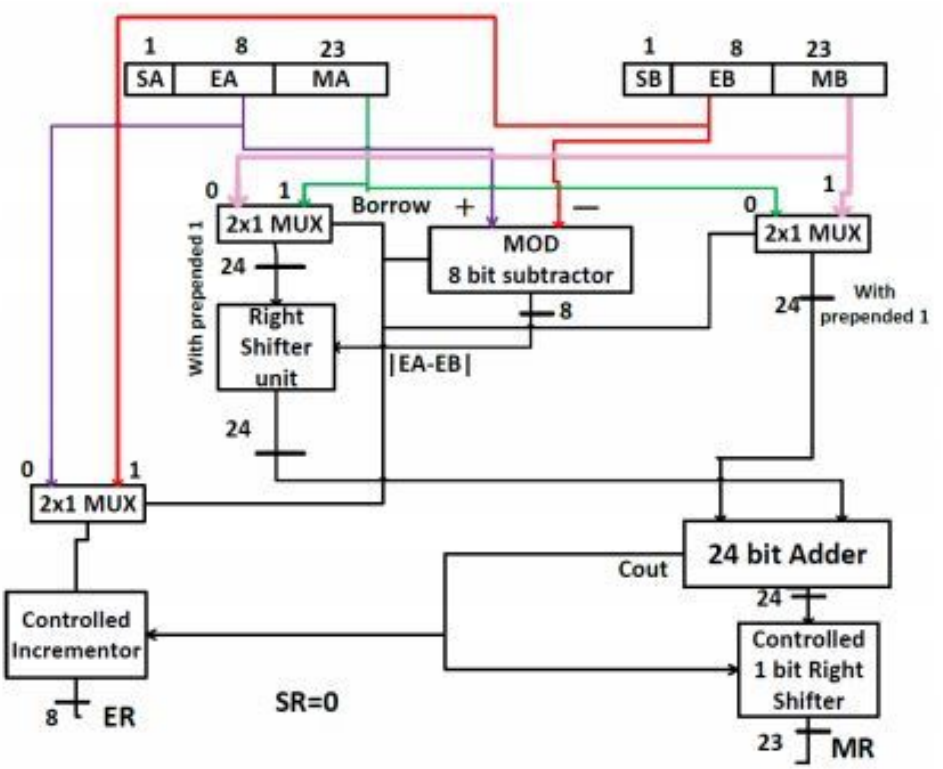
FLOATING POINT ADDER USING IEEE 754 SINGLE PRECISION FLOATING POINT FORMAT



TUESDAY LAB- GROUP 9

Gursharan Ahir 16EC10021

Harsh Barelia 16EC10023

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. We need a FIFO of size 8 at the input of the adder and need 8 single port RAMs at the output of the adder to store the resulting addition. Verify the functionality of the architecture by writing Verilog code using Structural style of modelling and performing post route simulation on FPGA for 8 sets of input.

INTRODUCTION

IEEE 754 is one of the most efficient in most cases. IEEE 754 has 3 basic components:

1) **The Sign of Mantissa:** 0 represents a positive number while 1 represents a negative number.
2) **The Exponent:** The exponent field needs to represent both positive and negative exponents.
3) **The Mantissa :** It represents the significant digits of the number . Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

ALGORITHM BRIEF 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 incremented 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.

HARDWARE ARCHITECTURE

In our project, we implemented the following hardwares in the modules.

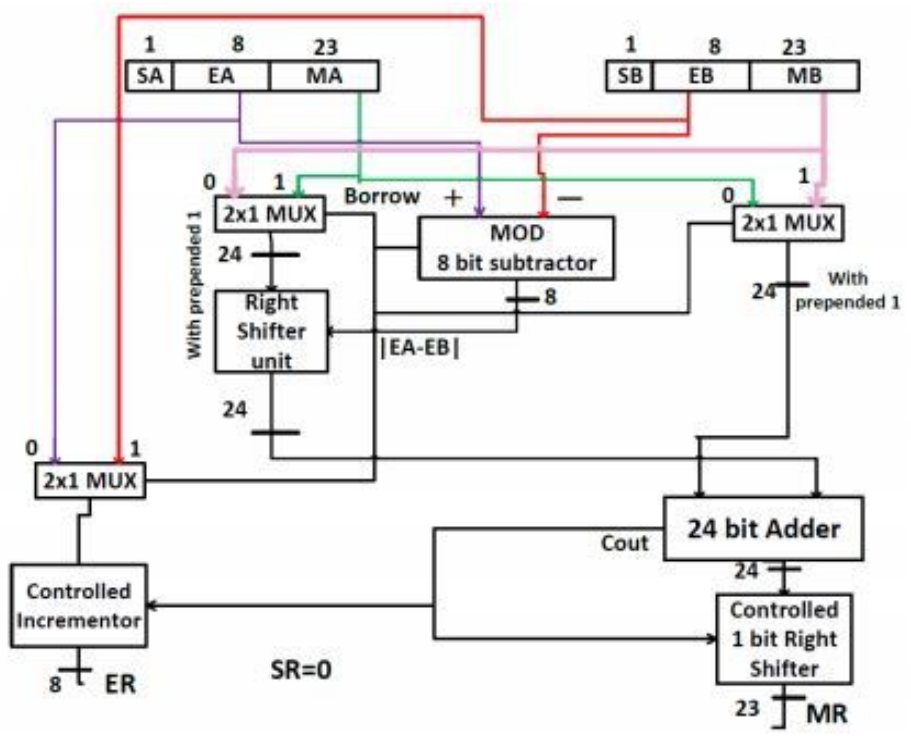
- 2x1 Multiplexers(1 bit, 8 bits and 23 bits)
- Barrel Shifters (implemented through multiplexers)
- 24 Bit Adder (implemented though half and full adders)
- 8-Bit subtractor (implemented through full subtractors)
- Controlled Incrementer(implemented through full and half adders)

Different Modules Used:

```
module FloatingPointAdder(
   input [31:0] A,
   input [31:0] B,
   output [31:0] Out
       );
     wire [22:0]MA;
     wire [22:0]MB;
     wire [7:0]EA;
     wire [7:0]EB;

//Procuring the value of Mantissa MA and MB
assign MA[22:0]=A[22:0];
assign MB[22:0]=B[22:0];

//Procuring the value of Exponents EA and EB
assign EA = A[30:23];
assign EB = B[30:23];

//Calculation of  the magnitude of the subtraction of the exponents
//This will be used to decide the input given to right shifter
wire [7:0]Modulo;
wire Borrow;
Sub_Result subtract(EA,EB,Modulo,Borrow);
//The input with lower exponent value passed to the right shifter unit

//Multiplexer 1 required for the right shifter unit
//Multiplexer 2 required for the 24 Bit adder
wire [23:0]mux1out;
wire [23:0]mux2out;
Mux24 rightshiftertop(MB,MA,Borrow,mux1out);
```

```
Mux24 addertop(MA,MB,Borrow,mux2out);

wire [23:0]outshift;
//control input for the barrel_shifter procured from the output of
subtraction operation(Max 5 bits required)
wire [4:0]shiftdiff;
assign shiftdiff = Modulo[4:0];
BarrelShifter rightshift(mux1out,outshift,shiftdiff);

//24 Bit Adder
wire [23:0]adderout;
wire cout;
Adder_24Bit A1(mux2out,outshift,adderout,cout);

//The maximum of the both inputs' exponents is calculated and used as
final exponent if cout = 0, otherwise incremented
wire [7:0]maxexp ;
Mux_8 expmax(EA,EB,Borrow,maxexp);

//Calculation of the exponent of the output after observing the carry
operation of the 24 bit adder
wire [7:0]expfinal;
ControlledIncrementor finexp(cout,maxexp,expfinal);
wire [23:0]finalM;

//Calculating the selection lines for final shifting operation
wire [4:0]select;
assign select[4:1] = 4'b0000;
assign select[0] = cout;

BarrelShifter finalshifter(adderout,finalM,select);

assign Out[31] = 0;//Positive Number Addition
```

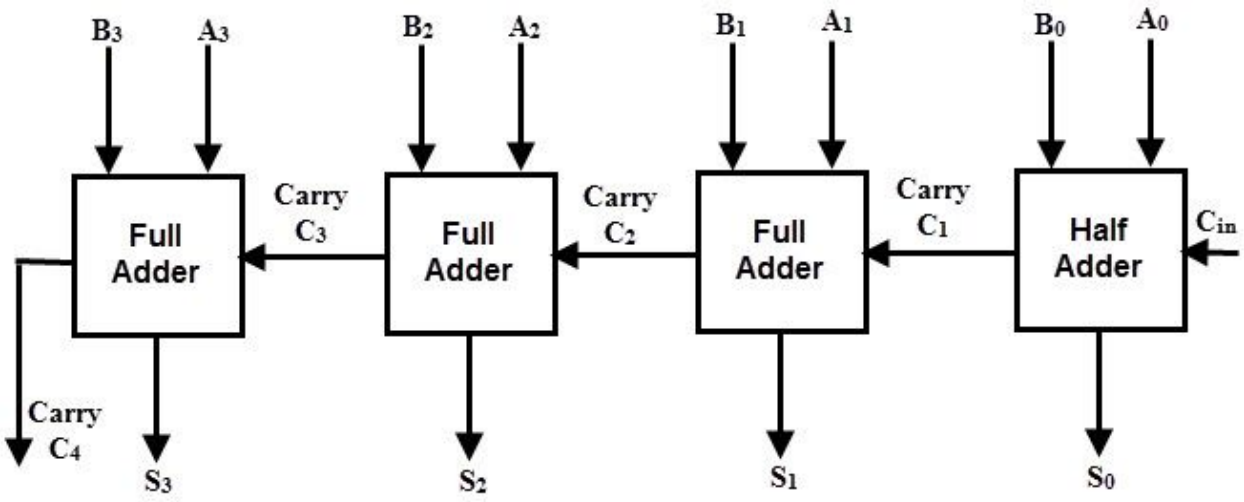
```
assign Out[30:23] = expfinal;
assign Out[22:0] = finalM[22:0];

endmodule
```



Structure of our Floating Point Adder Implementation

```
module Adder_24Bit(
   input [23:0] A,
   input [23:0] B,
   output [23:0] Out,
   output Cout
   );

 wire [23:0] Cin;
 HalfAdder H1(A[0],B[0],Out[0],Cin[0]);

 genvar j;
 generate
  begin
   for(j=1;j<=23;j=j+1)
    begin
     FullAdder F1(A[j],B[j],Cin[j-1],Out[j],Cin[j]);
    end
  end
 assign Cout = Cin[23];
endgenerate

endmodule
```

24 Bit Adder was implemented in a similar structure but with 23 Full adders and 1 Half adder



```
module BarrelShifter(
   input [23:0] In,
   output [23:0] Out,
   input [4:0] Shift
   );

 wire [23:0]a;
 genvar i;

 generate
        begin:b1
                for(i=0;i<23;i=i+1)
                        begin:b2
                        Mux M(In[i] , In[i+1] , Shift[0] , a[i]);
                        end
         Mux M1(In[23] , 1'b0 , Shift[0] , a[23]);
         end
endgenerate

wire [23:0]a1;
genvar j , k;

generate
        begin:b3
                for(j=0;j<22;j=j+1)
                        begin:b4
                        Mux M2(a[j] , a[j+2] , Shift[1] , a1[j]);
                        end
            for(k=22;k<24;k=k+1)
                        begin:b5
```

```
                    Mux M3(a[k] , 1'b0 , Shift[1] , a1[k]);
                end
        end
endgenerate

genvar p , q;
wire [23:0]a2;

generate
        begin:b6
                for(p=0;p<20;p=p+1)
                        begin:b7
                         Mux M4(a1[p] , a1[p+4] , Shift[2] , a2[p]);
                        end
                for(k=20;k<24;k=k+1)
                        begin:b8
                         Mux M5(a1[k] , 1'b0 , Shift[2] , a2[k]);
                        end
        end
endgenerate

genvar x , y;
wire [23:0]a3;

generate
        begin
                for(x=0;x<16;x=x+1)
                 begin:b9
                  Mux M6(a2[x] , a2[x+8] , Shift[3] , a3[x]);
                 end
                for(y=16;y<24;y=y+1)
                 begin:b10
                  Mux M7(a2[y] , 1'b0 , Shift[3] , a3[y]);
```

```
                end
        end
endgenerate

genvar s , t;
wire [23:0]a4;

generate
        begin:b11
        for(s=0;s<8;s=s+1)
                begin:b12
                 Mux M8(a3[s] , a3[s+4] , Shift[4] , a4[s]);
                end
        for(t=8;t<24;t=t+1)
                begin:b13
                 Mux M9(a3[t] , 1'b0 , Shift[4] , a4[t]);
                end
        end
endgenerate

assign Out = a4;

endmodule
```

```
module Complement2s(

    input [7:0] A,
    output [7:0] Out
    );

wire [7:0]w;
genvar j;
 generate
  begin
   for(j=0;j<8;j=j+1)
    begin
     Mux M(1'b1,1'b0,A[j],w[j]);
    end
  end
endgenerate

wire [7:0]c;
HalfAdder H1(w[0],1'b1,Out[0],c[0]);

genvar i;
        generate
                begin
                        for(i=1;i<8;i=i+1)
                          begin
                            FullAdder F1(w[i],1'b0,c[i-1],Out[i],c[i]);
                          end
                end
        endgenerate

endmodule
```

Binary representation of 5 is: 0 1 0 1

1's Complement of 5 is: 1 0 1 0

2's Complement of 5 is: (1's Complement + 1) i.e.

1 0 1 0 (1's Compliment)

\+ 1

1 0 1 1 (2's Complement i.e. -5)

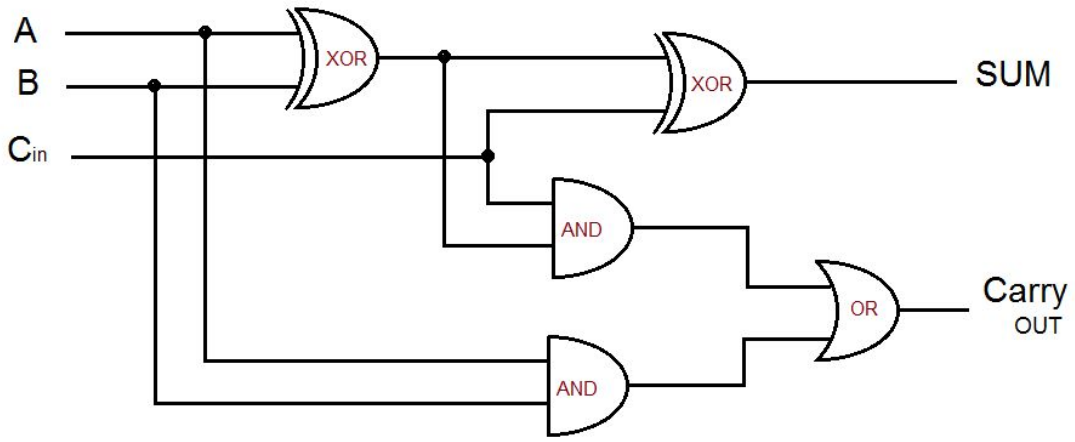
Example to compute 2’s complement

```
module ControlledIncrementor(

    input A,
    input [7:0] Z,
    output [7:0] Out
    );

wire [7:0]w;
wire [7:0]Cin;

assign w =(A==1'b1)?1:0;

HalfAdder H(Z[0],w[0],Out[0],Cin[0]);

genvar j;
generate
 begin
  for(j=1;j<8;j=j+1)
   begin
   FullAdder F1(Z[j],w[j],Cin[j-1],Out[j],Cin[j]);
   end
 end
endgenerate

endmodule
```

//

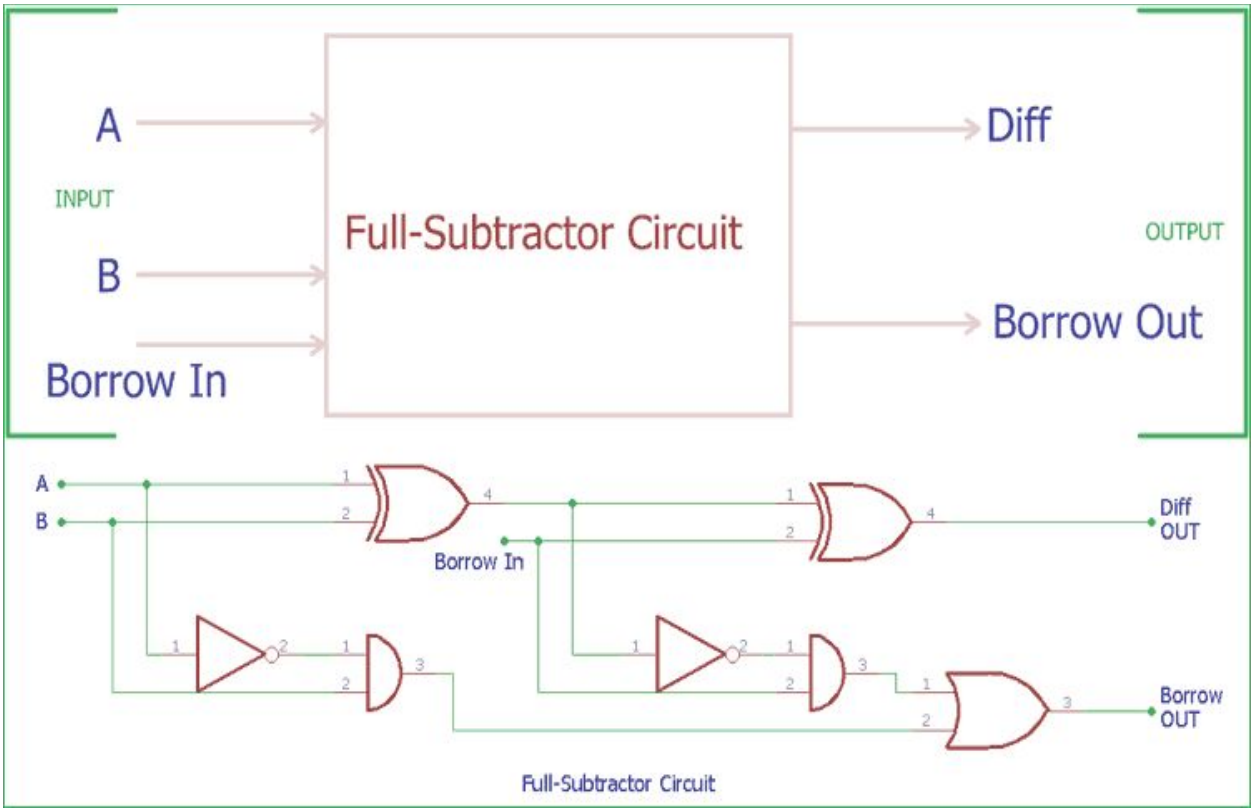
```
module FullAdder(
   input A,
   input B,
   input Cin,
   output S,
   output Cout
   );

wire w1,w2,w3;
xor(w1,A,B);
xor(S,w1,Cin);
and(w2,w1,Cin);
and(w3,A,B);
or(Cout,w3,w2);

endmodule
```

Structural implementation of Full Adder



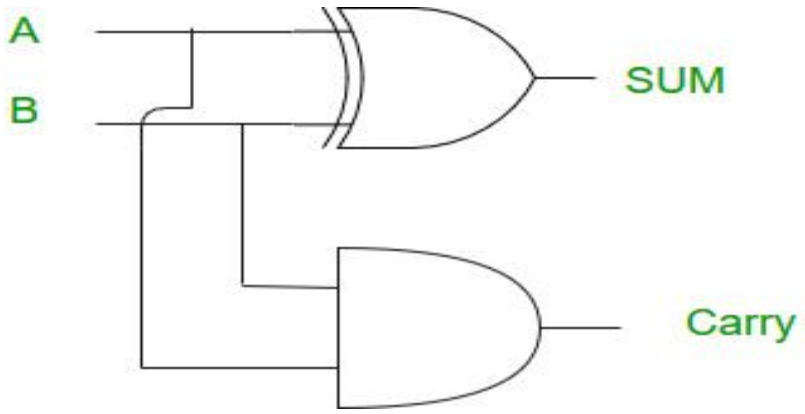
//

```
module FullSubtractor(

    input A,
    input B,
    input Bin,
    output D,
    output Bout
    );

wire w1,w2,w3,w4;

xor(w1,A,B);
xor(D,w1,Bin);

and(w2,~A,~B,Bin);
and(w3,A,~B,~Bin);
and(w4,~A,B,~Bin);
and(w5,A,B,Bin);

or(Bout,w2,w3,w4,w5);

endmodule
```



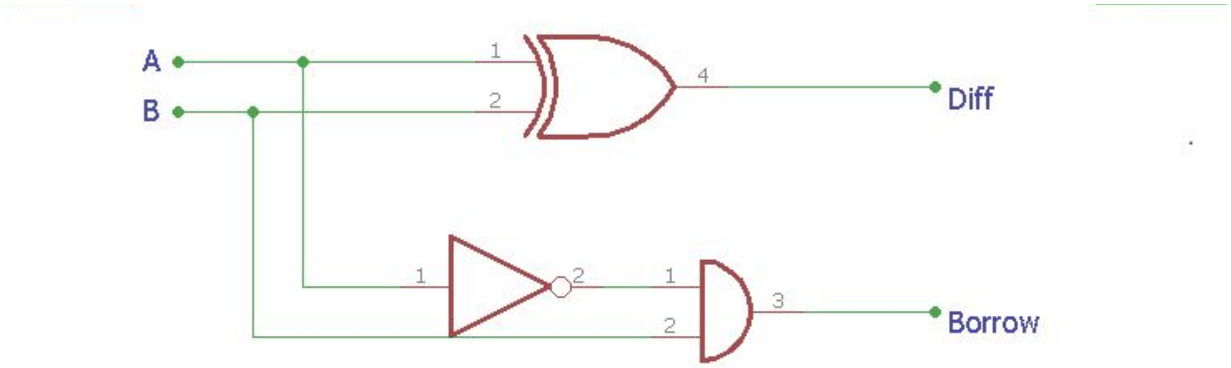
Structural Implementation of Full Subtractor Circuit

```
module HalfAdder(

    input A,
    input B,
    output S,
    output C

    );

xor(S,A,B);
and(C,A,B);

endmodule
```



Structural Implementation of Half Adder

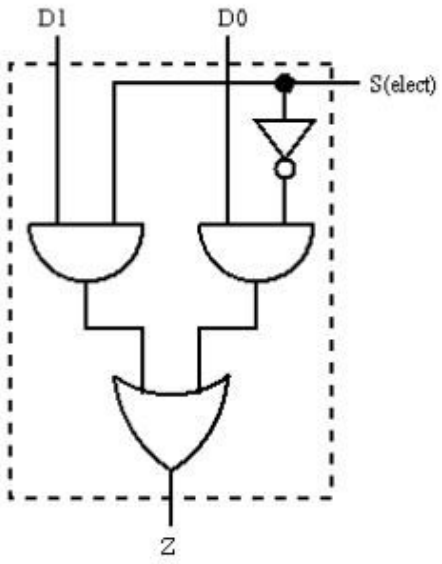
```
module HalfSubtractor(
    input A,
    input B,
    output D,
    output Bout
    );

xor(D,A,B);
and(Bout,~A,B);

Endmodule
```



Half-Subtractor Circuit

Structural Implementation of Half Subtractor Circuit

```
module Mux(

    input In0,
    input In1,
    input S,
    output Out
    );

wire w1,w2;

and(w1,~S,In0);
and(w2,S,In1);
or (Out,w1,w2);

endmodule
```



Structural Implementation of 2x1 Multiplexer

```
module Mux24(

    input [22:0] A,
    input [22:0] B,
    input S,
          output [23:0] Out
    );

genvar i;

generate
 begin
  for(i=0;i<23;i = i + 1)
    Mux M(A[i],B[i],S,Out[i]);
  end
endgenerate

assign Out[23]=1'b1;

endmodule
```

//

```
module Mux_8(
   input [7:0] A,
   input [7:0] B,
   input C,
   output [7:0] Out
   );

 genvar j;

 generate
  begin
   for(j=0;j<=7;j=j+1)
    begin
     Mux M(A[j],B[j],C,Out[j]);
    end
   end
endgenerate

endmodule
```

//

```
module Subtractor_8Bit(

    input [7:0] A,
    input [7:0] B,
    output [7:0] D,
    output Bout
    );

wire [7:0] w;

HalfSubtractor H1(A[0],B[0],D[0],w[0]);
FullSubtractor F1(A[1],B[1],w[0],D[1],w[1]);
FullSubtractor F2(A[2],B[2],w[1],D[2],w[2]);
FullSubtractor F3(A[3],B[3],w[2],D[3],w[3]);
FullSubtractor F4(A[4],B[4],w[3],D[4],w[4]);
FullSubtractor F5(A[5],B[5],w[4],D[5],w[5]);
FullSubtractor F6(A[6],B[6],w[5],D[6],w[6]);
FullSubtractor F7(A[7],B[7],w[6],D[7],w[7]);

assign Bout=w[7];

endmodule
```

//

```
module Sub_Result(

    input [7:0] A,
    input [7:0] B,
    output [7:0] Out,
          output b
    );

wire [7:0]d;
wire [7:0]d1;

Subtractor_8Bit S(A,B,d,b);

Complement2s C(d, d1);

assign Out=(b == 1'b1)?d1:d;

endmodule
```

//

```
module FIFO(
   input [31:0] A,
   input Clk,
   output [31:0] Q
   );
 genvar i,j;
 wire [31:0] w1[0:8];
 wire [31:0] w0[0:8];

 for(j=0;j<32;j=j+1)
  begin :a1
    assign w1[0][j]=A[j];
  end

  generate
   for(i=0;i<8;i=i+1)
    begin:a2
      for(j=0;j<32;j=j+1)
        begin:a3
         Dflipflop d(w1[i][j],Clk,w1[i+1][j],w0[i+1][j]);
        end
      end
  endgenerate

  for(j=0;j<32;j=j+1)
    begin:a4
     assign Q[j]=w1[8][j];
    end

endmodule
```

//

*not incorporated in the main module

RESULTS

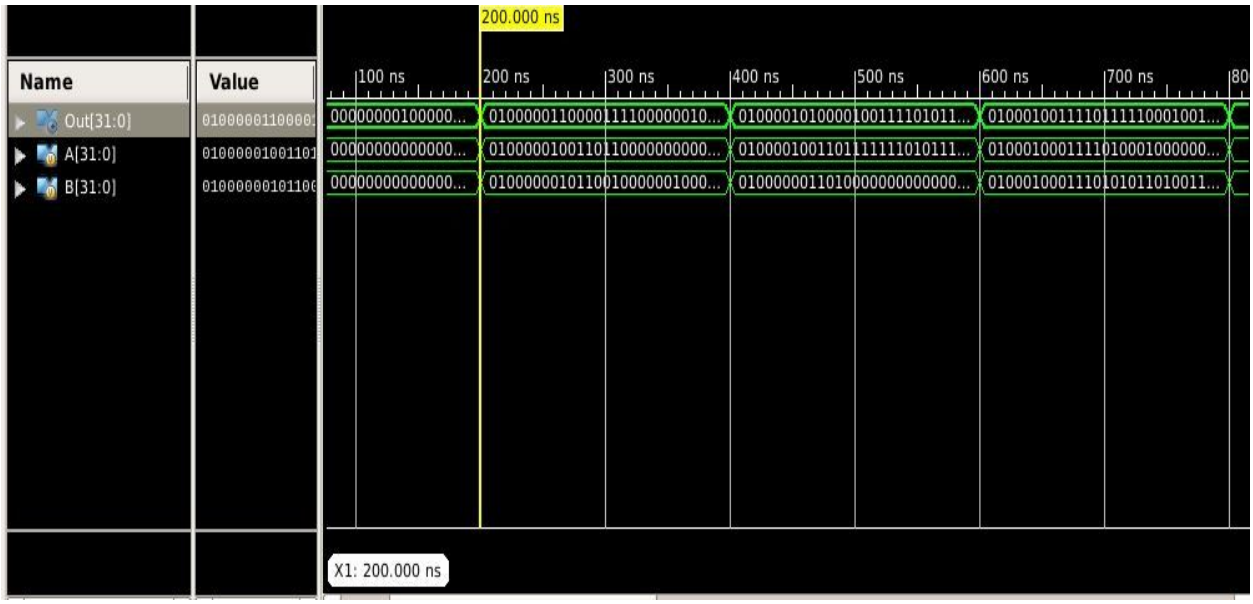
Test Bench

module Test;

```
// Inputs
reg [31:0] A;
reg [31:0] B;

// Outputs
wire [31:0] Out;

// Instantiate the Unit Under Test (UUT)
FloatingPointAdder uut (
        .A(A),
        .B(B),
        .Out(Out)
);

initial begin
        // Initialize Inputs
        A = 0;
        B = 0;

        // Wait 200 ns for global reset to finish
        #200;
         A=32'b01000001001101100000000000000001;   //11.375
         B=32'b01000000101100100000010000011011;   //5.56300
         //Expected Sum=16.938
         $monitor("Output:  %b ",Out);
```

```
        #200
         A=32'b01000010011011111110101110000101;      //59.979
         B=32'b01000000110100000000000000000000;      //6.5
         //Expected Sum=66.479
         $monitor("Output:  %b ",Out);


        #200
         A=32'b01000100011110100010000000000000;      //1000.5
         B=32'b01000100011101010110100111011011;      //981.654
         //Expected Sum=1982.1539
         $monitor("Output:  %b ",Out);


        #200
         A=32'b01000100000010010111111100101011;      //549.987
         B=32'b01000000101100100000010000011001;      //5.563
         //Expected Sum=555.499
         $monitor("Output:  %b ",Out);


    end

endmodule




////////////////////////////////////////////////
```

Simulation



Output

```
Finished circuit initialization process.
Output:  01000001100001111000000100000111
Output:  01000010100001001111010111000010
Output:  01000100111101111100010011101101
Output:  01000100000010101110001100110011
ISim>
```

DISCUSSIONS AND DIFFICULTIES

- Single precision floating point adder provides less precise results than its' double precision counterpart.
- We were able to implement the 8 Bit FIF0 as an individual module but were not able to incorporate it in our floating point module. This was mainly because of our inability to account for the elapsed 8 clock pulses after the arrival of any one input.
- The calculated values of the output matched the expected values to a great precision.
- The expected values of the floating point adder were obtained from online IEEE 754 converter which were comparatively difficult to obtain by hand.

REFERENCES

- http://www.binaryconvert.com/convert_float.html?hexadecimal=41878107
- https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/
- http://cse.iitkgp.ac.in/~goutam/pds/pdsLect/lect15.pdf
- https://en.wikipedia.org/wiki/Single-precision_floating-point_format