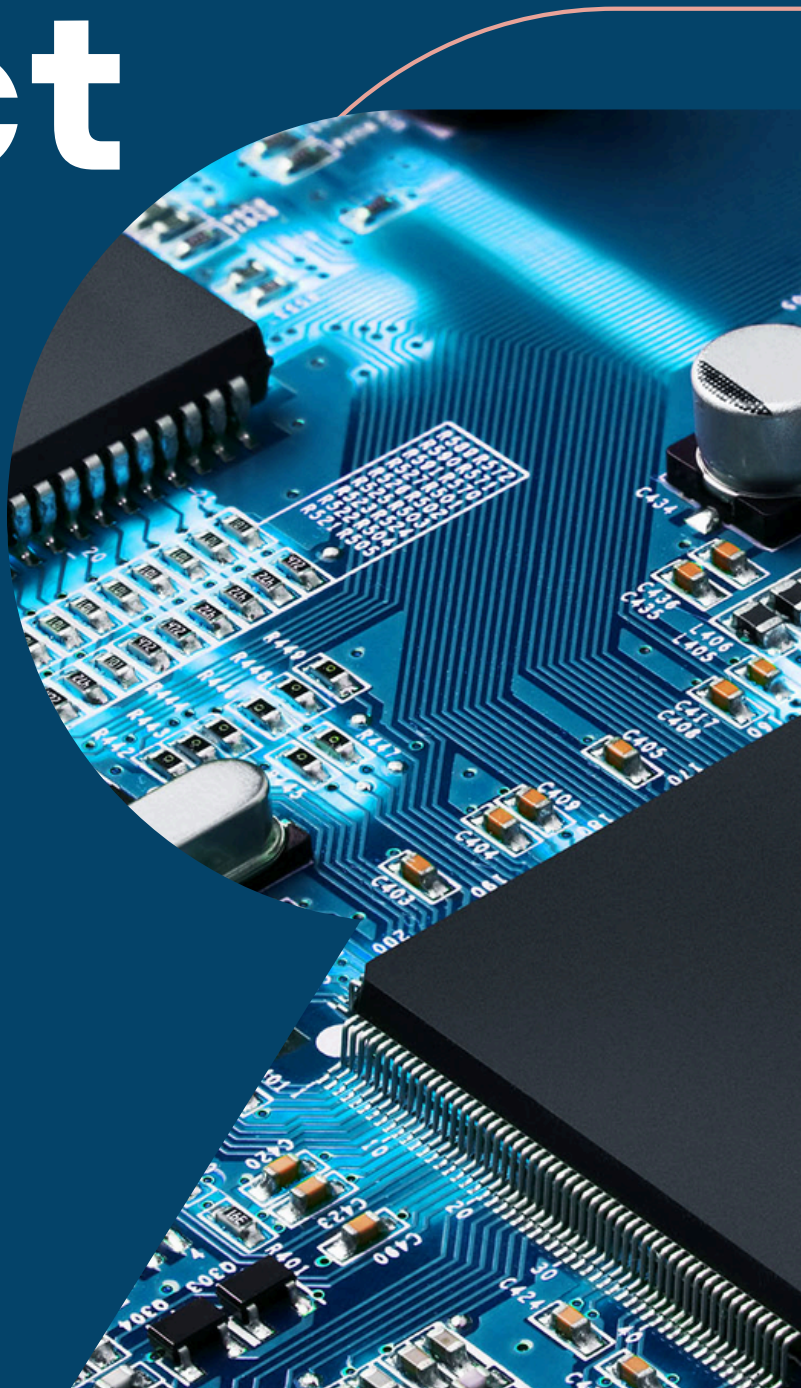
Computer Arithmetic Training

Final Project

STMICROELECTRONICS



SUBMITTED TO

Instructor:

Eng. Ahmed Abdelsalam

SUBMITTED BY

Ahmad Essam AbdelAty

Date: 3/4/2025

Table of Contents

01 Introduction

In this Report, I will go over the process of creating this Floating Point Adder / Subtractor, the purpose of each module, and how i handled the Special Cases(Infinity, NaN, etc.).

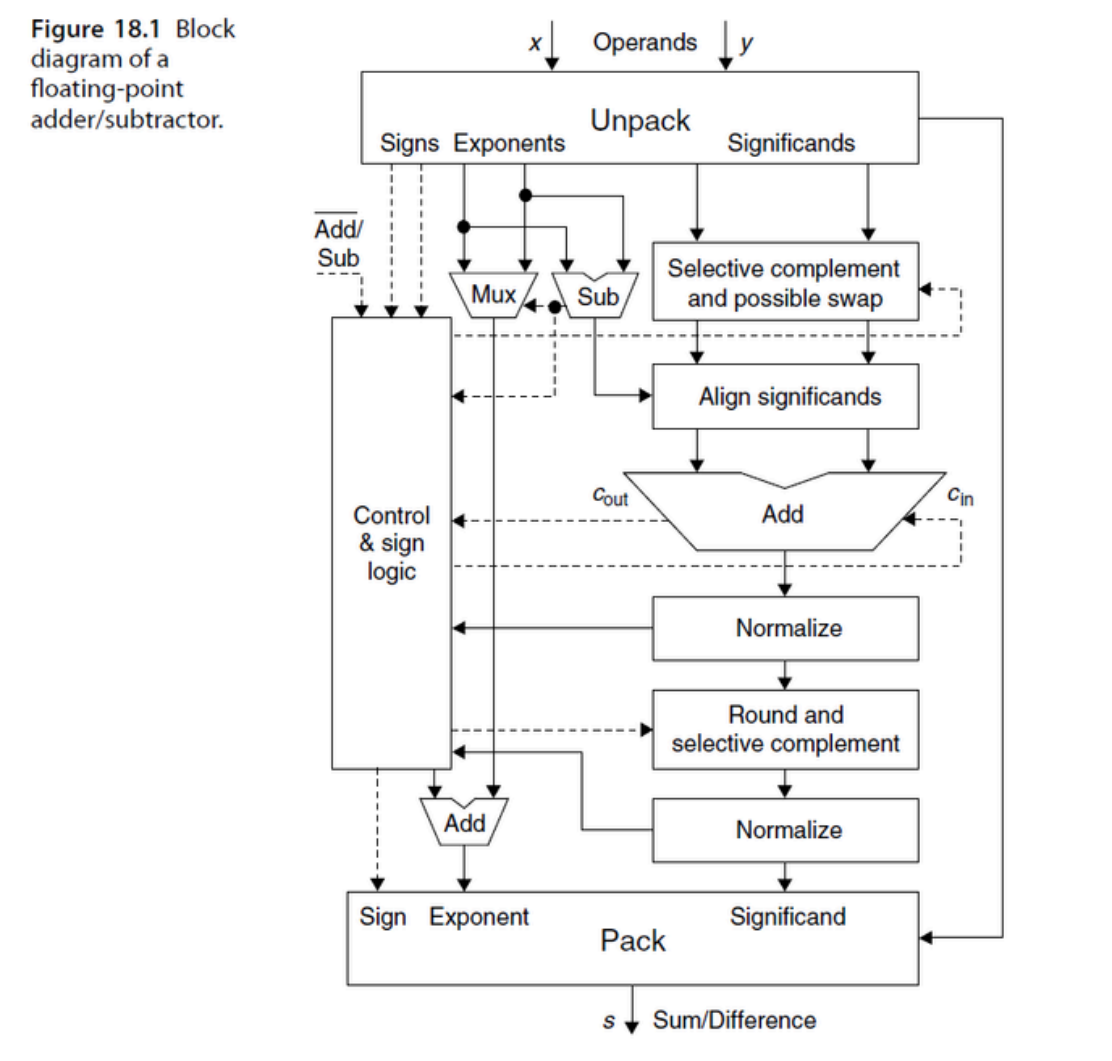


Figure 18.1 Block diagram of a floating-point adder/subtractor.

02 Unpack Module

This is the first module in the architecture, it is responsible for unpacking the input IEEE 754 single precision operands into: signs, exponents, and significands. it also detects whether the significands are equal, whether they are non-zero, and whether one of the exponents is stacked, i.e whether it is equal to 8'b11111111. When assigning the significands, we also include the implicit/hidden one, and the sign bit.

```
(*DONT_TOUCH = "TRUE"*) module Unpack (
    input [31:0] operand_1, operand_2,
    output operand_1_sign, operand_2_sign, significand_equal, sig_non_zero, exp_stacked,
    output [7:0] operand_1_exponent, operand_2_exponent,
    output [24:0] operand_1_significand, operand_2_significand
);

    assign operand_1_sign = operand_1[31];
    assign operand_2_sign = operand_2[31];

    assign operand_1_exponent = operand_1[30:23];
    assign operand_2_exponent = operand_2[30:23];

    assign operand_1_significand = {1'b0, 1'b1, operand_1[22:0]};
    assign operand_2_significand = {1'b0, 1'b1, operand_2[22:0]};

    assign significand_equal = (operand_1[22:0] == operand_2[22:0]);

    assign sig_non_zero = |(operand_1[22:0] ) | |(operand_2[22:0]);

    assign exp_stacked = &(operand_1_exponent) | &(operand_2_exponent);

endmodule
```

03 Exponent Sub Module

The following module is the Exponent Subtraction Module, it is responsible for comparing the exponents of the inputs to determine whether we need to align significands later on, if the exponent of the 1st operand is less than the 2nd , it outputs a signal called "sign" telling the alignment module to switch the 2 operands, as the shifter is only available for the 2nd operand(more on that later). Finally, like in the Unpack module, i output a signal to determine whether the exponent of the larger operand is non-zero.

```
(*DONT_TOUCH = "TRUE"*) module Exponent_Sub (
    input [7:0] operand_1_exponent, operand_2_exponent,
    output reg [7:0] result,
    output reg sign, exp_non_zero
);

    always @(*) begin
        if (operand_1_exponent > operand_2_exponent) begin
            result = operand_1_exponent - operand_2_exponent;
            sign = 0;
        end
        else begin
            result = operand_2_exponent - operand_1_exponent;
            sign = 1;
        end

        exp_non_zero = |(operand_1_exponent);
    end

endmodule
```

04 SCPS Module

SCPS(Selective Complement and Possible Swap) Module is responsible for Complementing the 1st Operand of the operation is a subtraction and to swap the operands if the exponent of the 2nd Operand is greater than that of the 1st.

```
(*DONT_TOUCH = "TRUE"*) module SCPS ( //Selective Complement and Possible Swap
    input [24:0] operand_1_significand, operand_2_significand,
    input swap, comp,
    output reg [24:0] operand_1_significand_result, operand_2_significand_result
);

    always @(*) begin
        if (swap) begin
            operand_1_significand_result = operand_2_significand;
            operand_2_significand_result = operand_1_significand;
        end
        else begin
            operand_1_significand_result = operand_1_significand;
            operand_2_significand_result = operand_2_significand;
        end

        if (comp) begin
            operand_1_significand_result = ~operand_1_significand_result;
        end
    end

endmodule
```

05

Align Significands Module

The inputs of this module are the significands of the operands, the Shift Amount coming from the Exponent Sub Module, and finally complement signal to extend the 1st Operand. I then declare five reg's to store the shifted operand in at each stage of the Shifter, and 3 wires for the rounding module.

Inputs and Reg Declaration

```
(*DONT_TOUCH = "TRUE"*) module Align_Significands (
    input comp,
    input [24:0] operand_1_significand, operand_2_significand,
    input [7:0] shamt,
    output [27:0] operand_1_significand_shifted, operand_2_significand_shifted
);

    reg [25:0] w1;
    reg [27:0] w2;
    reg [31:0] w3;
    reg [39:0] w4;
    reg [55:0] w5;

    wire guard, round, sticky;
```

05

Align Significands Module

This is the Shifter, it works as follows, it consists of 5 cascading 2x1 multiplexers, each level of multiplexers has the following inputs: either the same output of the multiplexer before it, or this same output but shifted right by 2^(the number of level), so for the first level it would be shifted by 2^0 = 1, for the second level it would be shifted by 2^1 = 2, so on and so forth.

Shifter

```
always @(*) begin   // Complemented Operand 1 as the textbook stated
    if (shamt[0]) begin //shift by 1
        w1 = {operand_2_significand[24], operand_2_significand};
    end
    else begin
        w1[25:1] = operand_2_significand;
        w1[0] = 0;
    end

    if (shamt[1]) begin //shift by 2
        w2 = {{2{w1[25]}}, w1};
    end
    else begin
        w2[27:2] = w1;
        w2[1:0] = 0;
    end

    if (shamt[2]) begin //shift by 4
        w3 = {{4{w2[27]}}, w2};
    end
    else begin
        w3[31:4] = w2;
        w3[3:0] = 0;
    end

    if (shamt[3]) begin //shift by 8
        w4 = {{8{w3[31]}}, w3};
    end
    else begin
        w4[39:8] = w3;
        w4[7:0] = 0;
    end

    if (shamt[4]) begin //shift by 16
        w5 = {{16{w4[39]}}, w4};
    end
    else begin
        w5[55:16] = w4;
        w5[15:0] = 0;
    end
end
```

05

Align Significands Module

This is the Output Logic of this module, we assign values for the 3 wires responsible for the rounding scheme: Guard, Round, and Sticky bits. The Guard bit is the bit right after the LSB of the Mantissa, the Round bit is the bit just after it, and the Sticky bit is the result of ORing all the remaining bits with each other. Then, we prepare the output of the aligned significand of the 2nd Operand, each being 28 bits: first is the sign bit, next is the implicit one, after that is the Mantissa, and finally the Guard, Round, and Sticky bits. If the shift amount is greater than 31 bits, the output of the 2nd Operand's significand is zero, as all the bits are already shifted out of the Mantissa. For the 1st Operand, we just insert the input 1st Operand significand as is in the most significant 25 bit, and concatenate with it 3 bits based on whether the input is complemented or not, which is signified by the "comp" signal.

Output Logic

```
assign guard = w5[30];
assign round = w5[29];
assign sticky = |(w5[28:0]);

assign operand_1_significand_shifted = {operand_1_significand, {3{comp}}};
assign operand_2_significand_shifted = (shamt[7:5])? 0 : {w5[55:31], guard, round, sticky};
```

06 CPA Module

The Carry Propagate Adder (CPA) is responsible for the Addition of the aligned operands, i decided to implement it using the Parallel Prefix Adder I previously made, the 32-bit Ladner-Fischer Adder, as it has a delay of log(n), where n is the number of bits, i only utilized the first 29 bits, the first bit for the Carry in, the next 28 bits are the Operands, while taking the Sum from the bits [28:1] of the output, and sign bit as the MSB of the Sum. Finally, for the Overflow flag, I compare the signs of the operands and the Sum, if both the operands are positive and the Sum is negative, then an overflow occured

Carry Propagate Adder

```
(*DONT_TOUCH = "TRUE"*) module CPA (
    input [27:0] operand_1_significand, operand_2_significand,
    input cin,
    output cout, overflow, sign_bit,
    output [27:0] sum
);

    wire [31:0] ladner_out;

    Adder_Top_Module #(32) Ladner({3'b0, operand_1_significand, cin}, {3'b0, operand_2_significand, cin}, ladner_out);

    assign cout = ladner_out[29];

    assign sum = ladner_out[28:1];

    assign sign_bit = sum[27];

    assign overflow = (operand_1_significand[27] & operand_2_significand[27] & ~sum[27]) || (~operand_1_significand[27] & ~operand_2_significand[27] & sum[27]);

endmodule
```

06 CPA Module

Ladner-Fischer Parallel Prefix Adder:

Top Module

```
(*DONT_TOUCH = "TRUE"*)module Adder_Top_Module #(
    parameter N = 32
) (
    input [N-1:0] a, b,
    output [N-1:0] s
);

    wire [N-1:0] g_in, p_in, g_out, p_out;
    genvar j;

    assign g_in = a & b;
    assign p_in = a ^ b;

    //Instantiaing Ladner-Fischer Prefix Adder
    Ladner_Fischer_Prefix_Adder #(N) LF(g_in, p_in, g_out, p_out);

    generate //Computing Sum
        for (j = 0; j < N - 1; j = j + 1) begin :Sum_Block
            assign s[j + 1] = p_in[j + 1] ^ g_out[j];
        end
        assign s[0] = p_out[0];
    endgenerate

endmodule
```

Prefix Circuit

```
(*DONT_TOUCH = "TRUE"*) module Prefix_Circuit (
    input [1:0] x0, x1,
    output [1:0] y
);

    assign y[0] = x0[0] & x1[0]; //Calculating P
    assign y[1] = (x0[1] & x1[0]) | x1[1]; //Calculating G

endmodule
```

Ladner-Fischer Parallel Prefix Adder

```
generate
    for (i = 0; i < $clog2(N); i = i + 1) begin :Level_1
        for (j = 0; j < N; j = j + (2**(i + 1))) begin :Level_2
            Prefix_Circuit PC_0(w[i][(2**(i) - 1) + j], w[0][(2**i) + j], w[i + 1][(2**i) + j]);
            for (c = 1; c < (i + 1); c = c + 1) begin :Level_3
                for (l = 0; l < (2**(c - 1)); l = l + 1) begin :Level_4
                    Prefix_Circuit PC(w[i][(2**(i) - 1) + j], w[c][(2**i) + 2**(c - 1) + l + j], w[i + 1][(2**i) + 2**(c - 1) + l + j]);
                end
            end
        end
    end
endgenerate

generate
    for (z = 0; z < N; z = z + 1) begin :generate_x
        assign w[0][z] = {g_in[z], p_in[z]};
    end
        assign s[0] = w[0][0];
        for (z = 1; z < ($clog2(N) + 1); z = z + 1) begin :generate_level
            for (r = (2**(z - 1)); r < 2**z; r = r + 1) begin :generate_bit
                assign s[r] = w[z][r];
            end
        end
        for (z = 0; z < N; z = z + 1) begin :generate_out
            assign {g_out[z], p_out[z]} = s[z];
        end
endgenerate
```

07 1st Normalize Module

This Module consists of 3 parts: Arbitrary Left Shifter, 1-Bit Right Shifter, and Leading Zeros Counter. The 1-Bit Right Shifter shifts if the MSB of the sum is one if the result is positive and vice versa, or if there is an overflow, shifting in this case we OR the Round and Sticky bits in order to not lose any precision. The Arbitrary Left Shifter takes it's shift amount value from the Leading Zeros Counter, and it also consists of cascading multiplexers like the Align Significands Module. Finally, the Leading Zeros Counter is a simple Priority Encoder Tree.

1st Normalize Module

```
(*DONT_TOUCH = "TRUE"*) module Normalize_1 (
    input [27:0] sum,
    input mode, overflow,
    output [7:0] shamt,
    output [27:0] shifted_sum
);

    wire [27:0] shifter_out;
    wire [7:0] shifter_shamt;

    Zero_One_Counter ZOC(sum, mode, overflow, shifter_shamt);

    Shifter shift(sum, shifter_shamt, shifter_out);

    assign shifted_sum = ((shifter_out[27] ^ mode) | overflow)? {1'b0, shifter_out[27:2], (shifter_out[1] | shifter_out[0])} : shifter_out;

    assign shamt = shifter_shamt - ((shifter_out[27] ^ mode) | overflow);

endmodule
```

07 1st Normalize Module

The Leading Zeros Counter is a Tree of multiple smaller priority encoder of various sizes, 32 bits, 16, bit, and 4 bits.

Leading Zeros Counter

```
(*DONT_TOUCH = "TRUE"*) module Zero_One_Counter (
    input [27:0] sum,
    input mode, overflow,
    output [7:0] shamt
);

    wire [27:0] encoder_in;
    wire [4:0] encoder_out;

    LZD_32 encoder({encoder_in[26:0], 5'b0}, encoder_out);

    assign shamt = (overflow)? 8'b0 : {3'b0, encoder_out};

    assign encoder_in = (mode)? ~sum : sum;

endmodule
```

32-Bit Encoder

```
(*DONT_TOUCH = "TRUE"*) module LZD_32 (
    input [31:0] in,
    output [4:0] out
);

    wire a0, a1;
    wire [3:0] y0, y1;

    //Instantiation
    LZD_16 inst0_16(in[31:16], a0, y0);
    LZD_16 inst1_16(in[15:0], a1, y1);

    assign out[4] = a0;

    assign out[3:0] = (a0)? y1 : y0;

endmodule
```

16-Bit Encoder

```
(*DONT_TOUCH = "TRUE"*) module LZD_16 (
    input [15:0] in,
    output all_zero,
    output [3:0] out
);

    wire a0, a1, a2, a3, dummy;
    wire [1:0] y0, y1, y2, y3, y4;

    //Instantiation
    LZD_4 inst0_4(in[15:12], a0, y0);
    LZD_4 inst1_4(in[11:8], a1, y1);
    LZD_4 inst2_4(in[7:4], a2, y2);
    LZD_4 inst3_4(in[3:0], a3, y3);

    LZD_4 instf_4({!a0, !a1, !a2, !a3}, dummy, y4);

    assign out[3:2] = y4;

    assign out[1:0] = (y4[1])? ((y4[0])? y3 : y2) : ((y4[0])? y1 : y0);

    assign all_zero = ~|(in);

endmodule
```

4-Bit Encoder

```
(*DONT_TOUCH = "TRUE"*) module LZD_4 (
    input [3:0] in,
    output reg all_zero,
    output reg [1:0] out
);

    always @(*) begin
        casex (in)
            4'b1xxx: out = 2'b00;
            4'b01xx: out = 2'b01;
            4'b001x: out = 2'b10;
            4'b0001: out = 2'b11;
            default: out = 2'b00;
        endcase

        all_zero = ~|(in);
    end

endmodule
```

07 1st Normalize Module

Arbitrary Left Shifter

Body

```
always @(*) begin
    if (shamt[0]) begin //shift by 1
        w1 = sum << 1;
    end
    else begin
        w1 = sum;
    end

    if (shamt[1]) begin //shift by 2
        w2 = w1 << 2;
    end
    else begin
        w2 = w1;
    end

    if (shamt[2]) begin //shift by 4
        w3 = w2 << 4;
    end
    else begin
        w3 = w2;
    end

    if (shamt[3]) begin //shift by 8
        w4 = w3 << 8;
    end
    else begin
        w4 = w3;
    end

    if (shamt[4]) begin //shift by 16
        w5 = w4 << 16;
    end
    else begin
        w5 = w4;
    end

    if (shamt[5]) begin //Shift by 32
        w6 = w5 << 32;
    end
    else begin
        w6 = w5;
    end

end
```

Port and Wire Declaration

```
(*DONT_TOUCH = "TRUE"*) module Shifter (
    input [27:0] sum,
    input [7:0] shamt,
    output [27:0] shifted_sum
);

    reg [27:0] w1;
    reg [27:0] w2;
    reg [27:0] w3;
    reg [27:0] w4;
    reg [27:0] w5;
    reg [27:0] w6;
```

Output Logic

```
assign shifted_sum = w6;
```

08 Round Module

The Round Module is responsible for making the decision whether to round up or down the sum based on the Guard, Round, Sticky bits, and most importantly, the LSB of the Mantissa. First, if the output of the CPA is negative, we complement the normalized sum to perform rounding properly and prepare the Mantissa for packing. Next, we shift if the Guard bit is 1 and either of the Round or Sticky or LSB of the Mantissa are one. Finally, we truncate the least significant 3 bits as those contained the GRS Bits, and we round up the remaining bits if the logical expression above holds true.

Round Module

```
(*DONT_TOUCH = "TRUE"*) module Round (
    input [27:0] shifted_sum,
    input sign,
    output [24:0] rounded_significand
);

    wire guard, round, sticky;
    wire round_up;
    reg [27:0] shifted_sum_comp;
    integer i;


    always @(*) begin
        shifted_sum_comp = ((sign)? ~shifted_sum : shifted_sum ) + sign;
    end

    assign guard = shifted_sum_comp[2];
    assign round = shifted_sum_comp[1];
    assign sticky = shifted_sum_comp[0];

    assign round_up = (guard & ((round | sticky) | (shifted_sum_comp[3])));

    assign rounded_significand = shifted_sum_comp[27:3] + round_up;

endmodule
```

09

2nd Normalize Module

The 2nd Normalize Module is fairly simple, when rounding, and overflow may occur, thus we must be prepared to perform a 1-bit right shift again if that happens. Since the output of the Round Module is always a positive number, we check if the sign bit is one after rounding,which indicates an overflow. Therefore, we shift right once and send a control signal to the Control Unit to increment the exponent of the final result.

2nd Normalize Module

```
(*DONT_TOUCH = "TRUE"*) module Normalize_2 (
    input overflow,
    input [24:0] rounded_significand,
    output normalize_2_shamt,
    output [24:0] shifted_significand
);

    assign normalize_2_shamt = rounded_significand[24];

    assign shifted_significand = (normalize_2_shamt)? ({1'b0, rounded_significand[24:1]}) : rounded_significand;       //Right shift if control is one

endmodule
```

10 Control Unit Module

Now for the Control Unit, one of the biggest modules in the design, shown below are some simple control signals that are either sent to other modules or used inside the Control Unit. Some of these signals are:
-"significand_comp": determine whether the 1st Operand's Significand is greater than the 2nd for sign logic.
-"scps_swap": to determine whether the 2 operands should be swapped for alignment due to difference in exponents.
-"round_comp": to determine whether the value is negative and should be complemented in round module.

Simple Control Signals

```
wire significand_comp, exp_comp, exp_equal, absolutely_greater;

assign significand_comp = operand_1_significand > operand_2_significand;

assign exp_comp = ~exponent_sub_sign;
assign exp_equal = ~(|(exponent_sub_result));

assign scps_comp = operand_1_sign ^ operand_2_sign ^ add_or_sub;
assign cpa_cin = scps_comp;

assign absolutely_greater = exp_comp | (exp_equal & significand_comp);

assign scps_swap = exponent_sub_sign;

assign round_comp = (cpa_sign_bit ^ cpa_overflow);
```

10 Control Unit Module

Final Sign Logic: done by listing a 4 input truth table of the following inputs: 1st Operand sign, 2nd Operand sign, add_or_sub, and finally if the 1st Operand is greater than the 2nd Operand, using the absolutely_greater signal. Next, I used a K-Map to minimize the logic, which resulted in this expression.
Final Exponent Logic: first it determines which of the exponents is larger by the sign of the Exponent Sub Module, and subtract from it the amount of left shifts, and add to it the amount of right shifts.

Final Sign and Final Exponent Logic

```
final_sign = (~absolutely_greater & !operand_2_sign & add_or_sub) | (~absolutely_greater & operand_2_sign & !add_or_sub) | (absolutely_greater & operand_1_sign);

final_exp = ((exponent_sub_sign)? (operand_2_exponent) : (operand_1_exponent)) - normalize_1_shamt + normalize_2_shamt;
```

10 Control Unit Module

The Next Part is handling the special cases: Zero, NaN, and Infinity. In this case if one of the inputs is NaN then the output is the same NaN but we flip the MSB of the Mantissa to be one if it is not, to output QNaN aka Quiet NaN, which indicates that the input was invalid or the operation itself is invalid. Subtracting 2 Infinities will also result in QNaN.
If both the operands are equal and they are neither Infinity or NaN and the operation is a subtraction, then the result will immediately will be equal to Zero.
If both of the inputs are infinity and the operation is addition, the output is infinity, or if both the numbers are large enough and the operation is also an addition, then the output is infinity.

Special Cases Section

```
if (exp_stacked && ((sig_non_zero) || (significand_equal & scps_comp))) begin
    special_cases = NAN;
end
else if (significand_equal & exp_equal & !exp_stacked & ( (!sig_non_zero & !exp_non_zero) | (sig_non_zero & exp_non_zero & scps_comp) )) begin
    special_cases = ZERO;
end
else if ((exp_stacked & !sig_non_zero & !scps_comp) | (!exp_stacked & !scps_comp & (final_exp == 8'b11111111))) begin
    special_cases = INF;
end
else begin
    special_cases = 0;
end
```

```
localparam ZERO = 2'b01;
localparam INF = 2'b10;
localparam NAN = 2'b11;
```

10 Control Unit Module

The Next special cases are as follows: if both the inputs are zero and the operation is a subtraction, the result will be negative zero, if the output is zero and the operation was (-1st operand - - 2nd operand), the output will be negative zero.
If one of the inputs is NaN, then the output will be the NaN operand, if the result is NaN and none of the inputs are NaN, the output will be a simple qNaN, with the exponent all ones and the MSB of the Mantissa equal 1 and the rest are zeros. This values are driven on the wire penultimate_significand, which dictates the Mantissa of the output in the case that the output is a NaN.

More Special Cases

```
if (significand_equal & exp_equal & !exp_non_zero & !sig_non_zero & scps_comp) begin
    final_sign = 1;
end
else if ((special_cases == ZERO) & scps_comp & add_or_sub & operand_1_sign) begin
    final_sign = 1;
end

if ((operand_1_exponent == 8'b11111111) & (|operand_1_significand)) begin
    final_exp = operand_1_exponent;
    penultimate_significand = operand_1_significand;
    final_sign = operand_1_sign;
end
else if ((operand_2_exponent == 8'b11111111) & (|operand_2_significand)) begin
    final_exp = operand_2_exponent;
    penultimate_significand = operand_2_significand;
    final_sign = operand_2_sign;
end
else begin
    penultimate_significand = {3'b1, 22'b0};
end
```

11 Pack Module

Finally, we combine the final values of the significand, exponents, and the sign. Also, keeping in mind the special, in case the result is zero, all the bits are zero except the sign is determined by the Control Unit. If the result is NaN, the sign and the exponent are given by the Control Unit, and the significand is determined by the penultimate_significand input coming from the Control Unit. If the result is Infinity, the sign is still coming from the Control Unit, and the exponent is all ones and the significand is all zeros. Finally, if none of those cases are true, the result comes out as it from normal computation.

Pack Module

```
(*DONT_TOUCH = "TRUE"*) module Pack (
    input [24:0] final_significand, penultimate_significand,
    input [7:0] final_exp,
    input [1:0] special_cases,
    input final_sign,
    output reg [31:0] result
);

    localparam ZERO = 2'b01;
    localparam INF = 2'b10;
    localparam NAN = 2'b11;

    always @(*) begin
            case (special_cases)
                2'b00: result = {final_sign, final_exp, final_significand[22:0]};
                ZERO : result = {final_sign, 31'b0};
                INF  : result = {final_sign, 8'b11111111, 23'b0};
                NAN  : result = {final_sign, final_exp, 1'b1, penultimate_significand[21:0]};
                default: result = 0;
            endcase
    end

endmodule
```

12 Top Module

Finally, we connect all the modules together in the order of the pipeline.

Port List

```
(*DONT_TOUCH = "TRUE"*) module FP_Add_Sub_Top (
    input add_or_sub,
    input [31:0] operand_1, operand_2,
    output [31:0] result
);
```

Instantiation

```
//Instantiation
Unpack up(operand_1, operand_2, operand_1_sign, operand_2_sign, significand_equal, sig_non_zero, exp_stacked, operand_1_exponent, operand_2_exponent,
        operand_1_significand, operand_2_significand);

Exponent_Sub es(operand_1_exponent, operand_2_exponent, exponent_sub_result, exponent_sub_sign, exp_non_zero);

SCPS scps(operand_1_significand, operand_2_significand, scps_swap, scps_comp, operand_1_significand_scps_result, operand_2_significand_scps_result);

Align_Significands as(scps_comp, operand_1_significand_scps_result, operand_2_significand_scps_result, exponent_sub_result, operand_1_significand_shifted,
                      operand_2_significand_shifted);

CPA cpa(operand_1_significand_shifted, operand_2_significand_shifted, cpa_cin, cpa_cout, cpa_overflow, cpa_sign_bit, cpa_sum);

Normalize_1 n1(cpa_sum, round_comp, cpa_overflow, normalize_1_shamt, shifted_sum);

Round rnd(shifted_sum, round_comp, rounded_significand);

Normalize_2 n2(cpa_overflow, rounded_significand, normalize_2_shamt, shifted_significand);

Control_Unit ctrl(operand_1_sign, operand_2_sign, add_or_sub, exponent_sub_sign, cpa_sign_bit, normalize_2_shamt, cpa_overflow, significand_equal, sig_non_zero,
                  exp_non_zero, exp_stacked, normalize_1_shamt, exponent_sub_result, operand_1_significand, operand_2_significand, operand_1_exponent,
                  operand_2_exponent, scps_swap, cpa_cin, round_comp, scps_comp, final_sign, special_cases, final_exp, penultimate_significand);

Pack p(shifted_significand, penultimate_significand, final_exp, special_cases, final_sign, result);
```

13 Testbench

I utilized 4 method to verify my design, the first method is using a simple Testbench.

Test Cases to verify corner cases

```
//Instantiation:
FP_Add_Sub_Top dut(add_or_sub, operand_1, operand_2, result);

initial begin
    add_or_sub = 0;
    operand_1 = 32'b01000000101000000000000000000000; // 5.0
    operand_2 = 32'b01000000010000000000000000000000; // 3.0
    #10;
    add_or_sub = 1;
    operand_1 = 32'b01000001001000000000000000000000; // 10.0
    operand_2 = 32'b01000000001000000000000000000000; // 2.5
    #10;
    add_or_sub = 0;
    operand_1 = 32'b00111111111000000000000000000000; // 1.5
    operand_2 = 32'b00111111111000000000000000000000; // 1.5
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01000000001100000000000000000000; // 2.75
    operand_2 = 32'b00111111000000000000000000000000; // 0.5
    #10;
    add_or_sub = 1;
    operand_1 = 32'b01000000010000000000000000000000; // 3.0
    operand_2 = 32'b01000000101000000000000000000000; // 5.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01111111011111111111111111111111; // big number
    operand_2 = 32'b01111111011111111111111111111111; // big number
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01001110100000000000000000000000; // 1.07e9
    operand_2 = 32'b00110010100000000000000000000000; // 1.49e-8
    #10;
    add_or_sub = 1;
    operand_1 = 32'b01001110100000000000000000000000; // 1.07e9
    operand_2 = 32'b00110010100000000000000000000000; // 1.49e-8
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01000000011111111111111111111111; // 3.999999
    operand_2 = 32'b01000000000000000000000000000001; // 2.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01000000100111111111111111111111; // 4.999999
    operand_2 = 32'b00111111100111111111111111111111; // 1.25
    #10;
```

```
    add_or_sub = 1;
    operand_1 = 32'b01001001000000000000000000000001; // 52488
    operand_2 = 32'b01001001000000000000000000000000; // 52488
    #10;
    add_or_sub = 1;
    operand_1 = 32'b00111111111111111111111111111111; // 2.0
    operand_2 = 32'b01000000000000000000000000000000; // 2.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b00111111100000000000000000000001; // 1.0
    operand_2 = 32'b00111010100000000000000000000001; // 0.000976563
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01111111011111111111111111111111; // big number
    operand_2 = 32'b00000000000000000000000000000001; // 1.4e-45
    #10;
    add_or_sub = 0;
    operand_1 = 32'b00111111100000000000000000000001; // 1.0
    operand_2 = 32'b00111111100000000000000000000001; // 1.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b00111111100000000000000000000000; // 1.0
    operand_2 = 32'b00110000000000000000000000000001; // 4.66e-10
    #10;
    add_or_sub = 0;
    operand_1 = 32'b00111111111111111111111111111111; // 2.0
    operand_2 = 32'b10111111111111111111111111111110; // -2.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01000000010000000000000000000000; // 3.0
    operand_2 = 32'b01000000010000000000000000000000; // 3.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01111111011111111111111111111111; // Largest finite float
    operand_2 = 32'b01000000000000000000000000000000; // 2.0
    #10;
    add_or_sub = 1;
    operand_1 = 32'b01000001000100000000000000000000; // 9.0
    operand_2 = 32'b01000001000011111111111111111111; // Slightly less than 9.0
    #10;
    add_or_sub = 0;
    operand_1 = 32'b01111111000000000000000000000000; // Large float
    operand_2 = 32'b00111111100000000000000000000000; // 1.0
    #10;
```

13 Testbench

I utilized 3 method to verify my design, the first method is using a simple Testbench.

More Test Cases and Hierarchical calling to check for certain errors

```
add_or_sub = 0;
operand_1 = 32'b01111111100000000000000000000000; // Positive infinity
operand_2 = 32'b01111111100000000000000000000000; // Positive infinity
#10;
add_or_sub = 1;
operand_1 = 32'b01111111100000000000000000000000; // Positive infinity
operand_2 = 32'b01111111100000000000000000000000; // Positive infinity
#10;
add_or_sub = 1;
operand_1 = 32'b00000000000000000000000000000000; // Zero
operand_2 = 32'b00000000000000000000000000000000; // Zero
#10;
add_or_sub = 1;
operand_1 = 32'b11111111011111111111111111111111; // Largest negative float
operand_2 = 32'b11000000000000000000000000000000; // -2.0
#10;
add_or_sub = 0;
operand_1 = 32'b01000001001000000000000000000000; // 10.0
operand_2 = 32'b11000001001000000000000000000000; // -10.0
#10;
add_or_sub = 1;
operand_1 = 32'b00111111111111111111111111111111; // 1.11111111111111111111
operand_2 = 32'b00111111100000000000000000000001; // 1.00000000000000000001
#10;
add_or_sub = 1;
operand_1 = 32'b01000100010000000000000000000000; // 1.1 × 2^10
operand_2 = 32'b01000011101111111111111111111111; // 1.01111111111111111111 × 2^9
#10;
add_or_sub = 1;
operand_1 = 32'b01000100001010101010101010101010; // 1.01010101010101010101 × 2^10
operand_2 = 32'b01000100010101010101010101010101; // 1.10101010101010101010 × 2^10
#10;
add_or_sub = 1;
operand_1 = 32'b01000011011111111111111111111111; // 1.11111111111111111111 × 2^7          to test overflow
operand_2 = 32'b01000011000000000000000000000001; // 1.00000000000000000001 × 2^7
#10;
add_or_sub = 1;
operand_1 = 32'b01000000110011001100110011001101; // 1.10011001100110011001101 × 2^2          to test guard
operand_2 = 32'b01000000001101101101101101101101; // 1.01101101101101101101 × 2^1
#10;
add_or_sub = 1;
operand_1 = 32'b01111110100100100100100100100100; // 1.00100100100100100100 × 2^126         extreme exponent
operand_2 = 32'b01111110000000000000000000000000; // 1.00000000000000000000 × 2^125
#10;
```

```
//Hierarchical Calling:
assign operand_1_significand_shifted = dut.operand_1_significand_shifted;
assign operand_2_significand_shifted = dut.operand_2_significand_shifted;
assign cpa_sum = dut.cpa_sum;
assign shifted_sum = dut.shifted_sum;
assign rounded_significand = dut.rounded_significand;
assign round_up = dut.rnd.round_up;
assign shifted_significand = dut.shifted_significand;
assign sign = dut.rnd.sign;
assign final_sign = dut.final_sign;
assign final_exp = dut.final_exp;
assign shifted_sum_comp = dut.rnd.shifted_sum_comp;
assign cpa_overflow = dut.cpa_overflow;
assign cpa_sign_bit = dut.cpa_sign_bit;
assign normalize_1_shamt = dut.normalize_1_shamt;
assign guard = dut.as.guard;
assign round = dut.as.round;
assign sticky = dut.as.sticky;
assign round_comp = dut.round_comp;
assign scps_comp = dut.scps_comp;
assign significand_equal = dut.significand_equal;
assign exp_equal = dut.ctrl.exp_equal;
assign exp_stacked = dut.exp_stacked;
assign sig_non_zero = dut.sig_non_zero;
assign exp_non_zero = dut.exp_non_zero;
assign operand_1_sign = dut.ctrl.operand_1_sign;
assign operand_2_sign = dut.ctrl.operand_2_sign;
assign absolutely_greater = dut.ctrl.absolutely_greater;
assign shifter_shamt = dut.n1.shifter_shamt;
```

13 Testbench

For further Verification, I used a Golden Model made using a small C++ Code to compare with the results of my design, as Verilog does not support a "float" data type. It takes 2 Binary Inputs and outputs the result in binary-coded form and in decimal form. It works as follows: the input is a 32 bit number stored in a 32-bit unsigned integer data type. The address of this variable is then stored in a float pointer and then dereferenced, thus this 32-bit value is now a float.
After executing the operation on this float, the address of the variable storing the result is then stored in a pointer of unsigned int data type, thus the raw binary value stored in the address is now treated as an integer instead of a IEEE 754 single precision number. This value is then dereferenced again and is converted to binary to be printed and compared to the results in the Testbench.

Golden Model

```
uint32_t x, y;
int add_or_sub = 0;
float fresult;

x = 0b01001001000000000000000000000001;
y = 0b01001001000000000000000000000000;

float xf = *(float*)&(x);
float yf = *(float*)&(y);
add_or_sub = 1;

if (add_or_sub)
    fresult = xf - yf;
else fresult = xf + yf;
unsigned int result = *(unsigned int*)&(fresult);

bitset<32> binary(result);

cout << xf << endl << yf << endl << fresult << endl;

cout << binary << endl;
```

13 Testbench

In my research, I found out that SystemVerilog supports float operations, using the data type "shortreal" which is a 4-bit, single precision float variable. So, i decided to write a similar testbench to the one i wrote in Verilog but using SystemVerilog, to automate the comparison and debugging processes.
Note: as i have not written SystemVerilog before, i utilized the help of "Deepseek" to help me with the syntax and to know the features of the language.
I used this Testbench to simulate 1 Million Test Cases, to be sure that there are no errors in my design passing all necessary checks, i would like to note that this design does not support subnormal inputs or outputs, thus if either the inputs or outputs where subnormal, they did not undergo any error checks.

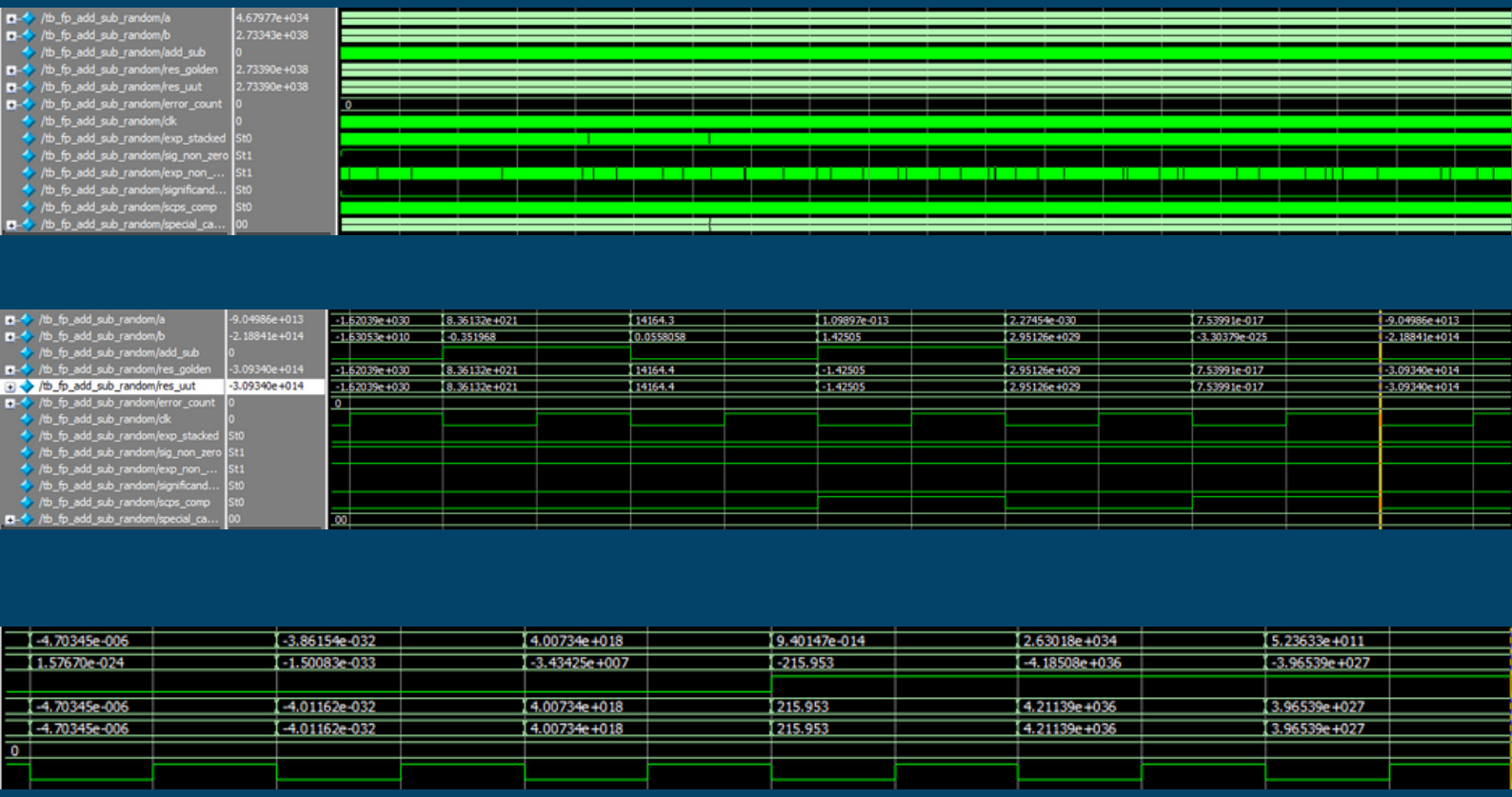
Simulation Transcript

```
# Starting random test with 1000000 iterations...
# Completed 0/1000000 tests (0 errors)
# Completed 100000/1000000 tests (0 errors)
# Completed 200000/1000000 tests (0 errors)
# Completed 300000/1000000 tests (0 errors)
# Completed 400000/1000000 tests (0 errors)
# Completed 500000/1000000 tests (0 errors)
# Completed 600000/1000000 tests (0 errors)
# Completed 700000/1000000 tests (0 errors)
# Completed 800000/1000000 tests (0 errors)
# Completed 900000/1000000 tests (0 errors)
#
# Test completed:
#   Total tests : 1000000
#   Errors      : 0
#   Error rate  : 0.0000%
```

13 Testbench

Shown below is the simulation waveform and snippets of some test cases

Simulation Waveform



13 Testbench

Snippets of Testbench:

```
// Golden reference function
function automatic logic [31:0] fp_add_sub_golden(
    input logic [31:0] a_bits,
    input logic [31:0] b_bits,
    input logic add_sub
);
    shortreal a_real, b_real, result_real;
    a_real = $bitstoshortreal(a_bits);
    b_real = $bitstoshortreal(b_bits);

    if (add_sub) result_real = a_real - b_real;  // Subtraction
    else         result_real = a_real + b_real;  // Addition

    return $shortrealtobits(result_real);
endfunction
```

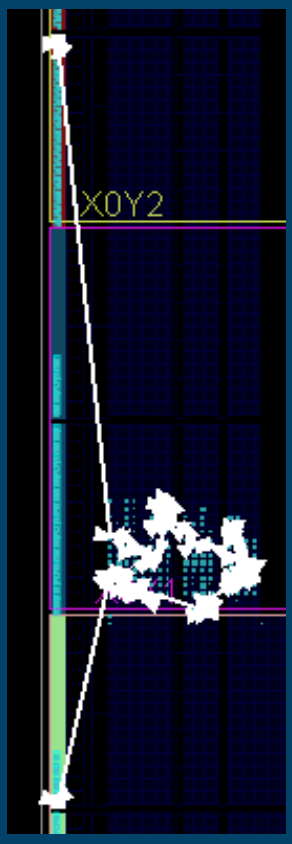
```
// Test procedure
initial begin
    $display("Starting random test with %0d iterations...", NUM_TESTS);
    $timeformat(-9, 0, "ns", 10);

    // Initialize random seed
    $urandom(SEED);

    for (int i = 0; i < NUM_TESTS; i++) begin
        // Generate random inputs
        a = $urandom();
        b = $urandom();
        add_sub = $urandom_range(0, 1);

        // Calculate golden reference
        res_golden = fp_add_sub_golden(a, b, add_sub);

        // Apply to UUT
        @(posedge clk);

        // Check results
        @(negedge clk);
        if ((res_golden != res_uut) & (((a[30:23] != 0) & (b[30:23] != 0)) & (res_golden[30:23] != 0))) begin
            error_count++;
            $display("[%0t] Error #%0d at iteration %0d:", $time, error_count, i);
            $display("  a       = 32'h%h (%f)", a, $bitstoshortreal(a));
            $display("  b       = 32'h%h (%f)", b, $bitstoshortreal(b));
            $display("  op      = %s", add_sub ? "SUB" : "ADD");
            $display("  UUT     = 32'h%h (%f)", res_uut, $bitstoshortreal(res_uut));
            $display("  Golden  = 32'h%h (%f)", res_golden, $bitstoshortreal(res_golden));
        end

        // Progress reporting
        if (i % 100_000 == 0) begin
            $display("Completed %0d/%0d tests (%0d errors)", i, NUM_TESTS, error_count);
        end
    end
```

14 Utilization Report

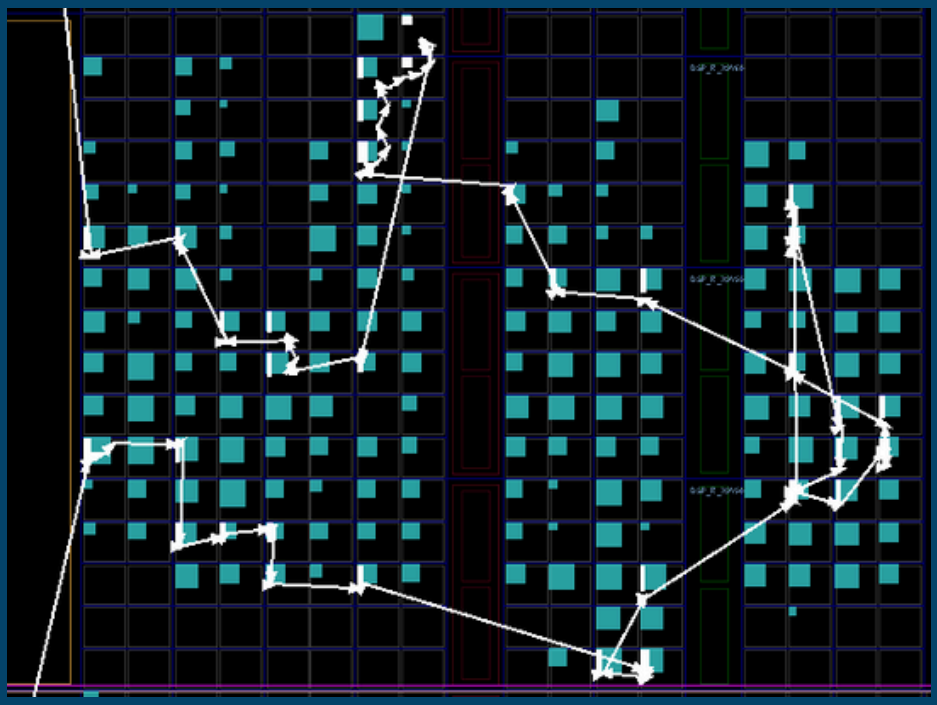
Utilization Report

Name	Slice LUTs (64000)	Slice (16000)	LUT as Logic (64000)	Bonded IOB (400)
N FP_Add_Sub_Top	595	180	595	97
as (Align_Significands)	86	28	86	0
cpa (CPA)	138	48	138	0
Ladner (Adder_Top...	137	48	137	0
LF (Ladner_Fisc...	80	36	80	0
ctrl (Control_Unit)	88	35	88	0
es (Exponent_Sub)	15	5	15	0
n1 (Normalize_1)	161	56	161	0
shift (Shifter)	71	26	71	0
ZOC (Zero_One_Co...	57	21	57	0
encoder (LZD_32)	40	15	40	0
inst0_16 (LZD...	19	8	19	0
inst0_4 (LZ...	2	1	2	0
inst1_4 (LZ...	2	2	2	0
inst2_4 (LZ...	2	2	2	0
inst3_4 (LZ...	2	1	2	0
instf_4 (LZ...	2	2	2	0
inst1_16 (LZD...	19	8	19	0
inst0_4 (LZ...	2	2	2	0
inst1_4 (LZ...	2	1	2	0
inst2_4 (LZ...	2	1	2	0
inst3_4 (LZ...	2	2	2	0
instf_4 (LZ...	2	2	2	0
n2 (Normalize_2)	12	4	12	0
p (Pack)	22	14	22	0
rnd (Round)	28	14	28	0
scps (SCPS)	25	12	25	0
up (Unpack)	20	9	20	0

15 Timing Report



Critical Path:



Critical Path Delay:

```
Timing Report

Slack (MET) :             12.454ns  (required time - arrival time)
  Source:                 operand_2[25]
                            (input port clocked by clk  {rise@0.000ns fall@25.000ns period=50.000ns})
  Destination:            result[24]
                            (output port clocked by clk  {rise@0.000ns fall@25.000ns period=50.000ns})
  Path Group:             clk
  Path Type:              Max at Slow Process Corner
  Requirement:            50.000ns  (clk rise@50.000ns - clk rise@0.000ns)
  Data Path Delay:        37.521ns  (logic 11.865ns (31.622%)  route 25.656ns (68.378%))
  Logic Levels:           41  (CARRY4=8 IBUF=1 LUT1=2 LUT2=6 LUT3=10 LUT4=4 LUT5=2 LUT6=7 OBUF=1)
  Input Delay:            0.000ns
  Output Delay:           0.000ns
  Clock Uncertainty:      0.025ns
```

15 Timing Report

Constraints File:

```
1 create_clock -name clk -period 50.000 -waveform {0.000 25.000}
2 set_input_delay -clock clk 0 [get_ports add_or_sub]
3 set_input_delay -clock clk 0 [get_ports operand_1]
4 set_input_delay -clock clk 0 [get_ports operand_2]
5 set_output_delay -clock clk 0 [get_ports result]
```

Timing Summary:

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	12.454 ns	Worst Hold Slack (WHS):	2.942 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32	Total Number of Endpoints:	32

All user specified timing constraints are met.

Delays:

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception
Path 1	12.454	41	38	88	operand_2[25]	result[24]	37.521	11.865	25.656	50.0	clk	clk	
Path 2	12.579	41	38	88	operand_2[25]	result[27]	37.396	11.898	25.498	50.0	clk	clk	
Path 3	12.685	41	38	88	operand_2[25]	result[28]	37.290	11.896	25.394	50.0	clk	clk	
Path 4	12.815	41	38	88	operand_2[25]	result[25]	37.160	11.894	25.266	50.0	clk	clk	
Path 5	12.827	41	38	88	operand_2[25]	result[26]	37.148	11.909	25.239	50.0	clk	clk	
Path 6	12.836	41	38	88	operand_2[25]	result[22]	37.139	11.851	25.288	50.0	clk	clk	
Path 7	12.846	41	38	88	operand_2[25]	result[30]	37.129	11.845	25.284	50.0	clk	clk	
Path 8	12.971	41	38	88	operand_2[25]	result[23]	37.004	11.867	25.137	50.0	clk	clk	
Path 9	13.173	41	38	88	operand_2[25]	result[21]	36.802	11.688	25.114	50.0	clk	clk	
Path 10	13.371	41	38	88	operand_2[25]	result[20]	36.604	11.686	24.918	50.0	clk	clk	

16 Drive Link

PRESS TO ACCESS DRIVE

17 Github Repository

PRESS TO ACCESS REPOSITORY

18 References

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford University Press, 2010.

[2] S. Roy, *Advanced Digital System Design: A Practical Guide to Verilog Based FPGA and ASIC Implementation*. Cham, Switzerland: Springer, 2023.

[3] Eng. Ahmed Abdelsalam Slides, STMicroelectronics.