COE 308 – Computer Architecture
Term 082 – Spring 2009

Project 1: Writing, Simulating, and Testing MIPS Assembly Code

Objectives:

- Using the MARS MIPS simulator tool
- Writing, simulating, and testing MIPS assembly language code
- Solving linear matrices and estimating performance by counting instruction frequencies
- Doing floating-point arithmetic in software to understand it thoroughly
- Teamwork

Problem 1: Counting Words in a Text File and Finding their Frequency

Write and test a MIPS assembly language program to count the words in a text file and compute their frequency. The program should do the following:

- Open a text file and read all characters into an array. The maximum number of characters to be read should be limited to the size of the array, which should be 100,000 characters. MARS provides the system calls for opening a file, reading from a file, etc.
- Traverse the array character by character and detect the beginning and end of each word. A word is defined here to contain only letters (capital or lowercase). Other characters (spaces, commas, periods, parentheses, digits, etc.) should not be counted. Convert all letters to uppercase and convert all non-letter symbols to white space.
- Construct a second array to contain all unique words encountered in the first array and their frequencies. For example, if a word appears 100 times in the first array then it should appear once in the second array and its frequency should be 100.
- Sort the words in the second array according to their frequency and output the top *N* words that have the highest frequencies.

A sample run is shown below:

```
Enter input text filename: input.txt
How many words to output: 7

Top 7 words with highest frequencies

THE             151
A               120
THAT            69
YOU             56
FOR             42
HAS             37
ARRAY           21
```

Problem 2: Matrix Multiplication and Counting Instruction Frequencies

Write and test a MIPS assembly language program to perform matrix multiplication of *N* by *N* matrices of double-precision floating-point numbers. In your program, define the space of three 100×100 matrices, where the maximum value of *N* is fixed at 100. Initialize the first two matrices from two input text files and produce the result matrix in an output text file.

The matrix data should be read from a text file in row-major order. $N \times N$ signed integers should be read from a text file. Each integer should be read from a separate line. Prompt the user to enter the name of the text file, then open and read the text file line by line. Each line should be converted from an integer string to a floating-point number. You need a procedure to convert the integer string character-by-character to a floating-point number.

Write a procedure to do matrix multiplication of $N \times N$ matrices, where *N* is passed to the procedure as a parameter. *All matrix operations should be done using the double-precision floating-point instructions*. The output matrix should be written to a text file. Convert each double-precision floating-point number to an integer string. You will need a procedure to convert the floating-point number to an integer string. Write the integer string on a separate line in the output text file. Test and verify the matrix multiply procedure, by examining the result matrix in the output text file.

After succeeding in matrix multiplication and producing the correct result, you will analyze the MIPS code of the matrix multiply procedure, to have a better understanding of instruction frequencies. *You will count the dynamic number of instructions that are executed at runtime to determine their frequencies in the matrix multiply procedure*. You need a total of four counters to count instructions for the following classes of instructions:

- Class 1 is for ALU instructions
- Class 2 is for floating-point instructions
- Class 3 is for load and store instructions
- Class 4 is for branch and jump instructions

You will *augment the code of the matrix multiply procedure with additional instructions* to count the original number of instructions. You need four counters. At the beginning of the procedure, initialize all counters to zeros. Before each instruction, insert additional instructions to count that instruction. For example, if the original instruction is **addiu** then increment the counter of ALU instructions before the instruction itself. If the same instruction is executed 100 times (in different loop iterations), it will be counted as 100. Count only the real instructions. For pseudo-instructions, count the equivalent real instructions. Make sure that your additional code does not interfere with the original program code. Count only the original instructions of the matrix multiply procedure, not the new ones that you have added. Display the statistics that you have produced. A sample run is show below:

```
Enter the matrix size N: 10
Enter the first matrix filename: m1.txt
Enter the second matrix filename: m2.txt
Enter the result matrix filename: m3.txt

Counting Instructions in the Matrix Multiply Procedure:
Total               instructions = ???
ALU                 instructions = ??, Percentage = ?%
Floating-Point      instructions = ??, Percentage = ?%
Load & Store        instructions = ??, Percentage = ?%
Branch & Jump       instructions = ??, Percentage = ?%
```

Problem 3: Single-Precision Floating-Point Addition in Software

Write and test a MIPS assembly language program to do single-precision floating-point addition in software rather than in hardware. The procedure *floatadd* should receive its input parameters in **`$a0`** and **`$a1`** (as single-precision floating-point numbers) and produce its result in **`$v0`** (as single-precision float). **You cannot use the floating-point addition instruction `add.s` to do the addition**. Only integer instructions are allowed. Write additional procedures, if needed, to extract the fields, normalize, and round the result significand.

You should also make sure to handle special cases:

- Zero, infinity, and NaN
- Overflow and underflow
- Denormalized numbers

Round the result to the nearest even, which is the default rounding mode in IEEE 754 standard. This is the only rounding mode that should be supported.

Use the **add.s** instruction to check the result of the floating-point addition against the result produced by the *floatadd* procedure to ensure correctness.

Write a *main* procedure to call and test the *floatadd* procedure. Specifically, you should ask the user to input two floating-point numbers and to print the result.

A sample run should look as follows:

```
Enter 1st float: 1.25e-4
Enter 2nd float: 0.75e-3
Result of floatadd: 8.75e-4
Result of add.s:    8.75e-4
```

Tool

Use the MARS tool to write, execute, and test your code. To get started, familiarize yourself with the MARS MIPS simulator. You should familiarize yourself with the assembly language syntax and system calls. The MARS Help provides a description of all the system calls that are needed to complete this project. It also provides a list of all the basic and pseudo instructions.

Groups

Two or at most three students can form a group. Make sure to write the names of all the students involved in your group on the project report.

Coding and Documentation

Develop the code for the given problems with the following aspects in mind:

- Correctness: the code works properly
- Completeness: all cases have been covered
- Efficiency: the use of relevant instructions and algorithms
- Documentation: the code is well documented through the appropriate use of comments.

Report Document

The project report must contain sections highlighting the following:

- **Program Design**

  Specify clearly the design of each procedure giving detailed description of the algorithm used/developed and the implementation details.

- **Program Simulation**

  Describe all the simulator features that you have used for simulating your code with a clear emphasis on its advantages and limitations (if any), debugging for errors, the use of system calls and displaying the results of the program.

- **Program Output and Discussion**

  Provide snapshots of the Simulator window and show all the results.

  For program 1, provide input text files and show the top ten most frequently used words.

  For program 2, show only the statistics that you have produced for different runs for N = 10, 20, 50, and 100. Comment on these statistics, the complexity of the Matrix Multiply procedure, and the additional code that you inserted.

  For program 3, provide sample inputs and outputs and discuss all the cases that were handled by the *floatadd* procedure, such as normalized and denormalized numbers, zero, overflow, and underflow. Also test and demonstrate rounding.

- **Teamwork**

  Group members are required to divide the work equally among themselves, so that everyone is involved in algorithm design, program development, and debugging.

  Show clearly the division of work among the group members using a Chart and also prepare a Project execution plan showing the time frame for completing the subtasks of the project.

  Students who **helped** other team members should mention that to earn credit for that.

Submission Guidelines

All submissions will be done through WebCT. Submit one zip file containing the source code of programs 1, 2, and 3, and the report document. Also, submit a **hard copy** of the report in class.

Grading Policy

The grade will be divided according to the following components:

- Correctness of code: program produces correct results
- Completeness of code: all cases were handled properly
- Documentation of code: program is well documented
- Team Work : Participation and contribution to the project
- Report document

Late Policy

The project should be submitted on the due date by midnight. Late projects are accepted, but will be penalized 5% for each late day and for a maximum of 5 late days (or 25%). Projects submitted after 5 late days will not be accepted.