Integer multiplication

Suppose we have two unsigned integers, A and B, and we wish to compute their product. Let A be the multiplicand and B the multiplier:

$$
\begin{array}{llll}
& & A_{n-1}\ldots A_1 A_0 & \text{multiplicand} \\
& \times & B_{n-1}\ldots B_1 B_0 & \text{multiplier} \\
P_{2n-1} & \ldots & P_{n-1}\ldots P_1 P_0 & \text{product}
\end{array}
$$

If the multiplier and multiplicand each have n bits, then the product has at most 2n bits, since

$$(2^n - 1) * (2^n - 1) < 2^{2n} - 1.$$

For example, if we are multiplying two 32 bit unsigned integers, then we may need as many as 64 bits to represent the product.

Recall your grade school algorithm for multiplying two numbers. We can use the same algorithm if the numbers are written in binary. The reason is that shifting a binary number one bit to the left is the same as multiplying by 2, just as in decimal multiplying by ten shifts digits left by one. In the case of binary, each 1 bit b_i in the multiplier means we have a contribution 2^i times the multiplicand, and so we add the shifted multiplicands.

$$
\begin{array}{r}
1001101 \\
\times\ 0010111 \\
\hline
1001101 \\
10011010 \\
100110100 \\
0000000000 \\
10011010000 \\
000000000000 \\
+0000000000000 \\
\hline
\text{product}
\end{array}
$$

In grade school algorithm, we compute all the left shifts in advance, building up several rows of numbers. Then, this set of numbers is added together to yield the product (as above).

Let's consider an alternative algorithm that avoids building a big circuit to adds the n numbers. (We only consider the binary case, but the same idea would work for decimal too.) The algorithm generates the rows one by one, shifting the multiplicand at each step. When bit B_i of the multiplier is 1, the shifted multiplicand is added to an accumulated total. When bit B_i is 0, nothing is added to the total.

What hardware do we need to multiply two $n = 32$ bit unsigned numbers?

- shift registers: the multiplicand is shifted left. If we are multiplying two 32 bit numbers then we use a 64 bit shift left register for the multiplicand. (We shift the multiplicand to the left at most 32 times.)

- a 64 bit adder

- a 32 bit register for the multiplier. We use a shift right register, and read the least significant bit (LSB, i.e. bit 0) to decide whether to add the shifted multiplicand on each step or not.

- a counter to keep track of how many times we've shifted.

"Algorithm" for multiplying two unsigned integers:

```
//
// Let P, A, B be the product, multiplicand, and multiplier registers.
//
     P = 0
     load A so lower 32 bits contain multiplicand (upper 32 bits are 0)
     load B so it contains multiplier (32 bits)
     for counter = 0 to 31 {
          if (LSB of B is 1)
                P := P + A
          endif
          shift A left by one bit
          shift B right by one bit
     }
```
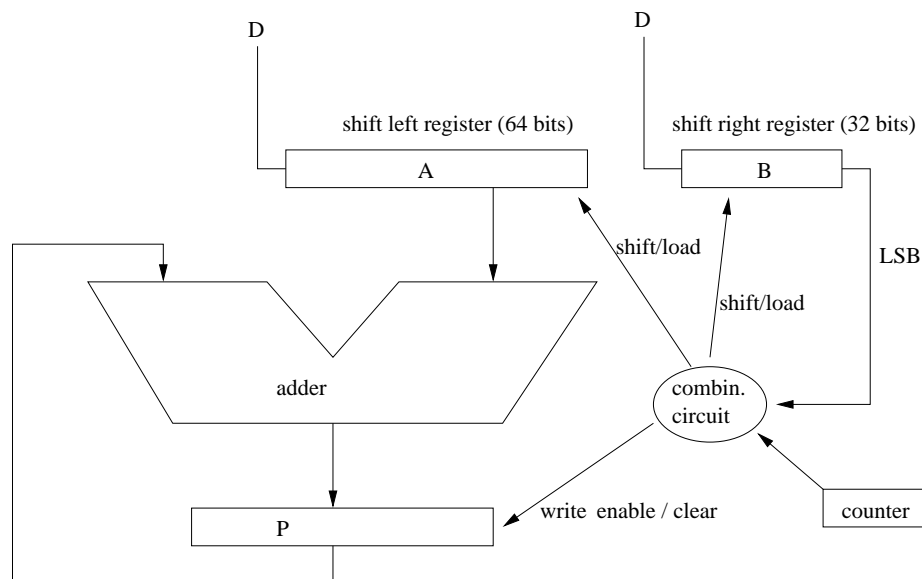

We must provide a control signal to the shift registers telling them whether to shift or not at each clock cycle. For example, on the first clock cycle of the instruction, we would load the multiplier and multiplicand registers, rather than shifting the values in these registers. On subsequent clock cycles, we would perform the shift.

Similarly, we must provide a signal to the product register telling it whether we should write in the value coming out of the ALU, or whether we should clear the product register. This choice would depend on whether we are in the register initalization phase and (if not in that phase) whether the LSB of the B register is 1.

Notice that, within the "for loop", all these instructions can be performed in parallel. At the end of a given clock cycle, we update the value of all three registers P, A, B.

Fast integer multiplication (sketch only – <u>not on exam</u>)

In the lecture notes, I sketched out an alternative scheme which performs n-bit multiplication quickly. This scheme first performs $\frac{n}{2}$ additions in $\log n$ steps rather than n steps. The idea is to take the n rows defined by the grade school multiplication algorithm (see page 1) and use $\frac{n}{2}$ fast adders to add rows 1 and 2, 3 and 4, 5 and 6, etc. Then the outputs from these adders could be paired and run through $\frac{n}{4}$ adders which would give the sum of rows 1 to 4, 5 to 8, etc. The bits would need to be shifted appropriately in doing so, but this does require shift registers; it just requires that certain output lines be fed into the correct input lines, and 0's be fed in where appropriate.

In principle the multiplication can be done in one clock cycle, using only combinational circuits. However, the clock cycle would need to be long to ensure that the all signals (carries, etc) have propagated through this circuit. Since the same clock is used for all gates in the processor (including the ALU) this means we would need a slow clock, which is not what we want.

To avoid the slow clock requirement, it is common to break the multiplication into stages and have a multiplication run in multiple clock cycles. This can be done by using registers to hold intermediate results. In particular, the $\frac{n}{2}$ sums could be written to $\frac{n}{2}$ registers at the end of the multiplication's first clock cycle. The next $\frac{n}{4}$ sums could be written to $\frac{n}{4}$ registers at the end of the second clock cycle. And $\log n$ clock cycles would be needed in total to compute the multiplication. This is still faster than the grade school algorithm which took n steps.

Integer division

When we perform integer division, say 78/21, or more generally "dividend/divisor", the result is two numbers, namely a quotient and a remainder.

$$78 = 3 * 21 + 15$$

$$\text{dividend} = \text{quotient} * \text{divisor} + \text{remainder}$$

Suppose we have two 32 bit binary numbers: a dividend and a divisor. Suppose they are both positive numbers. How do we compute the quotient and the remainder ? As with multiplication, we can apply the "long division" algorithm we learned in grade school.

Here is a more precise description of the "algorithm". (Do not memorize this for the exam. But make sure you understand it.) As with multiplication, we need a set of shift registers. See slides for a rough sketch of the corresponding circuit.

<u>"Algorithm" for long division:</u>
       divisor := 2^n * divisor
       quotient := 0
       remainder := dividend
       for i = 0 to n
             shift quotient left by one bit
             if (divisor $\leq$ remainder)
                  set LSB of quotient to 1
                  remainder := remainder - divisor
             shift divisor to right

A few notes: First, the first line of this algorithm is just the usual first step in long division of placing the divisor to the left of the dividend. When you carry out long division by hand in base 10, you don't think of this as multiplying by 10^n where n is the highest power of the dividend. But this is essentially what you *are* doing. Second, the comparison "divisor ≤ remainder" can be done by computing the difference (divisor - remainder) and checking whether the result is less than zero. In a twos complement representation, this can be done by checking if the MSB (most significant bit) is 1.

Floating point addition

Let's now consider floating point operations: +,-,*,/. We begin with addition. Suppose we want to add two single precision floating point numbers, x and y.

$$
\begin{aligned}
x &= 1.10101000000000000000101 * 2^{-3} \\
y &= 1.00100100001010101010101 * 2^{2}
\end{aligned}
$$

where x has a smaller exponent than y. We cannot just add the significands bit by bit, because then we would be adding terms with different powers of 2. Instead, we need to rewrite one of the two numbers such that the exponents agree. The usual way to do this is to rewrite the smaller of the two numbers, that is, rewrite the number with the smaller exponent. So, for the above example, we rewrite x so that it has the same exponent as y:

$$
\begin{aligned}
x &= 1.10101000000000000000101 * 2^{-3} \\
&= 1.10101000000000000000101 * 2^{-5} * 2^{2} \\
&= 0.00001101010000000000000000101 * 2^{2}
\end{aligned}
$$

Notice that x is no longer normalized, that is, it does not have a 1 to the left of the binary point.

Now that x and y have the same exponent, we can add them.

$$
\begin{array}{rl}
& 1.00100100001010101010101 * 2^{2} \\
+ & \underline{0.00001101010000000000000000101 * 2^{2}} \\
= & 1.00110001011010101010101000101 * 2^{2}
\end{array}
$$

The sum is a normalized number. But the significand is more than 23 bits. If we want to represent it as single precision float, then we must truncate (or round off, or round up) the extra bits.

It can also happen that the sum of two floats is not normalized. If, instead of y, we had

$$
z = 1.11111000000000000000000 * 2^{2}
$$

then check for yourself that $x + z$ would not be normalized. So, to represent the result as single precision float, we would have to normalize again by shifting (right) the significand and truncating, and changing the exponent by adding to it size of the shift.

Floating point subtraction

How can you do subtraction of two single precision floats? Recall from our discussion of integers that you do subtraction of two integers, say $i - j$ by taking the twos complement of i and adding $i + (-j)$.

Using twos complement works for addition on floats as well, but you need to be careful because the leftmost bit of a normalized number is 1, but this doesn't *not* mean that the number is negative (as it does in the case of signed integers). Rather, you need to consider the fact that

$$y = 1.10101000000000000000101 * 2^{-3}$$

is really the number

$$y = \underline{\ldots 0001}.10101000000000000000101 * 2^{-3}$$

To take the negative of this number, we just need to find out what number to add to it to get zero. But the idea here is exactly as it was for the integers. Take the complement of each bit, and then add 1 to the least significant bit. That is,

$$-y = \underline{\ldots 1110}.01010111111111111111011 * 2^{-3}$$

Now notice that there are infinitely many upper bits which are 1's.

Also notice that when we subtract one number from another, the result can be quite small (if the two numbers are near identical) so many shifts may be needed.

Floating point multiplication and division

Multiplying two floating point numbers is easy. We reduce the problem to integer multiplication, plus some bookkeeping with the exponents. To see the idea, suppose we work with 3 bits of significand instead of 23. Here is an example of how it goes.

$$x_1 = 1.101 * 2^{-7} = 1101 * 2^{-10}$$

$$x_2 = 1.001 * 2^4 = 1001 * 2^1$$

Since $(1101)_{two} * (1001)_{two} = 13 * 9 = 117 = (1110101)_{two}$, we get

$$x_1 * x_2 = 1110101 * 2^{-9} = 1.110101 * 2^{-3}.$$

With 23 bits of significand, we multiply each float by 2^{23} 2^{-23}. The product with the 2^{23} term moves the binary point to the right of the significand, giving us an integer. For example,

$$
\begin{aligned}
x &= 1.10101000000000000000101 * 2^{-3} \\
&= 110101000000000000000101 * 2^{-26} \\
y &= 1.00100100001010101010110 * 2^2 \\
&= 100100100001010101010110 * 2^{-21}
\end{aligned}
$$

We then multiply two 24 bit integers (24, not 23, since there is the leading 1 that needs to be included), and add the exponents. Finally, we normalize.

What circuitry do we need to perform a multiplication? We need circuitry for carrying out (unsigned) integer multiplications. We also needs circuitry for adding exponents and adding values (23) to exponents.

So you should be able to appreciate the floaing point multiplication is more complicated than integer multiplication, but not much so.

What about floating point addition? If we want to compute x/y, we can perform the same trick as above and reduce the problem to the division of two integers, times an exponent. So division of two floating points requires slightly more circuitry (and about the same amount of time) as integer division.

Overflow and underflow

The result of a floating point operation is typically an approximation since the result has more significant bits than the original operands and so roundoff is necessary. Worse yet, the result may not be representable as a normalized number. When multiplying two large floats which are normalized numbers, it can easily happen that the exponent of the product is greater than 127 which is the maximum exponent for single precision. Such a situation is called *overflow*. Similarly, if we are multiplying two small numbers, then the exponent of the product can easily be less than -126, and so the result also cannot be represented as single precision float. This situation is called *underflow*. Both overflow and underflow need to be detected, so there is additional circuitry for this. (We have seen a similar circuitry for integer addition – See Exercises 2 Question 11.)