Double Precision Floating Point Pipelined-Multiplication

CS6230 CAD for VLSI Course Project Report

Submitted by

Varun M (EE20B149)
Harish R (EE20B044)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS

December 2023

This report delves into the design and implementation of a Dual Precision Floating Point Pipelined Booth's Multiplier. The multiplier is a critical component in digital signal processing and scientific computing applications, and the use of Booth's algorithm combined with dual precision floating-point arithmetic enhances its efficiency and precision. This report explores the underlying principles, architecture, and advantages of this specialized multiplier.

1. Introduction:

Multiplication is a fundamental operation in digital systems, and optimizing its performance is crucial for various applications. Booth's algorithm is a multiplication algorithm that efficiently reduces the number of partial products generated during multiplication. When combined with dual precision floating-point arithmetic, it provides enhanced accuracy for scientific and engineering calculations.

2. Booth's Algorithm:

Booth's algorithm is a multiplication algorithm that reduces the number of partial products generated during the multiplication process. It achieves this by examining adjacent pairs of bits in the multiplier and replacing them with a partial product based on a set of rules. This results in a more streamlined multiplication process, reducing the overall computational load.

3. Dual Precision Floating Point Arithmetic:

Dual precision floating-point arithmetic is a method of representing real numbers in a computer, providing extended precision compared to single precision. It typically uses 64 bits to represent a number, with separate sign, exponent, and mantissa fields. This increased precision is especially valuable in scientific and engineering applications where accuracy is paramount.

4. Architecture of Dual Precision Floating Point Pipelined Booth's Multiplier:

Booth's algorithm is a technique for efficient multiplication of signed binary integers in 2's complement notation. While not directly applied to floating point numbers themselves, it plays a crucial role in optimizing the multiplication of their mantissas, which are essentially integers.

Floating-point multiplication involves four main steps:

1. Separate signs and exponents: Extract the sign bits and exponents of both operands.
2. Multiply mantissas: This is where Booth's algorithm comes in.
3. Adjust exponent: Add the exponents, accounting for the hidden bit.
4. Normalize the result: Shift the mantissa and adjust the exponent if needed.

Role of Booth's Algorithm:

1. Optimize mantissa multiplication: Booth's algorithm reduces the number of additions/subtractions needed compared to the standard binary multiplication algorithm. This leads to faster and more efficient floating-point multiplication.

2. Handle signed integers: Booth's algorithm inherently handles signed numbers in 2's complement notation, which is the standard format for mantissas. This avoids the need for additional sign conversion steps.

How Booth's Algorithm is applied:

1. Convert mantissas to 2's complement.
2. Treat the least significant bit (LSB) as the sign bit and the remaining bits as the magnitude.
3. Apply Booth's algorithm to the mantissas as if they were integers.
4. The resulting product is the product of the mantissas.

Benefits of using Booth's Algorithm:

- Reduced number of operations: Compared to the standard binary multiplication algorithm, Booth's algorithm requires fewer additions/subtractions, leading to faster execution.
- Less hardware complexity: As Booth's algorithm involves simpler operations, it translates to less complex hardware implementations, saving space and power.
- Improved accuracy: By reducing the number of operations, Booth's algorithm minimizes the potential for rounding errors, thereby improving accuracy.

Limitations of Booth's Algorithm:

- More complex than the standard algorithm: Booth's algorithm requires additional logic and calculations compared to the standard binary multiplication algorithm.
- Error handling needs special attention.

Overall, Booth's algorithm significantly improves the efficiency and accuracy of floating-point multiplication by optimizing the multiplication of the mantissas. Its simplicity and effectiveness have made it a popular choice for implementing floating-point operations in various hardware and software systems.

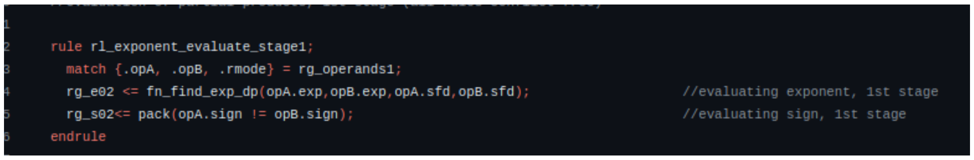
The dual precision floating-point pipelined Booth's multiplier combines the efficiency of Booth's algorithm with the extended precision of dual precision floating-point arithmetic. The architecture typically involves multiple stages of pipelining to maximize throughput. Each stage of the pipeline is responsible for specific operations such as partial product generation, accumulation, and rounding.

5. Implementation:

All the rules we have implemented are conflict free. We have implemented a six stage pipeline divided as follows:

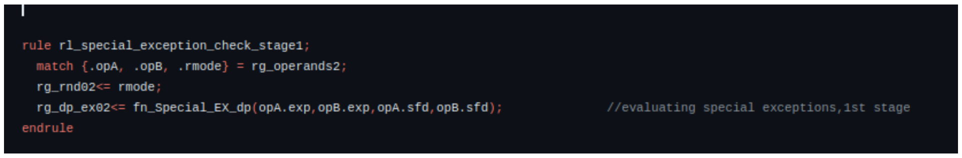
Stage 1:

We evaluate the sign of the floating point numbers as follows:



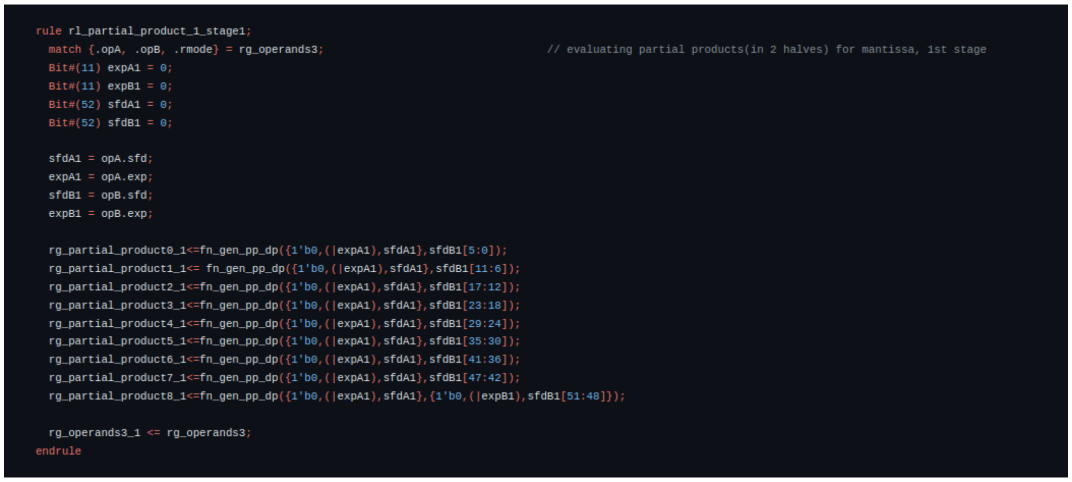
```
  rule rl_exponent_evaluate_stage1;
    match {.opA, .opB, .rmode} = rg_operands1;
    rg_e02 <= fn_find_exp_dp(opA.exp,opB.exp,opA.sfd,opB.sfd);                   //evaluating exponent, 1st stage
    rg_s02<= pack(opA.sign != opB.sign);                                          //evaluating sign, 1st stage
  endrule
```

We evaluate special exceptions such as infinity, zero and nan.



```
rule rl_special_exception_check_stage1;
  match {.opA, .opB, .rmode} = rg_operands2;
  rg_rnd02<= rmode;
  rg_dp_ex02<= fn_Special_EX_dp(opA.exp,opB.exp,opA.sfd,opB.sfd);              //evaluating special exceptions,1st stage
endrule
```

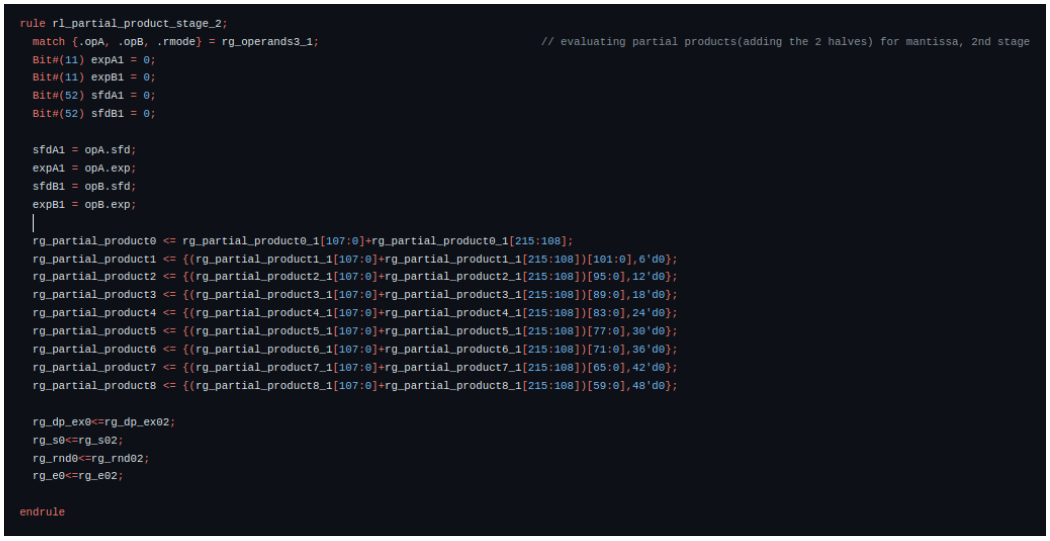
Evaluating the partial products in 2 halves for mantissa. The fn_gen_pp_dp function calculates the partial products.



```
rule rl_partial_product_1_stage1;
  match {.opA, .opB, .rmode} = rg_operands3;                              // evaluating partial products(in 2 halves) for mantissa, 1st stage
  Bit#(11) expA1 = 0;
  Bit#(11) expB1 = 0;
  Bit#(52) sfdA1 = 0;
  Bit#(52) sfdB1 = 0;

  sfdA1 = opA.sfd;
  expA1 = opA.exp;
  sfdB1 = opB.sfd;
  expB1 = opB.exp;

  rg_partial_product0_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[5:0]);
  rg_partial_product1_1<= fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[11:6]);
  rg_partial_product2_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[17:12]);
  rg_partial_product3_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[23:18]);
  rg_partial_product4_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[29:24]);
  rg_partial_product5_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[35:30]);
  rg_partial_product6_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[41:36]);
  rg_partial_product7_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},sfdB1[47:42]);
  rg_partial_product8_1<=fn_gen_pp_dp({1'b0,(|expA1),sfdA1},{1'b0,(|expB1),sfdB1[51:48]});

  rg_operands3_1 <= rg_operands3;
endrule
```

Stage 2:

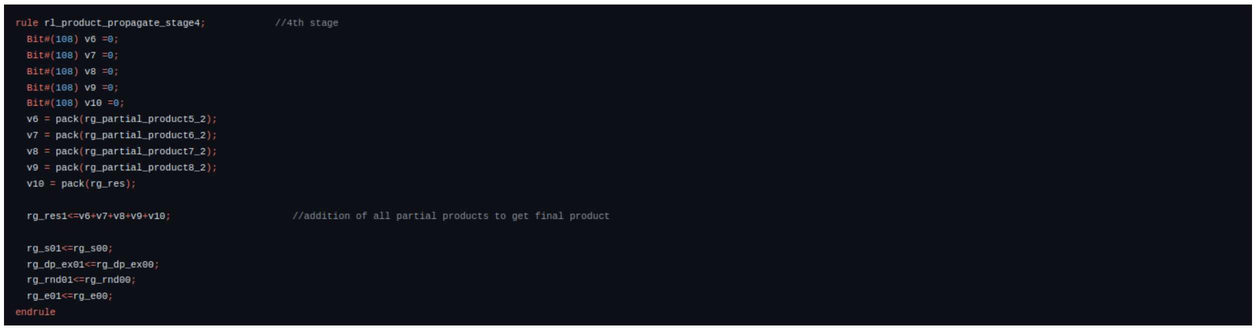
This is the stage which performs the addition to calculate the partial products.



```
rule rl_partial_product_stage_2;
  match {.opA, .opB, .rmode} = rg_operands3_1;                          // evaluating partial products(adding the 2 halves) for mantissa, 2nd stage
  Bit#(11) expA1 = 0;
  Bit#(11) expB1 = 0;
  Bit#(52) sfdA1 = 0;
  Bit#(52) sfdB1 = 0;

  sfdA1 = opA.sfd;
  expA1 = opA.exp;
  sfdB1 = opB.sfd;
  expB1 = opB.exp;

  rg_partial_product0 <= rg_partial_product0_1[107:0]+rg_partial_product0_1[215:108];
  rg_partial_product1 <= {(rg_partial_product1_1[107:0]+rg_partial_product1_1[215:108])[101:0],6'd0};
  rg_partial_product2 <= {(rg_partial_product2_1[107:0]+rg_partial_product2_1[215:108])[95:0],12'd0};
  rg_partial_product3 <= {(rg_partial_product3_1[107:0]+rg_partial_product3_1[215:108])[89:0],18'd0};
  rg_partial_product4 <= {(rg_partial_product4_1[107:0]+rg_partial_product4_1[215:108])[83:0],24'd0};
  rg_partial_product5 <= {(rg_partial_product5_1[107:0]+rg_partial_product5_1[215:108])[77:0],30'd0};
  rg_partial_product6 <= {(rg_partial_product6_1[107:0]+rg_partial_product6_1[215:108])[71:0],36'd0};
  rg_partial_product7 <= {(rg_partial_product7_1[107:0]+rg_partial_product7_1[215:108])[65:0],42'd0};
  rg_partial_product8 <= {(rg_partial_product8_1[107:0]+rg_partial_product8_1[215:108])[59:0],48'd0};

  rg_dp_ex0<=rg_dp_ex02;
  rg_s0<=rg_s02;
  rg_rnd0<=rg_rnd02;
  rg_e0<=rg_e02;

endrule
```

Stage 3 and Stage 4:

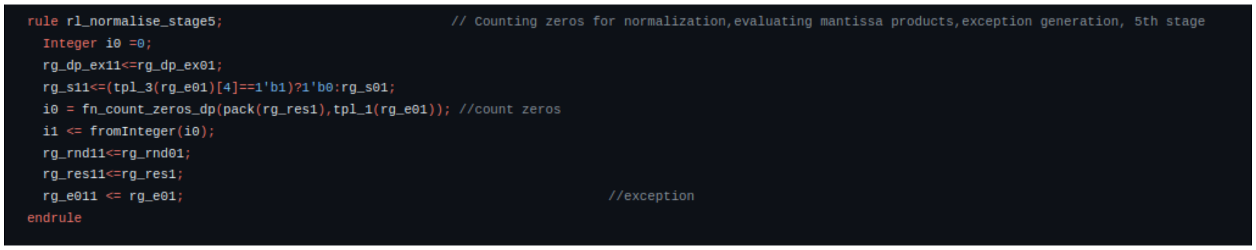
These two stages add the partial products. We have divided into 2 stages keeping in mind speed and resource trade off.



```
rule rl_product_propagate_stage4;          //4th stage
  Bit#(108) v6 =0;
  Bit#(108) v7 =0;
  Bit#(108) v8 =0;
  Bit#(108) v9 =0;
  Bit#(108) v10 =0;
  v6 = pack(rg_partial_product5_2);
  v7 = pack(rg_partial_product6_2);
  v8 = pack(rg_partial_product7_2);
  v9 = pack(rg_partial_product8_2);
  v10 = pack(rg_res);

  rg_res1<=v6+v7+v8+v9+v10;                     //addition of all partial products to get final product

  rg_s01<=rg_s00;
  rg_dp_ex01<=rg_dp_ex00;
  rg_rnd01<=rg_rnd00;
  rg_e01<=rg_e00;
endrule
```

Stage 5:

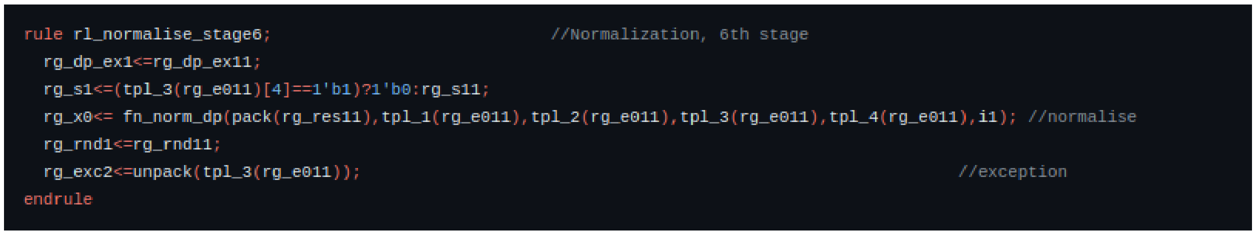
This stage counts the necessary zeros for the normalization stage.



```
rule rl_normalise_stage5;                         // Counting zeros for normalization,evaluating mantissa products,exception generation, 5th stage
  Integer i0 =0;
  rg_dp_ex11<=rg_dp_ex01;
  rg_s11<=(tpl_3(rg_e01)[4]==1'b1)?1'b0:rg_s01;
  i0 = fn_count_zeros_dp(pack(rg_res1),tpl_1(rg_e01)); //count zeros
  i1 <= fromInteger(i0);
  rg_rnd11<=rg_rnd01;
  rg_res11<=rg_res1;
  rg_e011 <= rg_e01;                                            //exception
endrule
```

Stage 6:

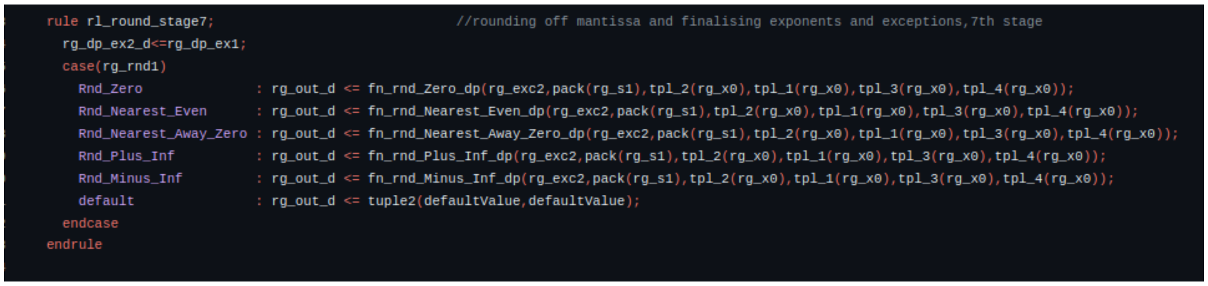
This is the normalization stage.



```
rule rl_normalise_stage6;                               //Normalization, 6th stage
  rg_dp_ex1<=rg_dp_ex11;
  rg_s1<=(tpl_3(rg_e011)[4]==1'b1)?1'b0:rg_s11;
  rg_x0<= fn_norm_dp(pack(rg_res11),tpl_1(rg_e011),tpl_2(rg_e011),tpl_3(rg_e011),tpl_4(rg_e011),i1); //normalise
  rg_rnd1<=rg_rnd11;
  rg_exc2<=unpack(tpl_3(rg_e011));                                                     //exception
endrule
```

Stage 7:

This is the handling of the exceptions stage which performs the rounding of to 0, infinity and -infinity.



```
rule rl_round_stage7;                               //rounding off mantissa and finalising exponents and exceptions,7th stage
  rg_dp_ex2_d<=rg_dp_ex1;
  case(rg_rnd1)
    Rnd_Zero              : rg_out_d <= fn_rnd_Zero_dp(rg_exc2,pack(rg_s1),tpl_2(rg_x0),tpl_1(rg_x0),tpl_3(rg_x0),tpl_4(rg_x0));
    Rnd_Nearest_Even      : rg_out_d <= fn_rnd_Nearest_Even_dp(rg_exc2,pack(rg_s1),tpl_2(rg_x0),tpl_1(rg_x0),tpl_3(rg_x0),tpl_4(rg_x0));
    Rnd_Nearest_Away_Zero : rg_out_d <= fn_rnd_Nearest_Away_Zero_dp(rg_exc2,pack(rg_s1),tpl_2(rg_x0),tpl_1(rg_x0),tpl_3(rg_x0),tpl_4(rg_x0));
    Rnd_Plus_Inf          : rg_out_d <= fn_rnd_Plus_Inf_dp(rg_exc2,pack(rg_s1),tpl_2(rg_x0),tpl_1(rg_x0),tpl_3(rg_x0),tpl_4(rg_x0));
    Rnd_Minus_Inf         : rg_out_d <= fn_rnd_Minus_Inf_dp(rg_exc2,pack(rg_s1),tpl_2(rg_x0),tpl_1(rg_x0),tpl_3(rg_x0),tpl_4(rg_x0));
    default               : rg_out_d <= tuple2(defaultValue,defaultValue);
  endcase
endrule
```

6. Verification Strategies:

We have two stages of verification:

- Bluespec System Verilog Verification: Here we test whether we are getting the output after every clock cycle to ensure a fast multiplier.
- Cocotb Framework base Verification : Here we test the logic of our code by verifying it with a golden test case created in python.

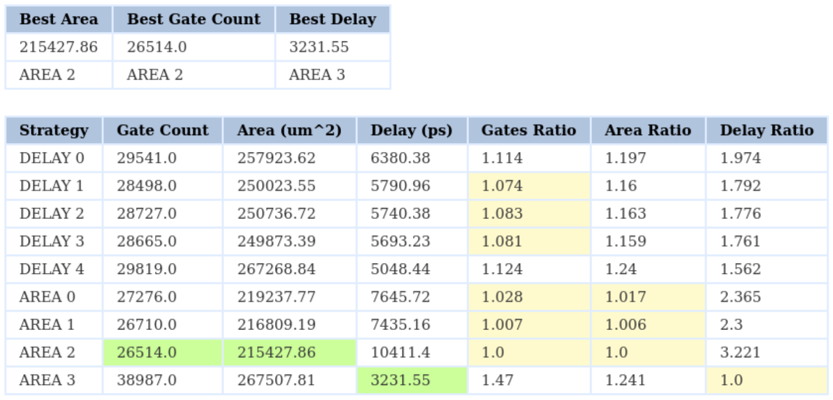
We will be sending the data according every clock cycle and we will be getting the data after every clock cycle post cycle #6. We have verified the code verilog file using cocotb framework. We have checked for corner cases like -inifinity ,+infiinty and 0.

7.Area Estimation:

For synthesis of our design, We first generate verilog modules from the bluespec module. Then, We use the open tool framework **OpenLane** to generate the synthesis reports for the given module. We run various area estimation strategies that provide us with a variety of optimizations. We use the standard cell library **sky130_fd_sc_hd (130nm)** provided as a default from the openlane optimization.

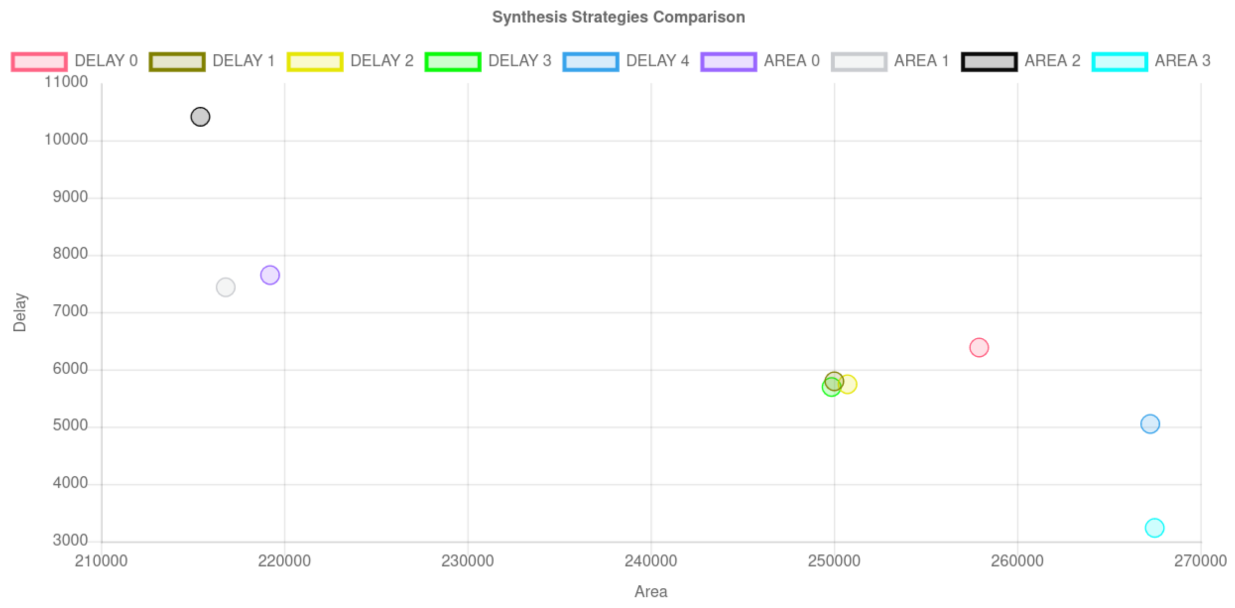
The details are listed below.

Best Area	215427.86 um^2
Best Gate Count	26514
Best Delay	3231.55 ps



Best Area	Best Gate Count	Best Delay
215427.86	26514.0	3231.55
AREA 2	AREA 2	AREA 3

Strategy	Gate Count	Area (um^2)	Delay (ps)	Gates Ratio	Area Ratio	Delay Ratio
DELAY 0	29541.0	257923.62	6380.38	1.114	1.197	1.974
DELAY 1	28498.0	250023.55	5790.96	1.074	1.16	1.792
DELAY 2	28727.0	250736.72	5740.38	1.083	1.163	1.776
DELAY 3	28665.0	249873.39	5693.23	1.081	1.159	1.761
DELAY 4	29819.0	267268.84	5048.44	1.124	1.24	1.562
AREA 0	27276.0	219237.77	7645.72	1.028	1.017	2.365
AREA 1	26710.0	216809.19	7435.16	1.007	1.006	2.3
AREA 2	26514.0	215427.86	10411.4	1.0	1.0	3.221
AREA 3	38987.0	267507.81	3231.55	1.47	1.241	1.0



8. Advantages:

- **Increased Precision:** The use of dual precision floating-point arithmetic ensures higher precision in the multiplication results, critical for applications where accuracy is paramount.
- **Reduced Latency:** Pipelining reduces the latency of the multiplication process, allowing for faster calculations.
- **Efficient Resource Utilization:** Booth's algorithm reduces the number of partial products, leading to more efficient use of computational resources.

9. Applications:

The Dual Precision Floating Point Pipelined Booth's Multiplier finds applications in various fields, including:

- **Scientific Computing:** Particularly in simulations and numerical analysis where high precision is essential.
- **Digital Signal Processing:** Accelerating complex signal processing algorithms that involve extensive multiplication operations.
- **Financial Modeling:** For accurate and efficient handling of financial computations.

10. Challenges and Considerations:

- **Complexity:** The design and implementation of a dual precision floating-point pipelined multiplier can be complex, requiring careful consideration of timing, resource utilization, and precision issues.
- **Power Consumption:** Pipelined architectures may consume more power, necessitating energy-efficient design considerations.

11. Conclusion:

The Dual Precision Floating Point Pipelined Booth's Multiplier represents a significant advancement in the field of digital arithmetic, offering a balance between precision, speed, and resource efficiency. Its applications in scientific computing, digital signal processing, and financial modeling make it a valuable component in modern computational systems.

12. Future Developments:

Future work may focus on further optimizing the design for power efficiency, exploring alternative multiplication algorithms, and adapting the multiplier for emerging technologies such as quantum computing.

In conclusion, the Dual Precision Floating Point Pipelined Booth's Multiplier is a powerful tool that combines the strengths of Booth's algorithm and dual precision floating-point arithmetic, making it a key component in high-performance computing systems.