CS6230 : Project 1 - MAC Design

Pushpa Chaudhary ee21b109
Fardeen Razif ee21b046

28 October 2024

1 BSV Code

- Overall, the code consists of 2 top modules, 2 cocotb testbenches and each top module works for both data types
- One top module implements pipelining while the other doesn't

1.1 Unpipelined Design

- Consists of the **five** methods as specified in the design and **four** functions for the computations
- When S = 1, A and B are assumed to be in **int8** and C in **int32** format
- When S = 0, A and B are assumed to be in **bf16** and C in **fp32** format
- For Integer Addition, we have defined a 32-bit "ripple_carry_adder" function
- For Integer Multiplication, the "mult" function
  - Performs multiplication by adding and shifting the result in a register
  - The addition is done using the previously defined "ripple_carry_adder" function
- For Floating Point Addition, the "float_add" function
  - Aligns the mantissa of the smaller exponent
  - Adds or subtracts based on the sign using the "ripple_carry_adder" function
  - Normalizes the final result if required and then rounds off the answer
- For Floating Point Multiplication, the "custom_mult" function
  - Multiplies the Mantissas using the same shift and add approach
  - We then round off the Result of the product and adjust the exponent
  - We combine the sign, normalized mantissa and adjusted exponent to get the final product

1.2 Pipelined Design

- We have implemented a 2 stage pipeline design where the first stage performs the multiplication and the second stage performs addition
- FIFOs used in the testbench
  - **a_ififo** : Used to store the input A
  - **b_ififo** : Used to store the input B
  - **c_ififo** : Used to store the input C
  - **tcs_ififo** : Used to store the input S
  - **product_ofifo** : Used to store the output of the Multiplication Stage
  - **result_ofifo** : Used to store the output of the Addition Stage
- We also define 2 rules for performing the computations
- The multiplication rule
  - Stores the inputs from "a_ififo", "b_ififo" and "tcs_ififo" into **Bit** structures
  - Calls the appropriate multiplication function
  - Dequeues "a_ififo", "b_ififo"
  - Enqueues "product_ofifo", "c_ififo"
- The addition rule
  - Stores the results of the previous multiplication operation from "product_ofifo", "c_ififo" and "tcs_ififo" into **Bit** structures
  - Calls the appropriate addition function
  - Dequeues "product_ofifo", "c_ififo", "tcs_ififo"
  - Enqueues "result_ofifo"

2 Cocotb Testbench

2.1 For Unipelined Design

- **IMPORTANT:** The S value is given as input to in testbench and must be changed to run a particular data type
- Specifically, this line in the code **dut.s1_or_s2_tcs.value = 0;** determines whether "int" inputs are considered or "float" inputs are considered
- In this testbench, it takes us **3 clock cycles** to get the output after giving the input
- We first initialize the MAC DUT and start a 10 microsecond period clock for the simulation
- It then loops over the A, B, C values setting each on the respective DUT input port and captures the DUT's output port after processing it
- We execute an assert statement at the end to verify if the simulation passed all the test cases

2.2 For Pipelined Design

- Unlike the previous case, here we use the first 4 cycles to load the inputs before the main processing loop
- This staging phase is done to ensure the input values of the first test case is initialized properly
- After the initial setup, in every clock cycle, a new set of inputs are given to the DUT to evaluate and outputs the results corresponding to the previous input
- Everything else remains the same

3 Coverage

- We do a walking 1 pattern and walking 0 pattern tests
- A walking 1 pattern is applied to the inputs A and B, where only one bit is set to 1 at a time across all possible bits for 8-bit values, while varying the 32-bit input C
- For each combination of inputs, the following steps are executed:
  - The values are assigned to the DUT inputs.
  - A series of clock cycles (RisingEdge) is waited on to allow the DUT to process the inputs.
  - The DUT's output is logged and compared against the expected output and an assertion checks for output correctness
- The walking 0 pattern test is similar to the above test
- The coverage data is exported to a YAML file