MAC Module Design and Verification Report

Name : Deepak S (MM22B011)

October 29, 2024

1 Introduction

This report details the design and verification of a Multiply-Accumulate (MAC) module developed using Bluespec System Verilog (BSV) for deep learning computations. The MAC module supports two data formats: `int8` and `bf16`. The verification framework is built with **cocotb**, a coroutine-based cosimulation tool in Python, used to simulate and validate the functionality of the MAC operations. Verification utilizes test data generated by an NVIDIA GPU, provided in binary format.

2 Microarchitecture and Function Descriptions

The MAC module design incorporates functions for two primary configurations:

- **Unpipelined MAC Operation**: This configuration executes operations sequentially in a single cycle, ideal for scenarios where latency is prioritized over throughput.
- **Pipelined MAC Operation**: This design leverages a pipelined architecture, increasing throughput by allowing operations to overlap across multiple cycles.

2.1 Functional Units and Their Roles

Several core functions are implemented to support both configurations, managing data types, arithmetic operations, and internal state maintenance. Key functions and their roles are explained below.

2.1.1 Function `convert_int8_to_int32(A, B)`

This function takes two 8-bit signed integer inputs, `A` and `B`, and performs type extension and multiplication. It returns a 32-bit integer result, representing the product of the two inputs. This function is particularly essential for S1 format operations:

$$\texttt{Result\_int32} = \texttt{A} \times \texttt{B}$$

In int8 operations, this function supports data alignment, allowing a seamless transition to the accumulation phase.

2.1.2 Function convert_bf16_to_fp32(A, B)

For bfloat16 operations, this function performs type extension, converting two bf16 operands into fp32. First, it multiplies A and B with rounding to the nearest value and pads the mantissa to 23 bits to match fp32:

$$\texttt{Result\_fp32} = \texttt{Round}(\texttt{A} \times \texttt{B})$$

This result is then used in subsequent accumulation with fp32 values.

2.1.3 Function accumulate_result(C, Product)

The accumulate_result function manages the core accumulation phase for both int8 and bf16 operations, adding the multiplication product (either int32 or fp32) to the 32-bit operand C. In bfloat16 operations, the function uses a type-safe addition operation to maintain precision in the result:

$$\texttt{MAC\_output} = \texttt{C} + \texttt{Product}$$

This function is optimized to handle any overflow that may occur due to the addition operation, ensuring accuracy.

2.1.4 Pipeline-Specific Functions

In the pipelined MAC module, additional helper functions manage data staging across multiple pipeline registers. These functions are responsible for latching intermediate products into pipeline registers, maintaining synchronized data flow across stages, and updating the final MAC output upon completion of all stages.

3 Verification Methodology

3.1 Verification Framework

The cocotb framework provides a robust and high-level Python API for generating test stimuli and validating MAC outputs against expected results. The verification utilizes NVIDIA-provided test cases, with inputs formatted as binary files across both data formats, int8 and bf16.

3.2 Verification Data Files

Each format-specific verification data set includes 1000 test cases, as detailed below:

- **S1 (int8) Format**

  - `A_binary.txt` and `B_binary.txt`: Contain 8-bit signed integer values.
  - `C_binary.txt`: Contains 32-bit signed integer values.
  - `MAC_binary.txt`: Contains the 32-bit expected output values from the MAC operation.

- **S2 (bf16) Format**
  - `A_binary.txt` and `B_binary.txt`: Contain `bf16` values.
  - `C_binary.txt`: Contains `fp32` values.
  - `MAC_binary.txt`: Contains the expected `fp32` output values from the MAC operation.

3.3 Verification Procedure for S2 (bf16) Format

The `bf16` verification workflow includes additional precision handling steps:

1. **Multiplication and Rounding**: Multiply operands `A` and `B`, round the product, and extend the mantissa to convert it to `fp32`.
2. **Accumulation**: Add the `fp32` result to operand `C`, producing the final MAC result.

Each computed MAC output is then compared against `MAC_binary.txt`, with a tolerance of 2 Least Significant Bits (LSBs) for accuracy.

3.4 Pipeline Verification (Pipelined MAC Module)

The pipelined module undergoes verification to ensure each stage processes values accurately while maintaining correct data flow. This involves checking:

- The correct latching of intermediate results between stages.
- Synchronization of inputs and outputs across pipeline stages.
- Throughput validation by verifying consistent MAC results over consecutive cycles.

4 Results and Observations

The verification results demonstrate accurate operation for both MAC configurations:

- **Accuracy**: Both modules met the expected accuracy, passing 1000 test cases per format with a 2 LSB tolerance.
- **Performance**: The pipelined module achieved higher throughput than the unpipelined module, with no data loss or output discrepancies across pipelined stages.

- **Edge Case Handling**: The MAC module handled all edge cases effectively, including zero, maximum, and minimum input values for each format.

5 Conclusion

The MAC module demonstrates robust performance and accuracy across both configurations and supported data formats. The cocotb framework proved effective in validating functional and edge-case scenarios. This verification methodology ensures the MAC module is reliable for high-performance deep learning computations.

6 GitHub Repository

For full project source code and documentation, refer to our GitHub repository at:

[https://github.com/mm22b011-deepaks/DL_Accelerator_HDL.git]