HPPS - Assignment 1

pwn274 and npd457

December 2021
Loopless - thank god for VSCode's vertical editing.

1 Introduction

Generally, we tried implementing our own solutions, not taking inspiration from other groups or the internet. We expect that this will be reflected in the lacking quality of our code.

In test_numbers.c we have tried testing the addition and multiplication with some different cases. The numbers all come out as expected, so we seem to have done something right.

As with all numbers in computers, the limited size of the binary representation (here 8 bits) allows for over- and underflow issues. Most of our algorithms seems robust, but as we are still not completely comfortable with two's-complement, there might be some problems with positive numbers turning negative that we have not found.

2 Converting to and from C integers

After some soul searching, we realized that taking any number mod 2 will return the last bit in its binary representation. Along with integer division by 2, this can be used to extract every bit in a binary number.

The division by two is simply a bitshift to the right, meaning the n'th bit can be found by:

```
bit = abs((x>>n)%2);
```

Converting the other way, we implemented . Putting either a 0 or a 1 at a specific point can be achieved by bit-shifting either a 0 or a 1 to the specified spot. Here | is the bitwise or-operator:

```
num_with_extra_bit = num|(bit_to_int(x.b1)<<1);
```

3 Implementing addition

Addition required some figuring out on pen and paper. We came up with the following combination of logical/boolean operators for calculating any bit and remainder:

```
bit = xor(xor(a, b), c);

remainder = or(or(and(a,b),and(a,c)),and(b,c));
#Checks if two or three of the bits are 'true'
#Probably not the most optimal implementation
```

Here a, b and c are the bits from the added numbers and any overflowing remainder from previous calculations.

Since we are emulating an old-fashioned physical logical circuit, the addition is done by stringing multiple of these together.

4 Implementing negation

As we are working with a two's-complement representation, negation of a number can be shown to simply require a negation the individual bits and then adding one. Negation of a bit can be done by xor'ing with 1.

```
flipped = xor(x,ones);
negated = add(flipped, one);
```

Here *xor* is an implementaion of bitwise xor, and *ones* is a byte purely consisting of ones.

5 Implementing multiplication

Multiplication was by far the most tricky concept to implement. We took inspiration from the following hint:

$$z = \sum_{i=0}^{k-1} x \cdot y_i \cdot 2^i \tag{1}$$

We will simply start out by showing the code, and we can break it down later:

```
result += bits8_shiftleft(bits8_and(bits8_fill(nth_bit_from_x),y),n
    );
```

Now, starting from the inside out. We start by using bits8_fill(bit) to create a new byte purely consisting of the n'th bit from the first number. We then use a bitwise *and* between this and the second number to either get the second

number or a byte of all zeros. In a simpler world this would have been an if-statement.

Now the number is leftwards bitshifted by n, as this corresponds to multiplying by 2 n times. This is then added onto a variable holding the result. So for every 1 in the first number, the second number is added along with the corresponding power of two - exactly as equation 1 shows.

Again, we are quite certain this is not the best implementation of multiplication, but it works!

6 This is a section to answer the following questions

6.1 Does bits8_add() provide "correct" results if you pass in negative numbers in two's complement representation? Why or why not?

Yes! Running tests by adding all permutations of positive and negative numbers (meaning all combinations of a specific number, both in its negative and positive form. e.g.: $1+1, 1+(-1), (-1)+1$ etc.) none of the tests failed. This is not surprising as addition of two's complement has the same bit-level representation as addition of unsigned integers.

6.2 Does bits8_mul() provide "correct" results if you pass in negative numbers in two's complement representation?

Again, testing multiplication by two positive, two negative and a positive and a negative number, every example gave the expected result. This again does not surprise as the truncated bit representations (after overflow has been removed) after multiplication has the same bit representation for both signed and unsigned integers.

6.3 How would you implement a function bits8_sub() for subtracting 8-bit numbers?

We implemented it as the following one-liner:

```
return bits8_add(x,bits8_negate(y));
```

Subtracting by a number is the same as adding the inverse. □