DESIGN OF SINGLE PRECISION FLOAT ADDER (32-BIT NUMBERS) ACCORDING TO IEEE 754 STANDARD USING VERILOG

By
SUMAN MAHATO

Objective :

Design a floating point adder that takes two 32 bit single precision floating point input values that come serially with a time difference of 8 clock cycles between two inputs and stores the resultant value into eight RAMs. Perform the addition operation for 8 sets of input values.

32 bits Floating Point Adder Design :

The main goal of this project is the implementation of a 32bit Floating Point Adder with VHDL code. The format and the main features of the standard have been described before but nothing about the steps to achieve the target has been said. The first logical step is trying to specify what operations should be done to obtain a proper addition or subtraction. Once the idea will be clear the block diagram of the entire code will be designed.

- Addition/Subtraction Steps :

Following the established plan, the way to do the operations (addition/subtraction) will be set. This point will be also used to try to explain why these steps are necessary in order to make clearer and easier the explanation of the code in the next section. The different steps are as follows:

1. Extracting signs, exponents and mantissas of both A and B numbers. As it has been said, the numbers format is as follows:

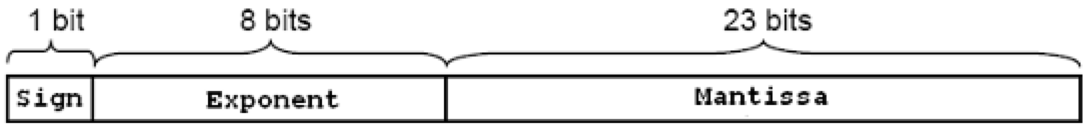


Fig: Floating Point Number format

Then the first step is finding these values.
2. Treating the special cases
• Operations with A or B equal to zero

- Operations with ±∞
- Operations with NaN

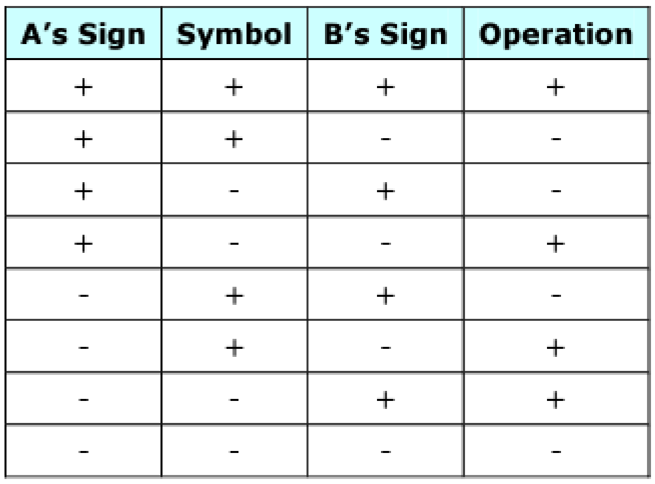
3. Finding out what type of numbers are given:
- Normal
- Subnormal
- Mixed

4. Shifting the lower exponent number mantissa to the right [Exp − 1 Exp 2] bits. Setting the output exponent as the highest exponent.
A’s Exponent → 3   B’s Exponent →.-1   Difference (A-B)--> 4

Number B: 1 1 0 1 0 0 1   —->   0 0 0 0 1 1 0 1 0 0 1

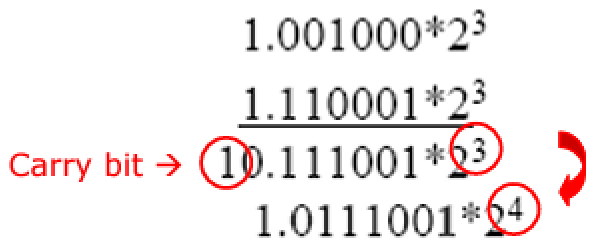
5. Working with the operation symbol and both signs to calculate the output sign and determine the operation to do.

Table 1. *Sign Operation*



A’s Sign	Symbol	B’s Sign	Operation
+	+	+	+
+	+	-	-
+	-	+	-
+	-	-	+
-	+	+	-
-	+	-	+
-	-	+	+
-	-	-	-

6. Addition/Subtraction of the numbers and detection of mantissa overflow (carry bit)



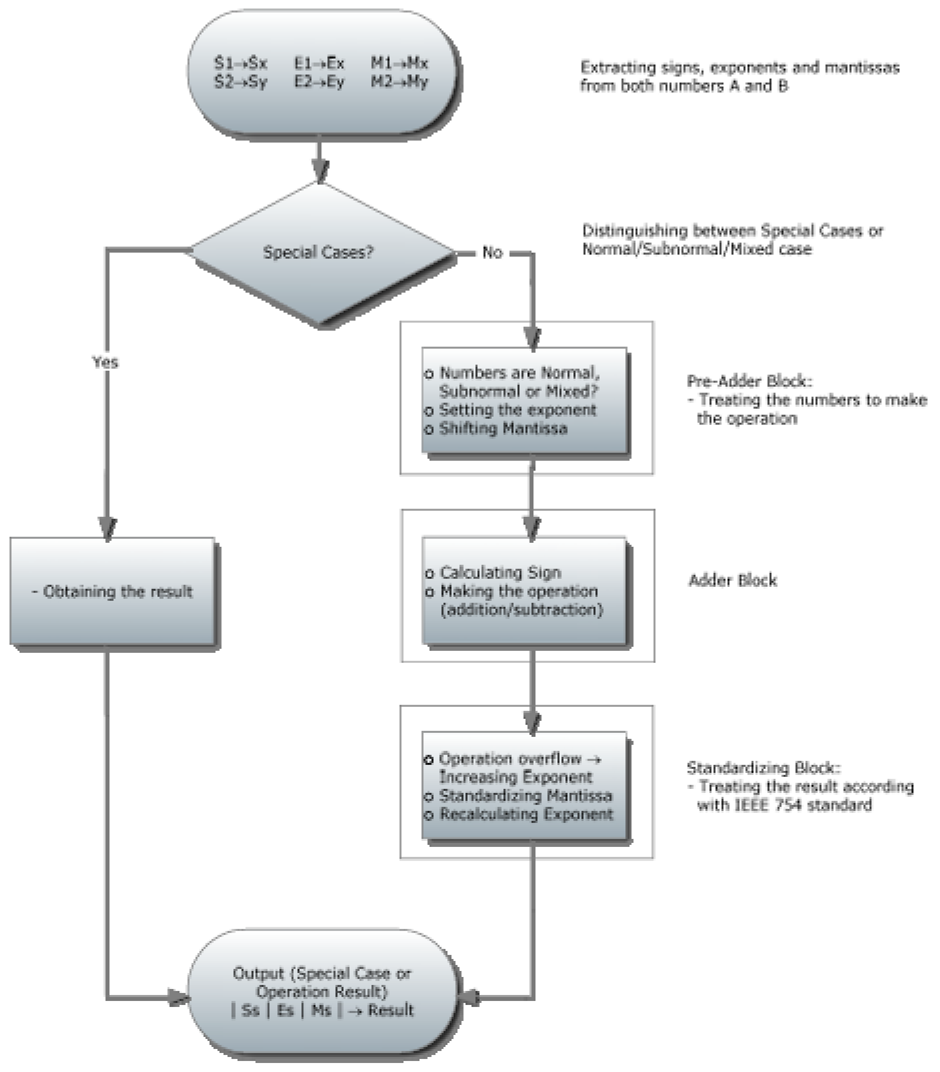
7. Standardizing mantissa shifting it to the left up the first one will be at the first position and updating the value of the exponent according with the carry bit and the shifting over the mantissa.

0.1010101 *2^3 → 1.010101* 2^2

8. Detecting exponent overflow or underflow (result NaN or ±∞) This is the way forward to proper operation. Obviously there are some parts which have to be discussed because there will be more aspects to be taken into account but this will happen in next sections where the code will be explained.

- Block Diagram

The main idea has been described before. Once the different steps to follow have been explained it is time to start to think in the code implementation. In this subsection a first block diagram –as a draft will be made. It still does not go into the most difficult points because in the next section, once a division of the project in three parts will be done, a complete description of each step will be performed. These three parts are as follows: • PreAdder Block • Adder Block • Standardizing Block They make reference to the three main processes of the project. First the numbers should be treated (pre-adder) in order to perform the operation properly (adder) and finally, standardizing the result according with the standard IEEE 754 (standardizing). In figure 6, a first approximation of the design has been done:


S1→Sx E1→Ex M1→Mx
S2→Sy E2→Ey M2→My
Extracting signs, exponents and mantissas from both numbers A and B
Special Cases?
No
Yes
Distinguishing between Special Cases or Normal/Subnormal/Mixed case
o Numbers are Normal, Subnormal or Mixed?
o Setting the exponent
o Shifting Mantissa
Pre-Adder Block:
- Treating the numbers to make the operation
- Obtaining the result
o Calculating Sign
o Making the operation (addition/subtraction)
Adder Block
o Operation overflow → Increasing Exponent
o Standardizing Mantissa
o Recalculating Exponent
Standardizing Block:
- Treating the result according with IEEE 754 standard
Output (Special Case or Operation Result)
| Ss | Es | Ms | → Result

Brief Algorithm Description

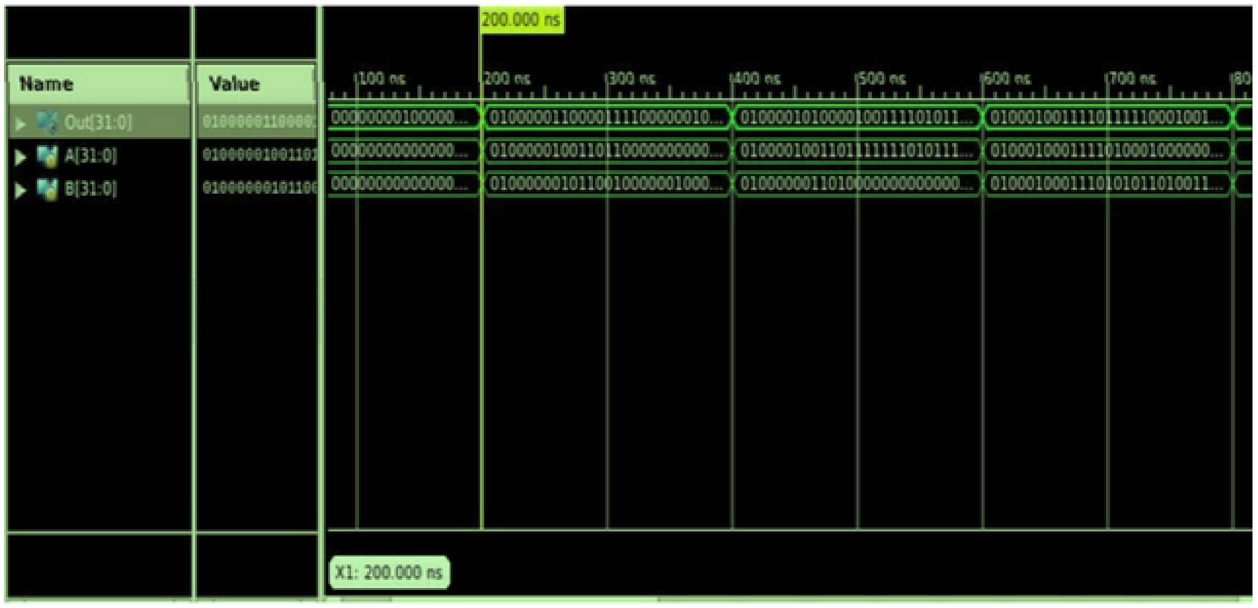
- We need to add inputs A and B to form Out. Out=A+B
- Here, A and B need to have the same exponents if they are to be added i.e EA=EB.
- EA and EB are fed to the 8 bit subtractor and the borrow if 0 shows A>B and B has to be right shifted.
- The exponent difference (EA-EB) provides us with a measure of the shift required in the lower exponent input.

- Right shift the mantissa of the lower exponent input by the exponent Difference. Thus, both the inputs have the same exponent as a result.
- The final mantissa is taken to be maximum of the exponents of the input and is further increment depending on the carry of the adder.
- Depending on the carry output of the adder, the final mantissa is right shifted by 1 or 0;
- The final exponent and mantissa are thus calculated.

Directory Description

- Codes directory contains the required modules of 32-bit floating point adder along with the test bench.
    - Modules:
        - FloatingPointAdder.v - Main Module
        - Adder_24Bit.v
        - BarrelShifter.v
        - Complement2s.v
        - ControlledIncrementor.v
        - FullAdder.v
        - FullSubtractor.v
        - HalfAdder.v
        - HalfSubtractor.v
        - Mux.v
        - Mux24.v
        - Mux_8.v
        - Subtractor_8Bit.v
        - Sub_Result.v
        - FIFO.v
    - Test Bench:
        - BC.v

Simulation :



```
Name        Value              200.000 ns
Out[31:0]   0100000110000      00000000100000...  01000001100001111000000010...  01000010100001001111101011...  01000100111101111110001001...
A[31:0]     0100000100110      00000000000000...  01000001001101100000000000...  01000010011011111111010111...  01000100011111010001000000...
B[31:0]     0100000001011      00000000000000...  01000000101100100000001000...  01000000110100000000000000...  01000100011101010111010011...
X1: 200.000 ns
```