Lab 3 - *BignumLab*

Due: *Mon, Feb 10^{th}, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring ’14)

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 by running:

```
tar -xvf bignumlab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `support/MkBigNumUtil.sml`
3. * `MkBigNumAdd.sml`
4. * `MkBigNumSubtract.sml`
5. * `MkBigNumMultiply.sml`
6. * `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `bignumlab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the “*Handin your work*” link.

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.

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` (i.e. 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 Addition

We define the operator

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

where `x ++ y` computes the number obtained by adding x to y. Your task is to implement this function such that on input with m and n bits, your solution will have $O(m+n)$ work and $O(\lg(m+n))$ span. 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.

Task 4.1 (5%). Your associative function for scan should have the type

```
f :  (carry*carry) -> carry
```

Fill in the table below with the values that `f` should return when given some input `(A, B)`, where `A` is an element of the left column and `B` is an element of the top row.

For example, the output of `f(GEN, PROP)` should be in the second cell of the first empty row.

	GEN	PROP	STOP
GEN			
PROP			
STOP			

This table should be helpful in checking for associativity of your scan function.

You might also want to work out a few small examples to understand what is happening. For example, do you see a pattern in the following?

```
  1000100011
+ 1001101001
```

For more inspiration, you should try and recall how you utilized copy scan to solve The Pittsburgh Skyline Problem.

Task 4.2 (35%). Implement the addition function 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. For a sanity check, our solution has under 40 lines with comments.

4.2 Subtraction

Task 4.3 (15%). In `MkBigNumSubtract.sml`, implement the subtraction function

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

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 the *two's complement* representation for negation, which you should recall from 15-122 and 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.

4.3 Multiplication

Task 4.4 (30%). In `MkBigNumMultiply.sml`, implement the function

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

where `x ** y` computes the number obtained by multiplying x by y. For full credit, if the larger number has n bits, the work of 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 `par3` 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 should we divide up the problem? The first thing that comes to mind might be to proceed by dividing the numbers each into their most-significant and least-significant halves:

$$A = p2^{n/2} + q \qquad\qquad A = \begin{array}{|c|c|}\hline \quad p \quad & \quad q \quad \\ \hline\end{array}$$

$$B = r2^{n/2} + s \qquad\qquad B = \begin{array}{|c|c|}\hline \quad r \quad & \quad s \quad \\ \hline\end{array}$$

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 and 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 4 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 3 multiplications.

4.4 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 in `Tester.sml`, which should simplify 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 `IntInf` values (some example test cases are included for guidance). To run the tests:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
- Tester.testAdd ();
- Tester.testSub ();
- Tester.testMul ();
```

Please note that `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.

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 $g \in O(f)$. For 1 and 2, you may use any method (brick, tree, or substitution) to prove your bound correct; for 3, you must use the substitution method.

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