CPSC 313
06W Term 2
Problem Set #3

Due: Sunday, February 4, 2007 at 11:59 PM (thirteen-hour grace period)

1. Write in C a function `int gcd(int x, int y);` that returns the greatest common divisor of its two arguments. You may assume that they are both positive. Your function should compute the gcd using subtraction only. The algorithm is:

   - if x == y, gcd(x, y) = x
   - if x > y gcd(x, y) = gcd(x - y, y)
   - if x < y gcd(x, y) = gcd(x, y - x)

   Write four versions of the algorithm:

   - a recursive version
   - a version that uses a `for` loop
   - a version that uses a `while` loop
   - a version that uses a `do ... while` loop

   Compile each version with gcc -S -O2.

   (a) Turn in (on paper) all 4 versions of your .c files and their corresponding, gcc generated, .s files.

   (b) Describe the differences in the generated code of the four versions. Don't worry about trivial syntactic differences (like one version chooses register `%ebx` to hold some value while another version chooses a different register. You should look for differences that matter (semantic differences) such as:

      i. Differences in the number of registers used inside the loop
      ii. Differences in the number of branch instructions inside the loop

      Also note versions that are identical except for trivial syntactic differences.

2. Write an IA32 assembly-language procedure that uses a switch statement that is implemented by a jump table to determine whether an input number is a member of the Fibbonacci sequence (1, 1, 2, 3, 5, 8, 13, ...) and less than 100.

   int isFib (int n);

   Should return 1 if n is a Fibbonacci number and n < 100; it should return 0 otherwise. Handle all possible values of n. Comment your code. Your solution should be in a file named isFib.s.

   Write a test program in C in the file isFibMain.c, link it with your isFib.o to get an executable and run it.

   Hand this one in using `handin` (isFib.s and isFibMain.c).

3. The program below computes $base^{*np}$. Focusing on the procedure power(), answer the following two questions. Note: This procedure does strange things with its arguments in order to get the assembly language code to be interesting for this question. There is no other reason for the first argument to be passed by reference or for the variable n to exist. And, it probably would have made more sense to reverse the order of the two arguments. But there is a method to my madness ...

```
int power(int* np, int base)
{
    int n;
    if (*np == 0)
        n = 1;
    else {
        n = *np-1;
        n = base * power(&n,base);
    }
    return n;
}
```

   (a) The assembly language file produced by a certain compiler is at

   http://www.ugrad.cs.ubc.ca/~cs313/2006W2/code/power.s

   Comment every line of the procedure power(). You do not need to comment main(). Your comments should relate the assembly code back to the C-language source, as much as possible.

   (b) Now modify the assembly code for the power() procedure so that it uses only one register (specifically, `%esp`) to point to the runtime stack. You will need to eliminate all use of `%ebp` from power(). You may not use another register (other than `%esp`) to point to the stack.

   Test your program by compiling the modified .s file to create an executable (gcc -o power power.s will do) and executing it. The point of this exercise to better understand the runtime stack and the benefit of using two registers to point to it (i.e., `%esp` and the one you are removing `%ebp`).

   Hand this one in on paper.

4. This is all about caller-save and callee-save registers.

   (a) Consider a function A, which calls a function B 10 times in a loop. Function B doesn't call any other functions or use any of the callee-save registers.

   Now, suppose that A needs to maintain some value, `x`, in a register across the call to B. With each of the following assumptions, compute how many times the register selected to hold `x` will be saved and restored to/from the stack. Explain each answer.

      i. If A puts `x` in a caller-save register.

      ii. If A puts `x` in a callee-save register.

(b) Consider a function C, which calls a function D 10 times in a loop. Function D further calls a function E 20 times in a loop. Function E doesn't call any other functions or use any of the callee-save registers.

Now, suppose that both C and D each need to maintain some value, `x` and `y` respectively, in a register across the call to D and E respectively. With each of the following assumptions, compute how many times the register selected for `x` and `y` will be saved and restored to/from the stack. Explain each answer.

i. If C puts `x` in a caller-save register and D puts `y` in (the same) caller-save register.

ii. If C puts `x` in a caller-save register and D puts `y` in a callee-save register.

iii. If C puts `x` in a callee-save register and D puts `y` in a caller-save register.

iv. If C puts `x` in a callee-save register and D puts `y` in (the same) callee-save register.

Hand this one in on paper.