Assignment #6

(max = 95)

Read the rest of chapter 3 (starting at page 196) in the *Computer Organization and Design* text, including section 3.11, which is under Course Materials as **CD3.11.pdf**. This would also be an appropriate time to go through Appendix B (we have been referring to various sections of this Appendix in our last few assignments). I have provided an extensive set of notes ("Notes for Assignment #6) on this reading that can be found under Course Notes. Please refer to these notes as you carefully work through the assigned reading.

Afterwards, submit answers for the following problems (for questions 1-5, it is imperative that you show your work):

1. In a Von Neumann architecture, groups of bits have no intrinsic meanings by themselves. What a bit pattern represents depends entirely on how it is used. As an example, let us look at 0x0C000000. (8 points)

   a) As a two's complement integer, what decimal value does this represent?
   b) As an unsigned integer, what decimal value does this represent?
   c) Interpreted as an instruction, exactly what instruction is this?
   d) As a single-precision floating point number, what decimal value does this represent (express as a decimal number … with one digit to the left of the decimal place … times 2 to some decimal power).

2. Repeat question 1 using the value 0xC4630000. For part d) round to six significant digits. (12 points)

3. Do Exercise 3.23 on page 239 in the text (give result in binary and in hexadecimal). (5 points)

4. Do Exercise 3.24 on page 239 in the text (give result in binary and in hexadecimal). (4 points)

5. Given the following denormalized single precision floating point number:

   $$800C\ 0000_{16}.$$

   What is the value of this floating point number (express answer as a decimal number … with one digit to the left of the decimal place … times 10 to some power; round to eight significant digits)? (7 points)

6. I suspect that all of you are familiar with the transcendental number, e. Many applications in mathematics involve computing various powers of e. It can be proven that

   $$e^x = 1 + x/1 + x^2/2! + x^3/3! + \ldots$$

for all values of x. Of course, since this is an infinite sum, so we can't hope to actually sum all of these values up! But the good news is that the later terms get so small that a partial sum can provide a very nice approximation for the value of e^x. You are to write a double precision function (result returned in $f0) called exp with one double precision parameter (in $f12), along with a little driver program for testing your function. Your function should use the summation formula as an approximation for the value of e^x, using the following guide for terminating the summation:

> If the next term divided by the summation so far is less that 1.0e-15, then terminate the summation (and don't even bother to add in that next term). [One might be tempted to just stop if the next term is less that 1.0e-15, but my proposed guide is more sensitive to the relative size of the actual summation.]

Even though the summation is valid for all values of x, there is a problem with convergence when you use negative values "bigger" than -20. Therefore, your exp function should compute the value of $e^{|x|}$ instead, and then invert the result (this process should be handled by the function exp, not by your driver program). You can expect your program to have overflow problems when tested with values of x somewhere around 708 (or -708).

Here is a sample execution of my code:

```
Let's test our exponential function!
Enter a value for x (or 999 to exit): 1
Our approximation for e^1 is 2.7182818284590455
Enter a value for x (or 999 to exit): 0
Our approximation for e^0 is 1
Enter a value for x (or 999 to exit): 3.75
Our approximation for e^3.75 is 42.521082000062762
Enter a value for x (or 999 to exit): -1
Our approximation for e^-1 is 0.36787944117144228
Enter a value for x (or 999 to exit): 700
Our approximation for e^700 is 1.0142320547349994e+304
Enter a value for x (or 999 to exit): -700
Our approximation for e^-700 is 9.8596765437598214e-305
Enter a value for x (or 999 to exit): 1.0e-10
Our approximation for e^1e-010 is 1.0000000001
Enter a value for x (or 999 to exit): -1.0e-10
Our approximation for e^-1e-010 is 0.99999999989999999
Enter a value for x (or 999 to exit): 999
Come back soon!
```

Don't forget to document your code! Submit a separate file called **exp.s** as well as placing your code in this assignment submission; the Mentor will clarify what I mean by this. (45 points)

7. You should recall writing a little factorial function in Assignment #2. In Assignment #5 we examined why we were so limited in the values of n that could be used when testing that factorial function. The limitation for integers was, of course, the 32 bits (or 31, if signed) that are available for representing those integers. What if, instead, we computed factorials using double precision floating point numbers? There are two obvious advantages: 1) since the fraction portion of double precision numbers is 53 bits long, we can maintain more significant

digits; and 2) since floating point numbers maintain an exponent, we can calculate much larger factorials (but the answers will eventually be not exact).

Here is a function called **dpfact** (for double precision factorial) that I wrote to explore this idea. [SPECIAL NOTE: You should **not** use this function when writing the program for problem 6; you will want to avoid computing x^n and n! as separate values since both go to infinity.] The function computes the factorial iteratively (rather than recursively). The function expects an integer parameter (n) in register $a0, and returns the factorial of that number as a double precision value (in register $f0).

```
dpfact: li      $t0, 1              # initialize product to 1.0
        mtc1    $t0, $f0            # move integer to $f0
        cvt.d.w $f0, $f0            # convert it to a double

again:  slti    $t0, $a0, 2         # test for n < 2
        bne     $t0, $zero, done    # if n < 2, return

        mtc1    $a0, $f2            # move n to floating register
        cvt.d.w $f2, $f2            # and convert to double precision

        mul.d   $f0, $f0, $f2       # multiply product by n

        addi    $a0, $a0, -1        # decrease n
        j       again               # and loop

done:   jr      $ra                 # return to calling routine
```

Here is a short demonstration of my program's execution:

```
Welcome to the double precision factorial tester!
Enter a value for n (or a negative value to exit): 1
1! is 1
Enter a value for n (or a negative value to exit): 16
16! is 20922789888000
Enter a value for n (or a negative value to exit): 50
50! is 3.0414093201713376e+064
Enter a value for n (or a negative value to exit): 500
500! is 1.#INF
Enter a value for n (or a negative value to exit): -1
Come back soon!
```

To complete this exercise, you will need to add a little driver program to my code (call your file **dpfact.s** … to start you out, I have put my code for the function **dpfact** in a file by that name under Course Materials) and answer the following questions. You should be able to use the driver program from your Assignment #2 submission with very minor revisions. Submit a separate file called **dpfact.s** as well as placing your code in this assignment submission; the Mentor will clarify what I mean by this. (14 points)

Here are the questions:

1) What is the largest value of n for which my function produces an exact answer [you may need to use a calculator (like the one on your PC that handles lots of digits) to verify this.]?
2) Notice that rather than throw an exception when the value of n gets too large, my code simply produces an infinite result. What is the smallest value of n for which my function produces an infinite result?

Your assignment is due by 11:59 PM (Eastern Time) on the assignment due date (consult Course Calendar on course website).