Laboratory 3:

FLOATING POINT ADDER/SUBTRACTOR

OBJECTIVES

- Getting to know how to describe a floating-point arithmetic, which could do add, subtract, and multiply operations.
- Design and implement digital circuits using FSM.
- Download the circuit into the FPGA chip and test its functionality.

PREPARATION FOR LAB 3

- Finish Pre Lab 3 at home.
- Students have to simulate all the exercises in Pre Lab 3 at home. All results (codes, waveform, RTL viewer, …) have to be captured and submitted to instructors prior to the lab session. *If not, students will not participate in the lab and be considered absent this session.*

REFERENCE

Laboratory 3:
FLOATING POINT ADDER/SUBTRACTOR

Floating point numbers allow computers to perform operations on a wide range of numbers. According to the IEEE standards, floating point numbers are of the form

$$(-1)^S * (1+F) *2^E$$

S is the sign bit, which determines whether the number is positive or negative.

F – fraction – holds the significant bits of the floating point number.

E is the exponent that (1+F) is raised to.

A 32 bit floating point number is standard, however fo simpification, we will be using an 8 bit representation. We will have a sign bit, 3 bits for the exponent, and the remaining 4 bits will be devoted to the fraction. This will allow us to represent a resolution as small as 1/128 and the smallest number we can represent is 1/8. Eight-bit floating point numbers are not useful for performing extremely accurate calculations, but it does demonstrate the operation of a floating point adder.

Laboratory 3:

FLOATING POINT ADDER/SUBTRACTOR

EXPERIMENT

Objective: Implement a floating point adder/subtractor using System Verilog description and download the cicuit into the FPGA chip..

Requirement:

You are desired to design a floating-point adder/subtractor should perform correctly in the normal cases. Besides several pins needs to indicate some extreme cases:

- Zero detection: when the result is zero, zero detection pin will be 1.

Inputs	Operand 1	A[7:0]	8-bit normalized input
	Operand 2	B[7:0]	8-bit normalized input
	Selection	S	1-bit input, addition/subtraction selection
Outputs	Operation result	Result[7:0]	8-bit normalized output
	Zero detection	Z	1-bit ouput

Table 1: IO definition

Instruction:

Input A and B have 8 bits, in which the sign is represented by bit [7], the exponent value is represented by bit [6:4], and the remaining is for Fraction value. Output Result also need to be normalized as input signal.

The design have some sub-modules that perform floating point calculations:

- Identify which number is larger, which number is smaller.
- Indentify the amount to right shift the operand which has smaller exponent.
- Right shift fraction value of the smaller operand to align decimal points.
- Calculate the two's compliment of the shifted fraction, only needed in the case of subtraction or equivalent case (i.e. adding a negative number to a positive number).
- Add the two fractions together.
- Normalize the fraction and exponent value so it's back in floating point representation.
- Determine sign of the final value.
- Detect zero: the result is zero if the signs of A and B are different and there is no difference in the fraction and exponent.

Laboratory 3:

FLOATING POINT ADDER/SUBTRACTOR

1. Create a new Quartus project for your circuit.
2. Use switches as inputs, LEDRs as outputs.
3. Compile your project. Download the circuit into the FPGA chip and test its functionality.

Check: Your report has to show two results:

The Code:

```
module Exponent(
input [2:0] Exp_A, Exp_B,
output [2:0] Bigger_exp,
output [3:0] Exp_diff
);

assign Exp_diff = Exp_A - Exp_B;

always_comb begin
        if (Exp_diff[3] == 0)
                Bigger_exp <= Exp_A;
        else
                Bigger_exp <= Exp_B;
end

endmodule

module add_sub_fract(
input Sign_A, Sign_B, S,
input [5:0] Shifted_fract, SixBitBigger_fract,
output [5:0] Mag_fract
);

logic Same_sign;

assign Same_sign = Sign_A ~^ Sign_B;

always_comb begin
        if (S == 1) begin
                if (Same_sign == 1)
                        Mag_fract <= SixBitBigger_fract - Shifted_fract;
                else
                        Mag_fract <= SixBitBigger_fract + Shifted_fract;
                end
        else begin
                if (Same_sign == 1)
                        Mag_fract <= SixBitBigger_fract + Shifted_fract;
```

```
                  else
                            Mag_fract <= SixBitBigger_fract - Shifted_fract;
                  end
end

endmodule
```

```
module Exp_Fract(
input [5:0] Mag_fract,
input [2:0] Bigger_exp,
output [3:0] Result_fract,
output [2:0] Result_exp,
output OV
);

always_ff begin
        if (Mag_fract[5] == 1) begin
                  if (Bigger_exp <= 6) begin
                            Result_exp <= Bigger_exp + 1;
                            Result_fract <= Mag_fract[4:1];
                            OV <= 1'b0;
                            end
                  else begin
                            OV <= 1'b1;
                            Result_exp <= 3'b111;
                            Result_fract <= 4'b1111;
                            end
                  end
        else if (Mag_fract[4] == 1) begin
                  Result_exp <= Bigger_exp;
                  Result_fract <= Mag_fract[3:0];
                  OV <= 1'b0;
                  end
        else if (Mag_fract[3] == 1) begin
                  if (Bigger_exp >= 1) begin
                            Result_exp <= Bigger_exp - 1;
                            Result_fract <= {Mag_fract[2:0] , 1'b0};
                            OV <= 1'b0;
                            end
                  else begin
                            OV <= 1'b1;
                            Result_exp <= 3'b111;
                            Result_fract <= 4'b1111;
                            end
```

```
                end
        else if (Mag_fract[2] == 1) begin
                if (Bigger_exp >= 2) begin
                        Result_exp <= Bigger_exp - 2;
                        Result_fract <= {Mag_fract[1:0] , 2'b0};
                        OV <= 1'b0;
                        end
                else begin
                        OV <= 1'b1;
                        Result_exp <= 3'b111;
                        Result_fract <= 4'b1111;
                        end
                end
        else if (Mag_fract[1] == 1) begin
                if (Bigger_exp >= 3) begin
                        Result_exp <= Bigger_exp - 3;
                        Result_fract <= {Mag_fract[0] , 3'b0};
                        OV <= 1'b0;
                        end
                else begin
                        OV <= 1'b1;
                        Result_exp <= 3'b111;
                        Result_fract <= 4'b1111;
                        end
                end
        else if (Mag_fract[0] == 1) begin
                if (Bigger_exp >= 4) begin
                        Result_exp <= Bigger_exp - 4;
                        Result_fract <= 4'b0;
                        OV <= 1'b0;
                        end
                else begin
                        OV <= 1'b1;
                        Result_exp <= 3'b111;
                        Result_fract <= 4'b1111;
                        end
                end
        else OV <= 1'b0;
        end

endmodule
```

module Exponent(

```
input [2:0] Exp_A, Exp_B,
output [2:0] Bigger_exp,
output [3:0] Exp_diff
);

assign Exp_diff = Exp_A - Exp_B;

always_comb begin
        if (Exp_diff[3] == 0)
                Bigger_exp <= Exp_A;
        else
                Bigger_exp <= Exp_B;
end

endmodule
```

module Fraction(

```
input [3:0] Fract_A, Fract_B, Exp_diff,
output [3:0] Bigger_fract, Smaller_fract
);

logic [3:0] Fract_diff;
assign Fract_diff = Fract_A - Fract_B;

always_comb begin
        if (Exp_diff == 0) begin                        //Exp_A = Exp_B
                if (Fract_diff[3] ==  0) begin
                        Bigger_fract <= Fract_A;
                        Smaller_fract <= Fract_B;
                end
                else begin
                        Bigger_fract <= Fract_B;
                        Smaller_fract <= Fract_A;
                end
        end

        else begin                                                      //Exp_A <> Exp_B
                if (Exp_diff[3] == 0) begin
                        Bigger_fract <= Fract_A;
                        Smaller_fract <= Fract_B;
                end
                else begin
                        Bigger_fract <= Fract_B;
                        Smaller_fract <= Fract_A;
                end
        end
end

endmodule
```

module Result(

```
input [7:0] A, B,
input Result_sign, S,
input [2:0] Result_exp,
input [3:0] Result_fract,
output Z,
output [7:0] Result
);

always_comb begin
        if (A[6:0] == B[6:0] && A[7] != B[7] && S == 0) begin
                Z <= 1'b1;
                Result <= '0;
                end
        else begin
                Z <= 1'b0;
                Result <= {Result_sign, Result_exp, Result_fract};
                end
end

endmodule
```

module Shift(

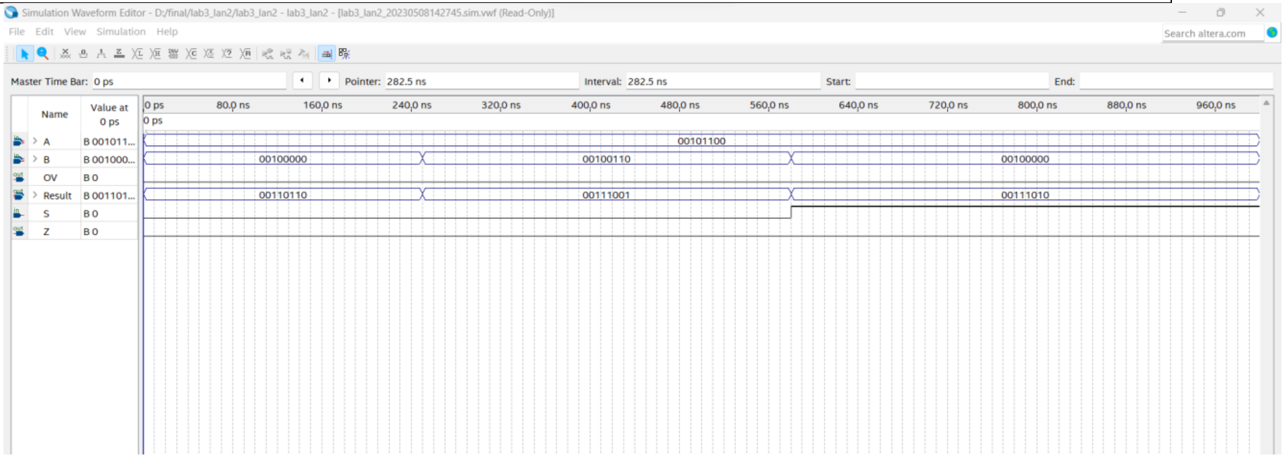
```
input  logic [3:0] Smaller_fract, Exp_diff,
output logic [5:0] Shifted_fract
);

logic [3:0] Shift_amount;
logic [4:0] Lead1;

always_comb begin
        if (Exp_diff[3] == 0)
                Shift_amount = Exp_diff;
        else
                Shift_amount = -Exp_diff;
end

assign Lead1 = {1'b1, Smaller_fract};
assign Shifted_fract = Lead1 >> Shift_amount;
```

```
endmodule

module Sign(
input [2:0] Exp_A, Exp_B,
input [3:0] Fract_A, Fract_B,
input Sign_A, Sign_B,
output Result_sign
);


always_comb begin
        if (Exp_A == Exp_B) begin
                if (Fract_B > Fract_A) Result_sign <= Sign_B;
                else Result_sign <= Sign_A;
        end

        else begin
                if (Exp_B > Exp_A) Result_sign <= Sign_B;
                else Result_sign <= Sign_A;
        end
end

endmodule

module lab3_lan2(
input S,
input [7:0] A, B,
output [7:0] Result,
output OV,
output Z
);

logic Sign_A, sign_B;
logic [2:0] Exp_A, Exp_B;
logic [3:0] Fract_A, Fract_B;
logic [2:0] Bigger_exp;
logic [3:0] Exp_diff;
logic [3:0] Bigger_fract, Smaller_fract;
logic [5:0] Shifted_fract;
logic [5:0] SixBitBigger_fract;
logic [5:0] Neg_Shifted_fract;
logic [5:0] Mag_fract;
logic [3:0] Result_fract;
logic [2:0] Result_exp;
logic Result_sign;

assign Sign_A = A[7];
assign Exp_A = A[6:4];
assign Fract_A = A[3:0];
assign Sign_B = B[7];
assign Exp_B = B[6:4];
assign Fract_B = B[3:0];

Exponent
exponent(.Exp_A(Exp_A),.Exp_B(Exp_B),.Bigger_exp(Bigger_exp),.Exp_diff(Exp_diff));

Fraction
fraction(.Fract_A(Fract_A),.Fract_B(Fract_B),.Exp_diff(Exp_diff),.Bigger_fract(Bigger_fract),.S
maller_fract(Smaller_fract));

Shift shift(.Smaller_fract(Smaller_fract),.Exp_diff(Exp_diff),.Shifted_fract(Shifted_fract));

assign SixBitBigger_fract = {2'b01, Bigger_fract};
assign Neg_Shifted_fract = -Shifted_fract;

add_sub_fract
addsub(.Sign_A(Sign_A),.Sign_B(Sign_B),.S(S),.Shifted_fract(Shifted_fract),.SixBitBigger_fract
(SixBitBigger_fract),.Mag_fract(Mag_fract));

Exp_Fract
expfract(.Mag_fract(Mag_fract),.Bigger_exp(Bigger_exp),.Result_fract(Result_fract),.Result_exp
(Result_exp),.OV(OV));

Sign
sign(.Sign_A(Sign_A),.Sign_B(Sign_B),.Exp_A(Exp_A),.Exp_B(Exp_B),.Fract_A(Fract_A),.Fra
ct_B(Fract_B),.Result_sign(Result_sign));

Result
result(.A(A),.B(B),.S(S),.Result_sign(Result_sign),.Result_exp(Result_exp),.Result_fract(Result_
fract),.Result(Result),.Z(Z));

endmodule
```

- The waveform to prove the circuit works correctly.

Laboratory 3:

FLOATING POINT ADDER/SUBTRACTOR

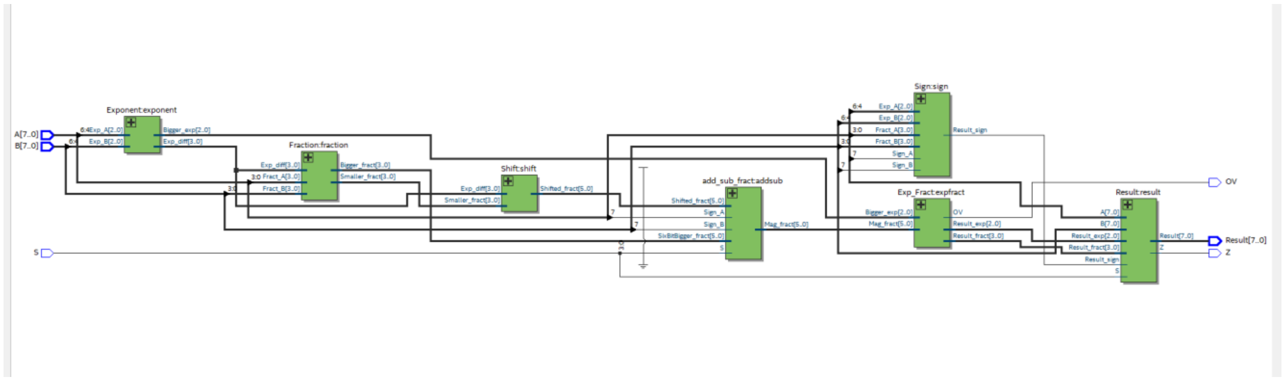


Simulation Waveform Editor - D:/final/lab3_lan2/lab3_lan2 - lab3_lan2 - [lab3_lan2_20230508142745.sim.vwf (Read-Only)]

Master Time Bar: 0 ps Pointer: 282.5 ns Interval: 282.5 ns Start: End:

Name	Value at 0 ps	Waveform
A	B 001011...	00101100
B	B 001000...	00100000, 00100110, 00100000
OV	B 0	
Result	B 001101...	00110110, 00111001, 00111010
S	B 0	
Z	B 0	

- The result of RTL viewer.



A[7..0]
B[7..0]
S
Exponent:exponent
Fraction:fraction
Shift:shift
add_sub_fract:addsub
Sign:sign
Exp_Fract:expfract
Result:result
OV
Result[7..0]
Z