Lab 3: Floating Point Muliplier

Introduction

In this lab, you will build a floating point multiplier. In Part I, you will design a simplified 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. The multiplier in Part I will not have to deal with special cases (e.g., NaN, Overflow, Underflow etc.) In Part II, you will extend the multiplier from Part I to support these special cases. Lastly, in Part III, you will modify you multiplier to work with arbitrarily sized floating point numbers.

Part I: Simple Floating Point Multiplier

The floating point multiplication algorithm you will use for part I will be a simplified version of the one covered in lecture. You will not have to worry about special cases such as infinity, not a number (i.e., NaN), overflow and underflow for this Part. Your circuit must take two IEEE 754 single precision 32-bit floating point numbers and produce their product as a 32-bit floating point result in a single cycle.

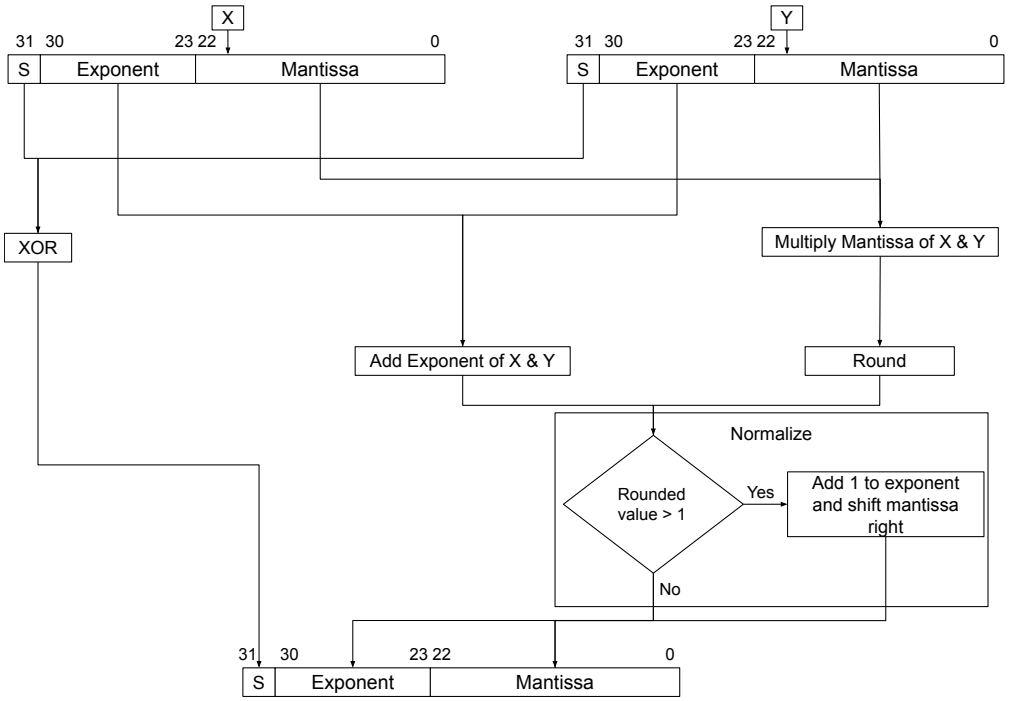

Figure 1: Simple floating point multiplication flow diagram

Figure 1 shows the multiplication algorithm for two inputs X & Y. The algorithm works as follows:

- First, XOR the sign bits of X & Y to get the sign of the result.

- Next, add the exponents of X & Y.
- Multiply the (23-bit) mantissas of X & Y. Along with the hidden 1 in each mantissa, this results in a 48-bit product.
- Round the 48-bit product by truncating the least significant 23-bits. You should be left with a 25-bit number; 2 hidden bits and 23 mantissa bits.
- Normalize the mantissa by shifting it right and adding one to the exponent if the result is greater 1. Note that shifting right is equal to a division by two. This means the exponent must be increased by 1. For example, 23.05 x 10^3 $->$ 2.305 x 10^4.
- Truncate the 2 most significant bits of the result mantissa as they are hidden bits.

Example

We illustrate this process by means of an example. Note that values within parenthesis () represent 'hidden bits', which are values that are used in calculation but not stored along with the number. Consider the multiplication of X = -18.0 & Y = 9.5. In IEEE-754 representation, these values are:

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

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 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. (The series of ... represent all 0s.)

$$X_{mantissa} \times Y_{mantissa} = (01).0101011000000000...000$$

We see that the MSB of this product is 0. This means that the mantissa is already in normal form. However, if the MSB had been 1, we would then need to shift the mantissa right and increment the exponent by 1 (as described above). But we don't need to do that in this case.

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

Next, we round the value after the binary point, down from 48 bits to 25 bits.

$$Product_{rounded} = (01).0101011000000000000000000$$

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.

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

Shown in binary, the sum of the exponents is calculated as follows:

1000 0011 (131 = 127 + 4)
+ 1000 0010 (130 = 127 + 3)

0000 0101 (7)
+ 1000 0001 (-127)

1000 0110 (134 = 127 + 7)

This gives us the result of -171 (i.e., -18.0 × 9.5), which in IEEE-754 FP format is. Once again, the hidden bit is shown for clarity but is not part of the 32-bit number itself.

$$Result = 1\ 10000110\ (1)\ 01010110000000000000000$$

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

Table 1: List of features for floating point numbers, condition of the features and the expected output

Part II: IEEE-754 Special Cases

In this part, you will extend your algorithm to cover special cases such as infinity, not a number (NaN), zero, overflow and underflow. Table 1 lists special cases to check for and set flags for them accordingly. In Table 1, E stands for exponent value, M stands for mantissa value and EB stands for exponent bias. For the IEEE-754 32-bit representation, E = 8 and EB = 2^8 - 1 = 255. Similarly, for this format:

- Overflow occurs when the sum of the exponents exceeds 127 (i.e., the largest value which is defined in bias-127 exponent representation). When this occurs, the exponent is set to 128 (i.e, the E field is set to 255) and the mantissa is set to zero (indicating + or - infinity).
- Underflow 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). If M = 0, the number is exactly zero. **NOTE:** Underflow occurs when subtracting the bias.

In your code for part II, the occurrence of these special cases is indicated by means of dedicated outputs. You must check for these features 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.

Part III: N-bit Floating Point Mulitplier

Thus far, your design has needed to work with a standard 32-bit IEEE-754 floating point format. However, this is not the only format of floating point in use today. The IEEE-754 standard also details 'short' 16-bit FP values, as well as 64-bit and 128-bit values as well. There are also non-IEEE formats such as the `bfloat16` format, shown in Figure 2.

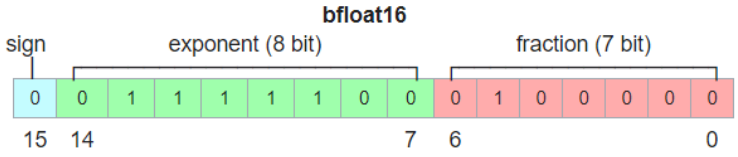

Figure 2: Example of non IEEE-754 FP representation: BFloat16

In part III you must extend your multiplier from part II to be parameterized so that your hardware can work with any arbitrary size of floating point number. Your multiplier in Part III must still handle special cases as shown in Table 1.

Shown below is the module inputs and outputs for Part III, with the parameters needed to configure your multiplier. You can set these parameters via the testbench as shown in the 'Creating generic hardware' Tutorial on Quercus. The default values for EXP and MAN 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 BIAS (the value to be subtracted from the exponent). You must also set the various 'flags' such as inf, NaN etc. if any of those conditions are true for the inputs provided.

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

endmodule
```

Example

To demonstrate how Part III should work, we illustrate with an example of multiplying two Bfloat16 values. For Bfloat16, EXP = 8 and MAN = 7 (as shown in Figure 2). Once again, note that values within parenthesis () represent 'hidden bits', which are values that are used in calculation but not stored along with the number. We once again consider the multiplication of X = -18.0 & Y = 9.5, but this time using the Bfloat16 format. 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.

$$Product_{normalised} = 01.01010110000000$$

Next, we round the value after the binary point, down from 16 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.

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

The sum of the exponents is:

  1000 0011 (131 = 127 + 4)
+ 1000 0010 (130 = 127 + 3)

  0000 0101 (7)
+ 1000 0001 (-127)

  1000 0110 (134 = 127 + 7)

This gives us the result of -171 (i.e., -18.0 × 9.5), which in Bfloat16 format is:

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

Preparation

- Familiarize yourself with the floating point number representation, particularly the IEEE-754 format. You can do so by referring to *ieee754-1985-STANDARD_cleanedup.pdf* document on Quercus
- Name the finished code part1.sv, part2.sv and part3.sv respectively.

Submission

Your should submit part1.sv, part2.sv and part3.sv using the instructions provided in the submission document.