COE 308 – Computer Architecture
Term 052 – Spring 2006

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

Due Saturday, April 22, 2006 by Midnight

Objectives:

- Using a MIPS simulator such as MARS or SPIM
- Writing and Testing MIPS assembly language code
- Analyzing MIPS code and counting instruction frequencies
- Doing floating-point arithmetic in software to understand it thoroughly
- Teamwork

Problem 1: Generating Prime Numbers

Write and test a MIPS assembly language program to compute and print the first *n* prime numbers. A number *x* is prime if no number except 1 and *x* divides it evenly. A sample run should look as follows. Limit the input *n* to a maximum of 10,000 prime numbers.

```
How many Prime Numbers: 10
Generated Prime Numbers:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

Store the generated prime numbers in an array before displaying them. You should have the following procedures:

- *generate_prime*(*n*, *array*): Iterate over the integers starting from 2, testing if each is prime and generating *n* prime numbers. The prime numbers should be stored in *array*. The parameter *n* should be passed in $a0 and the *array* address should be passed in $a1.
- *test_prime*(*x, array*): Return 1 in $v0 if *x* is prime and 0 otherwise. To test if *x* is prime, it is sufficient to divide *x* by the prime numbers that were stored so far in *array*.
- *display_prime*(*n*, *array*): display the *n* prime numbers in *array*.

Problem 2: Program Analysis and Counting Instruction Frequencies

In this problem, you will analyze the MIPS code of Problem 1 to have a better understanding of instruction frequencies. You will count the number of instruction of various classes to determine their frequencies in the program that you have already developed for Problem 1. You need a total of 9 counters to count instructions for the following classes of instructions:

- Class 1 is for ALU R-type instructions: add, addu, sub, subu, and, or, xor, nor, slt, sltu, sll, srl, sra, sllv, srlv, srav, mfhi, mflo, mthi, mtlo.
- Class 2 is for ALU I-type instructions: addi, addiu, andi, ori, xori, lui, slti, sltiu.
- Class 3 is for multiply instructions: mult, multu.
- Class 4 is for divide instructions: div, divu.
- Class 5 is for load instructions: lb, lbu, lh, lhu, lw.
- Class 6 is for store instructions: sb, sh, sw.
- Class 7 is for branch instructions: beq, bne, blez, bgtz, bltz, bgez.
- Class 8 is for jump instructions: j, jal, jr, jalr.

- Class 9 is for system call instruction: syscall.

You will *augment the code of Program 1 with additional instructions* to count the original number of instructions. To do this, make a copy of Program1 and call it Program2. Before each instruction, insert additional instructions to count that instruction. For example, if the original instruction is *add* then increment the counter of Class 1 by inserting additional instructions to do the increment. Count only the real instructions. For pseudo-instructions, count the equivalent real instructions. Store the nine counters in an array. Make sure that your additional code does not interfere with the original program code. Count only the original instruction of Program 1, not the new ones that you have added in Program 2.

At the end of the program, display the statistics that you have produced. A sample run is show below:

```
How many Prime Numbers: 10
Total      instructions = ???
ALU R-type instructions = ??, Percentage = ?%
ALU I-type instructions = ??, Percentage = ?%
Multiply   instructions = ??, Percentage = ?%
Divide     instructions = ??, Percentage = ?%
Load       instructions = ??, Percentage = ?%
Store      instructions = ??, Percentage = ?%
Branch     instructions = ??, Percentage = ?%
Jump       instructions = ??, Percentage = ?%
Syscall    instructions = ??, Percentage = ?%

Generated Prime Numbers:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

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 *float_multiply* 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 and compute the result exponent and the result significand.

You should also make sure to handle special cases:

- Multiplication by 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.

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

A sample run should look as follows:

```
Enter 1st float in hexadecimal: 46d80000
Enter 2nd float in hexadecimal: bee00000
Result of multiplication in hexadecimal: c63d0000
```

Tools:

Use MARS or SPIM MIPS simulator to test your code. MARS is very similar but has a better GUI than SPIM. My recommendation is to try both before deciding on which one to use in this project.

To get started, familiarize yourself with the MARS or SPIM simulator. You should also familiarize yourself with the assembly language syntax and system calls for reading and printing integers and strings. Appendix A in Patterson and Hennessy book has a description of the SPIM simulator. A number of links have been provided for MARS and SPIM to help you get started.

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.

Submission Guidelines:

All submissions will be by email sent to:

mudawar@ccse.kfupm.edu.sa ; s217043@kfupm.edu.sa

Subject: COE 308 Project 1

Attach one zip file containing the source code of program 1, 2 and 3, as well as a report. Make sure that all 3 programs are well documented.

Report:

The report should contain the names of all the group members and the division of work among the group members (how the work was divided and who has done what). It should also contain sample runs for program 1, 2, and 3. For program 1, show in the report the output for 100 prime numbers produced by the program, and discuss the approach that you used for testing for prime. For program 2, show only the statistics that you have produced for different runs for 10, 50, 100, 500, and 1000 prime numbers (do not show the prime numbers). Comment on these statistics and the additional code that you inserted. For program 3, discuss all the cases that were handled by the *float_multiply* procedure, and provide many sample inputs and outputs making sure to test all cases, such as normalized numbers, denormalized numbers, zero, infinity, and NaN. Make sure also to test and demonstrate rounding.

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
- Participation and contribution to the project
- Report document
- Bonus: efficiency of code

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.