One of the biggest and slowest circuits in an arithmetic and logic unit is the multiplier.

We'll start by developing a straightforward implementation and then, in the next section, look into tradeoffs to make it either smaller or faster.

Here's the multiplication operation for two unsigned 4-bit operands broken down into its component operations.

This is exactly how we learned to do it in primary school.

We take each digit of the multiplier (the B operand) and use our memorized multiplication tables to multiply it with each digit of the multiplicand (the A operand), dealing with any carries into the next column as we process the multiplicand right-to-left.

The output from this step is called a partial product and then we repeat the step for the remaining bits of the multiplier.

Each partial product is shifted one digit to the left, reflecting the increasing weight of the multiplier digits.

In our case the digits are just single bits, i.e., they're 0 or 1 and the multiplication table is pretty simple!

In fact, the 1-bit-by-1-bit binary multiplication circuit is just a 2-input AND gate.

And look Mom, no carries!

The partial products are N bits wide since there are no carries.

If the multiplier has M bits, there will be M partial products.

And when we add the partial products together, we'll get an N+M bit result if we account for the possible carry-out from the high-order bit.

The easy part of the multiplication is forming the partial products - it just requires some AND gates.

The more expensive operation is adding together the M N-bit partial products.

Here's the schematic for the combinational logic needed to implement the 4x4 multiplication, which would be easy to extend for larger multipliers (we'd need more rows) or larger multiplicands (we'd need more columns).

The M*N 2-input AND gates compute the bits of the M partial products.

The adder modules add the current row's partial product with the sum of the partial products from the earlier rows.

Actually there are two types of adder modules.

The full adder is used when the modules needs three inputs.

The simpler half adder is used when only two inputs are needed.

The longest path through this circuit takes a moment to figure out.

Information is always moving either down a row or left to the adjacent column.

Since there are M rows and, in any particular row, N columns, there are at most N+M modules along any path from input to output.

So the latency is order N, since M and N differ by just some constant factor.

Since this is a combinational circuit, the throughput is just 1/latency.

And the total amount of hardware is order N^2.

In the next section, we'll investigate how to reduce the hardware costs, or, separately, how to increase the throughput.

But before we do that, let's take a moment to see how the circuit would change if the operands were two's complement integers instead of unsigned integers.

With a two's complement multiplier and multiplicand, the high-order bit of each has negative weight.

So when adding together the partial products, we'll need to sign-extend each of the N-bit partial products to the full N+M-bit width of the addition.

This will ensure that a negative partial product is properly treated when doing the addition.

And, of course, since the high-order bit of the multiplier has a negative weight, we'd subtract instead of add the last partial product.

Now for the clever bit.

We'll add 1's to various of the columns and then subtract them later, with the goal of eliminating all the extra additions caused by the sign-extension.

We'll also rewrite the subtraction of the last partial product as first complementing the partial product and then adding 1.

This is all a bit mysterious but… Here in step 3 we see the effect of all the step 2 machinations.

Let's look at the high order bit of the first partial product X3Y0.

If that partial product is non-negative, X3Y0 is a 0, so all the sign-extension bits are 0 and can be removed.

The effect of adding a 1 in that position is to simply complement X3Y0.

On the other hand, if that partial product is negative, X3Y0 is 1, and all the sign-extension bits are 1.

Now when we add a 1 in that position, we complement the X3Y0 bit back to 0, but we also get a carry-out.

When that's added to the first sign-extension bit (which is itself a 1), we get zero with another carry-out.

And so on, with all the sign-extension bits eventually getting flipped to 0 as the carry ripples to the end.

Again the net effect of adding a 1 in that position is to simply complement X3Y0.

We do the same for all the other sign-extended partial products, leaving us with the results shown here.

In the final step we do a bit of arithmetic on the remaining constants to end up with this table of work to be done.

Somewhat to our surprise, this isn't much different than the original table for the unsigned multiplication.

There are a few partial product bits that need to be complemented, and two 1-bits that need to be added to particular columns.

The resulting circuitry is shown here.

We've changed some of the AND gates to NAND gates to perform the necessary complements.

And we've changed the logic necessary to deal with the two 1-bits that needed to be added in.

The colored elements show the changes made from the original unsigned multiplier circuitry.

Basically, the circuit for multiplying two's complement operands has the same latency, throughput and hardware costs as the original circuitry.