HW3: Advanced MIPS and Faster Multiplication

This assignment gives more practice with MIPS instructions and using logic to write assembly language solutions to problems. You should complete this assignment with your same partner. **Make sure that both of your names are at the top of your file in comments**, and that you comment well in general.

Deliverables: The file *username1_username2.asm* containing your solution program.

Faster Multiplication of Positive Integers

Given the limited instructions available to use, you probably wrote your multiplication program in HW1 using some variation of the following algorithm:

```
to multiply x * y:
set result to 0
add x to result y times
```

This works just fine, however it is not very efficient. In fact it is O(y), or in other words it requires executing c*y instructions where c is some constant. Much more efficient is to use the algorithm you would use if multiplying 2 large numbers by hand, which involves summing up the partial results created by multiplying one number by each digit of the 2nd number individually. For example, you could break down the multiplication of 2 3-digit numbers in the decimal system as follows:

```
XYZ * ABC = XYZ*C + XYZ*B*10 + XYZ*A*100
```

Note that the *10 and *100 for the 2nd and 3rd partial results are the same as adding the appropriate number of extra 0's to the end of each, as you do when actually multiplying by hand. In essence you actually *shift* each partial result to the left, with the first partial result shifted left 0 digits, and each further partial result shifted left 1 more digit than the previous.

This method of multiplication works the same no matter what base number system you are using, so you can use the exact same technique to multiply 2 binary numbers. However, even if you use shifting instead of multiplying by 10 and 100, the above equation still uses multiplication itself to get each partial result, while our goal is to use only addition. This is where the binary number system makes things simpler. Try writing out the multiplication of the base-10 numbers 234 and 1,011 by hand using the above method. Make sure you put 1,011 as the 2nd number in the multiplication. What pattern do you see?

```
234*1,011 = 234*1 + 234*1*10 + 234*0*100 + 234*1*1,000

= 234 + 2340 + 0 + 234000
```

Since in binary the only possible digits are 1 and 0, you'll only ever be multiplying by a 1 or a 0 to get the next partial sum, and then shifting it left the appropriate number of digits. This means that each term in the final equation will either be 234 shifted left some number of digits, or simply 0. You don't actually need to do any multiplying, but can use logic and conditionals along with shifting to determine the value of each term.

This faster method is just O(# digits in the 2nd operand): at most 32 digits in MIPS.

Your Task

Write a program in MIPS that performs multiplication using this faster method. Most of the credit will be given for a solution that works correctly and runs in O(# digits) time as described above, however for full credit you should make your solution efficient and correctly handle edge cases. For example, to be efficient you should loop as few times as possible, and you should minimize the number of basic instructions executed each time through the loop (somewhere around 10 is fine, though you can do it with less). Edge cases include 0 or a negative number as one or both of the operands – ideally you could design an algorithm that works for any case without needing extra checks. You do NOT have to check for overflow in this program, though it's useful to think about how you would.

I highly recommend you come up with an algorithm in pseudo-code first and TEST IT on some example numbers before trying to debug actual MIPS assembly code. For this task you may use the following larger subset of MIPS instructions (purple denotes new instructions added):

- any of the load and store instructions (e.g. *lw*, *sw*)
- any of the add and subtract instructions (e.g. *add*, *addi*)
- any of the branch/jump instructions (e.g. *beq*, *bne*, *bgt, j*)
- any of the shift instructions (e.g. *sll, srl, sllv*)
- any of the logical operation instructions (e.g. *and, or, nor)*

Clearly as before you may NOT use instructions *mult* or *div*.

As in the last HW, you should **load the 2 numbers to multiply from the first 2 words of the data section of memory and store the result in the 3rd word**.