Version 1.1 ***due:*** *Mon, September 23 @ 11:59pm*

1 Introduction

The purpose of this assignment is to provide some background on algorithms used to perform arithmetic operations on integers based solely off binary representations. Your goal will be to implement addition, subtraction, and multiplication operations for arbitrary-length bit sequences.

2 Files

After downloading the assignment tarball from Autolab, extract the files from it by running `tar -xvf bignumlab-handout.tgz` from a terminal window. You should see the following files:

1. `bignumlab.pdf`
2. `submit`
3. `sources.cm`
4. `support.cm`
5. `lib/`
6. `BIGNUM.sig`
7. `MkBigNumUtil.sml`
8. `Tester.sml`
9. `* Tests.sml`
10. `* MkBigNumAdd.sml`
11. `* MkBigNumSubtract.sml`
12. `* MkBigNumMultiply.sml`

You should only modify the last 3 files, denoted by `*`. Additionally, you should create a file called `written.pdf` which contains the answers to the written part of the assignment.

3 Submission

Before submitting, ensure that `written.pdf` exists and is a valid PDF document. Then run `./submit` from within the `parenlab` directory. You *must* do this from a unix.andrew.cmu.edu machine.

Important note: You will still need to visit the Autolab website to view the results of the tests that Autolab runs against your code (which will contribute to part of your grade for this assignment). It is important to always do this for every lab to make sure that your code runs as you expect it to.

4 Bignum Arithmetic

In this problem, you will implement functions to support *bignum*, or arbitrary-precision arithmetic. Native hardware integer representations are typically limited to 32 or 64 bits, which is sometimes insufficient for computations which result in very large numbers. Some cryptography algorithms, for example, require the use of large primes which require over 500 bits to represent in binary. This motivates the implementation of an arbitrary-precision representation which can support such operations.

4.1 Logistics

We represent an integer with the type `bignum` which is defined as a `bit seq`, where

```
datatype bit = ZERO | ONE
```

We adopt the convention that if x is a `bignum`, then x is non-negative, and x_0 is the least-significant bit. Furthermore, if x represents the number 0, x is an empty sequence—and if $x > 0$, the right-most bit of x must be `ONE` (that is to say, there cannot be any trailing zeros following the most significant `ONE` bit). **You must follow this convention for your solutions.**

Our `bignum` implementation will support addition, subtraction (assuming the number never goes negative), and multiplication. The starter code already has the `bignum` type declared and the infix operators `**`, `--`, `++` defined for you in `MkBigNumUtil.sml`.

4.1.1 Testing

You are *not* required to submit test cases for this lab. However, you should test your own code to convince yourself of its correctness.

To aid with testing, we have provided a testing structure, `Tester`, which should simpify the testing process. `Tester` will look at the file `Tests.sml`, containing the lists `testsAdd`, `testsSub`, and `testsMul`, in which you should put put your test inputs for `++`, `--` and `**` respectively. Inputs should be specified as a tuple of `IntInfs`; some example test cases are included for guidance. Run the tests by entering the following at the REPL:

```
CM.make "sources.cm";
Tester.testAdd ();
Tester.testSub ();
Tester.testMul ();
```

Please note that the tester only has defined behavior over valid inputs to the functions being tested. That is to say, if you pass `--` a test case where you subtract the larger number from the smaller number, you will likely see a test failure.

Also note that our testing infrastructure allows you to independently test your `++`, `--` and `**`. This means that you may complete the tasks in any order you choose.

4.2 Specification

4.2.1 Addition

Task 4.1 (35%). Implement the addition function

```
++ :  bignum * bignum -> bignum
```

in the functor `MkBigNumAdd` in `MkBigNumAdd.sml`. For full credit, on input with m and n bits, your solution must have $O(m+n)$ work and $O(\lg(m+n))$ span. Our solution has under 40 lines with comments.

The main challenge in meeting the cost bound lies in propagating the carry bits. For example, try adding 1 to $(11101111111111)_2$ and you will see a "ripple effect." You should use `scan` to get around this, but you need to come up with an associative binary operator. As a hint, we have also provided you with an additional datatype which you may find useful:

```
datatype carry = GEN | PROP | STOP
```

where `GEN` stands for generating a carry, `PROP` for propagating a carry, and `STOP` for stopping a carry. You might want to work out a few small examples to understand what is happening. Do you see a pattern in the following example?

```
1000100011  +
1001101001
```

For more inspiration, you should try and recall how you utilized copy scan in order to solve the skyline problem.

4.2.2 Subtraction

Task 4.2 (15%). Implement the subtraction function

```
-- :  bignum * bignum -> bignum
```

in the functor `MkBigNumSubtract` in `MkBigNumSubtract.sml`, where `x -- y` computes the number obtained by subtracting y from x. We will assume that $x \geq y$; that is, the resulting number will always be non-negative. You should also assume for this problem that ++ has been implemented correctly. For full credit, if x has n bits, your solution must have $O(n)$ work and $O(\lg n)$ span. Our solution has fewer than 20 lines with comments.

Perhaps the easiest way to implement subtraction is to use *two's complement* representation for negation, which you should recall from 15-122 or 15-213. For a quick review: we can represent positive numbers in k bits from 0 to $2^{k-1}-1$, reserving the most significant bit as the "sign bit." For any integer x representable in k bits in two's complement, $-x$ is simply the number y such that $x + y = 2^k$. Then, we can negate x by simply flipping all the bits and adding 1.

4.2.3 Multiplication

Task 4.3 (30%). Implement the function

```
** :  bignum * bignum -> bignum
```

in `MkBigNumMultiply.sml`. For full credit, if the larger number has n bits, your solution must satisfy

$$W_{**}(n) = 3 \cdot W_{**}\left(\frac{n}{2}\right) + O(n)$$

and have $O(\lg^2 n)$ span. You should use the following function in the `Primitives` structure:

```
val par3 :  (unit -> 'a) * (unit -> 'b) * (unit -> 'c) -> 'a * 'b * 'c
```

to indicate three-way parallelism in your implementation of `**`. You should assume for this problem that `++` and `--` have been implemented correctly, and meet their work and span requirements. Our solution has 40 lines with comments.

Suppose we are given two n bit numbers. The first question to ask ourselves is, how we might divide up the problem? The first thing that comes to mind might be to proceed by dividing the numbers each into their most-significant half and their least-significant half:

$$A = p2^{n/2} + q \qquad A = \boxed{p}\boxed{q}$$
$$B = r2^{n/2} + s \qquad B = \boxed{r}\boxed{s}$$

So then, the product $A \cdot B$ is simply

$$A \cdot B = pr \cdot 2^n + (ps + rq) \cdot 2^{n/2} + qs$$

That is, to compute $A \cdot B$, we need to compute pr, ps, rq, qs—that's a total of 4 multiplication operations which can be done in parallel. The size of these numbers are also only $n/2$. In addition to this, we will need 2 shift operations and 3 adds. Notice that as we perform four recursive multiplies at this level, we have the recurrence

$$W_{**}(n) = 4 \cdot W_{**}\left(\frac{n}{2}\right) + 3 \cdot W_{++}(n) + O(n)$$

which solves to $O(n^2)$ assuming that $W_{++}(n) \in O(n)$. This is far too slow. Here's a hint:

$$(p+q)*(r+s) = pr + ps + rq + qs$$
$$ps + rq = (p+q)*(r+s) - pr - qs$$

Notice that by simple arithmetic manipulations, we may calculate all terms necessary to compute the product of A and B using only three multiplications.

5 Recurrences

Task 5.1 (15%). Determine the complexity of the following recurrences. Give tight Θ-bounds, and justify your steps to argue that your bound is correct. Recall that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$. You may use any method (brick method, tree method, or substitution) to show that your bound is correct, except that you must use the substitution method for problem 3.

1. $T(n) = 3T(n/2) + \Theta(n)$
2. $T(n) = 2T(n/4) + \Theta(\sqrt{n})$
3. $T(n) = 4T(n/4) + \Theta(\sqrt{n})$ (Prove by substitution.)