CPE/CSC 315, Spring 2016 Retz

Lab 3: Floating Point Numbers

Purpose: to gain an understanding of how floating point numbers are represented, and to understand how floating point operations are carried out. In order to gain an understanding of how this is done in hardware, we write a simulation of a few floating point operations – using a combination of MIPS assembler and fixed point C operations. This is an emulation of floating point operation using fixed point arithmetic.

The MIPS portion of this lab uses SPIM, combined with your earlier bintohex conversion routine to allow display of hexadecimal numbers. The C portion of this lab project makes use of the lab gcc compilers, or you may use the servers at unix12.csc.calpoly.edu (login using your calpoly login) to run gcc. (Note: you need a ssh shell, such as PuTTY, to get connected to the server.)

In the C portion of this assignment, you may start with floating point numbers as floating point data type test cases, but then convert them to fixed point integer representation so that you can implement the operation in fixed point arithmetic – as is fundamental to a hardware implementation.

Note that you can convert a C floating point number into a fixed point unsigned integer variable bit-for-bit using the C typecast capability as follows:

```
float g;
unsigned int k;

g = 1.25;
k = (unsigned int) * (unsigned int *) &g;
```

This takes the 32-bit floating point number named g and stores it in a 32-bit unsigned integer named k. Once stored as a floating point integer, you can extract the various fields: sign, exp, and fraction - as bit fields.

For each of the problems below, you should allow entry (by either an input statement or by a constant declaration of the float) of each of the parameters for calculation by your floating point function – as implemented in fixed point format.

Note that you can verify format of the binary floating point number by printing it in hexadecimal using your bintohex routine followed by a syscall to print string. In the case of C functions, you can use printf ("k value = %08X\n", k);

1. Unpack IEEE 754 argument – obtain components (MIPS assembler)

This function accepts a parameter formatted as IEEE754 single precision value, and obtains the sign, unbiased exponent, and fraction fields. This program will insert the "hidden 1" to produce the actual fraction field to be used in the computation. You will use a MIPS register to store each of these fields.

This function assumes a 32-bit floating point value contained in $a0, and extracts each of the fields: a sign, a 2's complement exponent, and a 2's complement fraction in $t0, $t1, and $t2, respectively. The extracted exponent removes the "bias" and stores it as a positive or negative 32-bit number. The extracted fraction re-inserts the "hidden 1" and stores it as a 2's complement number with the binary point to the right of the sign bit.

2. Normalize an (exponent, fraction) (MIPS assembler)

This function normalizes a number where the exponent is contained in $t1 and the fraction is contained in $t2. It performs shifts on the fraction and adjustment of the exponent in order to represent the number as "normalized", where the sign bit of the fraction is different than the bit next to the sign bit.

The program must check for the special case of a fraction of 0, which would be considered normalized. The result is returned in $t1 and $t2.

3. Single precision floating point add (in C)

float single_float_add(float a, float b)

This function performs a floating point addition of two numbers, returning a result formatted as IEEE 754 format. Returns a single precision floating point number. (NOTE – all computation must be done using integer arithmetic!) Note that before performing the addition you must place the two numbers on a common scale, then perform the addition, then normalize the result.

See the notes above regarding "casting" of floats to integers.

4. Single precision floating point subtract (in C)

single_float_subtract(a, b)

This function performs a floating point subtraction (a-b) of two numbers, returning a result formatted as IEEE 754 format. Returns a single precision floating point number. (NOTE – all computation must be done using integer arithmetic!) See note in item 2, above.

5. Single precision floating point multiply (in C)

single_float_multiply(a,b)

Returned format must follow IEEE 754 formatting, and must be normalized. Must correctly handle multiplication of numbers of differing sign.

6. Pack IEEE 754 Result (in C)

This function is useful for all of the above operations, taking a set of fixed point fields and placing them in IEEE 754 number (single precision) format. This assumes a sign, unbiased exponent, and fraction fields and produces the standard 32-bit IEEE 754 format result (a float).

What to Hand In: Completed and fully documented listing of your program functions and the calling program in each case. See Lab Guidelines for overall format.