

## CS6230\_MAC\_Unit\_project REPORT

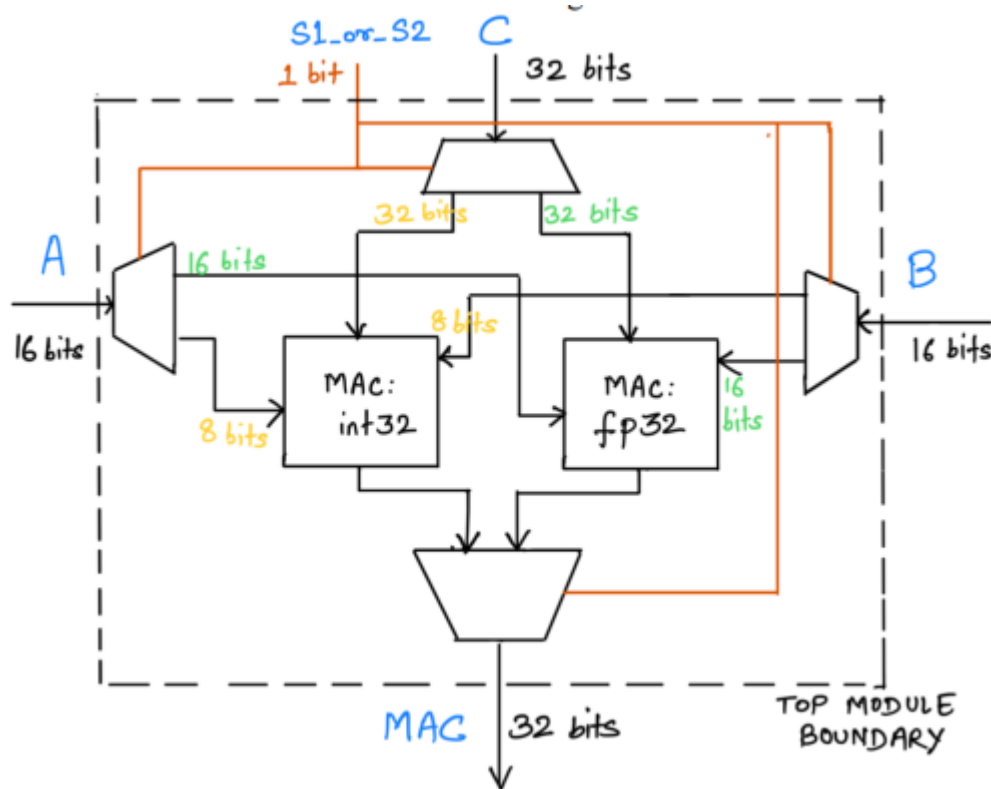
Done by: DANIEL MARK ISAAC

ROLL NO: NS24Z353

### Description:

This report summarises the MAC project.

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.

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

This report summarises a) MAC Unpipelined design.

**Approach taken:**

The following list is 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

### Reference Model:

The reference models are developed based on analysing the given testcases.

**Both MAC int and MAC float reference models are created and successfully verified with the given testcases**

The reference models are written in python. The calculations are performed using string manipulations. This decision is made because python does not know it is an N bit integer other than 32 bit integer. This fact caused problems when dealing with negative numbers and hence the decision.

**The reference models performs all calculations manually and does not use any in built python data types (for varying bit widths).**

### INT\_MAC RM:

```
@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

### FLOAT\_MAC RM:

#### Main function:

```
@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)

    # 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)
```

## Bfloat16\_mul function:

```
def bfloat16_mul(A,B):
    A_sign = int(A[0],2)
    A_exp = int(A[1:9],2)
    A_frac = int("1"+A[9:],2)

    B_sign = int(B[0],2)
    B_exp = int(B[1:9],2)
    B_frac = int("1"+B[9:],2)

    bias = int("10000001",2)

    AB_sign = bin(A_sign ^ B_sign)[2:]
    AB_exp = bin((A_exp + B_exp + bias) & 0xFF)[2:][-8:]

    # Multiplication of mantissa
    temp_A = bin(A_frac)[2:]
    for i in range(len(temp_A)):
        if(temp_A[-1] == "1"):
            break
        temp_A = temp_A[:-1]

    temp_B = bin(B_frac)[2:]
    for i in range(len(temp_B)):
        if(temp_B[-1] == "1"):
            break
        temp_B = temp_B[:-1]

    nob_A = len(temp_A)
    nob_B = len(temp_B)

    temp_AB = int(temp_A,2) * int(temp_B,2)
    nob_AB = len(bin(temp_AB)[2:])

    exp_adj = nob_AB - (nob_A + nob_B - 1)
    AB_exp = bin(int(AB_exp,2) + exp_adj)[2:]

    round_ret = round_bfloat16(bin(A_frac * B_frac)[3:])
    if(round_ret[1] == 0):
        AB_frac = round_ret[0]
    else:
        AB_exp = bin(int(AB_exp,2) + round_ret[1])[2:]
        AB_frac = round_ret[0]

    return AB_sign + AB_exp.rjust(8,"0") + AB_frac
```

### Round\_bfloat16 function:

```
def round_bfloat16(A):
    # input and output are str
    A += "0"
    adj_exp = 0
    round_bit = A[7]
    if(round_bit == "0"):
        return [A[:7], adj_exp]
    elif(round_bit == "1"):
        temp = A[8:]
        if(int(temp, 2) == 0 and A[6] == "0"):
            return [A[:6]+"0", adj_exp]
        else:
            # possibility of carry being generated is there
            carry_check = bin(int(A[:7], 2)+1)[2:]
            if(len(carry_check) > 7):
                adj_exp = len(carry_check) - 7
                return [carry_check[1:8], adj_exp]
            else:
                return [bin(int(A[:7], 2)+1)[2:].rjust(7, "0"), adj_exp]
```

### fp3\_add function:

```
def fp32_add(A, B):
    A_sign = A[0]
    A_exp = int(A[1:9], 2)
    A_frac = "1"+A[9:].ljust(23, "0")

    B_sign = B[0]
    B_exp = int(B[1:9], 2)
    B_frac = "1"+B[9:]

    exp_diff = A_exp - B_exp

    if(exp_diff < 0):
        A_sign, B_sign = B_sign, A_sign
        A_exp, B_exp = B_exp, A_exp
        A_frac, B_frac = B_frac, A_frac

    for i in range(abs(exp_diff)):
        B_frac = "0" + B_frac

    Blen_bef_add = len(B_frac)
    C_exp = A_exp
```

```

if(A_sign == B_sign):
    # addition ...
    A_frac = A_frac.ljust(Blen_bef_add,"0")
    sum_val = bin(int(A_frac,2) + int(B_frac,2))[2:]
    # carry detection
    if(len(sum_val) > Blen_bef_add):
        # Adjust exponent
        exp_add_diff = len(sum_val) - Blen_bef_add
        A_exp += exp_add_diff

    round_ret = round_fp32(sum_val[1:])
    if(round_ret[1] == 0):
        rounded_sum = round_ret[0]
    else:
        A_exp = bin(int(A_exp,2) + round_ret[1])[2:]
        rounded_sum = round_ret[0]

    return A_sign + bin(A_exp)[2:].rjust(8,"0") + rounded_sum

```

```

else:
    A_frac = A_frac.ljust(Blen_bef_add,"0")
    if(exp_diff == 0):
        if(A_frac < B_frac):
            A_sign = B_sign
            A_frac, B_frac = B_frac, A_frac
        diff_val = bin(int(A_frac,2) - int(B_frac,2))[2:]

        if(len(diff_val) < Blen_bef_add):
            # Adjust exponent
            exp_sub_diff = len(diff_val) - Blen_bef_add
            A_exp -= abs(exp_sub_diff)

        round_ret = round_fp32(diff_val[1:])
        if(round_ret[1] == 0):
            rounded_sum = round_ret[0]
        else:
            A_exp = A_exp + round_ret[1]
            rounded_sum = round_ret[0]

    return A_sign + bin(A_exp)[2:].rjust(8,"0") + rounded_sum

```

round\_fp32 function:

```
def round_fp32(A):
    # input and output are str
    A = A.ljust(25,"0")
    adj_exp = 0
    round_bit = A[23]
    if(round_bit == "0"):
        return [A[:23],adj_exp]
    elif(round_bit == "1"):
        temp = A[24:]
        if(int(temp,2) == 0 and A[22] == "0"):
            return [A[:22]+"0",adj_exp]
        else:
            # possibility of carry being generated is there
            carry_check = bin(int(A[:23],2)+1)[2:]
            if(len(carry_check) > 23):
                adj_exp = len(carry_check) - 23
                return [carry_check[1:24], adj_exp]
            else:
                return [bin(int(A[:23],2)+1)[2:].rjust(23,"0"), adj_exp]
```

decode\_bfloat\_16 function: (For checking purposes)

```
def decode_bfloat_16(A):
    A_sign = A[0]
    A_exp = int(A[1:9],2) - 127
    A_frac = A[9:]

    L = list(range(1,24))
    for i in range(len(L)):
        L[i] = 1/2**L[i]

    ans = 0
    for i in range(len(A_frac)):
        if(A_frac[i] == "1"):
            ans += L[i]

    ans += 1
    ans = ans * 2**A_exp

    if(A_sign == "1"):
        ans = ans * -1

    return ans
```

decode\_fp32 function: (For checking purposes)

```
def decode_fp32(A):
    A_sign = A[0]
    A_exp = int(A[1:9], 2) - 127
    A_frac = A[9:]

    L = list(range(1, 32))
    for i in range(len(L)):
        L[i] = 1/2**L[i]

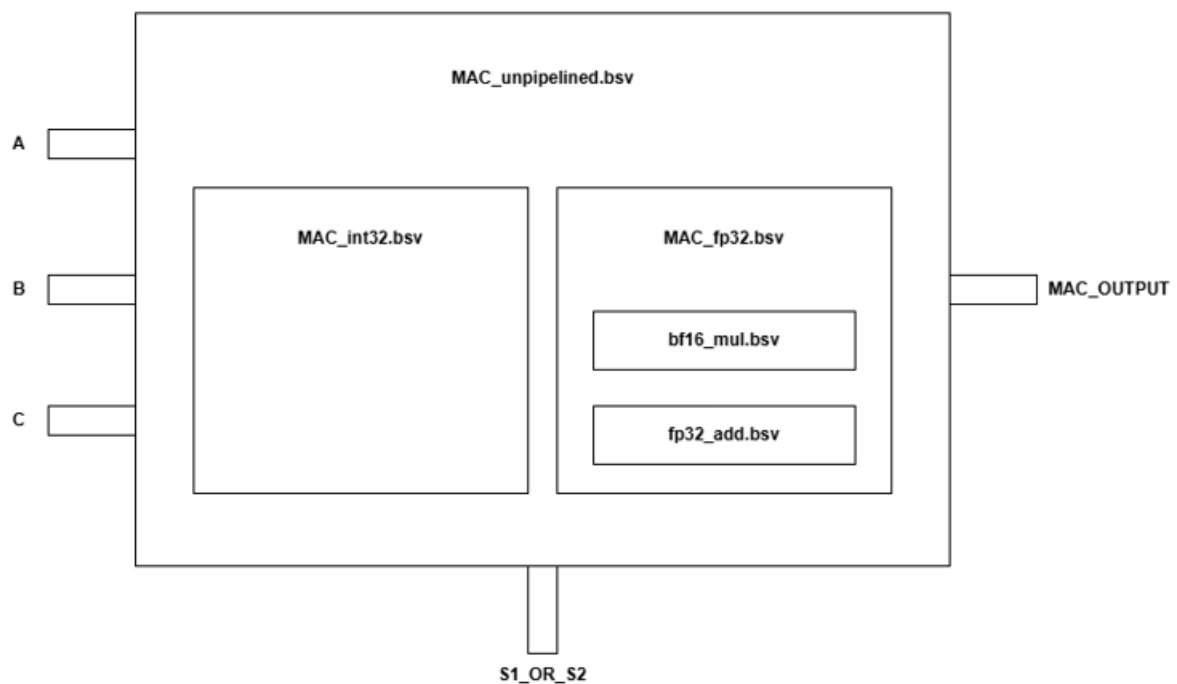
    ans = 0
    for i in range(len(A_frac)):
        if(A_frac[i] == "1"):
            ans += L[i]

    ans += 1
    ans = ans * 2**A_exp

    if(A_sign == "1"):
        ans = ans * -1

    return ans
```

Design architecture:



**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
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#(32) rca_32bit(Bit#(32) ab, Bit#(32) c);
Bit#(32) outp = 0;
Bit#(1) carry = 0;
outp[0] = ab[0] ^ c[0];
carry = ab[0] & c[0];
for(Integer i = 1; i < 32; i = i + 1)
begin
    outp[i] = ab[i] ^ c[i] ^ carry;
    carry = (ab[i] & c[i]) | (ab[i] ^ c[i]) & carry;
end

return outp;
endfunction:rca_32bit
```

The 32-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 bits input A
- 2) 16 bits input B
- 3) 32 bits input C

Gives the following as output:

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

Gives the following as output:

- 1) Output of Bfnum type

Bfnum type:

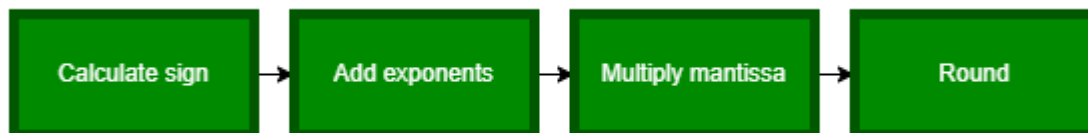
Bfnum is a structure with 1 bit sign, 8 bits exponent and 7 bits mantissa as its members as shown below:

```
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:  $\text{Exp\_A} + \text{Exp\_B} - 127$

When we take 2’s complement of 127 we get:  $\text{Exp\_A} + \text{Exp\_B} + 0b10000001$ , 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  $\text{Exp\_A} + \text{Exp\_B} + 0b10000001$ .

### Multiply\_mantissa:

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

```

rule r1_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

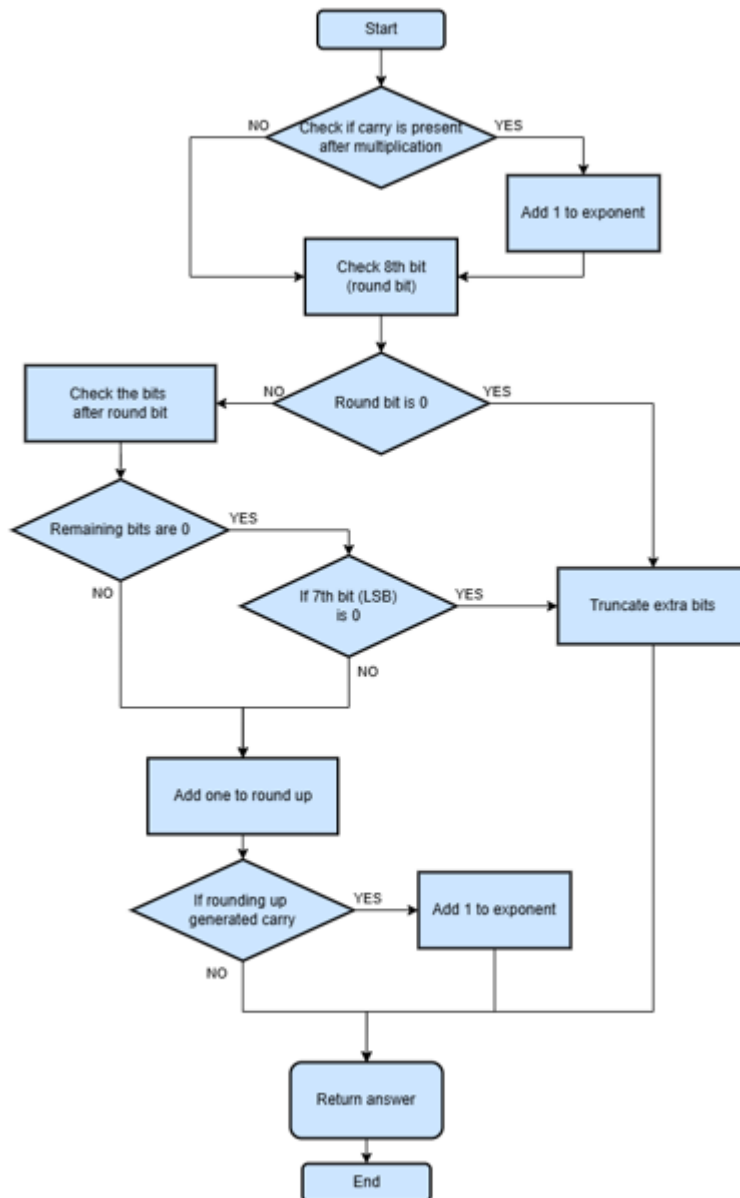
```

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

  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
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 bits input A
- 2) 32 bits 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:

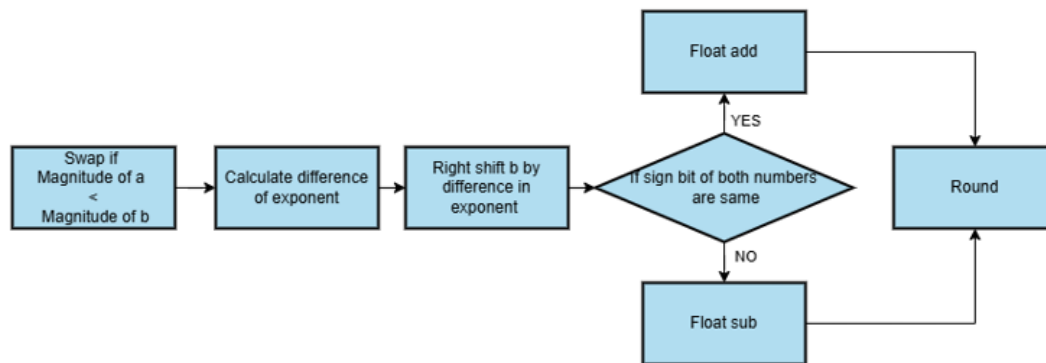
```
typedef struct {  
    Bit#(1) sign;  
    Bit#(8) exponent;  
    Bit#(23) fraction;  
} Fpnum deriving (Bits, Eq);
```

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
  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_bits(fp_a.exponent, twos_compliment(fp_b.exponent));
endrule

```

It is just ripple carry adder with second input fed as twos complement 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 lose 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_done == True);
    do_add <= True;
    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_done == True);
    do_sub <= True;
    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
```

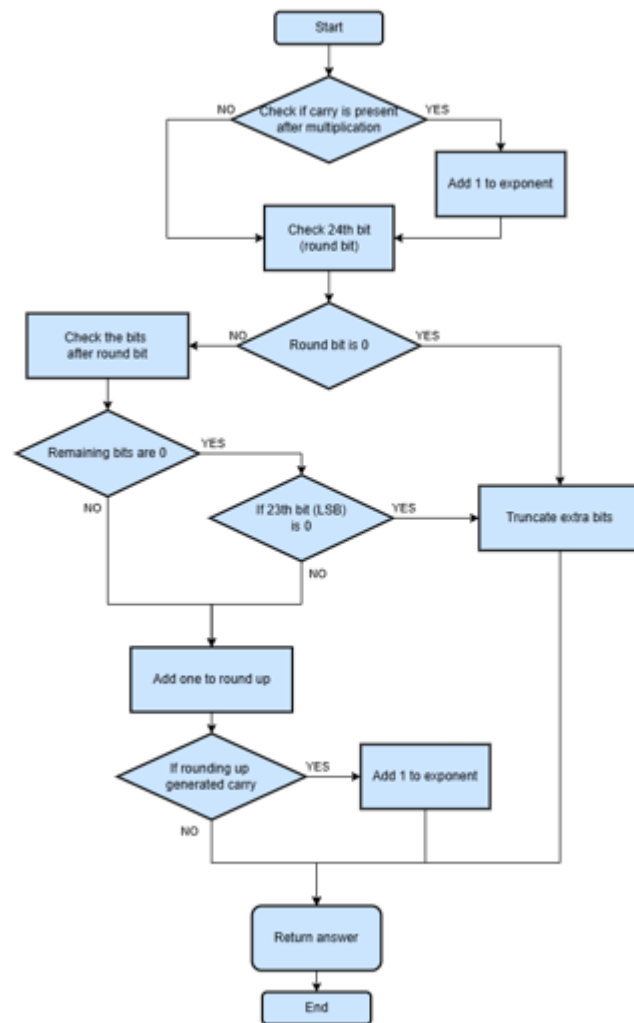
Subtraction using rca:

```
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. Below is the flowchart:



Note that the logic is the same bit the checking of round bit is at 24<sup>th</sup> bit and LSB is 23<sup>rd</sup> 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

(The BSV codes for these functions are too large to paste here and coherently explain, yet they follow the flowchart to the tee)

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: Neagive 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 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)

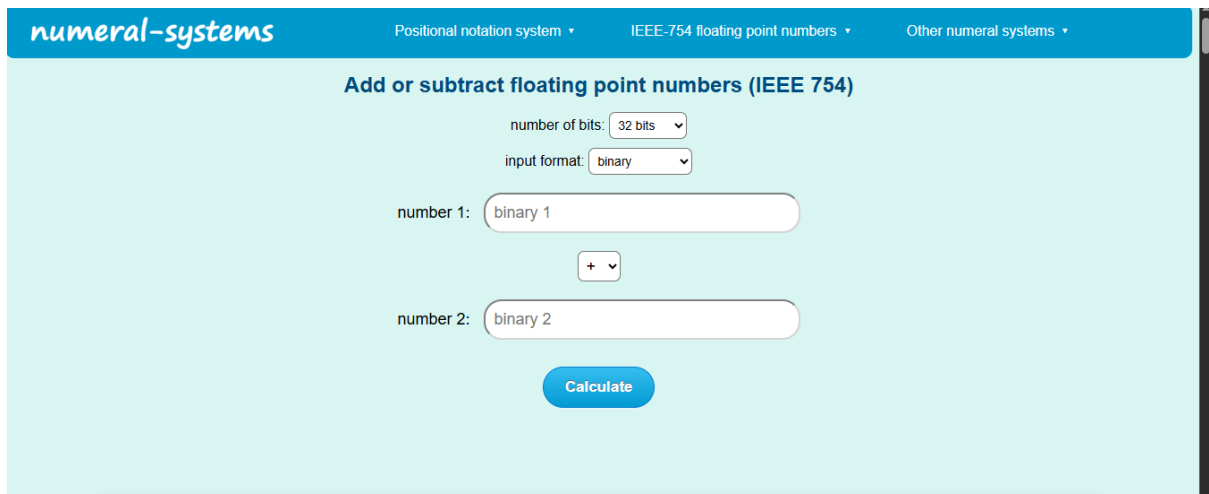
A	B	AB
POSITIVE	POSITIVE	POSITIVE
POSITIVE	NEGATIVE	NEGATIVE
NEGATIVE	POSITIVE	NEGATIVE
NEGATIVE	NEGATIVE	POSITIVE

### 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 calculator:



The screenshot shows the 'numeral-systems' website with a blue header. The main content area is light blue and titled 'Add or subtract floating point numbers (IEEE 754)'. It features two input fields for 'number 1' and 'number 2', both containing the text 'binary 1' and 'binary 2' respectively. Above the input fields are two dropdown menus: 'number of bits' set to '32 bits' and 'input format' set to 'binary'. Between the input fields is a small dropdown menu with a '+' sign. Below the input fields is a blue 'Calculate' button.



## Webscrapping code:

```
1  from selenium import webdriver
2  from selenium.webdriver.common.by import By
3  import time
4
5  from selenium.webdriver.chrome.options import Options
6  chrome_options = Options()
7  chrome_options.add_experimental_option("detach", True)
8
9  web = webdriver.Chrome(options=chrome_options)
10 web.get('https://numeral-systems.com/ieee-754-add/')
11
12 file = open("Padded_negAB_output.txt", "r")
13 AB = file.readlines()
14 file.close()
15
16 file = open("negC_binary.txt", "r")
17 C = file.readlines()
18 file.close()
19
20 file = open("NN_MAC_binary.txt", "w")
21
22
23 Cookies = web.find_element(By.XPATH, '//*[@id="cookie-banner-buttons-container"]/button[2]')
24 input_1 = web.find_element(By.XPATH, '//*[@id="number-input-1"]')
25 input_2 = web.find_element(By.XPATH, '//*[@id="number-input-2"]')
26 Submit = web.find_element(By.XPATH, '//*[@id="submit-button"]')
27
28
29
30
31 Cookies.click()
32
33
34
35 for i in range(len(AB)):
36     input_1.send_keys(AB[i])
37     input_2.send_keys(C[i])
38
39     Submit.click()
40     # time.sleep(5)
41     Output = web.find_element(By.XPATH, '//*[@id="result-path-container"]/div[8]')
42     file.write(Output.text+"\n")
43     print(f"Done {i+1} {Output.text}")
44
45     input_1.clear()
46     input_2.clear()
47
48 file.close()
49 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, alternating ones are tested.

GTKWAVE is used for debugging.

#### **Testbench:**

A testbench is written : test\_mkMAC\_unpipelined.py

It can test the following:

- 1) 9000 given + expanded testcases (float and Int combined)
- 2) Random inputs testing
- 3) Coverage calculations

The assertions are made between both RTL and Reference model and between RTL and given testcases.

## Results:

MAC\_UNPIPELINED\_TEST\_RESULT is log file containing the simulation output as a proof.

```
9346 Inp A: 120 Inp B: 120 Inp C: 268435456 RTL: 268449856 RM: 268449856 TESTCASE 9331
9347 Inp A: 121 Inp B: 121 Inp C: 4294443007 RTL: -509648 RM: -509648 TESTCASE 9332
9348 Inp A: 122 Inp B: 122 Inp C: 2 RTL: 14886 RM: 14886 TESTCASE 9333
9349 Inp A: 123 Inp B: 123 Inp C: 33554432 RTL: 33569561 RM: 33569561 TESTCASE 9334
9350 Inp A: 124 Inp B: 124 Inp C: 268435456 RTL: 268450832 RM: 268450832 TESTCASE 9335
9351 Inp A: 125 Inp B: 125 Inp C: 2147483647 RTL: -2147468024 RM: -2147468024 TESTCASE 9336
9352 Inp A: 126 Inp B: 126 Inp C: 1073741824 RTL: 1073757700 RM: 1073757700 TESTCASE 9337
9353 Inp A: 127 Inp B: 127 Inp C: 32 RTL: 16161 RM: 16161 TESTCASE 9338
9354 2339555000.00ns INFO test_MAC_unpipelined passed
9355 2339555000.00ns INFO *****
9356 ** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
9357 *****
9358 ** test_mkMAC_unpipelined.test_MAC_unpipelined PASS 2339555000.00 24.68 94797631.82 **
9359 *****
9360 ** TESTS=1 PASS=1 FAIL=0 SKIP=0 2339555000.00 28.60 81813153.72 **
9361 *****
9362
9363 - :0: Verilog $finish
9364 make[2]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Unpipelined/MAC_unpipelined'
9365 make[1]: Leaving directory '/home/shakti/Desktop/MAC_Project_ns24z353/CS6230_MAC_Unit_project/Unpipelined/MAC_unpipelined'
```

Refer coverage\_MAC\_unpipelined.yml for coverage report.

The given testcases and expanded testcases passed (including corner cases).

The random inputs fail Float MAC occasionally because of NAN (Not a Number) being provided as inputs.