Lab 4: Floating Point Multiplier

Introduction

In this lab, you must build hardware to multiply floating point numbers. In Part I, you must design an IEEE-754 32-bit floating point multiplier, which will take two 32-bit floating points numbers as input and produce a 32-bit floating point number as a result. In Part II, you need to modify you multiplier to work with arbitrarily sized floating point numbers. You are allowed to use the built in + and * operators in your hardware for this lab.

IEEE-754 32-bit floating point representation

Figure 1 shows the bit mapping for the IEEE-754 32-bit floating point format, consisting of 1 sign bit (S), 8 exponent bits (E) and 23 mantissa bits (M).

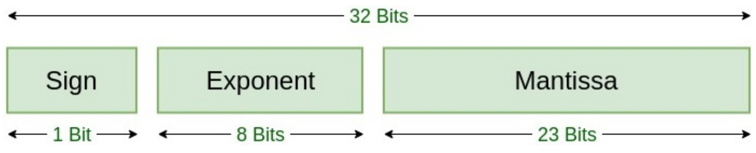

Figure 1: IEEE-754 32-bit floating point representation

The number can be represented in decimal as $(-1)^S \times 2^{E-127} \times (1.M)$. The exponent is stored in 'biased exponent' format, where the entire range of values (i.e., -127 to 128) is stored as an 8-bit number (from 0 to 255). Also, the 1 in the $(1.M)$ term is called a 'hidden one'. It is used during calculations but is not stored along with the number itself.
For example, the 32-bit number 01000000110100000000000000000000 can be converted to decimal as follows. First, we split the number into the 3 sections: $S = 0$; $E = 10000001 = 129_{10}$; $M = 10100000000000000000000$. Plugging these values in the equation above gives us: $(-1)^0 \times 2^{129-127} \times (1.101)$ [1] As the values after the decimal point correspond to inverse powers of two, the bits of the mantissa are: 1×2^{-1}, 0×2^{-2} and 1×2^{-3}. Including the hidden one, this gives us a mantissa value of $1 + 0.5 + 0.125 = 1.625$. The number in decimal is: $2^2 \times 1.625 = 6.5$. With this representation in mind, let's look at the operation of multiplication for floating point numbers.

Part I: 32-bit Floating Point Multiplier

In this Part, you must design a floating point multiplier which will multiply two 32-bit numbers, in IEEE-754 format and produce their product in a single cycle. In addition, you must check for special values (such as inf, `NaN` (i.e., Not a number), overflow and underflow).
Figure 2 shows the multiplication algorithm for two inputs X & Y. The algorithm works as follows:
1. First, XOR the sign bits of X & Y to get the sign of the result.
2. Next, add the exponents of X & Y.

[1] As all the bits after this in the mantissa are 0, we can ignore them.

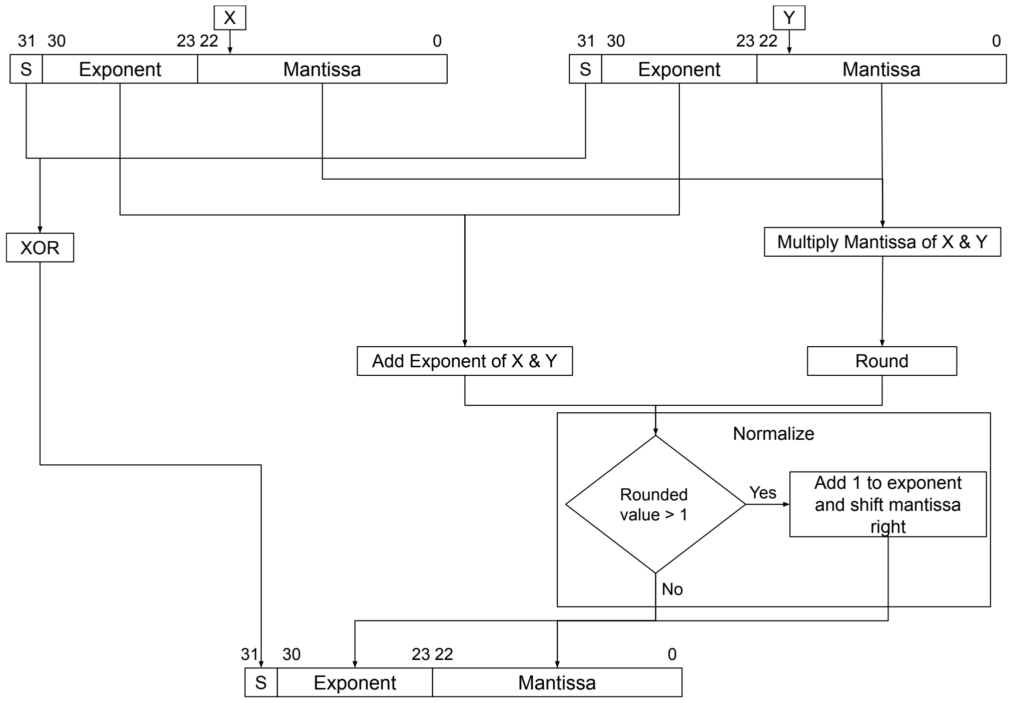

Figure 2: Simple floating point multiplication flow diagram

3. Multiply the (23-bit) mantissas of X & Y. Along with the hidden 1 in each mantissa, this results in a 48-bit product.
4. Round the 48-bit product by truncating the least significant 23-bits. You should be left with a 25-bit 'rounded value' comprising 2 hidden bits and 23 mantissa bits.
5. As the final mantissa should have a 'hidden 1', you must first normalize the rounded value. If this rounded value is greater than 1, shift the value right by 1 bit. Since shifting right by 1 bit is equal to a division by two, this requires the exponent to be increased by 1. In decimal, for example, 23.05×10^3 is the same as 2.305×10^4.
6. As the 2 most significant bits of the mantissa are now normalized, they must be 01 and can therefore be ignored. The 1 is the 'implicit' 1 in the final number.

Example

Let's look at an example. Consider the multiplication of X = -18.0 & Y = 9.5. Note that values within parenthesis () represent 'hidden bits', which are values that are used in calculation but not stored along with the number. In IEEE-754 representation, X and Y are given as:

$$X = 1\ 10000011\ 00100000000000000000000 \qquad Y = 0\ 10000010\ 00110000000000000000000$$

For clarity, we show the sign bit, the exponent bits and the mantissa bit with a space between them. First, we XOR the sign of both numbers to get the sign of the result. In this case, as X is negative and Y is positive, $X \oplus Y$ is negative. Next, we extract the mantissas and add a 1 for normalization to the most significant bit making the mantissa 24-bits instead of 23-bits.

$$X_{mantissa} = 1.00100000000000000000000 \qquad Y_{mantissa} = 1.00110000000000000000000$$

Multiplying these mantissas yields a 48-bit result with 46-bits to the right of the binary point. '...' represents all 0s.

$$X_{mantissa} \times Y_{mantissa} = 01.0101011000000000...000$$

Since the MSB of this product is 0, the mantissa is already in normal form and we do not need to normalize it. Ignoring the first two bits (as we do not store them), we are left with 46 bits after the decimal point.

$$Product_{normalised} = (01).01010110000000000...000$$

Next, we round the value after the binary point by truncating the 23 LSBs, down from 46 bits to 23 bits.

$$Product_{rounded} = (01).01010110000000000000000$$

Lastly, we add the exponents of the two numbers. But keep in mind the exponents are stored with a bias of 127. Thus, when we add both exponents, we are adding 2*127 to the final number. Thus we must subtract 127 to get the correct final exponent.

$$Product_{exponent} = X_{exponent} + Y_{exponent} - 127$$

The exponents for X & Y are 10000011 and 10000010 respectively. Thus, the sum of the exponents is calculated as follows:

$$Product_{exponent} = 131 + 130 - 127 \; Product_{exponent} = 134$$

Since the final exponent also has a bias of 127, the actual exponent is $134 - 127 = 7$. Now, putting together the sign, exponent and mantissa, we get the final product as:

$$Result = 110000110(1)01010110000000000000000$$

This is -171 in binary, which is the product of -18.0×9.5.

IEEE-754 Special Cases

Once your basic multiplier is working correctly, you must add support for special cases such as Zero, Underflow, Overflow and Not-a-Number. We describe each of these below:

- **Zero** is the simplest case to understand and also check for. It is the lowest possible value that can be set in this format. This format has both +0 and -0 so you should check for both.
- **Underflow** means the number is too small to be represented in this format and occurs when the sum of the exponents is less than -126, (i.e, the most negative value which is defined in bias-127 exponent representation). When this occurs, the exponent is set to -127 ($E = 0$) and M is set to any value **other than** 0, to differentiate it from the Zero condition above.
- **Overflow** is used to indicate that the number is too big to be represented in this format. This happens when the sum of the exponents exceeds 127 (i.e., the largest value which is defined in bias-127 exponent representation). If this is the case, the exponent is set to 128 ($E = 255$) and the mantissa is set to zero. This represents ∞ in this format. Note that the sign must be preserved to indicate + or - ∞.
- **Not-a-number** (NaN) occurs when the operation returns a result which cannot be represented in this format. Examples include division by 0, taking the square root of -1 etc.

Special case	**Condition**	**Expected Output**
Zero	E = 0, M = 0	S = 0, E = 0, M = 0
Underflow	E < 0	S = 0, E = 0, M != 0
Not-a-Number	E = EB , M != 0	S = 0, E = EB, M = 0
Overflow	E > EB	

Table 1: Special cases for floating point numbers.

Table 1 lists the special cases to check for and what you should output for each. E stands for exponent value, M stands for mantissa value and EB stands for exponent bias (i.e., $EB = 2^{8-1} - 1 = 127$).

The occurrence of these special cases is indicated by means of dedicated outputs in your module. You must check for these special cases on both the inputs and the outputs of your multiplier and set the result to the expected value accordingly. For example, if either of the inputs is 0, you should set the output to 0 as shown in the Table 1. Similarly, if either of your inputs is ∞, you should set the output as shown for the overflow case. The full module declaration is provided in the code block below.

```
module part1(
input [31:0] X,
input [31:0] Y,
output [31:0] result
output zero, underflow, overflow, nan
);
    // Your code goes here.
endmodule
```

Testing your design

As with previous labs, you must write a test bench to verify your 32-bit floating point multiplier. However, you should think about what cases you need to test. Here are several things to keep in mind when writing your testbench.

1. Unlike previous labs, your inputs are 32-bits in size. Testing every possible input combination would involve $2^6 4$ inputs. To avoid this, you should **think carefully** about cases you need to test. For example, do you have to test combinations of numbers with different signs? What about combinations where one of the mantissas is zero?
2. Similarly, what are the 'corner cases' related to the different outputs you need to test?
3. **Warning:** Be careful when comparing floating point numbers in your testbench. You may see that the numbers you get from your hardware will not exactly match the numbers you generate in your testbench. What do you think could be the cause of this? Given this, you should think about how to check that your hardware is working correctly.

You must be ready to explain to your TA what test cases you picked and justify why you feel that you have sufficiently tested your design. A significant portion of your mark for this lab will be based on the test cases you pick.

Part II: N-bit Floating Point Multiplier

32-bit IEEE-754 floating point values is not the only format of floating point in use today. The IEEE-754 standard also details 'short' 16-bit floating point, as well as 64-bit and even 128-bit formats. There are also non-IEEE formats such as the `bfloat16` format, shown in Figure 3.

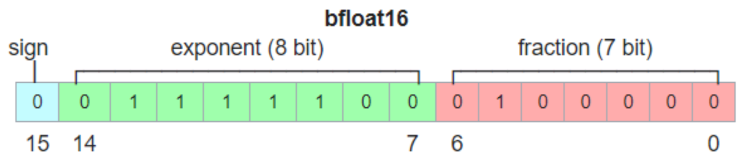

Figure 3: Example of non IEEE-754 FP representation: BFloat16

In Part II you must modify your multiplier from part I to be parameterized so that your hardware can work with any arbitrary floating point format. Your multiplier in Part II must still handle special cases as shown in Table 1. Note that Table 1 applies for any floating point format.

Shown below is the module inputs and outputs for Part II, with the parameters needed to configure your multiplier. The default values for E and M are set to the IEEE-754 standard. You must calculate the

BITS (i.e., the total number of bits in the number) as well as the *EB* (the value to be subtracted from the exponent). You must also set the various 'flags' such as zero, NaN etc. if any of those conditions are true for the inputs provided.

```
module part3#(
  parameter E = 8,           // Number of bits in the Exponent
  parameter M = 23,          // Number of bits in the Mantissa
  parameter BITS = /*TODO*/, // Total number of bits in the floating point number
  parameter EB = /*TODO*/    // Value of the bias, based on the exponent.
)(
  input [BITS - 1:0] X,
  input [BITS - 1:0] Y,
  output [BITS - 1:0] result
  output zero, underflow, overflow, nan
);

endmodule
```

Example

We use Bfloat16 to demonstrate how the multiplier in Part II should work. For Bfloat16, E = 8 and M = 7 (as shown in Figure 3). We once again consider the multiplication of X = -18.0 & Y = 9.5, so you can compare with the example done in Part I. As before, the values within parentheses represent 'hidden bits', which are values that are used in calculation but not stored along with the number. Thus, in BFloat16 representation, these values are:

$$X = 1\ 10000011\ (1)\ 0010000 \qquad Y = 0\ 10000010\ (1)\ 0011000$$

First, we XOR the sign of both numbers to get the sign of the result. Next, we extract the mantissas and add a 1 for normalization to the most significant bit making the mantissa 8-bits instead of 7-bits:

$$X_{mantissa} = 1.0010000 \qquad Y_{mantissa} = 1.0011000$$

Multiplying these mantissas yields a 16-bit result with 14-bits to the right of the binary point:

$$X_{mantissa} \times Y_{mantissa} = 01.010101100000000$$

We see that the MSB of this product is 0 indicating that the mantissa is already in normal form. In this case we just need to truncate the first two bits so that we get the 14 bits after the decimal point.

$$Product_{normalised} = (01).01010110000000$$

Next, we round the value after the binary point, down from 16 bits to 7 bits. Next, we round the value after the binary point by truncating the 7 LSBs, down from 14 bits to 7 bits.

$$Product_{rounded} = (01).0101011$$

Lastly, we add the exponents of the two numbers. Similar to IEEE-754, Bfloat16 uses 8-bits for the exponent so we must subtract 127 to get the correct final exponent. Thus, the procedure to calculate the exponent for Bfloat16 is the same as the one shown in Part II.
This gives us the result of -171 (i.e., -18.0 × 9.5), which in Bfloat16 format is:

$$Result = 1\ 10000110\ (1)\ 0101011$$

The exponent bits are the same for IEEE-754 32-bit and Bfloat16 as they both use an 8-bit exponent. But you must perform the calculations to do this for arbitrary formats.

In-Lab Demo

1. **Part I**: (4 marks) Demo your working IEEE-754 32-bit floating point multiplier. Explain how you picked the right test cases to test your design fully works.
2. **Part II**: (6 marks) Demo your working parameterized floating point multiplier. You must show that you design works for a **few** (non IEEE-754 32-bit) floating point formats. Exactly how many is up-to you. You must justify the number of formats, why you picked those and the specific test inputs for each format. Lastly, you must also explain how you verified that you are getting the right numerical values in your testbench.

NOTE: If your Part II is working, you can use your Part II code to demo Part I as well, by configuring the parameters for IEEE-754 32-bit. However, this will not count as one of the formats for the Part II demo.