CS224 - Spring 2017 - Lab #3 (Version 1 March 12)

Sorting Floating Point Numbers with MIPS Assembly Language Routines

Dates: Section 1, Monday, 20 March, 13:40-17:30
Section 3, Tuesday, 21 March, 13:40-17:30
Section 2, Wednesday, 22 March, 13:40-17:30
Section 4, Thursday, 23 March, 13:40-17:30
Section 5, Friday, 24 March, 8:40-12:30
Section 6, Friday, 24 March, 13:40-17:30

Purpose: To understand and use system calls for random numbers, dynamic memory allocation, and system time; argument passing and stack use with subroutines; bit and field operations (for masking and shifting); sorting algorithms; and limitations on performance (due to algorithm "big O", input size, run-time patience, etc).

DUE DATE/TIME OF PART 1 --SAME FOR ALL SECTIONS

a. Please drop your written Preliminary Design Report into the box provided in front of the lab before 12:59 pm on Monday March 20th. No late submissions will be accepted!
b. Please **upload your Preliminary Design Report** to the Unilica Assignment for Preliminary Work by 13:30 o'clock on Monday March 20th.

DUE TIME OF PART 2—DIFFERENT FOR EACH SECTION:

a. You have to demonstrate your Part 2 lab work to the TA for grade by **12:15** in the morning lab and by **17:15** in the afternoon lab. Your TAs may give further instructions on this.
b. At the conclusion of the demo for getting your grade, you will **upload your lab work** to the Unilica Assignment, for similarity testing by MOSS. Please see Part 3 for further instructions on MOSS submission.

Part 1. Preliminary Work and Preliminary Design Report (30 points)

a. Research and read about sorting algorithms. The excellent article on Wikipedia is a good starting point. Books and other internet sources can also be consulted, until you have a good understanding of the various kinds of algorithms and their pros and cons.
b. Determine the two algorithms you will use for Part 2 e and Part 2 f. One should be a slow algorithm with $O(n^2)$ performance, the other a fast algorithm with O(n log n) performance. [Suggestion: try to increase your skills by learning to implement (i.e. write the code, test it and debug it) new things you don't already know]
c. (2 points) Your Preliminary Design Report should be neat hard-copy document, stapled together, prepared with a word processor. The cover page should provide the following information:

CS224
Section # ___
Spring 2017
Lab # ____
Your Full Name
Your Bilkent ID

d. (28 points) Each section below should be contained in an independent section, and each section should begin on a new page. Each section should have its own header explaining what it is and does, followed by a program listing. Each of the following sections is worth 4 points:
   i. The code for *special_case*, as described in Part 2 a.
   ii. The code for *get_rand_FP*, as described in Part 2 b.
   iii. The code for *FillArray*, as described in Part 2 c.
   iv. The code for *CompareFP*, as described in Part 2 d.
   v. The code for *SlowSort*, as described in Part 2 e.
   vi. The code for *FastSort*, as described in Part 2 f.
   vii. The code for *Lab3*, as described in Part 2 g.

Part 2. Building a program to sort Floating Point numbers, using subroutines (70 points)

All MIPS assembly code must obey the convention for the use of registers! Even when you have a full-view of all routines (main, non-leaf and leaf), you must still not violate the protocol, saying "it is OK to use this register" or "I don't have to protect that on stack", etc. Remember: code is maintained over a lifetime, so all users of that code must have the assurance that the code fully obeys the convention for register usage!! Build and use good habits now, even as a student, of being a "team player" who cooperates with everyone else!

a. **(10 points)** Write a subprogram, called *special_case*, which receives one argument in $a0 and delivers one result in $v0. If the input argument, interpreted as an IEEE 754 32-bit floating point number, is one of the special cases (+/- ∞, NaN, zero, denorm), then the return value is 1 , meaning "yes". Otherwise, the return value is 0 meaning "no". Your *special_case* routine must not use any floating point instructions; it can only use the integer instructions to look at the bit field representing the exponent E of the number and determine if it is all 0s or all 1s. [Hint: use shifting instructions, and/or masking instructions, to isolate the bits in that field in order to test them.]

To show that this routine is working correctly and get 10 points, you should write a very simple test program that gives values to the *special_case* routine, and receives the result back. Using MARS, you must prove to the grader that for *special_case* inputs, the return value is 1, and for regular f.p. numbers, the return value is 0.

b. **(10 points)** Write a subprogram, called *get_rand_FP*, that uses the system call for obtaining a random integer and uses the *special_case* subprogram that you have created above, to implement the following: when called, *get_rand_FP* obtains a 32-bit random integer and checks to see, if it were to be interpreted as an IEEE 754 32-bit floating point number, whether that number is a special case, or a regular single-precision f.p. number. If it is a special case, it tries again, asking for a new random integer, continuing until it gets a value which would be interpreted as a regular single-precision IEEE 754 floating point value. The subroutine *get_rand_FP* returns a single value in $v0, which is the non-special-case f.p. number.

To show that this routine is working correctly and get 10 points, you should write a very simple test program that calls *get_rand_FP* in a loop, and stores the returned values into memory. Using MARS, you must prove to the grader that *get_rand_FP* is getting random values which are non-special-case f.p. numbers and which don't repeat.

c. **(10 points)** Write a subprogram, called *fillArray*, which receives one argument in $a0 and delivers one result in $v0. The input argument is the number N of array elements that will be needed, and the return value is the starting address of the array of single-precision floating point non-special-case values of the filled array. The *fillArray* subprogram should repeatedly call *get_rand_FP* to obtain the values it needs to fill the array. Since the size of the array (N) is an input parameter, the array cannot be declared in advance (at assembly time) and instead must be determined at run-time. This means that the array is a dynamic data structure and must be located in dynamic data memory (see Figure 6.31). [Hint: dynamic data memory is obtained at runtime from the operating system. In MIPS, you can do this with a syscall in your code.]

To obtain the 10 points, you must show that your code is working correctly for the largest possible value of N that MARS can handle, and that any larger value (even N+1) will not work correctly. So you will need to write a very simple test program that calls *fillArray*, and then use it with different values of N to find, by experimentation, the largest array that can be created. Using MARS, you must prove to the grader that *fillArray* is getting filling the array with N random non-special-case single-precision f.p. numbers, but that it cannot make an array with N+1 such values.

d. **(10 points)** Write a subprogram, called *CompareFP*, which receives two arguments in $a0 and $a1, and delivers two results in $v0 and $v1. The two arguments are each 32-bit single-precision floating point numbers, and the two results are the same two 32-bit single-precision f.p. numbers. The *CompareFP* subroutine determines which one is the larger number, and returns that one in $v0 and the smaller one in $v1. *CompareFP* cannot use any MIPS floating point operations, it must use only integer instructions to determine the sign, exponent, and fraction parts of the two numbers, and compare them. [Hint: if the signs are different, the larger number is determined. If the signs are the same but the exponents are different, the larger number is determined. If both sign and exponent are the same, then

the fraction bits will determine which number is larger. A floating point coprocessor will implement this comparison in algorithm, but you will implement it in software using integer instructions.

To show that this routine is working correctly and get 10 points, you should write a very simple test program that repeatedly calls *CompareFP* with different pairs of test input values, and stores the returned values into memory. Using MARS, you must show the grader that *CompareFP* is correctly doing the comparison for cases when the numbers are different in sign, in exponent, and in fraction parts. And you must show that the correct results are obtained for each input pair, independent of which is in $a0 and which is in $a1

e. **(10 pts.)** Write a slow sorting subprogram, called *SlowSort*, that sorts an array of IEEE 754 single-precision non-special-case values from largest to smallest in time proportional to $O(N^2)$. The subroutine *SlowSort* will have 2 input arguments: the starting address of the array of values to be sorted will be in $a0, and the number of values in the array will be in $a1. *SlowSort* will sort the values in place, so that upon return to the callee, the N values are sorted from greatest to smallest, and the largest value is at the memory location which was originally given in $a0.

To show that this routine is working correctly and get 10 points, you should write a very simple test program that does the following: creates a small array of f.p. numbers in memory, then calls *SlowSort.* Using MARS, you must prove to the grader that *SlowSort* correctly sorts the array values from greatest to least, and that it does it using a correct implementation of one of the "big O"(N^2) sorting algorithms. [Hint: you will need to use breakpoints and single-step mode in MARS "execution" of your MIPS program, in order to trace the steps of the sorting algorithm. NO CREDIT will be given <u>if you cannot explain what your sorting algorithm's code is doing, and why !!</u>

f. **(10 pts.)** Write a fast sorting subprogram, called *FastSort*, that sorts an array of IEEE 754 single-precision non-special-case values from largest to smallest in time proportional to O(N logN). The subroutine *FastSort* will have 2 input arguments: the starting address of the array of values to be sorted will be in $a0, and the number of values in the array will be in $a1. *FastSort* will sort the values in place, so that upon return to the callee, the N values are sorted from greatest to smallest, and the largest value is at the memory location which was originally given in $a0.

To show that this routine is working correctly and get 10 points, you should write a very simple test program that does the following: creates a small array of f.p. numbers in memory, then calls *FastSort.* Using MARS, you must prove to the grader that *FastSort* correctly sorts the array values from greatest to least, and that it does it using a correct implementation of one of the "big O"(N logN) sorting algorithms. [Hint: you will need to use breakpoints and single-step mode in MARS "execution" of your MIPS program, in order to trace the steps of the sorting algorithm. NO CREDIT will be given <u>if you cannot explain what your sorting algorithm's code is doing, and why !!</u>

g. **(10 pts.)** Write a main program, called *Lab3*, which performs the following tasks:

--welcomes the user, and explains what the program will do

--prompts the user for a number N which will be the size of the array used

--tests N for correctness, and if acceptable, creates and fills an array[N] with IEEE 754 single-precision non-special-case values. If not correct, it re-prompts for a correct value of N.

--reports to the use the number of seconds and milliseconds required for the code to execute the above "create and fill" tasks

--reports to the user that it is ready to do the sorting task, asks which sort is to be done (N^2 or N logN) and gets the response

--performs the sort of the N-element array of values

--reports to the use the number of seconds and milliseconds required for the code to execute the sorting task, using the algorithm that was selected.

--asks if the user wants to do the above tasks again. If yes, prompt for N and continue as above. If no, then output an exit message and end cleanly.

To show that this *Lab3* program is working correctly and get 10 points, you should write it (using calls to *FillArray*, *SlowSort* and *FastSort*, plus other calls and I/O syscalls as needed) and get it running, then prove to the grader that it works by demonstrating correct behavior. To get the full 10 points, experiment and find the value of N that is needed:

- For the SlowSort to take 30 seconds
- For the FastSort to take 30 seconds
  (If you cannot find it by experimentation because N would be larger than MARS allows, then calculate it based on measurements and justify your answer)

Part 3. Submit your code for MOSS similarity testing

After demonstrating your working (or partially working) codes to the grader, you should immediately submit your MIPS codes for similarity testing to the Unilica > Assignment specific for your section. You will upload one file, named **name_surname_SecNo_MIPS.txt,** containing the 6 subroutines and 1 main program described above in Part 2. Be sure that the file contains exactly and only the codes which are specifically detailed in Part 2. Check the specifications! *Even if you didn't finish, or didn't get the MIPS codes working correctly, you must submit your code to the Unilica Assignment for similarity checking.* Failure to submit your codes will result in a lab score of 0. Your codes will be compared against all the other codes in all sections of the course, by the MOSS program, to determine how similar it is, as an indication of plagiarism. **So be sure that the code you submit is code that you actually wrote yourself!** [Warning: DON'T use code found somewhere on the internet, since others might also find and use it, and MOSS will determine that yours is similar to theirs!] All students must upload their Part 2 code to Unilica > Assignment while the TA watches, at the end of your demo-for-grading time. Submissions made without the TA observing will be deleted, resulting in a lab score of 0.

Part 4. Cleanup

1) After saving any files that you might want to have in the future to your own storage device, erase all the files you created from the computer in the lab.
2) When applicable put back all the hardware, boards, wires, tools, etc where they came from.
3) Clean up your lab desk, to leave it completely clean and ready for the next group who will come.

LAB POLICIES

1. You can do the lab only in your section. Missing your section time and doing in another day is not allowed.
2. Students will earn their own individual lab grade. The questions asked by the TA will have an effect on your individual lab score.
3. Lab score will be reduced to 0 if the code is not submitted for similarity testing, or if it is plagiarized. MOSS-testing will be done, to determine similarity rates. Trivial changes to code will not hide plagiarism from MOSS—the algorithm is quite sophisticated and powerful. Please also note that obviously you should not use any program available on the web, or in a book, etc. since MOSS will find it. The use of the ideas we discussed in the classroom is not a problem.
4. You must be in lab, working on the lab, from the time lab starts until your work is finished and you leave.
5. No cell phone usage during lab.
6. Internet usage is permitted only to lab-related technical sites.
7. For labs that involve hardware for design you will always use the same board provided to you by the lab engineer.