Project Report on

**IMPLEMENTATION OF BFLOAT16 ARITHMETIC ON FPGA**

Submitted By

Matru Prasad Mohanty, 23EC01056

Swastik Mantri, 23EC01045



School of Electrical and Computer Sciences

Indian Institute of Technology Bhubaneswar

**CONTENTS:**

1. Introduction
   1. What are Floating Point Numbers?
   2. What is the Bfloat16 representation?
   3. Significance of Bfloat16 numbers
2. Bfloat16 Arithmetic
   1. Bfloat16 Addition and Subtraction
   2. Bfloat16 Multiplication
3. Conclusion
4. References
5. INTRODUCTION
   1. What are Floating Point Numbers?

Floating Point Numbers are a way to represent real numbers, including decimal and integers, within computer system. They can represent very small as well as very large values. Fixed point numbers represent numbers with a fixed number of digits before and after the decimal point but floating-point numbers allow for a dynamic range by expressing numbers in scientific range.

A floating-point number is represented in the form:

Value= (-1)sign x 1.mantissa x 2exponent

Here, Sign: indicates the sign of the number (0 for positive, 1 for negative)

Mantissa: represents the precision

Exponent: represents the range of the number

The most common standard for floating-point arithmetic is **IEEE-754**, which defines various precisions such as 32-bit (single), 64-bit (double), and more recently, 16-bit formats for reduced-precision tasks.

* 1. What is the Bfloat16 representation?

The bfloat16 (brain floating point) floating point format is a computer number format occupying 16 bits in the computer memory and it represents a wide dynamic range of numeric values by using a floating radix point. This format is a shortened (16-bit) version of the 32-bit IEEE 754 single precision floating point representation with the intent of accelerating machine learning and near sensor computing.

Bfloat16 has the following format:

Sign bit: 1 bit

Exponent width: 8 bits

Significand precision:  8 bits (7 explicitly stored, with an implicit leading bit)

The bfloat16 binary floating-point exponent is encoded using an offset-binary representation, with the zero offset being 127; also known as exponent bias in the IEEE 754 standard.

* Emin = 01H−7FH = −126
* Emax = FEH−7FH = 127
* Exponent bias = 7FH = 127
  1. Significance of Bfloat16 numbers
* Bfloat16 is used for faster training and inference in AI/ML related tasks since many deep learning models are tolerant to lower precision.
* It has 8 exponent bits so it avoids issues like overflow or underflow seen in FP16.
* Supported by hardware and provide hardware acceleration.

1. **BFLOAT 16 ARITHMETIC**

We have performed various arithmetic operations on floating point numbers in the Bfloat16 format like addition, subtraction and multiplication.

Let the 2 Bfloat16 numbers X and Y be

X= (-1)sign1.m1.Be1

Y= (-1)sign2.m2.Be2

Here,

sign1 and sign2 represent the signs of the 2 numbers

m1 and m2 represent the mantissa of the 2 numbers (including the implicit 1)

e1 and e2 represent the exponent of the 2 numbers

Using an unpack module, we unpack the provided 16 bit bfloat numbers into the sign, exponent and mantissa.

* 1. **Bfloat16 Addition and Subtraction**

For performing the addition and subtraction operation on the 2 numbers X and Y, we have implemented the following algorithm:

* + 1. Defining the operation:

The algorithm for performing addition and subtraction have many similar steps so we have performed addition and subtraction using one module in our Verilog code. The addition and subtraction operations have been defined as ‘0’ and ‘1’ respectively in our code.

* + 1. Sign Handling:

If the operation was ‘1’, that is subtraction, then we flipped the second operand. This step we help us in doing the subtraction operation as addition of negative number.

* + 1. Exponent comparison and swapping the operands:

The exponents must match for addition or subtraction. We will always keep the larger exponent in e1. If e2>e1, then we swap the operands.

* + 1. Alignment of the Mantissas:

Here we right shift the mantissa of the operand with smaller exponent to match the exponent of the other operand.

* + 1. Operation based on the signs:

Since, earlier we have changed the sign of the operands based on the operations, now we perform the operation as per the signs. We take the xor of the signs.

If the xor is giving zero, that is, the sign of both the operands is same, then we perform the addition of the aligned mantissas.

After addition if overflow occurs, then we will perform the normalization operation.

If the xor is giving one and the sign of both the operands is different, then we perform subtraction of the mantissas. To ensure that the subtraction result is non negative, we compare the magnitudes m1 and m2. If m1<m2, then we swap them and flip the sign of the result.

* + 1. Normalization:

This step is performed to ensure that there is no overflow and to ensure that the mantissa begins with the implicit one. After addition, if there is an overflow, then we normalize by right shifting by one place and increasing the exponent by one.

After subtraction, the result may be with leading zeros. But the mantissa must start with the implicit one, so we use a loop to find out the leading one from left and count the number of initial zeros. For ‘k’ leading zeros the mantissa is left shifted by k bits and the exponent is decreased accordingly.

* + 1. Rounding operation using the PGRS bits:

Initially we have taken a larger mantissa range so that the above operations can be performed easily without the loss of precision. Now we round the mantissa to size of 8. Here we perform round to the nearest even.

Rules for rounding:

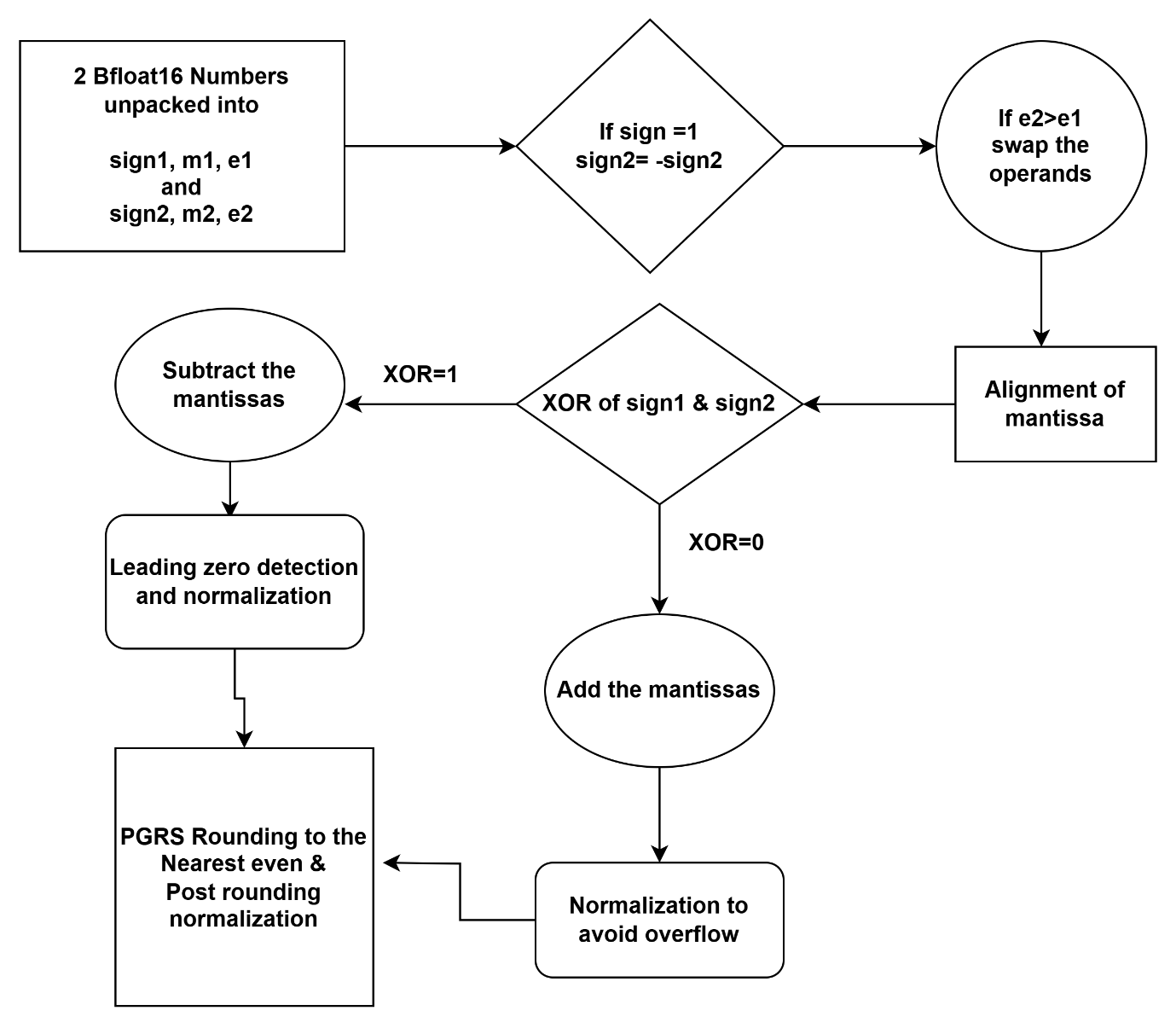
* The last bit in the 8-bit mantissa (including the initial one) is the pre bit. The next bit is the guard bit, after that the next one is the round bit and sticky bits are all the bits afterwards. We take the ‘or’ of all the sticky bits.
* If the guard bit is set, then we check for 2 more cases:
* Either of the round bit or the sticky bits is set
* The p bit is set

In both the above cases, we add a one to the mantissa. The one is added at the p bit position. If the p bit is set, it means that the mantissa is odd, so we add a one to convert it to the nearest even.

After rounding, there is chance that an overflow may have occurred due to the addition of one, so we perform one more step of normalization in order to fit the mantissa.

* + 1. Zero result handling:

If the mantissa becomes too small after normalization (i.e., upper bits are zero), we set the exponent to 0 to indicate a zero result.



Pipeline for Addition and Subtraction

* 1. **BFloat-16 Multiplication**
     + 1. Defining the operation

In our code, the multiplication module is called when the value of the operation “op” is 2.

* + - 1. Sign Handling:

Sign Handling in multiplication is comparatively a lot easier. The multiplication of two numbers gives a negative sign if either of a and b are negative and gives positive sign when either both are negative or both are positive. This corresponds to just assigning the sign of the output of their product as the XOR of the signs of the two numbers a and b. If only either of a or b is 1 then only will the sign be considered as negative(If sign bit is set, number is considered negative).

* + - 1. Exponent handling

The exponent of two numbers is simply added when they are multiplied. A finer detail to note is that in the code we are subtracting 127 after adding the two exponents: the reason for this is biasing. When we store the BFloat numbers we store the exponents in the range 0-255 which is biased by subtracting 127 when displaying the number to show the actual exponent. Thus, each exponent that is stored in the BFloat format is actually 127 greater than its actual value. Due to this, we need to remove an extra 127 by subtracting it to ensure the correctness of the result.

* + - 1. Mantissa Handling

Two 8 bit numbers when multiplied results in a number that requires 16 bits for storing. Thus, we simply multiply the two mantissa and store it in a 16 bit number.

* + - 1. Normalization

When we multiply two 8 bit numbers, the result is a 16 bit number. Now, initially we have 7 bits each after the decimal point in both the normalized inputs. This will lead to a result that will have 14 bits after the decimal point and may result in a value greater than or equal to 2. Thus, a normalization step will be required which checks if the mantissa is greater than 2 (by checking if the 15th bit is set) then left shifts the mantissa by 1 and increments the exponent by 1 leading to normalized result.

* + - 1. Rounding operation using the PGRS bits:

Initially we have taken a larger mantissa range so that the above operations can be performed easily without the loss of precision. Now we round the mantissa to size of 8. Here we perform round to the nearest even.

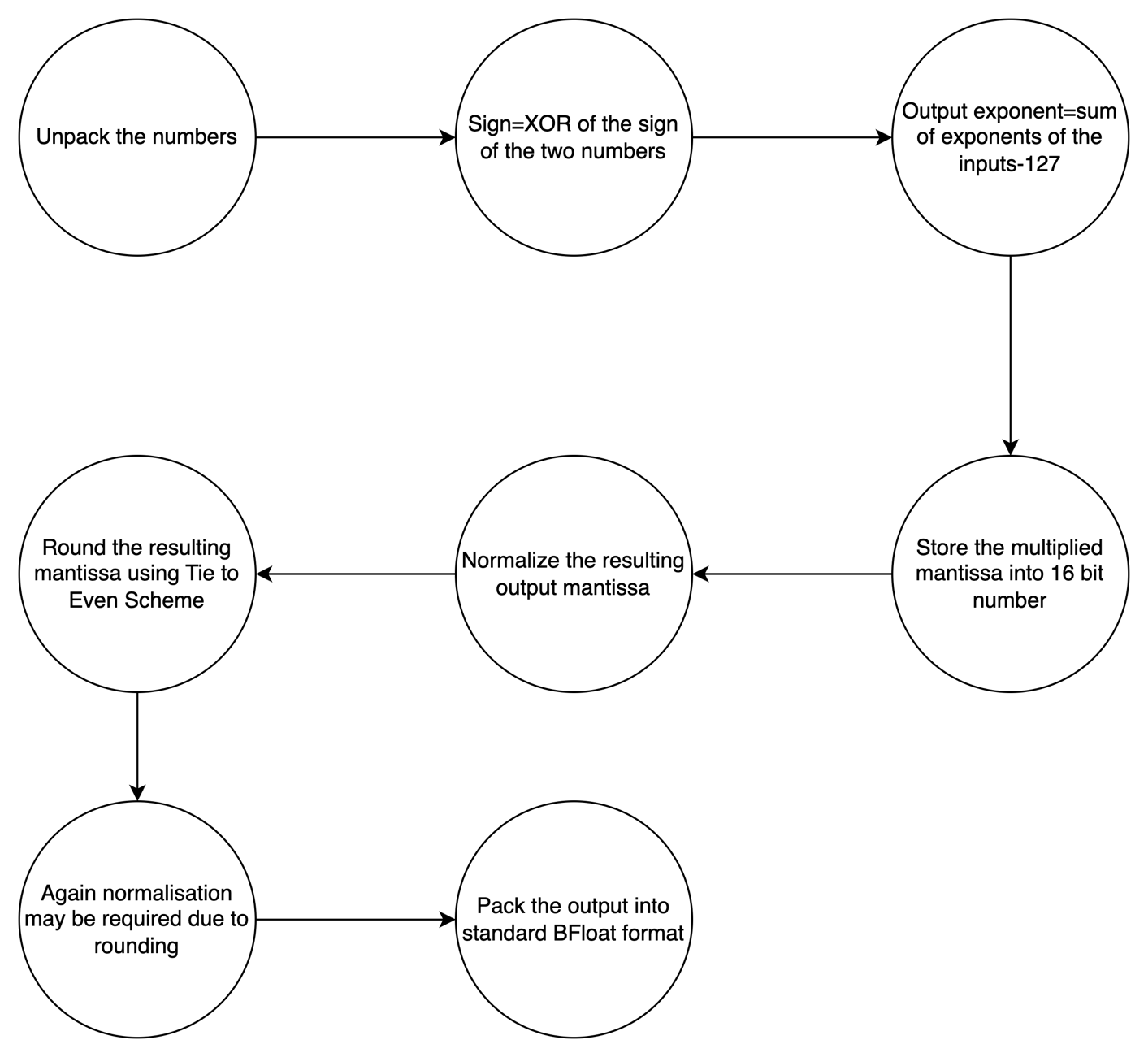
Rules for rounding:

* + The last bit in the 8-bit mantissa (including the initial one) is the pre bit. The next bit is the guard bit, after that the next one is the round bit and sticky bits are all the bits afterwards. We take the ‘or’ of all the sticky bits.
  + If the guard bit is set, then we check for 2 more cases:
  + Either of the round bit or the sticky bits is set
  + The p bit is set

In both the above cases, we add a one to the mantissa. The one is added at the p bit position. If the p bit is set, it means that the mantissa is odd, so we add a one to convert it to the nearest even.

After rounding, there is chance that an overflow may have occurred due to the addition of one, so we perform one more step of normalization in order to fit the mantissa.

An important thing to note is that no leading zero detection is required as both inputs are normalized and cannot result in a number less than 1.



Pipeline for BFloat Multiplication

1. **Conclusion**

The implementation of BFloat16 arithmetic on FPGA highlights the balance between hardware efficiency and numerical performance. Through careful handling of unpacking, normalization, and rounding—especially using the PGRS method—we were able to achieve accurate and consistent arithmetic operations including addition, subtraction, and multiplication. The simplicity of BFloat16, due to its reduced bit-width and retained exponent range, makes it well-suited for AI and ML workloads on hardware platforms. This project not only deepened our understanding of floating-point computation but also provided hands-on experience in Verilog-based hardware design, aligning theory with practical application.

1. **References**

1)Book: Synthesis of Arithmetic Circuits FPGA, ASIC and embedded systems by Jean-Pierre Deschamps, Géry Jean Antoine Bioul and Gustavo D. Sutter

2) <https://www.youtube.com/watch?v=bbkcEiUjehk>

3) <https://youtu.be/yvdtwKF87Ts?si=b7ocv7n7StnCpF2P>

4) <https://youtu.be/4VU7ctvMRfs?si=geJq7P1626CmglEb>

**GitHub repository link:** <https://github.com/matru777/Bfloat16_fpga_project>