CSE30321 Computer Architecture I
Lab4: MIPS Assembly and XSPIM

Assigned: Wed. October 8th. Due: Friday, October 17

A. Objectives

- Learn and understand the MIPS instruction set architecture
- Become familiar with MIPS assembly language and learn to program with it.
- Learn to use XSPIM, an assembler and simulator for MIPS assembly programs.
- Enhance understanding of the pointer concept in high level languages.

B. References

- Textbook by Patterson and Hennessy.
- Lecture Notes 4.
- Tutorial on `XSPIM`.

C. A Simple Exercise with XSPIM

C.1 What to Accomplish

- Find errors in a simple program
- Simulate the program with XSPIM

C.2 Procedure

1. Go over the tutorial on XSPIM. A link to it is on the Lab Resources page.
2. Copy the files in `/afs/nd.edu/courses/cse/cse30321.01/labs/lab4` to your own space.
3. One of the files that you have copied is `bubblesort.s`. It is an MIPS assembly implementation of the bubblesort algorithm. There are some simple bugs in the program. Find them, fix them, and document them in your lab report. You may also use *XSPIM* to find errors (but it may take longer).
4. Run the fixed code in XSPIM and answer the following questions:
   - How many bytes of **data** are used by the program?
   - How many instructions did XSPIM automatically add to the code between "main" and the first instruction? What are they? Why were they added?
   - What is the first instruction executed and where is it in memory?
   - What instructions does XSPIM expand the assembly instruction "`la $S3, A`" into? Why?
   - What is the value displayed for K in hex? What is its decimal equivalent? Where and how did you find the value for K?

- There is no "load immediate" in MIPS, but XSPIM can translate this into another single real machine instruction. What is it?

D. A Simple MIPS Assembly Program

D.1 What to Accomplish

- Write your own simple MIPS assembly program
- Test the assembly code with XSPIM

D.2 Procedure

1. Recall the program `FreqCalc.s` that you developed for the six-instruction processor in Lab2. The pseudocode for this program is repeated below:

   ```
   a = [1 24 15 24];
   i = 0;
   j = 24; //the value to search for
   count = 0;
   for(i = 0; i < 4; i++)
       if a[i] == j
       count++;
   ```

   Now assume that the length of array `a` is unknown but the array ends with "-1". Furthermore, assume that the value to search for (`j`) and the result (`count`) are stored in memory. Write a MIPS assembly program for the above pseudocode.

2. Simulate your program with `XSPIM` and keep track of how many instructions it takes to search an array of certain size.

3. Derive a general formula for the instruction count assuming array size of N and p percent of the items in the array being the value to search for.

E. Doubly-Linked List

E.1 What to Accomplish

- Implement an ordered (smallest to largest) doubly-linked list in MIPS assembly. You should implement procedures to insert, delete and find elements. Your `find` procedures *should be recursive*. Procedure `insert` allocates memory for a new node in the list. Procedure `delete` disconnects the node from the list, but does not need to deallocate memory.
- Test the assembly code with XSPIM

E.2 Procedure

1. You are given a sample program (`ll.c`) that shows how one can implement the doubly-linked list data structure in C. (You have already copied this file to your own space in Part C.) Study this program so you understand the basic functions for a doubly-linked list data structure. If you want to implement the list differently, please include the corresponding C/C++ code or pseudocode.

2. You are also given a partially implemented assembly program corresponding to the C program (`ll.s`). Again study this program to help you familiarize with the procedure implementation in MIPS. Feel free to use any part (or none) of the code. If you decide to use the whole sample code, **you will need to add '`insert`', '`delete`', and '`find`' procedures**, as well as any other procedures that you might need to complete those actions.

3. Complete the MIPS assembly code and test it with XSPIM.

4. It is a good practice to implement one procedure and test it thoroughly before moving on to another procedure.

E.3 Useful Tips

1. Because XSPIM does not have an operating system running, you need to do your own memory allocation. This can be achieved by making a global data object. Make SURE that this is the LAST global data member. (Why?)

   ```
   .data
   ADDR: .word ADDR
   .text
   .globl main # Required:: main
   main: lw $t0, addr
   addi $t0, $t0, 16 #word-align
   sw $t0, addr
   ...
   ```

   Use this variable as a pointer to the next available location to allocate space. When you need to allocate a new node, return this address, and increment the variable another X bytes, where X is the size of your node. In the sample C code, there is a procedure called '`newnode`' that malloc's the memory for a new 'node' structure before initializing it. Below is how this procedure was translated into MIPS assembly:

   ```
   .globl newnode
   newnode: # a0 = new data
   lw $t0, addr
   add  $t1, $t0, 16 # Word aligned
   sw  $t1, addr
   sw  $a0, 0($t0)
   sw  $0, 4($t0)
   sw  $0, 8($t0)
   add $v0, $t0, $0
   jr $ra
   .end
   ```

   Note that each new node was aligned on 16-byte boundaries. This is not necessary, but may be helpful in debugging.

2. XSPIM automatically allocates 1 MB of memory to the program in the DATA section of the Data Segments section (addresses 0x10000000 to 0x10020000). Therefore, we are slightly

limited in the size of the linked list we create, but 1MB will be enough for this project. Typically, dynamic memory allocation is not done in the Data section of the memory (part of the Basic Block Segment), but rather on the Heap; a section of memory between the Data and the Stack. XSPIM does not have support for a heap, so we have to make do with the data segment.

3. In case you are not familiar with the doubly-linked linked list, here is a crash course on it. You may also want to refer to your Data Structures textbook for additional information. In a *linked list*, each node contains a pointer, which contains the address of the next element. Links are convenient because, unlike an array structure, we do not have to declare the size of the memory we plan to use. A singly-linked list, however, may be inconvenient because we can only traverse the list in one direction.

4. For a *doubly-linked list*, a pointer also points to the preceding node. Now we can traverse the list in both directions. Disadvantages to this are the cost of an additional pointer and an increase in overhead when a node is removed (because now two pointers need to be updated instead of one). Practically, however, these costs are negligible and doubly-linked lists are far more widespread than single-linked lists.

   It is important to note that the list must be **LINKED**. This means that although your nodes may sit next to each other in memory, they are NOT to be accessed sequentially.

F. What to Turn In

You are not required to write a formal report for this lab. However, you do need to write a summary which addresses the questions raised throughout this lab assignment. Your summary should also discuss the following points:

- When XSPIM is started, before a program is loaded, some code (starting at 0x00400000) is already present. This code is executed by the simulator before starting any user's code. **Explain what this code is doing for your program.**
- Discuss your experience with XSPIM and MIPS assembly. Compare it to working with the six-instruction processor assembly and simulator. Are there features of either simulator that you miss in the other? How much "easier" or "harder" is MIPS compared to the six-instruction ISA?

In addition to the summary, you must place your MIPS assembly code developed for this lab in one of the group members dropbox. Please indicate the location of these files at the beginning of your report.