CSE 410 Section 2 – Integers and Bitwise Operators

Hi there! Welcome back to section, we're happy that you're here 😊

IEC Prefixing System
We often need to express large numbers and the preferred tool for doing so is the IEC Prefixing System!

- Kibi- (Ki) = \(2^{10} \approx 10^3\)
- Mebi- (Mi) = \(2^{20} \approx 10^6\)
- Gibi- (Gi) = \(2^{30} \approx 10^9\)
- Tebi- (Ti) = \(2^{40} \approx 10^{12}\)
- Pebi- (Pi) = \(2^{50} \approx 10^{15}\)
- Exbi- (Ei) = \(2^{60} \approx 10^{18}\)
- Zebi- (Zi) = \(2^{70} \approx 10^{21}\)
- Yobi- (Yi) = \(2^{80} \approx 10^{24}\)

Practice With Prefixes
Write the following numbers using IEC Prefixes:

\(2^{16} = \) 2 Ki
\(2^{34} = \) 2 Pi
\(2^{27} = \) 2 Gi
\(2^{61} = \) 2 Zi

Write the following numbers as powers of 2:

64 Gi = 2^{6}\)

Signed Integers With Two’s Complement
Two’s complement is the standard solution for representing signed integers. Remember that:
- The most significant bit has a negative value, all others have a positive value
- Otherwise exactly the same as unsigned integers

A neat trick for flipping the sign of a two’s complement number: flip all the bits and add 1.

Two’s Complement using 4 bit binary strings:
Notice how the largest number is 7 whereas the smallest number is -8. Also notice that there is a nice symmetry between numbers and their negative counterparts except -8, because its counterpart is taken to represent 0 in the circle.

- Addition is exactly the same as with an unsigned number.
- Arithmetic operations on binary numbers can lead to overflow. The easiest way to detect overflow is to keep in mind of the representational range of binary numbers.

**Two’s Complement Practice**
For the following questions assume an 8 bit integer. Answer each question for the case of a two’s complement number and unsigned number.

1. What is the largest integer? The largest integer + 1?

2. How do you represent the numbers 0, 1, and -1?

3. How do you represent 17, -17?

4. Prove that the two’s complement inversion trick is valid (i.e. that x and \(\bar{x} + 1\) sum to 0).

5. Explain why binary, decimal, or hexadecimal might be the preferred base of choice over the other bases. Give an example/context for your explanation!

6. What is the largest positive number we can represent with a 12-bit signed two's complement integer? Write out the bit pattern and the value in decimal!

**Another Problem**
Let \(x=0xEB\) and \(y=0x7\) be integers stored on a machine with a word size of 4 bits.

1. What is the hex value of the result of adding these two numbers?

2. Interpreting these numbers as unsigned integers, what is the decimal result of adding \(x + y\)?

3. Interpreting \(x\) and \(y\) as two's complement integers, what is the decimal result of computing \(x - y\)?

4. In one word, what is the phenomenon happening in question 2?
Bitwise Operators and Masking

C provides bitwise commands for AND(&), OR(|), XOR(^), and NOT(~). Ignoring NOT for now, let's see what happens when we reduce the 2-input gates to 1-input gates by fixing the second input.

Let x be the input. Fill in the following blanks with either 0, 1, x, or \( \bar{x} \) (NOT x):

\[
\begin{align*}
x & \& 0 &= \\
x & \& 1 &= \\
x & \mid 0 &= \\
x & \mid 1 &= \\
x ^ \wedge 0 &= \\
x ^ \wedge 1 &= \\
\end{align*}
\]

**FACT:** Masking is very commonly used in bitwise operations. Masking helps to either clear or to extract bits from a bit string. AND a mask of all ones with the bit string to extract the bits. XOR a mask of all ones with the bit string give the complement of the bit string.

Consider the following mask, and two bit strings from which we want to extract the final bit:

- mask = 00000001
- value1 = 10011011
- value2 = 10011100

Fill in the result:

mask & value1 = ______________

mask & value2 = ______________

The zeros in the mask *mask off* the first seven bits and only let the last bit show through. Masks can also be built up by operating on several flags, usually with inclusive OR:

- flag1 = 00000001
- flag2 = 00000010
- flag3 = 00000100

mask = flag1 \mid flag2 \mid flag3 = ______________

**Mask Practice**

Fill in the blank below to complete the function mod32 such that the function will return the remainder of x when divided by 32. The first blank is a bitwise operator. The second blank should be a decimal number.

```c
int mod32(int x) {
    return x ___ ___;
}
```

Finish the function check nth bit. The function checks and returns the nth bit of the given argument x.

```c
int check_nth_bit (int x, int n) {
    __________________;
    return ____ & 1;
}
```