Lab Goals

- Practice some binary multiplication and division
- Review how to represent floating-point numbers
- Implement a two MIPS programs to multiply two binary numbers

Assignment Details

We’re getting closer to wrapping up our discussion on the internal representation of binary numbers and the operations that we can perform on them. This week’s lab covers binary multiplication and division from Friday and Monday, as well as floating-point numbers from the lecture this morning.

Part 1: Binary Multiplication and Division (10 points)

Please answer each of the following questions about binary multiplication and division (for two points each). Please show all of your work for partial credit.

1. Convert \((47)_{10}\) into binary and multiply by \((8)_{10}\). Report your result in hexadecimal.

2. Convert \((47)_{10}\) into binary and divide by \((16)_{10}\). Report your result in decimal.

3. Multiply together \((10101)_2\) and \((1010)_2\) using the long multiplication method. Report your result in binary.

4. Divide \(\frac{(10101)_2}{(1010)_2}\) using the long division method. Report your result in hexadecimal.

5. Divide \(\frac{(10101)_2}{(6F)_{16}-(-111)_{10}}\) using the long division method. Report your result in binary.
Part 2: Floating-Point Numbers (10 points)

Please answer each of the following questions about floating point numbers (for two points each). Again, please show all of your work for partial credit.

1. Convert $(+18.25)_{10}$ to floating point binary normal form. Report your result in binary.

2. Convert $(-18.1875)_{10}$ to floating point binary normal form. Report your result in binary.

3. Convert $(+18)_{10}$ to floating point binary normal form. Report your result in binary.

4. Convert $(+8.25)_{10}$ to floating point binary normal form. Report your result in hexadecimal.

5. Convert $(-15.999755859375)_{10}$ to floating point binary normal form. Report your result in hexadecimal.

Part 3: Implementing Multiplication (10 points)

Multiplication (and division) in MIPS and MARS are a bit different than addition and subtraction, simply because multiplying two 32-bit numbers can result in up to a 64-bit output. This gets even more complicated because integers can be no longer than 32 bits, so we only want half of that potential 64-bit output. MIPS uses two additional registers ($hi$ and $lo$) to support the multiplication operation, though these registers are hidden from view during the computation. The following code snippet demonstrates multiplication:

```
mult $t1, $t2
mflo $t0
```

In this code, we have the multiplicand stored in $t1$ and the multiplier stored in $t2$ (or the other way around, since multiplication is a commutative operation). The $mult$ instruction performs the multiplication computation, storing the result in those $hi$ and $lo$ registers. The $mflo$ instruction then moves the value stored in $lo$ into register $t0$ for use throughout the remainder of the program.

**For the first 5 points in this section**, create a MIPS program called $MultiplyBuiltIn.asm$ that will clearly prompt the user to enter two numbers (print requests rather than just wait for input), multiply the inputs together, and output the result. Check the MARS SysCall list online (a link is provided in a previous lab and on the course schedule page) to see which sys$call$ number supports reading an integer from a user. Please comment the program thoroughly.

MIPS providing a multiplication instruction is too easy, so we’ll also implement our own procedure using the flowchart from Friday’s binary multiplication lecture. **For the second 5 points in this section**, create a MIPS program called $MultiplyImplemented.asm$. Rather than dealing with user I/O, you can simply hard-code $multiplicand$ and $multiplier$.words into the .data section of this program (you can also reuse code from the first program if you wish to perform user I/O). You will need to keep track of multiplicand, multiplier, and product registers as you work your way through the computation, and output the product when the computation is finished. The following instructions that we have not discussed yet will be necessary:
• **sll (Shift Left Logical)** – Shifts the contents of a register to the left by a provided number of bits. The format of the instruction is “sll $dest, $source, shamt” where $dest is the register where the shifted bits will be placed, $source is the register where the pre-shifted bits are located, and **shamt** is the number of bits to shift to the left.

• **srl (Shift Right Logical)** – Shifts the contents of a register to the right by a provided number of bits. The format of the instruction is “srl $dest, $source, shamt” where $dest is the register where the shifted bits will be placed, $source is the register where the pre-shifted bits are located, and **shamt** is the number of bits to shift to the right.

• **beq (Branch on EQual)** – The opposite of the not-equal branch bne, this instruction will branch to the provided label if the data in the other two inputs or registers are equal. The format of the instruction is “beq $in1, $in2, LABEL”. You will need this (or bne) to control your 32x loop through the flowchart, as well as to decide whether to follow the left or right branch after testing Multiplier0.

• **andi (Logical AND with an Immediate)** – The instruction “andi $t0, $s0, 0x1” will retrieve the rightmost bit of $s0 and store it in $t0. We’ll discuss why this works in a few classes, but you’re welcome to try to figure it out yourselves.

This second program is certainly more complicated than the first, but is well within your ability to handle. I recommend assembling and running often, stepping through the code to ensure that each instruction is running as you expect. Please comment the program thoroughly.

**Submission Details**

For this assignment, your submission to your BitBucket repository should include the following:

1. Your responses to the questions in Part 1. [If you have hand-written the solutions to these exercises, please submit them to TA Victor Zheng instead of BitBucket.]

2. Your responses to the questions in Part 2. [If you have hand-written the solutions to these exercises, please submit them to TA Victor Zheng instead of BitBucket.]

3. Commented versions of your MultiplyBuiltIn.asm and MultiplyImplemented.asm code.

Before you turn in this assignment, you also must ensure that the course instructor has read access to your BitBucket repository that is named according to the convention cs210f2016-<your user name>. 