# CS6230_MAC_Unit_project REPORT

Done by: DANIEL MARK ISAAC

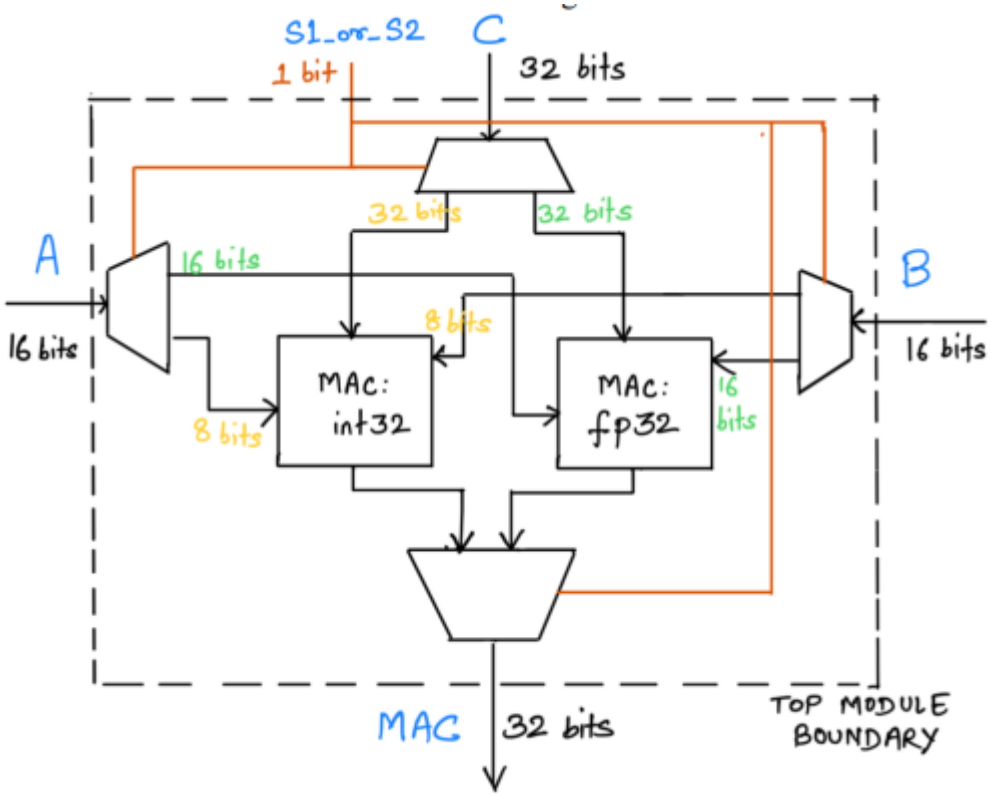
ROLL NO: NS24Z353

**DESCRIPTION:**

This report summarises the MAC project.

**GIVEN SPECIFICATION:**

The following diagram illustrates the specification:




The following is the design requirements:

Implement the MAC module using Bluespec System Verilog (BSV), as mentioned in the beginning of this specification, **without using the + or * operators**. As a part of this assignment, you would have to implement the following design variants.

a) Implement the MAC module as an **Unpipelined** design.
b) Modify the implementation into a **Pipelined** design.

**APPROACH TAKEN:**

The following list elaborates the approach taken to tackle this project:

1) Understand integer multiplication and addition in binary

2) Written MAC_int bsv code using “*” and “+” operators

3) Verified the MAC_int bsv code using cocotb testbench

4) Replaced “*” and “+” with ripple carry adder and multiplier module

5) Verified the MAC_int bsv code using cocotb testbench

6) Understood bfloat16 multiplication worked out by hand

7) Figured out the rounding strategy expected by the given testcases using manual calculations

8) Used online float32 subtractor to do MAC – C to get A*B output (given testcases did not have A*B values)

9) Created a python reference model to replicate the bfloat16 multiplication

10) Tested the reference model with given testcases and did bugfixes till all cases passed

11) Expanded the given testcases by flipping the sign bits to get negative inputs

12) Tested the reference model against the expanded testcases

13) Created bfloat multiplication code in bsv with acquired understanding from creating reference model (using “*” and “+” operators)

14) Tested the bsv code with the given testcases, did bug fixes till all cases passed

15) Replaced “*” and “+” operators with ripple carry adders and multipliers written for int MAC.

16) Verified the above bsv code with given testcases

17) Created reference model in python for fp32 addition

18) Realised that there is a need for correct values when inputs are negative (given testcases are all positive)

19) Webscrapped a online calculator by giving negative version of computed A*B values and negative version of C and obtained expected values for negative inputs in float addition.

20) Updated the float add reference model such that it passes for the expanded testcases.

21) Created bsv code for float addition using understanding acquired via writing reference model. (Using “*” and “+” operators)

22) Verified the bsv code against the expanded testcases (lot of bug fixing involved)

23) Replaced “*” and “+” operators with ripple carry adders and multipliers.

24) Verified the bsv code against the expanded testcases

25) Merged Int MAC and float MAC into single bsv module.

26) Ran all given tests and expanded tests and all passed

27) Handled corner cases when zero is given as input and returned as output in float MAC.

28) Included corner cases along with coverage in testbench

29) Updated testbench to drive random inputs to RTL.

30) Restricted the random inputs, so that “Nan”s are filtered before giving to RTL. All random testcases passed**. (By this step, unpipelined design is complete)**

31) Included pipeline FIFOs in Top MAC bsv file and verified that all testcases passed.

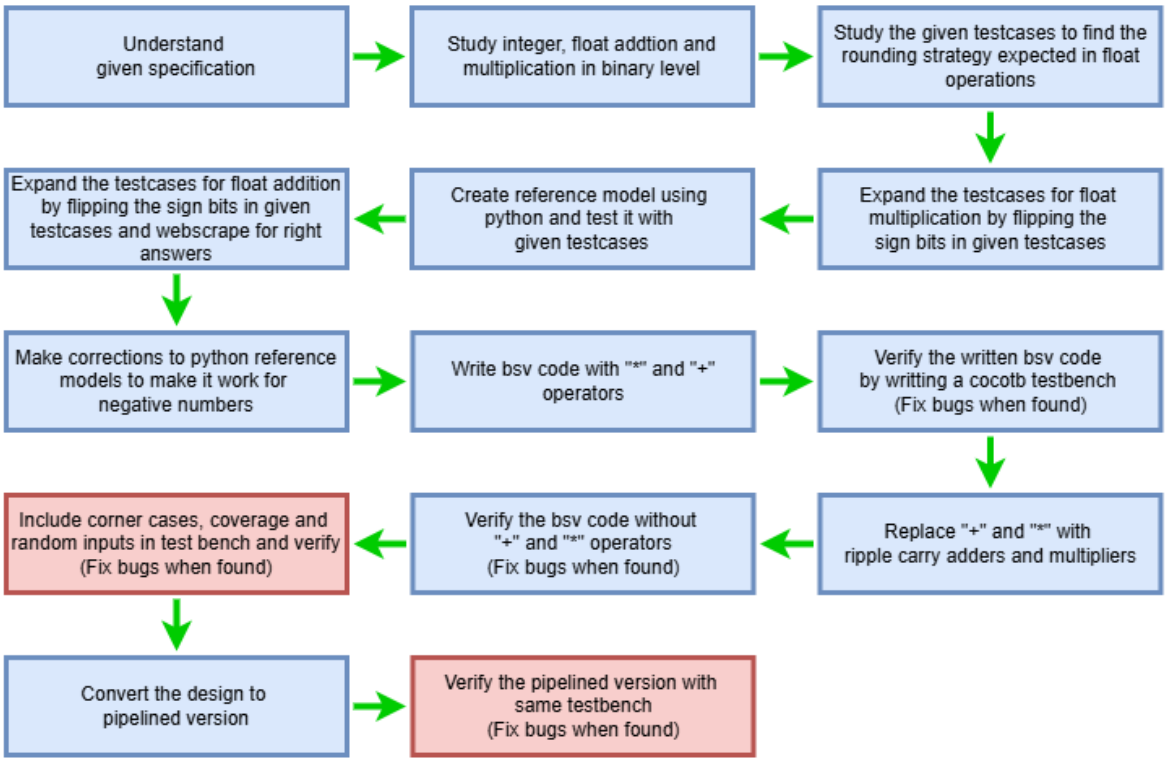
32) Included pipeline FIFOs in MAC Int bsv file and verified that all testcases passed.

33) Included pipeline FIFOs in MAC float bsv file and verified that all testcases passed.

34) Included pipeline FIFOs in MAC float mul bsv file and verified that all testcases passed.

35) Included pipeline FIFOs in MAC float add bsv file and verified that all testcases passed**. (By this step, pipelined design is complete)**

The above steps are summarised in the following flow chart:



The blue boxes in the above flowchart shows the intermediate steps taken. The red boxes indicate the phases of project, where stable, verified and working bsv codes are obtained. The first red box encountered in the above flow chart, corresponds to working unpipelined design, while the last red box corresponds to working pipelined design.

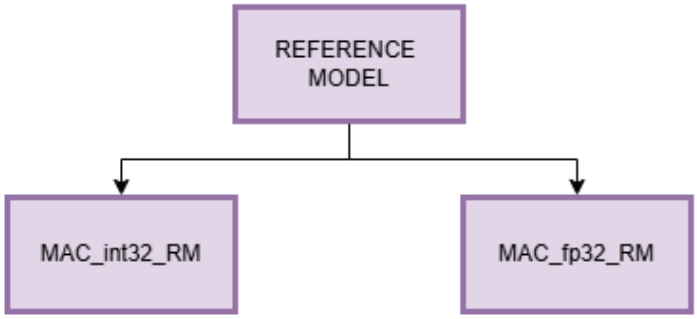
## REFERENCE MODEL:

The reference models are developed based on analysing the given testcases. Both MAC int and MAC float reference models are written in python**.**

**The reference models perform all calculations using string manipulations and does not use any in built python data types (for varying bit widths).**

The above decision is made because of the fact, python does not know it is an N bit integer other than 32-bit integer. This caused lot of problems when dealing with negative numbers and hence the decision was made.

**These reference models were successfully verified with the given testcases.**




## INT_MAC RM:

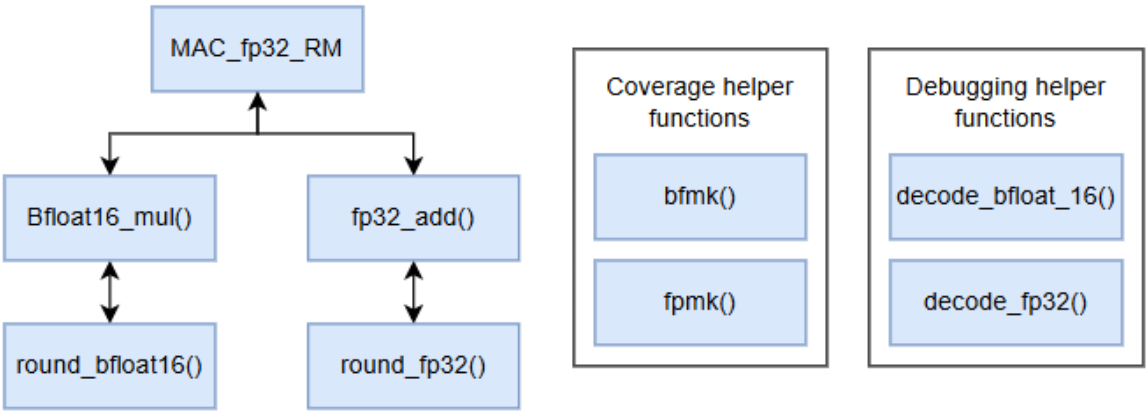
The following image shows the reference model written for Int MAC in python:

```
@MAC_INT_coverage
def MAC_int32_RM(A,B,C):
        temp = A*B+C
        if(temp & 0x80000000):
                return (((temp & 0xFFFFFFFF) ^ 0xFFFFFFFF) + 1) *(-1)
        else:
                return (A*B+C) & 0xFFFFFFFF
```

The if-else logic deals with negative numbers. Basically, if it sees that the sign bit is 1, it takes twos compliment, puts negative sign at the front and returns the value. If sign bit is 0, it masks the last 32 bits (To ignore overflow) and returns the value

**FLOAT_MAC RM:**

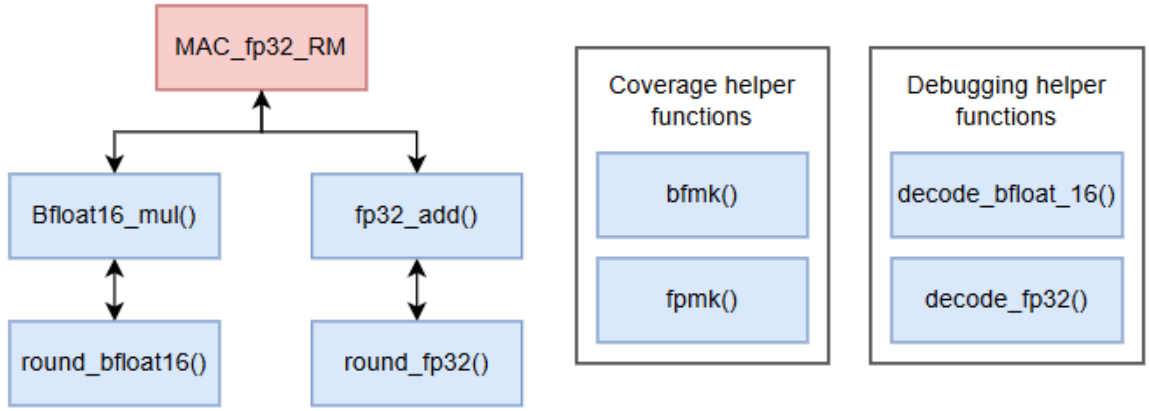
The float mac reference model is made up of several python functions. The following image illustrates how different functions are being called.




The above image will be used in the upcoming sections to explain the reference model. The relevant box will be coloured red in each section.

**MAC_fp32_RM:**

This is the top-level python function which orchestrates the other python functions to compute A*B+C in float point operations.



```python
@MAC_FLOAT_coverage
def MAC_fp32_RM(A,B,C):
# Float multiplication
        if(A[1:] == "0"*15 or B[1:] == "0"*15):
                AB = "0"*16
        else:
                AB = bfloat16_mul(A,B)
                if(AB == "EXCEPTION"):
                    return "EXCEPTION"

# Float addition
        if(C[1:] == "0"*31):
                C = "0"*32
        if(AB[1:].ljust(31,"0") == C[1:] and AB[0] != C[0]):
                return "0"*32
        if(AB == "0"*16):
                return C
        elif(C == "0"*32):
                return AB.ljust(32,"0")
        else:
                return fp32_add(AB,C)
```

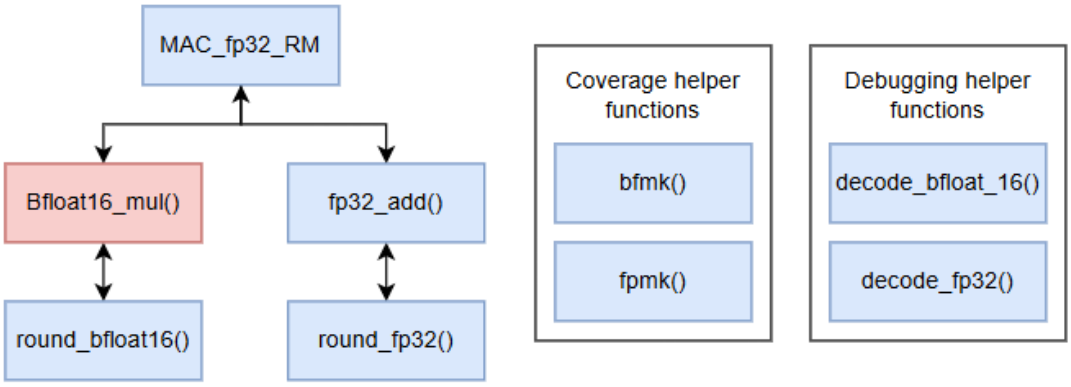
It calls bfloat16_mul() and fp32_add() functions to do A*B+C on floating numbers. The if-else Under “# Float multiplication” comment, checks if one of the inputs is zero. If yes, it sets the variable “AB” as 16-bit zero. Else it computes the floating point multiplication by calling bfloat16_mul() and stores it in variable “AB”.

bfloat16_mul() will return the string “EXCEPTION” if at any point in the computation, it identifies a NaN (Not a Number).

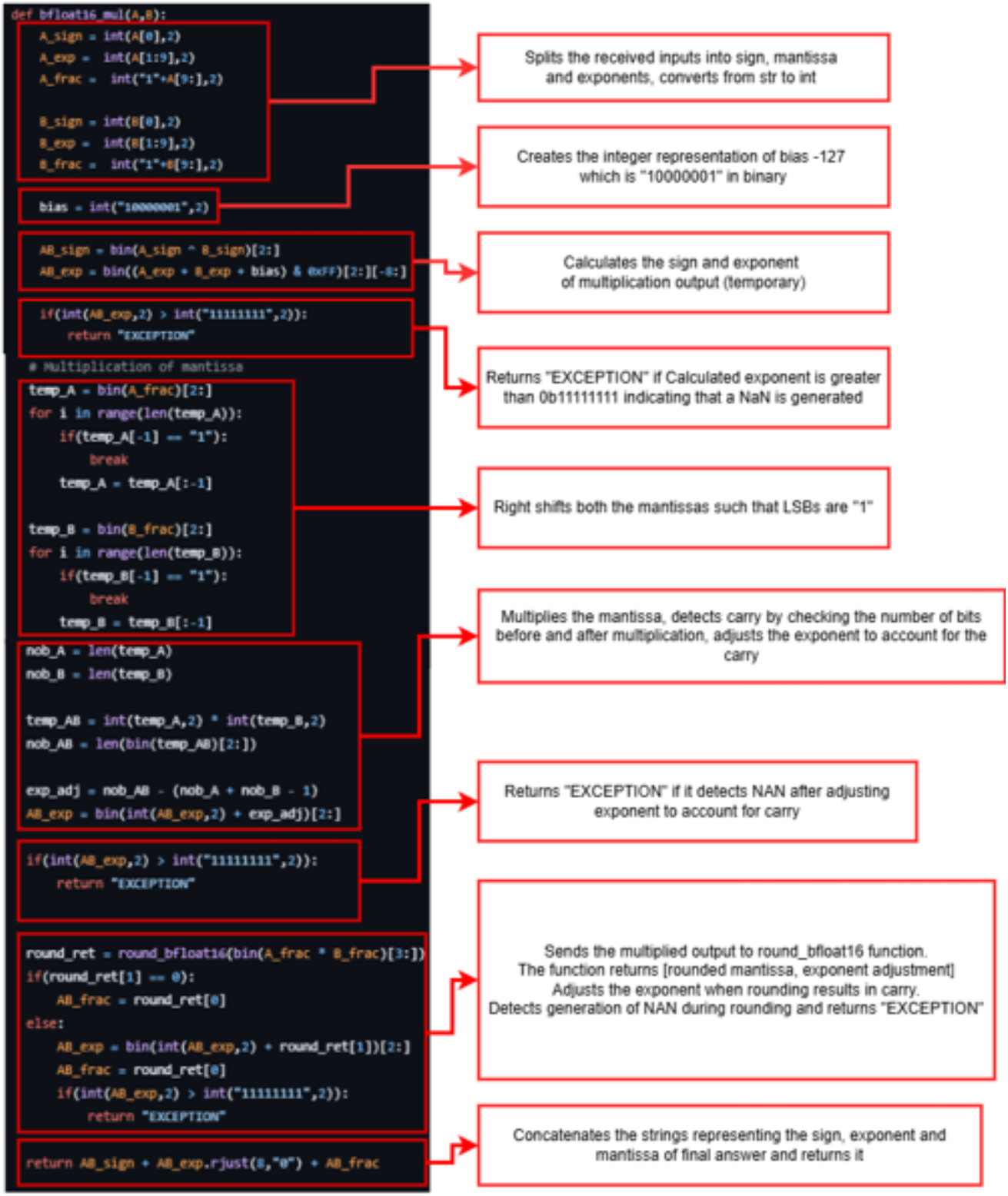
The first if condition under “# Float addition”, checks if the input C is a negative zero, and converts it to positive zero. The second if condition checks if AB = -C, if yes, it returns zero. The next if condition checks if AB is zero, if yes, it returns C. The next elif condition checks if C is zero, if yes, it returns AB. The last else condition call fp32_add() function to do float addition.

**bfloat16_mul function:**

This function performs float multiplication.

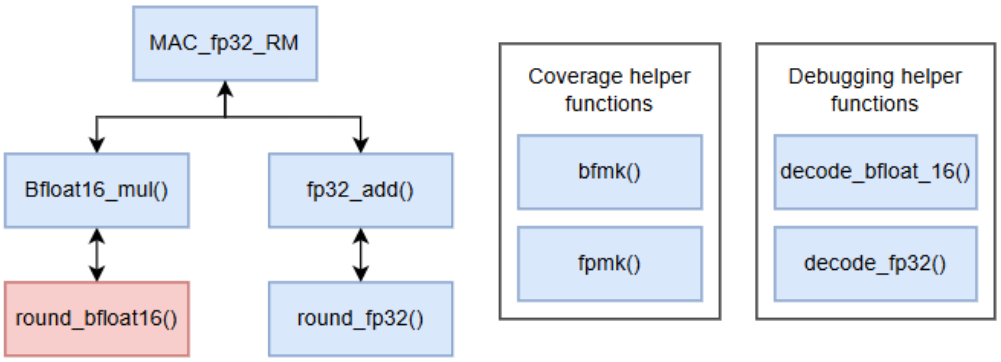


The following image explains the bfloat16_mul() function in detail.

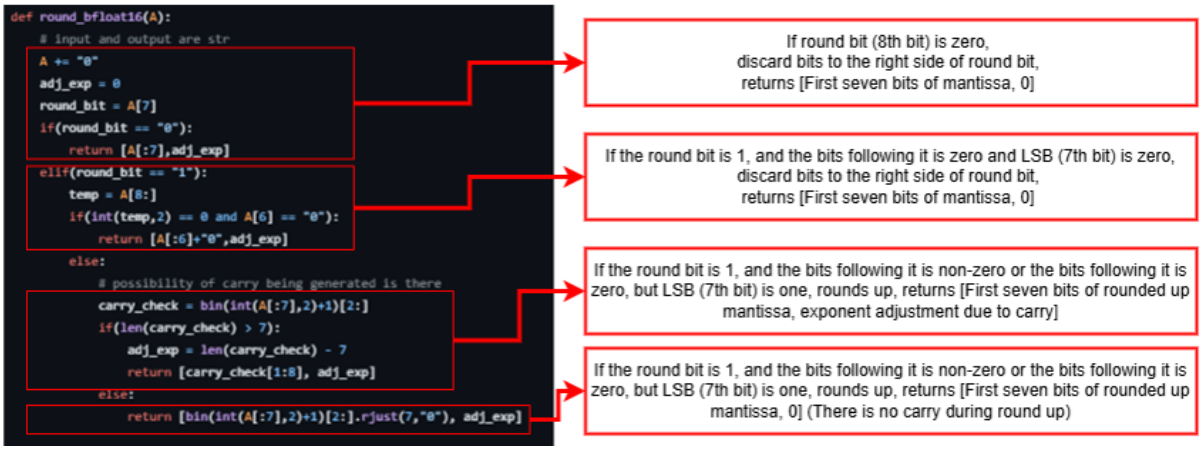


**round_bfloat16 function:**

This function rounds the output of floating-point multiplication.

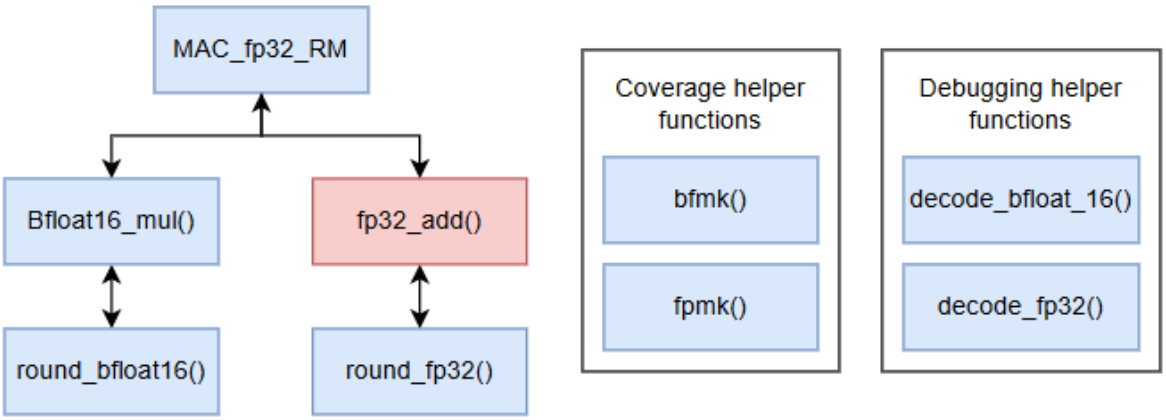



The following image explains round_bfloat16() function in detail.

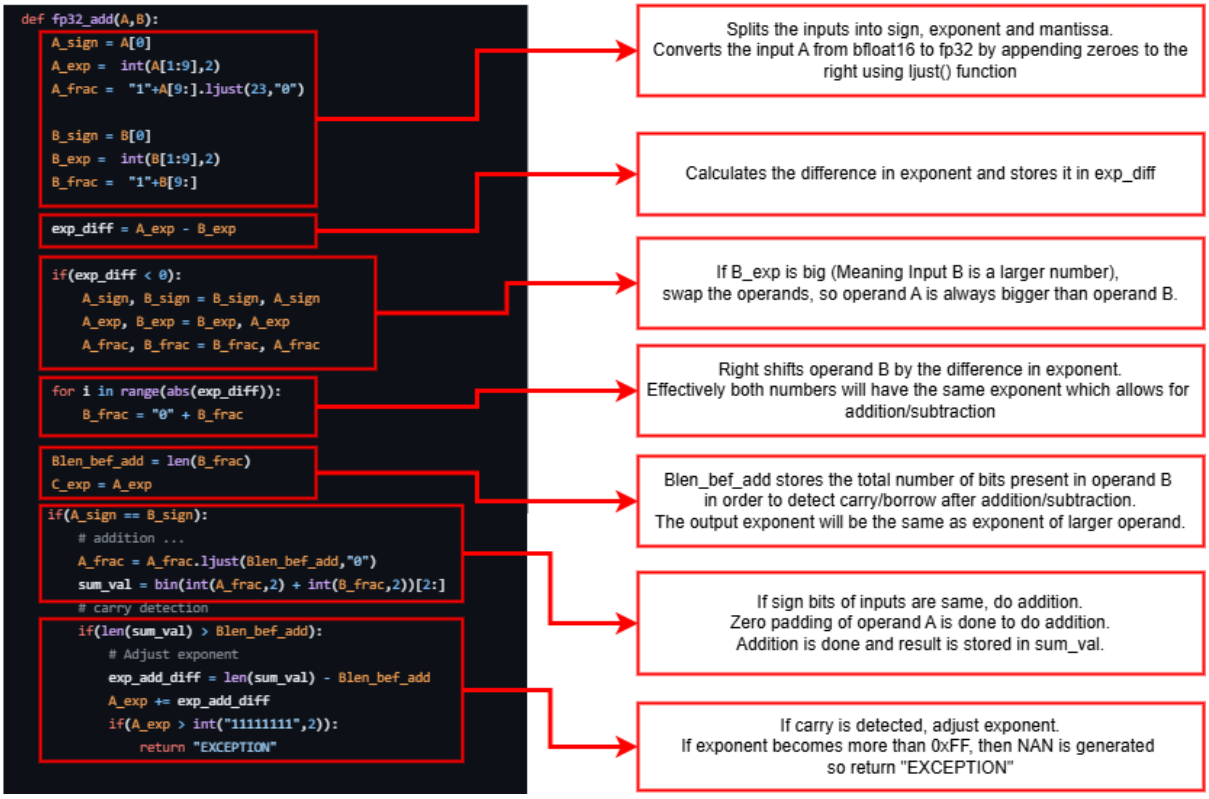


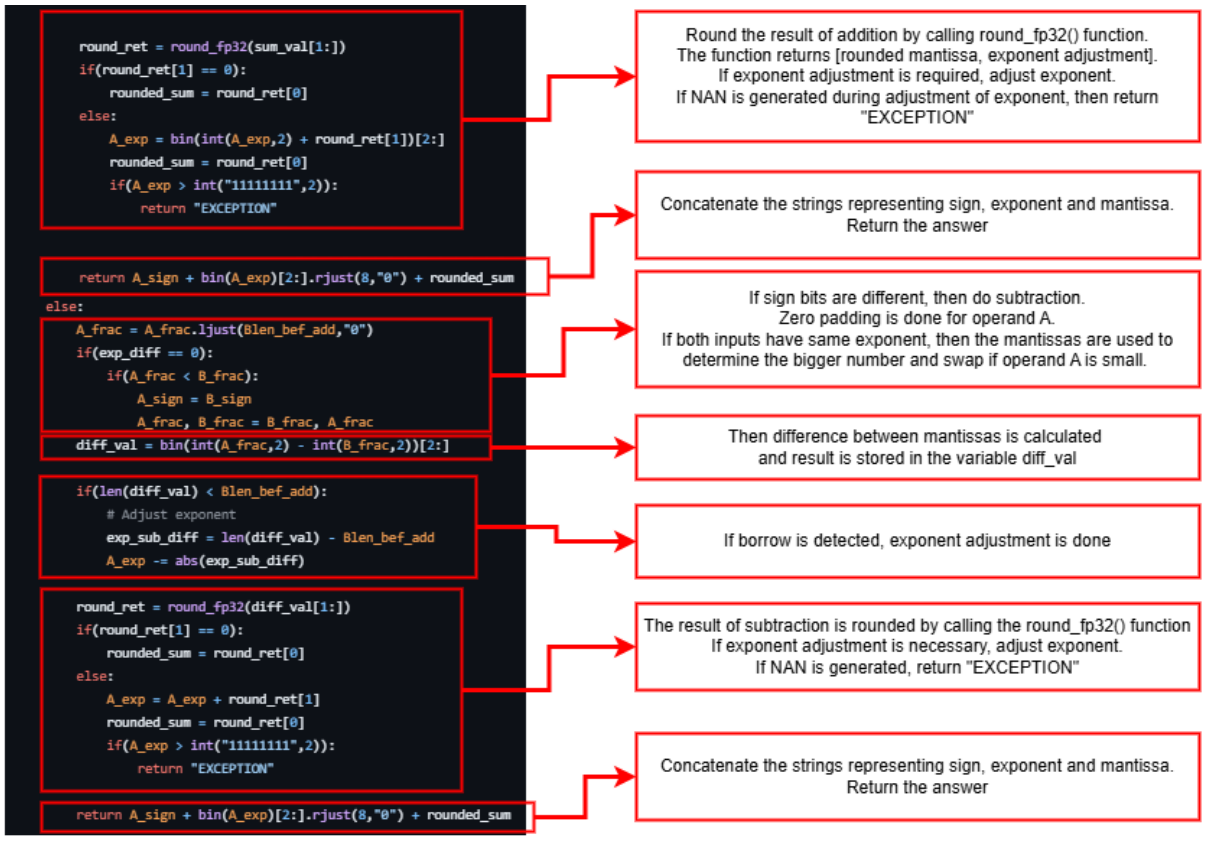

**fp32_add function:**

This function performs the floating-point addition of two numbers.



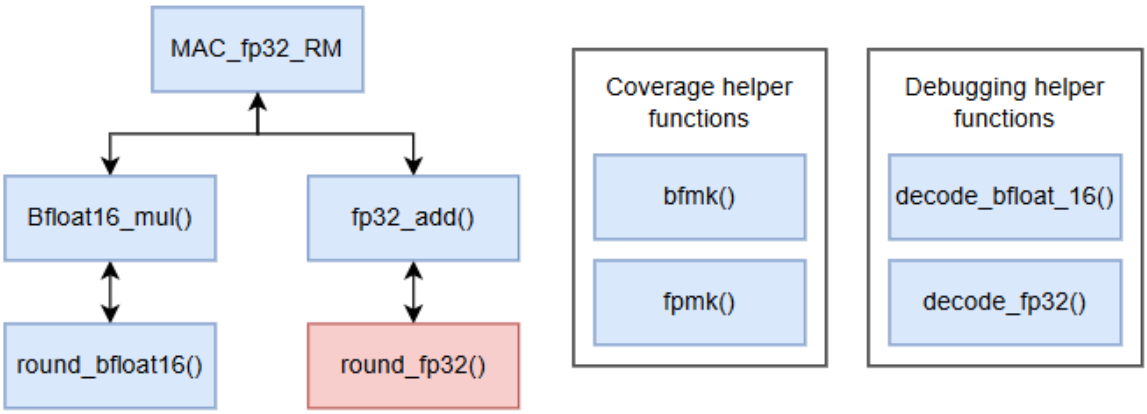
The following image explains fp32_add() function in detail.



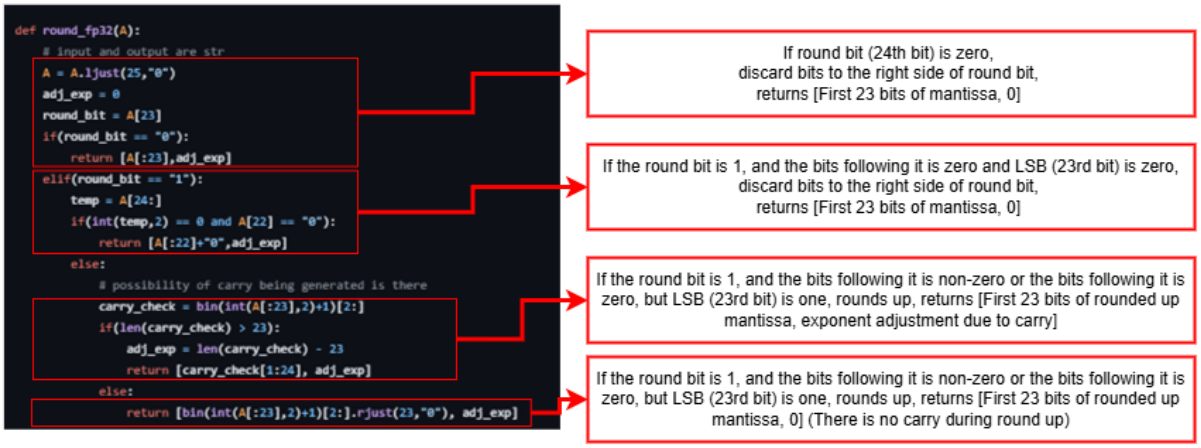



**round_fp32 function:**

This function rounds the output of floating-point addition.

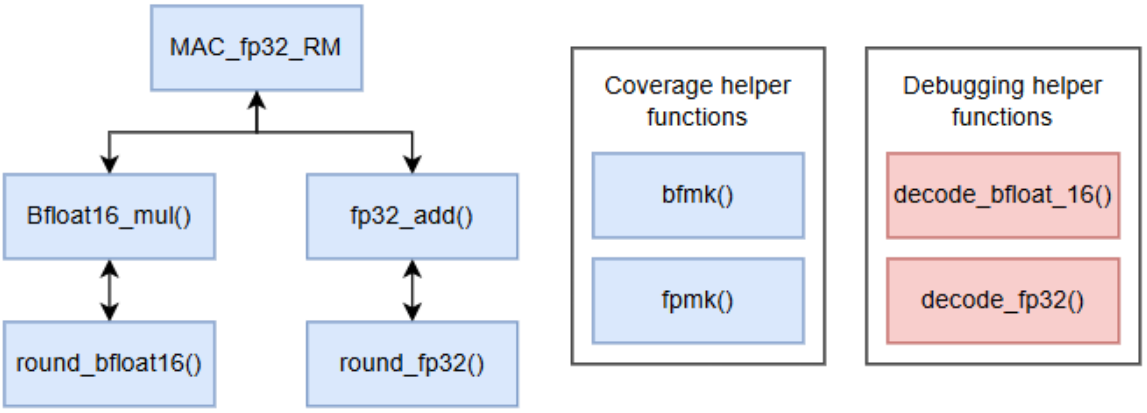



The following image explains round_fp32() function in detail.



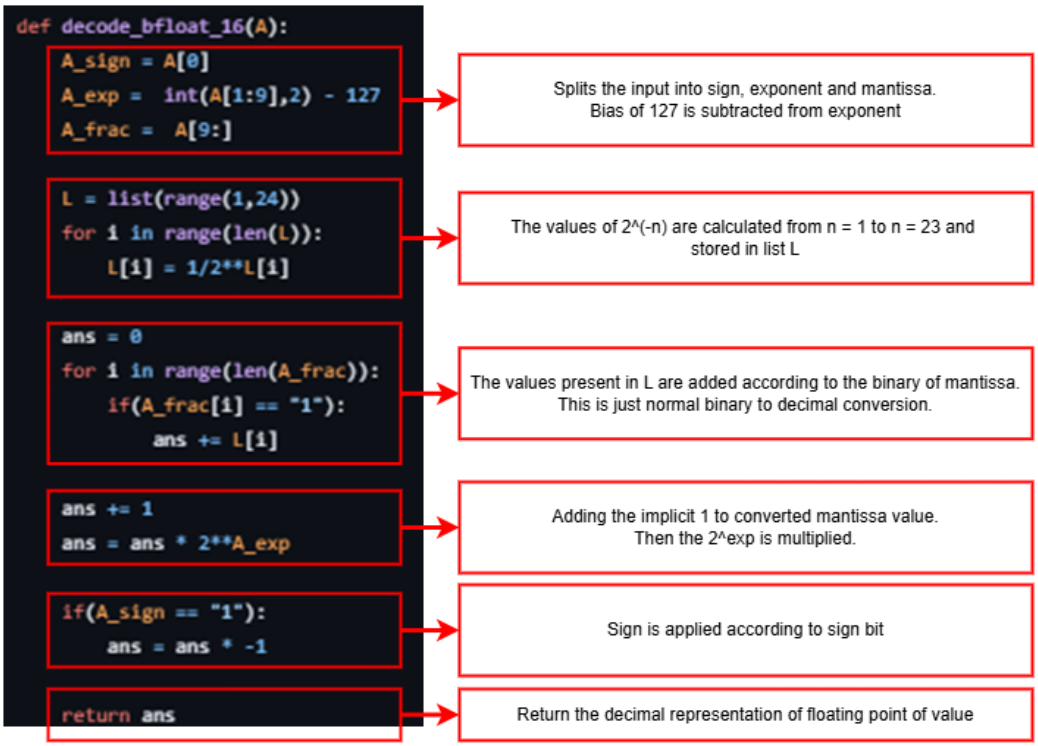

**Debugging helper functions:**

Two functions, decode_bfloat_16() and decode_fp32() are written which will convert the binary representation of floating-point numbers into decimal numbers. These functions were useful during debugging phase.



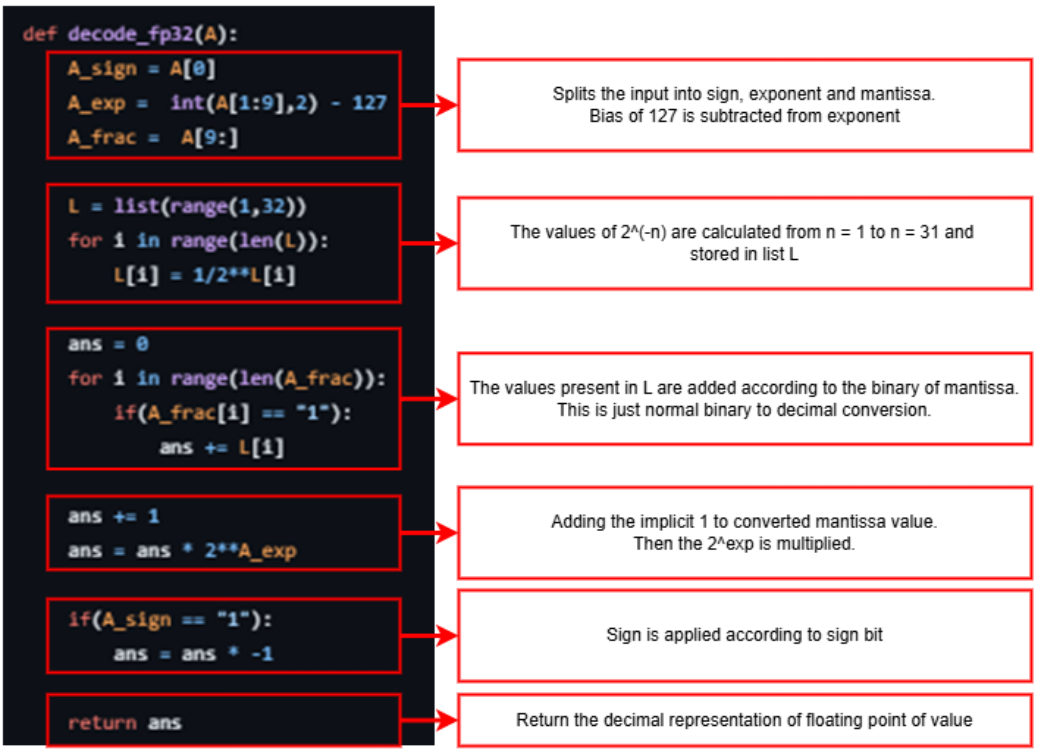
**decode_bfloat_16 function: (For checking purposes)**

The following image explains decode_bfloat_16() function in detail.



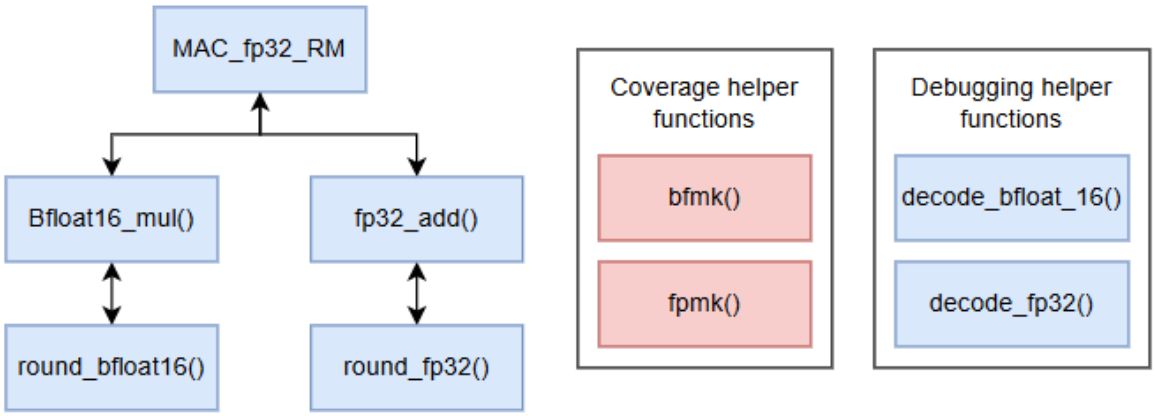

**decode_fp32 function: (For checking purposes)**

The following image explains decode_fp32() function in detail.



**Coverage helper functions:**

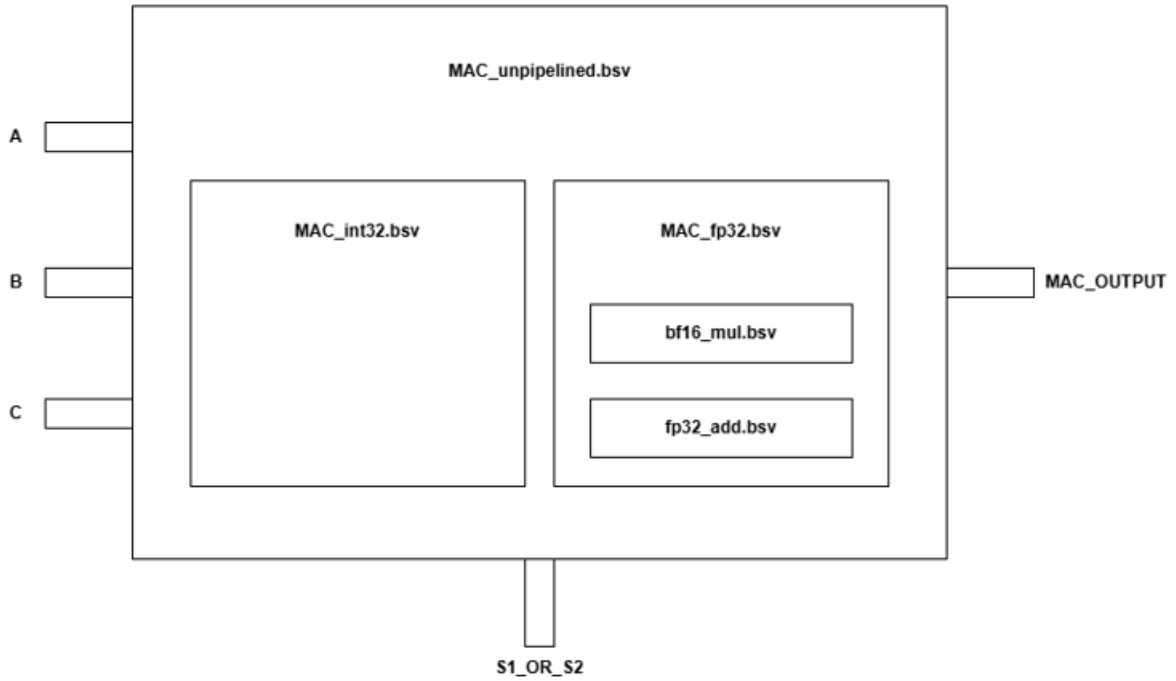
Two functions, bfmk() and fpmk() are written, which will obtain the integer representations of sign, exponent and mantissa and return the string representation of floating-point binary. These functions are useful when defining coverage bins.




```
def bfmk(S,E,M):
    return bin(S)[2:].ljust(1,"0")+bin(E)[2:].ljust(8,"0")+bin(M)[2:].ljust(7,"0")

def fpmk(S,E,M):
    return bin(S)[2:].ljust(1,"0")+bin(E)[2:].ljust(8,"0")+bin(M)[2:].ljust(23,"0")
```

# UNPIPLINED DESIGN

**DESIGN ARCHITECTURE:**




The above image illustrates the module hierarchy. The following sections will explain each module in detail.

**MAC_unpipelined.bsv:**

Gets the inputs A (16 bits), B (16 bits), C (32 bits) and S1_or_S2 (1 bit) and gives them to the instantiated modules of MAC_int32.bsv and MAC_fp32.bsv according to value of S1_or_S2.

If S1_or_S2 is 0 => Give inputs and get output from MAC_int32 module

If S1_or_S2 is 1 => Give inputs and get output from MAC_fp32 module

**MAC_int32.bsv:**

Takes the following as input:

1) Lower 8 bits of input A
2) Lower 8 bits of input B

3) 32-bit input C

Gives the following as output:

1) 32-bit MAC output

The Integer MAC is implemented as a single block.

**The following image shows the logic used for multiplication:**

```
rule rl_multiply(got_A && got_B && got_C && count != 5'd0 && reset_completed == True);
if(rg_B[0] == 1)
begin
        if(count == 5'd1)
        begin
                partial_store <= rca(partial_store , signExtend(twos_compliment(rg_A)));
        end
        else
        begin
                partial_store <= rca(partial_store , signExtend(rg_A));
        end
end
rg_A <= rg_A << 1;
rg_B <= rg_B >> 1;
count <= count - 1;
endrule
```

The register “partial store” accumulates the partial products obtained at each cycle. The “count” register is initialised to 9. The multiplier is stored in rg_A and multiplicand is stored in rg_B.

At each cycle, the LSB of multiplicand is checked and if it is “1”, signExtended rg_A is added to partial_store and stored in partial_store itself. If LSB is “0”, partial_store remains unchanged. The additions are done using ripple carry adders.

Regardless of LSB of multiplicand, after the updation of partial_store, rg_A is shifted left and rg_B is shifted right and count is decremented.

When the count reaches “1” (Last cycle), the partial product is subtracted from the partial_store. Subtraction is done by using twos compliment procedure.

**The following image shows the logic used for addition:**

```
function Bit#(16) rca(Bit#(16) a, Bit#(16) b);
Bit#(16) outp = 0;
Bit#(1) carry = 0;
outp[0] = a[0] ^ b[0];
carry = a[0] & b[0];
for(Integer i = 1; i < 16; i = i + 1)
begin
        outp[i] = a[i] ^ b[i] ^ carry;
        carry = (a[i] & b[i]) | (a[i] ^ b[i]) & carry;
end

return outp;
endfunction:rca
```

The 16-bit ripple carry adder is used to replace the “+” operator used in multiplication.

The addition is done using the Boolean expression of sum and carry of half and full adder. The overflowing carry bit is ignored.

Different functions with the same logic with differing bit widths are created and used throughout the code to eliminate the usage of “+” operator to the maximum extent.

**The following image shows the logic used for twos compliment:**

```
function Bit#(16) twos_compliment(Bit#(16) num);
Bit#(16) mask = 16'hFFFF;
Bit#(16) temp = 16'd0;
temp = num ^ mask;
temp = rca_16bit(temp,1);
return temp;
endfunction:twos_compliment
```

First the input number is XOR’ed with 0xFFFF. This will invert all the bits. Then 16-bit ripple carry adder is used to add 1 to the XOR’ed output resulting in 2’s compliment output.

The above functions are coordinated by rules to provide input and get output from each other to give final Int MAC output.

**MAC_fp32.bsv:**

This is a submodule which further instantiates two other submodules: bf16_mul and fp32_add within it.

Takes the following as input:

1) 16-bit input A
2) 16-bit input B
3) 32-bit input C

Gives the following as output:

1) 32-bit MAC output

This code is mainly dominated by four rules which are shown below:

```
rule do_mul(got_A == True && got_B == True && got_C == True && mul_initiated == False);
    mul_initiated <= True;
    fmul.get_A(rg_a);
    fmul.get_B(rg_b);
endrule

rule get_mulres(mul_initiated == True);
    mul_completed <= True;
    rg_ab <= pack(fmul.out_AB());
endrule

rule do_add(got_A == True && got_B == True && got_C == True && mul_completed == True && add_initiated == False);
    add_initiated <= True;
    fadd.get_A(rg_ab);
    fadd.get_B(rg_c);
endrule

rule get_addres(add_initiated == True);
    fmac_completed <= True;
    mac_output <= fadd.out_AaddB();
endrule
```

do_mul rule is the first rule to fire. It will set mul_initiated and provides inputs to methods present within bf16_mul.bsv.

get_mulres rule will fire when both mul_initiated is true and output of multiplication is ready(Implicit firing condition). When fired, this will store multiplication output to rg_AB and set mul_completed as True.

do_add rule will fire after multiplication is done. It will set add_initiated as true and provide inputs to the methods present within fp32_add.bsv.

get_addres rule will fire when both add_initiated is true and output of addition is ready(Implicit firing condition). When fired, this will store addition output to mac_output and set fmac_completed as True.

fmac_completed triggers the value method, which will return the value to higher level module.

**bf16_mul.bsv:**

This module computes the floating multiplication of two Bfloat 16 numbers.

Takes the following as input:

1) 16-bit input A
2) 16-bit input B

Gives the following as output:

1) Output of Bfnum type

**Bfnum type:**

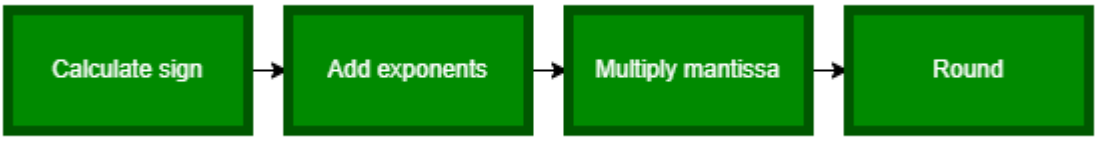
Bfnum is a structure with 1-bit sign, 8 bits exponent and 7 bits mantissa as its members as shown below:

```
typedef struct {
    Bit#(1) sign;
    Bit#(8) exponent;
    Bit#(7) fraction;
} Bfnum deriving (Bits, Eq);
```

The obtained inputs are populated in Bfnum as shown below:

```
method Action get_A(Bit#(16) a) if (!got_A);
    got_A <= True;
    bf_a <= Bfnum{ sign: a[15], exponent: a[14:7], fraction: a[6:0] };
endmethod

method Action get_B(Bit#(16) b) if (!got_B);
    got_B <= True;
    bf_b <= Bfnum{ sign: b[15], exponent: b[14:7], fraction: b[6:0] };
endmethod
```

After obtaining the inputs, the following flow is followed:



**Calculate sign:**

calculate_sign rule performs the sign calculation of the output bfnum after the inputs are provided. The logic is shown below:

```
rule calculate_sign(got_A == True && got_B == True && sign_calculated == False && handle_zero == False);
    if((bf_a.exponent == '0 && bf_a.fraction == '0) || (bf_b.exponent == '0 && bf_b.fraction == '0)) // To handle if one of the inputs is zero
    begin
            handle_zero <= True;
    end
    else
    begin
            sign_calculated <= True;
            sign_c <= bf_a.sign ^ bf_b.sign;
    end
endrule
```

This rule will perform XOR between the inputs sign bits to get the output sign bit [“else” part in the above code].

This rule will also detect the corner case where one of the inputs is zero [“if” part in the above code]. Upon detecting the corner case, it will set “handle_zero” as true to indicate to the other rules that corner case has occurred. Basically, if one of the inputs is zero, we can say output is zero without multiplying.

**Add exponents:**

Next step is to add up the exponents and subtract the bias of 127.

The above statement can be translated as: Exp_A + Exp_B – 127

When we take 2’s compliment of 127 we get: Exp_A + Exp_B + 0b10000001, which is just two additions in series. This calculation is achieved by the rule calculate_expone and the function add_exponents as shown below:

```
rule calculate_expone(got_A == True && got_B == True && sign_calculated == True && expone_calculated == False && handle_zero == False);
    expone_calculated <= True;
    calculate_mantissa <= True;
    exp_c <= add_exponents(bf_a.exponent , bf_b.exponent);
    temp_A <= zeroExtend({1'b1,bf_a.fraction});
    temp_B <= zeroExtend({1'b1,bf_b.fraction});
endrule
```

The above rule just provides input and gets output from add_exponents function and along with it, it will set few flags and prepares temp_A and temp_B registers for next step in calculation by pre-appending implicit 1 to mantissa.

```
function Bit#(8) add_exponents(Bit#(8) a, Bit#(8) b);
    Bit#(8) outp_inter = 8'b0;
    Bit#(8) outp = 8'b0;
    Bit#(8) bias = 8'b10000001;
    Bit#(1) carry = 1'b0;
    outp_inter[0] = a[0] ^ b[0];
    carry = a[0] & b[0];
    for(Integer i = 1; i < 8; i = i + 1)
    begin
            outp_inter[i] = a[i] ^ b[i] ^ carry;
            carry = (a[i] & b[i]) | (a[i] ^ b[i]) & carry;
    end

    carry = 1'b0;
    outp[0] = outp_inter[0] ^ bias[0];
    carry = outp_inter[0] & bias[0];
    for(Integer i = 1; i < 8; i = i + 1)
    begin
            outp[i] = outp_inter[i] ^ bias[i] ^ carry;
            carry = (outp_inter[i] & bias[i]) | (outp_inter[i] ^ bias[i]) & carry;
    end

    return outp;
endfunction:add_exponents
```

The add_exponents function shown above is just two ripple carry adders in series to do Exp_A + Exp_B + 0b10000001.

**Multiply_mantissa:**

The following image shows the logic used in multiplication of mantissa:

```
rule rl_multiply(got_A == True && got_B == True && count != 5'd0 && sign_calculated == True && expone_calculated == True && calculate_mantissa == True && handle_zero == False);
    if(temp_B[0] == 1)
    begin
            temp_prod <= rca(temp_prod , zeroExtend(temp_A));
    end
    temp_A <= temp_A << 1;
    temp_B <= temp_B >> 1;
    count <= count - 1;
endrule
```

The above rule is just an unsigned 8-bit multiplier. It uses 16-bit ripple carry adder shown below:

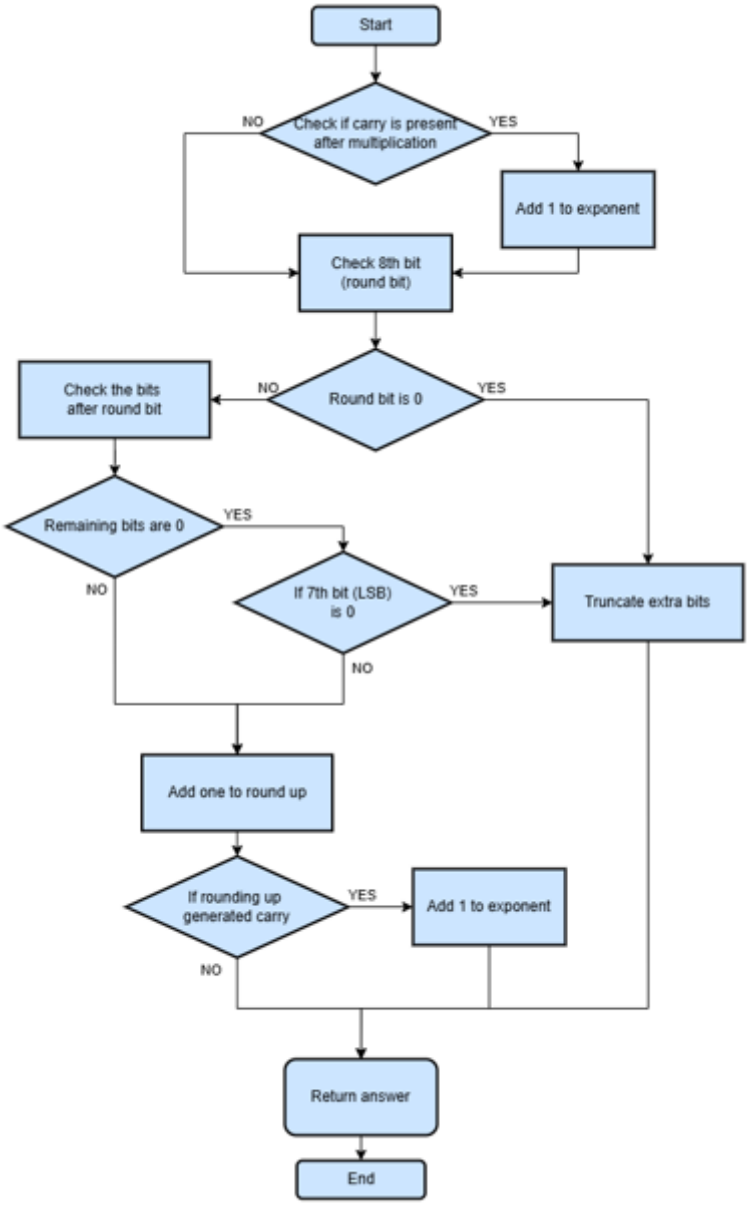
```
function Bit#(16) rca(Bit#(16) a, Bit#(16) b);
    Bit#(16) outp = 0;
    Bit#(1) carry = 0;
    outp[0] = a[0] ^ b[0];
    carry = a[0] & b[0];
    for(Integer i = 1; i < 16; i = i + 1)
    begin
            outp[i] = a[i] ^ b[i] ^ carry;
            carry = (a[i] & b[i]) | (a[i] ^ b[i]) & carry;
    end

    return outp;
endfunction:rca
```

The output of multiplication is stored in temp_prod.

**Round:**

The next step is rounding. The rounding strategy used is “Round to Nearest”

The following flow chart illustrates the rounding algorithm used:



The above flowchart is implemented in bsv as shown below:

```
function Bit#(15) round(Bit#(16) prod_out, Bit#(8) exp);
    Bit#(15) outp = 15'b0;
    Bit#(1) round_bit = 1'b0;
    Bit#(6) rem_nocarry = 6'b0;
    Bit#(7) rem_withcarry = 7'b0;
    Bit#(9) carry_type_a = 9'b0;
    Bit#(9) carry_type_b = 9'b0;

    if(prod_out[15] == 1'd1) // If carry is generated during multiplication
    begin
            exp = add_8bits(exp, 8'b1);
            round_bit = prod_out[7];
            // If round bit is 0 truncate the remaining bits
            if(round_bit == 1'd0)
            begin
                    outp = zeroExtend(prod_out[14:8]);
            end
            // If round bit is 1 do the following
            else
            begin
                  rem_withcarry = prod_out[6:0]; // To check the remaining bits
                  if(rem_withcarry == 7'd0 && prod_out[8] == 1'd0) // If remaining bits are 0 and LSB is also 0
                  begin
                            outp = zeroExtend(prod_out[14:8]); // Truncate the rest
                  end
                  else
                  begin // If remaining bits are non zero
                            carry_type_b = add_9bits(zeroExtend(prod_out[15:8]) , 9'b1); // Add one to round up
                            if(carry_type_b[8] == 1) // See if the above addition results in a carry
                            begin
                                    exp = add_8bits(exp, 8'b1); // Adjust exponent
                                    outp = zeroExtend(carry_type_b[7:1]);
                            end
                            else
                            begin // If there is no carry while rounding up
                                    outp = zeroExtend(carry_type_b[6:0]);
                            end
                  end
            end

    end
    else  // If carry is not generated during multiplication
```

```
    end
    else  // If carry is not generated during multiplication
    begin
            round_bit = prod_out[6];
            // If round bit is 0 truncate the remaining bits
            if(round_bit == 1'd0)
            begin
                    outp = zeroExtend(prod_out[13:7]);
            end
            // If round bit is 1 do the following
            else
            begin
                  rem_nocarry = prod_out[5:0]; // To check the remaining bits
                  if(rem_nocarry == 6'd0 && prod_out[7] == 1'd0) // If remaining bits are 0 and LSB is also 0
                  begin
                            outp = zeroExtend(prod_out[13:7]); // Truncate the rest
                  end
                  else
                  begin // If remaining bits are non zero
                            carry_type_a = add_9bits(prod_out[15:7], 9'b1); // Add one to round up
                            if(carry_type_a[8] == 1) // See if the above addition results in a carry
                            begin
                                    exp = add_8bits(exp, 8'b1); // Adjust exponent
                                    outp = zeroExtend(carry_type_a[7:1]);
                            end
                            else
                            begin // If there is no carry while rounding up
                                    outp = zeroExtend(carry_type_a[6:0]);
                            end
                  end
            end

    end
    outp = add_15bits(outp, (zeroExtend(exp) << 7));
return outp;
endfunction:round
```

After the rounding is done, the output Bfnum type is populated with the answer as shown below:

```
rule round_nearest(got_A == True && got_B == True && sign_calculated == True && expone_calculated == True && calculate_mantissa == True && count == 5'd0 && rounding_done == False && handle_zero == False);
    rounding_done <= True;
    man_c_and_final_exp <= round(temp_prod, exp_c);

endrule

rule assemble_answer(got_A == True && got_B == True && sign_calculated == True && expone_calculated == True && calculate_mantissa == True && rounding_done == True && assembled_answer == False && handle_zero == False);
    assembled_answer <= True;
    bf_c <= Bfnum{ sign: sign_c, exponent: man_c_and_final_exp[14:7], fraction: man_c_and_final_exp[6:0] };
endrule
```

The “assembled_answer”, when set, triggers the value method to return the computed answer to higher level module.

**Handling corner case:**

When one of the inputs is detected to be zero, multiplication does not happen but “assembled_answer” is set and output Bfnum “bf_c” is set to zero and eventually returned to higher level module in next cycle.

```
rule handle_case_zero(got_A == True && got_B == True && handle_zero == True && handled_zero == False);
    assembled_answer <= True;
    handled_zero <= True;
    bf_c <= Bfnum{ sign: '0, exponent: '0, fraction: '0 };
endrule
```

**fp32_add.bsv:**

This module computes the floating addition of a Bfloat 16 number and fp32 number.

Takes the following as input:

1) 16-bit input A
2) 32-bit input B

Gives the following as output:

3) Output of Fpnum type

**Fpnum type:**

Fpnum is a structure with 1 bit sign, 8 bits exponent and 23 bits mantissa as its members as shown below:

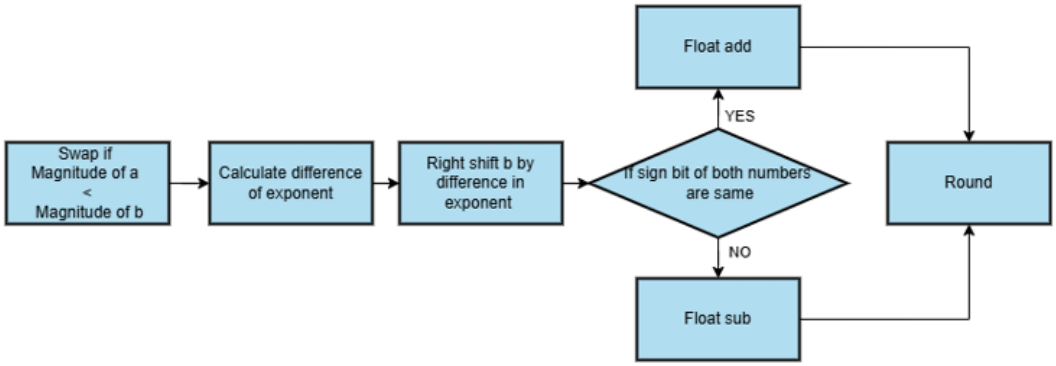
```
typedef struct {
    Bit#(1) sign;
    Bit#(8) exponent;
    Bit#(23) fraction;
} Fpnum deriving (Bits, Eq);
```

The obtained inputs are populated in Fpnum as shown below:

```
method Action get_A(Bit#(16) a) if (!got_A);
    got_A <= True;
    fp_a <= Fpnum{ sign: a[15], exponent: a[14:7], fraction: {a[6:0],16'b0} };
endmethod

method Action get_B(Bit#(32) b) if (!got_B);
    got_B <= True;
    fp_b <= Fpnum{ sign: b[31], exponent: b[30:23], fraction: b[22:0] };
endmethod
```

Input a is in bfloat 16. It is converted to fp32 format by the method get_A by appending 16 zeroes to the right side of mantissa.

After the inputs are obtained, the following flow is followed:




**Swap:**

The operands are swapped such that “a” is always bigger number. This will simplify the logic needed.

The following image shows the logic used:

```
rule swap_operands_if_needed(got_A == True && got_B == True && operands_swapped_if_needed == False && handle_zero == False && handle_oneinpzero == False);
    if(fp_a.exponent == fp_b.exponent && fp_a.fraction == fp_b.fraction && fp_a.sign != fp_b.sign) // Handles special case when addition results in zero
    begin
            handle_zero <= True;
    end
    else if((fp_a.exponent == '0 && fp_a.fraction == '0) || (fp_b.exponent == '0 && fp_b.fraction == '0)) // Handles special case when one of the inputs is zero
    begin
            if(fp_b.exponent == '0 && fp_b.fraction == '0)
            begin
                    fp_b.sign <= '0;
            end
            handle_oneinpzero <= True;
    end
    else
    begin
            if(fp_a.exponent < fp_b.exponent)
            begin
                    fp_a <= fp_b;
                    fp_b <= fp_a;
            end
            else if(fp_a.exponent == fp_b.exponent)
            begin
                    if(fp_a.fraction < fp_b.fraction)
                    begin
                            fp_a <= fp_b;
                            fp_b <= fp_a;
                    end
            end
    end
    operands_swapped_if_needed <= True;
endrule
```

This rule also identifies the following corner cases:

1) When A = -B (The answer is zero)
2) When either A or B is zero (The answer is simply the other non zero number)

To swap, first the exponents are compared, if they are same then mantissa is compared.

**Exponent difference:**

The below code calculates the difference in exponents:

```
rule calculate_expdiff(got_A == True && got_B == True && operands_swapped_if_needed == True && expdiff_calculated == False && handle_zero == False && handle_oneinpzero == False);
    temp_A <= {2'b01, fp_a.fraction, 25'b0};
    temp_B <= {2'b01, fp_b.fraction, 25'b0};
    expdiff_calculated <= True;
    expdiff <= add_8bits(fp_a.exponent, twos_compliment(fp_b.exponent));
endrule
```

It is just ripple carry adder with second input fed in twos compliment format

**Right shifting b:**

The below code contains the rule which right shifts the b input (lower magnitude)

```
rule add_prep(got_A == True && got_B == True && operands_swapped_if_needed == True && expdiff_calculated == True && add_prep_done == False && handle_zero == False && handle_oneinpzero == False);
    add_prep_done <= True;
    if(fp_a.sign == fp_b.sign)
    begin
            sign_c <= fp_a.sign;
            temp_B <= temp_B >> expdiff;
            do_add <= True;
    end
    else
    begin
            sign_c <= fp_a.sign;
            temp_B <= temp_B >> expdiff;
            do_sub <= True;
    end
endrule
```

temp_A and temp_B are chosen to be 50 bit registers **(An important design consideration)**

50 bits are sufficient because when exponent difference is large such that it causes the second input to right shift too much, round bit will become zero and eventually all bits of second input will be truncated.

We cannot use less than 49 bits, otherwise we won't be able to perform rounding correctly (We will loose bits needed to decide rounding flow due to right shift)

And we cannot use 49 bits in order to detect carry in float addition (to adjust exponent in upcoming steps).

So basically, 1 carry bit + 24 bits+1 round bit+ 24 bits(right shift worst case) = 50 bits are needed.

This rule also decides whether to add or subtract based on sign bits.

**Float add/sub:**

The following two rules perform float addition and subtraction respectively.

```
rule add(got_A == True && got_B == True && operands_swapped_if_needed == True && expdiff_calculated == True && add_prep
    do_add <= False;
    temp_sum <= add_50bits(temp_A, temp_B);
    round_addition_result <= True;
endrule

rule sub(got_A == True && got_B == True && operands_swapped_if_needed == True && expdiff_calculated == True && add_prep
    do_sub <= False;
    temp_sum <= sub_50bits(temp_A, temp_B);
    adj_sub <= True;
endrule
```

Addition and subtraction are done using 50 bit ripple carry adders.

**Addition using rca:**

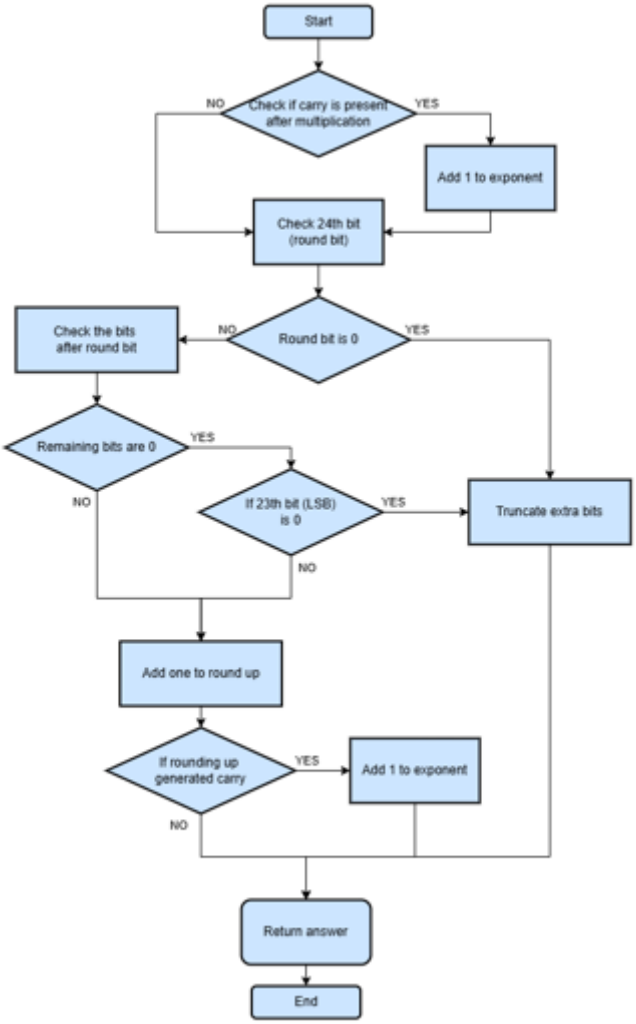
```
function Bit#(50) add_50bits(Bit#(50) a, Bit#(50) b);
    Bit#(50) outp = 50'b0;
    Bit#(1) carry = 1'b0;
    outp[0] = a[0] ^ b[0];
    carry = a[0] & b[0];
    for(Integer i = 1; i < 50; i = i + 1)
    begin
            outp[i] = a[i] ^ b[i] ^ carry;
            carry = (a[i] & b[i]) | (a[i] ^ b[i]) & carry;
    end

    return outp;
endfunction:add_50bits
```

**Subtraction using rca:**

```
 function Bit#(50) sub_50bits(Bit#(50) a, Bit#(50) b);
     Bit#(50) outp = 50'b0;
     Bit#(1) carry = 1'b0;
     Bit#(50) comp_b = 50'b0;
     comp_b = (b ^ '1) + 1;
     outp[0] = a[0] ^ comp_b[0];
     carry = a[0] & comp_b[0];
     for(Integer i = 1; i < 50; i = i + 1)
     begin
             outp[i] = a[i] ^ comp_b[i] ^ carry;
             carry = (a[i] & comp_b[i]) | (a[i] ^ comp_b[i]) & carry;
     end

     return outp;
 endfunction:sub_50bits
```

**Round:**

The rounding method used is same as used in float multiplication, with minor differences. Below is the flowchart:




Note that the logic is the same, but the checking of round bit is at $24^{th}$ bit and LSB is at $23^{rd}$ bit.

The rounding operation is split into two, one for addition and other for subtraction to account for the fact that in addition carry will be generated, and in subtraction borrow(MSB will become zero) will be generated and the relative positions of LSB and round bit will vary.

Rounding for addition: Captured by the function -> round_afteradd

Rounding for subtraction : Captured by the function -> round_aftersub

**round_afteradd:**

```
function Bit#(31) round_afteradd(Bit#(50) add_out, Bit#(8) exp);
    Bit#(31) outp = 31'b0;
    Bit#(1) round_bit = 1'b0;
    Bit#(24) rem_nocarry = 24'b0;
    Bit#(25) rem_withcarry = 25'b0;
    Bit#(25) carry_type_a = 25'b0;
    Bit#(25) carry_type_b = 25'b0;

    if(add_out[49] == 1'd1) // If carry is generated during addition
    begin
            exp = add_8bits(exp, 8'b1);
            round_bit = add_out[25];
            // If round bit is 0 truncate the remaining bits
            if(round_bit == 1'd0)
            begin
                    outp = zeroExtend(add_out[48:26]);
            end
            // If round bit is 1 do the following
            else
            begin
                  rem_withcarry = add_out[24:0]; // To check the remaining bits
                  if(rem_withcarry == 25'd0 && add_out[26] == 1'd0) // If remaining bits are 0 and LSB is also 0
                  begin
                            outp = zeroExtend(add_out[48:26]); // Truncate the rest
                  end
                  else
                  begin // If remaining bits are non zero
                            carry_type_b = add_25bits(zeroExtend(add_out[49:26]) , 25'b1); // Add one to round up
                            if(carry_type_b[24] == 1) // See if the above addition results in a carry
                            begin
                                    exp = add_8bits(exp, 8'b1); // Adjust exponent
                                    outp = zeroExtend(carry_type_b[23:1]);
                            end
                            else
                            begin // If there is no carry while rounding up
                                    outp = zeroExtend(carry_type_b[22:0]);
                            end
                  end
            end

    end
```

The code in the above image handles the rounding of addition result, when carry is generated.

```
    else  // If carry is not generated during addition
    begin
            round_bit = add_out[24];
            // If round bit is 0 truncate the remaining bits
            if(round_bit == 1'd0)
            begin
                    outp = zeroExtend(add_out[47:25]);
            end
            // If round bit is 1 do the following
            else
            begin
                  rem_nocarry = add_out[23:0]; // To check the remaining bits
                  if(rem_nocarry == 24'd0 && add_out[25] == 1'd0) // If remaining bits are 0 and LSB is also 0
                  begin
                          outp = zeroExtend(add_out[47:25]); // Truncate the rest
                  end
                  else
                  begin // If remaining bits are non zero
                          carry_type_a = add_25bits(add_out[49:25], 25'b1); // Add one to round up
                          if(carry_type_a[24] == 1) // See if the above addition results in a carry
                          begin
                                  exp = add_8bits(exp, 8'b1); // Adjust exponent
                                  outp = zeroExtend(carry_type_a[23:1]);
                          end
                          else
                          begin // If there is no carry while rounding up
                                  outp = zeroExtend(carry_type_a[22:0]);
                          end
                  end
            end

    end
    outp = add_31bits(outp, (zeroExtend(exp) << 23));
return outp;
endfunction:round_afteradd
```

The code in the above image handles the rounding of addition result, when carry is not generated.

**round_aftersub:**

```
function Bit#(31) round_aftersub(Bit#(50) sub_out, Bit#(8) exp);
    Bit#(31) outp = 31'b0;
    Bit#(1) round_bit = 1'b0;
    Bit#(24) rem_nocarry = 24'b0;
    Bit#(23) rem_bits = 23'b0;
    Bit#(25) carry_type_a = 25'b0;
    Bit#(25) carry_type_b = 25'b0;

    if(sub_out[48] == 1'd0) // If MSB is zero during subtraction
    begin
            exp = exp - 8'b1;
            round_bit = sub_out[23];
            // If round bit is 0 truncate the remaining bits
            if(round_bit == 1'd0)
            begin
                    outp = zeroExtend(sub_out[46:24]);
            end
            // If round bit is 1 do the following
            else
            begin
                  rem_bits = sub_out[22:0]; // To check the remaining bits
                  if(rem_bits == 23'd0 && sub_out[24] == 1'd0) // If remaining bits are 0 and LSB is also 0
                  begin
                          outp = zeroExtend(sub_out[46:24]); // Truncate the rest
                  end
                  else
                  begin // If remaining bits are non zero
                          carry_type_b = add_25bits(zeroExtend(sub_out[47:23]) , 25'b1); // Add one to round up
                          if(carry_type_b[24] == 1) // See if the above addition results in a carry
                          begin
                                  exp = add_8bits(exp, 8'b1); // Adjust exponent
                                  outp = zeroExtend(carry_type_b[23:1]);
                          end
                          else
                          begin // If there is no carry while rounding up
                                  outp = zeroExtend(carry_type_b[22:0]);
                          end
                  end
            end
```

The code in the above image handles the rounding of subtraction result, when borrow is generated.

```
    end
    else  // If MSB is not zero after subtraction
    begin
            round_bit = sub_out[24];
            // If round bit is 0 truncate the remaining bits
            if(round_bit == 1'd0)
            begin
                    outp = zeroExtend(sub_out[47:25]);
            end
            // If round bit is 1 do the following
            else
            begin
                  rem_nocarry = sub_out[23:0]; // To check the remaining bits
                  if(rem_nocarry == 24'd0 && sub_out[25] == 1'd0) // If remaining bits are 0 and LSB is also 0
                  begin
                            outp = zeroExtend(sub_out[47:25]); // Truncate the rest
                  end
                  else
                  begin // If remaining bits are non zero
                            carry_type_a = add_25bits(sub_out[49:25], 25'b1); // Add one to round up
                            if(carry_type_a[24] == 1) // See if the above addition results in a carry
                            begin
                                    exp = add_8bits(exp, 8'b1); // Adjust exponent
                                    outp = zeroExtend(carry_type_a[23:1]);
                            end
                            else
                            begin // If there is no carry while rounding up
                                    outp = zeroExtend(carry_type_a[22:0]);
                            end
                  end
            end

    end
    outp = add_31bits(outp, (zeroExtend(exp) << 23));
return outp;
endfunction:round_aftersub
```

The code in the above image handles the rounding of subtraction result, when borrow is not generated.

**Handling of corner cases:**

**Decrementing exponent:**

```
rule adjust_subres(got_A == True && got_B == True && operands_swapped_if_needed == True && expdiff_calculated == 
    if(temp_sum[48] == 1'b1)
    begin
            adj_done <= True;
            round_subtraction_result <= True;
    end
    else
    begin
            fp_a.exponent <= sub_8bits(fp_a.exponent, 8'b1);
            temp_sum <= temp_sum << 1;
    end

endrule
```

The above rule will decrement exponent upon detection of borrow.

**Handling zeroes:**

```
rule handle_zero_case(handle_zero == True && handle_oneinpzero == False);
    handle_zero <= False;
    assembled_answer <= True;
    fp_c <= Fpnum{ sign: '0, exponent: '0, fraction: '0};
endrule

rule handle_oneinpzero_case(handle_oneinpzero == True);
    handle_oneinpzero <= False;
    assembled_answer <= True;
    fp_c <= Fpnum{ sign: fp_a.sign | fp_b.sign, exponent: fp_a.exponent | fp_b.exponent, fraction: fp_a.fraction | fp_b.fraction};
endrule
```

The above rules will handle if either the addition results in zero or if either of the inputs is zero.

**Important design consideration: Negative zeroes are converted to positive zeros in this design**

Finally, there are rules which coordinates all of the above said functions to get expected answer.

**VERIFICATION:**

**The verification flow is very strict and does not allow any leniency in output even if the question allows for error in last 2 bits of LSB.**

The given testcases for Int contained both positive and negative numbers. BSV and Reference model both passed the given testcases

The float testcases had only 1000 positive testcases. First the BSV and RM are made to pass these testcases.

**Float mul testcases augmentation:**

Then the sign bit of A and B binary are flipped to get the negative versions of given testcases.

The Output AB sign bit is also flipped according to below table and verified both BSV and reference model (Found and fixed few bugs)

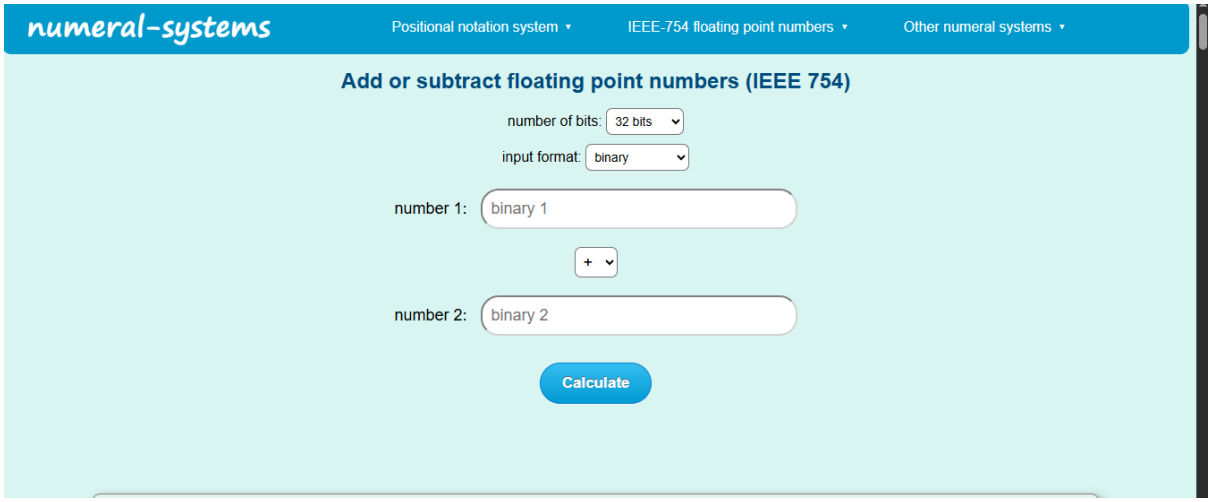
| A | B | AB |
|---|---|---|
| POSITIVE | POSITIVE | POSITIVE |
| POSITIVE | NEGATIVE | NEGATIVE |
| NEGATIVE | POSITIVE | NEGATIVE |
| NEGATIVE | NEGATIVE | POSITIVE |

**Float add testcases augmentation:**

Then wanted to test negative versions of given inputs to float add. But the given cases didn't have negative MAC output.

Then a decision is made to web scrap the online calculator: Add or subtract floating point numbers (IEEE 754) To automatically provide inputs to website and obtain outputs using selenium library in python.

**Preview of online calaculator:**



**Webscrapping code:**

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time

from selenium.webdriver.chrome.options import Options
chrome_options = Options()
chrome_options.add_experimental_option("detach", True)

web = webdriver.Chrome(options=chrome_options)
web.get('https://numeral-systems.com/ieee-754-add/')

file = open("Padded_negAB_output.txt","r")
AB = file.readlines()
file.close()

file = open("negC_binary.txt","r")
C = file.readlines()
file.close()

file = open("NN_MAC_binary.txt","w")


Cookies = web.find_element(By.XPATH,'//*[@id="cookie-banner-buttons-container"]/button[2]')
input_1 = web.find_element(By.XPATH,'//*[@id="number-input-1"]')
input_2 = web.find_element(By.XPATH,'//*[@id="number-input-2"]')
Submit = web.find_element(By.XPATH,'//*[@id="submit-button"]')
```

```
    Cookies = web.find_element(By.XPATH,'//*[@id="cookie-banner-buttons-container"]/button[2]')
    input_1 = web.find_element(By.XPATH,'//*[@id="number-input-1"]')
    input_2 = web.find_element(By.XPATH,'//*[@id="number-input-2"]')
    Submit = web.find_element(By.XPATH,'//*[@id="submit-button"]')

    Cookies.click()


    for i in range(len(AB)):
        input_1.send_keys(AB[i])
        input_2.send_keys(C[i])

        Submit.click()
        # time.sleep(5)
        Output = web.find_element(By.XPATH,'//*[@id="result-path-container"]/div[8]')
        file.write(Output.text+"\n")
        print(f"Done {i+1}  {Output.text}")

        input_1.clear()
        input_2.clear()

    file.close()
    print("FINISHED!!!")
```

The obtained testcases helped to discover the expected rounding strategy and helped in creating reference model.

Creating reference model gave the understanding needed to create BSV codes.

The following table summarises the testcases expansion for float add

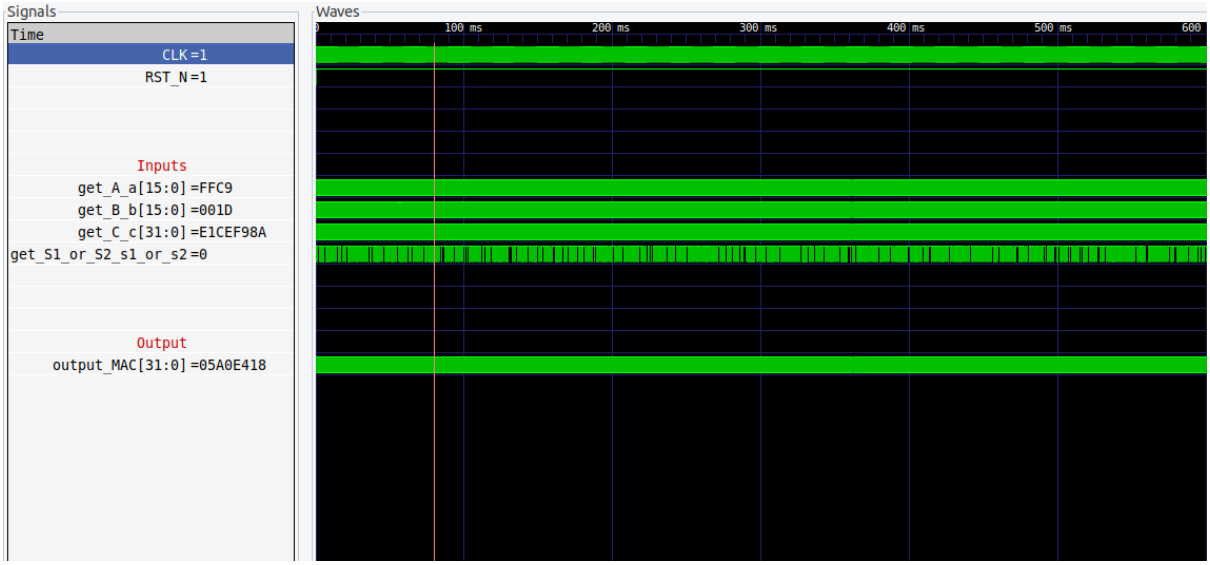
| A | B | AB | C | MAC |
|---|---|---|---|---|
| POSITIVE | POSITIVE | POSITIVE | POSITIVE | PP cases |
| POSITIVE | NEGATIVE | NEGATIVE | POSITIVE | NP cases |
| NEGATIVE | POSITIVE | NEGATIVE | POSITIVE | NP cases |
| NEGATIVE | NEGATIVE | POSITIVE | POSITIVE | PP cases |
| POSITIVE | POSITIVE | POSITIVE | NEGATIVE | PN cases |
| POSITIVE | NEGATIVE | NEGATIVE | NEGATIVE | NN cases |
| NEGATIVE | POSITIVE | NEGATIVE | NEGATIVE | NN cases |
| NEGATIVE | NEGATIVE | POSITIVE | NEGATIVE | PN cases |

Therefore, the testcases were expanded from 2000 to 9000

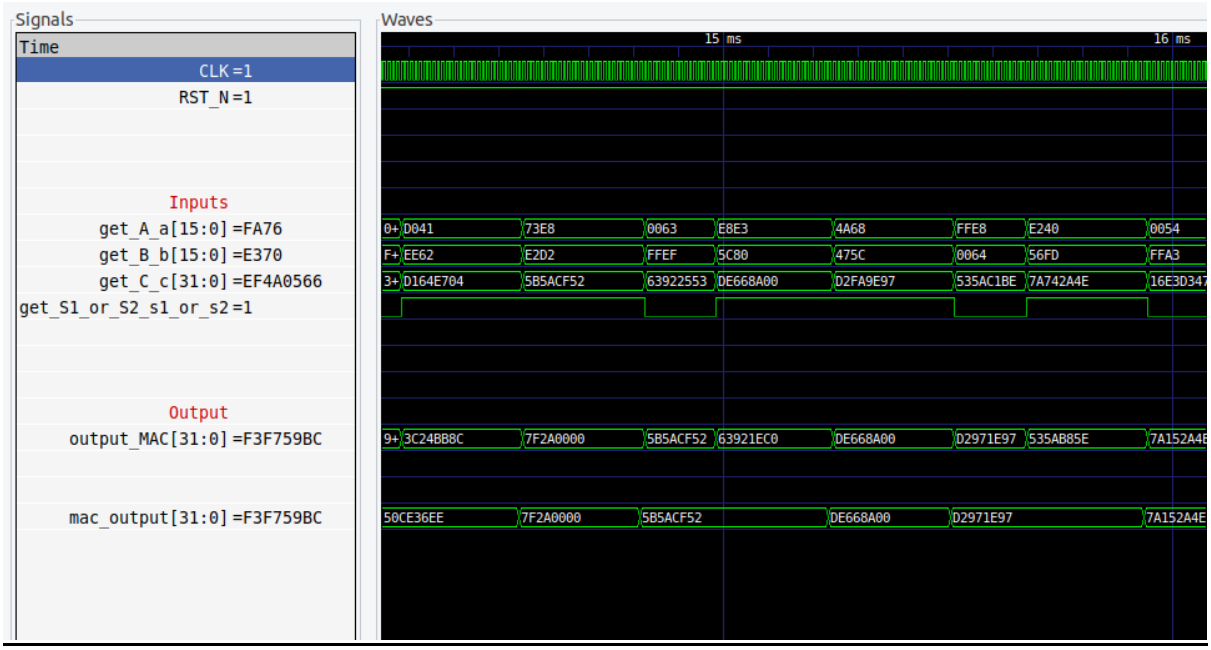
Then few corner cases with zeroes, all ones, walking ones, walking zeroes and alternating ones are tested.

GTKWAVE is used for waveform visualisation and debugging

The following waveform shows random inputs being fed to RTL and getting output.




The below waveform shows the zoomed in version of above waveform so that each signal can be seen clearly.



**TESTBENCH:**

A testbench is written: test_mkMAC_unpipelined.py

It can test the following:

1) 2000 given + expanded testcases (float and Int combined) = 9000 testcases
2) Corner testcases
3) Random inputs testing (NaN testcases are filtered out) = 15000 testcases
4) Coverage calculations

The assertions are made between both RTL and Reference model and between RTL and given testcases. **The last two bits leniency is not followed during assertion, rather all bits are checked and the testcase passes only if all bits are right.**

**Testbench explanation:**

The entire testbench is explained in this section. The test bench is divided into small chucks of images with explanation below each image.

```
import os
import random
from pathlib import Path

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge, ClockCycles
import logging as _log

from FLOAT_RM import *
from INT_RM import *
```

The above image shows the imports done in testbench. Notice FLOAT_RM and INT_RM being imported, these are reference models.

```
# Ports:
# Name                       I/O  size props
# RDY_get_A                    O     1
# RDY_get_B                    O     1
# RDY_get_C                    O     1
# RDY_get_S1_or_S2             O     1
# output_MAC                   O    32 reg
# RDY_output_MAC               O     1 reg
# CLK                          I     1 clock
# RST_N                        I     1 reset
# get_A_a                      I    16 reg
# get_B_b                      I    16 reg
# get_C_c                      I    32 reg
# get_S1_or_S2_s1_or_s2        I     1 reg
# EN_get_A                     I     1
# EN_get_B                     I     1
# EN_get_C                     I     1
# EN_get_S1_or_S2              I     1
```

The above image contains the interface obtained in generated Verilog present as comments for easier reference while coding.

```python
async def reset(dut):
    dut.RST_N.value = 1
    await RisingEdge(dut.CLK)
    dut.RST_N.value = 0
    await RisingEdge(dut.CLK)
    dut.RST_N.value = 1
    await RisingEdge(dut.CLK)
```

The above function resets the MAC unit. It is called only once at the beginning of test.

```python
async def give_input(dut,A,B,C,S):
    dut.get_A_a.value = A
    dut.get_B_b.value = B
    dut.get_C_c.value = C
    dut.get_S1_or_S2_s1_or_s2.value = S
    await RisingEdge(dut.CLK)
    dut.EN_get_A.value = 1
    dut.EN_get_B.value = 1
    dut.EN_get_C.value = 1
    dut.EN_get_S1_or_S2.value = 1
    await RisingEdge(dut.CLK)
    dut.EN_get_A.value = 0
    dut.EN_get_B.value = 0
    dut.EN_get_C.value = 0
    dut.EN_get_S1_or_S2.value = 0
```

The above function gives input to RTL.

```python
async def get_output_float(dut):
    await RisingEdge(dut.RDY_output_MAC)
    return dut.output_MAC.value
```

The above function gets float output from RTL

```python
async def get_output_int(dut):
    await RisingEdge(dut.RDY_output_MAC)
    rtl_answer =  dut.output_MAC.value
    str_ans = str(rtl_answer)
    if(str_ans[0] == "1"):
        rtl_answer = ((int(str_ans,2) ^ 0xFFFFFFFF) + 1) * -1
    else:
        rtl_answer = int(str(rtl_answer),2)
    return rtl_answer
```

The above function gets integer results from RTL and converts the negative numbers into assertable format.

```python
def create_random_float16():
    S,E = random.randint(0,1),random.randint(0,0xFF)
    if(E == 0xFF):
        M = 0
    else:
        M = random.randint(0,0x7F)
    return bfmk(S,E,M)

def create_random_float32():
    S,E = random.randint(0,1),random.randint(0,0xFF)
    if(E == 0xFF):
        M = 0
    else:
        M = random.randint(0,0x7FFFFF)
    return fpmk(S,E,M)
```

The above two functions are used to generate random floating-point numbers. Nan are avoided while generating random numbers by those if-else conditions.

```python
@cocotb.test()
async def test_MAC_unpipelined(dut):

    # Choose type of test
    test_float = 1
    test_int = 1
    test_random = 1
    test_indiv = 0
```

This is main testbench function. The user can choose whether to test float only, int only, random inputs or test a individual case.

```python
clock = Clock(dut.CLK, 10, units="us")
cocotb.start_soon(clock.start(start_high=False))
await reset(dut)
```

The code in above image starts the clock and resets the DUT.

```python
if(test_indiv == 1):
    await give_input(dut,int("1110111011110010",2),int("0101000001111100",2),int("11111110011101010000111001110111",2),1)
    rtl_output = await get_output_float(dut)
    print("RTL:",str(rtl_output))
```

The code above allows the user to test a particular input. This is useful for debugging problems.

```
LA = []
LB = []
LAB = []

# Test float

file_a = open("Values/combined_A_binary.txt","r")
LA = file_a.readlines()
file_a.close()

file_b = open("Values/combined_B_binary.txt","r")
LB = file_b.readlines()
file_b.close()

file_c = open("Values/combined_C_binary.txt","r")
LC = file_c.readlines()
file_c.close()

file_MAC = open("Values/combined_MAC_binary.txt","r")
LAB = file_MAC.readlines()
file_MAC.close()
```

The code above reads the text files containing the expanded testcases for float and stores in lists.

```
# Inserting special cases

# Float addition results in zero
LA = LA[:49] + ["0011111110000000"] + LA[49:]
LB = LB[:49] + ["0100000100101100"] + LB[49:]
LC = LC[:49] + ["11000001001011000000000000000000"] + LC[49:]
LAB = LAB[:49] + ["0"*32] + LAB[49:]

# Input A is zero
LA = LA[:100] + ["0"*16] + LA[100:]
LB = LB[:100] + ["0100000100101100"] + LB[100:]
LC = LC[:100] + ["10"*16] + LC[100:]
LAB = LAB[:100] + ["10"*16] + LAB[100:]

# Input B is zero
LA = LA[:1000] + ["0100000100101100"] + LA[1000:]
LB = LB[:1000] + ["0"*16] + LB[1000:]
LC = LC[:1000] + ["10"*16] + LC[1000:]
LAB = LAB[:1000] + ["10"*16] + LAB[1000:]

# Input C is zero
LA = LA[:1200] + ["0100000100101100"] + LA[1200:]
LB = LB[:1200] + ["0011111110000000"] + LB[1200:]
LC = LC[:1200] + ["0"*32] + LC[1200:]
LAB = LAB[:1200] + ["01000001001011000000000000000000"] + LAB[1200:]

# Input A is -ve zero
LA = LA[:2000] + ["1"+"0"*15] + LA[2000:]
LB = LB[:2000] + ["0100000100101100"] + LB[2000:]
LC = LC[:2000] + ["10"*16] + LC[2000:]
LAB = LAB[:2000] + ["10"*16] + LAB[2000:]
```

```
# Input B is -ve zero
LA = LA[:3000] + ["0100000100101100"] + LA[3000:]
LB = LB[:3000] + ["1"+"0"*15] + LB[3000:]
LC = LC[:3000] + ["10"*16] + LC[3000:]
LAB = LAB[:3000] + ["10"*16] + LAB[3000:]

# Input C is -ve zero
LA = LA[:5200] + ["0100000100101100"] + LA[5200:]
LB = LB[:5200] + ["0011111110000000"] + LB[5200:]
LC = LC[:5200] + ["1"+"0"*31] + LC[5200:]
LAB = LAB[:5200] + ["01000001001011000000000000000000"] + LAB[5200:]

# All inputs are zero
LA = LA[:7200] + ["0"*16] + LA[7200:]
LB = LB[:7200] + ["0"*16] + LB[7200:]
LC = LC[:7200] + ["0"*32] + LC[7200:]
LAB = LAB[:7200] + ["0"*32] + LAB[7200:]
```

The code in the above two images insert special cases to the list

```
# Corner cases indentified while analysing coverage
bfS = [0,1]*10
bfE = [0,0b11111110,0x55,0xAA,0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80,0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F]
bfM = [0,0b1111111,0x55,0x2A,0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x7E,0x7D,0x7B,0x77,0x6F,0x5F,0x3F,0x4,0x7E]

fpS = [0,1]*10
fpE = [0,0b11111110,0x55,0xAA,0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80,0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F]
fpM = [0,0x7FFFFF,0x555555,0x2AAAAA]*5


Bin_A = []
Bin_B = []
Bin_C = []

for i in range(20):
    temp_A = bfmk(bfS[i],bfE[i],bfM[i])
    temp_B = bfmk(bfS[i],bfE[i],bfM[i])
    temp_C = fpmk(fpS[i],fpE[i],fpM[i])
    RM_output = MAC_fp32_RM(temp_A,temp_B,temp_C)
    if(RM_output != "EXCEPTION"):
        Bin_A.append(temp_A)
        Bin_B.append(temp_B)
        Bin_C.append(temp_C)
```

The code in above image generates testcases with alternating ones, walking zeroes and walking ones. While permuting on special cases for individual components like sign, exponent and mantissa there is a possibility of generating a Nan, so the created float numbers are given to MAC_fp32_RM reference model and if the reference model returns “EXCEPTION” that testcase is not tested.

```
testcase_counter = 0
if(test_float == 1):
    print("TESTING FLOAT INPUTS")
    for i in range(len(LA)):
        testcase_counter += 1
        await give_input(dut,int(LA[i],2),int(LB[i],2),int(LC[i],2),1)
        rtl_output = await get_output_float(dut)
        assert str(rtl_output) == LAB[i].strip("\n") # assertion between RTL and expected value
        RM_output = MAC_fp32_RM(LA[i].strip("\n"),LB[i].strip("\n"),LC[i].strip("\n"))
        print("RTL:",str(rtl_output),"EXPECTED:",LAB[i].strip("\n"),"RM:",RM_output,f"TESTCASE {testcase_counter}")
        assert str(rtl_output) == RM_output # assertion between RTL and reference model value

    for i in range(len(Bin_A)):
        testcase_counter += 1
        await give_input(dut,int(Bin_A[i],2),int(Bin_B[i],2),int(Bin_C[i],2),1)
        rtl_output = await get_output_float(dut)
        RM_output = MAC_fp32_RM(Bin_A[i],Bin_B[i],Bin_C[i])
        print("RTL:",str(rtl_output),"RM:",RM_output,f"TESTCASE {testcase_counter}")
        assert str(rtl_output) == RM_output # assertion between RTL and reference model value
```

The above code provides float testcases as input to RTL, monitor the output and performs assertions.

The first for loop gives the expanded testcases as inputs an performs the following two-way assertions:

RTL output == Reference model output

RTL output == Given output in text file

The second for loop provides the float special cases and asserts the output. It performs just RTL output == reference model output, because given testcases did not expected answers for corner cases.

```
# Int MAC test

LA = []
LB = []
LC = []
LO = []

A_File = open("Values/A_decimal.txt","r")
A_List = A_File.readlines()
A_File.close()

B_File = open("Values/B_decimal.txt","r")
B_List = B_File.readlines()
B_File.close()

C_File = open("Values/C_decimal.txt","r")
C_List = C_File.readlines()
C_File.close()

O_File = open("Values/MAC_decimal.txt","r")
O_List = O_File.readlines()
O_File.close()

for i in range(len(A_List)-1):
    LA.append(eval(A_List[i].strip().strip('\n')))

for i in range(len(B_List)-1):
    LB.append(eval(B_List[i].strip().strip('\n')))

for i in range(len(C_List)-1):
    LC.append(eval(C_List[i].strip().strip('\n')))

for i in range(len(O_List)-1):
    LO.append(eval(O_List[i].strip().strip('\n')))
```

The code in above image reads the integer testcases from given text files and stores it in list for usage.

```
# Inserting special cases

# Int addition results in zero
LA = LA[:49] + [25] + LA[49:]
LB = LB[:49] + [1] + LB[49:]
LC = LC[:49] + [-25] + LC[49:]
LO = LO[:49] + [0] + LO[49:]

# Input A is zero
LA = LA[:100] + [0] + LA[100:]
LB = LB[:100] + [7] + LB[100:]
LC = LC[:100] + [555] + LC[100:]
LO = LO[:100] + [555] + LO[100:]

# Input B is zero
LA = LA[:500] + [15] + LA[500:]
LB = LB[:500] + [0] + LB[500:]
LC = LC[:500] + [100] + LC[500:]
LO = LO[:500] + [100] + LO[500:]

# Input C is zero
LA = LA[:700] + [100] + LA[700:]
LB = LB[:700] + [-2] + LB[700:]
LC = LC[:700] + [0] + LC[700:]
LO = LO[:700] + [-200] + LO[700:]

# All inputs are zero
LA = LA[:900] + [0] + LA[900:]
LB = LB[:900] + [0] + LB[900:]
LC = LC[:900] + [0] + LC[900:]
LO = LO[:900] + [0] + LO[900:]
```

The code in above image inserts special testcases in list

```
CA = list(range(-128,128))
CB = list(range(-128,128))
CC = [0,1,-1,0xFFFFFFFF,0x7FFFFFFF,0xAAAAAAAA,0x55555555,4294967294, 4294967293, 4294967291, 4294967287, 4294967279, 4294967263, 4294967231, 42

l = len(CC)
for i in range(len(CA)-1):
    CC.append(CC[random.randint(0,l-1)])
```

The code in above image generates testcases with alternating ones, walking zeroes and walking ones

```
count_1 = 0
if(test_int == 1):
    print("TESTING INTEGER INPUTS")
    for i in range(len(LA)):
        testcase_counter += 1
        await give_input(dut,LA[i],LB[i],LC[i],0)
        rtl_output = await get_output_int(dut)
        RM_int = MAC_int32_RM(LA[i],LB[i],LC[i])
        print(f"Inp A: {LA[i]} Inp B: {LB[i]} Inp C: {LC[i]} EXPECTED: {LO[i]} RTL: {rtl_output} RM: {RM_int} TESTCASE {testcase_counter}")
        assert rtl_output == LO[i]   # assertion between RTL and expected value
        assert rtl_output == RM_int  # assertion between RTL and reference model value


    for i in range(len(CA)):
        testcase_counter += 1
        await give_input(dut,CA[i],CB[i],CC[i],0)
        rtl_output = await get_output_int(dut)
        RM_int = MAC_int32_RM(CA[i],CB[i],CC[i])
        print(f"Inp A: {CA[i]} Inp B: {CB[i]} Inp C: {CC[i]} RTL: {rtl_output} RM: {RM_int} TESTCASE {testcase_counter}")
        assert rtl_output == RM_int  # assertion between RTL and reference model value
```

The above code provides int testcases as input to RTL, monitor the output and performs assertions.

The first for loop gives the expanded testcases as inputs an performs the following two-way assertions:

RTL output == Reference model output

RTL output == Given output in text file

The second for loop provides the int special cases and asserts the output. It performs just RTL output == reference model output, because given testcases did not expected answers for corner cases.

```
# Random inputs testing

filerand = open("Values/random.txt","w")
S = [random.randint(0,1) for _ in range(15000)]
retry = 1
#S = [0]*5000
```

The code in above image opens random.txt to store the random inputs for debugging purposes. S is a list which contains 15000 elements. It chooses between int MAC and float MAC.

```
if(test_random == 1):
    print("TESTING RANDOM INPUTS")
    for i in range(len(S)):
        testcase_counter += 1
        if(S[i] == 1):
            while(retry == 1):
                A = create_random_float16()
                B = create_random_float16()
                C = create_random_float32()
                RM_output = MAC_fp32_RM(A,B,C)
                if(RM_output != "EXCEPTION"):
                    break

            await give_input(dut,int(A,2),int(B,2),int(C,2),1)
            rtl_output = await get_output_float(dut)
            print("RTL:",str(rtl_output),"RM:",RM_output,f"TESTCASE {testcase_counter}")
            filerand.write("A: "+A+" B: "+B+" C: "+C+" RTL: "+str(rtl_output)+" RM: "+RM_output+"\n")
            assert str(rtl_output) == RM_output # assertion between RTL and reference model value


        elif(S[i] == 0):
            A = random.randint(-128,127)
            B = random.randint(-128,127)
            C = random.randint(-2147483648,2147483647)
            await give_input(dut,A,B,C,0)
            rtl_output = await get_output_int(dut)
            RM_int = MAC_int32_RM(A,B,C)
            print(f"Inp A: {A} Inp B: {B} Inp C: {C} RTL: {rtl_output} RM: {RM_int} TESTCASE {testcase_counter}")
            filerand.write("A: "+str(A)+" B: "+str(B)+" C: "+str(C)+" RTL: "+str(rtl_output)+" RM: "+str(RM_int)+"\n")
            assert rtl_output == RM_int # assertion between RTL and reference model value
```

If S had the value of “1” at any particular iteration, random float number is generated and passed through reference model to check whether any NaN is generated, the random number is repeatedly generated until a valid set of inputs are obtained. Then these inputs are passed to RTL and the output is asserted with reference model output.

If S had the value of “0” at any particular iteration, random int number is generated and passed to RTL and the output is asserted with reference model output.

```
coverage_db.export_to_yaml(filename="coverage_MAC_unpipelined.yml")
```

The above line writes the coverage report to coverage_MAC_unpipelined.yml file.

**COVERAGE:**

The following image shows the coverage definition in Integer reference model:

```
import cocotb
from cocotb_coverage.coverage import *

MAC_INT_coverage = coverage_section(
    CoverPoint('top.A', vname='A', bins = list(range(-128,128))),
    CoverPoint('top.B', vname='B', bins = list(range(-128,128))),
    CoverPoint('top.C', vname='C', bins = [0,1,-1,0xFFFFFFFF,0x7FFFFFFF,0xAAAAAAAA,0x55555555,4294967294, 4294967293, 4294967291, 4294967287, 4294
)
```

The following image shows the coverage definition in float reference model:

```
def bfmk(S,E,M):
    return bin(S)[2:].ljust(1,"0")+bin(E)[2:].ljust(8,"0")+bin(M)[2:].ljust(7,"0")

def fpmk(S,E,M):
    return bin(S)[2:].ljust(1,"0")+bin(E)[2:].ljust(8,"0")+bin(M)[2:].ljust(23,"0")



bfS = [0,1]*10
bfE = [0,0b11111110,0x55,0xAA,0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80,0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F]
bfM = [0,0b1111111,0x55,0x2A,0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x7E,0x7D,0x7B,0x77,0x6F,0x5F,0x3F,0x4,0x7E]

fpS = [0,1]*10
fpE = [0,0b11111110,0x55,0xAA,0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80,0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F]
fpM = [0,0x7FFFFF,0x555555,0x2AAAAA]*5


Bin_A = []
Bin_B = []
Bin_C = []

for i in range(20):
    Bin_A.append(bfmk(bfS[i],bfE[i],bfM[i]))
    Bin_B.append(bfmk(bfS[i],bfE[i],bfM[i]))
    Bin_C.append(fpmk(fpS[i],fpE[i],fpM[i]))


MAC_FLOAT_coverage = coverage_section(
    CoverPoint('top.FLOAT.A', vname='A', bins = Bin_A),
    CoverPoint('top.FLOAT.B', vname='B', bins = Bin_B),
    CoverPoint('top.FLOAT.C', vname='C', bins = Bin_C)
)
```

## **RESULTS:**

MAC_UNPIPELINED_TEST_RESULT is log file containing the simulation output as a proof.

```
RTL: 11111101110010000100110110111100 RM: 11111101110010000100110110111100 TESTCASE 24319
Inp A: -74 Inp B: -41 Inp C: 37575088 RTL: 37578122 RM: 37578122 TESTCASE 24320
RTL: 01101001011101111011111100110010 RM: 01101001011101111011111100110010 TESTCASE 24321
Inp A: 35 Inp B: -13 Inp C: 1109307083 RTL: 1109306628 RM: 1109306628 TESTCASE 24322
Inp A: 87 Inp B: -4 Inp C: -1838036958 RTL: -1838037306 RM: -1838037306 TESTCASE 24323
Inp A: -45 Inp B: 48 Inp C: -2136476802 RTL: -2136478962 RM: -2136478962 TESTCASE 24324
Inp A: -110 Inp B: -100 Inp C: -1876686720 RTL: -1876675720 RM: -1876675720 TESTCASE 24325
Inp A: 15 Inp B: -86 Inp C: 462520850 RTL: 462519560 RM: 462519560 TESTCASE 24326
RTL: 11010111011011011000001000110010 RM: 11010111011011011000001000110010 TESTCASE 24327
Inp A: -94 Inp B: -22 Inp C: 675347446 RTL: 675349514 RM: 675349514 TESTCASE 24328
RTL: 11011010110100010000000000000000 RM: 11011010110100010000000000000000 TESTCASE 24329
Inp A: 31 Inp B: 6 Inp C: 1491786703 RTL: 1491786889 RM: 1491786889 TESTCASE 24330
RTL: 01101101111001010010001110010000 RM: 01101101111001010010001110010000 TESTCASE 24331
Inp A: 63 Inp B: -117 Inp C: 1517102657 RTL: 1517095286 RM: 1517095286 TESTCASE 24332
Inp A: 118 Inp B: 108 Inp C: -1360988314 RTL: -1360975570 RM: -1360975570 TESTCASE 24333
Inp A: 36 Inp B: -42 Inp C: -95034339 RTL: -95035851 RM: -95035851 TESTCASE 24334
Inp A: -48 Inp B: 59 Inp C: 1913823461 RTL: 1913820629 RM: 1913820629 TESTCASE 24335
Inp A: -42 Inp B: 58 Inp C: 173289747 RTL: 173287311 RM: 173287311 TESTCASE 24336
Inp A: -47 Inp B: -104 Inp C: -707780485 RTL: -707775597 RM: -707775597 TESTCASE 24337
/home/shakti/.pyenv/versions/3.8.10/envs/py38/lib/python3.8/site-packages/cocotb/outcomes.py:36: ResourceWarning: unclosed file <_io.Te
xtIOWrapper name='Values/random.txt' mode='w' encoding='UTF-8'>
  return gen.send(self.value)
ResourceWarning: Enable tracemalloc to get the object allocation traceback
5527225000.00ns INFO     test_MAC_unpipelined passed
5527225000.00ns INFO     ********************************************************************************************************
                         ** TEST                                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                         ********************************************************************************************************
                         ** test_mkMAC_unpipelined.test_MAC_unpipelined   PASS   5527225000.00          96.39   57342238.30  **
                         ********************************************************************************************************
                         ** TESTS=1 PASS=1 FAIL=0 SKIP=0                  5527225000.00          96.44   57312878.26  **
                         ********************************************************************************************************

- :0: Verilog $finish
make[2]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Unpipelined/MAC_unpipelined'
make[1]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Unpipelined/MAC_unpipelined'
(py38) shakti@daniel-VirtualBox:~/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Unpipelined/MAC_unpipelined$
```

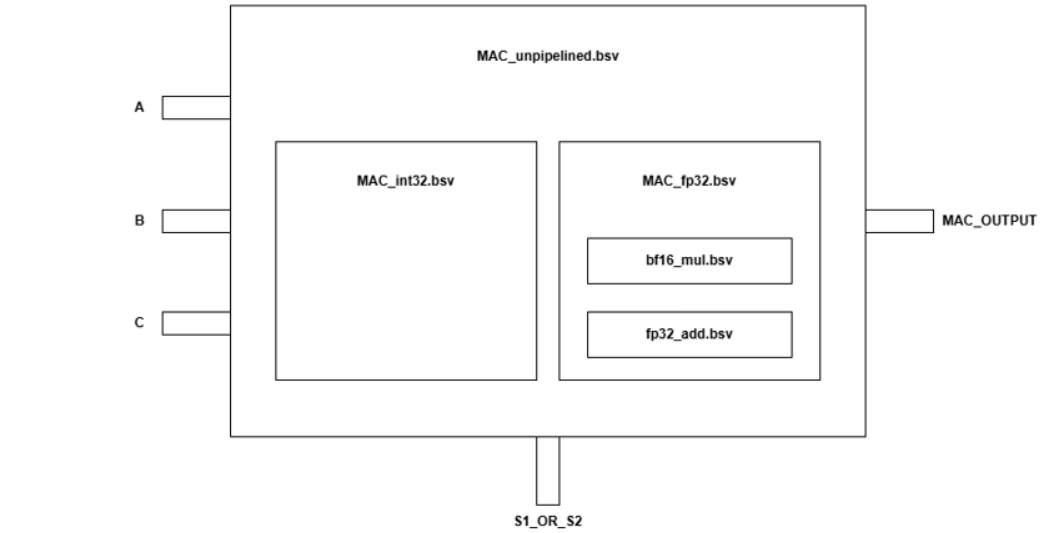
Refer coverage_MAC_unpipelined.yml for coverage report.

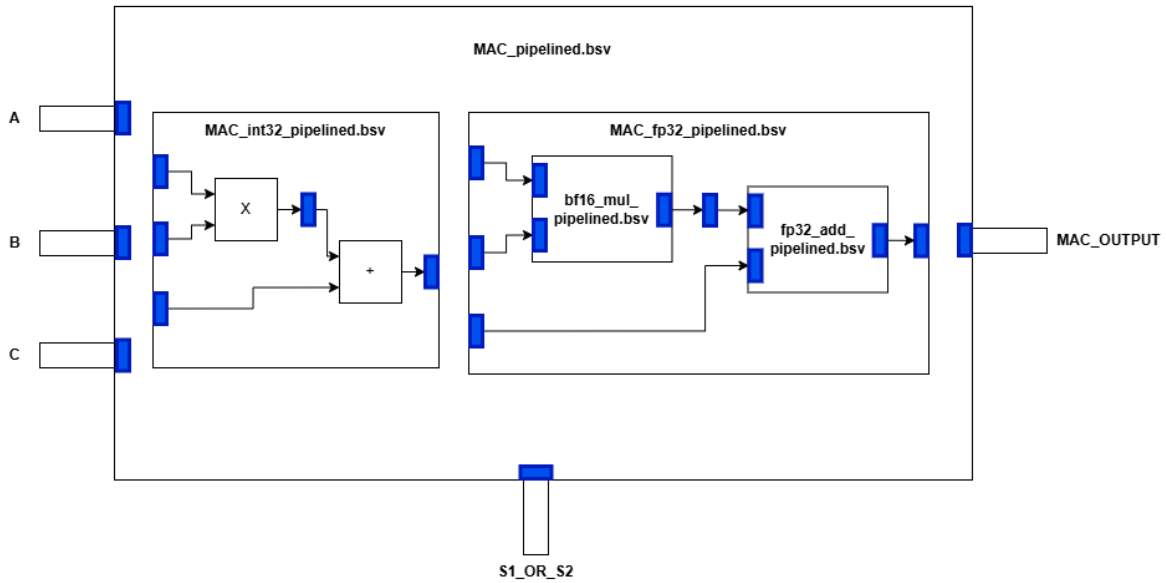
**The given testcases, expanded testcases(including corner cases) and random inputs passed! (With no leniency in output, all bits must be correct)**

# PIPLINED DESIGN

**DESIGN ARCHITECTURE:**

The following section deals with the pipelined version of MAC unit:




The blue boxes in the above diagram indicate the pipeline FIFOs added to the design.

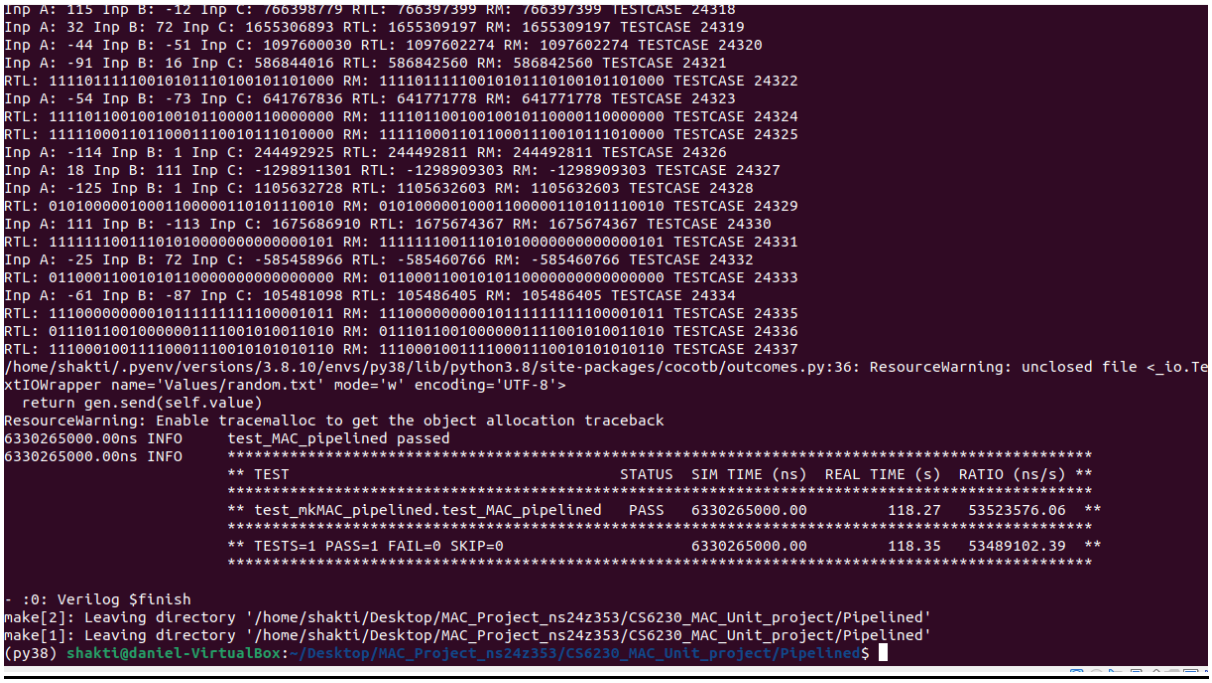
The pipelined FIFOs were added to each input and output of each block and multiplication output of int MAC and float MAC (Intermediate FIFO).

The addition of these FIFOs made it possible to remove many handshaking signals introduced in unpipelined design to orchestrate the rules. The bsv code looks way cleaner and readable because of the inbuilt handshaking (Implicit firing conditions that come with pipelined FIFO).

And one more design idea is implemented in pipelined version. All the struct definitions are put in one file called MAC_types.bsv and this file is imported wherever needed, this eliminated the instances where the bsv compiler gets confused with struct definitions being in multiple files during compilation.

**RESULT:**

MAC_PIPELINED_TEST_RESULT is log file containing the simulation output as a proof.




**The pipelined version is also tested with the same testbench and all testcases has passed! (With no leniency in output, all bits must be correct)**