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:

		$A_{n-1} \ldots A_1 A_0$	multiplicand
	$\times$	$B_{n-1} \ldots B_1 B_0$	multiplier
P_{2n-1}	$\ldots$	$P_{n-1} \ldots P_1 P_0$	product

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

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

which you can verify for yourself by multiplying out the left side. 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 decimal multiplication 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.

```
      1001101
    × 0010111
  ------------
      1001101
     10011010
    100110100
   0000000000
  10011010000
 000000000000
+0000000000000
  ------------
      product
```

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

Let's consider an alternative algorithm that avoids writing out all the rows with the shifted multiplicands. Instead the algorithm shifting the multiplicand (left i.e. multiply by 2) at each step i. 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 could we use to multiply two $n = 32$ bit unsigned numbers?

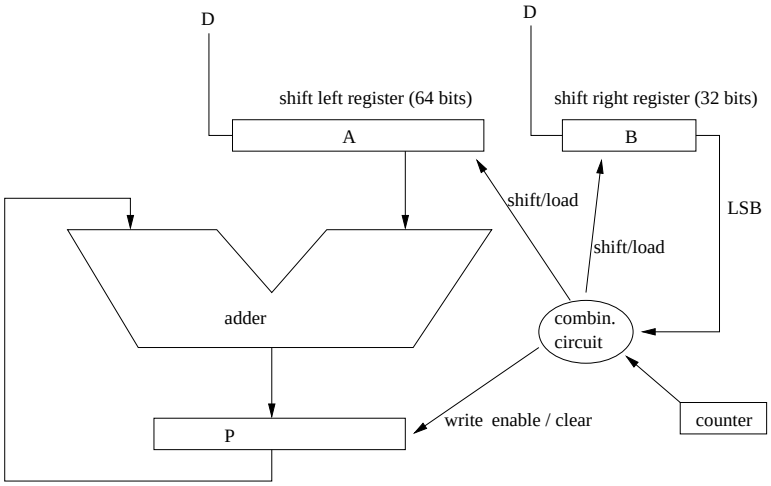
- shift registers: the multiplicand is shifted left. If we are multiplying two 32 bit numbers then we use a 64 bit shift left register for the multiplicand. (We shift the multiplicand to the left at most 32 times.)

- a 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 64 bit adder
- a 64 bit register which accumulates the product
- 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. On the first clock cycle, we would load the multiplier and multiplicand registers, rather than shifting the values in these registers. (Where would we load them from? We would load them from some other registers that are not drawn in the circuit.) On subsequent clock cycles, we would perform the shift.

Similarly, we must provide a signal to the product register telling it whether we should clear the product register (only on the first clock cycle), or whether we should write in the value coming out of the ALU (only on subsequent clock cycles and only when the LSB of the B register is 1.

Finally, notice that all the instructions within the "for loop" can be performed in parallel within a clock cycle.

Fast integer multiplication (log n argument is not on exam)

In the lecture, I briefly sketched out an alternative scheme which performs n-bit multiplication more quickly. The idea will be familiar to most of you, namely to those who have taken COMP 250. 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 all need to be aligned properly as in the table on page 1, but this doesn't need 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. Those of you who have already done COMP 250 know that $\log n$ 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 (sketch only – not on exams)

When we perform integer division on two positive integers, e.g. 78/21, or more generally "dividend/divisor", the result is two positive integers, namely a quotient and a remainder.

$$78 = 3 * 21 + 15$$

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

Suppose we have two 32 bit unsigned integers: a dividend and a divisor. Suppose they are both positive numbers. How do we compute the quotient and the remainder ? We can apply the "long division" algorithm we learned in grade school. As with multiplication, we need a set of shift registers. See slides for a rough sketch of the corresponding circuit. The algorithm is not obvious, but here it is for completeness (and fun?).

"Algorithm" for long division:

```
divisor := 2^n * divisor          // Make the divisor bigger than the dividend
quotient := 0
remainder := dividend
for i = 0 to n
        shift quotient left by one bit
        if (divisor ≤ 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 $\leq$ 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: +,-,*,/. Recall the discussion from lecture 2 where we wanted 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. *e.g.* We rewrite x so it has the same exponent as y:

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

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

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

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 the size of the shift).

You should appreciate what you need to do floating point addition. You need to compare exponents, shift right (x or y with smaller exponent), use a big adder, normalize, deal with case if result is not normalized, round off. Details of the circuit are omitted, but you should appreciate the many steps are may be involved.

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.

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

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

Since $(1101)_2 * (1001)_2 = 13 * 9 = 117 = (1110101)_2$ and $2^{-10} * 2^1 = 2^{-9}$, we can rewrite it as a normalize number as follows:

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

Now go back to the previous example in which x and y each have 23 bits of significand. We can write each of them as an integer times a power of 2:

$$\begin{aligned} x &= 1.10101000000000000000000 * 2^{-3} \\ &= 11010100000000000000000101 * 2^{-26} \end{aligned}$$

$$\begin{aligned} y &= 1.00100100001010101010101 * 2^{2} \\ &= 10010010000101010101010 * 2^{-21} \end{aligned}$$

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 floating point multiplication is more complicated than integer multiplication, but not *so much* more complicated. Note also that, for floating point division, the long division algorithm doesn't stop when the remainder is less than the divisor. Instead, it keeps going to get the negative powers of the base (2 or 10).

What about floating point division? 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.

Finite State Machines

Let's get a bit more general here. The past few lectures we have been talking about combinational and sequential circuits. For sequential circuits you should now have the idea that there may be multiple registers and each register has a value at each clock cycle. The registers change their values from one clock cycle to the next. There is an initial state (time = 0)which has certain values on the various wires of the circuit. And then the clock starts ticking and the value change in a deterministic (not random) way.

Such a set of circuits is an example of a *finite state machine*. For the circuit, each possible setting of the values in all the registers defines a state. There maybe other values (called "outputs") defined as well which are not necessarily represented by registers. For example, two register outputs might feed into a combination circuit which has outputs. There may also be values "inputs" that are sitting on wires but perhaps not written into the registers that they connect to (because a writeenable signal might be false). So at each time, we have inputs, outputs, and a state.

In COMP 330, you will study finite state machines without talking about circuits. You'll just talk about states, inputs, and outputs, and you'll talk about how the machine goes from state to state, given an input, and how it produces outputs at each time that depend on the state at that time (and perhaps on the input at that time).

In the lecture, I gave an example of a turnstile, like what you have in the Metro. There are only two things you can do at a turnstile, assuming that your OPUS pass has tickets in it. You can either put your pass down, or you can push against the turnstile. Verify for yourself that the table below captures the transitions from one state to the next.

input 0=coin 1=push	currentState 0=locked 1=unlocked	output 0= ~turn 1= turn	nextState 0=locked 1=unlocked
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

There is a lot to say about finite state machines, but I will leave it for COMP 330. The abstraction of inputs, outputs, and state transitions doesn't help us much in this course and so I'm going to stop writing now. But hopefully you get a whiff of the idea, and are asking yourself where this direction leads. Take COMP 330 and you'll find out.