1 Introduction

In this assignment, you will implement arithmetic operations on a *bignum* type for natural numbers. Bignums aren't limited by the word size of a particular machine, allowing us to manipulate numbers which are arbitrarily large.

Your implementations and analyses in this lab will reinforce your understanding of a variety of algorithmic design techniques, including divide-and-conquer, reduction, and contraction.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

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

from a terminal window on a unix/linux system. 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 that will be handed in terms of code

1. * `MkBignumAdd.sml`
2. * `MkBignumSub.sml`
3. * `MkBignumMul.sml`
4. `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment. Your written answers should be composed in a suitable word processor such as LaTeX, Word, OpenOffice, etc. *Handwritten (and scanned in) answers will NOT be accepted.* **Make sure your name is also in the `written.pdf` file.**

3 Submission

To submit your assignment to Autolab in Diderot, generate a handin archive `handin.tgz` with the command

```
make package
```

You can then submit your `handin.tgz` file Diderot. You should submit the answers to the written questions in *written.pdf* to Gradescope at `https://www.gradescope.com/courses/225303`.
Note that your code submitted to Autolab will not be graded until after the due date. It is your responsibility to test your code thoroughly before submitting.

3.1 Important points

1. Before submission, always, *always* verify that the files you will submit are indeed the versions that you intended to submit. Be very careful to NOT submit the wrong lab's `written.pdf`, blank files, or old versions of the files. **We will not entertain "*Oh no, I submitted the wrong file!*" emails or Diderot posts after the due date**. We will adopt a *zero-tolerance* approach to this, and you will receive the grade of your most recent submission (which may be subject to late submission penalties!) *You have been warned*.

2. If the homework has a written component, the answers to each written question in *written.pdf* should be on one page and each answer should fit on one page. If you decide not to answer one of the questions, still have the empty page in the file. This will make everyone's life easier in Gradescope.

3. If the written part has a question that asks you to show that your code has a certain complexity bound, **you should include a copy of your code in your answer and annotate it as suitable with relevant complexity bounds using comments and then argue or compute the resulting bounds. Answers to such question without a copy of the code you submitted will not be accepted.**

4. The questions below ask you to organize your solutions in a number of modules written almost from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

5. You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

6. This assignment requires the use of library code spread over several files. The compilation of this is orchestrated by the compilation manager through several `.cm` files. Instructions on how to use CM can be found in the file `cm.pdf` under Resources on Diderot.

7. Style grading for this assignment will be evaluated on a 5 point scale: -2 through +2. Style points will be added to your final grade from the rest of your homework. **You should review the full style guide available on the Diderot under Resources.**

8. **You should submit your solutions on or before the deadline on the top of the first page. Failure to do so will cause substantial deductions of your grade. Late submissions of up to a day will scale your homework grade by 70% (so if you were to get 100% as your grade, you will now get 70%). Late submissions of up to 2 days will scale your homework grade by 50%. Beyond 2 days, you will receive 0 as your homework grade. This scaling will be handled outside Autolab.**

4 Bignums

Typical modern hardware limits integer representations to 32 or 64 bits, which is sometimes insufficient. For example, some cryptographic algorithms use large primes requiring in excess of 500 bits. This motivates an implementation of unbounded integers.

4.1 Representation and Logistics

The bignums in this lab are typed as `Bit.t Seq.t`, where a `Bit.t` is either `Bit.ZERO` or `Bit.ONE`. We use the following conventions:

- All bignums are non-negative.
- The bits of a bignum are written from *least-significant* to *most-significant*. For example, the decimal number 4 (a.k.a. $100_{[2]}$) is encoded as $\langle$`Bit.ZERO`, `Bit.ZERO`, `Bit.ONE`$\rangle$.
- Bignums do not have trailing zeros. The number 0 is encoded as $\langle\rangle$, and all other bignums have a `ONE` in the most-significant position.

There are a number of small utilities available to make your life easier. In all of the coding tasks below, you have access to the structure `Bignum` ascribing to `BIGNUM` (see `support/BIGNUM.sig`), the structure `Bit` (see `support/Bit.sml`), and all functions from previous tasks.
Finally, note that the functions you are writing in this lab are written in *infix* notation. This is a syntactic convenience which is supported in SML with `infix` declarations. You will see these near the top of the files in this assignment.

4.2 Trimming

When manipulating bignums, you might encounter a situation where you have a bit sequence with trailing zeroes. This violates our conventions for the representation of bignums. To help handle this scenario, there is a function in the `Bignum` structure which removes trailing zeroes:

```
val trim : Bignum.t → Bignum.t
```

If x is a bignum with bit width n, then assume the expression (`Bignum.trim` x) has $O(n)$ work and $O(\log n)$ span. You may see its implementation in `support/MkBitSeqBignum.sml`.

4.3 Arithmetic

Task 4.1 (45 pts). In `MkBignumAdd.sml`, implement the function

```
val ++ : Bignum.t * Bignum.t → Bignum.t
```

where $(x \texttt{ ++ } y)$ adds the bignums x and y. If x and y are respectively n and m bits wide, then your algorithm must have $O(n+m)$ work and $O(\log(n+m))$ span.

Here are some miscellaneous notes and hints:

- We recommend implementing an algorithm similar to the standard pen-and-paper algorithm, except that you propagate carry bits using at most logarithmic span. (Use `scan` or `scanIncl`!)
- Try a few small examples. Observe where particular carry bits are forced, and where carry bits are otherwise copied from the previous column.
- Our solution is a bit over 40 lines with comments.

Task 4.2 (15 pts). In `MkBignumSub.sml`, implement the function

```
val -- : Bignum.t * Bignum.t → Bignum.t
```

where $(x \texttt{ -- } y)$ subtracts y from x. Assume that $y \leq x$. Your algorithm must have $O(n)$ work and $O(\log n)$ span, where x is n bits wide.

Perhaps the easiest way to go about this problem is to use the *two's complement* method of negation, which you should recall from 15-122. A quick review: for any k-bit number a, we define $-a$ to be the number b such that $a + b = 2^k$, ignoring any final carry out. Note that modulo 2^k, this is the same as saying $a + b = 0$, and therefore $b = -a$. It turns out that you can acquire $-a$ simply by **flipping all of a's bits** and then **incrementing** the result, again ignoring any final carry out. For instance assuming we represent out numbers with $k = 4$ bits, $1_{10} = 0001_2$. So $-1_{10} = 1110_2 + 1_2 = 1111_2$. So $1_{10} + (-1)_{10} = 0001_2 + 1111_2 = 10000_2 = 0000_2$, ignoring the final (leftmost) carry out so we get a 4 bit representation.

As a sanity check, our solution is less than 10 lines. You should use ++ for this task, so assume it has been implemented correctly.

Task 4.3 (30 pts). In `MkBignumMul.sml`, implement the function

```
val ** : Bignum.t * Bignum.t → Bignum.t
```

where `(x ** y)` multiplies the bignums x and y. Your solution must be a *divide-and-conquer* solution and must satisfy the following work and span recurrences where n is the bit width of the larger of x and y.

$$W(n) = 3\,W\left(\frac{n}{2}\right) + O(n)$$
$$S(n) = S\left(\frac{n}{2}\right) + O(\log n)$$

Here is some mathematical background on the divide and conquer algorithm:
If x and y are both n-bit numbers, then we can split them into their most-significant (p and r) and least-significant (q and s) halves, where q and s both have a bit width of $\lfloor n/2 \rfloor$.

$$x = 2^{\lfloor n/2 \rfloor}p + q \qquad x = \boxed{p}\boxed{q}$$
$$y = 2^{\lfloor n/2 \rfloor}r + s \qquad y = \boxed{r}\boxed{s}$$

We then have

$$\begin{aligned} xy &= \left(2^{\lfloor n/2 \rfloor}p + q\right)\left(2^{\lfloor n/2 \rfloor}r + s\right) \\ &= 2^{2\lfloor n/2 \rfloor}pr + 2^{\lfloor n/2 \rfloor}(ps + qr) + qs \end{aligned}$$

which at first seems to require 4 recursive multiplications on numbers of half the original bit width. This doesn't satisfy the recurrence given above. Notice, however, that we can use a different process to calculate the middle term:

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

Plugging this into the expression above, we have the following, which requires only three recursive multiplications.

$$xy = 2^{2\lfloor n/2 \rfloor}pr + 2^{\lfloor n/2 \rfloor}\big((p+q)(r+s) - pr - qs\big) + qs$$

This is known as the *Karatsuba* Algorithm, after Anatoly Karatsuba. This is not asymptotically the fastest algorithm for multiplying big numbers – there are few others. For example, an algorithm known as Schönhage-Strassen algorithm runs in work $O(n \log n \log\log n)$ but it outperforms Karatsuba only after numbers with several 10s of thousands of digits (i.e., numbers with values in the range roughly $2^{2^{15}}$ to $2^{2^{17}}$.)

Some miscellaneous notes and hints:

- You should use the function `Primitives.par3` to indicate 3-way parallelism:

  ```
  val par3 : (unit → α) * (unit → β) * (unit → γ) → α * β * γ
  ```

- You should use both ++ and −−, so assume they were implemented correctly. Oh, and don't forget about `Bignum.trim`, either. (See section 4.2 for its description.)

- Our solution is a bit over 30 lines with comments.

5 Testing

There are two ways to test your code.

1. In `Sandbox.sml`, write whatever testing code you'd like. You can then access the sandbox at the REPL:

```
- CM.make "sandbox.cm";
...
- open Sandbox;
```

2. In `Tests.sml`, add test cases according to the instructions given. Then run the autograder:

```
- CM.make "autograder.cm";
...
- Autograder.run ();
```

As usual, it is very important that you thoroughly test your code before you submit. After the deadline, we will grade your code with a private set of test cases.

6 Analysis

6.1 Karatsuba

Task 6.1 (10 pts). For each of the following recurrences, state whether the recurrence is leaf-dominated, root-dominated, or balanced. Give tight big-O bounds for each recurrence.

$$W(n) = 3\,W\left(\frac{n}{2}\right) + O(n)$$
$$S(n) = S\left(\frac{n}{2}\right) + O(\log n)$$

6.2 Bignum Widths

Note that we need $1 + \lfloor \log_2 x \rfloor$ bits to represent a number $x \geq 1$ in binary.

Task 6.2 (5 pts). Consider 2 bignums, each with bit width at most w. Carefully show that their sum has a bit width of at most $1 + w$.

Task 6.3 (10 pts). Consider k bignums, each with bit width at most w. Carefully show that their sum has a bit width of at most $1 + w + \lfloor \log_2 k \rfloor$.

6.3 The Cost of `scan`

For the sake of readability in this section, we write $+\!\!+$ for bignum addition and $\overline{x}$ for a bignum of value x. Consider a sequence S of n bignums, each of which has bit width upper bounded by m. We are interested in analyzing the work of $(\texttt{scan}\ +\!\!+\ \overline{0}\ S)$. Specifically, we want to show that the work of this expression satisfies the following recurrence, which you will then solve for a tight Big-O bound.

$$W(n, m) = W(n/2, m+1) + O(nm + n \log n)$$

To simplify the problem, let's assume n is a power of 2. Below is a modification of the code for `scan` from lecture, where we've replaced the associative function and identity with $+\!\!+$ and $\overline{0}$, respectively.

```
1  % Assume |S| is a power of 2
2  fun bignumAdditionScan S =
3    case |S| of
4      0 ⇒ (⟨ ⟩, 0̄)
5    | 1 ⇒ (⟨0̄⟩, S[0])
6    | n ⇒
7        let
8          val S' = ⟨S[2i] ++ S[2i+1] : 0 ≤ i < n/2⟩
9          val (R, t) = bignumAdditionScan S'
10         fun P(i) = if even(i) then R[i/2] else R[⌊i/2⌋] ++ S[i−1]
11       in
12         (⟨P(i) : 0 ≤ i < n⟩, t)
13       end
```

Task 6.4 (10 pts). Show that the *contraction* step (line 8) requires $O(nm)$ work.

Task 6.5 (5 pts). Argue that the *recursive* step (line 9) requires $W(n/2, m+1)$ work.

Task 6.6 (10 pts). Upper bound the bit width of $R[i]$ in terms of m and i. Give an exact bound; don't use asymptotics.

Task 6.7 (15 pts). Show that the *expansion* step (line 12) requires $O(nm + n\log n)$ work.

Task 6.8 (10 pts). It follows from the previous tasks that the work of (`scan` $+\!\!+$ $\overline{0}$ S) is given by the recurrence

$$W(n,m) = W(n/2, m+1) + O(nm + n\log n).$$

Solve this recurrence, giving a tight Big-O bound in terms of n and m.