/ 2 =	0	with a remainder of	1		____	/ 2 =	____	with a remainder of	____

Checkpoint 0

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

Using the right side of the table above, what is the binary representation of $49_{[10]}$? ____________

Although it is important that you are able to do these conversions yourself, there are a number of good conversion tools online. For example, we like this one for binary to decimal (and more).

Hexadecimal notation

Hexadecimal represents numbers in base 16. Every hex digit corresponds to exactly 4 binary digits (bits). Thus, a 32-bit **`int`** can be written using 8 hex digits. We do so in C0 by using the prefix `0x`: we enter the hex number $7F2C_{[16]}$ in C0 as `0x7F2C`. The hexadecimal representation allows us to "see" the bit structure of an **`int`** without having to write out 32 `1`'s and `0`'s. In fact, C0 does not support writing numbers in binary at all — only hex and decimal are supported.

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

Find the hex representation of the binary number $0011111010101101_{[2]}$. ____________

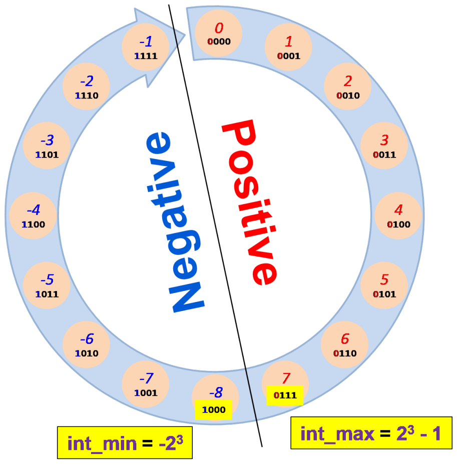
Find the decimal representation of the hexadecimal number `0x20`$_{[16]}$. ____________

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

____

Two's complement

So far we have only considered non-negative numbers, but C0's **`int`** type represents integers in the range $[-2^{31}, 2^{31})$. C0 uses *two's complement* to determine which 32-bit **`int`**'s it considers negative. In two's complement, the most significant bit has negative place value. The figure to the right shows how two's complement partitions the 4-bit numbers into positive and negative. Note that the leftmost bit is always `1` for negative numbers and always `0` for positive numbers (and zero). Because of this, we call this bit the *sign bit*.



The same can be done for numbers that are represented using k bits for any value of k, including 32 which is what C0 uses.

To convert a number in two's complement to decimal, we apply a variant of the powers of 2's method we saw earlier: we make the term for sign bit *negative*. For example, in a 4-bit world with two's complement,

$$1011_{[2]} = 1 \times \underset{\Uparrow}{-2^3} + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

which evaluates to $-5_{[10]}$.

What is the range of the values that can be represented using k bits using two's complement?

Checkpoint 1

Assume you are in a 7-bit world (where numbers are written as 7 bits) that uses two's complement:

What is $1101010_{[2]}$ in decimal? ______________________

What is $0011101_{[2]}$ in decimal? ______________________

Notice that the first value you computed here consists of the same binary digits as the example at the beginning of this recitation, but now that we're in a two's complement world, you interpret the binary representation differently!

Notable Bit Patterns

In C0's 32-bit world, what are (in binary and hex):

`int_min`: ______________________$_{[2]}$, ____________$_{[16]}$

`int_max`: ______________________$_{[2]}$, ____________$_{[16]}$

`-1`: ______________________$_{[2]}$, ____________$_{[16]}$

Remember that in C0, integers are always 32-bits! By using the `<util>` library, you can enter `int_min` and `int_max` as `int_min()` and `int_max()` — note the parentheses.

Checking for overflow

Because **`int`**'s are just 32 bits long, numbers that are not in the range $[-2^{31}, 2^{31})$ cannot be entered in C0. Such numbers can however emerge as the result of an arithmetic operation, something that is called overflow. For example, 4×2^{30} overflows since its mathematical value is 2^{32} which is outside the range $[-2^{31}, 2^{31})$.

Thus, an *overflow* is when the result of evaluating a C0 expression of type **int** is not the same as the result we get with true integer arithmetic.

Checkpoint 2

We want to write a function that adds to numbers only when no overflow can occur and fail a contract otherwise. Here's our first attempt:

```
int no_overflow_add(int a, int b)
/*@requires (a > 0 && b > 0 && a + b <= int_max())
         || (a < 0 && b < 0 && a + b >= int_min())
         || (a <= 0 && b >= 0)
         || (a >= 0 && b <= 0);
@*/
{
  return a + b;
}
```

This doesn't quite work as intended. Explain what is wrong with the precondition as written. How would you correct the precondition so that it fails whenever there would be an overflow?

Bit patterns

In C0, we use 32-bit **int**'s to represent a single integer. However, it's possible to use these 32 bits to encode other information using the 32 bits of an **int**.

It makes little sense to use arithmetic operations on such *bit patterns* (**int**'s we think of as representing information other than a single number). Instead, we manipulate bit patterns using a dedicated set of operations on **int**'s. These are the *bitwise operations* and the *shifts*.

What operators to use on a value of type **int** depends on what we see this value as.

- If we view it as a *number*, we should exclusively use arithmetic operators on it.
- If instead we view it as a *bit pattern*, we should manipulate it exclusively with bitwise operators and shifts.

Bitwise operators

C0 has four bitwise operations. They are called bitwise because they manipulate each bit in an **int** independently of the bits around it. Here's how they work on a single bit:

and

&	1	0
1	1	0
0	0	0

or

|	1	0
1	1	1
0	1	0

xor

^	1	0
1	0	1
0	1	0

complement

~	1	0
	0	1

In C0, the bitwise operators apply to entire **int**'s: they apply the above tables to each of the 32 positions of their operands.

Observe that the bitwise operators `~`, `&` and `|` work similarly to the logical operators `!`, `&&` and `||`. They are not interchangeable however: the former operate on values of type **int** while the latter operate on values of type **bool**.

Checkpoint 3

Assume we are in the 5-bit world.

What does $(01101_{[2]}\ \&\ 10101_{[2]})\ |\ (01010_{[2]}\ \hat{}\ 10110_{[2]})$ evaluate to?

Given a 32-bit **int** `p`, write an expression that sets bits 0–7 to 0, sets bits 16–23 to 1, and preserves the remaining bits. For example, for `p = 0x12345678`, this expression would evaluate to `0x12FF5600`.

Shifts

The two shift operators, `x << k` and `x >> k`, move bits around an **int** `x`. They take an **int** understood as a bit pattern and shift it left or right, respectively, by the specified `k` bits. The left shift `x << k` always sets the rightmost `k` bits of the result to `0`. Instead, the right shift `x >> k` copies the sign bit of `x` to the leftmost `k` bits of the result — this is called *sign extension*. Here are some 32-bit world examples:

$$\begin{array}{rcl}
1101\ 1111\ 0101\ 0010\ 1101\ 1111\ 0101\ 0010_{[2]} \ll 9 & = & 1010\ 0101\ 1011\ 1110\ 1010\ 010\mathbf{0\ 0000\ 0000}_{[2]} \\
0101\ 1111\ 0101\ 0010\ 0101\ 1111\ 0101\ 0010_{[2]} \gg 9 & = & \mathbf{0000\ 0000\ 0}010\ 1111\ 1010\ 1001\ 0110\ 1111_{[2]} \\
1101\ 1111\ 0101\ 0010\ 1101\ 1111\ 0101\ 0010_{[2]} \gg 9 & = & \mathbf{1111\ 1111\ 1}110\ 1111\ 1010\ 1001\ 0110\ 1111_{[2]} \\
80ABCDEF_{[16]} \gg 9 & = & FFC055E6_{[16]}
\end{array}$$

For either shift to be valid, the shift amount `k` must be between 0 (inclusive) and the number of bits in the representation, 32 here (exclusive).

Mixing bitwise and arithmetic operators

We said earlier that **int**'s seen as numbers should be manipulated exclusively with arithmetic operators, while **int**'s seen as bit patterns should be manipulated exclusively with bitwise operators and shifts.

There are very few exceptions to this rule. One is when using the left shift operation to quickly compute powers of two. It leverages this property:

$$\mathtt{x} \ll \mathtt{k} \quad = \quad \mathtt{x} \times 2^{\mathtt{k}}$$

(The right shift corresponds to a variant of division that always rounds towards negative infinity.)

Another exception is when we are dealing with numbers but are interested in aspects of their binary representation, like the value of the sign bit.

Checkpoint 4

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

Fun facts about integers. . .

. . . that may come handy.

- `int_max() + 1 == int_min()`
- `int_min() - 1 == int_max()`
- `-int_min() == int_min()`
- `-x == ~x + 1`

Take-home exercise: *prove that each of the above equalties hold in two's complement.*