CS130 Lab Exercise #8 – Multiplying Using "Shift & Add"

Overview

In an earlier lab, students created an assembly-level program to multiply two numbers together using a so-called "Cumulative Add" algorithm. In this lab exercise, students will put on their "Creative Thinking Caps" and, using the ARM simulator in *zyBooks* Section 2.24, develop an assembly language program to multiply using – let's call it – "Shift & Add". When we homo sapiens do multiplication in long-hand (remember that process?), we essentially are using this method, albeit in decimal not in binary. In this process, step-by-step we multiplied the upper number (called the multiplicand) by the lower number (called the multiplier) and generated a result called the product.

Multiplication of numbers in binary format is actually simpler since we do not need to know the ubiquitous "Times Table" which we all so fondly remember from arithmetic. First, we define a location – either a register or a location in memory, initialized originally to zero – where we will accumulate a sum of values. To effect binary multiplication, we examine the LSB (for Least Significant Bit, **`Bit 0`**) of the multiplier and, if it is a binary one, we add the multiplicand to our storage repository.

We next shift the multiplicand by one bit-position to the left using the instruction LSL (for Logical Shift Left), essentially multiplying the value of the multiplicand by 2. We now examine the next bit of the multiplier (**`Bit 1`**) and, if it is a binary one, we add the left-shifted multiplicand to our accumulating sum. We continue this process until all bits of the multiplier have been tested and acted upon – or not as the case may be. In theory, therefore, the number of Shift & Add operations need not exceed 1 + the number of the highest-order bit-position in the multiplier that contains a binary 1.

An example

Let's consider the case where we are to multiply 15 by 13 and expect to get a result of 195. Here, for simplicity, we will (a) assume a register size of 8-bits, (b) process unsigned numbers only, and (c) not allow for an overflow beyond eight bits. For this example, $15_{10} = 1111_2$, and $13_{10} = 1101_2$ Therefore, we expect: $0000\,1111_2 \times 0000\,1101_2 = 1100\,0011_2$. Below is presented a step-by-step procedure to effect this multiplication followed by two long-hand examples.

Step	Action
1	Initialize to zero the register or memory location which will be used to sum the various terms as the iteration progresses.
2	Examine the LSB of multiplier (**`Bit 0`**). If it is a 1, add the multiplicand to our running sum
3	Shift multiplicand left one position
4	Examine the next bit (**`Bit1`**) of multiplier. If it is a **`1`**, add the shifted multiplicand to running sum; if it is a **`0`**, do not sum.
5	Continue this process until all bits in the multiplier have been processed.
6	The final cumulative sum is the product of the two numbers.

Long-hand method:

```
  0 0 0 0 1 1 1 1
× 0 0 0 0 1 1 0 1
-----------------
  0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0
  0 0 0 0 1 1 1 1
0 0 0 0 0 1 1 1 1
-----------------
0 0 0 1 1 0 0 0 1 1
```

Also:

```
  0 1 1 1 1 1 1 1
× 0 0 0 0 0 0 1 0
-----------------
  0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 0
```

Above:

2 × 127 = 254

Possible problem to be considered

As stated above, the number of "Shift & Add" operations required depends upon the number of "used" bit-positions that define the multiplier. That is, we might have available 64-bit registers such as are found in LEGv8; however, not all 64 bit-positions are required to define the number. Two easier-to-visualize, 8-bit, cases are presented above. For the first case, we see the two terms together (multiplicand and multiplier) 'occupy' a total of "used" 8 bit-positions. Looking at the second case, it ap-

pears that we were able to get a valid multiplication without overflowing the size of any of the three registers even if we if we have 9 bit-positions used. Although not shown, reversing the multiplicand and the multiplier still produces a valid product without exceeding the register size of 8 bits.

Consider the following two numbers with an eye to LEGv8:

(a) The value **9876543210_{10}** is **10 0100 1100 1011 0000 0001 0110 1110 1010** in binary. Here we have used 34 of the 64 bit-positions if we are storing this number in a LEGv8 register. The consequence of this observation is that we cannot left-shift this number more than 30 positions or it will overflow the size of the LEGv8 register. Let's consider a second number:

(b) The value **1234567890_{10}** is **100 1001 1001 0110 0000 0010 1101 0010** in binary. Here we have 31 of the 64 bit-positions used if we are storing this number in a LEGv8 register. The consequence of this observation is that we cannot left-shift this number more than 33 positions or it will overflow the size of the LEGv8 register.

Considering these two numbers above as a pair to be multiplied, can we effect a valid multiplication in LEGv8 using 'Shift & Add'? We see that the number of bit-positions used = 34 + 32 = 65. Based upon our observation of the second example above (where the total number of 'used' bit-positions is 9 for two 8-bit registers), the Las Vegas Oddsmakers are saying "Yes we can!" (Careful!)

This exercise

Okay, so how do these observations apply to this lab exercise? One might conclude without further examination that our still-in-the-conceptual-stage "Shift & Add" algorithm can handle any number-pair that (a) does not have a total number of "used" bit-positions exceeding 65, and (b) does not produce a product too large to fit into a LEGv8 64-bit register or memory location. Observing he two examples above leads to the conclusion that it matters not which value is chosen as the multiplicand and which is selected ss the multiplier.

So, in this lab exercise we will be trying to answer the following questions:

1.) Are the conclusions presented above valid?
2.) How do we determine the number of bit-positions used for the multiplicand?
3.) How do we determine the number of bit-positions used by the multiplier?
4.) Does the sum of (2) and (3) above exceed 65?
5.) Does the product of the two numbers fit in a 64-bit LEGv8 register?

One last observation

It appears that the ARM simulator found in *zyBooks* Section 2.24 displays as negative any number in which **Bit 63** is a **1** even though the number may actually be an unsigned number.

Deliverables

NOTE: "Deliver" source code in a text file containing your "exported" assembly code.

1.) LEGv8 assembly code to calculate the number of bit-positions occupied in **2147483647**
2.) LEGv8 assembly code to calculate the number of bit-positions occupied in **4294967295**
3.) LEGv8 'Shift & Add' assembly code to multiply the numbers found in Deliverables (1) and (2)
4.) Use your code in Step (1) to calculate the number of bit-positions occupied in Example A above
5.) Use your code in Step (2) to calculate the number of bit-positions occupied in Example B above
6.) Use your code in Step (3) to multiply the numbers used in Deliverables (4) and (5)
7.) Is your result (i. e. the product) for Deliverable (3) correct?
8.) Is your result of Deliverable (6) correct? (Consider Question 5 in "This exercise" section above.)

For the requirements above, start by manually storing one number in Memory address **4000**, the second in Memory address **4008**, and **4000** in Register **X3**. You may consider creating a separate program for each of the requirements above, or, if you are more daring, one composite program that meets the requirements for (1) - (3) and one for (4) - (6). Carpe diem!

When you are satisfied with your splendid creations, submit your "Mary Poppins Practically Perfect" assembly code, properly named, to RSturlaCS130@GMail.com.

How many of you changed your name over the break to "Exercise", "Student", or "Lab Exercise"? The Las Vegas Oddsmakers are betting that there will be at least one student who did! Want to take that bet?