CAD FOR VLSI

PROJECT 1:

TOPIC: MULTIPLY ACCUMULATE (MAC) UNIT DESIGN

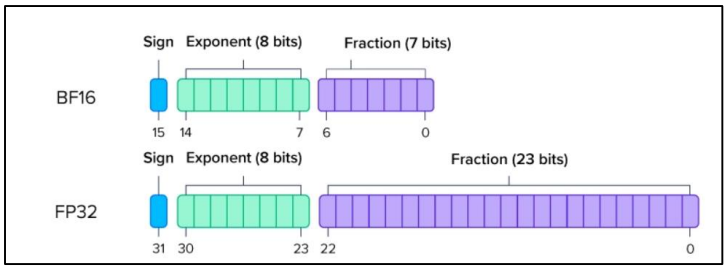
Problem Statement:

Designing a Multiply-Accumulate (MAC) module that supports MAC operations for the following data types:

- (A: int8 , B: int8 , C: int32) -> (MAC: int32)
- (A: bf16, B: bf16, C: fp32) -> (MAC: fp32)

A MAC operation on three numbers A, B and C is defined as: MAC = A*B+C

- int8 : Signed 8-bit integer
- int32: Signed 32-bit integer
- bf16 : Bfloat16 : 1 bit sign, 8 bits exponent, 7 bits mantissa = 16 bits
  BFloat16 number : $(-1)^{sign}$x 1.mantissa x $2^{(exponent+bias)}$
- fp32 : IEEE 754 32-bit floating point : 1 bit sign, 8 bits exponent, 23 bits mantissa = 32 bits
  Fp32 number : $(-1)^{sign}$x 1.mantissa x $2^{(exponent+bias)}$



MAC: INT32

- Unpipelined Int32 MAC Design Microarchitecture:

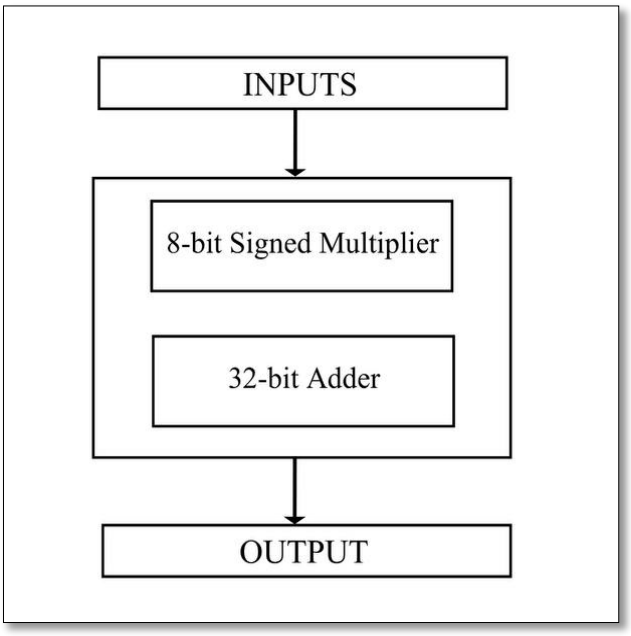


Fig 1. Block diagram of Unpipelined Int32 MAC

The combinational block in the unpipelined Int32 MAC can be divided into two parts:

- **8-bit Signed Multiplier**: Inputs A and B are signed 8-bit inputs. So, to multiply them 8-bit Signed Multiplier is used which is made with Carry Look Ahead adders.

- **32-bit Adder**: The product of A and B is added with C using a 32-bit Adder which is made using 4-bit Carry Look Ahead adders. Note: Carry Look Ahead adders with input bits greater than 4 is not efficient.

- Pipelined Int32 MAC Design Microarchitecture:

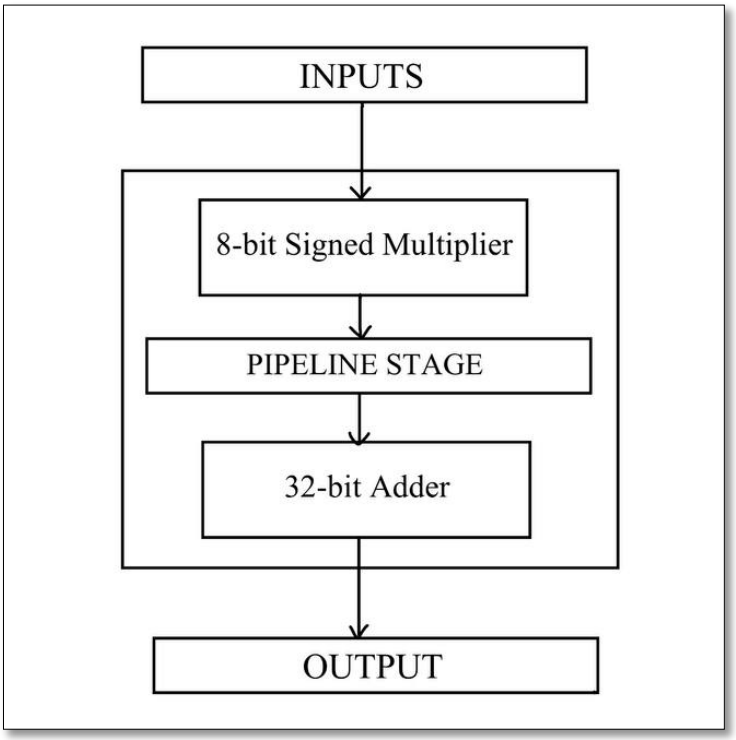


Fig 2. Block diagram of Pipelined Int32 MAC

For the pipelined design:

- Pipeline registers are inserted in between the multiplier and the adder.
- Due to this multiplier and adder blocks can function independently.

MAC: BFLOAT16

- Unpipelined BFloat16 MAC Design Microarchitecture:

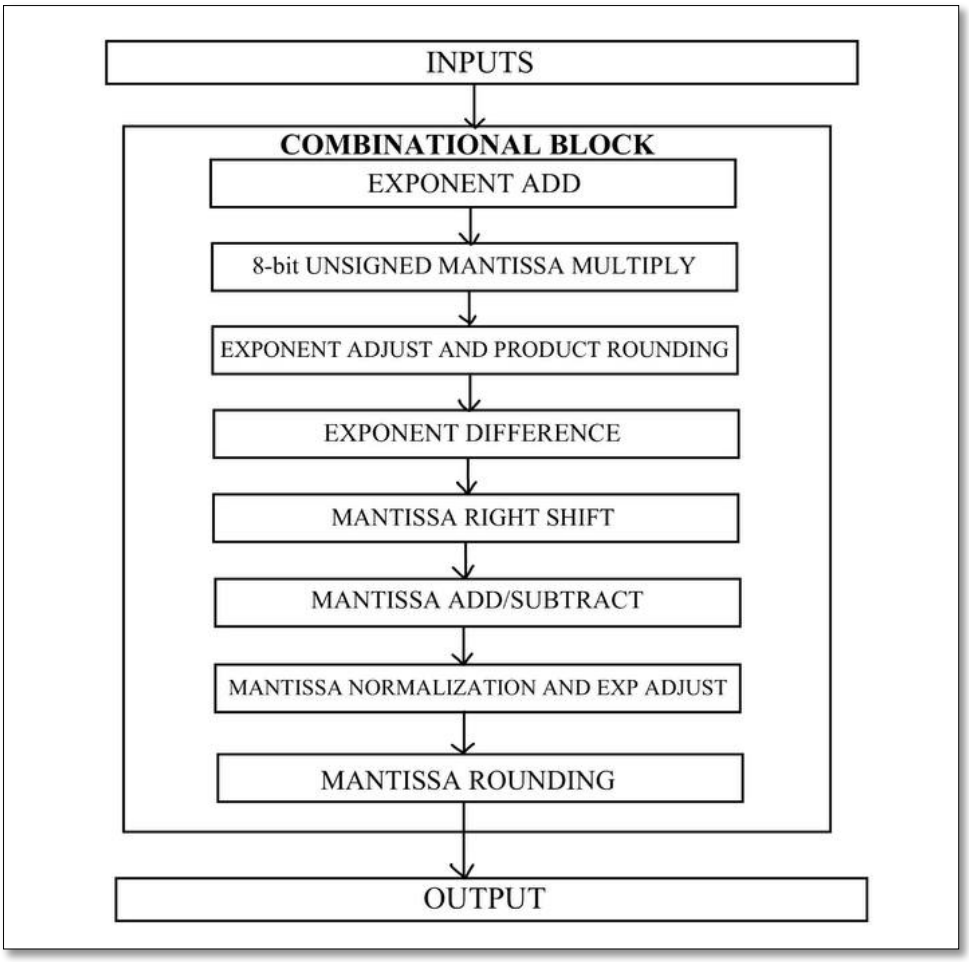


Fig 3. Block diagram of Unpipelined BFloat16 MAC

The combinational block in the unpipelined Int32 MAC executes in a single cycle but we can sub-divide it into the following parts according to the different functions performed inside the block:

- **Exponent Add**: Since, we are multiplying A and B. So their exponents of are added. The exponents of A and B are both biased with a bias value of 127. So when they are added, 127 needs to be subtracted from the sum since:

$$\textbf{Exponent of Product} = (\textbf{Exp\_A} - 127) + (\textbf{Exp\_B} - 127) + 127$$
$$= \textbf{Exp\_A} + \textbf{Exp\_B} - 127$$

- **8-bit unsigned Mantissa Multiply**: The mantissa's of A and B are multiplied using a 8-bit Unsigned Multiplier which is made using 4-bit Carry Look Ahead adders.

- **Exponent Adjust and Product Rounding:** After the multiplication of mantissa's, the result comes in 16 bits of the form {Eg: 10.11111…}. But we need it in the form of {1.011111…}, so we adjust the exponent accordingly is 16^{th} bit of the multiplication result is 1. The sign bit of the product will depend on the sign bit of A and B. If sign_A == sign_B, sign of product is positive else it is negative. Also, the product is Rounded to Nearest Even using the guard, round and sticky bits. The result thus obtained is padded with zero's to make the mantissa as 23 bits as compatible with FP32.

- **Exponent Difference:** Now the product of A and B i.e P(let) is ready to be added with C. So, first we compare the exponents of P and C and store them accordingly for further use. Then we subtract the exponents.

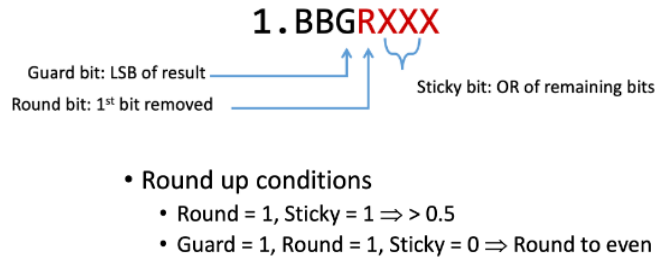
- **Mantissa Right Shift:** The mantissa of the number having the lesser exponent value among P and C is right shifted according to the exponent difference. This helps in aligning the decimal point.

  Eg: $1.110 * 2^3 + 1.101 * 2^1 = (1.110 + 0.01101) * 2^3$

- **Mantissa Add/Subtract:** The mantissa's of P and C are now added or subtracted based on the sign bit. If sign_P == sign_C , we need to add the mantissa's else we need to subtract them. The sign bit of the result will be equal to the sign of the operand having greater exponent.

- **Mantissa Normalization and Exponent Adjust:** After addition, we need to normalize the mantissa of the result to the form {1.mantissa}. For this we found the leading zero count and then left shifted the mantissa with that count value. Also, the exponent was adjusted with the value of shift.

- **Mantissa Round:** In the final step, we round the result using Round to Nearest Even logic.



- Pipelined BFloat16 MAC Design Microarchitecture:

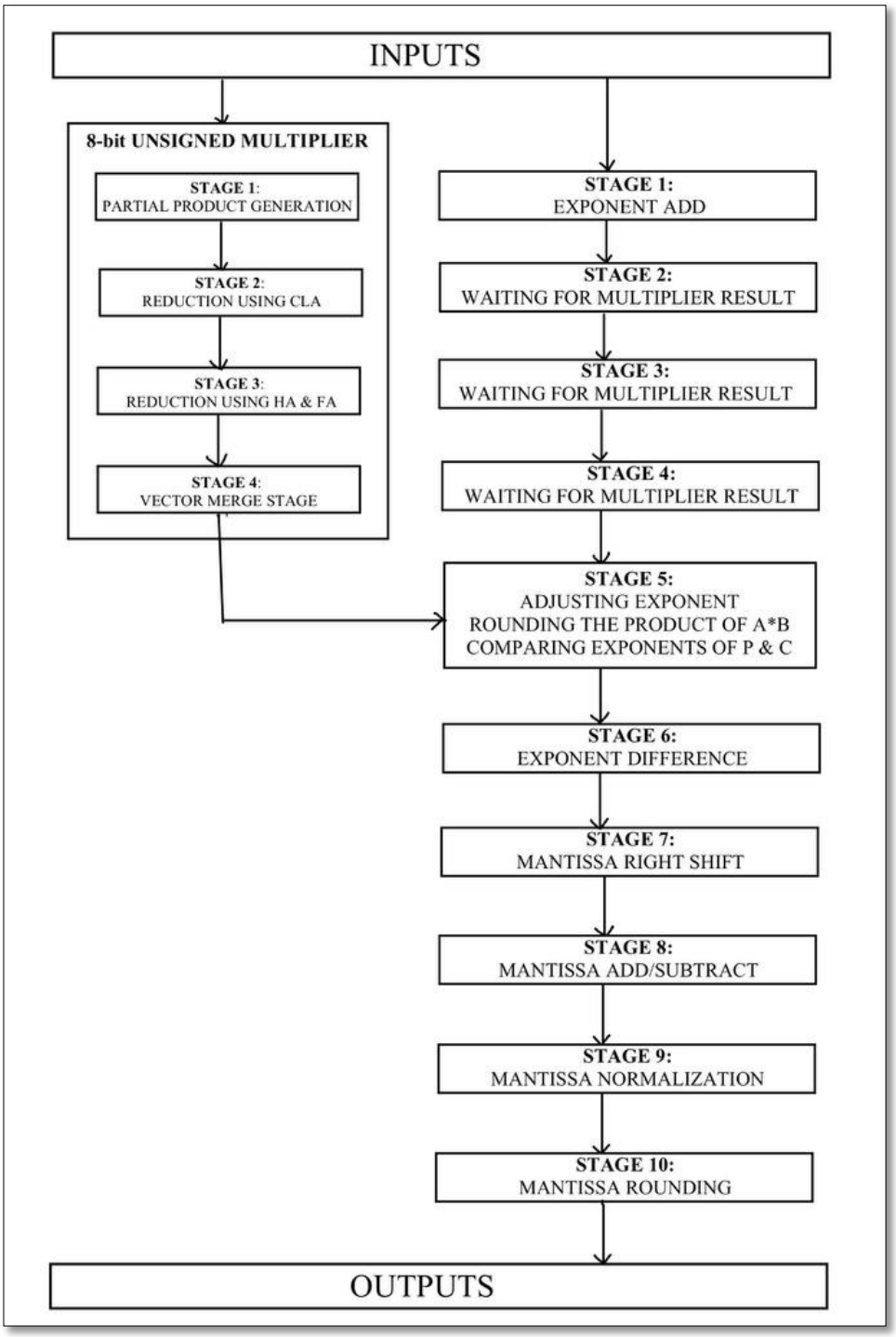


Fig 4. Block diagram of Pipelined BFloat16 MAC

The Pipelined BFloat MAC consists of 10 pipeline stages:

- **STAGE 1**: MULTIPLIER PARTIAL PRODUCT GENERATION AND MAIN MODULE EXPONENT ADDITION
  In the first stage, multiplier partial products are generated in parallel as the exponents of A and B are added. The exponents of A and B are both biased with a bias value of 127. So when they are added, 127 needs to be subtracted from the sum since:

  Exponent of Product = (**Exp_A** – 127) + (**Exp_B** – 127) + 127

- **STAGE 2:** REDUCTION USING CLA IN MULTIPLIER AND WAITING STAGE IN MAIN MODULE
  The mantissa's of A and B are multiplied using a 8-bit Unsigned Multiplier. The first reduction stage in the multiplier uses 4-bit Carry Look Ahead adders.

- **STAGE 3:** REDUCTION USING HA AND FA IN MULTIPLIER AND WAITING STAGE IN MAIN MODULE
  The second reduction stage in the unsigned multiplier uses Half Adders and Full Adders.

- **STAGE 4:** VECTOR MERGE STAGE USING RIPPLE CARRY ADDER IN MULTIPLIER AND WAITING STAGE IN MAIN MODULE
  The final reduction stage in the unsigned multiplier is the vector merge stage and it uses a Ripple Carry Adder.

- **STAGE 5:** EXPONENT ADJUST AND PRODUCT ROUNDING
  After the multiplication of mantissa's, the result comes in 16 bits of the form {Eg: 10.11111…}. But we need it in the form of {1.011111…}, so we adjust the exponent accordingly is 16^{th} bit of the multiplication result is 1. The sign bit of the product will depend on the sign bit of A and B.
  If sign_A == sign_B, sign of product is positive else it is negative. Also, the product is Rounded to Nearest Even using the guard, round and sticky bits. The result thus obtained is padded with zero's to make the mantissa as 23 bits as compatible with FP32.

- **STAGE 6:** EXPONENT DIFFERENCE
  Now the product of A and B i.e P(let) is ready to be added with C. So, first we compare the exponents of P and C and store them accordingly for further use. Then we subtract the exponents.

- **STAGE 7:** MANTISSA RIGHT SHIFT
  The mantissa of the number having the lesser exponent value among P and C is right shifted according to the exponent difference. This helps in aligning the decimal point.
  Eg: $1.110 * 2^3 + 1.101 * 2^1 = (1.110 + 0.01101) * 2^3$

- **STAGE 8:** MANTISSA ADD/SUBTRACT
  The mantissa's of P and C are now added or subtracted based on the sign bit. If sign_P == sign_C , we need to add the mantissa's else we need to subtract them. The sign bit of the result will be equal to the sign of the operand having greater exponent.

- **STAGE 9:** MANTISSA NORMALIZATION AND EXPONENT ADJUST
  After addition, we need to normalize the mantissa of the result to the form {1.mantissa}. For this we found the leading zero count and then left shifted the mantissa with that count value. Also, the exponent was adjusted with the value of shift.

- **STAGE 10:** MANTISSA ROUNDING
  In the final step, we round the result using Round to Nearest Even logic.

INTEGRATED DESIGN

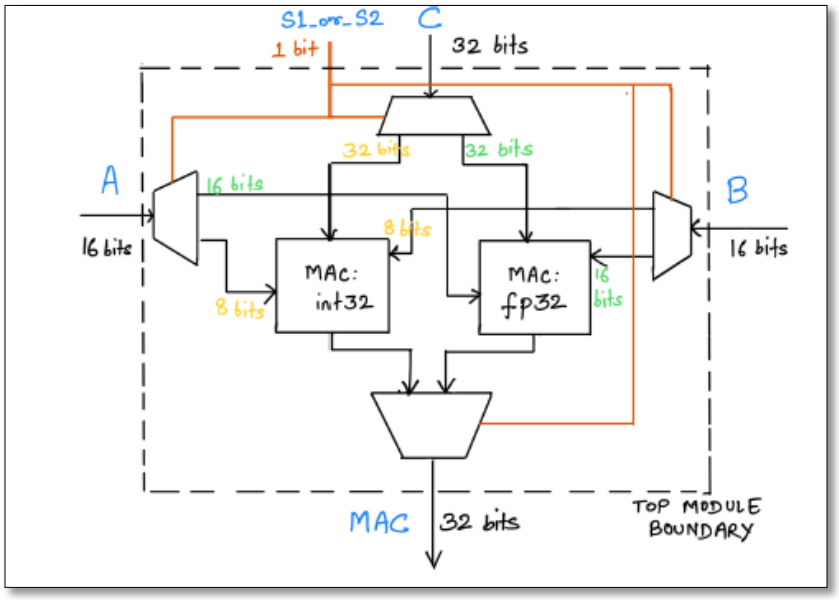


Fig 5. Architecture of the Integrated MAC design

The top level module **mk_mac** has the following interface definition:

- There are five methods in the interface:
  - **put_inp_A** : This method is for 16-bit input a
  - **put_inp_B** : This method is for 16-bit input b
  - **put_inp_C** : This method is for 32-bit input c
  - **put_inp_sel**: This method is for the 1-bit select input which selects the type of MAC operation desired. If sel == 1, the **lower 8-bits** of the A and B inputs along with the 32-bit C input is fed into the Int32 MAC else inputs are fed into the BFloat16 MAC

VERIFICATION STRATEGY:

- There are separate test benches for the pipelined and unpipelined design.
- There are two 16-bit inputs for A and B, one 32 bit input for C and 1 bit select input, which selects the mode of the MAC i.e. whether Int8 MAC has to be calculated or bfloat16 MAC has to be executed.
- Each test bench has two parts:
  - **Part1:**
    In this part A ,B and C are randomly generated and fed to the DUT through cocotb testbench.
    Same inputs are fed to our python model and outputs of both are compared.
    For Bfloat16 mode it will compared both outputs after shifting right by two bits, since liberty of 2 LSB's was given.
    For Int8 mode whole 32 bits are compared for both.

  - **Part2:**
    In this part we read the input files given to us and feed those inputs to DUT, and output from the DUT is compared with the output file given to us.
    All test cases passed for both Int8 and Bfloat16 testcases provided.
    This part of testbench has been commented and can always be uncommented and run for checking functionality.
    Mode=0 is for Bfloat mode and Mode=1 is for Int8 mode for all testbenches.

Python Module:
This module again has 4 inputs, A,B,C,sel. Here also sel=0 means Bfloat mode and sel=1 means Int8 mode.
For Int8 part 16 bit integer inputs for A nd B are converted to 8bit integer values, Then MAC operation is performed and result is again converted into its binary form and given as output.

For Bfloat16 part, 16 bit binary inputs for A and B are converted to Bfloat16 form by using Tensorflow library and similary 32 bit binary input is converted to FP32 format.
Then multiplication of Bfloat16 numbers is performed and it is rounded off to Bfloat16 format. The product is then converted to FP32 format and added to C which is already in FP32 format. The sum is then rounded off and then converted to binary string and given as output.

Coverage part was not checked because we were getting am error “coverage module not found” in the line where we import from cocotb_coverage.coverage.