Jon Crawford

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)
   Binary = 0000 1100 0000 0000 0000 0000 0000 0000 pos
   $(0 \times 16^7) + (12 \times 16^6) + (0 \times 16^5) + (0 \times 16^4) + (0 \times 16^3) + (0 \times 16^2) + (0 \times 16^1) + (0 \times 16^0) = 201{,}326{,}592$
   a) As a two's complement integer, what decimal value does this represent?
      201,326,592
   b) As an unsigned integer, what decimal value does this represent?
      201,326,592
   c) Interpreted as an instruction, exactly what instruction is this?
      Opcode – Address
      000011 00 0000 0000 0000 0000 0000 0000
      jal 0
   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).
      $-1^{sign} * 1 + fraction * 2^{exponent - bias}$
      Sign - Exponent – Fraction
      0 000 1100 0 000 0000 0000 0000 0000 0000
      $-1^0 * 1 + 0 * 2^{24 - 127}$
      1.0×2^{-103}
2. Repeat question 1 using the value 0xC4630000. For part d) round to six significant digits. (12 points)
   Binary = 1100 0100 0110 0011 0000 0000 0000 0000 neg
   $(12 \times 16^7) + (4 \times 16^6) + (6 \times 16^5) + (3 \times 16^4) + (0 \times 16^3) + (0 \times 16^2) + (0 \times 16^1) + (0 \times 16^0) = 3{,}294{,}822{,}400$

a.) $3,294,822,400 - 2^{32} = -1,000,144,896$
b.) 3,294,822,400
c.) Opcode - fmt - ft - offset
110001 00011 00011 0000 0000 0000 0000
lwc1 $v1 $v1 0
d.) $-1^{sign} * 1 + \text{fraction} * 2^{\text{exponent - bias}}$
Sign - Exponent – Fraction
1 100 0100 0 110 0011 0000 0000 0000 0000
Fraction = $(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0) + (0) + (0) + (1 \times 2^{-6})$
$+ (1 \times 2^{-7}) = 99/128$
Exponent = $(1 \times 2^{7}) + (0 \times 2^{6}) + (0 \times 2^{5}) + (0 \times 2^{4}) + (1 \times 2^{3}) + (0 \times 2^{2})$
$+ (0 \times 2^{1}) + (0 \times 2^{0}) = 136$
$-1^{1} * 1 + 99/128 * 2^{136 - 127}$
$-227/128 * 2^{9} = -1.77344*2^{9}$

3. Do Exercise 3.23 on page 239 in the text (give result in binary <u>and</u> in hexadecimal). (5 points)
63/2 = 31/2 = 15/2 = 7/2 = 3/2 = ½ = 11 1111
$.25 = 01/2^{2} = .01$
$11\ 1111.01 * 2^{0} = 1.1111101 * 2^{5}$ 5+127 = 132
132/2 = 66/2 = 33/2 =16/2 = 8/2 = 4/2 = 2/2 = ½ = 00100001 = 10000100
Bin = 0100 0010 0111 1101 0000 0000 0000 0000
Hex = 4 2 7 D 0 0 0 0
4. Do Exercise 3.24 on page 239 in the text (give result in binary <u>and</u> in hexadecimal). (4 points)
5 + 1023 = 1028
1028/2 = 514/2 = 257/2 = 128/2 = 64/2 = 32/2 = 16/2 = 8/2 =4/2 =2/2= ½
00100000001 = 100 0000 0100
Bin =0100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Hex = 4 0 4 F A 0 0 0 0 0 0 0 0 0 0 0
5. Given the following denormalized single precision floating point number:

$800C\ 0000_{16}$.

Bin = 1000 0000 0000 1100 0000 0000 0000 0000

1 000 0000 0 000 1100 0000 0000 0000 0000

Sign exponent fraction

$= -0.0011 * 2^{-127} = -.00011 * 2^{-126}$

$= -(2^{-130} + 2^{-131}) =$ -1.10202595389589453872069050365e-39

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) Approximately: -1.1020260e-39

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

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

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 than 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 than 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, <u>not</u> 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)

```
# Jon Crawford -- 4/23/18
# exp.s - A simple program to approximate e in double precision
#
#
# Register use:
#       $a0     parameter for syscall
#       $v0     syscall parameter
#       $f0     float syscall return value
#       $f12    float syscall parameter
#       $f14-16 constant values
#       $f4-f22 all calculation variables are declared inline

exp:    l.d     $f14, conE              # double E
        l.d     $f16, con0              # double 0.0
        l.d     $f10, con1      # double 1.0
        l.d     $f6,  con0              # double sum = 0
        l.d     $f4,  con1              # double n = 1

        mov.d   $f18, $f12              # copy of invalue
        mov.d   $f20, $f18

        abs.d   $f20, $f20              # abs(x);
        div.d   $f22, $f20, $f4         # z = x/n;
        add.d   $f6, $f10, $f22         # sum = 1 + z;


        calc:   add.d   $f4, $f4, $f10          # ++n;
                mul.d   $f22, $f20, $f22        # x*=z;
                div.d   $f22, $f22, $f4         # z = z/n;

                c.lt.d  $f22, $f14              # if (val < E)
                bc1t    done                    # break;

                add.d   $f6, $f6, $f22          # sum += val;

                j       calc                    # loop for next sum

done:   c.lt.d  $f18, $f16              # if (!copy of x < 0) skip;
        bc1f    skip

        div.d   $f6, $f10, $f6          # else, x = 1/x;
```

```
skip:   mov.d   $f0, $f6
        jr      $ra             # return to caller


main:   la      $a0, intro      # print intro msg
        li      $v0, 4
        syscall

loop:   la      $a0, inval      # prompt user
        li      $v0, 4
        syscall

        li      $v0, 7          # read value
        syscall

        l.d     $f14, nines
        c.eq.d  $f0, $f14
        bc1t    exit            # branch on nines

        mov.d   $f8, $f0        # save inval to print
        mov.d   $f12, $f0

        jal exp                 # call e^x function

        mov.d   $f2, $f0        # save e^x return value

        la      $a0, emsg       # print e^ msg
        li      $v0, 4
        syscall

        mov.d   $f12, $f8
        li      $v0, 3          # print invalue
        syscall

        la      $a0, ismsg      # print is
        li      $v0, 4
        syscall

        mov.d   $f12, $f2       # print e^x value
        li      $v0, 3
```

```
        syscall

        j       loop            # loop

exit:   la      $a0, end        # print end msg
        li      $v0, 4
        syscall

        li      $v0, 10                 # exit program
        syscall


        .data
intro:  .asciiz  "Let's test our exponential function!"
inval:  .asciiz  "\nEnter a value for x (or 999 to exit): "
emsg:   .asciiz  "Our approximation for e^"
ismsg:  .asciiz  " is "
end:    .asciiz  "Come back soon!\n"
nines:  .double  999.0
con1:   .double  1.0
con0:   .double  0.0
conE:   .double  0.000000000000001
```

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 <u>exact</u>).

   Here is a function called **dpfact** (for double precision factorial) that I wrote to explore this idea. [SPECIAL NOTE: You should **<u>not</u>** 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 <u>very</u> 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 <u>exact</u> answer [you may need to use a calculator (like the one on your PC that handles lots of digits) to verify this.]?

   21!

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?

   171

```
# Jon Crawford -- 4/21/18
# dpfact.s - A simple addition to our factorial program to
#         return a double precision factorial value.

# Register use:
#    $a0      parameter for syscall
#    $v0      syscall parameter
#    $f12     float return value
#    $f0      float syscall parameter
#    $f2      used for n
#    $t0      temporary use for calculation
#    $t1      index count

dpfact:     li       $t0, 1                # initialize product to 1.0
            mtc1     $t0, $f0
            cvt.d.w$f0, $f0

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


main:       la       $a0, intro          # Print intro msg
            li       $v0, 4
            syscall

loop:       la       $a0, inval          # Prompt user for value
            li       $v0, 4
            syscall

            li       $v0, 5              # Read in value from user
            syscall
            move     $a0, $v0            # Move invalue into $a0
```

```
            slti    $t1, $a0, 0             # Check for Sentinel & quit
            bne     $t1, $zero, quit

            li      $v0, 1                  # Print value before answer msg
            syscall

            jal dpfact                      # Function call


            la      $a0, answer             # Print answer msg
            li      $v0, 4
            syscall

            mov.d   $f12, $f0               # Print factorial value
            li      $v0, 3
            syscall

            j loop                          # Loop to prompt

quit:       la      $a0, end                # Print goodbye msg
            li      $v0, 4
            syscall

            li      $v0, 10                         # Exit program
            syscall

    .data
intro:      .asciiz "Welcome to the factorial tester!"
inval:      .asciiz "\nEnter a value for n (or a negative value to exit): "
answer:     .asciiz "! is "
end:        .asciiz "Come back soon!"
```

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