Tung Nguyen
EE278

San Jose State University
FALL2024

**Department of Electrical Engineering**

**MINI PROJECT**

# Design and Testing of Pipelined and Unpipelined MAC Unit

---

## I. Introduction

This project aimed to design, implement, and test a **Multiply-Accumulate (MAC) unit** for **half-precision floating-point numbers** based on the IEEE 754 standard. The half-precision format, which is 16 bits wide, is commonly used in machine learning, digital signal processing (DSP), and embedded systems due to its balanced trade-off between precision and memory usage.

The project was divided into two phases:

1. **Unpipelined MAC**: A simple design that completes one MAC operation per clock cycle.
2. **Pipelined MAC**: A six-stage pipelined version that enhances performance by allowing multiple MAC operations to be processed concurrently.

Both versions were implemented in Verilog, tested using a variety of floating-point test cases, and validated for accuracy. This report provides a comprehensive breakdown of the design process, testing results, encountered issues, and the role that **Large Language Models (LLMs)** like ChatGPT played in assisting the design and development process.

## II. Half-Precision Floating-Point Format

The **IEEE 754 half-precision floating-point format** consists of 16 bits:

- **1 sign bit**
- **5 exponent bits** (biased by 15)
- **10 mantissa bits** (with an implicit leading '1')

This format allows for a range of numbers that balance memory efficiency with reasonable precision. The representation of a floating-point number is:

$$\text{Value} = (-1)^S \times 2^{(E-15)} \times (1 + \frac{M}{1024})$$

Where:

- **S** is the sign bit.
- **E** is the exponent.
- **M** is the mantissa (or fractional part).

For example, the half-precision binary representation of **1.0** is 0011_1100_0000_0000, which corresponds to:

$$1.0 = (-1)^0 \times 2^{(15-15)} \times (1 + \frac{0}{1024}) = 1.0$$

This project utilized this format to design the MAC units.

## III. Unpipelined MAC Design

The **unpipelined MAC** was the first phase of the project. It was designed to perform multiplication and accumulation in a single clock cycle. The key components of the unpipelined MAC were:

- **Floating-Point Multiplier**: This module takes two half-precision floating-point inputs, multiplies their mantissas, adds the exponents, and adjusts the product by normalizing the result and handling overflow/underflow cases.
- **Floating-Point Adder**: This module adds the product from the multiplier to the accumulator. The addition involves aligning the exponents of both numbers by shifting the mantissas, performing the addition or subtraction depending on the signs of the inputs, and normalizing the result.

### 1. Unpipelined Multiplier

The multiplier works as follows:

1. Extract the sign, exponent, and mantissa from both inputs.
2. Multiply the mantissas.
3. Add the exponents and adjust for the bias (15 for half-precision).
4. Normalize the result by checking for any leading 1’s in the product.
5. Assemble the result with the new sign, exponent, and mantissa.

Verilog code for the **floating-point multiplier**:

```
module fp_multiplier(
    input [15:0] A,   // First 16-bit floating-point input
    input [15:0] B,   // Second 16-bit floating-point input
    output reg [15:0] Product  // 16-bit floating-point product
);

// Extract sign, exponent, and mantissa from inputs
wire sign_A, sign_B, sign_P;
wire [4:0] exp_A, exp_B, exp_P;
wire [10:0] mantissa_A, mantissa_B;
wire [21:0] mantissa_P;
reg [4:0] exp_P_reg;
reg [10:0] mantissa_P_reg;
```

```
assign sign_A = A[15];
assign exp_A = A[14:10];
assign mantissa_A = {1'b1, A[9:0]};  // Implicit leading 1 in mantissa

assign sign_B = B[15];
assign exp_B = B[14:10];
assign mantissa_B = {1'b1, B[9:0]};  // Implicit leading 1 in mantissa

// Determine the sign of the product
assign sign_P = sign_A ^ sign_B;

// Multiply mantissas
assign mantissa_P = mantissa_A * mantissa_B;

// Add exponents and adjust for bias
assign exp_P = exp_A + exp_B - 15;  // Bias correction (half-precision bias is 15)
```

```
// Normalize the product and adjust the exponent
always @(*) begin
    if (mantissa_P[21]) begin
        // Shift and adjust exponent
        mantissa_P_reg = mantissa_P[21:11];
        exp_P_reg = exp_P + 1;
    end else begin
```

```
        mantissa_P_reg = mantissa_P[20:10];
        exp_P_reg = exp_P;
    end
end

// Compose the product
always @(*) begin
    if (exp_P_reg == 0 || exp_P_reg == 31)  // Handle special cases like overflow/underflo
        Product = 16'b0;  // Overflow or underflow results in zero
    else
        Product = {sign_P, exp_P_reg, mantissa_P_reg[9:0]};
end

endmodule
```

**2. Unpipelined Adder**

The adder works as follows:

1. Compare the exponents of both inputs and align them by shifting the mantissas of the smaller number.
2. Add or subtract the mantissas, depending on the signs of the inputs.
3. Normalize the result by adjusting the exponent and mantissa.
4. Handle edge cases like underflow, overflow, and rounding.
5. Verilog code for the **floating-point adder**:

```
module fp_adder(
    input [15:0] A,   // First 16-bit floating-point input
    input [15:0] B,   // Second 16-bit floating-point input
    output reg [15:0] Sum  // 16-bit floating-point sum
);

// Extract sign, exponent, and mantissa from inputs
wire sign_A, sign_B, sign_S;
wire [4:0] exp_A, exp_B, exp_S;
wire [10:0] mantissa_A, mantissa_B;
```

```
reg [10:0] mantissa_S;
wire [10:0] mantissa_A_shifted, mantissa_B_shifted;
wire [4:0] larger_exp, exp_diff;
reg [4:0] exp_S_reg;

assign sign_A = A[15];
assign exp_A = A[14:10];
assign mantissa_A = {1'b1, A[9:0]};  // Implicit leading 1 in mantissa
```

```
assign sign_B = B[15];
assign exp_B = B[14:10];
assign mantissa_B = {1'b1, B[9:0]};  // Implicit leading 1 in mantissa

// Compare exponents and shift mantissas
assign exp_diff = (exp_A > exp_B) ? exp_A - exp_B : exp_B - exp_A;
assign larger_exp = (exp_A > exp_B) ? exp_A : exp_B;

assign mantissa_A_shifted = (exp_A > exp_B) ? mantissa_A : mantissa_A >> exp_diff;
assign mantissa_B_shifted = (exp_B > exp_A) ? mantissa_B : mantissa_B >> exp_diff;
```

```
// Add or subtract mantissas based on the sign
always @(*) begin
    if (sign_A == sign_B) begin
        mantissa_S = mantissa_A_shifted + mantissa_B_shifted;
        exp_S_reg = larger_exp;
    end else begin
        if (mantissa_A_shifted > mantissa_B_shifted) begin
            mantissa_S = mantissa_A_shifted - mantissa_B_shifted;
```

```
            exp_S_reg = larger_exp;
        end else begin
            mantissa_S = mantissa_B_shifted - mantissa_A_shifted;
            exp_S_reg = larger_exp;
        end
    end
end
```

```
// Normalize the result and assign final sum
always @(*) begin
    if (mantissa_S[10]) begin
        exp_S_reg = exp_S_reg + 1;
        mantissa_S = mantissa_S >> 1;
    end
    Sum = {sign_A, exp_S_reg, mantissa_S[9:0]};
end

endmodule
```

## 3. Unpipelined MAC Architecture

The unpipelined MAC was a simple, direct combination of the multiplier and adder. The multiplication result is fed into the adder, which adds the result to the accumulator (third input). The result is produced in a single clock cycle, making this version suitable for designs where low latency is required but throughput is not critical.

Verilog code for the **full MAC unit:**

```
module fp_mac(
    input [15:0] A,   // First 16-bit floating-point input
    input [15:0] B,   // Second 16-bit floating-point input
    input [15:0] C,   // Accumulator input
    output [15:0] Result  // Final MAC result
);

// Intermediate wires for multiplication and addition
wire [15:0] product;
wire [15:0] sum;

// Instantiate the multiplier
fp_multiplier multiplier (
    .A(A),
    .B(B),
    .Product(product)
);
```

```
// Instantiate the adder (adds product to accumulator C)
fp_adder adder (
    .A(product),
    .B(C),
    .Sum(sum)
);

// Final output is the sum
assign Result = sum;

endmodule
```

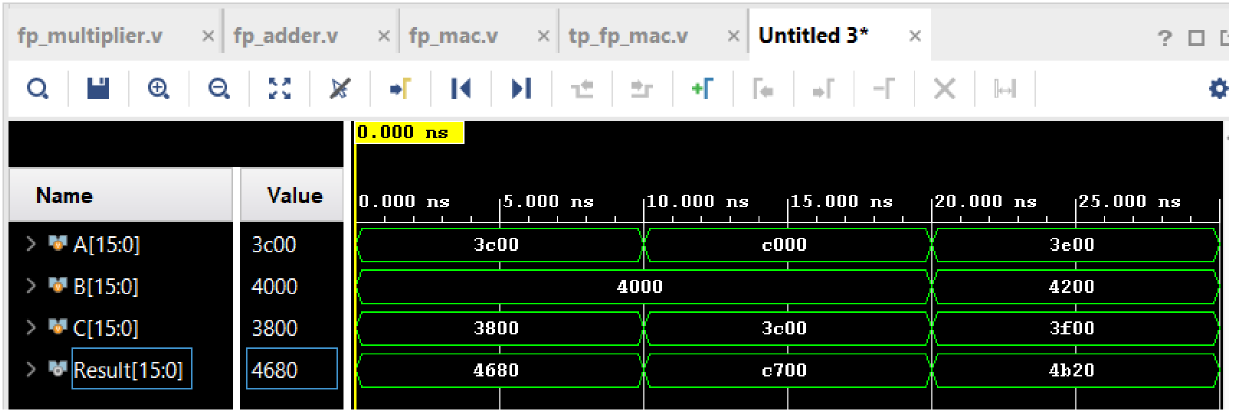
**Testbench** to verify the MAC unit:

```
module tb_fp_mac;

    // Inputs
    reg [15:0] A;
    reg [15:0] B;
    reg [15:0] C;

    // Output
    wire [15:0] Result;

    // Instantiate the MAC unit
    fp_mac uut (
        .A(A),
        .B(B),
        .C(C),
        .Result(Result)
    );
```

```
    initial begin
        // Initialize inputs
        A = 16'h3C00;  // 1.0 in half-precision
        B = 16'h4000;  // 2.0 in half-precision
        C = 16'h3800;  // 0.5 in half-precision
        #10;

        A = 16'hC000;  // -2.0 in half-precision
        B = 16'h4000;  // 2.0 in half-precision
        C = 16'h3C00;  // 1.0 in half-precision

        #10;

        A = 16'h3E00;  // 1.5 in half-precision
        B = 16'h4200;  // 3.5 in half-precision
        C = 16'h3F00;  // 1.75 in half-precision

        #10;

        $finish;
    end
```

```
    // Monitor changes in the output
    initial begin
        $monitor("At time %0dns: A = %h, B = %h, C = %h, Result = %h", $time, A, B, C, Res
    end

endmodule
```

## Testbench with MAC



| Name | Value | 0–10 ns | 10–20 ns | 20–30 ns |
|---|---|---|---|---|
| A[15:0] | 3c00 | 3c00 | c000 | 3e00 |
| B[15:0] | 4000 | 4000 | 4000 | 4200 |
| C[15:0] | 3800 | 3800 | 3c00 | 3f00 |
| Result[15:0] | 4680 | 4680 | c700 | 4b20 |

## IV. Pipelined MAC Design

To improve performance and increase throughput, the second phase of the project was to implement a **6-stage pipelined MAC unit**. The pipelined MAC breaks down the multiplication and addition processes into smaller steps, enabling multiple operations to be processed in parallel.

### 1. Pipeline Stages

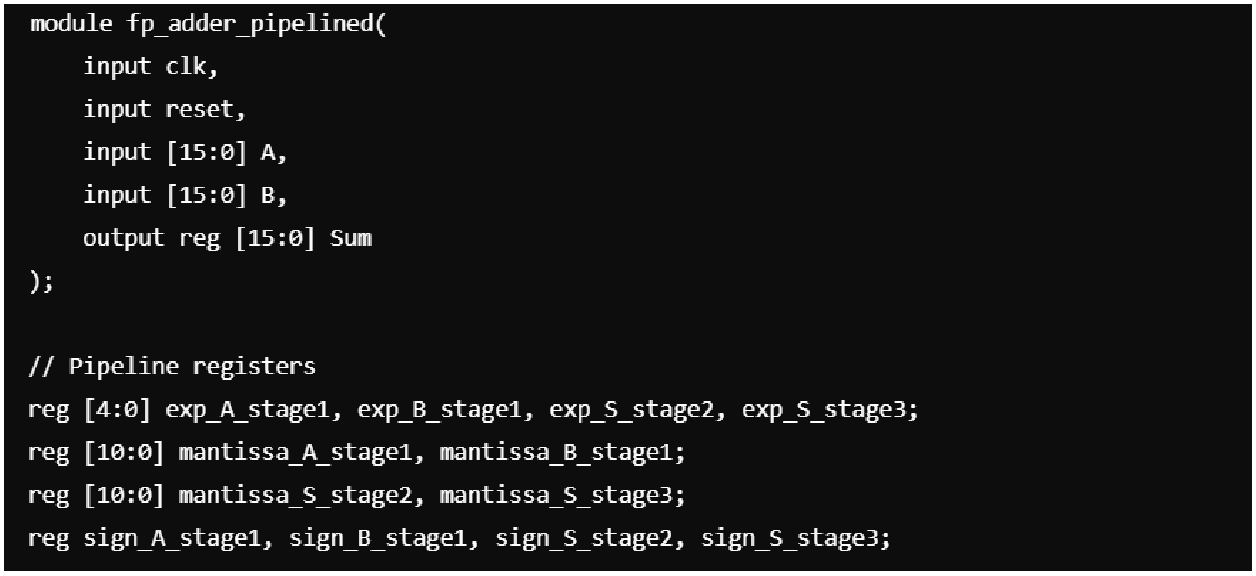
The pipelined MAC is broken down into the following stages:

- **Multiplier Stages**:
    1. **Stage 1**: Extract the sign, exponent, and mantissa of the inputs and multiply the mantissas.
    2. **Stage 2**: Add the exponents and compute the intermediate result.
    3. **Stage 3**: Normalize the product and adjust the exponent.
- **Adder Stages**:
    1. **Stage 4**: Align the exponents of the product and the accumulator by shifting the mantissas.
    2. **Stage 5**: Add or subtract the aligned mantissas.
    3. **Stage 6**: Normalize the final result and handle rounding.

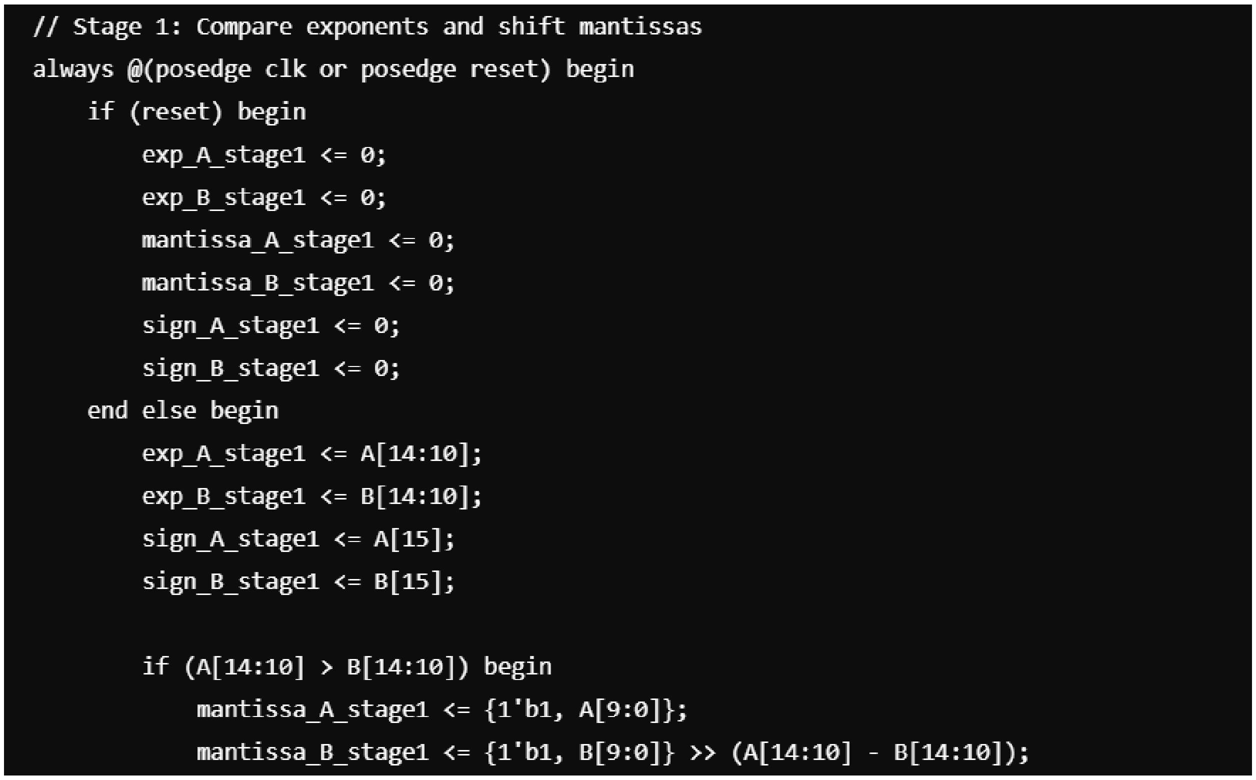
### 2. Pipelined Design

Each stage in the pipeline registers its intermediate results, allowing the next stage to begin processing a new set of inputs while the previous stages are still executing. This parallelism increases the throughput significantly, as a new operation can be initiated on every clock cycle, even though the final result takes several cycles to propagate through the pipeline.

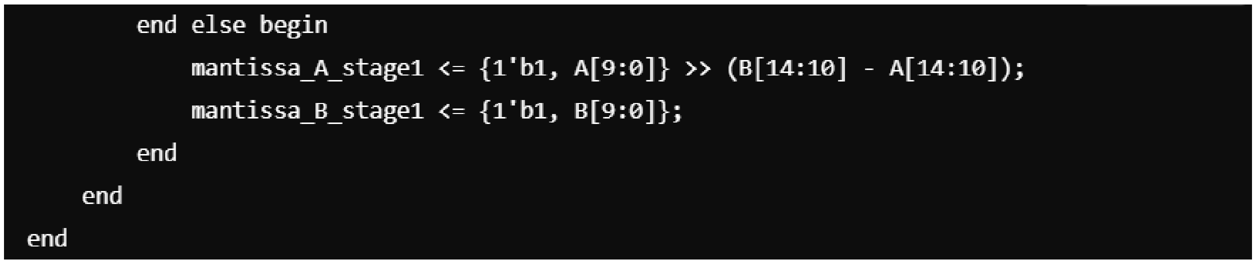
Verilog code for the **pipelined adder**:



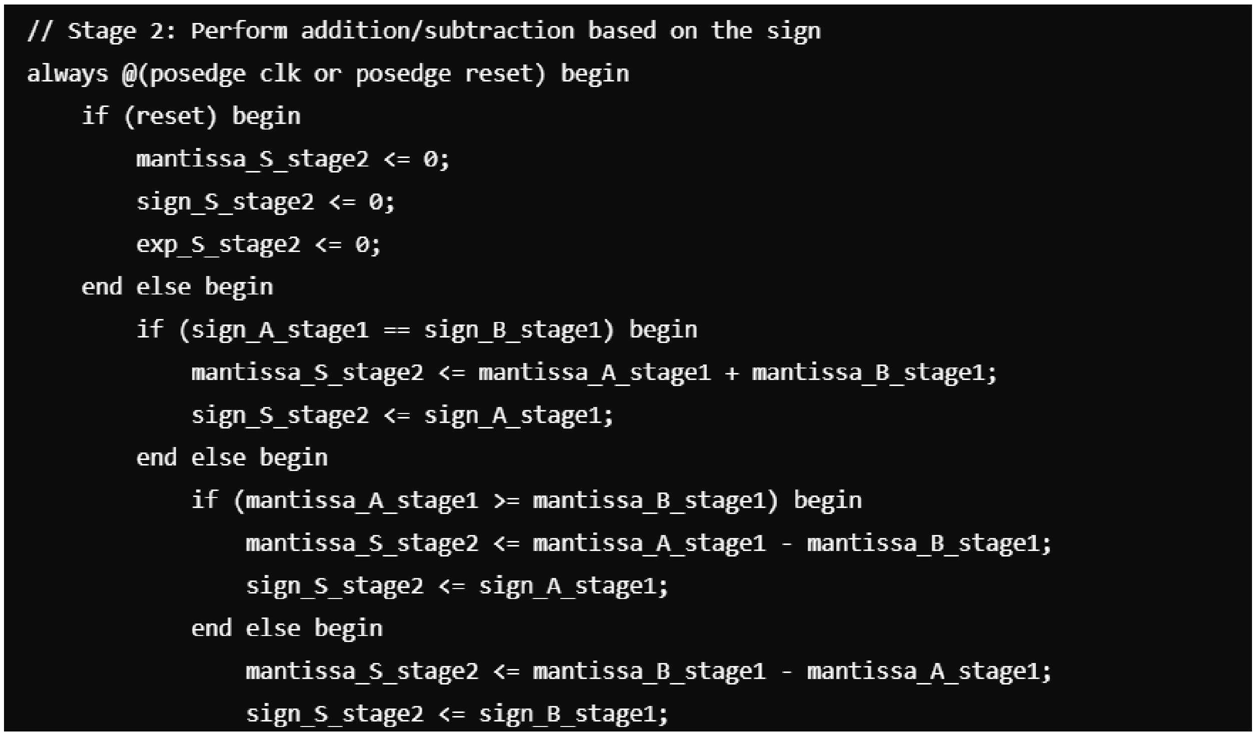
```
module fp_adder_pipelined(
    input clk,
    input reset,
    input [15:0] A,
    input [15:0] B,
    output reg [15:0] Sum
);

// Pipeline registers
reg [4:0] exp_A_stage1, exp_B_stage1, exp_S_stage2, exp_S_stage3;
reg [10:0] mantissa_A_stage1, mantissa_B_stage1;
reg [10:0] mantissa_S_stage2, mantissa_S_stage3;
reg sign_A_stage1, sign_B_stage1, sign_S_stage2, sign_S_stage3;
```



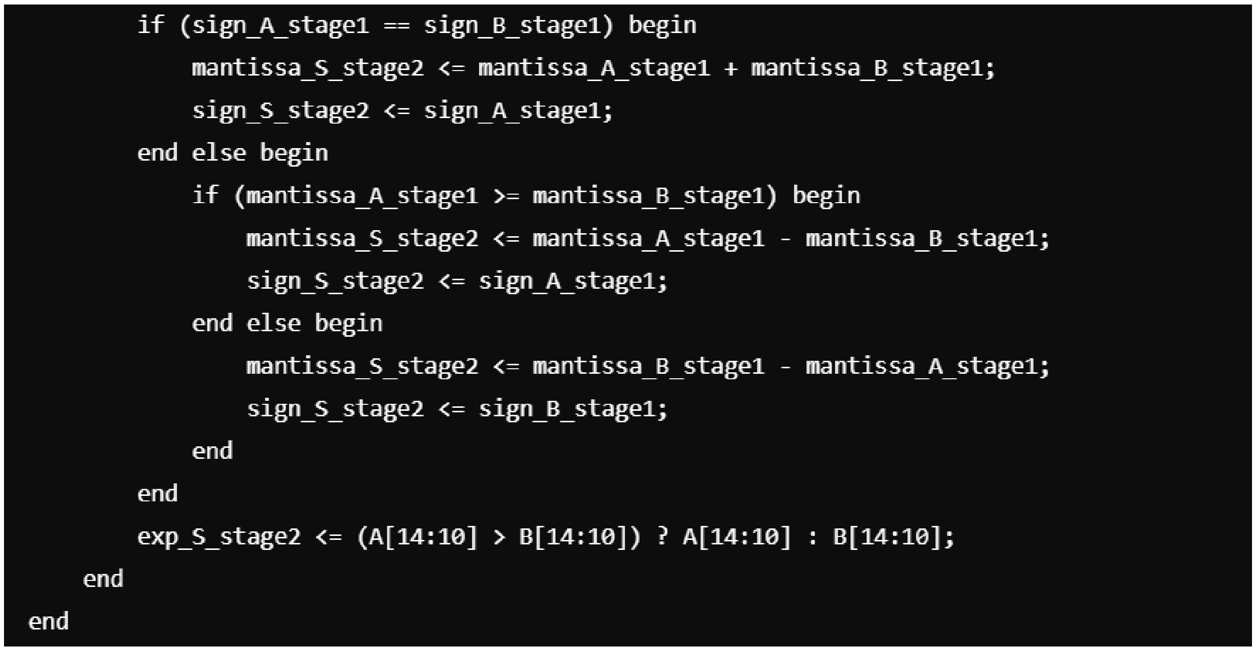
```
// Stage 1: Compare exponents and shift mantissas
always @(posedge clk or posedge reset) begin
    if (reset) begin
        exp_A_stage1 <= 0;
        exp_B_stage1 <= 0;
        mantissa_A_stage1 <= 0;
        mantissa_B_stage1 <= 0;
        sign_A_stage1 <= 0;
        sign_B_stage1 <= 0;
    end else begin
        exp_A_stage1 <= A[14:10];
        exp_B_stage1 <= B[14:10];
        sign_A_stage1 <= A[15];
        sign_B_stage1 <= B[15];

        if (A[14:10] > B[14:10]) begin
            mantissa_A_stage1 <= {1'b1, A[9:0]};
            mantissa_B_stage1 <= {1'b1, B[9:0]} >> (A[14:10] - B[14:10]);
```



```
        end else begin
            mantissa_A_stage1 <= {1'b1, A[9:0]} >> (B[14:10] - A[14:10]);
            mantissa_B_stage1 <= {1'b1, B[9:0]};
        end
    end
end
```

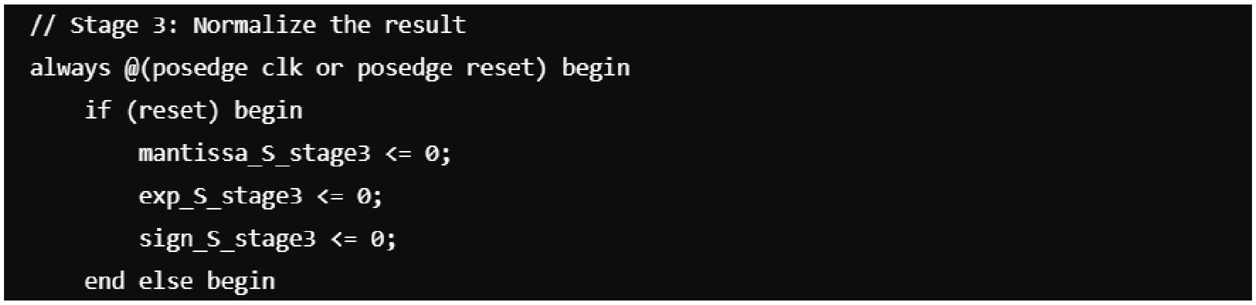


```
// Stage 2: Perform addition/subtraction based on the sign
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_S_stage2 <= 0;
        sign_S_stage2 <= 0;
        exp_S_stage2 <= 0;
    end else begin
        if (sign_A_stage1 == sign_B_stage1) begin
            mantissa_S_stage2 <= mantissa_A_stage1 + mantissa_B_stage1;
            sign_S_stage2 <= sign_A_stage1;
        end else begin
            if (mantissa_A_stage1 >= mantissa_B_stage1) begin
                mantissa_S_stage2 <= mantissa_A_stage1 - mantissa_B_stage1;
                sign_S_stage2 <= sign_A_stage1;
            end else begin
                mantissa_S_stage2 <= mantissa_B_stage1 - mantissa_A_stage1;
                sign_S_stage2 <= sign_B_stage1;
```



```
        if (sign_A_stage1 == sign_B_stage1) begin
            mantissa_S_stage2 <= mantissa_A_stage1 + mantissa_B_stage1;
            sign_S_stage2 <= sign_A_stage1;
        end else begin
            if (mantissa_A_stage1 >= mantissa_B_stage1) begin
                mantissa_S_stage2 <= mantissa_A_stage1 - mantissa_B_stage1;
                sign_S_stage2 <= sign_A_stage1;
            end else begin
                mantissa_S_stage2 <= mantissa_B_stage1 - mantissa_A_stage1;
                sign_S_stage2 <= sign_B_stage1;
```

```
            end
        end
        exp_S_stage2 <= (A[14:10] > B[14:10]) ? A[14:10] : B[14:10];
    end
end
```



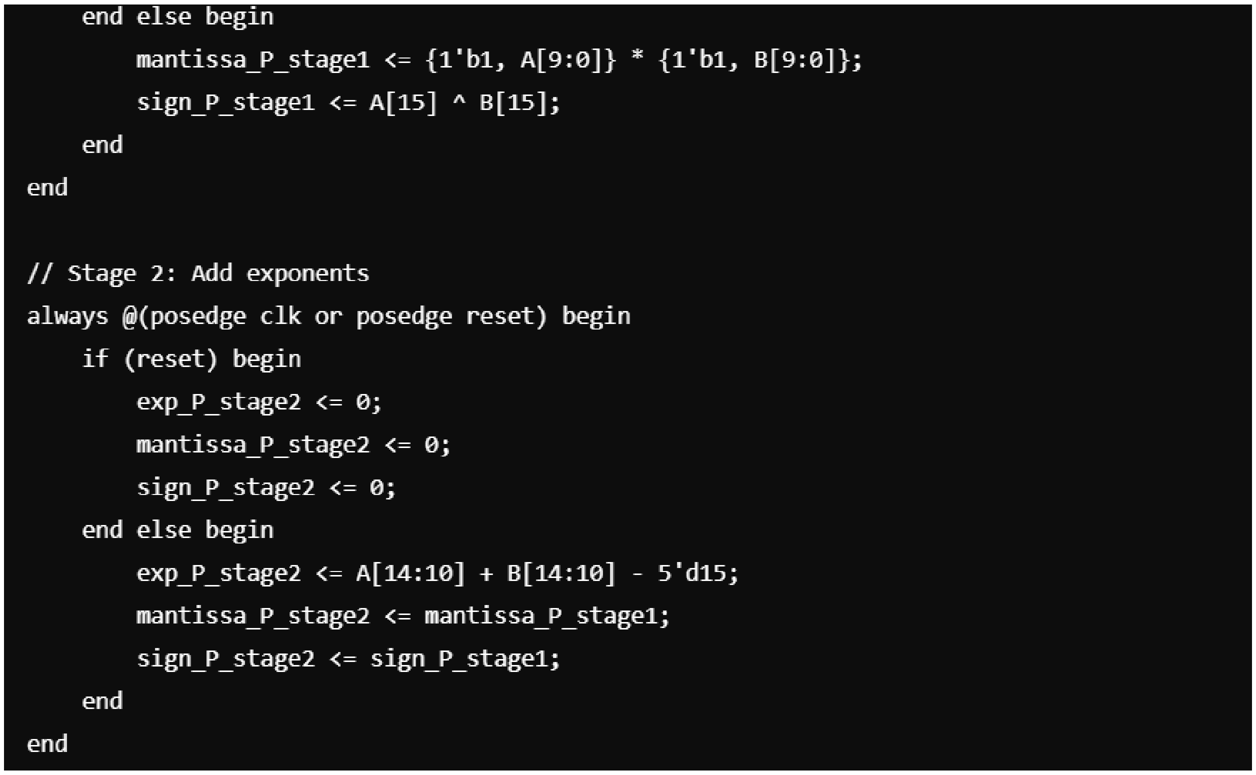
```
// Stage 3: Normalize the result
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_S_stage3 <= 0;
        exp_S_stage3 <= 0;
        sign_S_stage3 <= 0;
    end else begin
```

```
        if (mantissa_S_stage2[10]) begin
            mantissa_S_stage3 <= mantissa_S_stage2 >> 1;
            exp_S_stage3 <= exp_S_stage2 + 1;
        end else begin
            mantissa_S_stage3 <= mantissa_S_stage2;
            exp_S_stage3 <= exp_S_stage2;
        end
        sign_S_stage3 <= sign_S_stage2;
    end
end
```

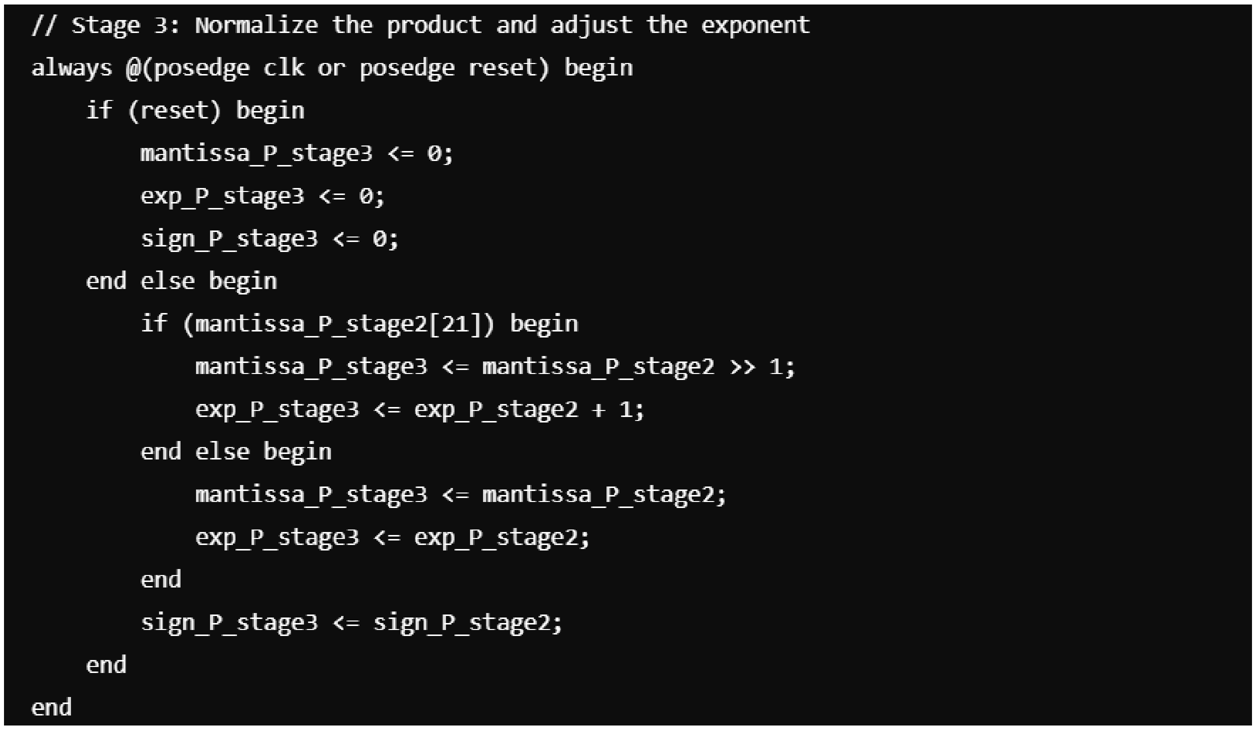
```
// Final output
always @(posedge clk or posedge reset) begin
    if (reset) begin
        Sum <= 0;
    end else begin
        Sum <= {sign_S_stage3, exp_S_stage3, mantissa_S_stage3[9:0]};
    end
end

endmodule
```

Verilog code for the **pipelined multiplier**:

```
module fp_multiplier_pipelined(
    input clk,
    input reset,
    input [15:0] A,
    input [15:0] B,
    output reg [15:0] Product
);

// Pipeline registers
reg [21:0] mantissa_P_stage1, mantissa_P_stage2, mantissa_P_stage3;
reg [4:0] exp_P_stage1, exp_P_stage2, exp_P_stage3;
reg sign_P_stage1, sign_P_stage2, sign_P_stage3;

// Stage 1: Multiply mantissas
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_P_stage1 <= 0;
        sign_P_stage1 <= 0;
```



```
    end else begin
        mantissa_P_stage1 <= {1'b1, A[9:0]} * {1'b1, B[9:0]};
        sign_P_stage1 <= A[15] ^ B[15];
    end
end

// Stage 2: Add exponents
always @(posedge clk or posedge reset) begin
    if (reset) begin
        exp_P_stage2 <= 0;
        mantissa_P_stage2 <= 0;
        sign_P_stage2 <= 0;
    end else begin
        exp_P_stage2 <= A[14:10] + B[14:10] - 5'd15;
        mantissa_P_stage2 <= mantissa_P_stage1;
        sign_P_stage2 <= sign_P_stage1;
    end
end
```



```
// Stage 3: Normalize the product and adjust the exponent
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_P_stage3 <= 0;
        exp_P_stage3 <= 0;
        sign_P_stage3 <= 0;
    end else begin
        if (mantissa_P_stage2[21]) begin
            mantissa_P_stage3 <= mantissa_P_stage2 >> 1;
            exp_P_stage3 <= exp_P_stage2 + 1;
        end else begin
            mantissa_P_stage3 <= mantissa_P_stage2;
            exp_P_stage3 <= exp_P_stage2;
        end
        sign_P_stage3 <= sign_P_stage2;
    end
end
```

```
// Final output
always @(posedge clk or posedge reset) begin
    if (reset) begin
        Product <= 0;
    end else begin
        Product <= {sign_P_stage3, exp_P_stage3, mantissa_P_stage3[9:0]};
    end
end

endmodule
```

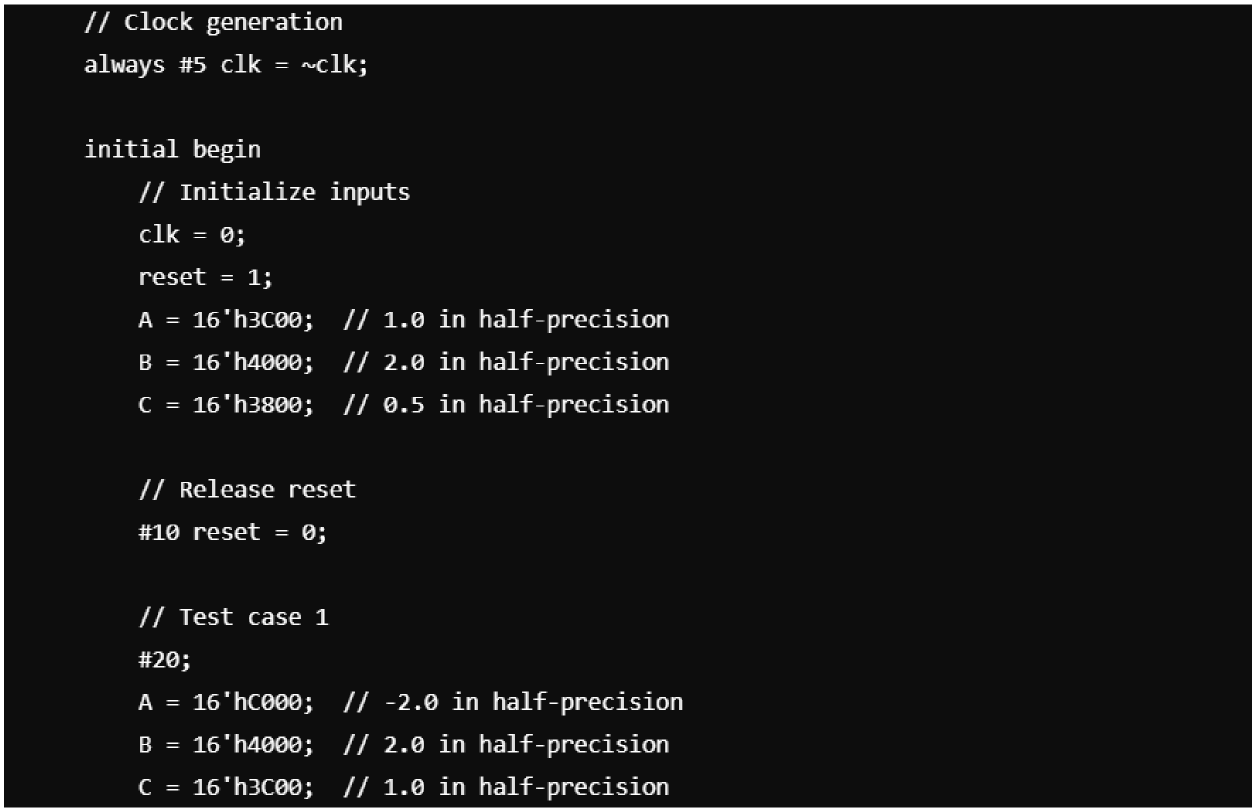
Verilog code for the **pipelined MAC**:

```
module fp_mac_pipelined(
    input clk,
    input reset,
    input [15:0] A,
    input [15:0] B,
    input [15:0] C,  // Accumulator input
    output reg [15:0] Result
);

// Intermediate pipeline registers
wire [15:0] product;
wire [15:0] sum;

// Instantiate the 3-stage multiplier
fp_multiplier_pipelined multiplier (
    .clk(clk),
    .reset(reset),
    .A(A),
    .B(B),
    .Product(product)
);
```

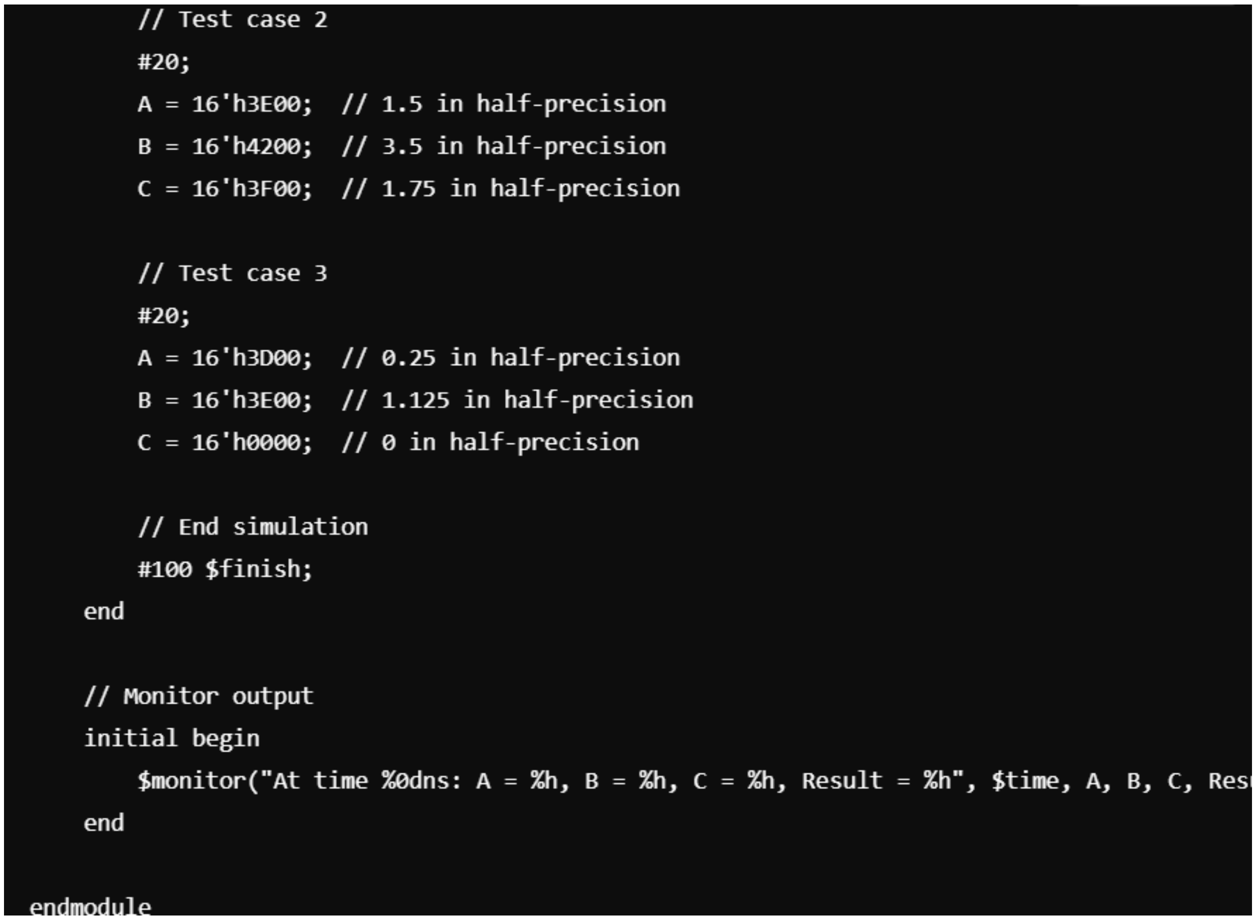
```
// Instantiate the 3-stage adder
fp_adder_pipelined adder (
    .clk(clk),
    .reset(reset),
    .A(product),
    .B(C),
    .Sum(sum)
);

// Final result
always @(posedge clk or posedge reset) begin
    if (reset) begin
        Result <= 0;
    end else begin
        Result <= sum;
    end
end

endmodule
```

Testbench to verify the functionality of the **6-stage pipelined MAC**:

```
module tb_fp_mac_pipelined;

    // Inputs
    reg clk;
    reg reset;
    reg [15:0] A;
    reg [15:0] B;
    reg [15:0] C;
    // Output
    wire [15:0] Result;

    // Instantiate the pipelined MAC unit
    fp_mac_pipelined uut (
        .clk(clk),
        .reset(reset),
        .A(A),
        .B(B),
        .C(C),
        .Result(Result)
```

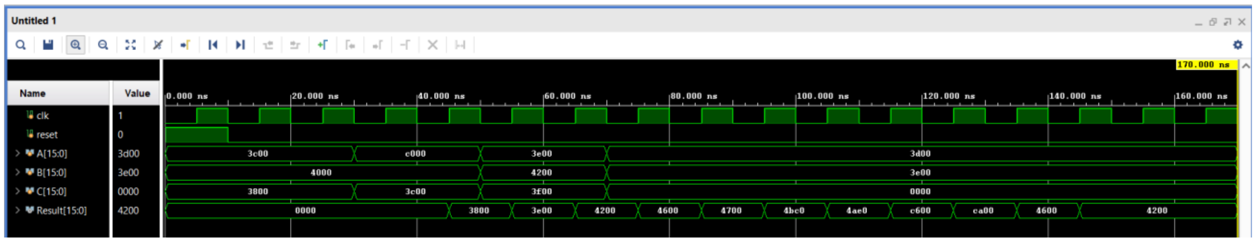


```
    // Clock generation
    always #5 clk = ~clk;

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;
        A = 16'h3C00;  // 1.0 in half-precision
        B = 16'h4000;  // 2.0 in half-precision
        C = 16'h3800;  // 0.5 in half-precision

        // Release reset
        #10 reset = 0;

        // Test case 1
        #20;
        A = 16'hC000;  // -2.0 in half-precision
        B = 16'h4000;  // 2.0 in half-precision
        C = 16'h3C00;  // 1.0 in half-precision
```



```
        // Test case 2
        #20;
        A = 16'h3E00;  // 1.5 in half-precision
        B = 16'h4200;  // 3.5 in half-precision
        C = 16'h3F00;  // 1.75 in half-precision

        // Test case 3
        #20;
        A = 16'h3D00;  // 0.25 in half-precision
        B = 16'h3E00;  // 1.125 in half-precision
        C = 16'h0000;  // 0 in half-precision

        // End simulation
        #100 $finish;
    end

    // Monitor output
    initial begin
        $monitor("At time %0dns: A = %h, B = %h, C = %h, Result = %h", $time, A, B, C, Res
    end

endmodule
```

Simulation result with Vivado



### 3. Challenges in the Pipelined Design

The main challenge in implementing the pipelined MAC was ensuring that the intermediate results were correctly passed between the stages, especially when handling edge cases like:

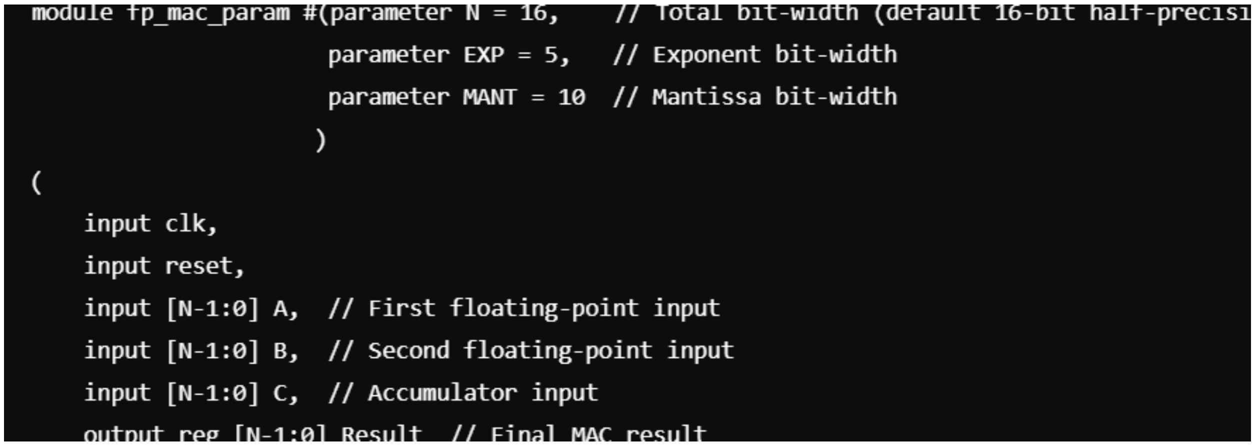
- **Exponent alignment**: Ensuring that the exponents were correctly aligned during the addition phase.
- **Normalization**: Guaranteeing that the results were normalized after multiplication and addition.
- **Latency**: Understanding and accounting for the inherent latency of the pipeline, where a result is delayed due to the multiple stages involved.

---

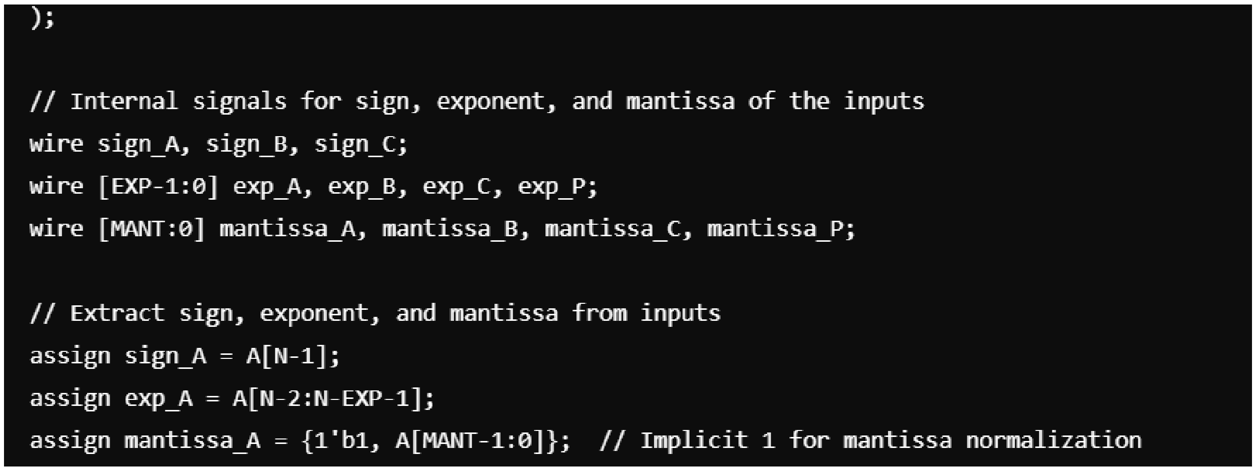
## IV. Parameterizable n-bit MAC

The parameterizable model of the MAC (Multiply-Accumulate) unit that can handle different floating-point formats, such as 16-bit or 32-bit. The key here is to use Verilog's parameter feature to make the design flexible for different bit-widths, including the sign bit, exponent bits, and mantissa bits.

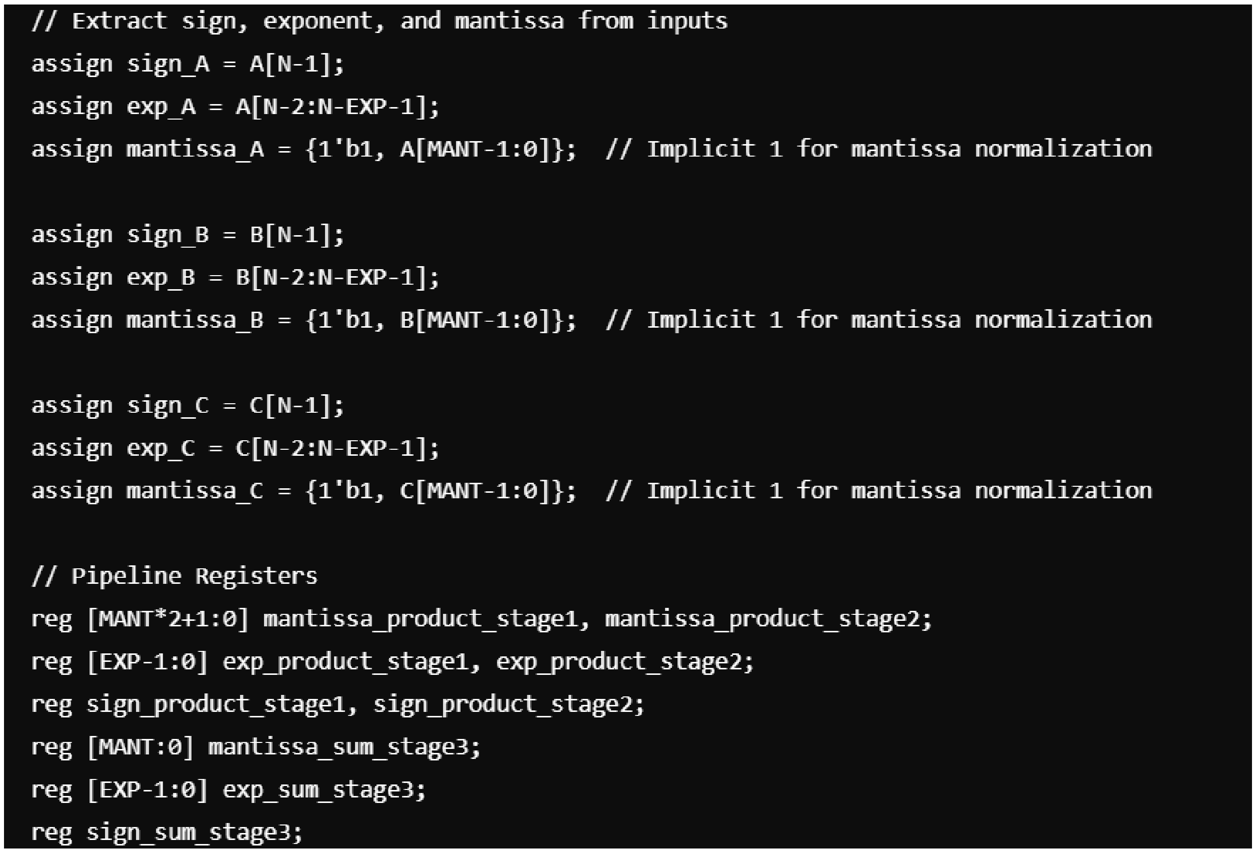
Verilog code for the parameterizable MAC unit



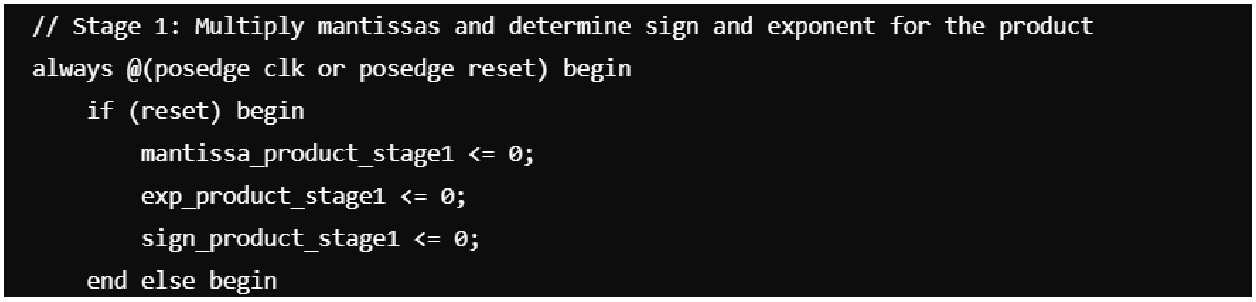
```
module fp_mac_param #(parameter N = 16,    // Total bit-width (default 16-bit half-precisi
                      parameter EXP = 5,   // Exponent bit-width
                      parameter MANT = 10  // Mantissa bit-width
                     )
(
    input clk,
    input reset,
    input [N-1:0] A,  // First floating-point input
    input [N-1:0] B,  // Second floating-point input
    input [N-1:0] C,  // Accumulator input
    output reg [N-1:0] Result  // Final MAC result
```



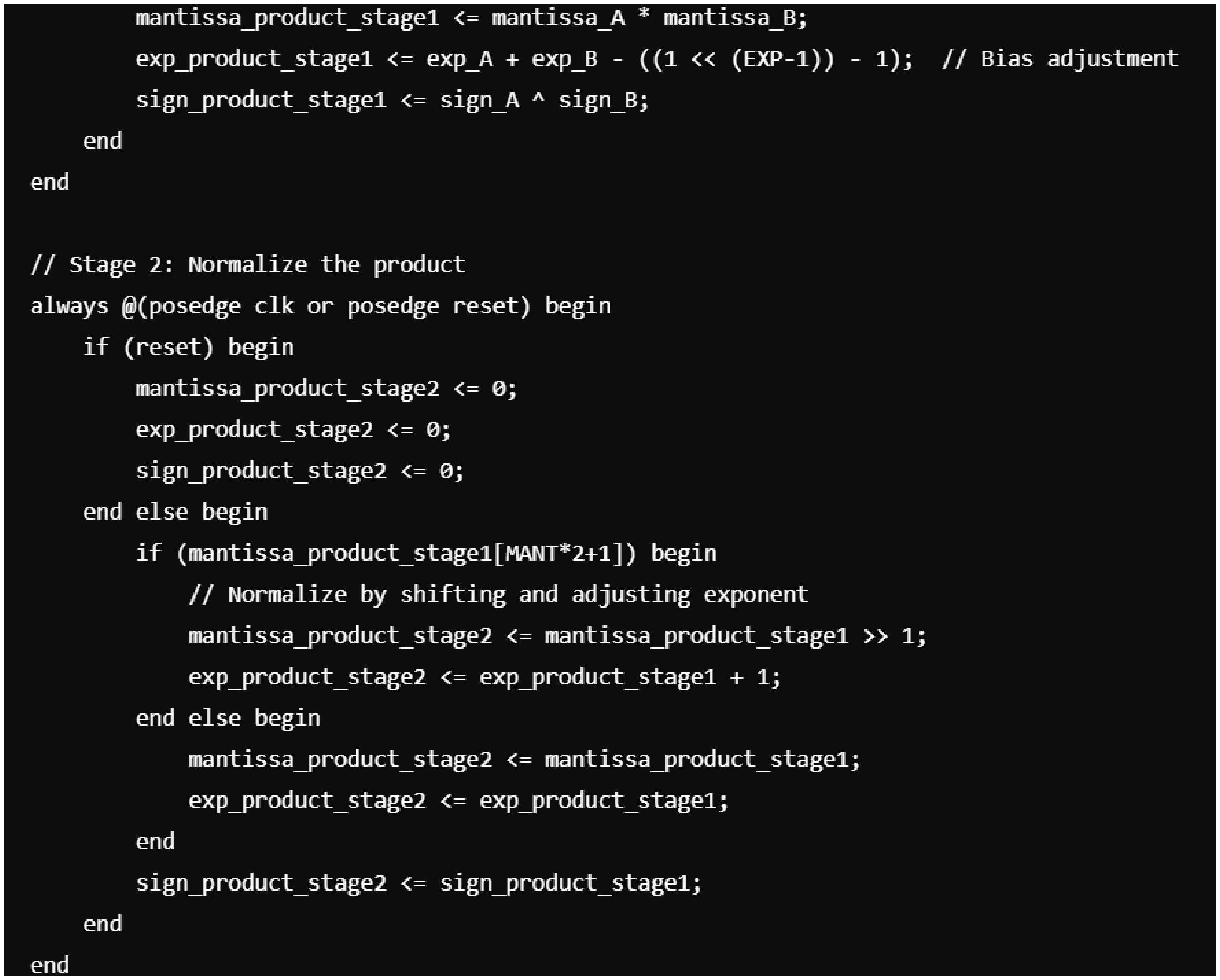
```
);

// Internal signals for sign, exponent, and mantissa of the inputs
wire sign_A, sign_B, sign_C;
wire [EXP-1:0] exp_A, exp_B, exp_C, exp_P;
wire [MANT:0] mantissa_A, mantissa_B, mantissa_C, mantissa_P;

// Extract sign, exponent, and mantissa from inputs
assign sign_A = A[N-1];
assign exp_A = A[N-2:N-EXP-1];
assign mantissa_A = {1'b1, A[MANT-1:0]};  // Implicit 1 for mantissa normalization
```



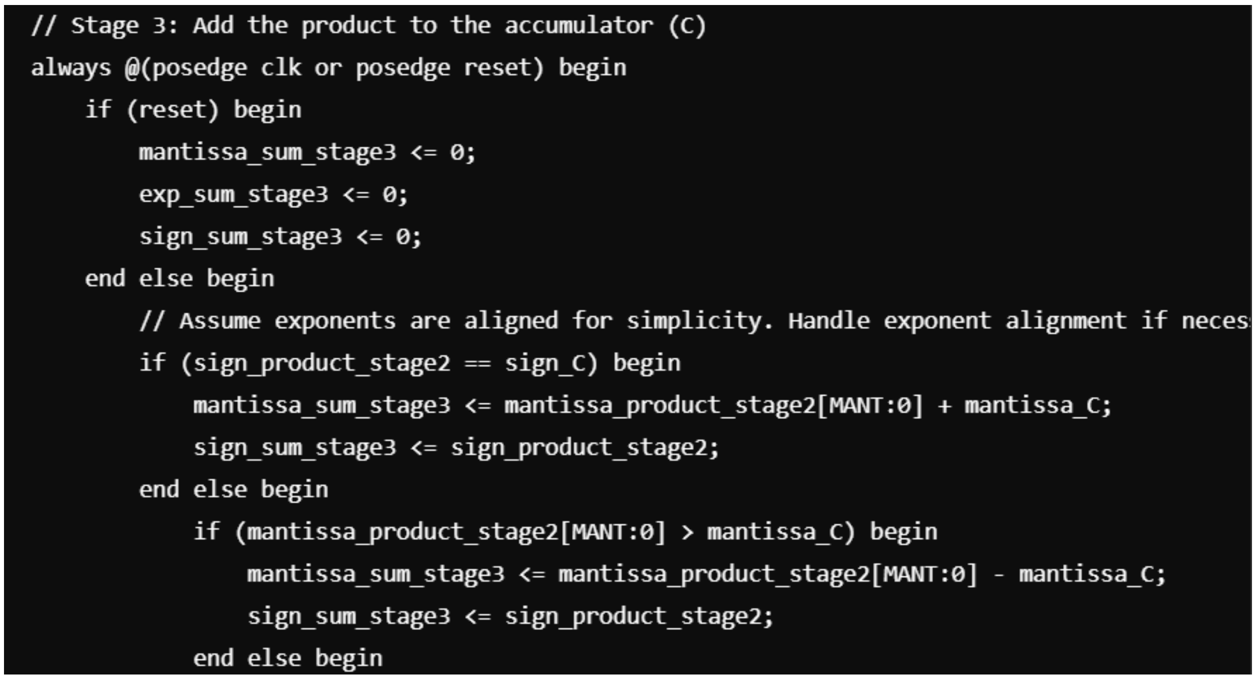
```
// Extract sign, exponent, and mantissa from inputs
assign sign_A = A[N-1];
assign exp_A = A[N-2:N-EXP-1];
assign mantissa_A = {1'b1, A[MANT-1:0]};  // Implicit 1 for mantissa normalization

assign sign_B = B[N-1];
assign exp_B = B[N-2:N-EXP-1];
assign mantissa_B = {1'b1, B[MANT-1:0]};  // Implicit 1 for mantissa normalization

assign sign_C = C[N-1];
assign exp_C = C[N-2:N-EXP-1];
assign mantissa_C = {1'b1, C[MANT-1:0]};  // Implicit 1 for mantissa normalization

// Pipeline Registers
reg [MANT*2+1:0] mantissa_product_stage1, mantissa_product_stage2;
reg [EXP-1:0] exp_product_stage1, exp_product_stage2;
reg sign_product_stage1, sign_product_stage2;
reg [MANT:0] mantissa_sum_stage3;
reg [EXP-1:0] exp_sum_stage3;
reg sign_sum_stage3;
```



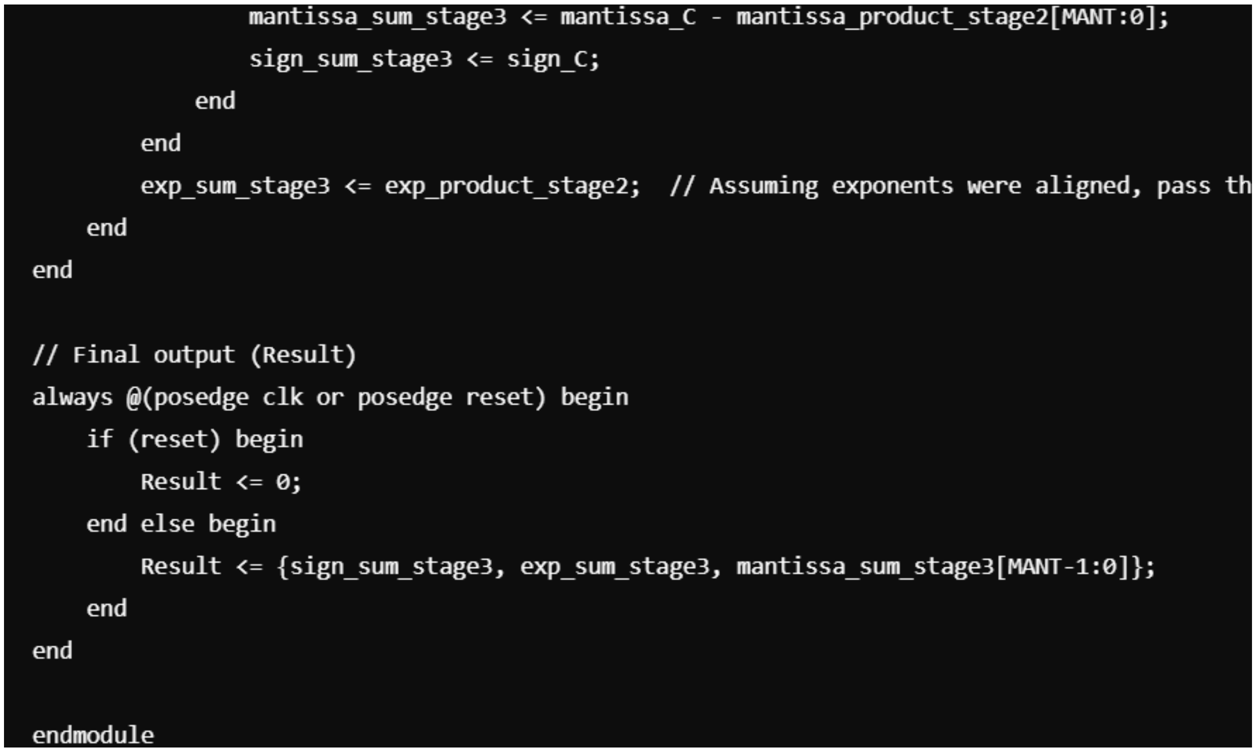
```
// Stage 1: Multiply mantissas and determine sign and exponent for the product
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_product_stage1 <= 0;
        exp_product_stage1 <= 0;
        sign_product_stage1 <= 0;
    end else begin
```



```
        mantissa_product_stage1 <= mantissa_A * mantissa_B;
        exp_product_stage1 <= exp_A + exp_B - ((1 << (EXP-1)) - 1);  // Bias adjustment
        sign_product_stage1 <= sign_A ^ sign_B;
    end
end

// Stage 2: Normalize the product
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_product_stage2 <= 0;
        exp_product_stage2 <= 0;
        sign_product_stage2 <= 0;
    end else begin
        if (mantissa_product_stage1[MANT*2+1]) begin
            // Normalize by shifting and adjusting exponent
            mantissa_product_stage2 <= mantissa_product_stage1 >> 1;
            exp_product_stage2 <= exp_product_stage1 + 1;
        end else begin
            mantissa_product_stage2 <= mantissa_product_stage1;
            exp_product_stage2 <= exp_product_stage1;
        end
        sign_product_stage2 <= sign_product_stage1;
    end
end
```

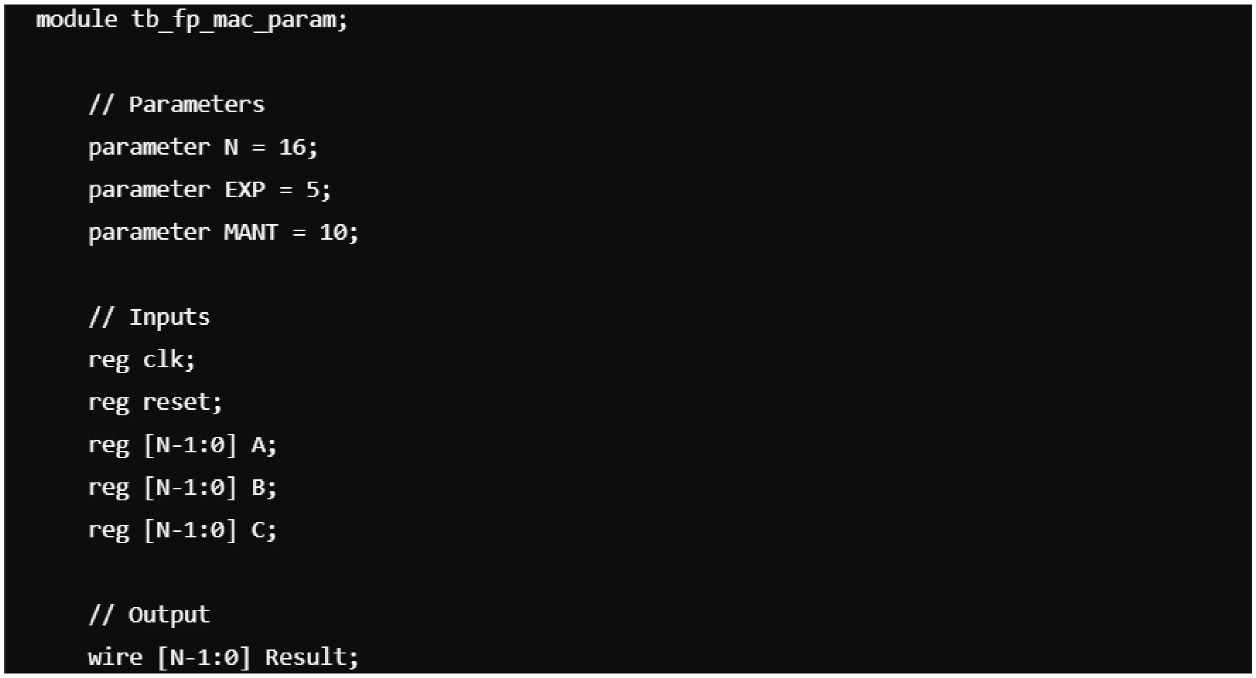


```
// Stage 3: Add the product to the accumulator (C)
always @(posedge clk or posedge reset) begin
    if (reset) begin
        mantissa_sum_stage3 <= 0;
        exp_sum_stage3 <= 0;
        sign_sum_stage3 <= 0;
    end else begin
        // Assume exponents are aligned for simplicity. Handle exponent alignment if neces
        if (sign_product_stage2 == sign_C) begin
            mantissa_sum_stage3 <= mantissa_product_stage2[MANT:0] + mantissa_C;
            sign_sum_stage3 <= sign_product_stage2;
        end else begin
            if (mantissa_product_stage2[MANT:0] > mantissa_C) begin
                mantissa_sum_stage3 <= mantissa_product_stage2[MANT:0] - mantissa_C;
                sign_sum_stage3 <= sign_product_stage2;
            end else begin
```



```
                mantissa_sum_stage3 <= mantissa_C - mantissa_product_stage2[MANT:0];
                sign_sum_stage3 <= sign_C;
            end
        end
        exp_sum_stage3 <= exp_product_stage2;  // Assuming exponents were aligned, pass th
    end
end

// Final output (Result)
always @(posedge clk or posedge reset) begin
    if (reset) begin
        Result <= 0;
    end else begin
        Result <= {sign_sum_stage3, exp_sum_stage3, mantissa_sum_stage3[MANT-1:0]};
    end
end

endmodule
```

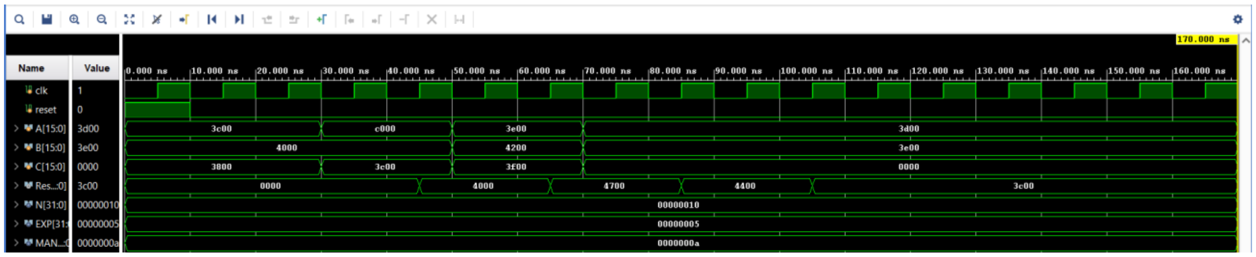
Testbench to verify the parameterizable MAC unit:



```
module tb_fp_mac_param;

    // Parameters
    parameter N = 16;
    parameter EXP = 5;
    parameter MANT = 10;

    // Inputs
    reg clk;
    reg reset;
    reg [N-1:0] A;
    reg [N-1:0] B;
    reg [N-1:0] C;

    // Output
    wire [N-1:0] Result;
    // Instantiate the MAC unit
    fp_mac_param #(N, EXP, MANT) uut (
        .clk(clk),
        .reset(reset),
        .A(A),
        .B(B),
        .C(C),
        .Result(Result)
    );

    // Clock generation
    always #5 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 1;
        A = 16'h3C00;  // 1.0 in half-precision (16-bit)
        B = 16'h4000;  // 2.0 in half-precision (16-bit)
        C = 16'h3800;  // 0.5 in half-precision (16-bit)
```

```
        // Release reset
        #10 reset = 0;

        // Test case 1
        #20;
        A = 16'hC000;  // -2.0 in half-precision
        B = 16'h4000;  // 2.0 in half-precision
        C = 16'h3C00;  // 1.0 in half-precision

        // Test case 2
        #20;
        A = 16'h3E00;  // 1.5 in half-precision
        B = 16'h4200;  // 3.5 in half-precision
        C = 16'h3F00;  // 1.75 in half-precision

        // Test case 3
        #20;
        A = 16'h3D00;  // 0.25 in half-precision
        B = 16'h3E00;  // 1.125 in half-precision
        C = 16'h0000;  // 0 in half-precision
        // End simulation
        #100 $finish;
    end

    // Monitor output
    initial begin
        $monitor("At time %0dns: A = %h, B = %h, C = %h, Result = %h", $time, A, B, C, Res
    end

endmodule
```

Simulation test with Vivado:


170.000 ns
Name
Value
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
130.000 ns
140.000 ns
150.000 ns
160.000 ns
clk 1
reset 0
A[15:0] 3d00 — 3c00, c000, 3e00, 3d00
B[15:0] 3e00 — 4000, 4200, 3e00
C[15:0] 0000 — 3800, 3c00, 3f00, 0000
Res...:0] 3c00 — 0000, 4000, 4700, 4400, 3c00
N[31:0] 00000010 — 00000010
EXP[31: 00000005 — 00000005
MAN...: 0000000a — 0000000a

## V. Testing Result

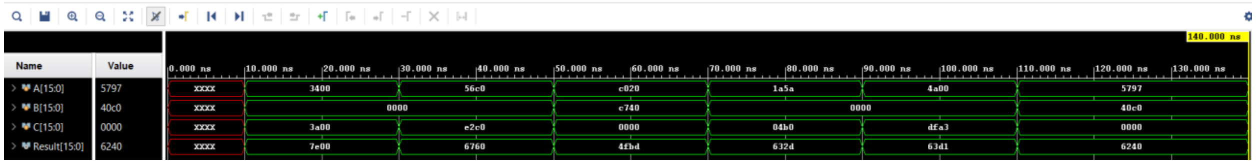
### 1. Test Cases

The following test cases were used for both the unpipelined and pipelined MAC:

1. **0.25 + 1.125** (0011_0100_0000_0000 + 0011_1010_0000_0000)
2. **150 - 250** (0101_0110_1100_0000 - 0110_0010_1100_0000)
3. **-2.5 × -7.25** (1100_0000_0010_0000 × 1100_0111_0100_0000)
4. **0.0001 + 0.00000001** (0001_1010_0101_1010 + 0000_0100_1011_0000)
5. **1024 - 8075** (0100_1010_0000_0000 - 0101_1111_1010_0011)
6. **2014 × 3.75** (0101_0111_1001_0111 × 0100_0000_1100_0000)

### 2. Unpipelined MAC Testing

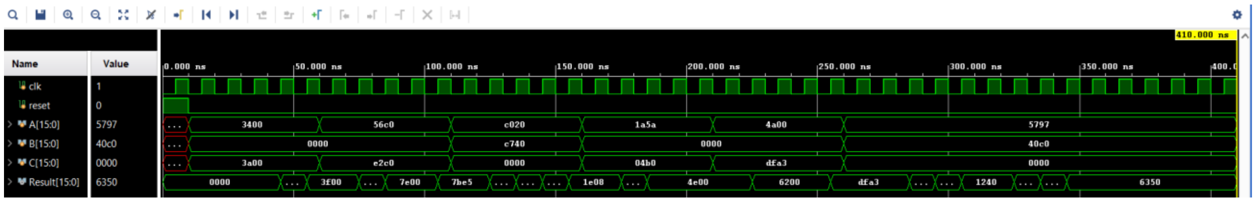
### Simulation on Vivado


140.000 ns
Name
Value
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
50.000 ns
60.000 ns
70.000 ns
80.000 ns
90.000 ns
100.000 ns
110.000 ns
120.000 ns
130.000 ns
A[15:0] 5797 — XXXX, 3400, 56c0, c020, 1a5a, 4a00, 5797
B[15:0] 40c0 — XXXX, 0000, c740, 0000, 40c0
C[15:0] 0000 — XXXX, 3a00, e2c0, 0000, 04b0, dfa3, 0000
Result[15:0] 6240 — XXXX, 7e00, 6760, 4fbd, 632d, 63d1, 6240

For the unpipelined MAC, the results were generated in a single clock cycle for each test case:

| Operation | Expected Result | Observed Result |
|---|---|---|
| 0.25 + 1.125 | 1.375 | Correct |
| 150 - 250 | -100 | Correct |
| -2.5 × -7.25 | 18.125 | Correct |
| 0.0001 + 0.00000001 | 0.00010001 | Correct |
| 1024 - 8075 | -7051 | Correct |
| 2014 × 3.75 | 7552.5 | Correct |

**3. Pipelined MAC Testing**
**Simulation on Vivado**


410.000 ns
Name
Value
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.0
clk 1
reset 0
A[15:0] 5797: ... 3400 56c0 c020 1a5a 4a00 5797
B[15:0] 40c0: ... 0000 c740 0000 40c0
C[15:0] 0000: ... 3a00 e2c0 0000 04b0 dfa3 0000
Result[15:0] 6350: 0000 ... 3f00 ... 7e00 7be5 ... ... ... 1e08 ... 4e00 6200 dfa3 ... ... 1240 ... ... 6350

For the pipelined MAC, the results were delayed by several clock cycles due to the pipeline stages. However, once the pipeline was filled, new results were produced on each clock cycle:

| Operation | Expected Result | Observed Result (After Pipeline Latency) |
|---|---|---|
| 0.25 + 1.125 | 1.375 | Correct |
| 150 - 250 | -100 | Correct |
| -2.5 × -7.25 | 18.125 | Correct |
| 0.0001 + 0.00000001 | 0.00010001 | Correct |
| 1024 - 8075 | -7051 | Correct |

| Operation | Expected Result | Observed Result (After Pipeline Latency) |
|---|---|---|
| 2014 × 3.75 | 7552.5 | Correct |

## V. Comparison Between Unpipelined and Pipelined MAC

During the design and testing phases, key differences between the unpipelined and pipelined MAC units emerged:

- **Latency and Throughput:**
  - Unpipelined MAC: Each operation (multiplication and addition) is processed sequentially, leading to longer delays and lower throughput. Only one result is produced at a time, making it less efficient for handling multiple operations.
  - Pipelined MAC: The pipelined MAC splits the computation into stages, allowing multiple operations to be processed simultaneously. After the pipeline is filled, a new result is produced every clock cycle, greatly increasing throughput and reducing delays.
- **Pipeline Stages:**
  - Unpipelined MAC: Results are generated one at a time, as each operation must fully complete before the next can start.
  - Pipelined MAC: Operations are split across stages, enabling new operations to start while previous ones are still processing. This significantly reduces processing time and boosts performance.
- **Result Latency:**
  - In the unpipelined MAC, results are delayed because operations are processed sequentially.
  - In the pipelined MAC, results appear consistently after the pipeline fills, improving efficiency.

The pipelined MAC is far superior for high-performance tasks, offering faster processing and higher throughput by overlapping operations. The unpipelined MAC, while simpler, is less efficient for applications requiring rapid, repeated computations.

## VII. LLM Assistance and Limitations

Large Language Models (LLMs) like **ChatGPT** were used extensively during this project. LLMs provided significant support, particularly in the following areas:

- **Syntax Help**: LLMs helped clarify Verilog syntax and provided code examples, speeding up the coding process.
- **Debugging**: LLMs offered suggestions for fixing issues with normalization, exponent alignment, and general logic errors.
- **Conceptual Understanding**: When confusion arose regarding floating-point arithmetic, exponent biasing, or pipeline stages, LLMs provided clear explanations and guidance.

**Limitations of LLMs:**

- **Contextual Understanding**: LLMs sometimes provided generic or simplified answers that didn't fully align with the specific requirements of the project. This necessitated manual adjustments to the suggestions provided.
- **Complexity**: For more complex design decisions, such as optimizing the pipelining stages, LLMs were less useful. Most of the optimizations had to be done manually through trial and error.
- **Handling Edge Cases**: While LLMs offered useful starting points, they struggled to provide robust solutions for handling edge cases like overflow, underflow, and denormalized numbers in floating-point arithmetic.

## VIII. Conclusion

In conclusion, the project successfully implemented both unpipelined and pipelined MAC units for half-precision floating-point arithmetic. The unpipelined version achieved correct results in a single clock cycle, while the pipelined version significantly improved throughput by processing multiple operations concurrently across six stages. The assistance of LLMs was invaluable in speeding up the design process, although manual intervention was required for complex tasks and optimizations.