EE515 VLSI SYSTEM DESIGN
Course Project

Fused Floating-Point Two-term Sum-of-Squares Unit

Submitted by,
DHARMJEET BHARTI
Roll No-224102406
May 3, 2023

Contents

1 OBJECTIVE

Design and implement an

- PROPOSED FUSED FLOATING-POINT TWO-TERM SUM-OF-SQUARES ARCHITECTURE.

2 Tools Used

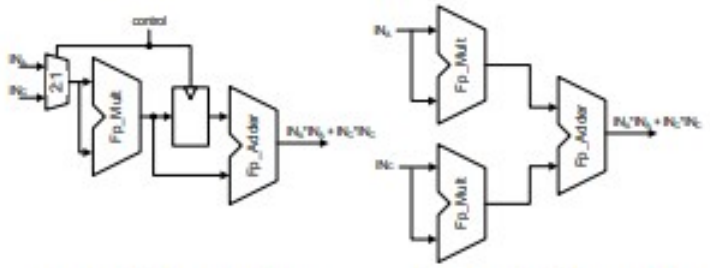
Xilinx vivado Version 2019

3 THEORY

Floating-point number systems are widely used for digital signal processing (DSP), graphics processing, application specific processing and scientific projects due to their wide dynamic range that provides freedom from overflow concerns. However, the floating-point arithmetic units have larger power consumption, area, and latency than a fixed-point arithmetic unit. It has been become important to design low-power floating-point arithmetic units to enable the floating-point units to be applied to mobile processors. Many techniques and architectures have been introduced for enhanced floating-point operation. A fused floating-point architecture has been developed to reduce power consumption and increase performance for units such as a fused floating-point addsubtract unit (Fused A-S) [1], a fused floating-point dot-product unit (Fused DP) [2], a fused multiply-add unit (FMA) [3], and a compound collision operation [4]. The goal in this paper is to introduce a new fused floatingpoint architecture for two-term sum-of-squares computation to obtain enhanced performance, power, and area. The fused sumof-squares (Fused SoSQ) unit will be compared with other floating-point unit such as a discrete dot-product unit, a discrete sum-of-square unit and a fused dot-product unit. The sum-of-squares operation is used for many applications such as filtering, Euclidian branching, pattern recognition, and vector normalization [5]. In addition, the sum-of-squares is used to compare the magnitude of complex numbers. Sum-ofsquares computations can take two terms or multi-terms. This paper is focusing on the two-term sum-of-squares computation with enhanced performance and low power consumption. The main

contributions of this paper are summarized below: A new fused floating-point architecture dedicated to twoterm sum-of-squares is introduced. A new compound addition method is suggested for fused sum-of-squares to enhance performance. This paper provides more design options for application specific processors.

Discrete Floating-Point Serial and Parallel Dot-Product Unit and Sum-of-Squares Unit The floating-point sum-of-squares computations can be executed by general discrete floating-point architectures such as a discrete serial or parallel floating-point dot-product unit implemented with conventional floating-point multipliers and a floating-point adder. Figure 1 shows the discrete serial and parallel floating-point dot-product architectures. For the serial unit, a floating-point multiplier, a floating-point adder, and registers are used. It needs two cycles to compute the sum-ofsquares. The discrete floating-point parallel dot-product unit is implemented with a floating-point adder and two multipliers that are synthesized separately. The parallel floating-point dotproduct units have larger area and power consumption, but much lower latency than the serial floating-point dot-product unit.



a) Serial dot-product b) Parallel dot-product

Figure 1. Discrete Floating-Point Dot Product Unit (after [2]).

A floating-point multiplier is designed with Dadda reduction. Figure 3 shows the floating-point multiplier with a compound addition. The Dadda reduction tree has 7 reduction stages; the delay of one stage is approximately the same as the delay of a full adder. After the final stage, a 47 bit sum and a 47 bit carry are obtained. To remove the carry-out from the bottom half addition from the critical path, a compound addition with advanced normalization, round and sticky bit generation is applied [7]. The exponent computation has a bias subtraction after an exponent bit addition. For a sum-of-squares floating-point arithmetic unit, the internal multipliers are replaced with squarers; a folded squarer is selected. The folded squarer size is around half that of a multiplier tree [8]. Figure 4 illustrates the floating-point squarer unit. Its sign bit is always 0. The exponent unit does not need addition for the input exponent; a one bit left shifted exponent is used.
.

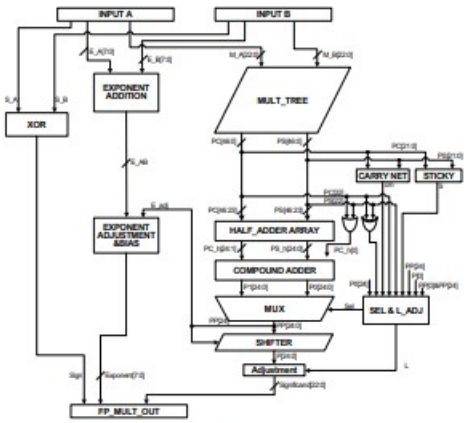


Figure 3. Floating-Point Multiplier (after [7]).

The sum-of-squares can be computed by the fused dotproduct unit for enhanced performance and reduced power consumption and area. In terms of speed, the fused dot-product unit is faster than the discrete serial and parallel floating-point dot-product unit. The power consumption and area of the fused dot-product unit is less than that of the discrete parallel dot-product unit and larger than the discrete serial dot-product unit [2]. Figure 5 shows the block diagram of the fused floatingpoint dot-product unit. In

the fused floating-point dot-product unit, the two internal significand multiplications generate four 47 bit carry and sum words. Based on the exponent comparison, the carry and sum words are swapped. The carry and sum words with the smaller exponent are aligned (two alignment units are employed). A significand addition is performed after the alignment.

.

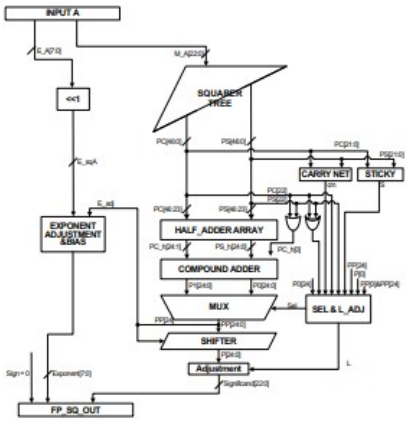


Figure 4. Floating-Point Squarer.

.

.

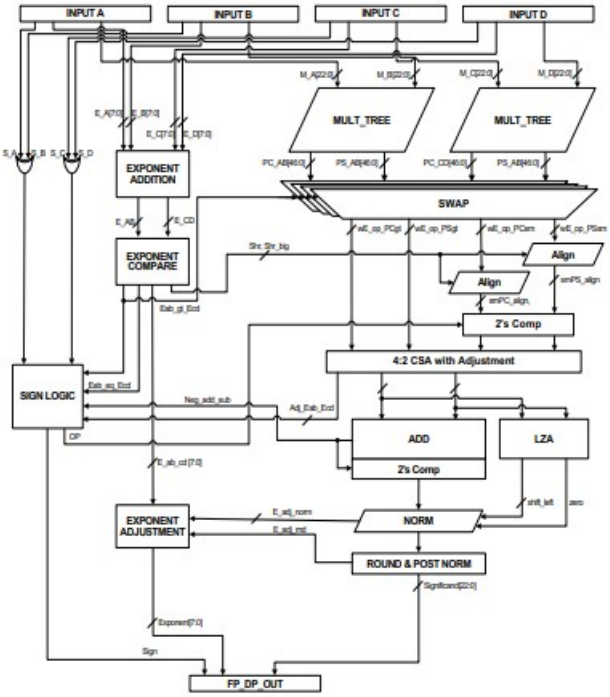


Figure 5. Fused Floating-Point Dot-Product Unit (after [2]).

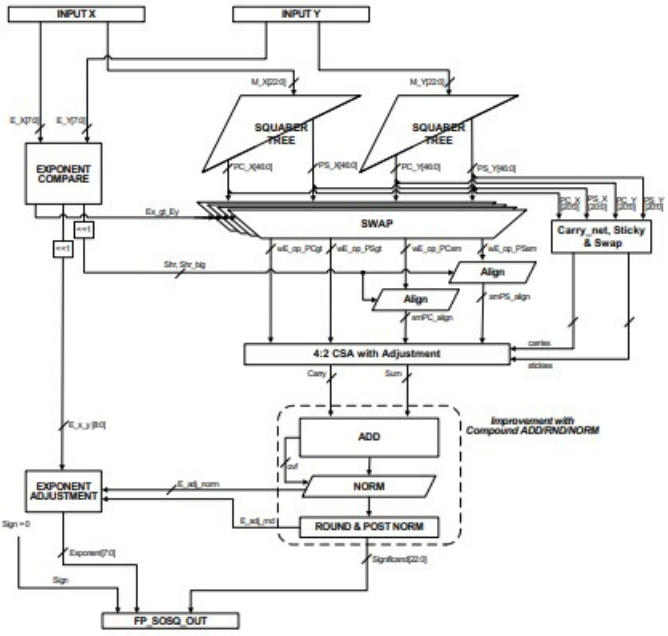


Figure 6. Fused Floating-Point Sum-of-Squares Unit.

4 Implementation

Common procedure is adopted for all the experiments to be performed:

- Launch the Xilinx Vivado tool.
- Open a new project, make a new file and write the Verilog code.
- Debug the code if there are any errors. After this we can start the simulation and get waveforms of input and input.
- Run synthesis and implementation successfully to see the schematic.
- Using constraint wizard, generate constraint file and launch post synthesis timing simulation.
- Using the TCL console, we can view delays as well as utilization and power reports.

5 Performance

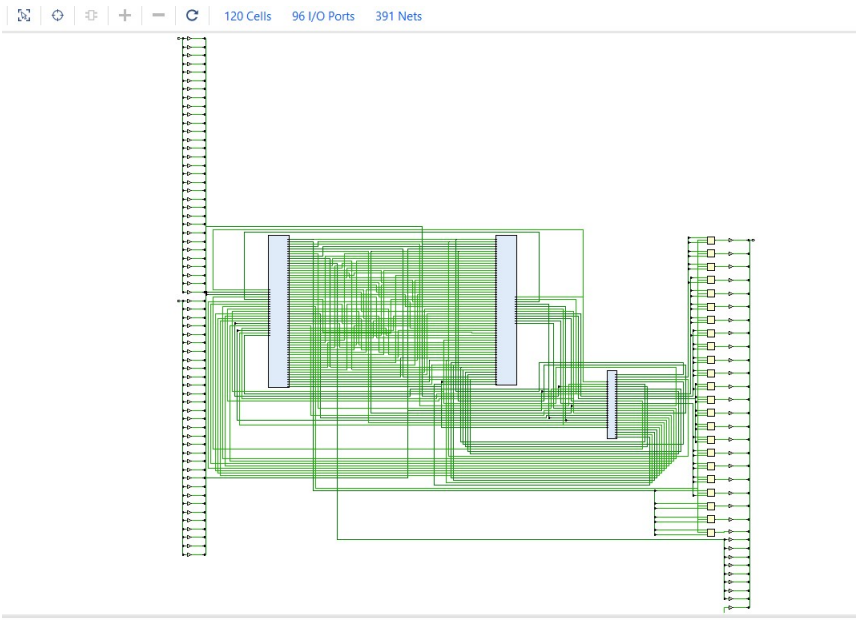


Figure 1: RTL schematic

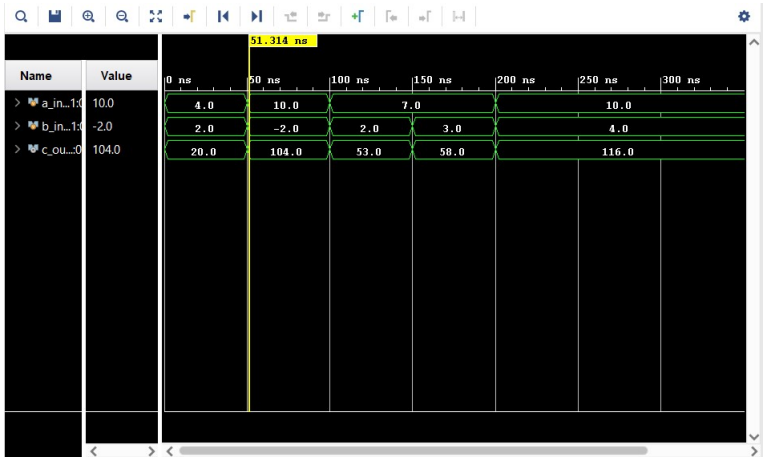


Figure 2: Behavioural Simulation

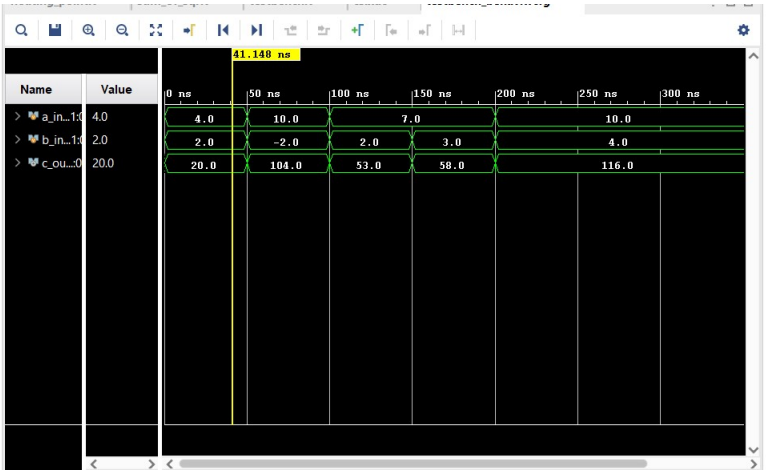


Figure 3: Post-synthesis Functional Simulation

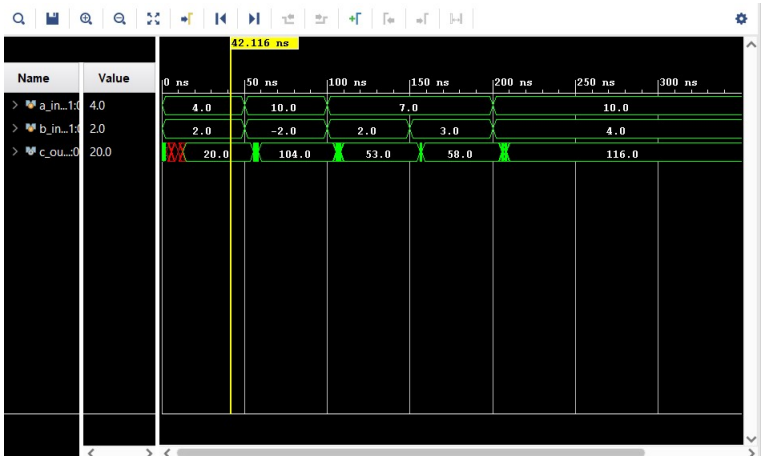


Figure 4: Post-Synthesis Timing Simulation

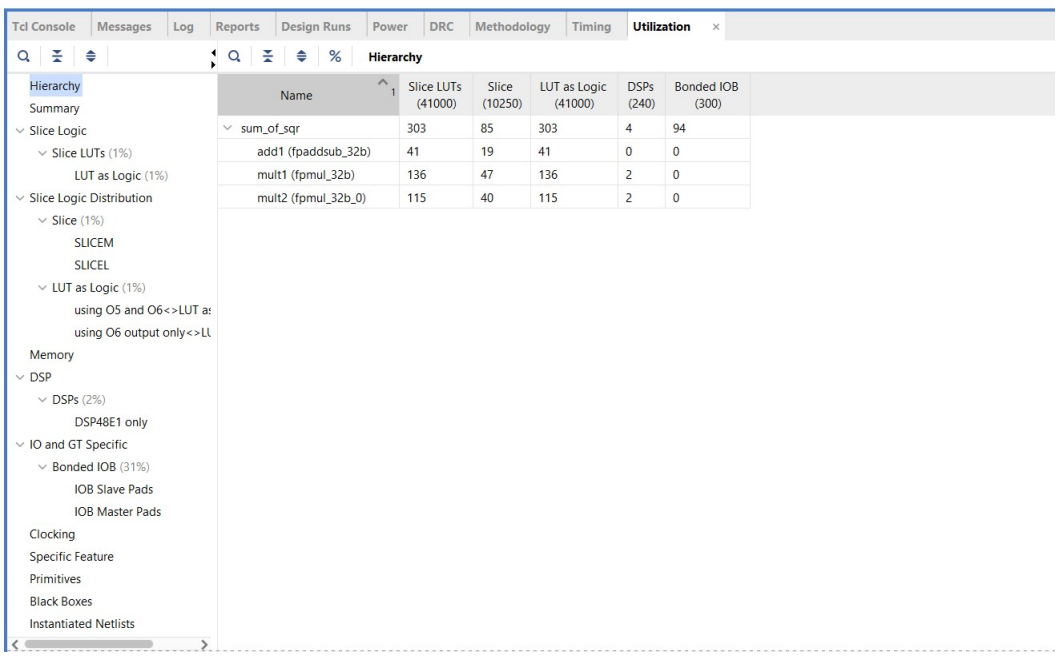


Figure 5: Utilization Report

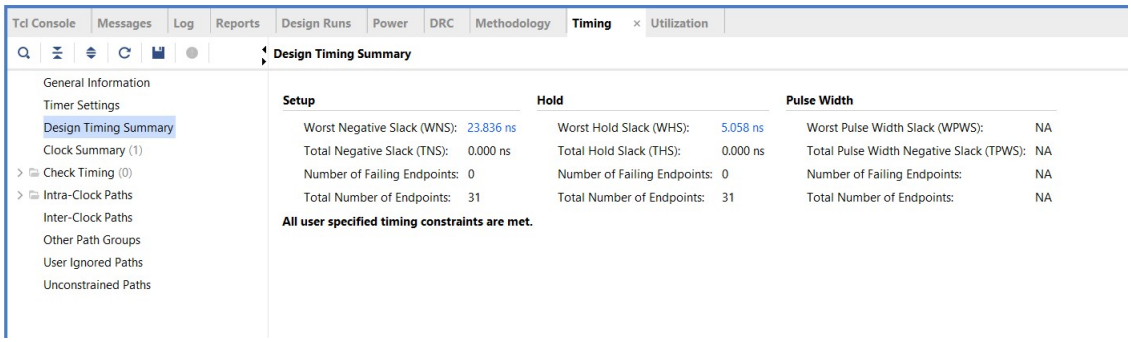


Figure 6: Timing Report

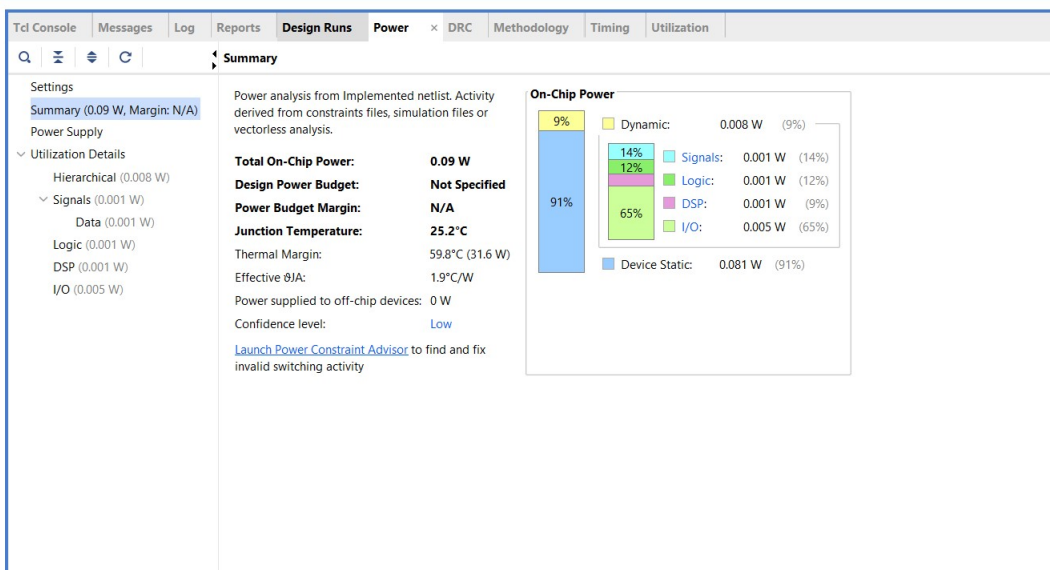


Figure 7: Power Report

6 Conclusions

- fused floating-point sum-of-squares unit with/without the advanced rounding system and presents a comparison of the nine types of floating-point arithmetic units for computing the sum-of-squares.

- The fused sum-of-squares unit has less latency, area, and power consumption than the floating-point dot-product units and the discrete parallel sumof-squares unit. Compared to the fused dot-product, the fused sum-of-squares unit shows less power consumption and better performance. This fused sum-of-squares unit is attractive for application-specific processing.

7 Appendix

7.1 Code

```
 **** MAIN MODULE ****
 `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 01/17/2023 09:36:25 AM
// Design Name:
// Module Name: 32bfpmul
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fpaddsub_32b(a_in, b_in, sub, c_out );
parameter m = 8, n = 23;
input [m+n:0] a_in, b_in;
input sub;
output [m+n:0] c_out;

wire sign, sub_b, a_is_big, b_is_big, both_equal, mant_abig;
wire [m-1:0] exp, exp_c, exp_d, shift_index;
wire [n+1:0] mant, mant_d, a_shifted, b_shifted, shifted;
```

```
assign a_is_big = a_in[m+n-1:n] > b_in[m+n-1:n];
assign b_is_big = a_in[m+n-1:n] < b_in[m+n-1:n];
assign both_equal = a_in[m+n-1:n] == b_in[m+n-1:n];
assign mant_abig = both_equal ?
                a_in[n-1:0] > b_in[n-1:0] : 'b0;

assign exp_c = a_is_big ? a_in[m+n-1:n] - b_in[m+n-1:n] :
                b_is_big ? b_in[m+n-1:n] - a_in[m+n-1:n] : 'b0;

assign a_shifted = b_is_big ? {1'b1,a_in[n-1:0]}>>(exp_c) :
                    {1'b1,a_in[n-1:0]};
assign b_shifted = a_is_big ? {1'b1,b_in[n-1:0]}>>(exp_c) :
                    {1'b1,b_in[n-1:0]};

assign sub_b = a_in[m+n] ^ (b_in[m+n] ^ sub);
assign mant_d = sub_b ? (a_is_big || mant_abig) ?
                a_shifted - b_shifted :
                b_shifted - a_shifted :
                a_shifted + b_shifted;
//assign mant_e = mant_abig ? {1'b1,a_in[n-1:0]} - {1'b1,b_in[n-1:0]} :
//                                {1'b1,b_in[n-1:0]} - {1'b1,a_in[n-1:0

assign sign = (a_is_big || mant_abig) ? a_in[m+n] : (b_in[m+n] ^ sub);
assign exp_d = a_is_big ? a_in[m+n-1:n] : b_in[m+n-1:n];

//corner case logic- not needed for normal operation
//mantshift mant_shifter(a_in[m+n-1:n], mant_e, shifted, shift_index );
mantshift mant_shifter(exp_d, mant_d, shifted, shift_index );
//-----------------------------------------------------

assign mant = /*both_equal&&*/sub_b ? shifted : mant_d>>mant_d[n+1];
assign exp = /*both_equal&&*/sub_b ? exp_d - shift_index : exp_d + mant

assign c_out = {sign,exp,mant[n-1:0]};

endmodule

module mantshift(exp, mant, shifted, shift_index );
```

```
parameter m = 8, n = 23;
    input [m-1:0] exp;
    input [n+1:0] mant;
    output reg [m-1:0] shift_index;
    output reg [n+1:0] shifted;
    reg [n+1:0] target;
    reg [$clog2(n)-1:0] cnt;
    always@(mant, exp) begin target = mant;
    shift_index = exp;
    for(cnt = 0; cnt < n+1; cnt = cnt + 1)begin
        if (target[cnt]) shift_index = n - cnt;
    end
    shifted = mant << shift_index;
    end
endmodule


module fpmul_32b(a_in, b_in, c_out );
parameter m = 8, n = 23;
input [m+n:0] a_in, b_in;
output [m+n:0] c_out;

wire sign, is_zero;
wire [m-1:0] exp, exp_c, exp_d;
wire [2*n+1:0] mant;
wire [2*n+1:0] mul;

assign sign = a_in[m+n] ^ b_in[m+n];
assign exp_c = a_in[m+n-1:n] + b_in[m+n-1:n] - {(m-1){1'b1}};
assign mul = {1'b1, a_in[n-1:0]} * {1'b1, b_in[n-1:0]};
assign is_zero = (a_in[m+n-1:0] == 'b0) || (b_in[m+n-1:0] == 'b0);

assign exp = exp_c + mul[2*n+1];
assign mant = mul>>(mul[2*n+1]);

assign c_out = is_zero ? 'b0 : {sign,exp,mant[2*n-1:n]};

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/01/2023 10:29:08 AM
// Design Name:
// Module Name: sum_of_sqr
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module sum_of_sqr(a_in, b_in, c_out );
parameter m = 8, n = 23;

input [m+n:0] a_in, b_in;
output [m+n:0] c_out;

wire [m+n:0] a_2, b_2;

fpmul_32b mult1(a_in, a_in, a_2);
fpmul_32b mult2(b_in, b_in, b_2);

fpaddsub_32b add1(a_2, b_2, 'b0, c_out);

endmodule
```

**** TEST BENCH ****

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/30/2023 10:47:58 PM
// Design Name:
// Module Name: testbench
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module testbench(    );
    reg [31:0] a_in, b_in;
    wire [31:0] c_out;

    sum_of_sqr DUT (a_in, b_in, c_out);

    initial begin
    a_in = 32'b0_10000001_00000000000000000000000;
    b_in = 32'b0_10000000_00000000000000000000000;
    #50 a_in = 32'b0_10000010_01000000000000000000000;
        b_in = 32'b1_10000000_00000000000000000000000;
    #50 a_in = 32'b0_10000001_11000000000000000000000;
        b_in = 32'b0_10000000_00000000000000000000000;
```

```
    #50 a_in = 32'b0_10000001_11000000000000000000000;
        b_in = 32'b0_10000000_10000000000000000000000;
    #50 a_in = 32'b0_10000010_01000000000000000000000;
        b_in = 32'b0_10000001_00000000000000000000000;
        #150
        $finish;
    end
endmodule
```