#AUSTIN RICKLI

#CALVIN CRAMER

#CHASE MAGUIRE

#NULL ATWOOD FloatingPointSimulator.asm

#RYAN GORMAN

#COMSC 142 FINAL PROJECT

**.data**

new\_line\_string:.asciiz "\n"

tab\_string: .asciiz "\t"

op\_input\_buffer:.space 2 #reserves 1 byte for an ASCII character, and null terminator

#allows user to only enter one character, and not wait for 'enter'

fp0: .float 0.0

#enumeration for operators: (1 for plus, 2 for minus, 3 for mult, 4 for div)

#macros are like functions

#all macros overwrite $v0 and $a0 (because they are syscalls)

#prints a new line

**.macro** PRINT\_NEWLINE

li $v0, 4

la $a0, new\_line\_string

syscall

**.end\_macro**

#prints a tab

**.macro** PRINT\_TAB

li $v0, 4

la $a0, tab\_string

syscall

**.end\_macro**

#prints a null-terminated string literal

**.macro** PRINT\_STR (%str)

.data

str\_to\_print: .asciiz %str

.text

li $v0, 4

la $a0, str\_to\_print

syscall

**.end\_macro**

#prints a register as an integer

**.macro** PRINT\_INT (%x)

li $v0, 1

add $a0, $0, %x

syscall

**.end\_macro**

#prints a register as a 32 bit floating point number

**.macro** PRINT\_FLOAT (%x)

li $v0, 2

mtc1 %x, $f12

syscall

**.end\_macro**

#prints a register in hexadecimal

**.macro** PRINT\_HEX (%x)

li $v0, 34

add $a0, $0, %x

syscall

**.end\_macro**

#prints a register in binary

**.macro** PRINT\_BIN (%x)

li $v0, 35

add $a0, $0, %x

syscall

**.end\_macro**

**.text**

**main:**

PRINT\_STR("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n") #user will enter at bottom of console

jal get\_user\_input #call get user input

la $s0, ($v0) #get op1 return value

la $t8, ($a0) #get op

la $s4, ($v1) #get op2

#gets parts of operand1

la $a0, ($s0) #going to call get\_sign

jal get\_sign #get sign of op1

la $s1, ($v0) #store sign

la $a0, ($s0) #going to call get\_exponent

jal get\_exponent #get exponent of op1

la $s2, ($v0) #store return value

la $a0, ($s0) #setting up to call get\_fraction

jal get\_fraction #get fraction of op1

la $s3, ($v0) #get fraction return value

#gets parts of operand2

la $a0, ($s4) #going to call get\_sign

jal get\_sign #get sign of op2

la $s5, ($v0) #store sign

la $a0, ($s4) #going to call get\_exponent

jal get\_exponent #get exponent of op2

la $s6, ($v0) #store return value

la $a0, ($s4) #setting up to call get\_fraction

jal get\_fraction #get fraction of op2

la $s7, ($v0) #get fraction return value

#s0 = op1, $s1 sign, $s2 exp, $s3 fraction

#s4 = op2, $s5 sign, $s6 exp, $s7 fraction

#t8 = operand

#output for testing purposes

#jal output\_test

#save needed registers on stack

addi $sp, $sp, -12

sw $s0, 8($sp) #op1

sw $s4, 4($sp) #op2

sw $t8, 0($sp) #operand

#store registers to call arithemtic function

#all arithmetic functions must use these registers as paramaters

la $v0, ($s1) #op1 sign

la $v1, ($s2) #op1 exp

la $a0, ($s3) #op1 fraction

la $a1, ($s5) #op2 sign

la $a2, ($s6) #op2 exp

la $a3, ($s7) #op2 fraction

#clear $s1-$s7 BECAUSE FUNCTIONS NEED TO ADHERE TO FUNCTION CALLING PROTOCOLS

li $s1, 0

li $s2, 0

li $s3, 0

li $s5, 0

li $s6, 0

li $s7, 0

#call appropriate function

beq $t8, 1, main\_plus #1,2,3,4 enumerations for plus, minus, mult, div

beq $t8, 2, main\_minus

beq $t8, 3, main\_mult

beq $t8, 4, main\_div

PRINT\_STR("ERROR: BAD OPERAND NUMBER: ")

PRINT\_INT($t8)

PRINT\_NEWLINE

j main\_end

**main\_plus:**

jal add\_fp

j main\_end\_arithmetic\_call

**main\_minus:**

jal sub\_fp

j main\_end\_arithmetic\_call

**main\_mult:**

jal mult\_fp

j main\_end\_arithmetic\_call

**main\_div:**

jal div\_fp

j main\_end\_arithmetic\_call

**main\_end\_arithmetic\_call:**

#store result of calculated answer

la $t0, ($v0)

#restore registers from stack

lw $t3, 0($sp) #operand

lw $t2, 4($sp) #op2

lw $t1, 8($sp) #op1

addi $sp, $sp, 12

#calculate the actual answer using the floating point processor

mtc1 $t1, $f1

mtc1 $t2, $f2

beq $t3, 1, main\_plus\_actual

beq $t3, 2, main\_minus\_actual

beq $t3, 3, main\_mult\_actual

beq $t3, 4, main\_div\_actual

PRINT\_STR("ERROR: BAD OPERAND NUMBER: ")

PRINT\_INT($t3)

PRINT\_NEWLINE

j main\_end

**main\_plus\_actual:**

add.s $f0, $f1, $f2

j main\_end\_actual\_call

**main\_minus\_actual:**

sub.s $f0, $f1, $f2

j main\_end\_actual\_call

**main\_mult\_actual:**

mul.s $f0, $f1, $f2

j main\_end\_actual\_call

**main\_div\_actual:**

div.s $f0, $f1, $f2

j main\_end\_actual\_call

**main\_end\_actual\_call:**

#store actual result in normal registers

mfc1 $t4, $f0

#$t0 calculated answer

#$t1 op1

#$t2 op2

#$t3 operand

#$t4 actual answwer

#display both results

PRINT\_NEWLINE

PRINT\_STR("Calculated:")

PRINT\_TAB

PRINT\_FLOAT($t0)

PRINT\_TAB

PRINT\_TAB

PRINT\_HEX($t0)

PRINT\_TAB

PRINT\_BIN($t0)

PRINT\_NEWLINE

PRINT\_STR("Actual: ")

PRINT\_TAB

PRINT\_FLOAT($t4)

PRINT\_TAB

PRINT\_TAB

PRINT\_HEX($t4)

PRINT\_TAB

PRINT\_BIN($t4)

PRINT\_NEWLINE

PRINT\_NEWLINE

#test wether results are equal

beq $t0, $t4, main\_answer\_good

PRINT\_STR("INCORRECT CALCULATED ANSWER")

j main\_answer\_end

**main\_answer\_good:**

PRINT\_STR("ANSWERS MATCH")

**main\_answer\_end:**

PRINT\_NEWLINE

#continue if user enters a certain letter (q or space to quit, enter to continue)

PRINT\_NEWLINE

PRINT\_STR("Press 'Enter' to redo, anything else to quit: ")

li $v0, 8

la $a0, op\_input\_buffer

li $a1, 2 #max chars

syscall

la $t5, op\_input\_buffer #temporarily get start address for character input

lb $t5, 0($t5) #get ASCII code for character entered

bne $t5, 0x0A, main\_end #branch to end if not equal to 'enter'

#clear registers for clenliness

li $v0, 0

li $v1, 0

li $a0, 0

li $a1, 0

li $a2, 0

li $a3, 0

li $t0, 0

li $t1, 0

li $t2, 0

li $t3, 0

li $t4, 0

li $t5, 0

li $t6, 0

li $t7, 0

li $s0, 0

li $s1, 0

li $s2, 0

li $s3, 0

li $s4, 0

li $s5, 0

li $s6, 0

li $s7, 0

li $t8, 0

li $t9, 0

l.s $f0, fp0

l.s $f1, fp0

l.s $f2, fp0

l.s $f12, fp0

j main #start from beginning

**main\_end:**

#ending program sequence

PRINT\_STR("\nExiting program...")

li $v0, 10 #end program by syscall to end

syscall

#end main

#######################################################################################################

**add\_fp:**

j sub\_fp

#end add

#######################################################################################################

#Subtract two numbers provided by calling function

#sign, exp, mant => vo, v1, a0; a1, a2, a3

**sub\_fp:**

addi $v1, $v1, 127

addi $a2, $a2, 127

**exp\_zero\_check:** #Checks if exponents are equal to ZERO

beq $v1, $zero, mant\_1\_zero

beq $a2, $zero, mant\_2\_zero

j sub\_cont

**mant\_1\_zero:** #Checks if op1 mantissa is equal to ZERO

beq $a0, $zero, ans\_is\_op2

j sub\_cont

**mant\_2\_zero:** #Checks if op2 mantissa is equal to ZERO

beq $a3, $zero, ans\_is\_op1

j sub\_cont

**ans\_is\_op2:** #Returns the value of Operand 2

addi $t7, $0, 2

add $a0, $a1, $zero

addi $a1, $a2, -127

add $a2, $a3, $zero

beq $t8, $t7, ans\_is\_neg2

j recombine\_fp

**ans\_is\_neg2:**

xori $a0, $a0, 0xFFFFFFFF

j recombine\_fp

**ans\_is\_op1:**

add $a2, $a0, $zero #Returns the value of Operand 1

add $a0, $v0, $zero

addi $a1, $v1, -127

j recombine\_fp

**sub\_cont:**

ori $t0, $a0, 0x00800000

ori $t1, $a3, 0x00800000 #change one from implicit to explicit

sll $t0, $t0, 6

sll $t1, $t1, 6 #shift number portion to desirable spot (leave one space for sign, one space for growth)

blt $v1, $a2, subeq1

blt $a2, $v1, subeq2 #branch to one of two subfunctions to set exponents equal and shift the number

j subbody #skip the subfunctions if exponents are equal

**subeq1:**

sub $t2, $a2, $v1

srlv $t0, $t0, $t2

add $v1, $v1, $t2

j subbody

**subeq2:**

sub $t2, $v1, $a2

srlv $t1, $t1, $t2

add $a2, $a2, $t2

j subbody

**subbody:**

sll $t2, $v0, 31

sll $t3, $a1, 31 #move sign to leftmost bit

or $t0, $t0, $t2

or $t1, $t1, $t3 #combine sign and number

beq $v0, $0, subskip1 #if positive, already in two's comp

xori $t0, $t0, 0x7FFFFFFF #flip bits aside from leftmost

addi $t0, $t0, 0x00000001 #add 1

**subskip1:**

beq $a1, $0, subskip2

xori $t1, $t1, 0x7FFFFFFF

addi $t1, $t1, 0x00000001

**subskip2:**

addi $t6, $0, 1

beq $t8, $t6, subadd

sub $t0, $t0, $t1 #subtract and get our answer

j addskip

**subadd:**

add $t0, $t0, $t1

**addskip:**

andi $t1, $t0, 0x80000000 #t1 gets our sign which is already in the right spot for our answer

beq $t1, $0, subskip3 #if positive, already in sign-mag

subi $t0, $t0, 1

xori $t0, $t0, 0x7FFFFFFF

**subskip3:**

addi $t2, $0, -2 #set a counter that will come up as -1 if the magnitude increased 1, or otherwise show the decrease

addi $t5, $0, 30

**subadjloop:**

addi $t2, $t2, 1 #increment counter

sll $t0, $t0, 1 #shift the answer left one

andi $t3, $t0, 0x80000000 #check if leftmost bit is a one

beq $t2, $t5, subzero #if nothing has come up as a one after 30 loops, the answer is zero

beq $t3, $0, subadjloop #loop if leftmost bit is still zero

sll $t0, $t0, 1 #knock off the implied one

srl $t0, $t0, 9 #set the mantissa into the correct spot

or $t0, $t0, $t1 #put the sign in place

sub $t1, $v1, $t2 #adjust exponent

sll $t1, $t1, 23 #move exponent

or $t9, $t0, $t1 #place exponent

add $v0, $t9, $0

jr $ra #return answer

**subzero:**

add $v0, $0, $0

jr $ra

#end sub

#######################################################################################################

**mult\_fp:**

#v0= first sign, v1= first exponent, a0= first mantissa

#a1= second sign, a2= second exponent, a3= second mantissa

# Checking to see if either of the values == 0

# Since the iee format will be broken up, i can just check if one or the other

# exponent & mantissa equals all zero, if so, the output =0.

addi $v1, $v1, 127

addi $a2, $a2, 127

#First, check if either = 0

beqz $v1, mult\_checkFirstMan #if the first exponent =0, check the mantissa

**mult\_clearOne:**

beqz $a2, mult\_checkSecondMan # if the second exponent =0, check the mantissa

**mult\_clearTwo:**

# Both are clear, so do math

xor $t6, $v0, $a1 #getting the new sign

add $t7, $v1, $a2 # getting new exponent

addi $t7, $t7, -254 # Getting the exponent to the actual, base ten exponent.

addi $t7, $t7, 127 #Shift it to biased, cannot call to recombine\_fp

ori $a0, $a0, 0x00800000

ori $a3, $a3, 0x00800000

lui $t1 ,0x8000

mult $a0, $a3

mfhi $t3

mflo $t0

sll $t3, $t3, 16

srl $t0, $t0, 16

or $t3, $t3, $t0

and $t4, $t3, $t1 #checking if needs to be normalized

beqz $t4, mult\_notNormal #if it passes, it needs to be normalized, so it will have 1 added to the exponent

addi $t7, $t7, 1

#Checking if there is underflow here

li $t5, 255

beq $t5, $t7, mult\_overflow #if this is true, overflow is detected

li $t5, 0

beq $t5, $t7, mult\_underflow

**mult\_notNormal:**

**mult\_shift:**

and $t2, $t3, $t1

sll $t3, $t3, 1

beqz $t2, mult\_shift

srl $t8, $t3, 9

#a1 exponent (8 bits) (signed, so will add 127 to it)

#a2 mantissa (23 bits)

#returns IEEE 754 single precision FP number from given parts in $v0

move $v0, $t8

sll $t6, $t6, 31 #sign bit to left-most bit

or $v0, $v0, $t6 #place sign bit in $v0

sll $t7, $t7, 24 #shift exp to left-most, then shift back (so as onlt the correct 8 bits are set)(32-8=24)

srl $t7, $t7, 1 #shift exp to proper place

or $v0, $v0, $t7 #place exponent in $v0

jr $ra #we done

**mult\_checkFirstMan:** # Checking the mantissa

beqz $a0, mult\_setToZero #If it equals zero, set it all == o

j mult\_clearOne

**mult\_checkSecondMan:** # Checking the second mantissa

beqz $a3, mult\_setToZero #If it equals zero, set it all == o

j mult\_clearTwo

**mult\_setToZero:** # If either of the tests passed, setting everything equal to 0

li $v0, 0

jr $ra

**mult\_overflow:**

#Overflow detected, do something here

**mult\_underflow:**

#underflow detected, do something else here

# Need rounding

#end mult

#######################################################################################################

**div\_fp:**

#Since mips only offers integer division, we need to work around this

#The algorithim to find the new mantissa is essentailly just a long divison algorithm

#If the divisor fits into the dividend, it returns a 1, grabs the remainder of that number

# then repeats from there. If not, it adds a zero to the dividend and tries again

#v0= first sign, v1= first exponent, a0= first mantissa THE NUMERATOR

#a1= second sign, a2= second exponent, a3= second mantissa THE DENOMINATOR

# Will not be calling recombine\_Fp, tis not needed

addi $v1, $v1, 127 #put exponents back in biased

addi $a2, $a2, 127

#Four cases: #/#: normal division, #/0: Infinity, 0/#: 0, 0/0: NaN

add $s1, $v1, $a0 #$s1 will be zero if op1 exp and fraction are zero (DISREGARDS SIGN BIT because -0.0

add $s2, $a2, $a3 #$s2 will be zero if op2 exp and fraction are zero IS SAME AS +0.0)

bnez $s1, div\_op1\_notZero

bnez $s2, div\_returnZero

li $v0, 0x7FFFFFFF #op1 == 0, op2 == 0, return NaN

jr $ra #NAN = sign 0, exp 1's, fraction 1's

**div\_returnZero:**

li $v0, 0 #op1 == 0, op2 == a number

jr $ra # 0/# == 0

**div\_op1\_notZero:**

bnez $s2, div\_ops\_notZero

li $v0, 0x7F800000 #op1 == a number, op2 == 0

jr $ra #return INF (sign =0, exp = 1's, fraction = 1's)

**div\_ops\_notZero:**

# both are non zero numbers, do actual division

xor $t6, $v0, $a1 #getting new sign GOOD

sub $t7, $v1, $a2 #getting new exponent GOOD

addi $t7, $t7, 127 #putting bias in place

#ori $a0, $a0,0x00800000 #add the implicit one to op1 fraction

#ori $a3, $a3,0x00800000 #add the implicit one to op2 fraction

sll $a3, $a3, 9 #shift only divisor (op2 fraction) to rightmost bit

li $t9, 0 #quotient register

li $t8, 0 #loop counter

**div\_loop:**

bgtu $t8, 32, div\_exit

subu $a0, $a0, $a3 #remainder -= divisor

sll $t9, $t9, 1 #shift quotient left 1

blez $a0, div\_quo\_gtZero #answer if negative (in 2's comp) if leftmost bit is a 1

addu $a0, $a0, $a3 #remainder += divisor (restore from earlier)

j div\_endif #set rightmost bit of quotient to 0 (by doing nothing)

**div\_quo\_gtZero:**

addi $t9, $t9, 1 #set rightmost bit of quotient to 1

**div\_endif:**

srl $a3, $a3, 1 #shift divisor right by 1

add $t8, $t8, 1 #increment counter

j div\_loop

**div\_exit:**

srl $t9, $t9, 9 #shift remainder back 9 bits

PRINT\_HEX($t9)

PRINT\_NEWLINE

#srl $a0, $t9, 9 #shift quotient back

#put it back together

sll $t6, $t6, 31 #sign bit to left-most bit

or $v0, $t9, $t6 #place sign bit and quotient

sll $t7, $t7, 24 #shift exp to left-most, then shift back (so as onlt the correct 8 bits are set)(32-8=24)

srl $t7, $t7, 1 #shift exp to proper place

or $v0, $v0, $t7 #place exponent in $v0

jr $ra #we done

#end div

#######################################################################################################

**get\_user\_input:**

#no arguments passed

#returns op1 in $v0,

#op2 in $v1,

#op in $a0 (1 for plus, 2 for minus, 3 for mult, 4 for div)

PRINT\_STR("Operand1:\t")

li $v0, 6 #get op1

syscall

mfc1 $t0, $f0 #store op1 in $t0

**gui\_get\_operand:**

PRINT\_STR("Operand:\t")

li $v0, 8 #get op character

la $a0, op\_input\_buffer

li $a1, 2 #max chars

syscall #will return once user enters 1 character

PRINT\_NEWLINE #so need to go to newline since user cannot press enter

la $t4, op\_input\_buffer #temporarily get start address for character input

lb $t4, 0($t4) #get ASCII code for character entered

beq $t4, 0x2B, gui\_plus #branch for each good operand

beq $t4, 0x2D, gui\_minus

beq $t4, 0x2A, gui\_mult

beq $t4, 0x2F, gui\_div

#print bad character string

PRINT\_STR("ONLY '+', '-', '\*', AND '/' ARE ALLOWED\n")

j gui\_get\_operand #wasn't a +,-,\*, or /. So retry

**gui\_plus:**

li $t1, 1 #store enumeration of plus in $t1

j gui\_good\_operand

**gui\_minus:**

li $t1, 2

j gui\_good\_operand

**gui\_mult:**

li $t1, 3

j gui\_good\_operand

**gui\_div:**

li $t1, 4

j gui\_good\_operand

**gui\_good\_operand:**

PRINT\_STR("Operand2:\t")

li $v0, 6 #get op2

syscall

mfc1 $t2, $f0 #store op2 in $t0

#store return values

la $v0, ($t0) #store op1

la $a0, ($t1) #store op

la $v1, ($t2) #store op2

#clear used registers

li $t0, 0

li $t1, 0

li $t2, 0

li $t4, 0

jr $ra #return

#end input

#######################################################################################################

**get\_fraction:**

#$a0 floating point number

#returns $v0, the unsigned fractional part of a 32 bit floating point number

#ie bit 0 to bit 22

#does not add the implicit 1

andi $v0, $a0, 0x007FFFFF #clear unwanted bits

jr $ra #return

#end get\_fraction

#######################################################################################################

**get\_exponent:**

#a0 floating point number

#returns signed part of a 32 bit floating point number

#bits 23 to 30

#basically gets bits and subtracts 127 from them

#remember this is 2^exponent, so the returned value wont be the same as 10^exponent

andi $v0, $a0, 0x7F800000 #clear all bits except exponent part

srl $v0, $v0, 23 #shift exponent to right most bit

sub $v0, $v0, 127 #shift for biased

jr $ra #return

#end get\_exponent

#######################################################################################################

**get\_sign:**

#a0 floating point number

#returns the sign of a 32 bit floating point number (0 for positive, 1 for negative)

#bit 31

andi $v0, $a0, 0x80000000 #clear all bits except for sign bit

srl $v0, $v0, 31

jr $ra #return

#end get\_sign

#######################################################################################################

**recombine\_fp:**

#a0 sign bit

#a1 exponent (8 bits) (signed, so will add 127 to it)

#a2 mantissa (23 bits)

#returns IEEE 754 single precision FP number from given parts in $v0

li $v0, 0 #set $v0 to 0

addi $a1, $a1, 127 #shift exponent to biased

sll $a0, $a0, 31 #sign bit to left-most bit

or $v0, $v0, $a0 #place sign bit in $v0

sll $a1, $a1, 24 #shift exp to left-most, then shift back (so as onlt the correct 8 bits are set)(32-8=24)

srl $a1, $a1, 1 #shift exp to proper place

or $v0, $v0, $a1 #place exponent in $v0

sll $a2, $a2, 9 #shift mantissa to cut off any higher order bits

srl $a2, $a2, 9 #shift to proper place

or $v0, $v0, $a2 #place mantissa in $v0

jr $ra #return

#end recombine\_fp

#######################################################################################################