15-122: Principles of Imperative Computation

Recitation 3 — Josh Zimmerman

Hexadecimal notation

For an example, convert the hex number 0x7f2c to binary.

Solution: Based on the table, we know that 7 corresponds to 0111, f corresponds to 1111, 2 corresponds to 0010, and c corresponds to 1100.

So, 0x7f2c = 0111111100101100.

Two's Complement

Now, let's formally prove that flipping the bits and adding 1 does, in fact, produce the negation. Work on this by yourself or with other people to prove this.

Solution: First, observe that for all `x`, `x + ~x == 11...11`. But when we add 1 to that, we simply get 0.

So, `x + ~x + 1 = 0`. With a bit of algebra, we can see that this means that `-x = ~x + 1`.

However, that only holds if `x` is not the minimum integer, since taking the negative of the minimum integer causes overflow. Thus, `-INT_MIN = INT_MIN`, and that is the one exception to this proof.

Overflow

So, clearly, overflow is bad. Write a precondition for a function `safe_add` that uses a precondition to ensure that overflow cannot happen.

```
int safe_add(int x, int y)
/*@requires x > 0 && y > 0 && x + y > 0
  @       || x < 0 && y < 0 && x + y < 0
  @       || x <= 0 && y >= 0
  @       || x >= 0 && y <= 0;
  @*/
{
   return x + y;
}
```

Solution: The cases where overflow can happen are as follows:

- Two positives add to form a negative
- Two negatives add to form a positive

Note that if we have a negative added to a positive, we can't possible cause overflow since the result will be larger than the negative number and smaller than the positive number.

So, our precondition can be split into four cases mirroring the possibilities for `x` and `y`, as above.