

This short article provides a divide and conquer multiplication algorithm for fixed length binary, specifically proving (validating) some lemmas for an efficient implementation.

Although examples themselves are never proofs, they do make things clearer, so our prototypical multiplication is as follows:

$$\begin{array}{rr} & 1001\ 1101 \\ \times & 1111\ 0010 \\ \hline & 1\ 0011\ 1010 \\ & 1001\ 1101\ 0000 \\ & 1\ 0011\ 1010\ 0000 \\ & 10\ 0111\ 0100\ 0000 \\ + & 100\ 1110\ 1000\ 0000 \\ \hline & 1001\ 0100\ 0110\ 1010 \end{array}$$

The intention here is to implement an efficient algorithm for standard hardware constraints (computing science). So in the above we're multiplying two 8-bit unsigned integers. Let's say an 8-bit block is the most our hardware (registers) can handle, in that case if using the builtin cpu multiplication you can't prevent an overflow.

The implementation of multiplication to handle this potential overflow relies almost entirely on the following mathematical property of integers (real numbers actually):

**Lemma 0.1**

$$0 \leq a, b < c \quad \Longrightarrow \quad ab < c^2$$

First note if $a = 0$ or $b = 0$ then this trivially holds. Otherwise the proof relies on the fact that if $d < e$ and $f > 0$ then $fd < fe$. It's pretty basic undergrad math, but let's go over it anyway, starting with the simple derivation:

$$\begin{array}{rrcl} & a & < & c \\ & 0 & < & b \\ \Longrightarrow & ab & < & bc \end{array}$$

With the same logic, we also have:

$$\begin{array}{rrcl} & b & < & c \\ & 0 & < & c \\ \Longrightarrow & bc & < & c^2 \end{array}$$

Putting these together we get:

$$\begin{array}{rrcccl} & ab & < & bc & < & c^2 \\ \Longrightarrow & ab & & < & & c^2 \end{array}$$

This result is entirely useful toward *positional (radix) notation* which are algorithm will be exploiting. I'm not giving the full algorithm here, but only the main lemma to correct for arithmetic overflow.

To begin, our lemma tells us we need at most a 16-bit block (or two 8-bit blocks) to do our multiplication: Let $m, n$ be two 8-bit blocks, then $m, n < 2^8$ and so $mn < 2^{16}$.

As for divide and conquer, we split each 8-bit block into two 4-bit blocks. Why? First note that multiplying two 4-bit block requires at most an 8-bit block without having to worry about overflow. This matters because the best optimized implementation isn't necessarily the most efficient algorithm. Making use of hardcoded (hardware circuits) is almost always faster than softcoded (software) approaches when possible. This is to say, it's better to use the hardware multiplication algorithm as much as you can rather than reimplementing an efficient software simulation of those same circuits—even if the software algorithm becomes a little bit more complicated.

You can also consider this divide and conquer approach related to positional notation. Let $m = m_1|m_0$ and $n = n_1|n_0$ be the respective 4-bit blocks with '|' the concatenation operator, then:

$$mn \quad = \quad (m_1|m_0)(n_1|n_0) \quad = \quad (m_1 2^4 + m_0)(n_1 2^4 + n_0) \quad = \quad m_1 n_1 2^8 + (m_1 n_0 + m_0 n_1)2^4 + m_0 n_0$$

In computing science, multiplying by powers of 2 is equivalent to shifting (<<). Let's let our resultant value equal $p := p_1 p_0$, where $p$ is 16-bit and $p_0, p_1$ are 8-bit blocks. Our resulting value then is algorithmically determined as follows:

1. assign $p_0 := m_0 n_0$.
2. assign $p_1 := m_1 n_1$.
3. Take the left shift of $m_1 n_0 + m_0 n_1$ by half the bit length, and accumulate (add) to $p_0$:

$$p_0 \mathrel{+}= (m_1 n_0 + m_0 n_1) << 4$$

4. Take the right shift of $m_1 n_0 + m_0 n_1$ by half the bit length, and accumulate (add) to $p_1$:

$$p_1 \mathrel{+}= (m_1 n_0 + m_0 n_1) >> 4$$

That's the general pseudo-code of the matter.

This algorithm is incomplete though, as we have not considered the case for arithmetic overflow caused by addition. Here there are two possible instances:

We know that both $m_1 n_0,\ m_0 n_1 < 2^8$ (8-bit), but adding them together might not be. An 8-bit unsigned can be at most $2^8 - 1$, and so $(2^8 - 1) + (2^8 - 1) = 2^9 - 2 < 2^9$. This is to say an overflow can be at most 1-bit more than our 8-bit blocks, so we have a pleasant upperbound, but we still need to account for such possible *carries*. If there is a carry, notice in step (4) of our pseudo-algorithm where we right shift: During that process we could shift it in before shifting in the remaining zeroes. This also means we wouldn't have to specially add it separately to $p_1$ later on.

We also need to consider the other carry for step (3) of our pseudo-algorithm: Addition there might carry which would be pushed along to $p_1$.

With our special considerations of arithmetic overflow caused by addition, you might ask the best way to determine this. The article "addition.pdf" (or "addition.tex") provides those details. Also, you might think there's a third possible carry I didn't mention when we add $p_1$ with the split shift, but it was stated above that $ab < c^2$ meaning we know in advance that it's impossible in this special and specific context.

Finally, although we discussed the specific 8-bit and 16-bit lengths, it's a natural generalization all of this to arbitrary bit lengths.