COE 308 – Computer Architecture
Term 091 – Fall 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: Solving Linear Equations

Gaussian elimination is a well-known technique for solving simultaneous linear systems of equations. Variables are eliminated one by one until there is only one left, and the discovered values of variables are back-substituted to obtain the values of other variables. In practice, the linear equations are represented as an augmented matrix *A*[*N*][*N*+1] with *N* rows and *N*+1 columns. The matrix is converted to an upper triangular matrix. Then back substitutions are used to produce the solution vector. Pseudo-code for *Gaussian* elimination is shown below.

```
procedure Gaussian (int N, float A[N][N+1]) {
   for k = 0 to n-2 do {
      for i = k+1 to N-1 do {
         factor = A[i][k]/A[k][k];
         for j = k+1 to N do {
            A[i][j] = A[i][j] – factor * A[k][j];
         }
         A[i][k] = 0
      }
   }
}
```

Pseudo-code for the *solve* procedure is shown below. This procedure is called after Gaussian elimination. It receives as input the upper triangular converted matrix *A*, and produces as output the solution vector *Sol*.

```
procedure Solve(int N, float A[N][N+1], float Sol[N]) {
  for (i=N-1; i>=0; i--) {
    Sol[i] = A[i][N];
    for (j=i+1; j<N; j++) {
      Sol[i] = Sol[i] - A[i][j] * Sol[j];
    }
    Sol[i] = Sol[i]/A[i][i];
  }
}
```

Write a MIPS assembly language program to perform Gaussian Elimination of floating-point matrices of size *N* by *N*+1, and to produce a solution vector of *N* floating-point elements. The matrix data should be read from a text file. *N* × (*N*+1) floats should be read from a text file. The numbers are separated by spaces or newline characters. Prompt the user to enter the name of the text file, then open and read the text file. Each number should be converted from a string format (the way it is read from a text file) to the single-precision binary format (the way it is stored in a floating-point register). Define a matrix *A* inside the program of size 100 × 101 floats and put a maximum limit on *N* to be 100. Read the numbers from the text file, convert them to single-precision float, and store them in matrix *A* in row-major order. Then, perform Gaussian elimination on matrix *A* and produce a solution vector. *All arithmetic operations should be done using the floating-point instructions.* Write the solution vector to an output text file, where each number is written on a separate line. A floating-point number should be converted to string format before writing it to a text file. MARS provides system calls for opening a file, reading from an input text file, and writing to an output text file. Test and verify your results.

A sample run is show below:

```
Enter Matrix Size (N): 10
Enter Matrix Input Filename: input.txt
Enter Solution Vector Output Filename: output.txt
```

The produced solution vector should be written to the output file: **`output.txt`**.

Problem 2: Program Analysis and Counting Instruction Frequencies

After succeeding in Gaussian elimination and producing a solution vector, you will analyze the MIPS code of the *Gaussian* and *Solve* procedures of Problem 1, 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.* 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 Gaussian procedure and the Solve 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, not the new ones that you have added.

At the end of the *Gaussian* procedure, display the statistics for this procedure. Do the same for the *Solve* procedure A sample run is show below:

```
Enter Matrix Size (N): 10
Enter Matrix Input Filename: input.txt
Enter Solution Vector Output Filename: output.txt

Statistics for the Gaussian Procedure:
Total               instructions = ???
ALU                 instructions = ??, Percentage = ?%
FPU                 instructions = ??, Percentage = ?%
Load & Store        instructions = ??, Percentage = ?%
Branch & Jump       instructions = ??, Percentage = ?%
```

```
Statistics for the Solve Procedure:
Total               instructions = ???
ALU                 instructions = ??, Percentage = ?%
FPU                 instructions = ??, Percentage = ?%
Load & Store        instructions = ??, Percentage = ?%
Branch & Jump       instructions = ??, Percentage = ?%
```

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

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

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 **div.s** instruction to check the result of the floating-point division against the result produced by the *floatdiv* procedure to ensure correctness.

Write a *main* procedure to call and test the *floatdiv* 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: 1.5e-4
Enter 2nd float: 0.75e-3
Result of floatdiv: 0.2
Result of div.s:    0.2
```

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.

  Discuss all the cases that were handled. For program 1, provide more than one input matrix text file and show the final solution vector for each run.

  For program 2, show only the statistics that you have produced for different runs for $N = 10, 25, 50$, and 100. Comment on these statistics, the complexity of the *Gaussian* and *Solve* procedures, 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 *floatdiv* 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, the report document, and test files. 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 for a maximum of 3 late days, but will be penalized. Projects submitted after 3 late days will not be accepted.