CS6230 : CAD for VLSI

Project 1: Multiply – Accumulate (MAC) Unit Design

Team Members :-

1. S. Bharath Kumar, CS23M004
2. Abinash Chaudhary, CS23M802

MAC is performed on three numbers A, B and C using MAC = A * B + C. We perform MAC implementation using two selections for different datatypes-

S1 - A(int8), B(int8), C(int32) and MAC(int32)
S2 – A(bf16), B(bf16), C(fp32) and MAC(fp32)

Unpipelined Design:- We use shift and add method for multiplication of two numbers and then add to the third input. For addition, we use ripple carry adder technique as we do not use + and * operators.

In our code, we have named the package name as *mac* which has to be same as filename. We have interface *MACInterface* with predefined methods for A, B, C, and S1_or_S2. For selection of S1 or S2, we use bool mode. False(0) for S1 and True(1) for S2. We create a module *mkMAC* to implement the interface and setup registers for storing the inputs A,B,C, selecting S1 or S2 and storing MAC results.

When we select S1 (False) – `function Bit#(32) ripple_add(Bit#(32) x, Bit#(32) y);` is a function to implement ripple carry adder without using + operator. We perform AND operation for carry and XOR operation for Sum. The for loop iterates bitwise over x and to do addition with carry. `function Bit#(32) shift_and_add_mul(Bit#(8) a, Bit#(8) b);` implements multiplication of two 8-bit numbers using shift and add method without using * operator. This function will shift a to the left and accumulates it to the product whenever the corresponding bit in b is 1.The product will not be accumulated when b is 0. We use these two methods to implement MAC when selecting S1 datatype. method `ActionValue #(Bit#(32)) start_MAC();` does the MAC calculation for the selected mode, here considering S1 mode, `shift_and_add_mul` for multiplying A and B and then `ripple_add` for adding it to C. method `Bit#(32) get_MAC_result();` returns the MAC result in 32-bit format (`mac_result_S1` for S1 mode).

When we select S2 (True) – For S2multiplication, we first convert bf16 to fp32 format by extending the mantissa without altering the sign and exponent part. Conversion is done using the function `function Bit#(32) bf16_to_fp32(Bit#(16) bf16);` After extending bf16 inputs A and B to fp32 format, the 1st bit is sign, next 8 bits for exponent and extend the mantissa to 23 bits by appending zeroes.
We use function `Bit#(32) multiply_bf16(Bit#(16) a_bf16, Bit#(16) b_bf16);` to perform multiplication of two bf16 without using * operator. This multiply function multiplies two bf16 numbers in 32-bit . The product's sign will be the XOR of the signs of A and B. Next, we perform exponent addition by adding the exponents of A and B and then adjusting it with bias (8d'127- 8 bit of 127 decimal). We do the mantissa

multiplication using bitwise shift-and-add method. We multiply mant_a and mant_b with a leading 1 including which it represents a 24-bitnormalised mantissa. The for loop iterates over each bit of mant_b and when each time, mant_b[i]=1, temp_A is left shifted and XORed with mant_product and accumulates the result. The final result of mantissa multiplication will be 48-bit mant_product. We consider it as normalised, if its highest bit mant_product[1] is 1(i.e.) mant_product is right -shifted by 1 bit and the exponent is incremented by 1 bit. Then, the final mantissa is assigned to final_mant and exp to final_exp. Then, after multiplication we need to add the result to C, for that we use `bitwise_add` to add the multiplication result to 32-bit C. Similar to S1 ripple carry adder, here also we XOR a and b bitwise and a carry is done using AND, OR operations. After obtaining the final sum, we check round bit (22) and extra bit (23), if both are 1, we round up the result by incrementing it by 1. The final result is thus stored in mac_result_S2 (32-bit format).

Verification – The function `apply_inputs` takes dut, and inputs a,b,c that are applied to the signals in the MAC module. It assigns values to dut.a, dut.b and dut.c and then waits for a rising edge of the clock dut.clk thus will sync the input and the clock. `mac_random_test` is used for randomized testing of the MAC design. `input_a_path, input_b_path, input_c_path and output_path` specifies paths to files containing test vectors and expected results required for our verification of MAC results. Every input file will be read and stored as list of integers (inputs_a, inputs_b, inputs_c) for a, b c in MAC calculation. The expected output values are stored in `expected_outputs` for comparison. We can reset the MAC design by setting dut.reset to high (1), holding it for 10ns and then resetting to low (0) and again wait for 10ns. num_tests will tell the number of randomized tests that needs to be performed.
Test loop – For testing, a random index will be chosen to select the test vectors from inputs_a, inputs_b, inputs_c and expected_outputs. The function `apply_inputs` is called to load inputs to MAC design. Then, after 10ns, the output will be received from dut.output.value.integer . The result with the expected_output, if it matches, it is passed else error will be thrown.