#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "fp.h"

int

computeFP(float val) {

// input: float value to be represented

// output: integer version in our representation

//

// Perform this the same way we did in class -

// either dividing or multiplying the value by 2

// until it is in the correct range (between 1 and 2).

// Your exponent is the number of times this operation

// was performed.

// Deal with rounding by simply truncating the number.

// Check for overflow and underflow -

// For underflow, return 0

// For overflow, return -1

//if val equal to 0, then done

if (val == 0)

return 0;

int e = 0;

//if value is larger than 2

if (val >= 2)

{

//let value divide 2 until value is between 1 and 2

while (val >= 2)

{

val = val / 2;

//everytime, E need to add 1;

e++;

}

}

//if the value is smaller than 1

else if (val < 1)

{

//let value multiply 2 until value is between 1 and 2

while (val < 1)

{

val = val \* 2;

//everytime, E need to delete 1

e++;

}

e = e \* -1;

}

//get the bias = 2^5 -1 = 31

int bias = 31;

//e = exponent - bias

int exp = bias + e;

//For overflow, return -1

if (exp >= 63)

return -1;

//For underflow, return 0

if (exp <= 0)

return 0;

//we only need the mantissa part

val = val -1;

float eightFractionBits = val \* pow(2,8);

int fraction = (int)eightFractionBits;

//fpVal is the value will be returned (integer version in our representation)

int fpVal = 0;

//pack the value

fpVal = fpVal | exp;

fpVal = fpVal << 8;

fpVal = fpVal | fraction;

return fpVal;

}

float getFP(int val) {

// Using the defined representation, compute the floating point

// value

// For denormalized values (including 0), simply return 0.

// For special values, return -1;

//get the fraction part from the val

float fraction = (val & 0xFF);

fraction = (fraction/(float)pow(2,8)) + 1;

//get the exponent part from the val

int exp = ((val >> 8) & 0x3F);

// For special values, return -1;

if (exp == 0x3F)

return -1;

// For denormalized values (including 0), simply return 0.

if (exp == 0)

return 0;

int bias = 31;

int e = exp - bias;

//convert the fraction part to the integer

//if exponent lager than 0 then the fraction need to multiply pow(2,e)

if (e > 0)

{

while (e > 0)

{

fraction = fraction \* 2;

e --;

}

}

// if exponent smaller than 0 then the fraction need to multiple pow(2,e) = /2^e

else if (e < 0)

{

while (e < 0)

{

fraction = fraction / 2;

e ++;

}

}

return fraction;

}

int

multVals(int source1, int source2) {

// You must implement this by using the algorithm

// described in class:

// Add the exponents: E = E1+E2

// multiply the fractional values: M = M1\*M2

// if M too large, divide it by 2 and increment E

// save the result

// Be sure to check for overflow - return -1 in this case

// Be sure to check for underflow - return 0 in this case

//get the fraction part from the source1 and source2;

float frac1 = (source1 & 0xFF)/(float)(pow(2,8)) + 1;

float frac2 = (source2 & 0xFF)/(float)(pow(2,8)) + 1;

//get the exponent part from the source1 and source2;

int exp1 = ((source1 >> 8) & 0x3F);

int exp2 = ((source2 >> 8) & 0x3F);

//add the E1 and E2 get the E3 for the multVals

int e1 = exp1 - 31;

int e2 = exp2 - 31;

int e3 = e1 + e2;

//calculate the return fraction from the fraction1 and fraction2

float retFrac = frac1 \* frac2;

//if retFraction bigger than two must rejustify

if (retFrac >= 2)

{

retFrac = retFrac / 2;

e3++;

}

//if (E3 + bias) equal to 111111(63) - overflow

if ((e3 + 31) > 62)

return -1;

//if (E3 + bias) equal to 000000(0) - underflow

if ((e3 + 31) < 1)

return 0;

int retExp = e3 + 31;

retFrac = retFrac - 1;

float eightFractionBits = (float)retFrac \* pow(2,8);

int fraction = eightFractionBits;

//multVal is the value will be returned (integer version in our representation)

int multVal = 0;

//pack the multvalue

multVal = multVal | retExp;

multVal = multVal << 8;

multVal = multVal | fraction;

return multVal;

}

int

addVals(int source1, int source2) {

// Do this function last - it is the most difficult!

// You must implement this as described in class:

// If needed, adjust one of the two number so that

// they have the same exponent E

// Add the two fractional parts: F1' + F2 = F

// (assumes F1' is the adjusted F1)

// Adjust the sum F and E so that F is in the correct range

//

// As described in the handout, you only need to implement this for

// positive, normalized numbers

// Also, return -1 if the sum overflows

//get the fraction part from the source1 and source2;

float frac1 = (source1 & 0xFF)/(float)(pow(2,8)) + 1;

float frac2 = (source2 & 0xFF)/(float)(pow(2,8)) + 1;

//get the exponent part from the source1 and source2;

int exp1 = ((source1 >> 8) & 0x3F);

int exp2 = ((source2 >> 8) & 0x3F);

//add the E1 and E2 get the E3 for the multVals

int e1 = exp1 - 31;

int e2 = exp2 - 31;

//check which exp is greater

if (e1 > e2)

{

int different = e1 - e2;

//find the different between E

for (int i=0; i<different; i++)

{

// justify frac2

frac2 = frac2 / 2;

e2++;

}

}

else if (e2 > e1)

{

int different = e2 - e1;

//find the different between E

for (int i=0; i<different; i++)

{

// justify frac1

frac1 = frac1 / 2;

e1++;

}

}

//when two frac have the same E, we can add them

float newFrac = frac1 + frac2;

//if the fraction is greater than 2

//justify fraction down

//(e1 = e2)

if (newFrac >= 2)

{

newFrac = newFrac / 2;

//add expoent (e1)

e1++;

}

newFrac = newFrac - 1;

//check the overflow

if ((e1 + 31) > 62)

return -1;

float eightFractionBits = (float)newFrac \* pow(2,8);

int fraction = eightFractionBits;

int exponent = e1 + 31;

//pack the addVal

int addVal = 0;

addVal = addVal | exponent;

addVal = addVal << 8;

addVal = addVal | fraction;

return addVal;

}