Lab 2

Design a custom IP in Vivado HLS to perform the following function using the AXI-Lite Interface:

```
def lab_2(A, B, C, Y):
      Y = log(A/B) + sqrt(B/C)
```

PYNQ memory-mapped I/Os (MMIO) do not support floating point numbers, therefore, we will be using fixed point numbers from the `ap_fixed` library provided by HLS.

Fixed Point Representation

This representation has a fixed number of bits for integer part and for fractional part. For example, if a given fixed-point representation is IIII.FFFF, then 4-bits are allocated for the integer and the fractional part each.

```
import <ap_fixed.h>
ap_fixed<W,I>
```

where , `W` is the word length in bits,

`I` is the number of bits allocated for the integer part,

`W-I` is the number of bits allocated for the fractional part

The smallest positive number that a fixed-point representation can denote is $1/2^{\mathrm{W-I}}$. This is also the resolution of the fixed-point number representation. Therefore, for a 8-bit representation with 4 bits reserved for the fractional part, we get a resolution of 2^{-4} = 0.0625.

The major advantage of using a fixed-point representation is performance. As the value stored in memory is an integer the CPU can take advantage of many of the optimizations that modern computers have to perform integer arithmetic without having to rely on additional hardware or software logic. This in turn can lead to increases in performance and when writing your apps, can therefore lead to an improved experience for your users.

However, there is a downside! Fixed Point Representations have a relatively limited range of values that they can represent. For instance, an 8-bit representation with 4 bits reserved for the integer part can only store values in the range (-2^3, 2^3-1) i.e. [-8, 7] with a resolution of 0.625 (= 2^{-4}).

To store larger numbers, we employ the floating-point number representation.

Floating Point Representation

Floating Point Notation is a way to represent very large or very small numbers precisely using scientific notation in binary. The scientific notation (in decimal) is as follows:

$$\pm \text{ mantissa x } 10^{\text{ exponent}}$$

The binary equivalent of the scientific notation is as follows:

$$\pm \text{ mantissa x } 2^{\text{ exponent}}$$

But since the scientific notation is flexible and provides multiple ways of writing the same number — for eg. $15 \text{ x } 10^{-1}$, $1.5 \text{ x } 10^{0}$, $150 \text{ x } 10^{-2}$ all represent 1.5 — thus we follow a common set of rules known as normalized scientific notation.

The Normalized Scientific Notation is stated as follows:
"We choose an exponent so that the absolute value of the mantissa remains greater than or equal to 1 but less than the number base."

Thus the correct representation of 1.5 is $1.5 \text{ x } 10^{0}$.

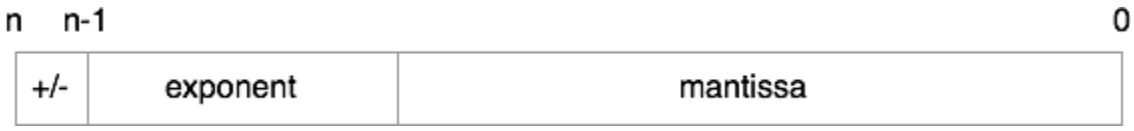
A few more examples, 10.1_2 is $1.01_2 \text{ x } 2^{1}$, and 0.111_2 is $1.11 \text{ x } 2^{-1}$.

Because of its wide use, the format used to store Floating Point numbers in memory has been standardized by the Institute of Electrical and Electronic Engineers as the **IEEE 754**. This standard defines a number of different binary representations that can be used when storing Floating Point Numbers in memory:

- Half Precision – Uses 16-bits of storage in total.
- Single Precision – Uses 32-bits of storage in total.
- Double Precision – Uses 64-bits of storage in total.
- Quadruple Precision – Uses 128-bits of storage in total.

In each of these cases, their basic structure is as follows:

$$(-1)^{\text{sign}} \text{ x mantissa x } 10^{\text{ exponent}}$$



Sign
If the **sign** bit is clear (a value of '0') the overall number is positive. If the bit is set (a value of '1') the number is negative.

Exponent
The **exponent** represents the power to which the mantissa will be raised. There are always a fixed number of exponent bits when storing a floating point representation in memory and the exact number of bits that are used is defined by the particular IEEE 754 representation (Single Precision, Double Precision etc).

The exponent needs to be capable of handling both positive and negative exponents. To avoid the complications of having to store the exponents in two's complement format, something called an **exponent bias** is used.

Exponent Bias is where the value stored for the exponent is offset from the actual exponent value by a bias. The bias is simply a number that is added to the exponent to ensure that the value that is stored is always positive.

Representation	Bits	Normal Range (Pre Bias)	Bias	Modified Range (Post Bias)
Half Precision	5	-14 to +15	+15	+1 to +30
SinglePrecision	8	-126 to +127	+127	+1 to +254
Double Precision	11	-1022 to +1023	+1023	+1 to 2046
Quadruple Precision	15	-16382 to +16383	+16383	+1 to + 32766

Biased values of 0 (all bits clear) and 30, for half precision, (all bits set) have special meaning.

Also, the exponent bias implies that to store -14 (for half precision representation) we store +1 in the exponent. Similarly to store 0, we store 0 + exponent_bias = +15

Mantissa (aka Significand) Bits
In the IEEE 754 representations, the mantissa is expressed in normalized form. The formats follow the same rules for normalization as we saw with Scientific Notation, and put the radix point after the first non-zero digit.

In binary though, we also get a nice little bonus!

As we are expressing our numbers in binary, we get the benefit of knowing that the first non-zero digit will always be a 1 (after all we can only have 1's or 0's). Given this, we are therefore able to drop that first bit, simply assuming it is there, and instead gain an additional (implicit) bit of precision for storing the fractional part.

When numbers are stored, we only store the part of the mantissa that represents the fractional part of the number, the part to the right of the radix point.

Representation	Precision Bits	Effective Precision
Half Precision	1 bit (implicit) + 10 bits (explicit)	11 bits
SinglePrecision	1 bit (implicit) + 23 bits (explicit)	24 bits
Double Precision	1 bit (implicit) + 52 bits (explicit)	53 bits
Quadruple Precision	1 bit (implicit) + 112 bits (explicit)	113 bits

Summary
In summary then, the IEEE 754 standard defines four main formats for the representation of binary floating point numbers in memory:

Representation	Total Bits	Sign Bit	Mantissa Bits	Bits
Half Precision	16	1	11 bits	5
SinglePrecision	32	1	24 bits	8
Double Precision	64	1	53 bits	11
Quadruple Precision	128	1	113 bits	15

Representing Zero
As we have seen, when representing numbers in Floating Point and storing them in memory, we write our numbers in a normalized form before dropping the implicit set bit before the radix point. When the numbers in memory are interpreted, the implicit bit is reinstated. This implicit assumption that the bit immediately to the left of the radix point is set to 1 causes problems though. What if we wanted to represent zero?

To get around that problem, the IEEE 754 standard defines zero as a special case and represents it by using an exponent of 0[1] and a mantissa of 0. Due to the sign bit still being available this leads to values of -0 and +0 the standard defines that they must be compared as equal.

[1] To store an exponent of 0, we store the exponent bias in the exponent bits.