3.2 Multiplication

> "Multiplication is vexation, Division is as bad; The rule of three doth puzzle me, And practice drives me mad."
> *Anonymous, Elizabethan manuscript, 1570*

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiplication.

First, let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps of multiplication and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. The following animation illustrates multiplying 1000_{ten} by 1001_{ten}.

PARTICIPATION ACTIVITY 3.2.1: Multiplication example.

Start ☐ 2x speed

Multiplicand	$1\,0\,0\,0_{ten}$
Multiplier	X $1\,0\,0\,1_{ten}$
	1 0 0 0
	0 0 0 0
	0 0 0 0
	1 0 0 0
Product	$1\,0\,0\,1\,0\,0\,0_{ten}$

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier, and shifting the intermediate product one digit to the left of the earlier intermediate products.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is $n + m$ bits long. That is, $n + m$ bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In this example, we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Although the decimal example above happens to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

PARTICIPATION ACTIVITY 3.2.2: Binary multiplication.

Consider $13_{ten} \times 6_{ten}$, or $1101_{two} \times 0110_{two}$. Fill in the missing values.

```
         1 1 0 1  (Multiplicand)
x        0 1 1 0  (Multiplier)
-----------------
         ? ? ? ?  (Partial product 1)
       ? ? ? ?    (Partial product 2)
     ? ? ? ?      (Partial product 3)
+  ? ? ? ?        (Partial product 4)
-----------------
   ? ? ? ? ? ? ?  (Product)
```

1) Partial product 1
   - ○ 0000
   - ○ 1101

2) Partial product 2
   - ○ 0000
   - ○ 1101

3) Partial product 3
   - ○ 0000
   - ○ 1101

4) Partial product 4
   - ○ 0000
   - ○ 1101

5) Product
   - ○ 11010
   - ○ 1001110

6) The largest product resulting from a

multiplication of a 7-bit multiplicand and a 7-bit multiplier is _____ bits long.

- 14
- 35

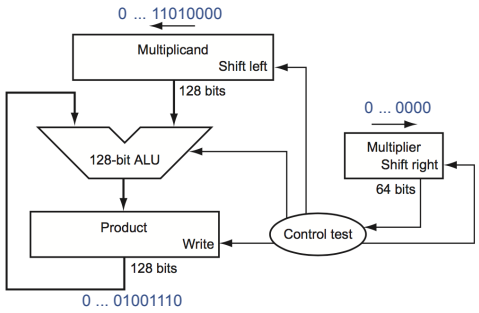
Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

Sequential version of the multiplication algorithm and hardware

This design mimics the algorithm we learned in grammar school; the figure below shows the hardware. We have drawn the hardware so that data flows from top to bottom to resemble more closely the paper-and-pencil method.

PARTICIPATION ACTIVITY 3.2.3: First version of the multiplication hardware for 64-bit multiplicand and multiplier (COD Figure 3.3).

Start 2x speed

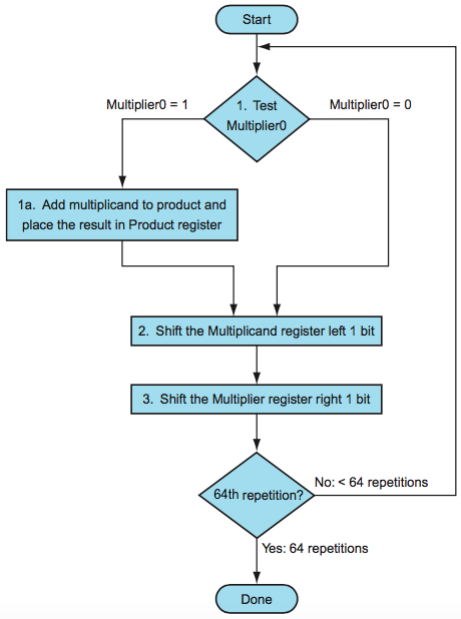


Let's assume that the multiplier is in the 64-bit Multiplier register and that the 128-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products. Over 64 steps, a 64-bit multiplicand would move 64 bits to the left. Hence, we need a 128-bit Multiplicand register, initialized with the 64-bit multiplicand in the right half and zero in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 128-bit Product register.

The figure below shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 64 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 200 clock cycles to multiply two 64-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take several clock cycles without significantly affecting performance. However, Amdahl's Law (see COD Section 1.10 (Fallacies and pitfalls)) reminds us that even a moderate frequency for a slow operation can limit performance.

Figure 3.2.1: The first multiplication algorithm, using the hardware shown in the above figure (COD Figure 3.4).

If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 64 times.



PARTICIPATION ACTIVITY 3.2.4: Sequential multiplication algorithm and hardware.

1) Each step of the multiplication algorithm shifts the Multiplier register 1 bit to the ____.
   - O right
   - O left

2) The Multiplier register is ____-bits wide.
   - O 64
   - O 128

3) Each step of the multiplication algorithm shifts the Multiplicand register 1 bit to the ____.
   - O right
   - O left

4) The Multiplicand register is ____-bits wide.
   - O 64
   - O 128

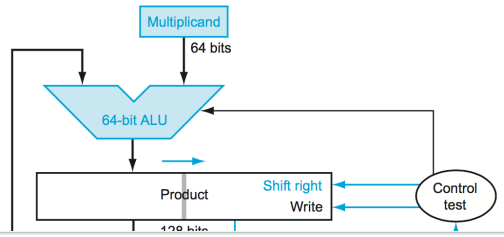
5) The Product register is ____-bits wide.
   - O 64
   - O 128
   - O 256

6) Each iteration of the multiplication algorithm consists of ____ basic steps.
   - O 3
   - O 7
   - O 64

This algorithm and hardware are easily refined to take one clock cycle per step. The speed up comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1. The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. The figure below shows the revised hardware.

Figure 3.2.2: Refined version of the multiplication hardware (COD Figure 3.5).

Compare with the first version in COD Figure 3.3 (First version of the multiplication hardware). The Multiplicand register, ALU, and Multiplier register are all 64 bits wide, with only the Product register left at 128 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 129 bits to hold the carry out of the adder, but it's shown here as 128 bits to highlight the evolution from COD Figure 3.3 (First version of the multiplication hardware).)



Hardware/Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. As mentioned in COD Chapter 2 (Instructions: Language of the Computer), almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

PARTICIPATION ACTIVITY 3.2.5: Refined multiplication hardware.

Refer to the refined multiplication hardware figure above (COD Figure 3.5).

1) The refined multiplication hardware halves the width of the Multiplicand register from 128-bits to 64-bits.
   - ○ True
   - ○ False

2) The Multiplier register is removed and placed inside of the _____ register.
   - ○ Product
   - ○ Multiplicand

3) The ALU adds the 128-bit Product and 64-bit Multiplicand, and then stores the result into the Product register.
   - ○ True
   - ○ False

Example 3.2.1: A multiply algorithm.

Using 4-bit numbers to save space, multiply $2_{ten} \times 3_{ten}$, or $0010_{two} \times 0011_{two}$.

Answer

The figure below shows the value of each register for each of the steps labeled according to COD Figure 3.4 (The first multiplication algorithm ...), with the final value of $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Figure 3.2.3: Multiply example using algorithm in COD Figure 3.4 (The first multiplication algorithm) (COD Figure 3.6).

The bit examined to determine the next step is circled in color.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000(1)	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000(0)	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000(0)	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

PARTICIPATION ACTIVITY 3.2.6: Multiply example using the first multiplication algorithm.

Consider the table in the above figure.

1) The initial 8-bit value of Product is _____.

Check Show answer

2) At the end of iteration 1, the 8-bit value of Product is _____.

Check Show answer

3) At the end of iteration 2, the 4-bit value of Multiplier is _____.

Check Show answer

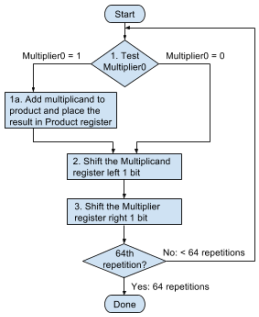
4) At the end of iteration 2, the 8-bit value of Multiplicand is _____.

Check Show answer

PARTICIPATION ACTIVITY 3.2.7: Multiplication algorithm steps and register values.

Consider the multiplication of $5_{10} \times 12_{10}$, or $0101_2 \times 1100_2$. Fill in the missing values for each of the steps labeled according to COD Figure 3.4 (The first multiplication algorithm ...). A copy of the multiplication algorithm figure is shown below to the right.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	1100	0000 0101	0000 0000
1	1: 0 ⇒ No operation	1100	0000 0101	0000 0000
	2: Shift left Multiplicand	1100	0000 1010	0000 0000
	3: Shift right Multiplier	0110	0000 1010	0000 0000
2	(a)	0110	0000 1010	0000 0000
	2: Shift left Multiplicand	0110	(b)	0000 0000
	3: Shift right Multiplier	(c)		0000 0000
3	(d)			0001 0100
	2: Shift left Multiplicand		0010 1000	0001 0100
	3: Shift right Multiplier	0001	0010 1000	0001 0100
4	1a: 1 ⇒ Prod = Prod + Mcand	0001	0010 1000	(e)
	2: Shift left Multiplicand	0001	0101 0000	
	3: Shift right Multiplier	0000	0101 0000	



0001 0100 1a: 1 ==> Prod = Prod + Mcand 1: 0 ==> No operation 0011

0011 1100

(a)

(b)

(c)

(d)

(e)

Reset

Signed multiplication

So far, we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember their original signs. The algorithms should next be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

It turns out that the last algorithm will work for signed numbers, if we remember that we are dealing with numbers that have infinite digits, and we are only representing them with 64 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower doubleword would have the 64-bit product.

Faster multiplication

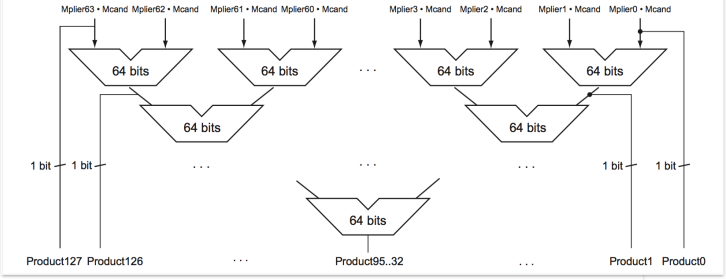
Moore's Law has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 64 multiplier bits. Faster multiplications are possible by essentially providing one 64-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.

A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 64 high. An alternative way to organize these 64 additions is in a parallel tree, as the figure below shows. Instead of waiting for 64 add times, we wait just the $\log_2(64)$ or six 64-bit add times.

Figure 3.2.4: Fast multiplication hardware (COD Figure 3.7).

Rather than use a single 64-bit adder 63 times, this hardware "unrolls the loop" to use 63 adders and

then organizes them to minimize delay.



In fact, multiply can go even faster than six add times because of the use of *carry save adders* (see COD Section A.6 (Faster Addition: Carry Lookahead) in COD Appendix A (The Basics of Logic Design)), and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see COD Chapter 4 (The Processor)).

Multiply in LEGv8

To produce a properly signed or unsigned 128-bit product, LEGv8 has three instructions: *multiply* (`MUL`), *signed multiply high* (`SMULH`) and *unsigned multiply high* (`UMULH`). To get the integer 64-bit product, the programmer uses `MUL`. To get the upper 64 bits of the 128-bit product, the programmer uses either `SMULH` or `UMULH`, depending on the types of multiplier and multiplicand.

Summary

Multiplication hardware simply shifts and adds, as derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of 2. With much more hardware we can do the adds in **parallel**, and do them much faster.

Hardware/Software Interface

LEGv8 multiply instructions do not set the overflow condition code, so it is up to the software to check to see if the product is too big to fit in 64 bits. There is no overflow if the upper 64 bits is 0 for `UMULH` or the replicated sign of the lower 64 bits for `SMULH`.

PARTICIPATION ACTIVITY 3.2.8: LEGv8 multiplication.

1) The multiplication hardware supports signed multiplication.
   - ○ True
   - ○ False

2) The unsigned multiply high (`UMULH`) instruction can return the upper 64 bits of a 128-bit product.
   - ○ True
   - ○ False

3) The multiply (`MUL`) instruction ignores overflow, while the signed multiply high (`SMULH`) and unsigned multiply high (`UMULH`) instructions detect overflow.
   - ○ True
   - ○ False

Provide feedback on this section