ICS 233 Project1 – Fall 2007
Computer Architecture and Assembly Language

Counting Words, Matrix Multiplication & Floating-Point Multiplication

Due Wednesday, November 28, 2007 by 12 Midnight

Objectives:

- Writing and Testing MIPS assembly language code
- Counting words in a text file and finding their frequency
- Doing floating-point arithmetic in software to understand it thoroughly
- Traversing matrices and estimating performance by counting instruction frequencies
- 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. The matrix data should be defined in the program itself under the data segment. Define six matrices (two 4×4, two 7×7, and two 10×10 matrices). Initialize them with arbitrary floating-point values. Write a procedure to do matrix multiplication of *n* × *n* matrices, where *n* is passed to the procedure as a parameter. Call the procedure three times to do the multiplication of the 4×4, 7×7, and 10×10 matrices. Define arrays to store the result matrices, and display the three output matrices. Test and verify your results.

After succeeding in matrix multiplication and producing the correct results, 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 five counters to count instructions for the following classes of instructions:

- Class 1 is for ALU instructions: **add, addu, addiu, or, ori, slt, sltu, sll, srl, mflo**, etc.
- Class 2 is for integer multiply instructions: **mult, multu**.
- Class 3 is for floating-point instructions: **add.d, mul.d**, etc.
- Class 4 is for load and store instructions: **lw, sw, ldc1**, **sdc1**, etc.
- Class 5 is for branch and jump instructions: **beq, bne, blez, bgtz, bltz, bgez, j, jr**, etc.

You will *augment the code of the matrix multiply procedure with additional instructions* to count the original number of instructions. 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 Class1 by inserting additional instructions to do the increment 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 matrix multiply, not the new ones that you have added.

At the end of the matrix multiply procedure, display the statistics that you have produced for each call of the procedure. A sample run is show below:

```
Multiplying 4x4 Matrices:
Total                instructions = ???
ALU                  instructions = ??, Percentage = ?%
Integer Multiply     instructions = ??, Percentage = ?%
Floating-Point       instructions = ??, Percentage = ?%
Load & Store         instructions = ??, Percentage = ?%
Branch & Jump        instructions = ??, Percentage = ?%

Output Matrix:
. . .

Multiplying 7x7 Matrices:
Total                instructions = ???
ALU                  instructions = ??, Percentage = ?%
```

```
Integer Multiply    instructions = ??, Percentage = ?%
Floating-Point      instructions = ??, Percentage = ?%
Load & Store        instructions = ??, Percentage = ?%
Branch & Jump       instructions = ??, Percentage = ?%

Output Matrix:
. . .

Multiplying 10x10 Matrices:
Total               instructions = ???
ALU                 instructions = ??, Percentage = ?%
Integer Multiply    instructions = ??, Percentage = ?%
Floating-Point      instructions = ??, Percentage = ?%
Load & Store        instructions = ??, Percentage = ?%
Branch & Jump       instructions = ??, Percentage = ?%

Output Matrix:
. . .
```

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

Write and test a MIPS assembly language program to do single-precision floating-point multiplication in software rather than in hardware. The procedure *floatmul* 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 multiply instruction `mul.s` to do the multiplication**. 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

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.

Write a *check* procedure to do the floating-point multiplication using the **mul.s** instruction. Compare the result of the *check* procedure against the result produced by the *floatmul* procedure to ensure correctness.

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

A sample run should look as follows. Allow the user to repeat the execution of the program.

```
Enter 1st float: 5.2e-12
Enter 2nd float: 7.3e+13
Result of floatmul: 379.6
Result of mul.s:    379.6
Repeat (Y/N)?
```

Tools:

Use MARS simulator to write and test your code.

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. Use a proper standard coding style.

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 showing all the results.

  Discuss all the cases that were handled. For program 1, provide two input text files and show the top twenty most frequent words.

  For program 2, show the statistics and the output of the matrix multiply procedure for different matrix sizes. Comment on these statistics, the complexity of the matrix multiply algorithm, and how the instruction counts change with the matrix size. Also comment on the additional code that you have inserted.

  For program 3, provide many sample inputs and outputs and discuss all the cases that were handled by the *floatmul* procedure, such as normalized 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, as well as the report document. Make sure that all programs are well documented.

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.