Spring 2018

SAN JOSÉ STATE UNIVERSITY

Department of Electrical Engineering

EE-275- ADVANCED COMPUTER ARCHITECTURE

MINI-PROJECT REPORT # 1

PIPELINED FLOATING-POINT ADDER DESIGN

Submitted to –

Prof. Chang Choo
Director of AI FPGA/DSP Hardware Laboratory

Submitted by –

Shivang Singh, MSEE
010939851

-- February 28th, 2018 –

TABLE OF CONTENTS

ABSTRACT

The main objective of this project is the learn the designing and implementation of a 5-stage pipelined IEEE single precision floating point adder. IEEE is the central and largest technical organization providing educational and technical advancements for electrical and electronic engineering peers worldwide. The floating-point representation is the representation in which a word is divided into 3 parts, which includes a sign bit, an exponent and a fraction part. Based on above knowledge and pre-defined rules for floating points, we are going to prepare a 5-staged pipelined adder. The pipelining ahead of the IEEE format adds multiple features of utilizing the resources in a very systematic and useful convenient manner.

In our report, we will also be discussing the history, architecture and designing of the 5-point adder, which will help us to understand the topic in a more well and organized manner. Also during the span of time, we will also see the code and eight test cases as examples for the utilization of this 5-point adder. The waveforms including the code are also seen in further report.

INTRODUCTION

In computer science, everything is done in a stationary configuration which contains a precise uniqueness. Among many, one is floating point arithmetic representation. In a very general and wide prospect, a number is a represented to a fixed number of significant digit and scaled using an exponent in some fixed base; and the base can be either of two, ten or sixteen.

The term floating point refers to the fact that a number's decimal point can float, that means it can be placed anywhere relative to the significant digits of the number. The position is expressed as the exponent component and thus the representation is of kind of scientific notation [1].

Floating point representation hugely allows a wider range of number within a limited bit space than compared to the fixed-point representation. In floating point representation, the word is divided into three parts, a sign, an exponent and a mantissa also known as a fraction.

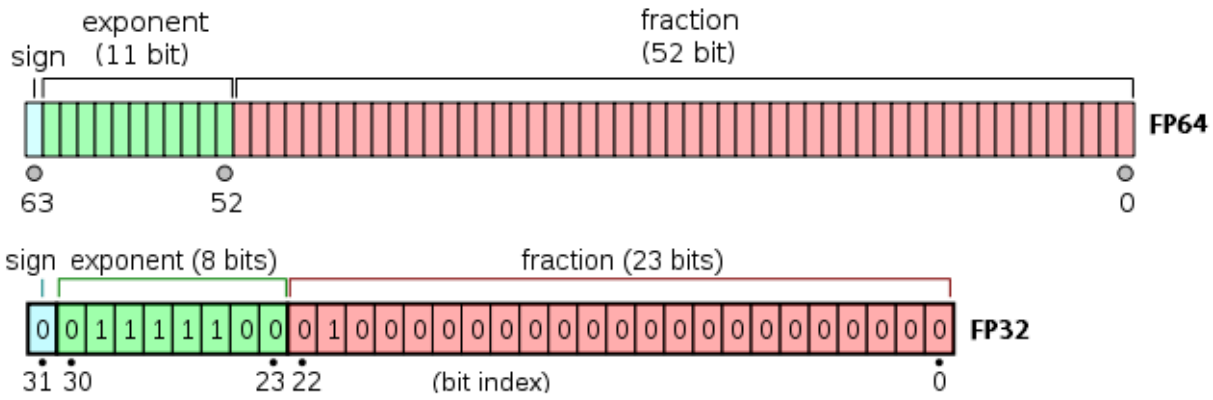
With the long span of time, a variety of floating point representation have been used in computers. But after since 1990s, the most commonly used representation is the one defined by IEEE 754 standard. The speed of the floating point is measured in terms of Floating Point operations per second (FLOPS), which is also a measure of your computers performance, especially when involved in high intensive high speed mathematical calculations. Also, a FPU (Floating Point Unit), is integral part of the computer system which is designed specially to carry out the operation of the floating-point numbers.

The floating-point format of the numbers in computer sciences is basically a number format which occupies 64 bits in the computer memory and represents wide range of numeric values by a floating radix point. They are generally used to represent fractional values. In IEEE 754-2008 standard, the 64-bit base 2 format is refereed as binary64 and was called double in IEEE 754-1985. It also specifies multiple additional floating point formats, including the 32-bit base 2 single precision or what we call today as base 10 representation. Now the IEEE 754-1985 is the most precious and adopted format for the representation and properties of floating point data types depending upon the computer manufacturer and computer model too.

The IEEE 754 standard specifies a binary 64 as having following things –

1. Sign Bit: 1 bit
2. Exponent: 11 bits
3. Significant precision: 52 bits

The figure showing the allocation of bits in single precision floating point and FP64 alignment.



Here the sign bit determines the sign of the number, including the fact when it is Zero too. The exponent field can be interpreted as either an 11-bit signed integer from 1024 and 1023 or any 11-bit unsigned integer from 0 to 2047. The 53-bit significand precision gives from 15 to 17 significant decimal digit precision. If a decimal string with atleast15 significant digits is converted to IEEE format, then converted back to decimal sting with same number of digits and then result should match the original number [3].

This data is presented in a tabular data with the IEEE 754 standards –

PARAMETER	**FORMAT**	
	Single Precision	**Double Precision**
Format width (in bits)	32	64
Precision fraction+ hidden bit	23+1	52+1
Exponent Width (in bits)	8	11
Max. value of exponent	+127	+1023
Min. value of exponent	-126	-1022

There are several exceptions in IEEE 754 also, which can be summarized as below–

EXCEPTIONS	**EXPLANATIONS**
Overflow	Result can be +/- infinity or max. value
Underflow	Result can be 0
Divide by Zero	Result can be +/- infinity
Invalid	Result is NOT A NUMBER (NaN)
Inexact	System rounding may be needed

HISTORY

The history of IEEE 754 which is a standard given by IEEE, an international organization; for floating point arithmetic established in 1985. Since the year 1985, it has become very popular and is being used tremendously in nearly all the hardware systems [4]. This standard defines several formats, such as:

- **Arithmetic Formats:** It contains a set of binary and decimal floating point data, including finite numbers, infinities and "not a number" category.
- **Interchange Formats:** Bit strings that are used to exchange floating point data in a compact form.
- **Rounding Rules:** Properties when rounding the numbers are rounding property is done.
- **Operations:** Arithmetic operations.
- **Exception Handling:** When there is an exceptional condition.

DESIGN

The designing in the 5-stage pipelined SP floating point adder is performed into 2 steps. They are illustrated as below –

STEP-1: DESIGNING OF AN UNPIPELINED SINGLE PRECISION FLOATING POINT ADDER-

In this step the single precision floating point adder is designed which is NOT pipelined and simply made to check the synthesis of the two numbers in 8 different test cases. Then the un-pipelined floating point adder is made to check the working of these 8 cases. Once this adder is successfully derived and made then it is further tried to transform it into the pipelining process to obtain our aim.

STEP-2: CHANGE IN DESIGN OF ADDER –

The second stage of the project deals with changing the design of the 5-stage pipelined adder with several conditions and is made to work for the eight pairs of input as given to us.

The table with the test cases as inputs in our adder is drawn below for consideration –

Test Case	A	B
1	98	169
2	99	-89
3	-45	79
4	-283	-66

5	0	0
6	0	-117
7	-110.125	99.875
8	110.875	99.125

The stages which will be combined in Step 2 will be:

Stage 1 - We will be comparing the exponents and determining the number of shifts which is needed to align the mantissa for making the exponents equal. This is also referred as Alignment-1.

Stage 2 - We will be right shifting the mantissa of the smaller exponent by the amount for alignment purpose only.

Stage 3 – Then we are comparing the two aligned mantissas and determine which one is the smaller among the two given to us. Also in the same step, we will take 2's complement of the smaller mantissa in case if their signs are different else not.

Stage 4 – We will now add the two aligned mantissas and further determine the number of shifts needed and the direction of the normalization of the result. This step is also known as Normalization. The normalization step is composed of two steps, among which this is the first one.

Stage 5 – We will now be shifting the mantissa to the required direction by the required amount and adjust the exponent and check if there any exceptional condition occurs or not. This is the final step of the Normalization process.

SIMULATION (Theory of Operation)

The simulation is done successfully with the proof of completion of the task shown by the following simulation waveforms. The code is written here, alongside the test cases which I tried in the simulation in VHDL, one by one.

The operation is done in 2 types –

1. Un-Pipelined Adder (including CODE & TEST-BENCH)

2. Pipelined Adder (including CODE & TEST-BENCH)

So now I will present the 1st part of Un-Pipelined Adder with its ***code***, ***test bench*** and simulation ***test samples*** and their ***waveforms.***

1. Un-Pipelined Adder

Un-Pipelined Source Code –

```
/*

FLOATING POINT IEEE 32 BIT FORMAT
Sign    Exponent    Mantissa
 1         8          23      (bits)



ADDER MODULE NAME: float_add_81
MODULE PINS:
        INPUT: a81(32 bit),b81(32 bit), clk81, reset_81
        OUTPUT: result_81(32 bit)
```

```
*/
module float_add_81 (result_81, a81, b81, clk81, reset_81);

    input [31:0]  a81, b81;
    input clk81, reset_81;
    output [31:0] result_81;

    wire [31:0] result_81;

    parameter reg_81=46;

    reg [7:0] exp_a81,exp_b81, exp_diff_81, result_exp_81;
    reg sign_a81, sign_b81;
    reg [22:0] men_a81, men_b81;
    reg [reg_81:0] fract_a81,fract_b81, fract_result_81;
    reg zeroa81,zerob81;
    reg result_sign_81;
    reg [31:0] final_result_81;
    integer renorm_81;

    parameter [reg_81:0] zero_81=0;


    assign result_81 = final_result_81;

    always @ (posedge clk81 or posedge reset_81) begin

    if (reset_81) begin
          exp_a81<= 0;
          exp_b81<= 0;
          exp_diff_81 <= 0;
          result_exp_81 <= 0;
          exp_a81 <= 0;              // Exponent
          exp_b81 <= 0;
          sign_a81 <= 0;             //Sign
```

```
        sign_b81 <= 0;
        men_a81 <= 0;                   // Mantissa
        men_b81 <= 0;
        fract_a81 <= 0;
        fract_b81 <= 0;
        final_result_81 <= 0;

end

else begin


    zeroa81 = (a81[30:0]==0)?1:0;
    zerob81 = (b81[30:0]==0)?1:0;
    renorm_81=0;
    if (b81[30:0] > a81 [30:0]) begin       // Smaller number always goes in the B part for both cases that makes the execution easy

        exp_a81 = b81[30:23];                                   // Exponent
        exp_b81 = a81[30:23];
        sign_a81 = b81 [31];                                    //Sign
        sign_b81 = a81[31];
        men_a81 = b81 [22:0];                                   // Mantissa
        men_b81 = a81 [22:0];
        fract_a81 = (zerob81)?0:{ 2'b1, b81[22:0],zero_81[reg_81:25]};
        fract_b81 = (zeroa81)?0:{ 2'b1, a81[22:0],zero_81[reg_81:25]};

    end
    else begin

        exp_a81 = a81[30:23];                                   // Exponent
        exp_b81 = b81[30:23];
        sign_a81 = a81 [31];                                    //Sign
        sign_b81 = b81[31];
        men_a81 = a81 [22:0];                                   // Mantissa
        men_b81 = b81 [22:0];
```

```
    fract_a81 = (zeroa81)?0:{ 2'b1, a81[22:0],zero_81[reg_81:25]};
    fract_b81 = (zerob81)?0:{ 2'b1, b81[22:0],zero_81[reg_81:25]};

end
result_sign_81 = sign_a81;
exp_diff_81 = exp_a81 - exp_b81;                    // Difference of Exponent

if(exp_diff_81 > 24) begin                          //Shift the mantissa by the Difference of the exponent
    result_exp_81 = exp_a81;
    fract_result_81 = fract_a81;
end

else begin
    result_exp_81 = exp_a81;

     fract_b81 = (exp_diff_81 [4]) ? {16'b0,fract_b81[reg_81:16]} : {fract_b81};
     fract_b81 = (exp_diff_81 [3]) ? {8'b0,fract_b81[reg_81:6]} : {fract_b81};
     fract_b81 = (exp_diff_81 [2]) ? {4'b0,fract_b81[reg_81:4]} : {fract_b81};
     fract_b81 = (exp_diff_81 [1]) ? {2'b0,fract_b81[reg_81:2]} : {fract_b81};
     fract_b81 = (exp_diff_81 [0]) ? {1'b0,fract_b81[reg_81:1]} : {fract_b81};

end

if (sign_a81 == sign_b81)begin                      // Add the both Mantissa and store it in result.
    fract_result_81 = fract_a81 + fract_b81;
end

else begin
    fract_result_81 = fract_a81 - fract_b81;
end


renorm_81=0;
```

```
if(fract_result_81[reg_81])                                    // Normalize the result by shifting the mantissa.
begin
   fract_result_81={1'b0,fract_result_81[reg_81:1]};
   result_exp_81=result_exp_81+1;
end

if(fract_result_81[reg_81-1:reg_81-16]==0) begin
   renorm_81[4]=1;
   fract_result_81={ 1'b0,fract_result_81[reg_81-17:0],16'b0 };
end
if(fract_result_81[reg_81-1:reg_81-8]==0) begin
   renorm_81[3]=1;
   fract_result_81={ 1'b0,fract_result_81[reg_81-9:0], 8'b0 };
end
if(fract_result_81[reg_81-1:reg_81-4]==0) begin
   renorm_81[2]=1;
   fract_result_81={ 1'b0,fract_result_81[reg_81-5:0], 4'b0 };
end
if(fract_result_81[reg_81-1:reg_81-2]==0) begin
   renorm_81[1]=1;
   fract_result_81={ 1'b0,fract_result_81[reg_81-3:0], 2'b0 };
end
if(fract_result_81[reg_81-1   ]==0) begin
   renorm_81[0]=1;
   fract_result_81={ 1'b0,fract_result_81[reg_81-2:0], 1'b0 };
end


if (exp_diff_81 <=24)                                          //  2nd normalization process
 begin

    if(fract_result_81 != 0) begin
        if(fract_result_81[reg_81-24:0]==0 && fract_result_81[reg_81-23]==1) begin

            if(fract_result_81[reg_81-22]==1) begin
```

```
                fract_result_81 = fract_result_81+{1'b1,zero_81[reg_81-23:0]};
            end

        end else begin
            if(fract_result_81[reg_81-23]==1) begin
                    fract_result_81=fract_result_81+{1'b1,zero_81[reg_81-24:0]};
            end
        end

        result_exp_81=result_exp_81-renorm_81;

            if(fract_result_81[reg_81-1]==0) begin
                result_exp_81=result_exp_81+1;
                fract_result_81={1'b0,fract_result_81[reg_81-1:1]};
            end
            end else begin
                result_exp_81=0;
                result_sign_81=0;
            end
      end
    //fractPreRound_18
  final_result_81 ={result_sign_81,result_exp_81,fract_result_81[reg_81-2:reg_81-24]};        // Final result
store in this register


 end

 end



endmodule
```

Un-Pipelined Test Bench –

```
//`timescale 1ns/10ps;

module add();

reg[31:0] a81,b81;
wire[31:0] result_81;
reg clk81, reset_81;
float_add_81 X_81 (.a81(a81),.b81(b81),.result_81(result_81),.clk81(clk81),.reset_81(reset_81));

initial begin
clk81 = 0;
reset_81 = 0;
#2;
reset_81 = 1;
#2;
reset_81 = 0;
end


initial forever #5 clk81 = ~clk81;


initial
  begin
  $monitor ("clk81= %d,reset_81 = %d,a81 =%d,b81 = %d,result_81 =%d",clk81,reset_81,a81,b81,result_81);
end



initial
begin


#15;
```

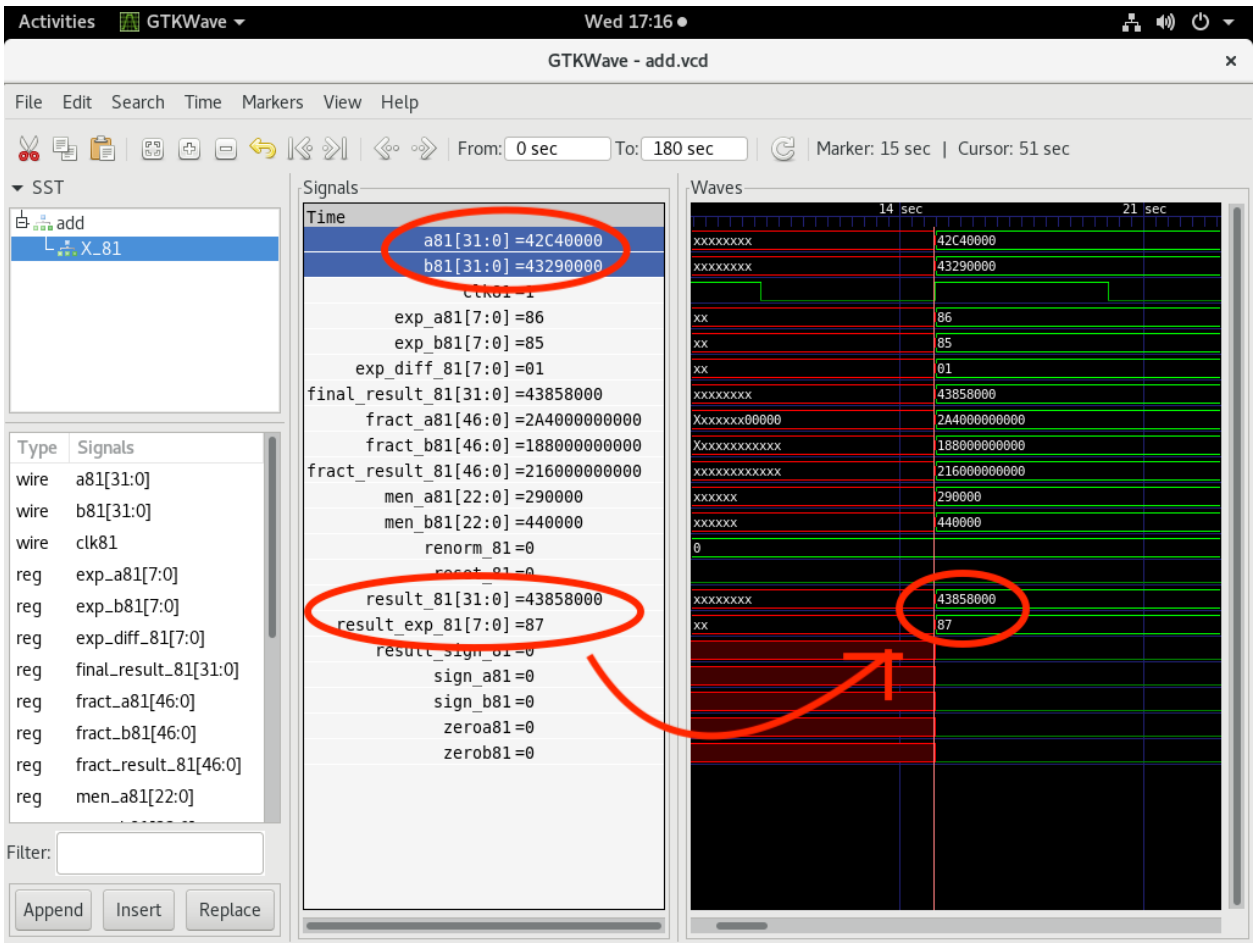
```
a81 = 32'b01000010110001000000000000000000;         // test case for a81=98 and b81=169
b81 = 32'b01000011001010010000000000000000;
#10;
a81 = 32'b01000010110001100000000000000000;           // test case for a81=99 and b81=-89
b81 = 32'b11000010101100100000000000000000;
#10;
a81 = 32'b11000010001101000000000000000000;         // test case for a81=-45 and b81=79
b81 = 32'b01000010100111100000000000000000;
#10;
a81 = 32'b11000011100011011000000000000000;        // test case for a81=-283 and b81=-66
b81 = 32'b11000010100001000000000000000000;
#10;
a81 = 32'b00000000000000000000000000000000;           // test case for a81=0 and b81=0
b81 = 32'b00000000000000000000000000000000;
#10;
a81 = 32'b00000000000000000000000000000000;           //test case for a81=0 and b81=-117
b81 = 32'b11000010111010100000000000000000;
#10
a81 = 32'b11000010110111000100000000000000;           // test case for a81=-110.125 and b81=99.875
b81 = 32'b01000010110001111100000000000000;
#10;
a81 = 32'b01000010110111011100000000000000;            // test case for a81=110.875 and b81=99.125
b81 = 32'b01000010110001100100000000000000;
#100
$finish;
end


initial begin
$dumpfile("add.vcd");
$dumpvars(0);
end

endmodule
```

Test Samples and their waveforms –

Test Case 1:

A = 98 (01000010110001000000000000000000)

B = 169 (01000011001010010000000000000000)

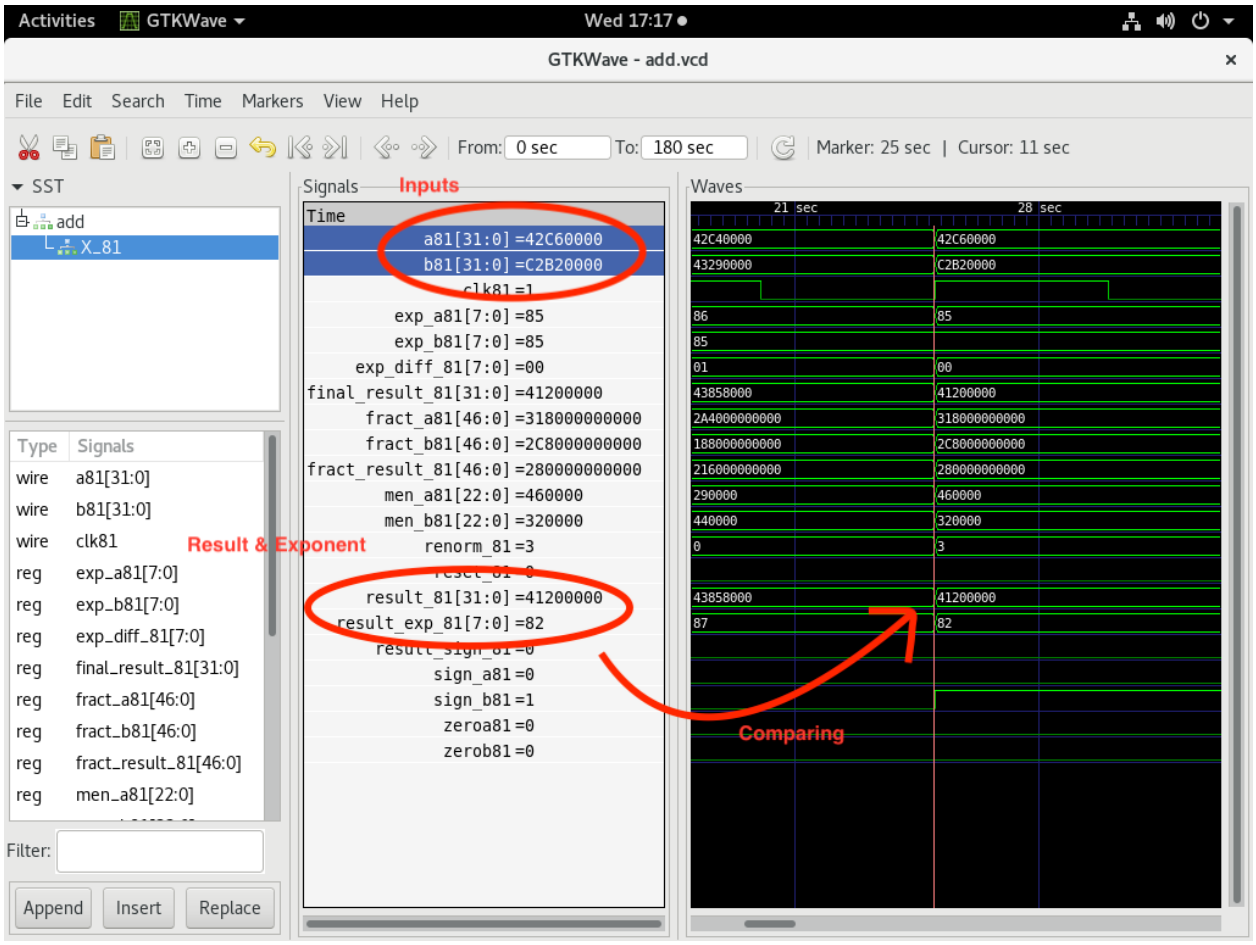


Here in this waveform we can see that the test case 1 with inputs of A=98 and B=169 are being taken into consideration and the final output is obtained in the form of 43850000 and the exponent is 87. This process takes complete 51 seconds for the task to complete, it is taking this much long since it is not pipelined and the second input is entered only after the output of the first set of input is calculated and shown here.

Test Case 2:

A = 99 (01000010110001100000000000000000)

B = -89 (11000010101100100000000000000000)

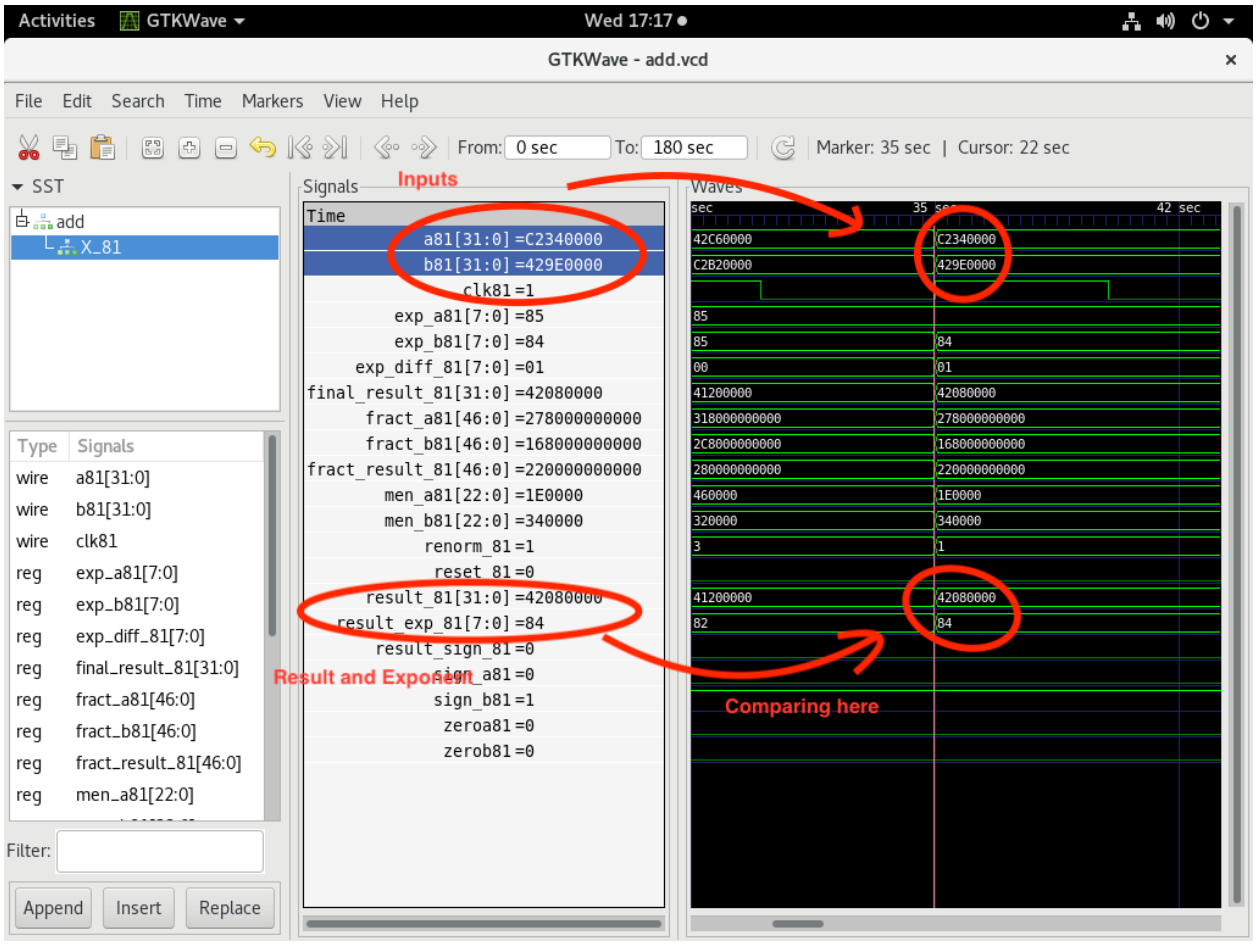


In this waveform, we can see that the 2 inputs are send and then the outputs are obtained and the time taken is 25 seconds. The output which is obtained has an exponent value of 82 and it can be seen in the waveform.

Test Case 3:

A = -45 (11000010001101000000000000000000)

B = 79 (11000010001101000000000000000000)

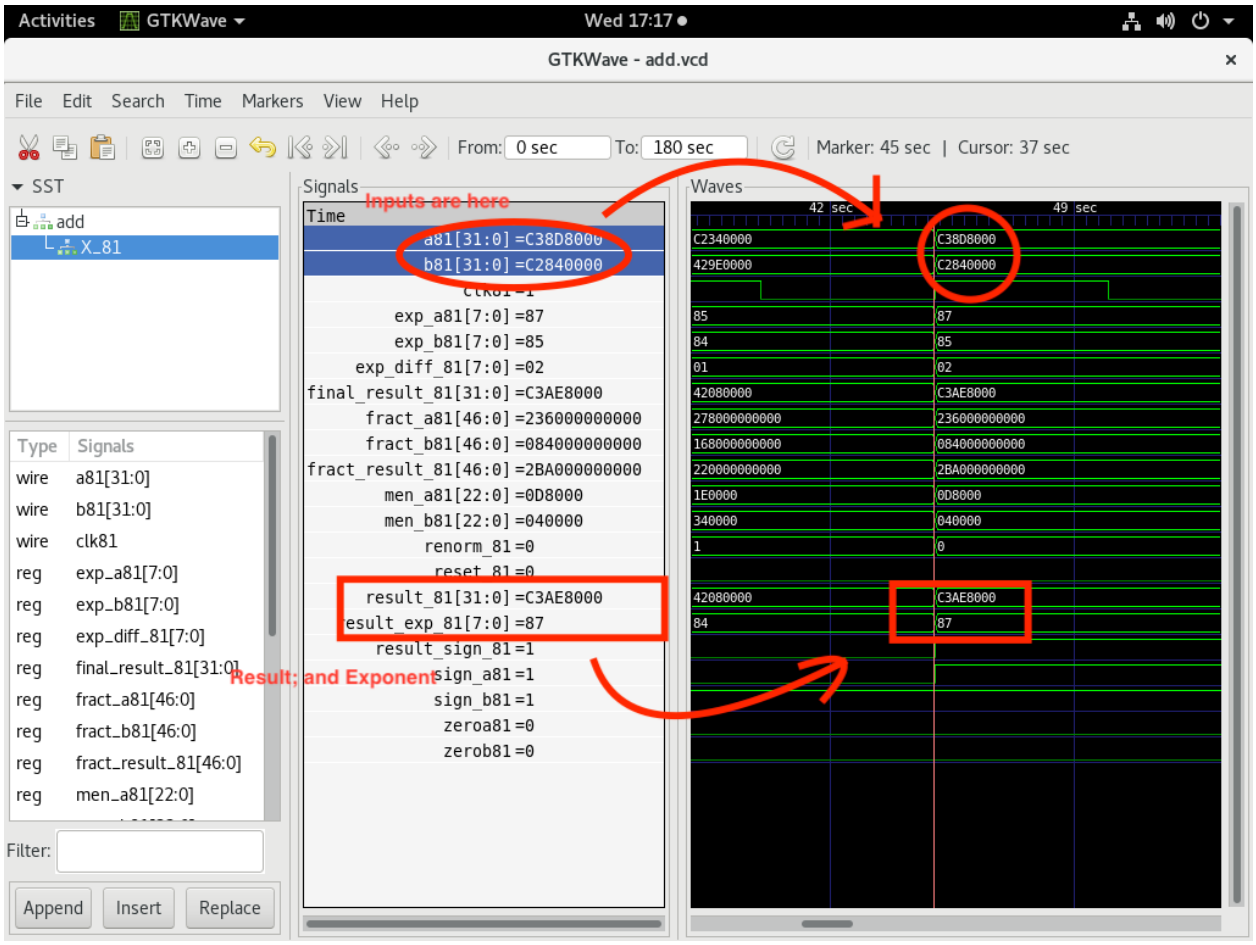


We can see in this waveform we can see that the test case 1 with inputs of A= -45 (C2340000) and B=79 (429E0000) are being taken into consideration and the final output is obtained in the form of 42080000 and the exponent is 84. This process is taking much long time since it is not pipelined and the second input is entered only after the output of the first set of input is calculated and shown here.

Test Case 4:

A = -283 (11000011100011011000000000000000)

B = -66 (11000010100001000000000000000000)

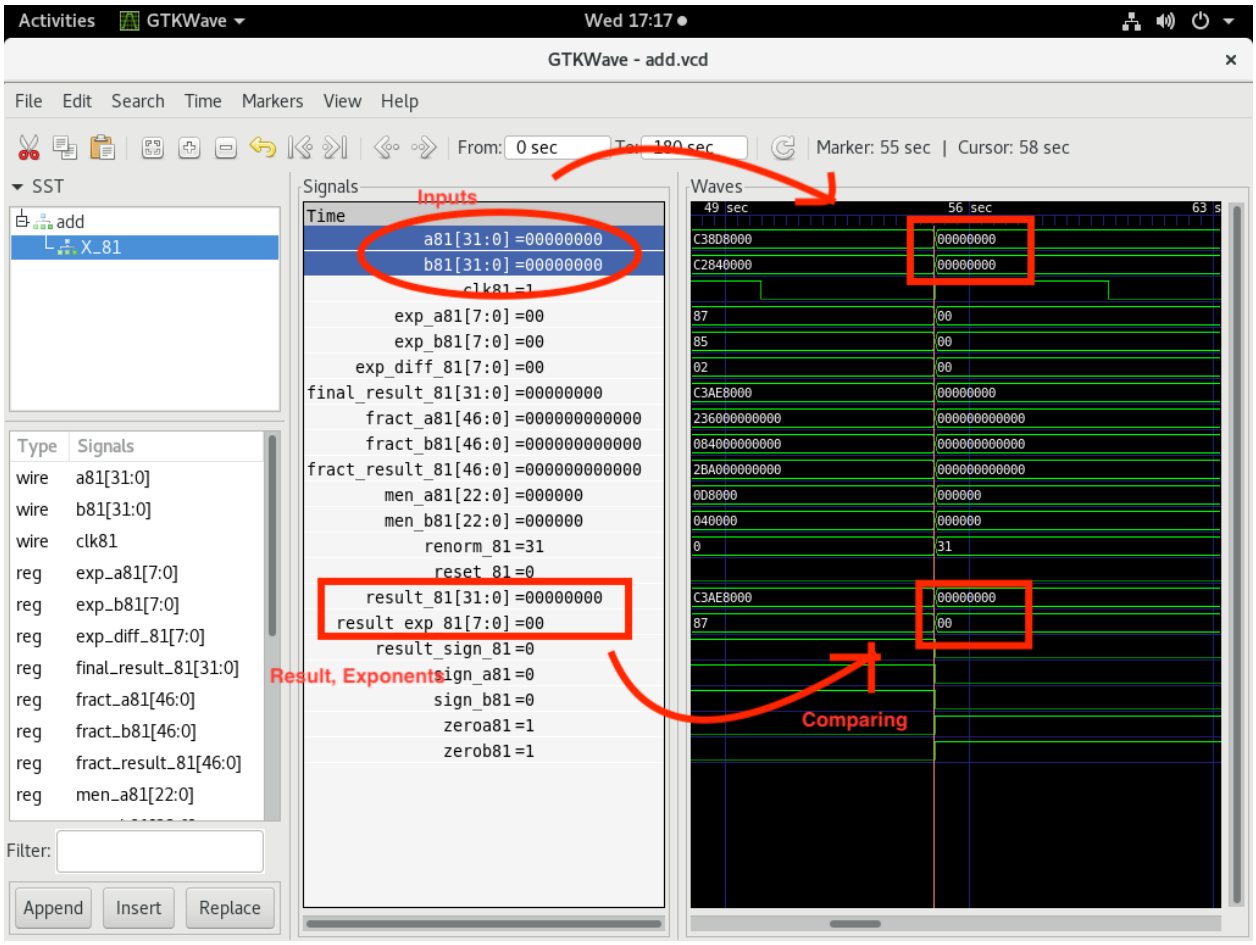


We can see in this waveform we can see that the test case 1 with inputs of A= -283 (C3808000) and B=-66 (C2840000) are being taken into consideration and the final output is obtained in the form of C3AE8000 and the exponent is 87. This process is taking much long time since it is not pipelined and the second input is entered only after the output of the first set of input is calculated and shown here.

Test Case 5:

A = 0 (00000000000000000000000000000000)

B = 0 (00000000000000000000000000000000)

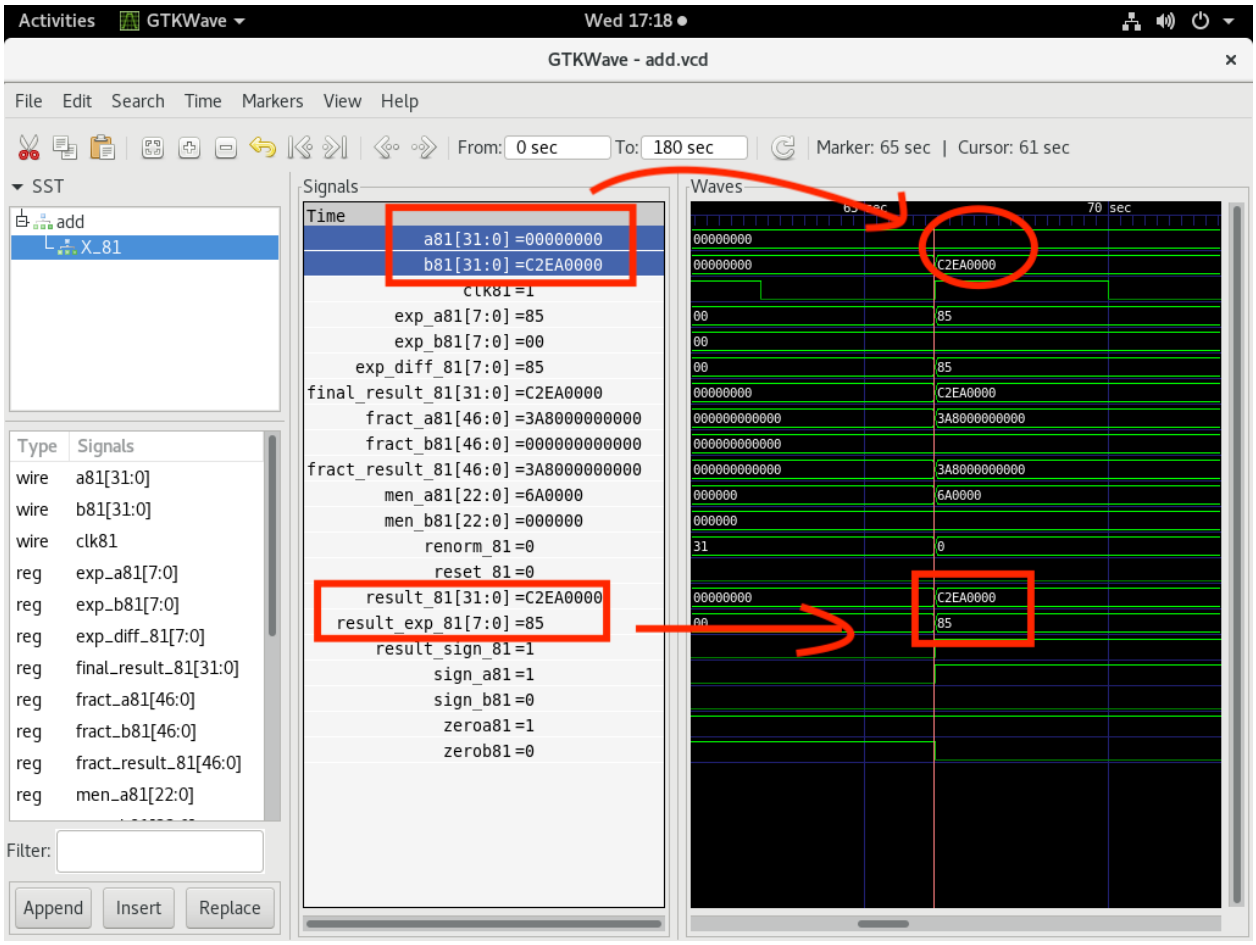


We can see in this waveform we can see that the test case 1 with inputs of A= 0 (00000000) and B= 0 (00000000) are being taken into consideration and the final output is obtained in the form of 00000000 and the exponent is 00.

Test Case 6:

A = 0 (00000000000000000000000000000000)

B = -117 (11000010111010100000000000000000)

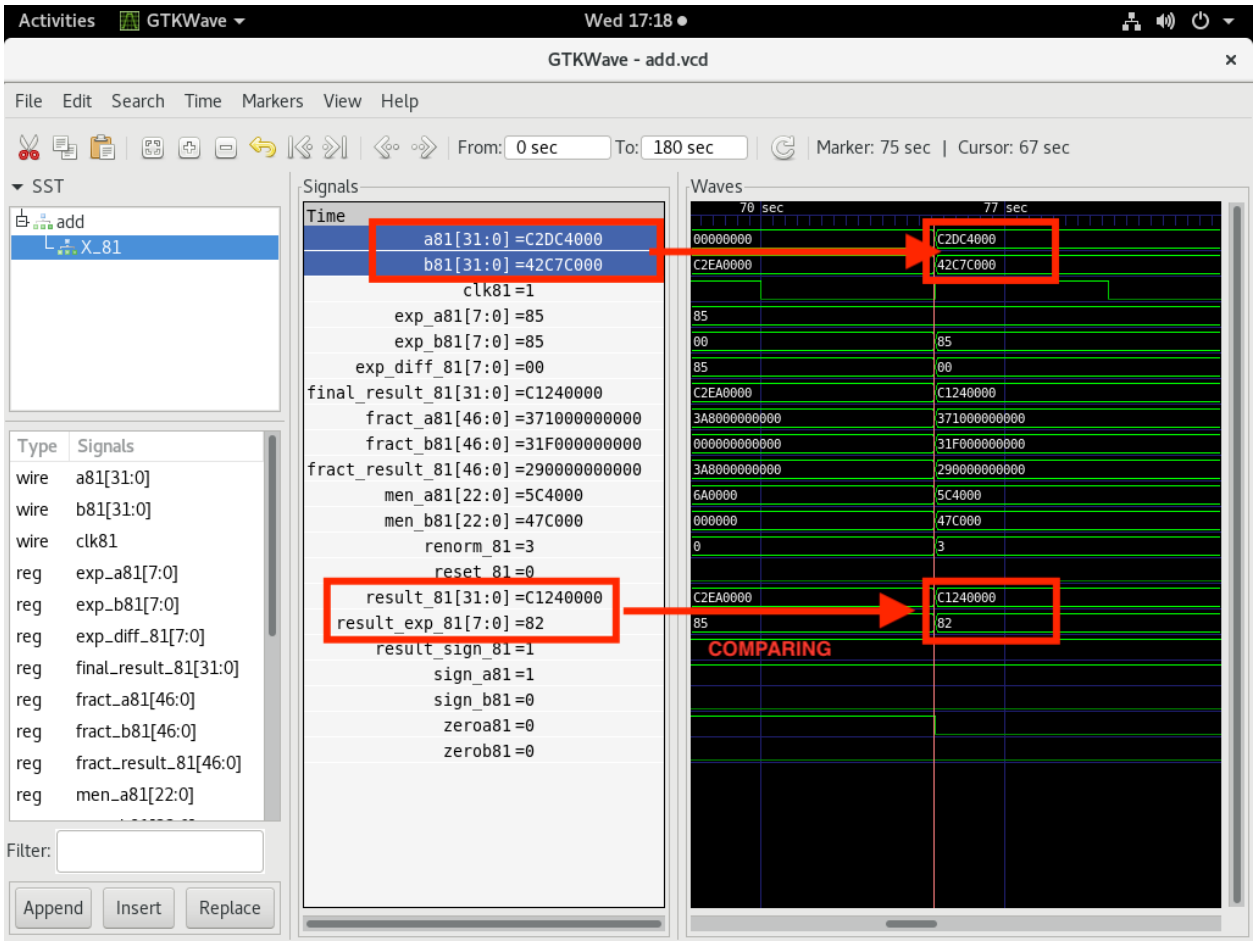


We can see in this waveform we can see that the test case 1 with inputs of A= 0 (00000000) and B= -117 (C2EA0000) are being taken into consideration and the final output is obtained in the form of C2EA0000 and the exponent is 85. This process is taking much long time since it is not pipelined and the second input is entered only after the output of the first set of input is calculated and shown here.

Test Case 7:

A = -110.125 (11000010110111000100000000000000)

B = 99.875 (01000010110001111100000000000000)

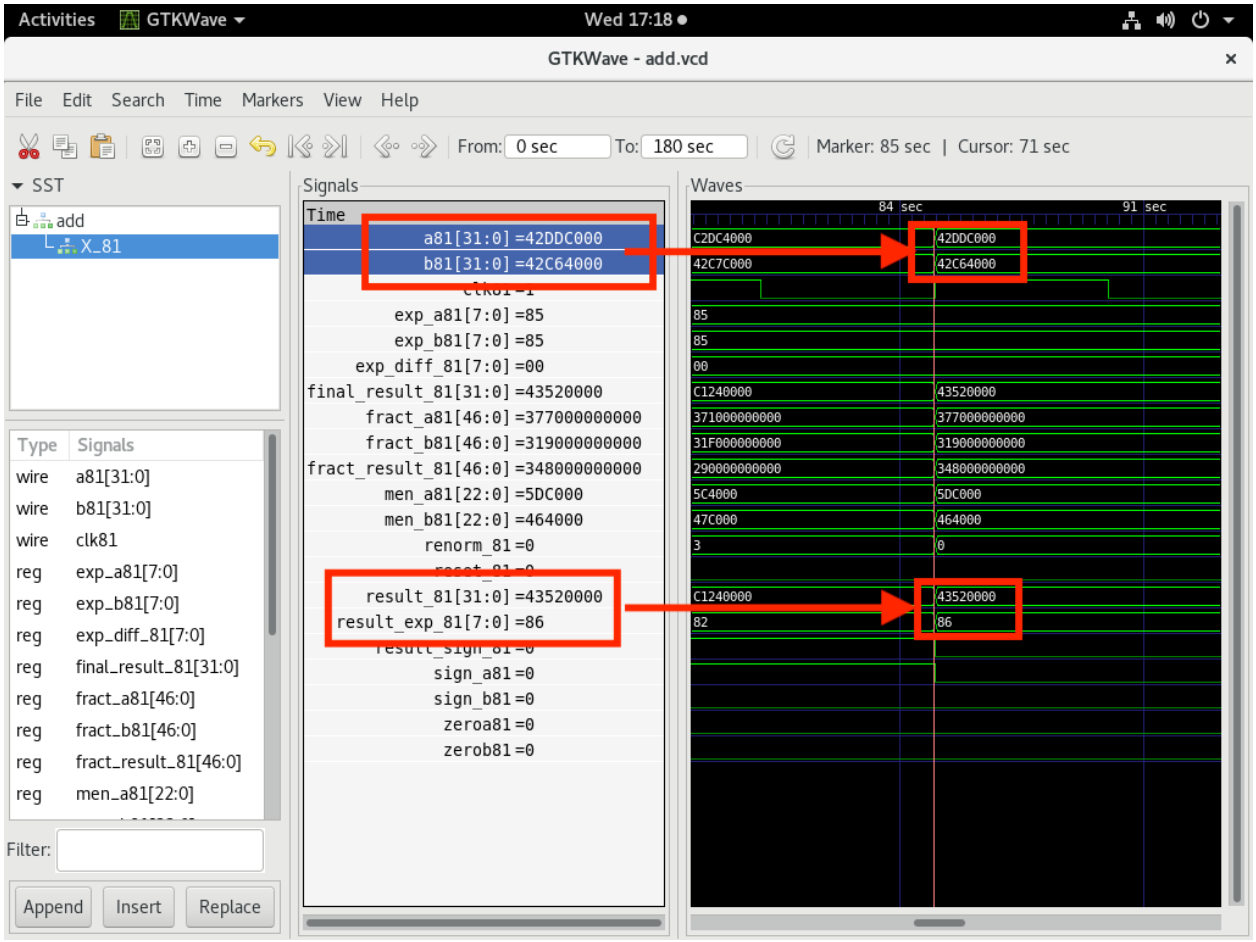


We can see in this waveform we can see that the test case 1 with inputs of A= -110.125 (C2DC4000) and B= 99.875 (42C7C000) are being taken into consideration and the final output is obtained in the form of C1240000 and the exponent is 82. This process is taking much long time since it is not pipelined and the second input is entered only after the output of the first set of input is calculated and shown here.

Test Case 8:

A = 110.875 (01000010110111011100000000000000)

B = 99.125 (01000010110001100100000000000000)



We can see in this waveform we can see that the test case 1 with inputs of A= -110.875 (42DDCOOO) and B= 99.125 (42C64000) are being taken into consideration and the final output is obtained in the form of 43520000 and the exponent is 86.

2. Pipelined Adder

Now I will present the 2ndt part of project which contains the pipelined source code and eight examples showing the working of pipelined adder with its ***code***, ***test bench*** and simulation ***test samples*** and their ***waveforms.***

For designing the pipelined adder, we pipeline the non-pipelined adder into 5 stages, so that the adder can work in pipelining or being parallel in service.

Pipelined Source Code –

```
/*FLOATING POINT IEEE 32 BIT FORMAT
Sign    Exponent    Mantissa
 1         8          23      (bits)



ADDER MODULE NAME: float_add_81
MODULE PINS: 81
        INPUT: a81(32 bit),b81(32 bit), clk81, reset_81
        OUTPUT: result_81(32 bit)


*/



module float_add_81 (result_81, a81, b81, clk81, reset_81);

   input [31:0]  a81, b81;
   input clk81, reset_81;
   output [31:0] result_81;
   wire [31:0] result_81;
```

```
parameter reg_81=46;
// Name of all registers and wires are given per stages for easy understanding.

reg [7:0] exp_a81,exp_b81, exp_diff_81, result_exp_81,exp_a2_81,exp_b2_81, exp_diff2_81,exp_diff3_81,exp_diff4_81,exp_diff5_81,result_exp2_81,result_exp3_81,result_exp4_81,result_exp5_81,result_exp6_81;

reg sign_a81, sign_b81, sign_a2_81, sign_b2_81, sign_a3_81, sign_b3_81;

reg [22:0] men_a81, men_b81;

reg [reg_81:0] fract_a81,fract_b81, fract_result_81,fract_a2_81,fract_a3_81,fract_b2_81,fract_b3_81, fract_result2_81, fract_result3_81, fract_result4_81,fract_result5_81,fract_result6_81;

reg zeroa81,zerob81;

reg result_sign_81, result_sign2_81, result_sign3_81, result_sign4_81, result_sign5_81, result_sign6_81;
reg [31:0] final_result_81;
integer renorm_81,renorm2_81,renorm3_81,renorm4_81,renorm5_81;
parameter [reg_81:0] zero_81=0;


assign result_81 = final_result_81;

always @ (posedge clk81) begin

////////////////////  STAGE - 1  //////////////////
   zeroa81 = (a81[30:0]==0)?1:0;
   zerob81 = (b81[30:0]==0)?1:0;
   renorm_81=0;
   if (b81[30:0] > a81 [30:0]) begin       // Smaller number always goes in the B part for both cases that makes the execution easy

      exp_a81 = b81[30:23];              // Exponent
      exp_b81 = a81[30:23];
```

```
    sign_a81 = b81 [31];                //Sign
    sign_b81 = a81[31];
    men_a81 = b81 [22:0];                // Mantissa
    men_b81 = a81 [22:0];
    fract_a81 = (zerob81)?0:{ 2'b1, b81[22:0],zero_81[reg_81:25]};
    fract_b81 = (zeroa81)?0:{ 2'b1, a81[22:0],zero_81[reg_81:25]};

end
else begin

    exp_a81 = a81[30:23];                // Exponent
    exp_b81 = b81[30:23];
    sign_a81 = a81 [31];                //Sign
    sign_b81 = b81[31];
    men_a81 = a81 [22:0];                // Mantissa
    men_b81 = b81 [22:0];
    fract_a81 = (zeroa81)?0:{ 2'b1, a81[22:0],zero_81[reg_81:25]};
    fract_b81 = (zerob81)?0:{ 2'b1, b81[22:0],zero_81[reg_81:25]};

end
result_sign_81 = sign_a81;
exp_diff_81 = exp_a81 - exp_b81;          // Difference of Exponent


////////////////////     STAGE - 2     ////////////////////
// Take the data from the stage 1 and the shift the mentis by the difference of the Exponent

if(exp_diff2_81 > 24) begin
    result_exp2_81 = exp_a2_81;
    fract_result2_81 = fract_a2_81;
end

else begin
    result_exp2_81 = exp_a2_81;
    //normalize_b81=0;
```

```
   //fract_b2_81 = fract_b2_81 >> (exp_diff2_81-1);
    fract_b2_81 = (exp_diff2_81 [4]) ? {16'b0,fract_b2_81[reg_81:16]} : {fract_b2_81};
    fract_b2_81 = (exp_diff2_81 [3]) ? {8'b0,fract_b2_81[reg_81:6]} : {fract_b2_81};
    fract_b2_81 = (exp_diff2_81 [2]) ? {4'b0,fract_b2_81[reg_81:4]} : {fract_b2_81};
    fract_b2_81 = (exp_diff2_81 [1]) ? {2'b0,fract_b2_81[reg_81:2]} : {fract_b2_81};
    fract_b2_81 = (exp_diff2_81 [0]) ? {1'b0,fract_b2_81[reg_81:1]} : {fract_b2_81};

end
 ////////////////////      STAGE - 3     ////////////////////
// Take the data from the stage 2 and add or subtract the mantissa in this stage



if (sign_a3_81 == sign_b3_81)begin
   fract_result3_81 = fract_a3_81 + fract_b3_81;
end

else begin
   fract_result3_81 = fract_a3_81 - fract_b3_81;
end

////////////////////      STAGE - 4      ////////////////////
// Normalize the added mantissa (first time)



renorm4_81=0;

if (exp_diff4_81 <= 24) begin


   if(fract_result4_81[reg_81])
   begin
      fract_result4_81={1'b0,fract_result4_81[reg_81:1]};
      result_exp4_81=result_exp4_81+1;
```

```
    end

    if(fract_result4_81[reg_81-1:reg_81-16]==0) begin
        renorm4_81[4]=1;
        fract_result4_81={ 1'b0,fract_result4_81[reg_81-17:0],16'b0 };
    end
    if(fract_result4_81[reg_81-1:reg_81-8]==0) begin
        renorm4_81[3]=1;
        fract_result4_81={ 1'b0,fract_result4_81[reg_81-9:0], 8'b0 };
    end
    if(fract_result4_81[reg_81-1:reg_81-4]==0) begin
        renorm4_81[2]=1;
        fract_result4_81={ 1'b0,fract_result4_81[reg_81-5:0], 4'b0 };
    end
    if(fract_result4_81[reg_81-1:reg_81-2]==0) begin
        renorm4_81[1]=1;
        fract_result4_81={ 1'b0,fract_result4_81[reg_81-3:0], 2'b0 };
    end
    if(fract_result4_81[reg_81-1   ]==0) begin
        renorm4_81[0]=1;
        fract_result4_81={ 1'b0,fract_result4_81[reg_81-2:0], 1'b0 };
    end

end
////////////////////     STAGE - 5     ////////////////////////////
// Normalize the added mantissa and exponent (second time)
 if (exp_diff5_81 <=24)
 begin

     if(fract_result5_81 != 0) begin
          if(fract_result5_81[reg_81-24:0]==0 && fract_result5_81[reg_81-23]==1) begin

              if(fract_result5_81[reg_81-22]==1) begin
                  fract_result5_81 = fract_result5_81+{1'b1,zero_81[reg_81-23:0]};
              end
```

```
            end else begin
               if(fract_result5_81[reg_81-23]==1) begin
                        fract_result5_81=fract_result5_81+{1'b1,zero_81[reg_81-24:0]};
               end
            end

            result_exp5_81=result_exp5_81-renorm5_81;

               if(fract_result5_81[reg_81-1]==0) begin
                  result_exp5_81=result_exp5_81+1;
                  fract_result5_81={1'b0,fract_result5_81[reg_81-1:1]};
               end
               end else begin
                  result_exp5_81=0;
                  result_sign5_81=0;
               end
         end

   //Final Output
   final_result_81 ={result_sign6_81,result_exp6_81,fract_result6_81[reg_81-2:reg_81-24]};


end

// for Resetting and pipeline another always block is used

always @ (posedge clk81 or posedge reset_81)   begin

   if (reset_81) begin
      exp_diff2_81 <= 0;
   fract_a2_81 <= 0;
   exp_a2_81 <= 0;
   exp_b2_81 <= 0;
   fract_b2_81 <= 0;
```

```
sign_a2_81 <=0;
sign_b2_81 <= 0;

sign_a3_81 <= 0;
sign_b3_81 <=0;
fract_result3_81 <= 0;
fract_a3_81 <= 0;
fract_b3_81 <= 0;
fract_result_81 <= 0;


fract_result4_81 <= 0;
fract_result5_81 <= 0;
fract_result6_81 <= 0;
result_sign_81 <= 0;
result_sign2_81 <= 0;
result_sign3_81 <= 0;
result_sign4_81 <= 0;
result_sign5_81 <= 0;
result_sign6_81 <= 0;
result_exp_81 <= 0;
result_exp2_81 <= 0;
result_exp3_81 <= 0;
result_exp4_81 <= 0;
result_exp5_81 <= 0;
result_exp6_81 <= 0;

renorm2_81  <= 0;
renorm3_81  <= 0;
renorm4_81  <= 0;
renorm5_81  <= 0;

exp_diff3_81<= 0;
exp_diff4_81<= 0;
exp_diff5_81<= 0;
```

```
exp_a81 = 0;
   exp_b81 = 0;
   sign_a81 = 0;
   sign_b81 = 0;
   men_a81 = 0;
   men_b81 = 0;
   fract_a81 = 0;
   fract_b81 = 0;


end


else begin
exp_diff2_81 <= #1 exp_diff_81;
fract_a2_81 <= #1 fract_a81;
exp_a2_81 <= #1 exp_a81;
exp_b2_81 <= #1 exp_b81;
fract_b2_81 <= #1 fract_b81;
sign_a2_81 <= #1 sign_a81;
sign_b2_81 <= #1 sign_b81;

sign_a3_81 <= #1 sign_a2_81;
sign_b3_81 <= #1 sign_b2_81;
fract_result3_81 <= #1 fract_result2_81;
fract_a3_81 <= #1 fract_a2_81;
fract_b3_81 <= #1 fract_b2_81;


fract_result4_81 <= #1 fract_result3_81;
fract_result5_81 <= #1 fract_result4_81;
fract_result6_81 <= #1 fract_result5_81;

result_sign2_81 <= #1 result_sign_81;
```

```
        result_sign3_81 <= #1 result_sign2_81;
        result_sign4_81 <= #1 result_sign3_81;
        result_sign5_81 <= #1 result_sign4_81;
        result_sign6_81 <= #1 result_sign5_81;

        result_exp2_81 <= #1 result_exp_81;
        result_exp3_81 <= #1 result_exp2_81;
        result_exp4_81 <= #1 result_exp3_81;
        result_exp5_81 <= #1 result_exp4_81;
        result_exp6_81 <= #1 result_exp5_81;

        renorm2_81  <= #1 renorm_81;
        renorm3_81  <= #1 renorm2_81;
        renorm4_81  <= #1 renorm3_81;
        renorm5_81  <= #1 renorm4_81;

        exp_diff3_81<= #1 exp_diff2_81;
        exp_diff4_81<= #1 exp_diff3_81;
        exp_diff5_81<= #1 exp_diff4_81;

        end
    end



endmodule
```

Pipelined Test Bench –

```
//`timescale 1ns/10ps;

module add();

reg[31:0] a81,b81;
wire[31:0] result_81;
reg clk81, reset_81;
float_add_81 X_81 (.a81(a81),.b81(b81),.result_81(result_81),.clk81(clk81),.reset_81(reset_81));

initial begin              // Clock Generator
clk81 = 0;
forever #5 clk81 = ~clk81;
end



initial
begin

 $monitor ("a81 =%b,   b81 = %b,  result_81 =%b",a81,b81,result_81);          // Displaying result

clk81 = 0;
reset_81 = 0;
#2;
reset_81 = 1;
#2;
reset_81 = 0;

// 8 Different Test Cases Are written below -


a81 = 32'b01000010110001000000000000000000;// test case for a81=98 and b81=169
b81 = 32'b01000011001010010000000000000000;
```

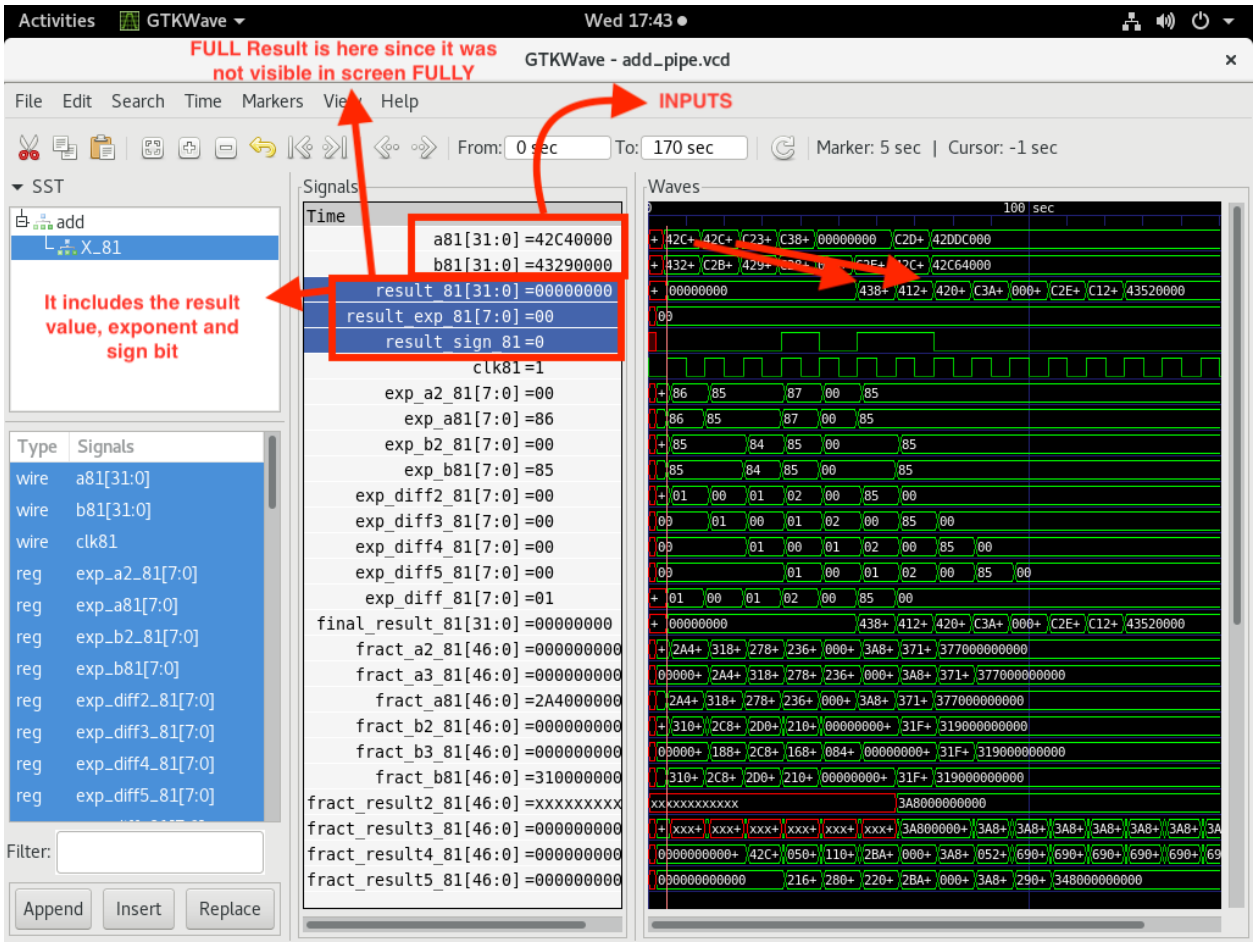
```
#10;
a81 = 32'b01000010110001100000000000000000;// test case for a81=99 and b81=-89
b81 = 32'b11000010101100100000000000000000;
#10;
a81 = 32'b11000010001101000000000000000000;// test case for a81= -45 and b81=79
b81 = 32'b01000010100111100000000000000000;
#10;
a81 = 32'b11000011100011011000000000000000;// test case for a81= -283 and b81=-66
b81 = 32'b11000010100001000000000000000000;
#10;
a81 = 32'b00000000000000000000000000000000;// test case for a81=0 and b81=0
b81 = 32'b00000000000000000000000000000000;
#10;
a81 = 32'b00000000000000000000000000000000;//test case for a81=0 and b81=-117
b81 = 32'b11000010111010100000000000000000;
#10;
a81 = 32'b11000010110111000100000000000000;// test case for a81=-110.125 and b81=99.875
b81 = 32'b01000010110001111100000000000000;
#10;
a81 = 32'b01000010110111011100000000000000;// test case for a81=110.875 and b81=99.125
b81 = 32'b01000010110001100100000000000000;
#100;
$finish;

end


initial begin
$dumpfile("add_pipe.vcd");
$dumpvars(0);
end

endmodule
```

Test Samples and their waveforms –

Test Case 1:

A = 98 (01000010110001000000000000000000)

B = 169 (01000011001010010000000000000000)



Here in this waveform we can CLEARLY see the effect of pipelining. This is the test case 1 with inputs of A=98 and B=169 being taken into consideration. We can see that **DUE TO SMALL SCREEN, THE COMPLETE OUTPUT IS NOT SHOWN IN WAVEFORM BUT INFORMATION IS GIVEN IN THE LEFT**

SIDE WHERE THE OUTPUT IS WRITTEN. It is also illustrated in the waveform image being attached here.

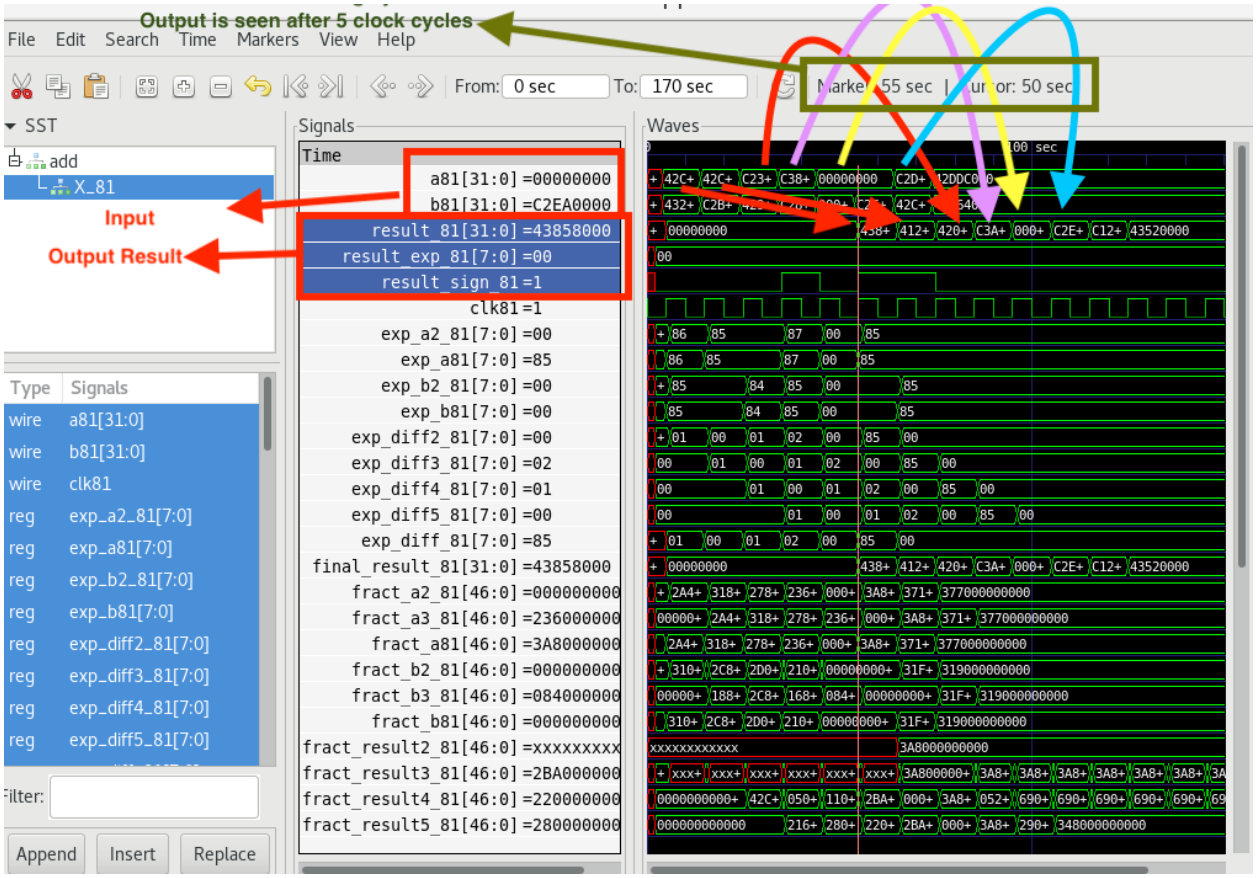
- ALSO, the inputs are marked and clearly shown.
- Here in this pipelined adder, we are getting the output after 5 clock cycles.
- 2nd input starts from the 2nd clock cycle and so for the output of the 2nd input values, the output is obtained after 5 clock cycles and the process goes on in the same manner.

- The proceeds how the inputs are being increased in the time cycle is also being shown and the pipelining process is doing its job and once the 1st input is cleared and move on to the second level, there is another input which is the second input inside the process.

Test Case 2:

A = 99 (01000010110001100000000000000000)

B = -89 (11000010101100100000000000000000)



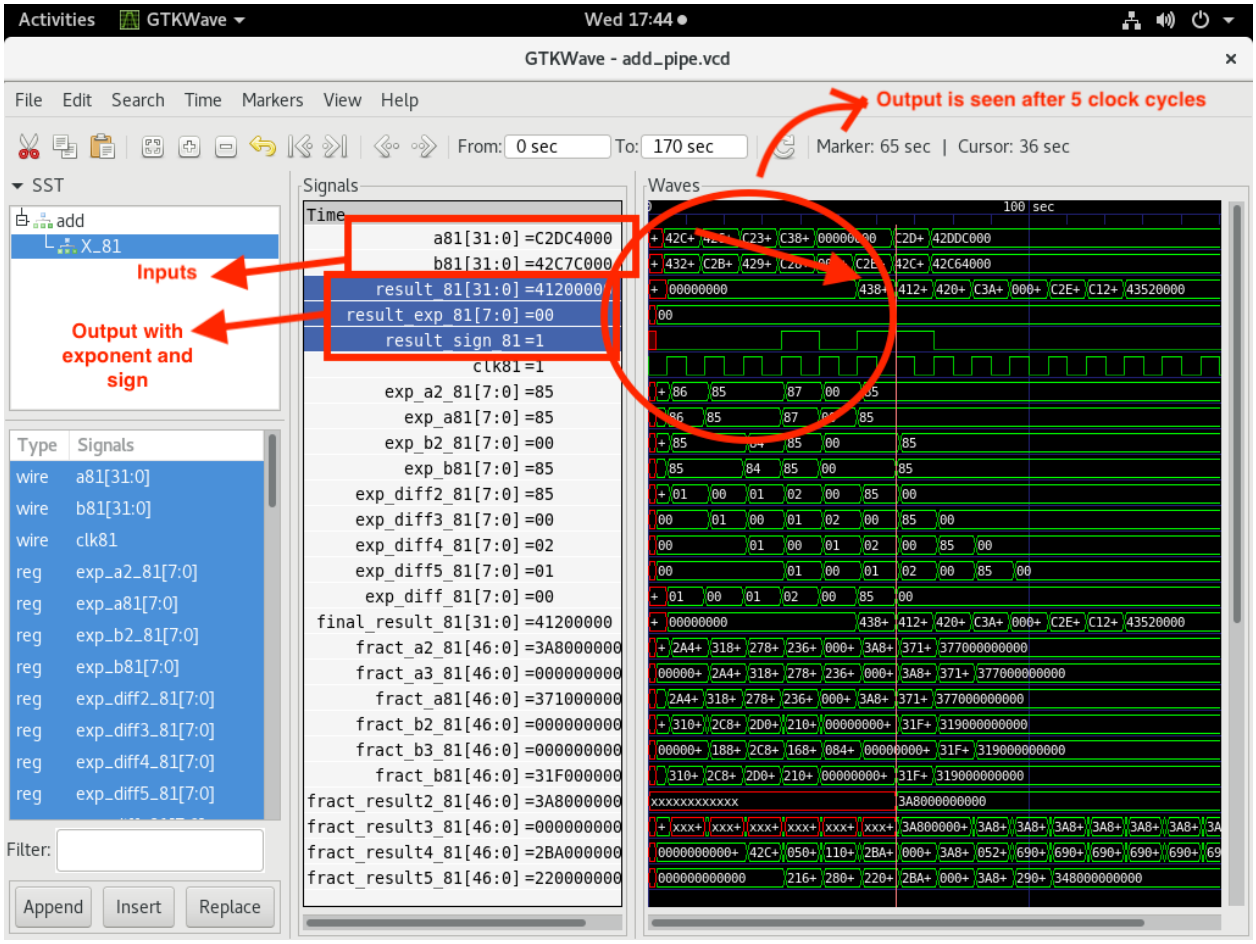
Here in this waveform we can CLEARLY see the effect of pipelining. This is the test case 1 with inputs of A=99 and B=-89 being taken into consideration. It is also illustrated in the waveform image being attached here.

- ALSO, the inputs are marked and clearly shown.
- Here in this pipelined adder, we are getting the output after 5 clock cycles.
- 2nd input starts from the 2nd clock cycle and so for the output of the 2nd input values, the output is obtained after 5 clock cycles and the process goes on in the same manner.
- *The jump of the 1st input and then the 2nd, 3rd and so on is also being shown with multiple color arrows.*

Test Case 3:

A = -45 (11000010001101000000000000000000)

B = 79 (11000010001101000000000000000000)

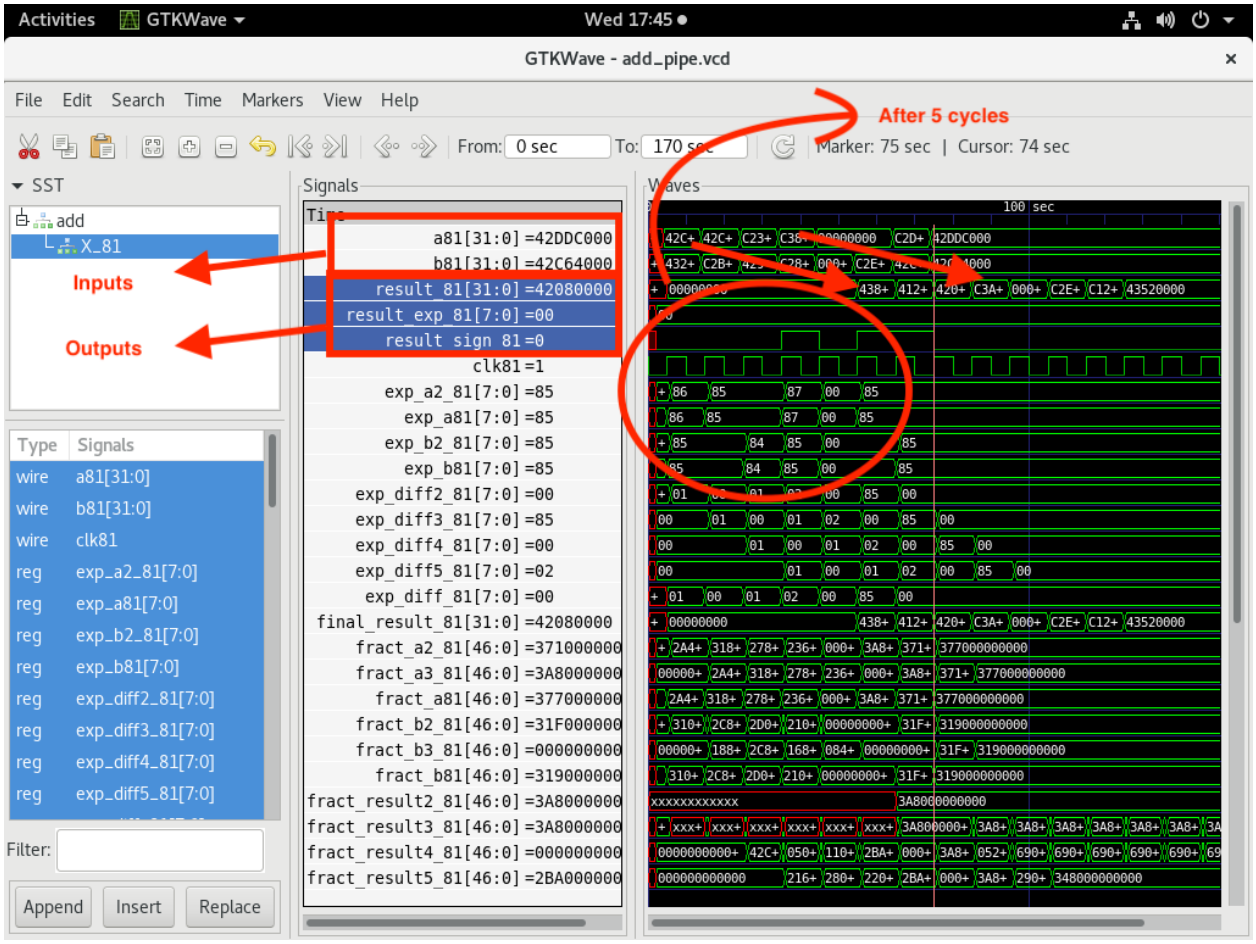


The inputs, outputs and other details are seen in this waveform, and due to small screen the full value of the output is shown on the left-hand side of the image.

Test Case 4:

A = -283 (11000011100011011000000000000000)

B = -66 (11000010100001000000000000000000)



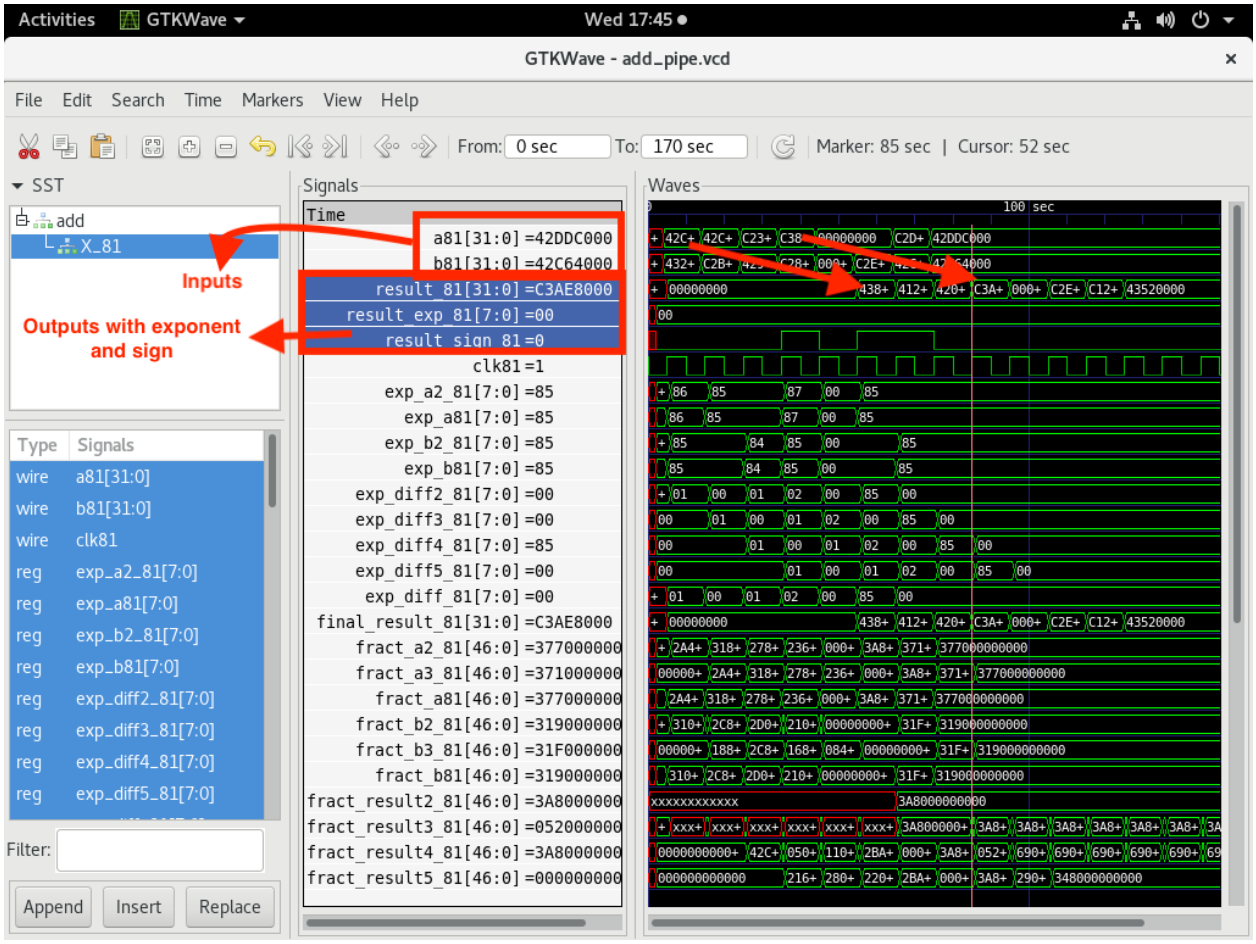
The inputs, outputs and other details are seen in this waveform, and due to small screen the full value of the output is shown on the left-hand side of the image.

The output is achieved after 5 clock cycles.

Test Case 5:

A = 0 (00000000000000000000000000000000)

B = 0 (00000000000000000000000000000000)



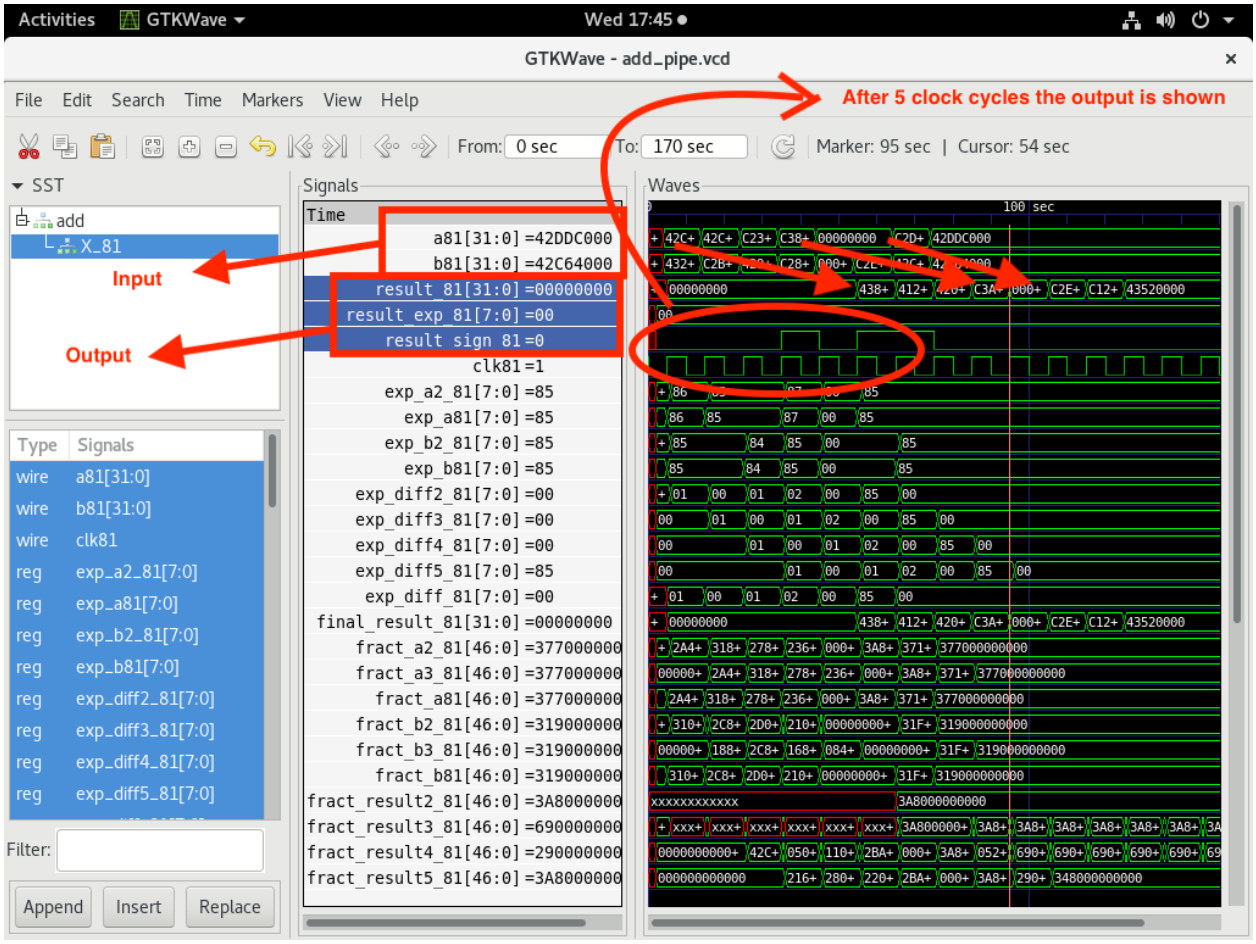
The inputs, outputs and other details are seen in this waveform, and due to small screen the full value of the output is shown on the left-hand side of the image.

The output is achieved after 5 clock cycles.

Test Case 6:

A = 0 (00000000000000000000000000000000)

B = -117 (11000010111010100000000000000000)



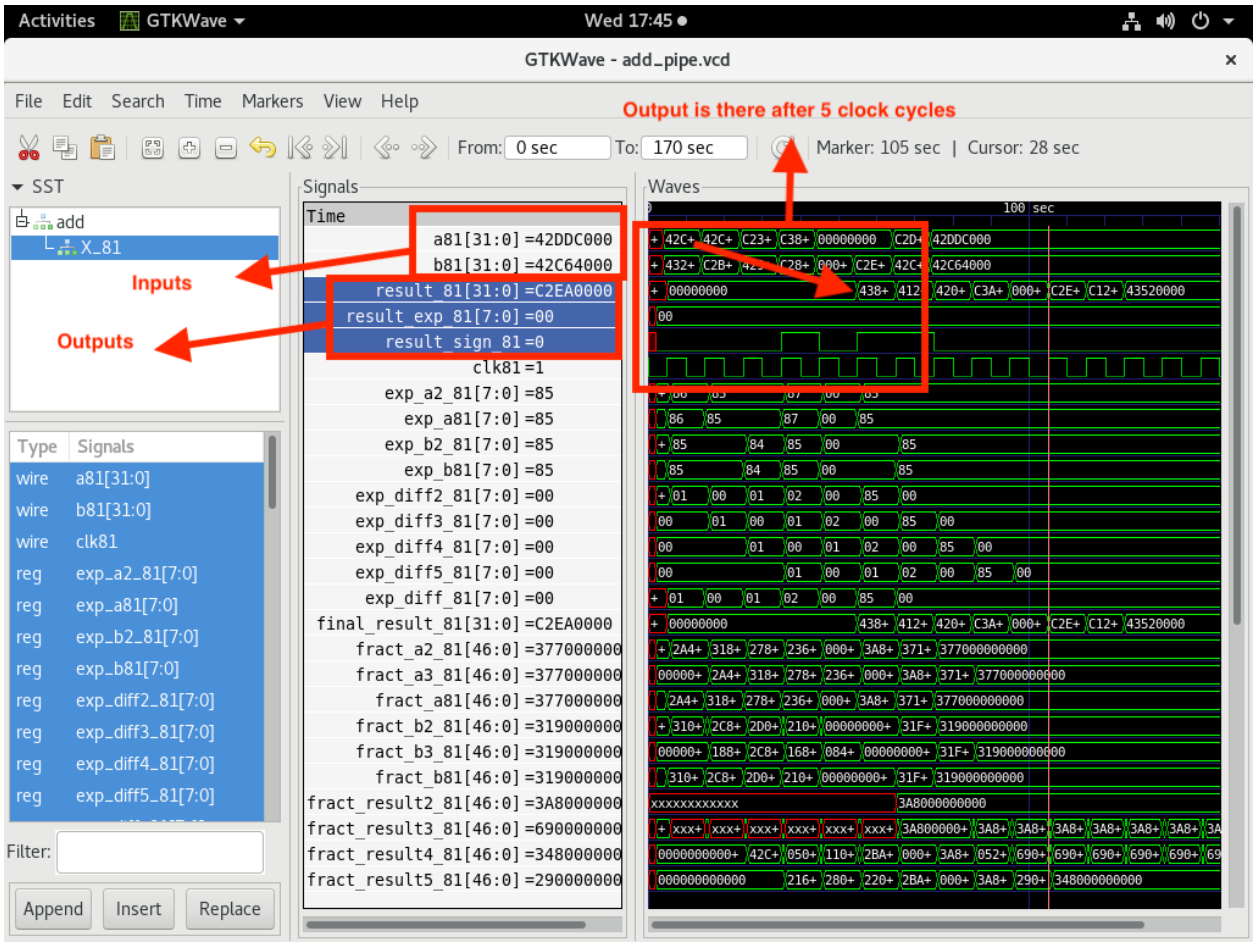
The inputs, outputs and other details are seen in this waveform, and due to small screen the full value of the output is shown on the left-hand side of the image.

The output is achieved after 5 clock cycles.

Test Case 7:

A = -110.125 (11000010110111000100000000000000)

B = 99.875 (01000010110001111100000000000000)



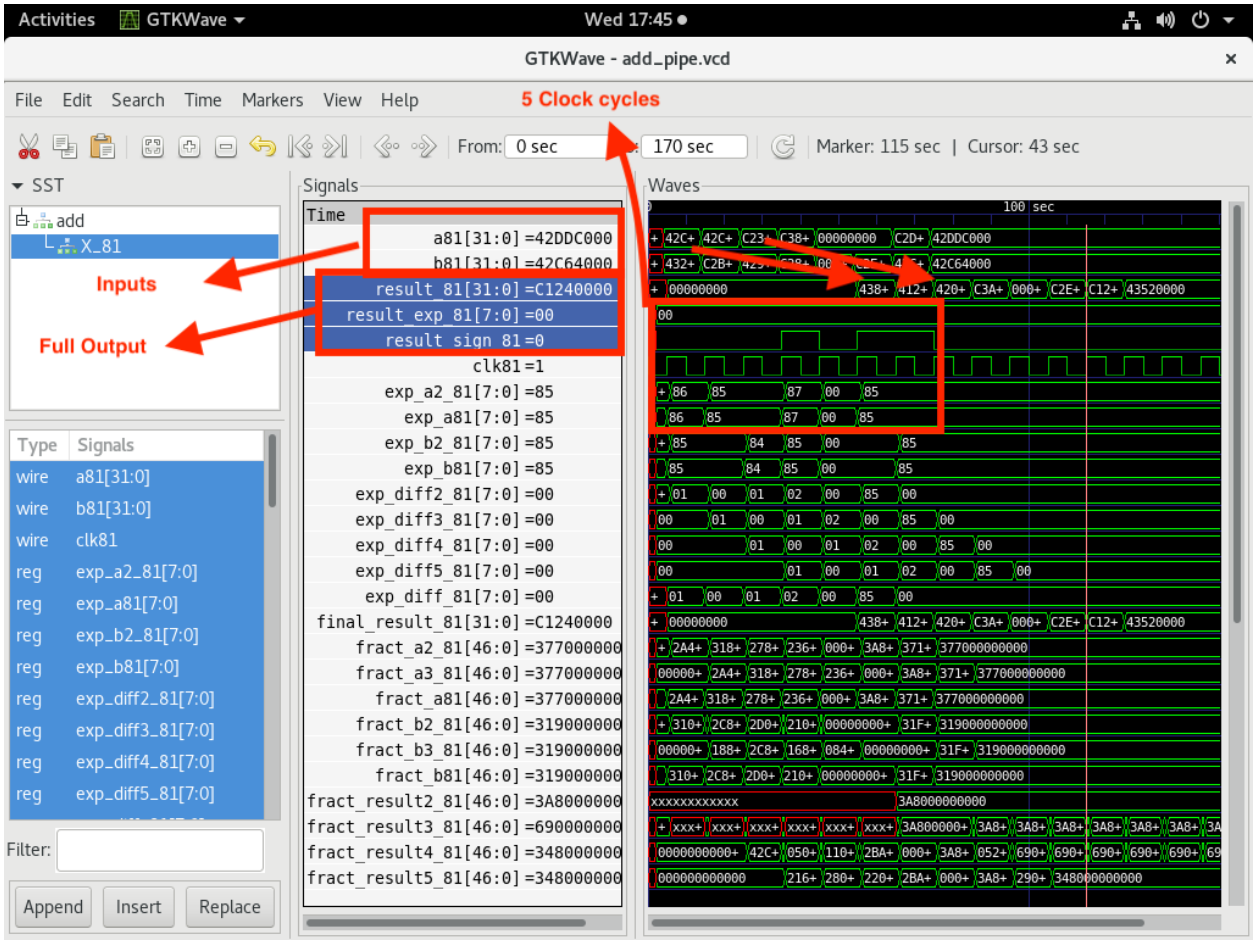
The inputs, outputs and other details are seen in this waveform, and due to small screen the full value of the output is shown on the left-hand side of the image.

The output is achieved after 5 clock cycles.

Test Case 8:

A = 110.875 (01000010110111011100000000000000)

B = 99.125 (01000010110001100100000000000000)



Here in this waveform we can **CLEARLY** see the effect of pipelining. This is the test case 1 with inputs of A=110.875 and B=99.125 being taken into consideration. The inputs are mentioned and the output is obtained after 5 clock cycles.

The pipelining is shown in the waveform and can be understood easily through the waveform, since the 2nd input is inputted while the 1st one is moved ahead after processing the output for it.

CONCULSION

In our project, we have successfully demonstrated the 5 - point adder which is using the special characteristic of being **PIPELINED** in nature. We divided the adder in five multiple stages, where the processing is being done.

We can see that the results have been calculated successfully and we have successfully reduced the time which was required to reduce the addition of the two inputs provided. We could achieve this process by the help of Pipelining technology, in which we divided and pipelined the circuit, such that the second input gets into the processing once the first input has been cleared and there is able space to accomplish the task of the second input. The clock cycle was reduced to five, due to which the output is obtained after five complete cycles and then second input is kept in to the layer for next working.

We have successfully achieved the goals for our project by reducing the time spent on the process of addition of two inputs. Also, we have successfully reduced the clock cycles during which the second input is considered for the processing. This is done with the help of pipelining and we have understood the concept of pipelining and implemented it in our project.

We can also include some more features in coming future out of curiosity. A further feature can be making a multiplier with similar steps for doing hexadecimal and other sort of numbers, other than floating point numbers. Also, we can also make a multi-layered model for calculation of complex mathematical problems, dealing with computer science.

REFERENCES

1. https://en.wikipedia.org/wiki/Floating-point_arithmetic
2. https://wccftech.com/nvidia-gm200-gpu-fp64-performance/
3. http://users.encs.concordia.ca/~asim/COEN_6501/Lecture_Notes/L4_Slides.pdf
4. https://en.wikipedia.org/wiki/IEEE_754