**ECE 260: Fundamentals of Computer Engineering – Lab #8  
More Fun with Floats**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Lab Partner(s) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides students with practice using **double-precision** floating-point values to calculate mathematical expressions and provides an overview of data dependencies in algorithms.

**2. Background**

The previous lab reviewed how to load values into the floating-point registers and how to perform basic arithmetic operations. In this lab, you will use that knowledge to perform more complex calculations. Information on comparison and branching instructions are included in Section 2.1 (same content as previous lab). Section 2.2 introduces several new floating-point instructions that will be helpful for this lab.

**2.1 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 1 shows the integer compare and branch instructions with the equivalent floating-point instruction.

Table 1: Comparison of integer, single-precision, and double-precision floating-point compare and branch instruction

|  |  |  |
| --- | --- | --- |
| **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 |

**2.2: Special Floating-Point Operations**

**Arithmetic Instructions**

Many other floating-point instructions will be necessary for this lab. First, are single- or double-precision floating-point arithmetic instructions. Each of these instructions uses the suffix *.s* or *.d* is used to indicate whether the operands and results are single- or double-precision floating-point values. Arithmetic instructions that you will find useful for this lab are: *add.s*/*add.d*, *sub.s*/*sub.d*, *abs.s*/*abs.d*, *mul.s*/*mul.d*, and *div.s*/*div.d*. These are floating-point addition, subtraction, absolute value, multiply, and divide. When using these instructions, the source operands must be in **$fX** registers. The destination register must also be an **$fX** register.

**Load/Store Instructions**

Loading floating-point numbers from memory can be accomplished using pseudo-instructions specifically for loading floating-point values: *l.s/l.d $fX, <label>*. These instructions load the first single- or double-precision floating-point value at the memory address specified by the label (similar to the *lw* instruction) and store it into the **$fX** register specified. Likewise, storing floating-point values can be accomplished using *s.s*/*s.d $fX, <label>*. There are some very important points to note when loading and storing double-precision floating-point values. Firstly, double-precision floating-point values are 8 bytes and consume TWO words of memory. That is, if you run the following instruction:

*s.d $f0, 0x10010000* # assume **$f0** contains to the 64-bit double 0x12345678\_A5A5A5A5  
 # where **$f1** contains 0x12345678 and **$f0** contains 0xA5A5A5A5

the double-precision floating-point value stored in registers **$f1**+**$f0** will get stored in memory starting at address 0x10010000. The contents of register **$f0**, the least-significant word, will get stored at address 0x10010000. The contents of register **$f1**, the most-significant word, will get stored at address 0x10010004.

**Also note, when storing and loading double-precision floating-point values, the memory address you specify MUST be doubleword aligned (this includes the stack).** That is, the memory address must be evenly divisible by 8. A quick way to determine if a memory address is aligned on a doubleword boundary is to examine the 3 least significant bits of the address (bits 2 downto 0). If the three least significant bits of the address are all 0, then the memory address is evenly divisible by 8 and is therefore aligned on a doubleword boundary. When viewing memory addresses in hexadecimal format, if the least-significant hexadecimal character is 0x0 or 0x8, then the memory address is evenly divisible by 8 and is therefore aligned on a doubleword boundary.

**Instructions to Move Data**

Occasionally, you may want to move values between floating-point registers. You can move data from one **$fX** register to another **$fX** register using the instruction *mov.s*/*mov.d*. Note that the *mov.d* instruction moves the contents of two registers. That is, if you run the following instruction:

*mov.d $f12, $f0* # assume **$f0** contains to the 64-bit double 0x12345678\_A5A5A5A5  
 # where **$f1** contains 0x12345678 and **$f0** contains 0xA5A5A5A5

the contents of register **$f0** will get moved (copied) into register **$f12** while the contents of register **$f1** will get moved (copied) into register **$f13**.

Other times, you may want to move values from standard **$sX** or **$tX** registers into floating-point registers. Of course, there are special instructions to move values into the floating-point coprocessor – *mfc1*/*mfc1.d* (“*move-from-coprocessor1*”) and *mtc1*/*mtc1.d* (“*move-to-coprocessor1*”). Just like the other double-precision instructions the *mfc1.d* and *mtc1.d* instructions operate on two registers. If you run the following instruction:

*mfc1.d $a0, $f0* # assume **$f0** contains to the 64-bit double 0x12345678\_A5A5A5A5  
 # where **$f1** contains 0x12345678 and **$f0** contains 0xA5A5A5A5

the contents of register **$f0** will get moved (copied) into register **$a0** while the contents of register **$f1** will get moved (copied) into register **$a1**. This is a bit-for-bit move. The registers **$a1**+**$a0** will contain the double-precision floating-point value that was stored in **$f1**+**$f0**. That is, the data is **not** automatically converted into a two’s-complement integer simple because it was moved into non-floating-point registers.

Moving data into the floating-point coprocessor works similarly but uses the *mtc1*/*mtc1.d* instruction. One oddity of the *mtc1*/*mtc1.d* instruction is that the **source operand is supplied before the destination operand**. If you run the following instruction:

*mtc1.d $a0, $f0* # assume **$a1** contains 0x12345678 and **$a0** contains 0xA5A5A5A5

the contents of registers **$a1**+**$a0** will get moved (copied) into registers **$f1**+**$f0**. That is, **$f1** will get set to 0x12345678 and **$f0** will get set to 0xA5A5A5A5. Note the implicit use of the **$a1** register due to the double-precision instruction.

Table 2: Additional single-precision, and double-precision floating-point instructions

|  |  |  |
| --- | --- | --- |
| **Integer Operations** | **Single-precision Floating-Point Ops** | **Double-precision Floating-Point Ops** |
| add | add.s | add.d |
| sub | sub.s | sub.d |
| abs | abs.s | abs.d |
| mul | mul.s | mul.d |
| div | div.s | div.d |
| lw | l.s | l.d |
| sw | s.s | s.d |
| move | mov.s | mov.d |
| N/A | mfc1 | mfc1.d |
| N/A | mtc1 \*\* | mtc1.d \*\* |

**Instructions to Convert Data**

Populating **$fX** registers can be done using the *l.s*/*l.d* instruction as shown earlier. With the *l.s*/*l.d* instruction **$fX** registers can be loaded with values that are stored in memory. Another method to get floating-point values into an **$fX** register is to first move an integer into the floating-point coprocessor using the *mtc1*/*mtc1.d* instruction and then convert that integer into either a single-precision or double-precision floating-point number using special family of conversion instructions. The family of conversion instructions has the format *cvt.X.Y* where a value is converted from format *Y* to the new format *X*. The values for *X* and *Y* can be “*w*”, “*s*”, or “*d*” to specify conversion to/from 32-bit words, 32-bit single-precision floating-point values, or 64-bit double-precision floating-point values, respectively. There are a total of six instructions in this conversion family: *cvt.w.s* / *c vt.w.d* / *cvt.s.w* / *cvt.s.d* / *cvt.d.w* / *cvt.d.s*. As an example, assume that you want to populate the register **$f16** with the double-precision floating-point value 25.0. You can use the following sequence of instructions:

*li $t0, 25* # load **$t0** with integer value 25 (this is NOT the double 25.0)  
 *mtc1.d $t0, $f16* # move **$t1**+**$t0** into **$f17**+**$f16** (note the source operand is first)  
 *cvt.d.w $f16, $f16* # convert contents of **$f17**+**$f16** into a double and store back into **$f17**+**$f16**

After the *cvt.d.w* instruction, registers **$f17+$f16** will contain the double-precision floating-point value 25.0.

Here are some additional examples:

*cvt.d.s $f0, $f1*  # convert single-precision value in register **$f1** into a double and store in **$f1**+**$f0**  
 # Note that this WILL overwrite the original single-precision value in register **$f1**

*cvt.w.s $f0, $f1*  # convert single-precision value in register **$f1** into a 32-bit integer and store in **$f0**

**3. Procedure**

**3.1 Calculating a Square Root via Newton’s Method**

[Newton’s Method](https://en.wikipedia.org/wiki/Newton%27s_method) is a method for approximating the roots of mathematical functions. In general, if *x* is a root of a function *f(x)*, repeated application of the function: can find a suitable value of *x* at some desired level of precision. After each iteration, xn is updated with the previous xn+1 and a new calculation is performed. The error in estimating x is found by .

In this part, you will need to implement a MIPS assembly program that uses Newton’s method to approximate the square root of a value *N*. That is, you will need to approximate the value of . When approximating the roots of , use repeated application of the function below until you have achieved the desired level of precision:

After each application of the above function, use the error computation shown above to determine if the error value is small enough to satisfy the desired level of precision. If not, apply the above function again. A C implementation that shows how to use Newton’s method to approximate the roots of is shown in Figure 1. An example of how to call that function is shown in Figure 2.

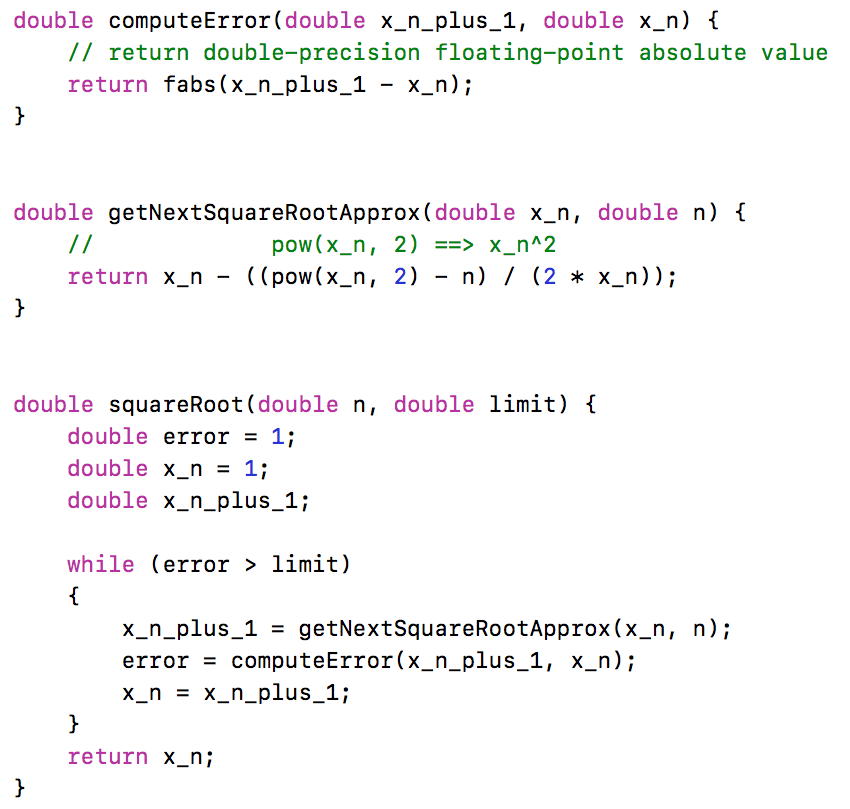


Figure 1: Newton’s method to approximate

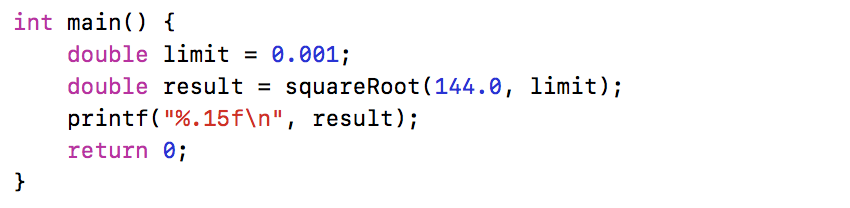


Figure 2: Driver code for Newton’s method approximation

**Programming Tasks**

In this lab, you will be writing several procedures. Remember to follow conventions. A new convention, not previously discussed, is that all **$fX** registers should be treated as preserved registers. That is, a CALLEE should preserve any **$fX** registers that it uses and restore those **$fX** registers for the parent prior to returning.

1) In MARS, open the file called lab08\_part1.asm in the MARS Editor. You’ll see the .data segment has already been populated with values for *n* and *limit*. The value *n* is the value that will be input into the “*squareRoot”* procedure. The *limit* value specifies the precision required for the approximation. By making *limit* smaller (i.e. adding more zeros), you can increase the precision of your approximation. You’ll also see some initialization code and driver code has been provided for you. You should not change, add, or remove any code from the *initialize* or *ece260\_main* sections. Finally, you should see labels for the three procedures that you are required to implement: “*computeError*”, “*getNextSquareRootApprox*”, and “*squareRoot*”.

2.a) Implement the “*computeError*” procedure. The procedure takes two double-precision floating-point values as input. It returns a single double-precision floating-point value. Recall that each double-precision floating-point value uses two 32-bit registers. Therefore, passing two double-precision floating-point values to the procedure will require all four **$aX** registers. Likewise, returning a double-precision floating-point value from the procedure will require both **$v0** and **$v1**. Since you can only perform floating-point computation using **$fX** registers, you will need to move input arguments from **$aX** registers into **$fX** registers. Likewise, you will need to move your computed result from **$fX** registers into **$vX** registers to return the result. **HINT: You may need to add more space to the stack than expected to get the stack pointer doubleword aligned. Start the procedure by AND-ing the stack pointer with the value 0x7. The result of this AND instruction will tell you how many bytes you need to move your stack pointer to get it double-word aligned. Subtract that value from the stack pointer before doing any other stack manipulations so you know that stack pointer is double-word aligned.**

2.b) Unit tests have been provided that will allow you to do some automated testing on the *“computeError”* procedure before continuing. Do not move onto the next step until you are certain that your “*computeError”* procedure operates correctly. To run the supplied unit tests for the “*computeError*” procedure, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab08\_More\_Fun\_with\_Floats  
make test\_part1\_computeError

You will see output that indicates if your procedure 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.a) Implement the “*getNextSquareRootApprox*” procedure. The procedure takes two double-precision floating-point values as input. It returns a single double-precision floating-point value. The double-precision floating-point result should be returned using both **$v0** and **$v1**. As shown in Figure 1, the “*getNextSquareRootApprox*” procedure should compute the result of the following equation and return the value :

**HINT:** You will need to initialize one of your **$fX** registers with the double-precision floating-point value 2.0. Try moving an integer value into an **$fX** register and then converting it into a double.

3.b) Unit tests have been provided that will allow you to do some automated testing on the “*getNextSquareRootApprox*” procedure before continuing. Do not move onto the next step until you are certain that your “*getNextSquareRootApprox*” procedure operates correctly. To run the supplied unit tests for the “*getNextSquareRootApprox*” procedure, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab08\_More\_Fun\_with\_Floats  
make test\_part1\_getNextSquareRootApprox

You will see output that indicates if your procedure 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.a) Implement the “*squareRoot*” procedure. The procedure takes two double-precision floating-point values as input. It returns a single double-precision floating-point value. The double-precision floating-point result should be an approximation of and should be returned using both **$v0** and **$v1**. This procedure will use your “*computeError*” *and* “*getNextSquareRootApprox*”procedures. Note that this procedure is called from the supplied driver code under the *ece260\_main* label. The supplied driver code loads the values *n* and *limit* from the *.data* section, moves them into **$a0** and **$a2** respectively, and performs a jump-and-link (*jal*) to call the “*squareRoot*” procedure. When the “*squareRoot*” procedure returns, the result is moved into register **$f12** and printed using a *syscall* instruction. **HINT:** For this procedure, you will need **$fX** registers for *n*, *error*, *limit*, *x\_n*, and *x\_n\_plus\_1*. Initialize these registers before starting your loop.

4.b) Unit tests have been provided that will allow you to do some automated testing on the “*squareRoot*” procedure before continuing. Do not move onto the next step until you are certain that your “*squareRoot*” procedure operates correctly. To run the supplied unit tests for the “*squareRoot*” procedure, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab08\_More\_Fun\_with\_Floats  
make test\_part1\_squareRoot

You will see output that indicates if your procedure 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.c) Compute several square roots by changing *n* in the .data section. Fill in Table 3 with your results. For some entries in your table, try changing your *limit* value to 0.00001 or 0.1 to see what happens.

Table 3: Square Roots

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **Expected Value** | **Limit Value** | **Computed Value** |
| 144.0 | 12.0 | 0.001 | 12.000000012408687 |
| 4.0 | 2.0 | 0.01 | 2.0000000929222947 |
| 16.0 | 4.0 | 0.00001 | 4.000000000000051 |
| 4096.0 | 64.0 | 0. 1 | 64.00001438575758 |

**Test Your Code**

Although you have already tested that various procedures above, you must still test to ensure that all of your procedures work together. Test all of your code together using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab08\_More\_Fun\_with\_Floats  
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.

**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 Lab08\_More\_Fun\_with\_Floats  
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.**