University of Manouba
NATIONAL SCHOOL OF COMPUTER SCIENCE OF MANOUBA

Report Sheet

Floating Point Arithmetic on FPGA

Realised by :
THABET Khalil
TLILI Hani

Professeur :
Mme. Kriaa Lobna

Summary

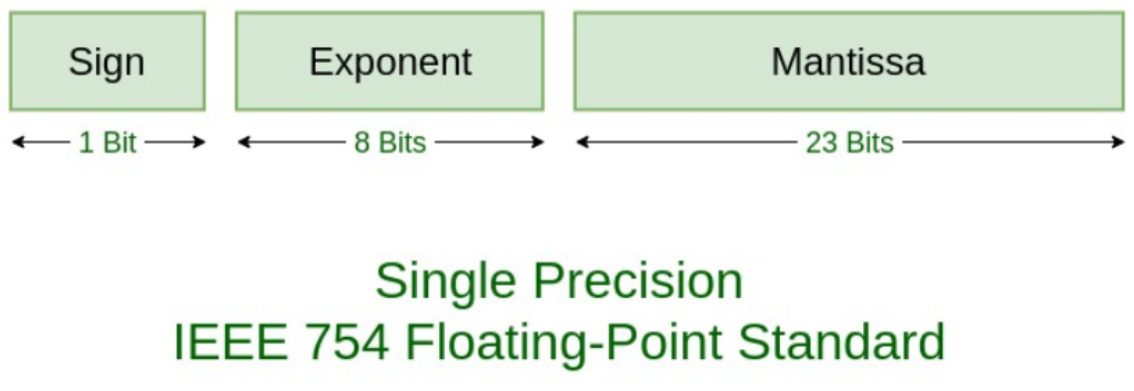
I - Introduction :

Floating point numbers are a way to represent real numbers in binary form. They are commonly used in scientific and engineering applications where a wide range of values and precision is required. They are represented by three fields: the sign, the exponent, and the mantissa. The exponent field represents the power of 2 by which the mantissa should be multiplied to give the final value of the number. The mantissa field contains the significant digits of the number. This representation allows for a much larger range of values than fixed-point representation, but it also means that floating point numbers have a limited precision. Arithmetic operations with floating point numbers are usually implemented using specialized hardware or software libraries that follow the IEEE 754 standard.

II - IEEE 754 Floating-Point Standard:

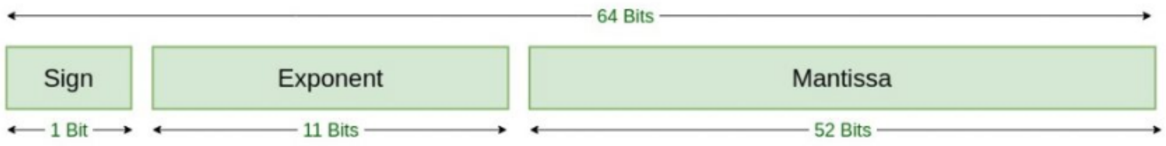
The IEEE 754 standard is a widely-used standard for representing floating point numbers in computers. It defines the format, precision, and behavior of floating point numbers in a standardized way.

According to the IEEE 754 standard, a binary floating-point number is represented by three fields: a sign field, an exponent field, and a significand (mantissa) field. The sign field is one bit, the exponent field is typically 8, 11 or 15 bits, and the significand field is typically 23, 52 or 113 bits. The overall format of a binary floating-point number is as follows: The sign bit is represented by a 0 for positive numbers and 1 for negative numbers. The exponent field represents the power of 2 by which the significand should be multiplied to give the final value of the number. The exponent is stored as an excess-n format where the actual exponent value is obtained by subtracting a bias value (usually 2^(n-1) -1, where n is the number of bits in the exponent field). The significand field contains the significant digits of the number. The significand is stored in a normalized form, where the most significant bit is always 1 and is not stored explicitly, this is called hidden bit. For example, a single-precision (32-bit) floating point number according to IEEE 754 standard has: 1-bit for sign 8-bit for exponent 23-bit for significand This representation allows for a wide range of values and a high level of precision, but it also means that floating point numbers have a limited precision. Additionally, the IEEE 754 standard also defines special values such as infinity, NaN (not a number) and denormalized numbers, which are used to handle exceptional cases such as overflow and underflow.



Ref : [IEEE Standard 754 Floating Point Numbers - GeeksforGeeks]

—---



Double Precision
IEEE 754 Floating-Point Standard

ref : IEEE Standard 754 Floating Point Numbers - GeeksforGeeks

—---

Example :

- 85.125
    - 85 = 1010101
    - 0.125 = ⅛ = 1 * 2^3 = 0.001
    - 85.125 = 1010101.001 =1.010101001 x 2^6
    - sign = 0
    - 1. Single precision: biased exponent 127+6=133
        - 133 = 10000101
        - Normalized mantissa = 010101001
        - we add 0's to complete the 23 bits
        - The IEEE 754 Single precision is:
            - 0 10000101 01010100100000000000000

III - FPU Arithmetic Operations:

a- Addition :

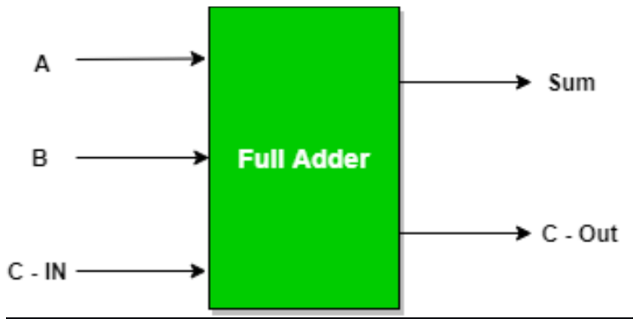
When adding two floating-point numbers following the 754 standard, the process can be broken down into several steps:

- Align the exponents: The exponents of the two numbers are compared and the number with the smaller exponent is shifted to the right, so that the exponents match. This is done to ensure that the mantissas (the fractional part of the number) are aligned before adding them together.
- Add the mantissas: The mantissas of the two numbers are added together. This is done by aligning the most significant bit of each mantissa, adding the bits, and carrying over any overflow.
- Normalize the result: After adding the mantissas, the result may not be in the correct format. The result must be normalized, which means adjusting the exponent and mantissa to ensure that the most significant bit of the mantissa is 1 and the exponent is within the specified range.
- Handle rounding: The result may be too large to fit in the 32-bit format. In this case, the least significant bits will be rounded off.

- Handle overflow and underflow: The result may be too large or too small to represent in the 32-bit format. In this case, the result will be set to the maximum or minimum representable.

  1 - Full adder:

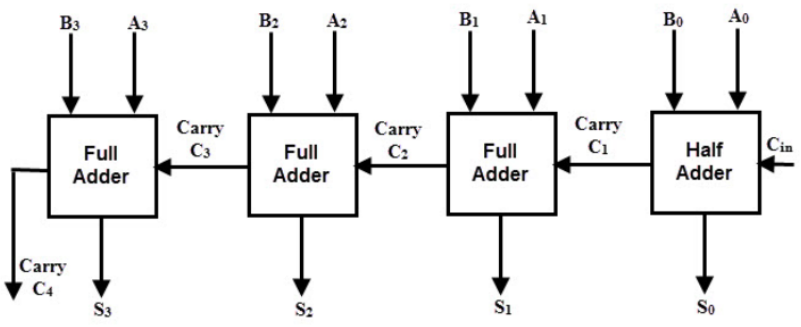
  Full Adder is the adder that adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM.



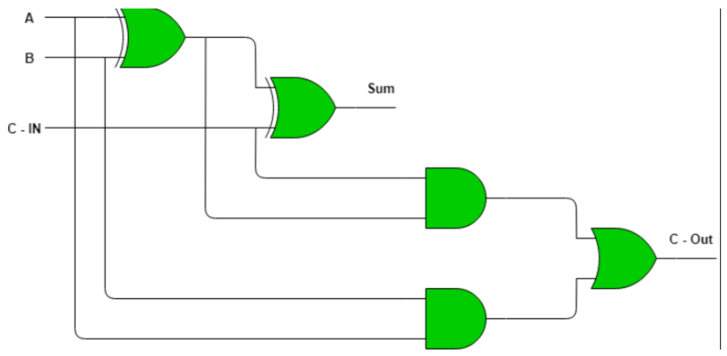
Full adder representation

Ref : [Full Adder in Digital Logic - GeeksforGeeks]

A full adder logic is designed in such a manner that can take eight inputs together to create a byte-wide adder and cascade the carry bit from one adder to another.



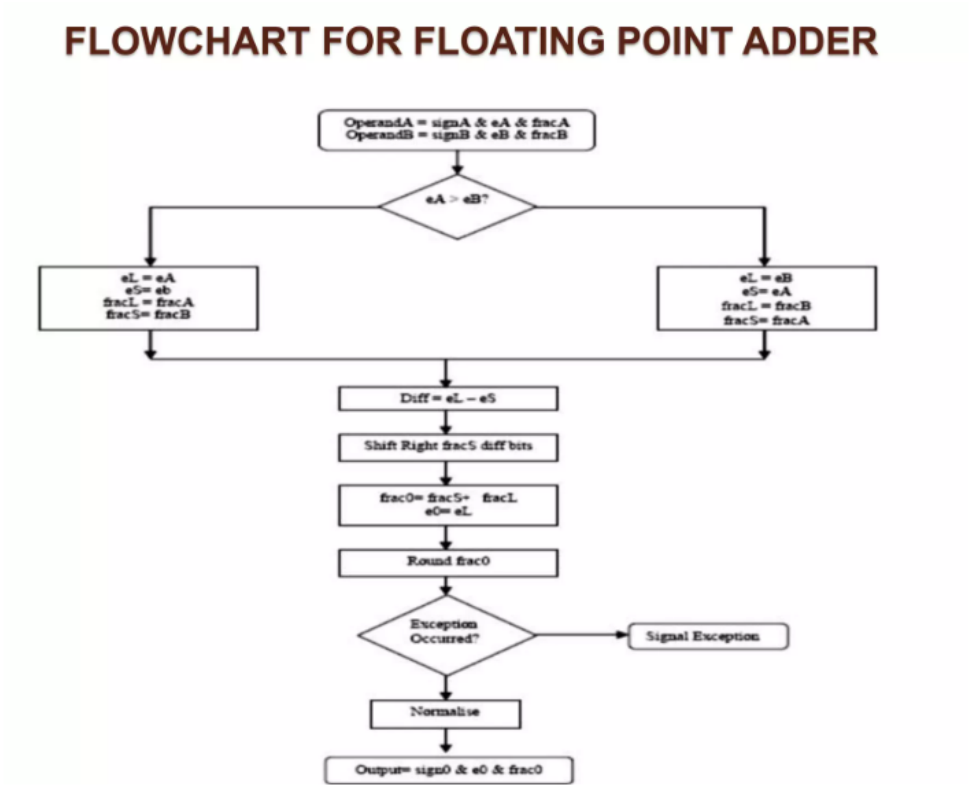
So to implement the full adder we represented the following logic circuit by :



Full Adder Logic Circuit

Ref : [Full Adder in Digital Logic - GeeksforGeeks]

- ➔ **Sum <= A XOR B XOR C-IN**
- ➔ **C-Out <= (A AND B) OR (C-IN AND (A XOR B))**



Ref : Surendra Bommavarapu - SlideShare

Interpretation :
To explain in depth the previous flowchart, we suppose we have two float numbers A and B. Then to represent these numbers in binary format we resort to the IEEE 754 single precision floating point standard.
OperandA will be interpreted by (signA & expA & fractionA)
OperandB will be interpreted by (signB & expB & fractionB)

1. At a first step we compare expA and expB
2. The number with the lowest exponent will be shifted to the right by | expA - expB |
3. We then add the two fractions and round up the result
4. Normalize the result
5. Output the result (sign0 & exp0 & fraction0)

Adding two floating points General principle:
Suppose we have :

X = 0100 0010 0000 1111 0000 0000 0000 0000
Y = 0100 0001 1010 0100 0000 0000 0000 0000

Sx = 0
expX = 10000100 = 132 , 132-127=5
X = 1.0001111 * 2^5
	Sy = 0
	expY = 10000011 = 131, 131 - 127 = 4
	Y = 1.01001 * 2^4 -> Y = 0.101001 * 2^5

Addition :

```
1.0001111   *   2^5
0.1010010  *   2^5
—-----------
1.1100001   *   2^5

5*127 => 132 = (10000100)base 2
result  = 0 10000100 1100001000000
```

We went on applying a 8 stage pipeline to calculate the sum of 2 floating points

- First Stage : Negating the expB
- Second Stage : Calculating the difference expA - expB
- Third Stage : Identifying the biggest and lowest exponents
- Fourth Stage : Shifting the small fraction (with small exponent) with the exponent difference.
- Fifth Stage : Adding the fractions
- Sixth Stage : Storage the max_exponent value in the exponent result
- Seventh Stage and eighth stage: correcting exponent result
-

b - Binary Subtraction Using 2's Complement:

Binary subtraction of numbers can be done by adding the 2's complement of the second number to the first number. Binary subtraction is just the binary addition of a negative number.

<u>1’s complement</u> is a system where negative numbers are represented with the inverse of their binary representation of the corresponding positive numbers. It enables negative numbers to be denoted in binary form.

What we are doing here, is essentially “flipping” the roles of 0 and 1. Hence instead of starting with 0000 as we normally do, we start from 1111.

Therefore, negative numbers can be represented like this:

(1111)2 is (−1)10

(1110)2 is (−2)10

(1101)2 is (−3)10

And so on.
For example, the 1’s complement of (100101)2 is (011010)2.

<u>2’s complement</u> of a number can be found by adding a 1 to the 1’s complement of a number.

From the example above, we find the 1’s complement of (100101)2 to be (011010)2.

<u>Binary Subtraction:</u>

In computers, subtraction of numbers is done using addition of one number with the 2's complement of the other.

For example:

(X-Y) = X + (2's complement of Y)

An Example:
As an example, let us subtract 5 from 7.

Binary representation of 5 is (0101)2.

The 1's complement of 5 is then: (1010)2.

2's complement of 5 = (1's complement of 5 + 1) = (1010+1)2 = (1011)2.

Now, 7 + (-5) = 0111 + 1011 = (1)0010.

Binary Subtraction of Floating Point numbers:

While subtracting two integer numbers is easy as shown above, subtraction of floating point numbers is where it gets complicated.
Let us see how to subtract (0110000010)2 from (0111000011)2

We'll assume both numbers are in floating point binary format with 6 bits for mantissa and 4 bits for exponent. Both the numbers are in 2's complement.

First, we write down the subtraction statement: (0110000010)2 −(0111000011)2

Next, we write in exponent form. To do this, we move the decimal point from the leftmost bit to the right till we encounter a '1', all the while increasing the exponent value by 1. Since we are dealing with binary numbers, the exponent will be raised to 2.

$0.11100*2^3 - 0.11000*2^2$
Since the exponents aren't equal, we increase the exponent of the subtrahend: $0.11100*2^3 - 0.01100*2^3$

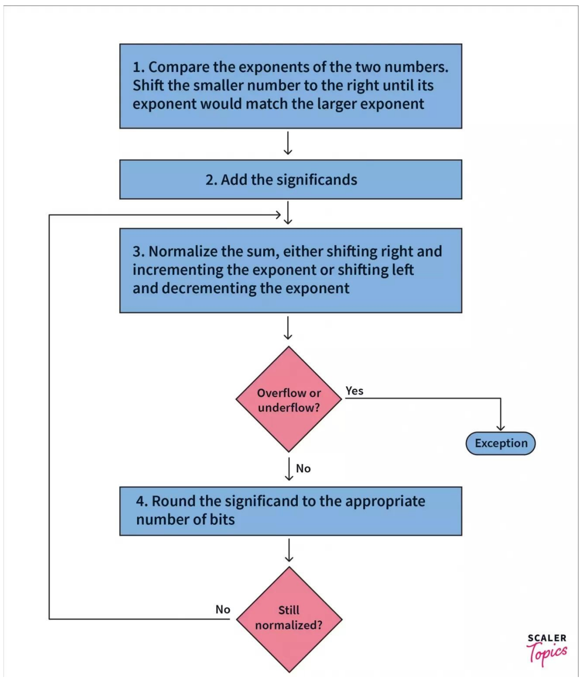
Now, we negate the mantissa of the second number. We do this by inverting the bits and adding one, i.e. taking 2's complement: 1.10011+1=1.10100

Now, we add the mantissas together: $0.11100*2^3 + 1.10100*2^3$
+0.11100+1.10100=(1)0.10000
We have a carryover of 1, to which we will get back to in the moment. The result now is $0.10000*2^3$

Now removing the decimal point and converting it to binary, we get: 010000 0011

Which is our answer. Hence, $0.11100*2^3 - 0.11000*2^2$

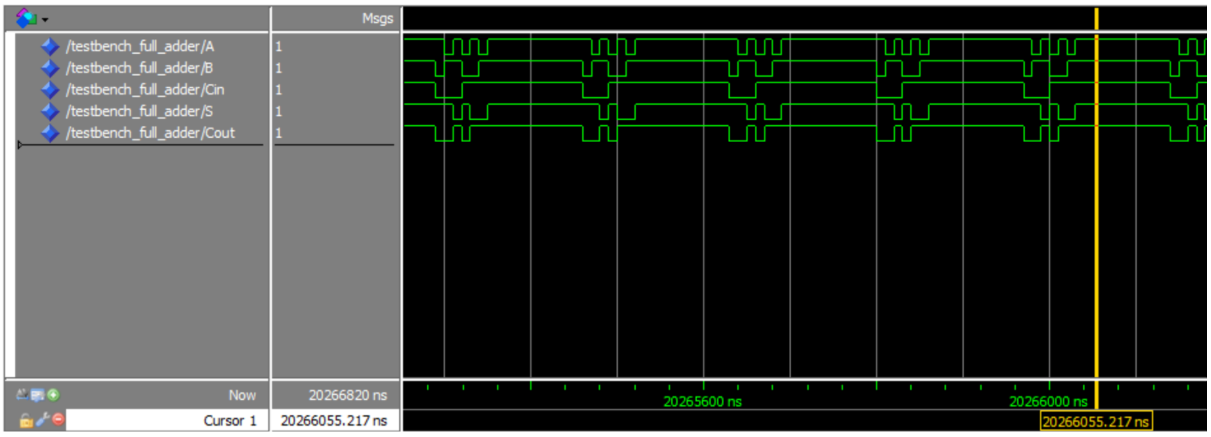
A flowchart of the subtraction of two floating point binary addition and subtraction is as follows:


1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
SCALER Topics

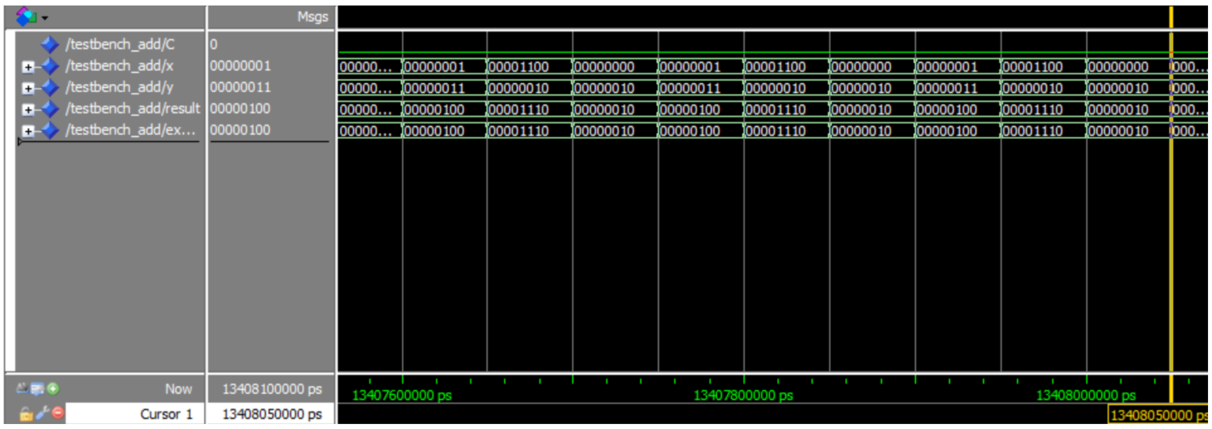
IV - Testing (Signed numbers):

Link : Projet FPU FPGA Testcase's Results

V - Simulation Results


Msgs
/testbench_full_adder/A 1
/testbench_full_adder/B 1
/testbench_full_adder/Cin 1
/testbench_full_adder/S 1
/testbench_full_adder/Cout 1
Now 20266820 ns
Cursor 1 20266055.217 ns
20265600 ns
20266000 ns
20266055.217 ns

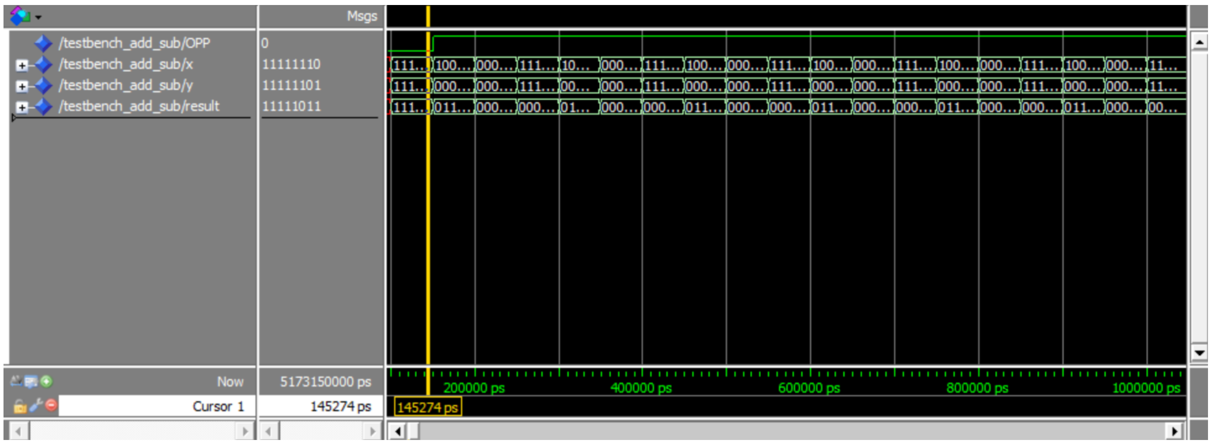
Full Adder TestBench



```
                          Msgs
/testbench_add/C          0
/testbench_add/x          00000001   00000... 00000001 00001100 00000000 00000001 00001100 00000000 00000001 00001100 00000000 000...
/testbench_add/y          00000011   00000... 00000011 00000010 00000010 00000011 00000010 00000010 00000011 00000010 00000010 000...
/testbench_add/result     00000100   00000... 00000100 00001110 00000010 00000100 00001110 00000010 00000100 00001110 00000010 000...
/testbench_add/ex...      00000100   00000... 00000100 00001110 00000010 00000100 00001110 00000010 00000100 00001110 00000010 000...

Now       13408100000 ps     13407600000 ps     13407800000 ps     13408000000 ps
Cursor 1  13408050000 ps                                                13408050000 ps
```

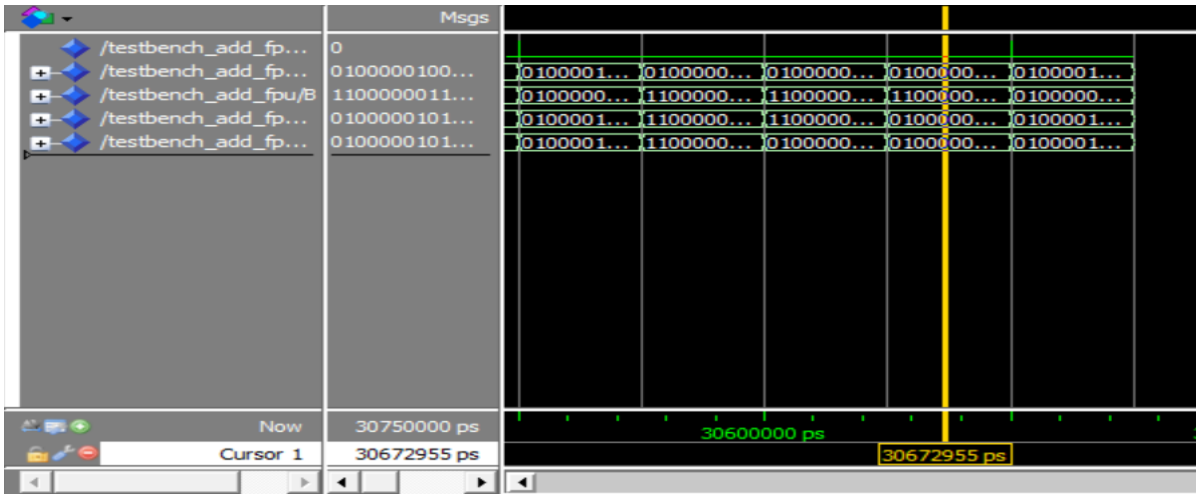
8-bit Addition TestBench



```
                              Msgs
/testbench_add_sub/OPP        0
/testbench_add_sub/x          11111110   111... 100... 000... 111... 10... 000... 111... 100... 000... 111... 100... 000... 111... 100... 000... 111... 100... 000... 11...
/testbench_add_sub/y          11111101   111... 000... 000... 111... 00... 000... 111... 000... 000... 111... 000... 000... 111... 000... 000... 111... 000... 000... 11...
/testbench_add_sub/result     11111011   111... 011... 000... 000... 01... 000... 000... 011... 000... 000... 011... 000... 000... 011... 000... 000... 011... 000... 00...

Now       5173150000 ps     200000 ps     400000 ps     600000 ps     800000 ps     1000000 ps
Cursor 1  145274 ps         145274 ps
```

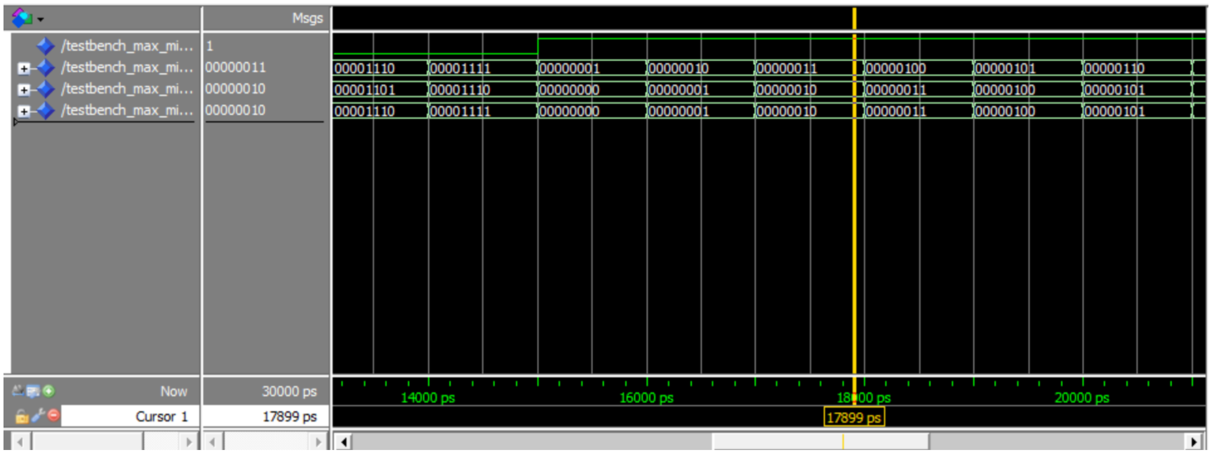
8-bit ADDITION & Subtraction TestBench



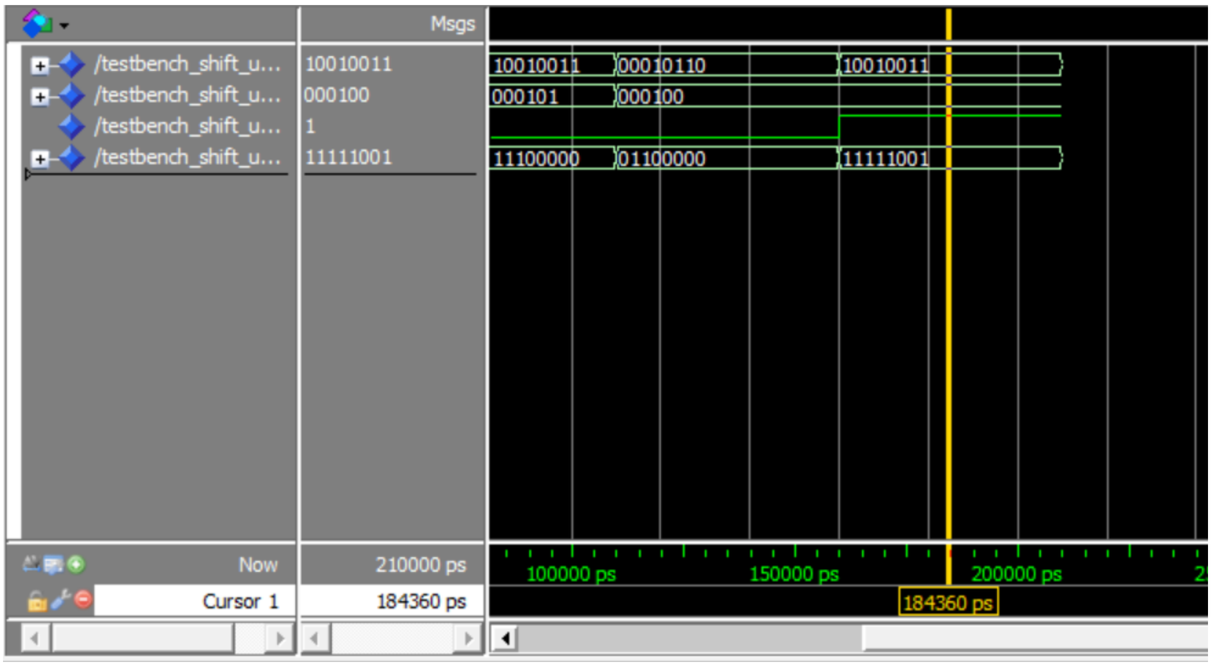
```
                              Msgs
/testbench_add_fp...          0
/testbench_add_fp...          0100000100...   0100001... 0100000... 0100000... 0100000... 0100001...
/testbench_add_fpu/B          1100000011...   0100000... 1100000... 1100000... 1100000... 0100000...
/testbench_add_fp...          0100000101...   0100001... 1100000... 1100000... 0100000... 0100001...
/testbench_add_fp...          0100000101...   0100001... 1100000... 0100000... 0100000... 0100001...

Now       30750000 ps       30600000 ps
Cursor 1  30672955 ps       30672955 ps
```

32-bit FPU Addition & Subtraction TestBench



8-bit Max-Min TestBench



Shift unit TestBench