

Systems Internals and Assembly Language: Assignment 4, Fall 2006

Do **ONE** of the following **individual** assignments. Write a report, and put the requested program in an appendix. **Q Comment your code well** so that the code is easy to read. **Q Show an example of the input and output** that is produced by the program. **Q Mention any limitations** that your program has. If you do an extra credit part, label it clearly in your documentation.

1. Sort. Implement the sort procedure on page 165 and 170. First get it running as is. See my file **Q:\InstructorFiles\Chase_Gene\assembly\swap.s**. You may initialize your array in a way similar to the way **swap.s** does it. Be careful though, since **swap.s** assumes the array contains bytes, not words. You will need to make several changes, such as including an **.align** directive for words, and accounting for the array elements being 4 bytes each.

Then modify it in one of the following ways—or **for extra credit**, both ways. Save your original working version, in case you experience difficulties with the modifications.

1.A: Floating point. Modify the sort program so that it sorts floating point numbers. You can initialize the array using the assembler directive **.float** (see Appendix A) and you can load the array values using floating point instructions like **l.s** and so on.

1.B: Pointers. Modify it so it uses pointers to the array elements, rather than recomputing addresses each time. For example, currently, the inner loop computes the address of element $v[j]$. It does this by multiplying j by 4 and adding it to the address of the array v . At the end of the inner loop, it subtracts one from j .

If it used pointers instead, it would simply subtract 4 from the address each time. To make this work, the outer loop must be responsible for computing the address of the original $v[j]$. The logic of it is tricky to get right, but if it works, the code will be more efficient, in a way similar to the example in section 3.11.

2. Bit-oriented programming. To practice using bit-oriented operations, write a program that stores a set of integers in a boolean array format. The program should have two phases:

Phase 1: Input some integers in the range 0..255, until the user enters a negative number.

Phase 2: Again let the user enter numbers, but this time, the computer should tell the user whether the number had been entered in phase 1. If the user enters a negative number, exit the program.

Your storage of the numbers entered in phase 1 should be boolean-array based. To represent that the number 100 is in the set, you turn bit 100 on. Since you need to represent 0..255 you need 8 registers, since $256/32=8$. Use registers $\$t0$ through $\$t7$ for this purpose. These should be initialized to all zeroes, and then if the user enters a number, change the corresponding bit to a one. For example, if the user enters the number 100, since $100/32 = 3$ remainder 4, put a one in bit 4 of register $\$t3$.

You will need to write code that converts the 100 into the appropriate register and bit. (This is similar to how two-dimensional arrays in Java are implemented in assembly, except that arrays would use memory instead of register bits.) Do this conversion in a constant number of steps, using division to get the quotient and remainder, instead of using a loop.

Extra credit 1: Have Phase 2 use a function that returns true or false depending on whether its integer parameter has been entered in Phase 1. Since you need to pass the eight registers as parameters, and by convention there are only four registers $\$a0$ through $\$a3$ that are used for parameter passing, you must pass some of the parameters using the stack. This is called “spilling registers.”

Extra credit 2: Use a **switch** case statement as show in the text instead of an “if / else if” construction.

3. Display of floating point representation Write an assembly program that can be used as a pedagogical illustration of the internal floating point representation. The program should be able to read in a floating point number and then output the binary representation in 3 parts, with a decimal interpretation of the exponent part. The dialogue might look something like this:

Enter a number: 2.5

In binary: sign= 0

exponent= 1000 0000, which is excess-127 notation for decimal 1

mantissa= 0100 0000 0000 0000 0000

Hint: The instruction **mfc1 \$t1 \$f12** moves the contents of floating point register $\$f12$ to general purpose register $\$t1$, where it can be torn apart. **Extra credit:** Get more fancy or include an option for double precision.

4. Do something interesting of your own design. Check with me first to make sure it is a suitable idea.