<u>CSE</u> <u>30321</u> – <u>Computer</u> <u>Architecture</u> <u>I</u> – <u>Fall</u> <u>2009</u>

Homework 05 – Efficient MIPS code and MIPS Procedure Calls

Assigned: September 29, 2009 **– Due:** October 8, 2009

This assignment should be done in groups of 3-4 people. (Please consult the guidelines given in the syllabus on performing group assignments.) Each team only needs to turn in one solution together with a cover sheet indicating the role of each team member. Your solutions must be typed.

This homework assignment will delve deeper into MIPS assembly, will stress the importance of generating efficient assembly code, and – now that we've covered the MIPS procedure call conventions in class – give you further practice with programs that invoke procedures.

<u>Problem</u> <u>1</u>: (35 points)

Consider the following fragment of C-Code:

```
for (j = 0; j < 200; j+=2) {
     p[j] = q[j+1] + 7;
}
```

Assume that *p* and *q* are arrays of 32-bit words that are stored in memory. The base address of *p* is in $s2 and the base address of *q* is in $s3.

<u>Part</u> <u>A</u> (10 points):
Write a sequence of MIPS instructions to execute the above high-level language code snippet. You must access memory by leveraging the loop index variable *j* and the starting addresses of *p* and *q* must be preserved so that they can be used again after the loop completes.

<u>Part</u> <u>B</u> (10 points):
Write another sequence of instructions that executes the above high-level language code snippet. Like before, you must preserve the starting addresses of *p* and *q* must be preserved so that they can be used again after the loop completes. However, this time, you are freed from the restriction of having to leverage the loop index when calculating memory addresses. How might you make this code more efficient?

<u>Part</u> <u>C</u> (5 points):
Assembly instructions are the interface between a high-level language (like C, C++, Java, etc.) and the hardware that actually executes what the programmer intends. Efficient translation of code written in a high-level language to assembly instructions is important in order to obtain the best possible performance. If we assume (a) the CPIs given in the table below and (b) that the machine the code that you wrote for Part A and Part B has a clock rate of 1 GHz, what is the performance improvement when you compare Part A to Part B?

Instruction Type	CPI
Jump / Branch	3 (T or NT)
Add, Addi, sll, multiply, sub, etc.	4
Load	5
Store	4

<u>Part</u> <u>D</u> (10 points):
Relatively speaking, the difference in execution time between the two versions of code that you wrote for Part A and Part B is minor and would be virtually unnoticeable. What if this loop is run 10^{12} times (not implausible for software projects that do things like climate modeling, code breaking, etc.)? What does your answer suggest?

Problem 2: (25 points)

The Fibonacci numbers are a sequence of numbers where the first two Fibonacci numbers are 0 and 1, and each remaining number is the sum of the previous two. (e.g. 0, 1, 2, 2, 3, 4, 5, 13, …). The **Fibonacci Number F_n** can be defined by: $F_n = F_{n-1} + F_{n-2}$.

Below, is a sequence of code that might be used to computer a Fibonacci number. However, there are a few "errors" in this code (e.g. where the MIPS conventions are not followed) and you will be asked to correct these errors.

```
FIB:   addi   $sp, $sp, -12
       sw     $ra, 0($sp)
       sw     $s1, 4($sp)
       sw     $a0, 8(4sp)
       slti   $t0, $a0, 1
       beq    $t0, $0, L1
       addi   $v0, $a0, L1
       j      EXIT

L1:    addi   $a0, $a0, -1
       jal    FIB
       addi   $s1, $v0, $0
       addi   $a0, $a0, -1
       jal    FIB
       add    $v0, $v0, $s1

EXIT:  lw     $ra, 0($sp)
       lw     $a0, 8($sp)
       lw     $s1, 4($sp)
       addi   $sp, $sp, 12
       jr     $ra
```

Part A (5 points):

The MIPS assembly program above computes the Fibonacci number of a given input N. The integer input is passed through register $a0, and the result is returned in register $v0. In the assembly code, there are a few errors. Correct them.

Part B (15 points):

For the recursive Fibonacci MIPS program above, assume that the input is 4. Rewrite the Fibonacci program to operate to operate in a non-recursive manner. Restrict your register usage to registers $s0-$s7. What is the total number of instructions used to execute the non-recursive version compared to the recursive version that you corrected for Part A (assuming that N is again 4). What conclusions can be drawn?

Part C (5 points):

Show the contents of the stack after each function call, assuming that the input is 4.

<u>Problem</u> <u>3</u>: (25 points)
Assume that the stack and static data segments are empty and that the stack pointer starts at address 0x7FFFFFFC. Assume the MIPS calling conventions and that function inputs are passed using register $a0-$a3 and returned in $v0-$v1. Assume that leaf functions may only use saved registers. The questions below will refer to the following C-code:

```
main()
{
      leaf_function(1);
}
int leaf_function (int f)
{
      int result;
      result = f + 1;
      if (f > 5)
            return result;
      leaf_function(result);
}
```

<u>Part</u> <u>A</u> (5 points):
Show the contents of the stack after each function call.

<u>Part</u> <u>B</u> (15 points):
Write MIPS code for the code given above. *Note – for any partial credit, each instruction must have a comment explaining what it does in the context of the program.*

<u>Problem</u> <u>4</u>: (15 points)
Consider the following MIPS assembly code:

```
f:      sub     $s0, $a0, $a3
        sll     $v0, $s0, 0x1
        add     $v0, $a2, $v0
        sub     $v0, $v0, $a1
        jr      $ra
```

<u>Part</u> <u>A</u> (5 points):
This code contains a mistake that violates the MIPS calling convention. What is this mistake and how should it be fixed?

<u>Part</u> <u>B</u> (5 points)
What is the C equivalent of this code? Assume that the function's arguments are named `a`, `b`, `c`, etc. in the C version of the function.

<u>Part</u> <u>C</u> (5 points):
As the point where this function is called, $a0, $a1, $a2, and $a3 have values 1, 100, 1000, and 30 respectively. What is the value returned by this function? If another function g is called from f, assume that the value returned from g is always 500.