CS 231 – Lab 7 – Floating-Point Assembly

In today's lab we will be writing assembly code that deals with floating-point numbers. We need to become familiar with how floating-point numbers are declared in the data segment, and how to use specific floating-point instructions. There are two kinds of floating-point numbers: single precision and double precision. In this lab we will just be using double-precision numbers. This means each number will occupy 64 bits, or 8 bytes, or 2 adjacent floating-point registers.

Create a new folder called lab7 on your USB drive or account, and do all of today's work in there. From the class Web site, you should see a folder called lab7. Download fcmp.s and merge.s. We will be modifying these programs today.

Floating-point Comparisons

When you want to compare two floating-point numbers, they must first reside in registers. And since we are dealing with double precision, we have to reference these numbers with even-numbered registers, such as $f12 and $f14.

Comparing two floating-point registers takes two instructions. First, there is a comparison instruction such as c.eq.d (compare equal double) and c.lt.d (compare less-than double). For single precision there are analogous instructions c.eq.s and c.lt.s. The result of a comparison is either true or false, so the second instruction will be a branch based on the outcome of the comparison. We either use bc1t (branch co-processor 1 true) or bc1f (branch co-processor 1 false) as appropriate. The funny name "co-processor 1" simply refers to the floating-point hardware.

For example, the following code tests to see if the values in $f12 and $f14 are equal. If they are, control will branch to the instruction labelled "do_true_case".

```
c.eq.d  $f12, $f14
bc1t  do_true_case
```

Note that the comparison instruction (c.eq.d) takes two register operands. There is no explicit destination as we saw for beq and other integer branches. Also note that when we write the floating-point branch instruction, we only need to specify the target to jump to. bc1t and bc1f do not take any register operands.

1. At this time, take a look at the program fcmp.s. Print out a hard copy so you can make notations. This unfinished program asks the user to enter two floating-point numbers, and then outputs their relationship (equal to, less than, etc.). Notice the strings declared in the data segment.

2. Run the program. What did it do? Examine the code that arranges for these output strings to appear on the screen. You should notice that it forgot to make some comparisons. Modify the program so that it performs the proper comparisons and can print the correct messages for the user. √

Merging Two Sorted Arrays

In this next example program, we will practice with arrays of double, loading and storing values, as well as making comparisons on floating-point numbers.

Print out the file merge.s. You will be making incremental modifications to this program.

The purpose of this program is to take two sorted lists (arrays) of doubles, and merge them into a larger sorted list. At the top of the program, I have declared list 1 and list 2 each to contain 10 specific values. For simplicity we use the value 0.0 to indicate the end of a list.

The third list has been declared to occupy up to 800 bytes, enough to store 100 doubles including the sentinel value 0.0. Of course, this will be more than enough room.

If you run the program, you notice that it doesn't do much yet. What we need to do is show the user what the first 2 lists look like before we begin the merging, and when we're done, we can show the user what the final list looks like.

1. First, implement the procedure "print_list" at the bottom of the program. I recommend that you use $s4 to represent the base address of the list to print. Notice that every time we call print_list we copy a specific list's base address into $s4. You may use $t4 to point to individual entries in the list. The values in the list should be printed one per line. √

2. Now it's time to merge. I have provided the essential algorithm for accomplishing this in a comment in the code, labelled "ALGORITHM TO PERFORM MERGE". There are 4 cases. In the first two cases, one of the lists (either list 1 or list 2) is now empty, so all we need to do is empty the remainder of the other list into list 3. Under the comment is space for you to implement this algorithm. Focus on the first two cases for now. (To save time, I recommend you use the empty_list function. This function assumes that $t4 is pointing to the first element you want to copy into the third list.) To test your code, it may be helpful to change the declarations of the arrays so that one of the lists is already empty from the start (by inserting a 0.0 at the start of a list). √

3. The last two cases of the merge algorithm comprise the typical situations. Most of the time you will need to figure out what the appropriate next number should be to put into list 3. Here, you need to use a comparison instruction like c.lt.d, and then branch the appropriate way. You will be using load and store instructions as well to move the values to and from registers. Also, be careful that you increment your array pointers by 8 whenever you go on to the next element: each double-precision number occupies 8 bytes. I have provided a diagnostic function "notify move" that you may use to print which element is going into the third list. It is designed to print the floating-point number in $f12. √