Big Multiplication Assignment

M. Andrew Moshier

February 19, 2021

The main task for this assignment is to implement a big natural number multiplication following the prototype

```
void bigmul64(uint64_t a[], int sz_a, uint64_t b[], sz_b, uint64_t c[], int sz_c);
```

The function should calculate $a = b \cdot c$ in the sense we have beern discussing in class (some details follow). The sizes of the three arrays should satisfy `sz_a` $\geq$ `sz_b` $+$ `sz_c`. There are other parts to the assignment discribed below. Read carefully.

1 *Big natural numbers: Theoretical Background*

I will avoid integers in this assignment. All values will be natural numbers. A big natural number can stored in a array of `uint64_t` values. The idea is that a `uint64_t` value x represents a natural number $x < 2^{64}$. To avoid clutter in our notation, let $B = 2^{26}$. So a big natural number stored in an array b of size n_b represents a value

I don't need to say $0 \leq x$ here because all values are non-negative.

$$\sum_{i<n_b} b[i] \cdot B^i < B^{n_b}$$

where each $b[i]$ is a 'digit'. More precisely, $b[i] < B$ for each $i < n_b$. So if we are given two such arrays (b and c), the product is

$$\left(\sum_{i<n_b} b[i] \cdot B^i\right) \cdot \left(\sum_{j<n_c} c[j] \cdot B^j\right) = \sum_{i<n_b} \sum_{j<n_c} b[i] \cdot c[j] \cdot B^{i+j}.$$

Though the expression on the right is mathematically correct, it does not represent the result as an array of `uint64_t` values. So the basic goal of your algorithm is to rearrange this sum to determine a `uint64_t` array a of size n_a so that

$$\sum_{k<n_a} a[k] \cdot B^k = \sum_{i<n_b} \sum_{j<n_c} b[i] \cdot c[j] \cdot B^{i+j}.$$

and $a[k] < B$ for each $k < n_b$. For this to be possible, the array sizes must be related by $n_a \geq n_b + n_c$.

2 *Half-word work-around*

A significant problem is that $\texttt{b}[i] \cdot \texttt{c}[i]$ is generally too big to fit into a single `uint64_t`. We only can be sure that $\texttt{b}[i] \cdot \texttt{c}[j] < B^2$. And C does not directly support correct multiplication of results exceeding B. So a work-around is to think of our arrays as consisting of `uint32_t` values instead.

I assume your machine has a 64bit architecture.

This can be done in C with no run-time overhead by directing the compiler to interpret a `uint64_t` array as a `uint32_t` array. This can be done as follows.

```
uint64_t a_64[]; // an array
const   uint32_t *a_32 = (uint32_t *) a64;
```

This declares `a_64` and `a_32` to refer to exactly the same location in memory. But $\texttt{a\_32}[i]$ is the i^{th} 32-bit entry. Let $C = 2^{32}$. Then arithmetically, the relation between `a_64` and `a_32` is summarized by

$$\sum_{i<n_a} \texttt{a\_64}[i] \cdot B^i = \sum_{k<2n_a} \texttt{a\_32}[k] C^k.$$

3 *Addition*

The 32 bit addition algorithm we discussed in class is this:

```
uint32_t addto32(uint32_t as[], int sz_a, uint32_t bs[], int sz_b) {
// Assume that sz_b <= sz_a
// Compute as += bs

int i;
uint32_t c = 0;
uint64_t s;
for (i=0; i< sz_b; i++) {
s = (uint64_t) as[i] + (uint64_t) bs[i] + (uint64_t) c; // s is a 33 bit value
c = s >> 32;
as[i] = (uint32_t) s;
}
for ( ; i< sz_a; i++) {
s = (uint64_t) as[i] + (uint64_t) c; // s is a 33 bit value
c = s >> 32;
as[i] = (uint32_t) s;
}
return c;
}
```

Assignment Part 1

Sketch a proof that this algorithm is correct. Specifically, show that at the beginning of each loop the invarant

$$\mathtt{c} + \sum_{k<i} \mathtt{as}[k] = \sum_{k<i} (\mathtt{as}'[k] + \mathtt{bs}[k])$$

where $\mathtt{as}'$ denotes the original value of `as`. [Hint: When $i = 0$, the two sums are empty. So they both equal 0.] Your proof sketch does not need to be completely formal. The important thing is to reason about what happens step-by-step in the loop bodies.

4 *Partial products*

If you could to use the `addto32` algorithm as-is to implement multiplication. But that leads to an inefficency caused by all the potential carry propagations. Some of that could be mitigated by changing the stopping condition on the second loop. After all, if the carry ever becomes 0, the second loop just adds 0 to each word from then on. That's a waste. But this only helps if carries do not propagate. In your elementary school algorithm, you do not add a whole bunch of single digit products. Instead to multiply $a = \mathtt{b} \cdot \mathtt{c}$ (in any base) to multiply all of `b` times one digit in `c`, adding that to the final result in the correct column. This is what we usual call a partial product.

Assignment Part 2

Modify the `addto32` code to calculate a partial product and add it to an existing array. The prototype should be

```
void partialprod32(uint32_t as[], int sz_a, uint32_t bs[], int sz_b, uint32_t d);
```

The code should compute $\mathtt{as}+=\mathtt{bs} * \mathtt{d}$. Work out the constraints on sizes and on `as` that ensure that the result will not overflow — a carry would not propagated out of the last addition. Modify the proof of Part 1 to prove that your implementation is correct.

5 *Products*

Finally, we reach the goal.

Assignment Part 3

Implement big natural number multiplication using `partialprod32`, and sketch a proof that it is correct. The proof should not be very difficult since you have a proof for `partialprod32`. The protoype for multiplication shoud be

```
void bigmul64(uint64_t a[], int sz_a, uint64_t b[], sz_b, uint64_t c[], int sz_c);
```

Note that you need to "convert" between `uint64_t` arrays and `uint32_t` arrays as mentioned above. Then the key idea is to figure out how to combine partial products in the correct "columnns".

6 Work submission

Implement your code in C in a *public* repository on github. You will submit the url for that repository. Simple as that.