<u>CSE 30321</u> – <u>Computer Architecture I</u> – <u>Fall 2010</u>

Lab 02: Programming in MIPS Assembly
Total Points: 100 points
Assigned: September 21, 2010
Due: October 7, 2010

1. <u>Goals</u>

Looking back to the course goals, this lab is directly related to Goal #4. Namely, at the end of the semester, you should be able to "explain how code written in (different) high-level languages (like C, Java, C++, Fortran, etc.) can be executed on different microprocessors (i.e. Intel, AMD, etc.) to produce the result intended by the programmer."

More specifically, the primary learning goal for this lab is to become more familiar with how computer hardware allows for the execution of recursive procedure calls.

2. <u>Introduction</u> <u>and</u> <u>Overview</u>

For this lab, you will be working with a recursive *mergesort* function. Mergesort will be (or may already have been) discussed in detail in your data structures class. However, a brief explanation is also included here.

Megresort can be used to sort an unordered list. The list is essentially subdivided into smaller and smaller lists, these are sorted, and the lists are then merged back together. An example is shown in Figure 1 below:

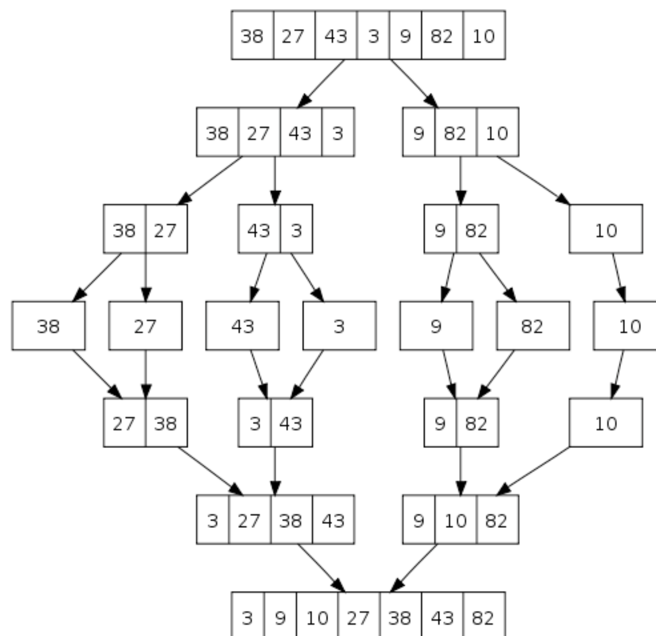

Figure 1: Graphical example of a merge sort[1].

"Mergesort can sort a file of N elements in N log N time even in the worst case. The prime <u>disadvantage</u> of *mergesort* is that extra space is required – equivalent to the size of the list (N) – to perform the sort. However, mergesort can be implemented so that it accesses data primarily in a sequential manner (i.e. one item after another). This can be <u>advantageous</u> – i.e. if sorting a linked list, where only sequential accesses are possible."[2]

[1] http://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/500px-Merge_sort_algorithm_diagram.svg.png
[2] Paraphrased from: "Algorithms in C" by Robert Sedgewick

In this lab, you will need to write a recursive *mergesort* function (and the corresponding *sort* function) in MIPS assembly. To help you get started, compilable C-code that successfully performs a *mergesort* of a 32 number array can be found at:

/afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/02/mergesort.c

If you examine this code carefully, you will see that there is a recursive function (*mergesort*) and a non-recursive function (*sort*). The first call to *mergesort* sorts one half of the array. The second call to *mergesort* sorts the other half of the array. The *sort* function then ensures that the elements in different array segments are merged back together in the proper order. (You will not need to write the *display* function in mergesort.c in MIPS assembly.)

Note that detailed comments and print statements have also been added to help you understand how these two functions can be combined to sort the array. (If you want to understand the functionality of merge sort better, you might direct the output of the code to a file to see – step-by-step – how the array is sorted.)

3. Utilities

To help you design, debug, and test your code (or any MIPS code), you should use the XSPIM utility ("SPIM" = "MIPS" backwards).

3.1 Overview

XSPIM is software that will help you to simulate the execution of MIPS assembly programs. It does a context and syntax check while loading an assembly program. In addition, it adds in necessary overhead instructions as needed, and updates register and memory content as each instruction is executed. Below, is a tutorial on how to use XSPIM.

- Go to the directory where your assembly language program is stored.

- Type /afs/nd.edu/user14/csesoft/bin/xspim at the prompt.
    - (Adding the full directory to your path, or making an alias for XSPIM will make the program easier to launch in the future.)
    - Also, note that the source for XSPIM can be found at:
        - http://pages.cs.wisc.edu/~larus/spim.html
        - (I have found it to be quite easy to compile and run locally.)

- A window will open as shown in Figure 2. The window is divided into five vertical sections:
    1. The *Register Content* section displays the content of all registers. (You only need to be concerned with the registers that we have discussed in class; ignore any others.)
    2. Each button in the *Command Console* corresponds to a command supported by the simulator.
    3. The *Text Segment* displays the MIPS instructions loaded into memory to be executed. From left to right, we have the memory addresses, the corresponding memory contents in hex, the actual MIPS instructions, and the corresponding assembly instructions.
    4. The *Data Segment* displays memory addresses and their values in the data and stack segments of the memory.
    5. The *Information Console* lists the actions performed by the simulator.

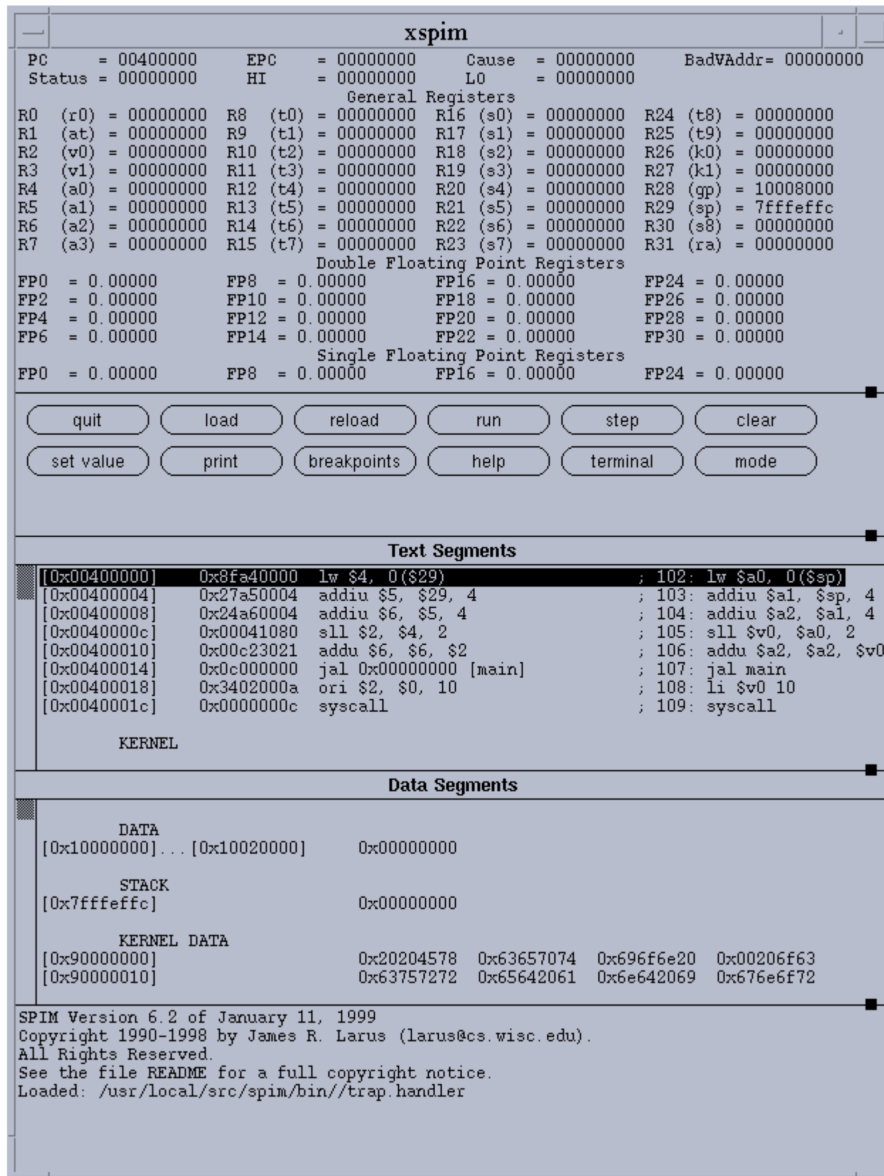

Figure 2: XSPIM window.

- The functions of the command buttons in the *Command Console* are summarized below:
  - quit: Exit from the simulator
  - load: Read an assembly program file into memory
  - reload: Reload the assembly program file
  - run: Execute a program to completion
  - step: Advance the execution of a program by a given step size
  - clear: Reinitialize registers or memory. (You have a choice of what to clear. To reload a program you modified, you need to clear both the registers and the memory.)
  - setvalue: Set a value in register or memory
  - print: Print a value in register or memory
  - breakpoint: Set or delete a breakpoint
  - help: Display the above message

- We will not use *terminal* and *mode* in this lab.

3.2 An Example Program

To help you get started with XSPIM, I've placed an example program that you can load (and immediately run) in the simulator. ***It will probably be helpful for you to quickly step through this example, as it will introduce you to a few syntactical statements that you will need to use in other programs that you write (to be run in XSPIM).***

This program (as seen in Figure 3) loads 5 pieces of data in memory, adds them together, and store's the result in memory. Be sure that you understand:

- How you can load data into memory
- How you can get the starting address of a data array
- How you exit a program cleanly


```
        #
        # Simple Summation program
        # cse30321
        #
        .data                           # Put Global Data here

N:      .word  5                        # loop count
X:      .word  2,4,6,8,10               # array of numbers to be added
SUM:    .word  0                        # location of the final sum


        .text                           # Put program here

        .globl main                     # globally define 'main'
main:   lw      $s0, N                  # load loop counter into $s0
        la      $t0, X                  # load the address of X into $t0
        and     $s1, $s1, $zero         # clear $s1 aka temp sum
loop:   lw      $t1, 0($t0)             # load the next value of x
        add     $s1, $s1, $t1           # add it to the running sum
        addi    $t0, $t0, 4             # increment to the next address
        addi    $s0, $s0, -1            # decrement the loop counter
        bne     $0, $s0, loop           # loop back until complete
        sw      $s1, SUM                # store the final total

        li      $v0, 10                 # syscall to exit cleanly from main only
        syscall                         #  this ends execution
        .end
```


Figure 3: Example program to load in XSPIM.

To run this program in XSPIM:

1. Download the program source from:
   /afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/02/addarray.s
2. Click on the "load" button and type the name of the program (i.e. addarray.s)
3. You can then run the program in 1 of 2 ways:
   a. You can simply press the "run" button – all instructions will be executed, and the final contents of memory and the register file will be reflected in the XSPIM window.
   b. You can step through the program in increments from 1 to N instructions by pressing the "step" button. For example, if you set the step size to 1, a new instruction will be executed every time that you press step in the popup window. The contents of memory, the register file, etc. will change dynamically. Additionally, the instruction being executed will be highlighted in the "Text Segment" window.

Note that at the conclusion of this program, $s1 should be equal to 0x1e (in hex) as the sum of 2_{10}, 4_{10}, 6_{10}, 8_{10}, and 10_{10} is 30_{10} (i.e. 11110_2 or $0x1e_{16}$). Also, note that when loading a program, you can type an absolute path to a .s file, but the default location is the directory that you are in when you launch XSPIM.

4. <u>Problem</u>

As a deliverable, you will essentially need to translate the provided C-code into MIPS assembly:

- Write the MIPS assembly for the *mergesort* and *sort* functions
- Your *mergesort* function must be recursive
- Mergesort should be called for the first time from main
- As is the C-code, your *mergesort* function should take as arguments the starting address of the array to be sorted, and the "endpoints" of the array segment that you wish to pass to *mergesort*.
- Your sort function will take the same arguments, but will also take the index of the "middle" element as an argument.
- Note: because of the way that XSPIM manages memory, you will also want to pass in the starting address of your "scratch space" (i.e. the address of the array *b* in the corresponding C-code).

Note that to simplify the code that you need to write, you can assume that the size of your array is a power of 2. (A real *mergesort* would need to handle the boundary cases in the merge loop that would make this lab much more complex.)

A skeleton has been provided at: /afs/nd.edu/coursefa.10/cse/cse30321.01/Labs/02/lab02.s

- Note that this skeleton includes the list of 16 numbers that you will need to sort.
- The scratch array *b* has also been initialized such that every element is equal to 0.

5. <u>What to Turn In</u>

- Put your code for lab02.s in the dropbox of one group member. (Create a folder called Lab02)
  - *Be sure that your code runs in XSPIM – as we will use it to test your code with a different array.*
- In the same directory, create a text file that lists the total number of instructions executed for your program.
- *There is no formal report required for this lab.*