11/21/2024

MADAN PARALKAR

PES2UG24EC805

CADD JOB ASSIGNMENT

**Floating Point Multiplier**

1. Floating Point Multiplier

(a) Steps for 32-bit Floating Point Multiplication

Extract Fields: Extract the sign, exponent, and mantissa from both 32-bit floating-point inputs.

Sign Calculation: XOR the sign bits of the two inputs to get the sign of the result.

Exponent Calculation: Add the exponents of the two inputs and subtract the bias (127 for single precision).

Mantissa Calculation: Multiply the mantissas of the two inputs. Normalize the result if necessary.

Normalization: Adjust the exponent and mantissa to ensure the mantissa is in the correct range.

Rounding: Apply round toward zero (truncate).

Assemble Result: Combine the sign, exponent, and mantissa to form the final 32-bit floating-point result.

SKETCH

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

\*sign \*exponent \*Mantissa

Input A (Sign | Exponent | Mantissa) -> Input B (Sign | Exponent | Mantissa) -> Sign Calculation -> Exponent Addition -> Mantissa Multiplication -> Normalization -> Rounding -> Assemble Result.

**System Verilog design code for floating point multiplier**

module fp\_multiplier (

input [31:0] a,

input [31:0] b,

output [31:0] result

);

wire sign\_a, sign\_b, sign\_result;

wire [7:0] exp\_a, exp\_b, exp\_result;

wire [23:0] mant\_a, mant\_b, mant\_result;

wire [47:0] mant\_mult;

wire [7:0] exp\_sum;

assign sign\_a = a[31];

assign sign\_b = b[31];

assign exp\_a = a[30:23];

assign exp\_b = b[30:23];

assign mant\_a = {1'b1, a[22:0]};

assign mant\_b = {1'b1, b[22:0]};

assign sign\_result = sign\_a ^ sign\_b;

assign exp\_sum = exp\_a + exp\_b - 8'd127;

assign mant\_mult = mant\_a \* mant\_b;

assign mant\_result = mant\_mult[47] ? mant\_mult[46:24] : mant\_mult[45:23];

assign exp\_result = mant\_mult[47] ? exp\_sum + 1 : exp\_sum;

assign result = {sign\_result, exp\_result, mant\_result[22:0]};

endmodule

Testbench

module test\_fp\_multiplier;

logic [31:0] a, b;

logic [31:0] result;

fp\_multiplier uut (

.a(a),

.b(b),

.result(result)

);

initial begin

$display("Time\t a\t\t\t b\t\t\t result");

$monitor("%0t\t %h\t %h\t %h", $time, a, b, result);

// Test cases

a = 32'h3f800000; // 1.0

b = 32'h40000000; // 2.0

#10;

a = 32'h40400000; // 3.0

b = 32'h40800000; // 4.0

#10;

a = 32'h3f800000; // 1.0

b = 32'h3f800000; // 1.0

#10;

a = 32'h3f800000; // 1.0

b = 32'h00000000; // 0.0

#10;

a = 32'h3f800000; // 1.0

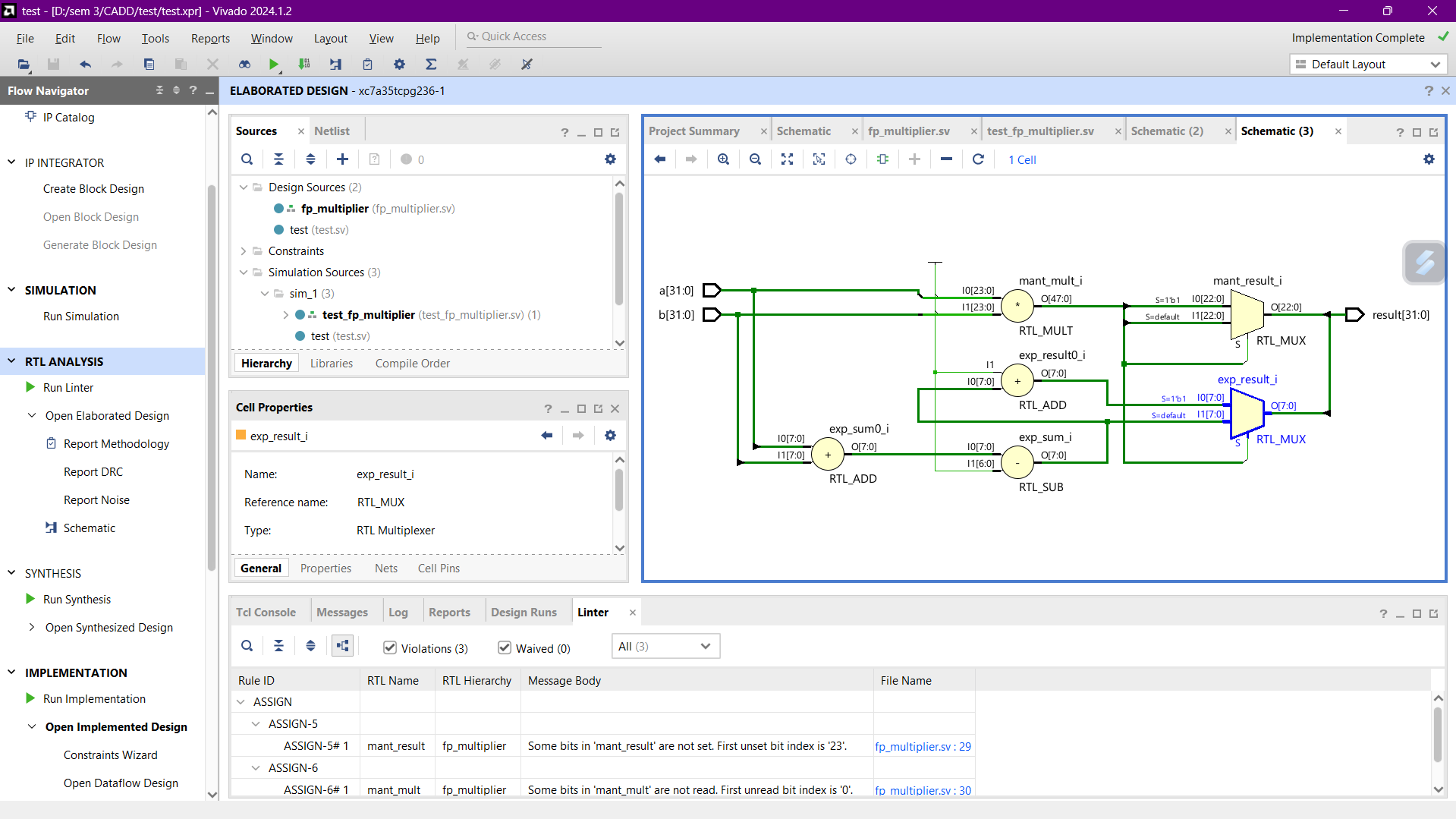
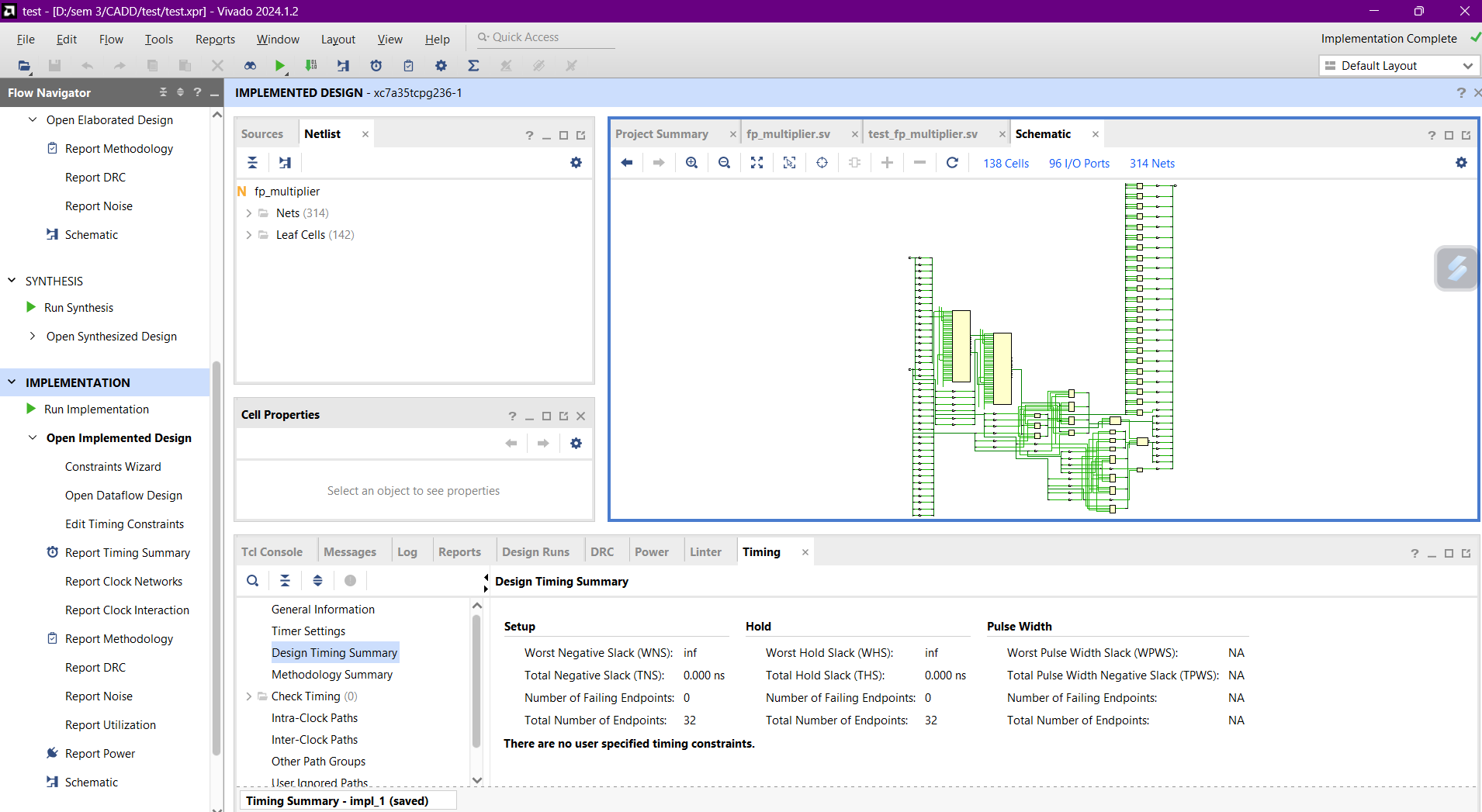
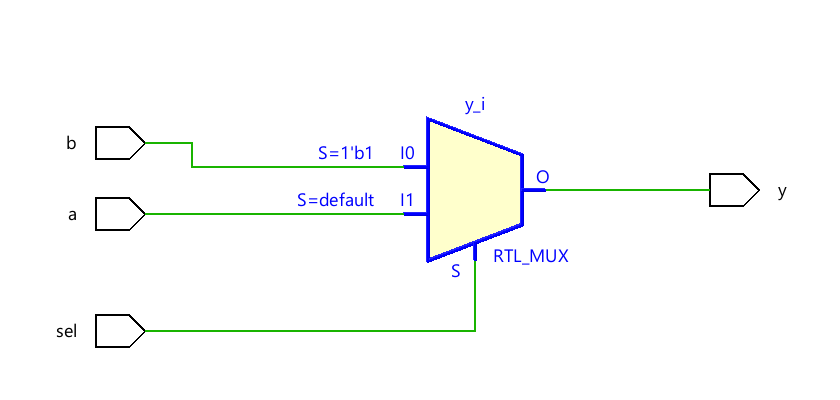
b = 32'hbf800000; // -1.0

#10;

$finish;

end

endmodule



**32-bit prefix adder**

Input A -> Input B -> Generate & Propagate -> Prefix Tree -> Sum Calculation -> Output Sum.

**System Verilog design code for 32-prefix adder**

module prefix\_adder (

input [31:0] a,

input [31:0] b,

output [31:0] sum

);

wire [31:0] g, p, c;

assign g = a & b;

assign p = a ^ b;

assign c[0] = 0;

genvar i;

generate

for (i = 1; i < 32; i = i + 1) begin

assign c[i] = g[i-1] | (p[i-1] & c[i-1]);

end

endgenerate

assign sum = p ^ c;

endmodule

**testbench**

module test\_prefix\_adder;

logic [31:0] a, b;

logic [31:0] sum;

prefix\_adder uut (

.a(a),

.b(b),

.sum(sum)

);

initial begin

$display("Time\t a\t\t\t b\t\t\t sum");

$monitor("%0t\t %h\t %h\t %h", $time, a, b, sum);

// Test cases

a = 32'h00000000; b = 32'h00000000; #10;

a = 32'h00000001; b = 32'h00000001; #10;

a = 32'hFFFFFFFF; b = 32'h00000001; #10;

a = 32'h12345678; b = 32'h87654321; #10;

a = 32'hAAAAAAAA; b = 32'h55555555; #10;

$finish;

end

endmodule

**Delay Calculation**

The delay of the 32-bit prefix adder can be calculated based on the number of stages in the prefix tree. Assuming each two-input gate delay is 100 ps, the delay can be estimated by counting the number of stages

**pipelined\_prefix\_adder**

module pipelined\_prefix\_adder (

input clk,

input [31:0] a,

input [31:0] b,

output reg [31:0] sum

);

reg [31:0] g, p, c;

always @(posedge clk) begin

g <= a & b;

p <= a ^ b;

end

always @(posedge clk) begin

c[0] <= 0;

for (int i = 1; i < 32; i = i + 1) begin

c[i] <= g[i-1] | (p[i-1] & c[i-1]);

end

end

always @(posedge clk) begin

sum <= p ^ c;

end

endmodule

