Assignment #6 Key

(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?

      In binary, the top bit is zero, so the number is positive. Working straight from base 16, we get a value of $12*16^6$ = 201326592.

   b) As an unsigned integer, what decimal value does this represent?

      Except for cases where the top bit is a 1, the unsigned value is the same as the two's complement value. So, the unsigned value is 201326592 as well.

   c) Interpreted as an instruction, exactly what instruction is this?

      The first six bits are 000011, so the opcode is 3, which is the J-format instruction, jal. The instruction in binary (in 6-26 format) is:
      0000111 00000000000000000000000000.
      This gives us: 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)?

      In binary (in 1-8-23 format): 0 00011000 00000000000000000000000.
      Since the sign is 0, the number is positive. The exponent is $2^4+2^3=16+8=24$.
      The fraction is 0. Putting it together: $(1+0)*2^{24-127}=1.0*2^{-103}$.

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

a) In binary, the top bit is one, so the number is negative.
Binary: 1100 0100 0110 0011 0000 0000 0000 0000.
Two's complement is 0011 1011 1001 1100 1111 1111 1111 1111 + 1, which gives us 0011 1011 1001 1101 0000 0000 0000 0000. In hex, this is 0x3B9D0000. Converting this hex number to decimal gives:
$3*16^7+11*16^6+9*16^5+13*16^4$
$=805306368+184549376+9437184+851968=1000144896$.
So, the two's complement number is <u>-1000144896</u>.

b) As an unsigned integer:
$12*16^7+4*16^6+6*16^5+3*16^4$
$=3221225472+67108864+6291456+196608=$<u>3294822400</u>.

c) The first six bits are 110001, so the opcode is 31 (hex), which is an I-format instruction, lwc1. The instruction is (in 6-5-5-16 format):
110001 00011 00011 0000000000000000.
This gives us: rs = 3, rt = 3, immediate = 0.
So, the instruction is: <u>lwc1 $3, 0($3)</u> or <u>lwc1 $f3, 0($v1)</u>.

d) In binary (1-8-23 format): 1 10001000 11000110000000000000000. Since the sign is 1, the number is negative. The exponent is $2^7+2^3= 128+8=136$.
The fraction is $2^{-1}+2^{-2}+2^{-6}+2^{-7}=0.5+0.25+0.015625+0.0078125=0.7734375$.
Putting it together, we get: $-(1+0.7734375)*2^{136-127} = \underline{-1.77344*2^9}$.

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

Converting 63.25 to binary, we get: 111111.01
Shifting "binary point" 5 places to the left, we get: $1.1111101*2^5$. The sign is 0 (positive number), the 1 to the left of the binary point is "hidden" and the exponent must have the bias of 127 added ($5+127= 132= 10000100_2$). We put it together: $\underline{0\ 10000100\ 11111010000000000000000_2} = \underline{427D\ 0000_{16}}$.

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

The number is still $1.1111101*2^5$. When using double precision, the bias is 1023, so we get an exponent of $5+1023=1028= 10000000100_2$. Putting all of this together gives us:
$\underline{0\ 10000000100\ 1111101000000000000000000000000000000000000000000000_2}$.
In hex, this is $\underline{404F\ A000\ 0000\ 0000_{16}}$.

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)

Convert the number to binary: 1000 0000 0000 1100 0000 0000 0000 0000
Group into sign, exponent and fraction: 1 0000000 00011000000000000000000.
Remembering that denormalized numbers (non-zero fractions with an biased exponent of zero) do not have an implied "1." in front, this gives us a value of $(-1)^1 \times 2^{-126} \times (2^{-4} + 2^{-5}) = -(2^{-130} + 2^{-131})$. Since 2^{-130} is about $7.3468397 \times 10^{-40}$ and 2^{-131} is about $3.6734198 \times 10^{-40}$, we add and get the decimal equivalent of (remembering that final answer is negative): $-1.1020260 \times 10^{-39}$.

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 9.85967654375982l4e-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)

```
# Stephen P. Leach -- 11/12/17
# exp.asm - a simple driver program that tests the exp function
#       using doubles.  The value of e^x is approximated with
#       the sum 1 + x + (x^2)/(2!) + (x^3)/(3!) + ...
#       If x is negative, compute e^|x| and invert.
# Register use:
#       $a0     syscall parameter
#       $v0     syscall parameter
#       $f0     exp and syscall return value
#       $f12    exp and syscall parameter
#       $f16    1.0
#       $f18    1.0E-15
#       $f20    previous/next term
#       $f22    n
#       $f24    temporary calculations
#       $f26    |x|

exp:    l.d     $f16, one               # $f16 will have 1.0
        l.d     $f18, tiny              # $f20 will have 1.0E-10
        mov.d   $f0, $f16               # initialize sum to 1.0
        mov.d   $f20, $f16              # initialize term to 1.0
        sub.d   $f22, $f16, $f16        # initialize n to 0.0
        abs.d   $f26, $f12              # $f26 will have |x|

next:   add.d   $f22, $f22, $f16        # increment n
        mul.d   $f20, $f20, $f26        # compute next term
        div.d   $f20, $f20, $f22

        div.d   $f24, $f20, $f0         # compute next term / sum
        c.lt.d  $f24, $f18              # see if time to quit
        bc1t    quit

        add.d   $f0, $f0, $f20          # otherwise, add next
        j       next                    # term to sum and loop

quit:   c.eq.d  $f26, $f12              # if x positive, leave
        bc1t    exit
        div.d   $f0, $f16, $f0          # otherwise, invert

exit:   jr      $ra                     # return to calling routine

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

loop:   la      $a0, req        # request a value for x
        li      $v0, 4
        syscall

        li      $v0, 7          # read the value
        syscall

        l.d     $f16, flag      # exit if 999 was entered
        c.eq.d  $f16, $f0
        bc1t    out
```

```
        la      $a0, ans1       # print text part of answer
        li      $v0, 4
        syscall

        mov.d   $f12, $f0       # display value of x
        li      $v0, 3
        syscall

        la      $a0, ans2       # print more text of answer
        li      $v0, 4
        syscall

        mov.d   $f12, $f0       # call function exp
        jal     exp

        mov.d   $f12, $f0       # display exp approximation
        li      $v0, 3
        syscall

        la      $a0, cr         # finish line of output
        li      $v0, 4
        syscall

        j       loop            # branch back for next value of x

out:    la      $a0, adios      # display closing
        li      $v0, 4
        syscall

        li      $v0, 10         # exit from the program
        syscall

        .data
intro:  .asciiz "Let's test our exponential function!\n"
req:    .asciiz "Enter a value for x (or 999 to exit): "
ans1:   .asciiz "Our approximation for e^"
ans2:   .asciiz " is "
cr:     .asciiz "\n"
adios:  .asciiz "Come back soon!\n"
one:    .double 1.0
zero:   .double 0.0
tiny:   .double 1.0e-15
flag:   .double 999.0
```

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.]?

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?

My program calculates the exact answer for n = 21.
The answer for n = 22 is close, but not exact!
The smallest value of n that produces an infinite result is n = 171.

Here is my driver program:

```
# Stephen P. Leach -- 11/05/15
# dpfact.asm - a simple driver program that tests the iterative
#           double precision factorial function
# Register use:
#           $a0       parameter for syscall and fact
#           $v0       parameter and return value for syscall
#           $s0       Used to save value of n
#           $f0       return value for fact
#           $t0       temporary calculations

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
        li        $v0, 4
        syscall

loop:   la        $a0, req                # request value of n
        li        $v0, 4
        syscall

        li        $v0, 5                  # read value of n
        syscall

        slt       $t0, $v0, $zero         # if n < 0, exit
        bne       $t0, $zero, out

        move      $s0, $v0                # save value of n
```

```
        move   $a0, $v0          # place value of n in $a0

        jal    dpfact            # call dpfact function

        move   $a0, $s0          # display result
        li     $v0, 1            # print value of n
        syscall

        la     $a0, ans          # print text part of answer
        li     $v0, 4
        syscall

        mov.d  $f12, $f0         # print double precision
        li     $v0, 3            # version of fact(n)
        syscall

        la     $a0, cr           # print carriage return
        li     $v0, 4
        syscall

        j      loop              # branch back and next value of n

out:    la     $a0, adios        # display closing
        li     $v0, 4
        syscall

        li     $v0, 10           # exit from the program
        syscall

        .data
intro:  .asciiz "Welcome to the double precision factorial tester!\n"
req:    .asciiz "Enter a value for n (or a negative value to exit): "
ans:    .asciiz "! is "
cr:     .asciiz "\n"
adios:  .asciiz "Come back soon!\n"
```

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