CS6230
Project 1: Multiply-Accumulate (MAC) Unit Design

JADAPALLI ASHOK KUMAR -EE24M094 BONDUGULA PRANAV-EE24M088

Project Description:

A MAC operation, performed on 3 numbers A, B, and C, is formulated as shown below.

- MAC = A*B + C

In this project, designing a Multiply-Accumulate (MAC) module that supports the MAC operations for the following data types.

- S1. (A: int8 , B: int8 , C: int32) -> (MAC: int32)
- S2. (A: bf16, B: bf16, C: fp32) -> (MAC: fp32)

Project Overview:

This project involves designing a versatile Multiply-Accumulate (MAC) unit capable of handling MAC operations for both integr and floating-point data types. The MAC operation which calculates MAC = A*B+C, is implemented for two specific data types:

- Integer MAC operation: (A:int8, B:int8, C:int32) -> (MAC: int32)
- Floating-point MAC operation using bf16 inputs and fp32 accumulation: (A: bf16, B: bf16, C: fp32) -> (MAC: fp32)

The design has a modular architecture, with three key modules

1. mkIntMac: Handles integer MAC operations
2. mkbf16Mac: Performs MAC operations for bf16 inputs and returns fp32 result
3. mkMacUnitTop: A top-level module that selects the appropriate MAC operation (integer or floationg point) based on a control signal s1_or_s2

This design also includes an interface, MacUnit_Ifc, that provides methods for setting input values and retrieving the result of the MAC computation.

Software Architecture:

The MAC unit composed of three main modules:

1. Top Module (mkMacUnitTop): This module determines whether to use integer or floating-point MAC operation based on the selection input s1_or_s2. It forwards inputs to the selected MAC module and retrieves results from it.
2. Integer MAC Module (mkIntMac):
   - This module performs MAC operations on integer values.
   - It includes helper functions for:
     - ✓ Bitwise addition for 8-bit and 32-bit integers.
     - ✓ Multiplication of 8-bit and 32-bit integers.
   - The result of the integer MAC operation, computed as MAC = A*B+C, is stored in a register (result).
3. Floating-Point MAC Module (mkbf16Mac):
   - This module handles MAC operations for bf16(bfloat16) values as inputs, returning the result as an fp32 value.
   - Key helper functions include:
     - ✓ Conversion from bf16 to fp32 format.
     - ✓ A 24-bit multiplication function to multiply mantissas of bf16 numbers.
     - ✓Fp32 addition and multiplication functions.
   - The module uses these functions to calculate the final fp32 MAC result, stored in the result register.

CS6230
Project 1: Multiply-Accumulate (MAC) Unit Design

JADAPALLI ASHOK KUMAR -EE24M094 BONDUGULA PRANAV-EE24M088

Interface (MacUnit_Ifc)

The interface defines five methods for managing the MAC unit:

- load_A(Bit#(16) a): Loads input A.
- load_A(Bit#(16) b): Loads input B.
- load_A(Bit#(16) c): Loads input C.
- load_s1_or_s2(Bit#(1) sel): Selects between integer and bf16 MAC operations(0 for integer, 1 for floating-point).
- Get_MAC(): Returns the result of the MAC operation.

Code Explanation:

The following sections provide a summary of each key component in the provided BSV code.

Integer MAC Module (mkIntMac)

This module supports integr MAC operations:

1. Registers:
   - reg_A, reg_B (8-bit) for the MAC operands A and B.
   - reg_C, result(32-bit) for operand C and the MAC result, respectively.
2. Addition Functions:
   - Int32_Addition and int8_Addition: Functions to add 32-bit and 8-bit integers bitwise, considering carry propagation.
3. Multiplication:
   - int_Multiplication: Computes the product of two signed 8-bit integers, extending to 32 bits and handling sign correction.
4. Rule for MAC operation:
   - rl_Compute_Int_Mac: Multiplies reg_A and reg_B, adds reg_C, and stores the result in result.

Floating-Point MAC Module (mkbf16Mac)

The bf16 MAC module performs MAC calculations in floating-point format:

1. Registers:
   - reg_A, reg_B (16-bit) for the bf16 inputs.
   - reg_C, result(32-bit) for the fp32 operand C and the MAC result, respectively.
2. Conversion and Arithmetic Functions:
   - bf16_to_fp32: Converts bf16 to fp32.
   - multiplication_fp32: Multiplies two fp32 numbers.
   - fp32_Addition: Adds two fp32 numbers, handling alignment and rounding
3. Rule for MAC operation:
   - Rl_compute_bf16_mac: Converts reg_A and reg_B from bf16 to fp32, performs MAC operation, and stores the result in result.

CS6230
Project 1: Multiply-Accumulate (MAC) Unit Design

JADAPALLI ASHOK KUMAR -EE24M094 BONDUGULA PRANAV-EE24M088

Top Module (mkMacUnitTop)

The top-level module manages the selection of the MAC operation type based on the s1_or_s2 signal.

1. Selection Logic:
    - The rl_select_mac_output rule checks the value of reg_s1_or_s2.
    - If reg_s1_or_s2 is 0, int_Mac (integer MAC) is used; if 1, bf16_Mac(floating-point MAC) is used.
    - The result is assigned to result.
2. Method Implementations:;
    - Load_A, load_B, load_C: Pass inputs to the selected MAC module.
    - Load_s1_or_s2: Sets reg_s1_or_s2 to choose between integer and floating-point MAC modules.
    - Get_MAC: Retrieves the result from the selected MAC module.

Conclusion:

This MAC unit design offers robust support for integer and floating-point MAC operations, effectively handling different data types with a unified interface. The modular architecture promotes extensibility, allowing additional data types to be integrated in future implementations.

CS6230
Project 1: Multiply-Accumulate (MAC) Unit Design

JADAPALLI ASHOK KUMAR -EE24M094 BONDUGULA PRANAV-EE24M088

Verification of MAC Unit:

Tools used: Cocotb(Python-based Testbench)

Test Environment and Setup:

Testing was performed using the Cocotb framework, which allows python to drive HDL simulations. This provides flexibility in writing complex test scenarios, checking outputs, and logging detailed results. The following setup was used.

1. Clock Signal: A 10us period clock, toggling asynchronously.
2. Reset Signal: Initial reset applied at start-up.
3. Enable Signals:
   - ✓ EN_load_A, EN_load_B, EN_load_C to control loading of values for A,B, and C.
   - ✓ EN_load_s1_or_s2 to select between integer and floating-point operations.
4. Tolerance for Floating-Point Operations: Due to floating-point precision limitations, a tolerance of ±2 LSBs was allowed.

Test Case Details:

Two primary test cases were implemented:

1. Test Case 1: Integer MAC Operation
   - Objective: Validate the MAC operation for integer inputs.
   - Test Inputs: Binary values for A (int8), B (int8) and C (int32) read from test files.
   - Expected Outputs: Pre-computed MAC results in binary format.
   - Process:
     - ✓ Apply each set of inputs, wait for multiple clock cycles for data propagation, and then capture the output.
     - ✓Compare the output from the DUT(Design Under Test) to the expected MAC value.
     - ✓Log any discrepencies.
   - Results: All integer test cases matched expected results within the error bounds.
2. Test Case 2: Floating-Point MAC Operation
   - Objective: Validate the MAC operation bf16 and fp32 floating-point inputs.
   - Test Inputs: Binary values for A (bf16), B (bf16) and C (fp32) read from test files.
   - Expected Outputs: Pre-computed MAC results in binary format.
   - Process:
     - ✓ Similar to the integer test, with an additional slicing step to compare bf16 MAC output.
     - ✓Tolerance of 2 LSBs was permitted due to rounding and precision limitations.
   - Results: Floating-point test cases also matched expected results within the specified tolerance.

CS6230
Project 1: Multiply-Accumulate (MAC) Unit Design

JADAPALLI ASHOK KUMAR -EE24M094 BONDUGULA PRANAV-EE24M088

Verification Methodology:

The verification methodology included:

1. Binary File Input Parsing: Custom Python functions were created to read binary test cases, parse them as BinaryValue objects, and ensure they matched the MAC unit's requirements.
2. Clock and Reset Handling: A common clock setup function managed consistent timing and reset conditions across all tests, ensuring synchronous dta loading and resetting.
3. Output Verification and Assertions: Results from the DUT were directly compared against expected values using assert statements. Any discrepancies triggered detailed error messages logging expected vs. actual results.
4. Model Validation: A reference model (model_macUnit) served as the golden model, ensuring any DUT output deviations were identified during verification.

Results and observations:

Integer MAC tests:

- Number of Test Cases: 1000
- Pass Rate: 100%
- Observations: All test cases matched exacly, with no observed deviations.

Floating-Point MAC Tests:

- Number of Test Cases:1000
- Pass Rate: 100%
- Observations: Minor variances ±2 LSB tolerance were observed, as expected, due to the bf16 rounding behaviour.

CS6230
Project 1: Multiply-Accumulate (MAC) Unit Design

JADAPALLI ASHOK KUMAR -EE24M094 BONDUGULA PRANAV-EE24M088

Verification Test Result: