Report

Student Name: Gong Zizhou Student Number: 124090142

1. Why bitwise operations work for arithmetic.

Numbers are represented by binary in computers, specifically two's complement. Arithmetic addition can be performed by XOR and AND bitwise operation. And other arithmetic operations can be decomposed into addition and shift.

2. How do you come up with such a solution.

First, I read the second chapter of the textbook and found the principle of arithmetic with only bitwise operation. For the two’s complement, we need first turn it into the unsigned integer, and then do bitwise operation in the unsigned context. For each kind of operation:

- **Add:** Use `a & b` to find the carry bits, and use `a ^ b` to sum the bits without carrying. Add `a ^ b` and `(a & b) << 1`. Since the carry bits might not be 0, we need to perform the addition for 32 times manually, for no loop are allowed. Finally, the carry bits are guaranteed to be 0, so the last summation part is the result.
- **Subtract:** Only need to turn b into -b, which is `add(~b, 1)`.
- **Multiply:** In primary school, we know that multiplication is derived from addition, so I mimic the idea. For every bit in b, I perform addition once. Finally, add all results from addition together, considering the weights of every bit.
- **Divide:** First, we can record the negativity of the result and we will directly apply the sign to the result at last. Intuitively, as long as we calculate how many bs are subtracted from a, until the remainder is less than b, we can get the quotient. And simply using while loop can solve the problem. However, if a is large enough and b is small enough, a timeout error could occur. Therefore, we need the subtraction be efficient. What if we subtract a multiple of b at a time? The multiple can be `b * (2 ^ n)` that is smaller than the remainder, thus requiring only 32 trials.
- **Modulo:** Alike the division, we just need the remainder part instead of the quotient.

3. What difficulties do you encounter, and how do you solve them.

1. I don't want to repeat the operation for 32 times in the addition implementation, but I can only comprimise.
2. I'm aware of the possible promblem that it may cause error if I use add(a_unsigned, b_unsigned) since add function accept only inputs in two's complement, but after searching on the Internet, I know that there will be an automatic type cast.
3. There will be an error in division without checking. For example, the initial i is 31 in the first loop, and if b_unsigned is 0x10000000, the processed b_modified will be 0x00000000, which will cause problem. So I utilized a check `if (!(b_unsigned >> (31 - i)))` to ensure there will not be the strange error.
4. In the test, I found that there were 4 modulo tests failed but all tests succeeded in the divide test. I found the reason that the sign of the modulo result is subjected to only the sign of a, which differs from the divide function.

4. Time complexity for your implementation.

- **Add:** O(1)
- **Subtract:** O(1)
- **Multiply:** O(1)
- **Divide:** O(1)
- **Modulo:** O(1)

5. Thoughts about unit testing.

Unit testing is an effective way to find potential bugs in each function. Without it, we can not predict if our functions will do well in future cases. Moreover, they can provide edge cases, which are often ignored by human. Also, once unit tests pass, code changes can be made with confidence without breaking existing functionality.

6. Other things you want to talk about.

None.