Iho Lopez Tobi
Il16d

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).

   a) Work:
      Write number in binary 0x0c000000 = 00001100000000000000000000

      Get the twos compliment by flipping all the bits then add 1:

      11110011111111111111111111111

      + 1

      11110100000000000000000000000
      Then convert to decimal **-4093640704**

   b) Get the binary representation if the number and then turn it into an int
   $(0 \times 16^7) + (12 \times 16^6) + (0 \times 16^5) + (0 \times 16^4) + (0 \times 16^3) + (0 \times 16^2) + (0 \times 16^1) + (0 \times 16^0) = 201326592$

   c) Work:
      Operation encoding 00001100000000000000000000
      First 6 numbers is the opcode which leads to the jal MIPS instruction with address at 0.

   d) Mantissa=0 Exponent=24 S=0
      127-24=103
      number = $1 * 1 * 2^{-103}$
      number = $9.8607613 * 10^{-32}$

a) 11110100000000000000000000000000 = **-4093640704**
b) 000011000000000000000000000000 = **201326592**
c) 000011 00000000000000000000000 = jal
d) 0 00011000 0000000000000000000
Mantissa=0 Exponent=24 S=0
127-24=103
number = 1 *1*2^{-103}
number = 9.8607613 * 10^{-32}

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

    a) 0xC4630000 = Binary: 1100 0100 0110 0011 0000 0000 0000 0000
    1's Complement: 0011 1011 1001 1100 1111 1111 1111 1111.
    2's Complement: 0011 1011 1001 1101 0000 0000 0000 0000.= **-1000144896**
    b) 1100 0100 0110 0011 0000 0000 0000 0000= 3,294,822,400
    c)lwc1 $v1,0($v1)
    d) number= 1100 0100 0110 0011 0000 0000 0000 0000
    the 1 indicates that is a negative number.
    Exp= 0x88= 136-127=9
    Fraction=(fist 8 digits) 1100 0100 0xC6= 12*16^-1+6*16^-2=0.7734375
    ANS= -1.7734375*2^9=-908

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

    63.25*10^0= 11111.01*2^0
    Normalize the number= 1.1111101^2^5
    Exponent= 127+5=132
    Binary value: 0 1000 0100 1111 1101 0000 0000 0000 0000
    Binary representation assuming IEEE754 single precision
    0 10000100 1111101 0000000000000000

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

    63.25*10^0= 11111.01*2^0
    Normalize the number= 1.1111101^2^5
    Exponent= 1023+5=1028
    Binary representation assuming IEEE754 double precision
    0 10000000100 1111101
    000000000000000000000000000000000000000000000

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)

800C 0000= 1000 0000 0000 11000 0000 0000 0000 0000
1 indicates that number is negative
Exponents is the following 8 digits (0000 0000)= 0-127
Then we denormalize the number

1*2^0=	1
1*2^0^-1=	0.5
0.5*2^-2=	0.25
0.25*2^-3=	0.125
0.125*2^-4=	0.0625
0.0625*2^-5=	0.03125

Total= 1+0.03125= 1.03125 then 1.03125+2^-127= -128.90625=
-1.2890625*10^-7

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

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

```
#Iho Lopez

# e ^ x = 1 + x / 1 + x ^ 2 / 2! + x ^ 3 / 3! + ...

    .text
exp:

    mov.d $f10 ,$f12    # f0 is result, $f12 is input
    abs.d $f12, $f12    # x = abs(x)
    l.d $f0, one        # f0 is 1        # $f12 = 999
    li $t0, 1           # count

    addi    $sp, $sp, -4  # adjust stack
    sw    $ra, 0($sp)         # save the return address

    jal loop

    lw    $ra, 0($sp)       # restore the return address
    addi    $sp, $sp, 4    # adjust pointer

    l.d $f12, zero
    c.lt.d $f6, $f10            # is input < 0 ?
    bc1f inverse                # then exit

    jr $ra

inverse:
    l.d $f12, one
    div.d $f0, $f12, $f0
    jr $ra

loop:

    li $t1, 0              #  next term
    l.d $f2, one
    addi    $sp, $sp, -4    # adjust stack
    sw    $ra, 0($sp)          # save return address
    jal term #term loop
    lw    $ra, 0($sp)             # restore return address
    addi    $sp, $sp, 4        # adjust stack pointer to pop 2 items

    l.d $f4, conse          # $f4 = 1.0e-15   the message is term
    div.d $f6, $f2, $f0       # f6 = res / sum
    c.lt.d $f6, $f4          # is divisor < 1.0e-15 ?
    bc1t expr
    nop
```

```
    add.d $f0, $f0, $f2
    addi $t0, $t0, 1

    j loop

term:
    beq $t0, $t1, termexit        # for t1 = 0 to t0
    mul.d $f2, $f12, $f2        # f2 = f2 * f12
    addi $t1, $t1, 1            # t1 += 1
    mtc1.d $t1, $f4
    cvt.d.w $f4, $f4
    div.d $f2, $f2, $f4            # f2 = f2 / f13
    j term

termexit:
    jr $ra

expr:
    jr $ra

exit:
    la    $a0, adios
    li    $v0, 4
    syscall

    li    $v0, 10                # exit
    syscall

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

    j expdriver

    expdriver:
    la    $a0, enter            # ask for input
    li    $v0, 4
    syscall

    li    $v0, 7               # read value of x as double
    syscall

    l.d $f12, flag            # $f12 = 999        f16 is f12
    c.eq.d $f0, $f12          # is x == 999 ?
    bc1t exit                 # exit
    nop

    la    $a0, answer         # display result message
    li    $v0, 4
    syscall

    li $v0, 3                #input
    mov.d $f12, $f0
    syscall
```

```
    la    $a0, is
    li    $v0, 4
    syscall

    jal exp                  # if x != 999 then call exp

    li $v0, 3               #result
    mov.d $f12, $f0
    syscall

# print exp
    j expdriver
```

```
    .data

intro:  .asciiz "Let's test our exponential function"
enter: .asciiz "\nEnter a value for x (or 999 to exit): "
answer:   .asciiz "\nOur approximation for e^"
is:    .asciiz " is "
adios: .asciiz "\nCome back soon!"

flag:  .double   999.0
zero:  .double    0.0
one:   .double     1.0
conse: .double   1.0e-15
```

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)

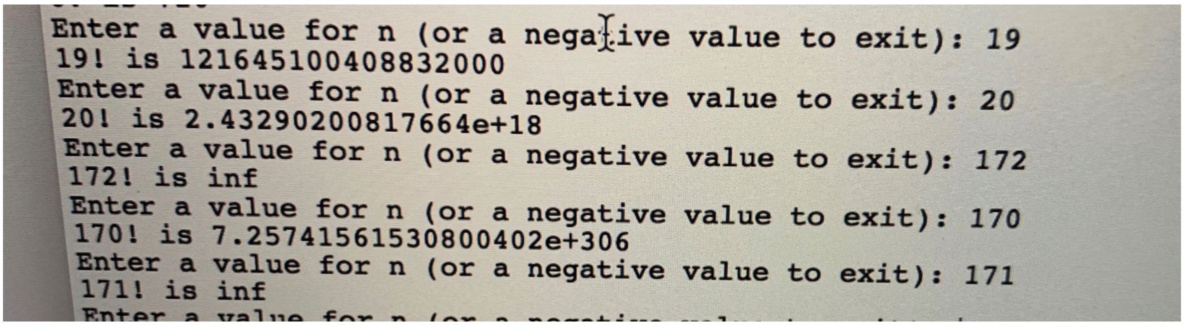
#Iho Lopez Tobi

```
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, enter       # "Request a value for n"
  li    $v0, 4
  syscall

  li    $v0, 5           # Read value of n
  syscall

  move  $a0, $v0         # save the value into $a0


  blt   $a0, $zero, out   # if n < 0, exit program

  li    $v0, 1           # Print the value of n
```

```
    syscall

    addi    $sp, $sp, -4     # push the stack
    sw    $a0, 0($sp)

    la    $a0, answer       # "! is "
    li    $v0, 4
    syscall

    lw    $a0, 0($sp)       # restore stack
    addi    $sp, $sp, 4

    jal    dpfact          # call the "dpfact"

    mov.d $f12, $f0        # copy the answer

    li    $v0, 3          # print $a0
    syscall

    la    $a0, newline    # print a newline
    li    $v0, 4
    syscall

    j    loop


out:  la    $a0, adios
      li    $v0, 4
      syscall

      li    $v0, 10        # exit from the program
      syscall

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

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.]? 19
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



```
Enter a value for n (or a negative value to exit): 19
19! is 121645100408832000
Enter a value for n (or a negative value to exit): 20
20! is 2.43290200817664e+18
Enter a value for n (or a negative value to exit): 172
172! is inf
Enter a value for n (or a negative value to exit): 170
170! is 7.257415615308004e+306
Enter a value for n (or a negative value to exit): 171
171! is inf
```

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