Problem Set 4  
Computer Science 240  
Fall 2014  
Due: Friday, October 3

Relevant Reading. Patterson and Hennessy §2.9 – §2.10

Introducing Challenge Problems. This problem set introduces so-called challenge problems. Challenge problems are optional, but they give you an opportunity to explore further if you are curious. See the end of the problem set.

Problem 1. (Tail recursion) Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with procedure calls. For example, consider a procedure used to accumulate a sum. The variable acc is an accumulator.

    int sum (int n, int acc) {
        if (n > 0)
            return sum(n - 1, acc + n);
        else
            return acc;
    }

Consider the procedure call sum(3,0). This will result in recursive calls to sum(2,3), sum(1,5), and sum(0,6), and then the result 6 will be returned four times. This recursive call of sum is referred to as a tail call, because the result from each level of recursion is not used by any further computations at any recursive levels – it is returned by each recursive call in turn. Tail recursion can be implemented very efficiently. We can implement the sum function above with the following MIPS code, assuming $a0 = n$ and $a1 = acc$:

    sum:  beq $a0, $zero, sum_exit # go to sum_exit if n is 0
           add $a1, $a1, $a0      # add n to acc
           addi $a0, $a0, -1     # subtract 1 from n
           j  sum                # go to sum

    sum_exit:       move $v0, $a1    # return value acc
                  jr  $ra          # return to caller

Notice that the compiled version of sum isn't really recursion at all. The recursion is translated as to iteration. This is true for all reasonably smart modern compilers: Tail recursive programs are compiled as iterations. In particular the assembly code for such subroutines do not include any jal instructions.

Write a program as in Problem 5 on Problem Set 3, except this time base your program on the following procedure and optimize the tail call to make your implementation efficient:
int fib_iter (int a, int b, int count) {
    if (count == 0)
        return b;
    else
        return fib_iter(a + b, a, count - 1);
}

Here, the first two parameters keep track of the previous two Fibonacci numbers computed. To compute $F(n)$, you have to make the procedure call fib_iter(1, 0, n) from your main program.

All the old rules about programming style are still in effect. The good news here is that you can recycle all of the introductory code from Problem 5 on Problem Set 3. You only need to rewrite the fib function.

At the beginning of your program include a comment stating whether or not your program works. Turn in a copy of your program and, if your program works, a printout of your program’s output for an input of 6.

Problem 2. Estimate the difference in performance between your solution to Exercise 3.6 and your solution to this exercise. You need not calculate the asymptotic computational complexity of each algorithm, a rough estimate will do.

Problem 3. Using the MIPS program below (familiar from Problem 4 on Problem Set 3), determine the instruction format for each instruction and the decimal values of each instruction field. See the MIPS card inside the cover of your textbook for help translating. Do not just paste the program in MARS to inspect it.

Problem 4. Show the single MIPS instruction or minimal sequence of instructions (not pseudoinstructions) for the following C statement:


Assume that $a$ corresponds to register $t3$ and the array $x$ has a base address of $6,400,000_{ten}$. You may not assume that this value is already in a register. Rather you must choose a register $reg$ write assemble code to move $6,400,000_{ten}$ into register $reg$. Note that $6,400,000_{ten}$ will not fit into sixteen bits.
**Problem 5.** Given your understanding of the addressing modes used by branch instructions:

a. Explain the conditions under which an assembler could not implement the branch instruction in the following code sequence *directly*:

```
here:   beq  $s0, $s2, there
...  ← possibly many instructions
there:  add  $s0, $s0, $s0
```

b. Describe a simple translation the assembler could perform to mitigate this problem.

– End of required problems –

**Challenge Problem 1.** Problem 2 asks you to estimate running time for your optimized implementation of a tail-recursive function. For this challenge problem, explain how tail-call optimization affects call-stack space utilization in the general case (not for fib_iter specifically) vs. an unoptimized implementation. What is the key insight about tail-recursive functions’ use of stack frames that affects their space utilization when optimized?