**ECE 260: Fundamentals of Computer Engineering – Lab #7  
Introduction to Floating-Point Operations**

**Name \_\_\_\_\_\_\_Le Quient Lewis \_\_\_\_\_\_ Lab Partner(s) \_\_\_\_N/A\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides students with practice using single-precision and double-precision floating-point instructions in MIPS. Students will utilize basic arithmetic, branching, and comparison instructions to implement numerical algorithms.

**2. Background**

**2.1 Overview of Floating-Point Operations**

Special hardware is required to implement arithmetic operations for floating-point values. This hardware is called a Floating-point Unit (FPU) and is typically integrated into a processor. The MIPS processor integrates an FPU and provides specific instructions to utilize the hardware. More specifically, the MIPS instruction set includes a variety of instructions to handle both *single-precision* (32-bit) and *double-precision* (64-bit) floating-point operations. Additionally, the MIPS instruction set contains specialized load, store, and comparison instructions for single-precision and double-precision floating-point values.

Many years ago, FPUs were implemented on a separate coprocessor chip that could be purchased with your CPU to provide additional functionality. Within the MIPS ISA, the FPU coprocessor was referred to as *coprocessor1*. Even though floating-point operations are now integrated directly into the CPU, all floating-point operations retain the *coprocessor1* nomenclature and will have the literal “*c1”* somewhere in the instruction. The MIPS hardware provides special floating-point registers to perform floating-point calculations. These floating-point registers are name **$f0** - **$f31**. While data can be moved between the floating-point and integer registers, they are not interchangeable inside an instruction. **A floating-point instruction will only operate on floating-point registers.**

**2.2 Arithmetic and Load/Store Floating-Point Instructions**

Table 1 below shows several arithmetic instructions with their single and double-precision floating-point analogs. The add, subtract, multiply, and divide operations are simple extensions of existing instructions. Note that the “.s” denotes a single-precision instruction while the “.d” denotes a double-precision instruction. Special move instructions (*mfc1* and *mtc1*) are required to move values between *traditional* registers (e.g. **$s0**-**$s8**) and floating-point registers (**$f0**-**$f31**).

Table 1: Comparison of integer, single-precision, and double-precision floating-point arithmetic operations

|  |  |  |
| --- | --- | --- |
| **Integer Operations** | **Single-precision Floating-Point Ops** | **Double-precision Floating-Point Ops** |
| add | add.s | add.d |
| sub | sub.s | sub.d |
| mul | mul.s | mul.d |
| div | div.s | div.d |
| lw | lwc1 | ldc1 |
| sw | swc1 | sdc1 |
| move | mfc1/mtc1 | mfc1.d/mtc1.d |

Single-precision floating-point operations are analogous to integer operations in that each operand consumes a single 32-bit word/register. However, in double-precision floating-point operations, each operand is 64-bits and therefore requires two registers. Thus, adding two doubles (**$f2** and **$f4**) and storing the result in **$f0**, (e.g. *add.d $f0, $f2, $f4*) requires six registers in total. Furthermore, **all double-precision floating-point values must be stored in even numbered registers $f0, $f2, $f4, etc**. When a double-precision floating-point value is placed into a register the value consumes the specified register AND the next numbered register. That is, a double-precision instruction like *add.d $f0, $f2, $f4* actually writes the 64-bit result into TWO registers, **$f0** and **$f1**. Of course, this means that if you previously had a single-precision value stored in **$f1** and then execute the *add.d* instruction shown above, you will overwrite the single-precision value in **$f1**. Likewise, if you execute the *add.d* instruction shown above and then later overwrite register **$f1** with a 32-bit value, you corrupt the contents of the double stored in register **$f0**. 😮

Single and double-precision floating-point values can be loaded/stored from/to memory in the same manner as integer values using load/store operations. However, as previously mentioned, double-precision values are 64 bits (8 bytes) long, instead of the familiar 32 bits (4 bytes). Furthermore, loading a double-precision value from memory into a floating-point register will consume the specified register and its adjacent neighbor in the same way that the double-precision arithmetic instructions did in the previous paragraph.

**2.3 Comparison and Branching with Floating-Point Values**

Comparisons and branches with floating-point values are similar to those of integer values. One distinction is that the results of all floating-point comparisons are stored in a special *FPCond* register. Floating-point comparison instructions automatically place the comparison result in that register. This is distinct from integer comparisons that use an *slt* instruction where the destination register of the result is specified. In conjunction with the *FPCond* register, two branch instructions are provided that branch based upon whether the *FPCond* contains a 1 (*bc1t*, branch when comparison is true) or a 0 (*bc1f*, branch when comparison is false). Consequently, the basic floating-point branch instructions only take one operand, a destination label for the branch if the branching instruction is true. Table 2 shows the integer compare and branch instructions with the equivalent floating-point instruction.

Table 2: Comparison of integer, single-precision, and double-precision floating-point compare and branch operations

|  |  |  |
| --- | --- | --- |
| **Integer Operations** | **Single-precision Floating-Point Ops** | **Double-precision Floating-Point Ops** |
| slt | c.lt.s (compare less than) | c.lt.d |
| slti | N/A | N/A |
| beq | bc1t | bc1t |
| bne | bc1f | bc1f |
| N/A | c.eq.s (compare equals) | c.eq.d |
| N/A | c.le.s (compare less than equals) | c.le.d |

**2.4 Visualizing Floats**

MARS provides several tools to help visualize floating-point values. The “*Coproc1*” tab displays the values of the **$fX** registers. The first column shows the register name; the second column displays the value if it were interpreted as a single-precision floating-point value; the third column displays the value if it were interpreted as a double-precision floating-point value. MARS does not know if the values in the **$fX** registers are 32 bit or 64 bit value and thus interprets the register values in both forms. As mentioned earlier, double-precision values require two 32-bit registers and are stored in even numbered **$fX** registers. Therefore, MARS does not provide an interpretation for double-precision values in the odd numbered **$fX** registers. Figure 1 shows the “Coproc1” tab in the MARS simulator and the simulator’s interpretation of the contents of the **$fX** registers. In the example in Figure 1, registers **$f0** and **$f1** can be interpreted as containing the single-precision values -99.689 and -123.123. Using the same register contents, register **$f0** can be interpreted to contain the double-precision value -3.913560136391803E14. Register **$f1** cannot be interpreted to contain a double-precision value. Attempting to store a double-precision value in an odd numbered **$fX** register will result in an error.

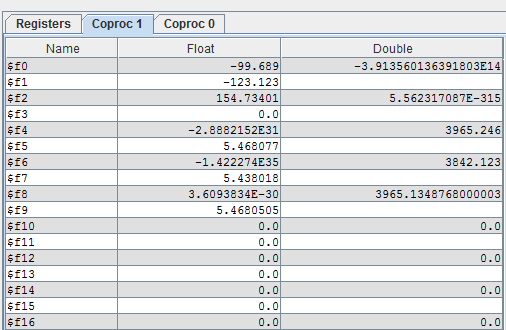


Figure 1: Visualizing floating-point values in MARS

**3. Procedure**

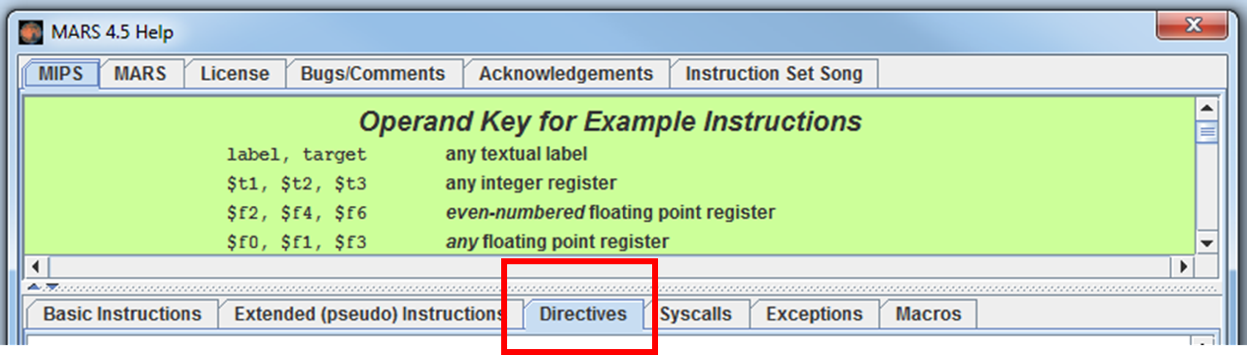
This lab has multiple parts. For each part, write your code in the specified file (all of the required .asm files are in the ***src* directory**). Writing your code in the wrong file will cause problems when you test your code.

**3.1 Loading and Adding Single and Double Floats**

In this section, you will load two lists of floating-point values into memory and perform arithmetic calculations on their contents. Prior to performing any calculations, you will need to load values from memory into a floating-point registers. **You do NOT need to write any procedures for this part**.

**Programming Tasks**

1) Open the file called lab07\_part1.asm in the MARS Editor. Read the MARS help file (press F1) and identify the two directives that are used to create lists of single- and double-precision values. You can find the list of directives that MARS supports under the “Directives” tab as shown in Figure 2.

  
Figure 2: The “Directives” tab in the MARS Help file

2) In the *.data* segment, create an array called “*singles*” that contains the single-precision floating-point values shown here: 1.4563, 2.4564, -4.2342, 65.334. Immediately after that, create a second array called “*doubles*” that contains the double-precision floating-point values shown here: 1.4563, 2.4564, -4.2342, 65.334 (yes, they are the same values as “*singles*”). Use the appropriate directives, as identified in part 1, to create these arrays.

3) Under the “*doSingles*” label in the *.text* section, perform the following computations:  
 a) *singles*[0] + *singles*[1] **(put this result in $f4)**  
 b) *singles*[1] + *singles*[2] **(put this result in $f5)**  
 c) *singles*[2] + *singles*[3] **(put this result in $f6)**  
Be sure to load the single-precision floating-point values from memory into **$fX** registers using the appropriate load instruction (see Table 1). You cannot do floating-point arithmetic on values stored in **$sX** or **$tX** registers. Use the “*Coproc1*” tab in MARS to view your results and then fill in Table 3 below. Are the results that you see in MARS the values that you expect?

Once you’re certain that your computations are correct, use the *mfc1* instruction to move the results from computations (a), (b), and (c) into registers **$s0**, **$s1**, and **$s2** respectively. Note, that the MARS simulator will interpret your values differently when you view the results in the **$sX** registers.

4) Under the “*doDoubles*” label in the *.text* section, perform the following computations:  
 a) *doubles*[0] + *doubles* [1] **(put this result in $f24)**  
 b) *doubles* [1] + *doubles* [2] **(put this result in $f26)**  
 c) *doubles* [2] + *doubles* [3] **(put this result in $f28)**  
DO NOT overwrite any of the registers that you used in the “*doSingles*” section! Be sure to load the double-precision floating-point values from memory into **$fX** registers using the appropriate load instruction (see Table 1). You cannot do floating-point arithmetic on values stored in **$sX** or **$tX** registers. Also, recall that each double-precision value consumes two **$fX** registers, not just one. Use the “*Coproc1*” tab in MARS to view your results and then fill in Table 3 below. Are the results that you see in MARS the values that you expect?

Once you’re certain that your computations are correct, use the *mfc1.d* instruction to move the results from computations (a), (b), and (c) into registers **$t0**, **$t2**, and **$t4** respectively. Note, that the MARS simulator will interpret your values differently when you view the results in the **$tX** registers.

Table 3: Expected and computed sums for each of the required floating-point addition operations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Computation Type** | **Computation** | **Op1** | **Op2** | **Expected Sum** | **Computed Sum** |
| Single-precision add | *singles*[0] + *singles*[1] | 1.4563 | 2.4564 | 3.9127 | 3.9127 |
| Single-precision add | singles[1] + *singles*[2] | 2.4564 | -4.2342 | -1.778 | -1.778 |
| Single-precision add | *singles*[2] + *singles*[3] | -4.2342 | 65.334 | 61.0998 | 61.0998 |
| Double-precision add | *doubles*[0] + *doubles*[1] | 1.4563 | 2.4564 | 3.9127 | 3.9127 |
| Double-precision add | *doubles*[1] + *doubles*[2] | 2.4564 | -4.2342 | -1.778 | -1.778 |
| Double-precision add | *doubles*[2] + *doubles*[3] | -4.2342 | 65.334 | 61.0998 | 61.0998 |

**Test Your Code**

Test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab07\_Introduction\_to\_Floating\_Point\_Operations  
make test\_part1

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**3.2 Calculating the Area of a Circle**

Within MIPS there are no instructions to load literal values (-12.2343, 34.98, π, etc.) directly into floating-point registers (i.e. no load immediate for floats). All floating-point constants (single- and double-precision) must be stored in the *.data* segment and then loaded from memory into **$fX** registers as needed. Utilize the *.data* segment to calculate the following values below. **You do NOT need to write any procedures for this part**.

**Programming Tasks**

1) Open the file called lab07\_part2.asm in the MARS Editor.

2) In the *.data* segment, create an array called “*eq1\_vals*” that contains the single-precision floating-point values shown here: 25.9, 3.14159265359. Immediately after that, create a second array called “*eq2\_vals*” that contains the double-precision floating-point values shown here: -44.321, 9.12, 2.53, 1.25. Use the appropriate directives, as identified in part 1, to create these arrays.

3) Under the “*eq1\_computation*” label in the *.text* section, perform the computations required for EQ#1 from the table below. Put your result for this computation into register **$f3**. Once you’re certain that your computation is correct, use the *mfc1* instruction to move the result from your **$f3** register into register **$s0**.

4) Under the “*eq2\_computation*” label in the *.text* section, perform the computations required for EQ#2 from the table below. Put your result for this computation into register **$f28**. Once you’re certain that your computation is correct, use the *mfc1.d* instruction to move the result from your **$f28** register into register **$t0**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Equations** | **Input Values** | **Expected Output** | **Computed Output** |
| **EQ #1** | **(perform as single-precision)** | r = 25.9 | A = 2,107.41176 | A = 2,107.4119 |
| **EQ #2** | **(perform as double-precision)** | x = -44.321  y = 9.12 | z =  4969.945195133509 | z = ?? |

**Test Your Code**

Test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab07\_Introduction\_to\_Floating\_Point\_Operations  
make test\_part2

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**4. Submission**

When you have finished your lab, demo your program for your instructor. Write your answers to the above questions electronically in this document. To submit your lab assignment, make sure your files have all been saved (*including this file*). In a Cygwin window type the commands:

cd h:  
cd ECE260  
cd Lab07\_Introduction\_to\_Floating\_Point\_Operations  
make submit

Enter your Marmoset username and password (which you should have received by email). Note that your password will not be echoed to the screen. Make sure that after you enter your username and password, you see a message indicating that the submission was successful. Log into [Marmoset](https://cs.ycp.edu/marmoset/login) via the web to check the files you submitted to ensure they are correct.

**DO NOT MANUALLY ZIP YOUR PROJECT AND SUBMIT IT TO MARMOSET.  
YOU MUST USE THE make submit COMMAND.**