CS6230 - CAD for VLSI: Project 2

Systolic Array Design

Karthikeyan R CS23S039
Arun Prabakhar CS23D012

Introduction

The report outlines the implementation of a 4×4 systolic array for matrix multiplication, designed using Bluespec SystemVerilog (BSV). The project builds on the MAC module developed in Assignment 1, focusing on architectural organization, functional methods, and verification methodologies. The design supports int8 and bfloat16 data types and is verified using the cocotb framework.

Architectural Details

1. Systolic Array

- The systolic array consists of 16 MAC units (4 rows × 4 columns) connected in a grid.
- Data flows:
    1. Horizontally: Matrix A values propagate row-wise across columns.
    2. Vertically: Matrix B values propagate column-wise down rows.
- Each MAC unit performs:
    1. Multiplication of inputs (a,b).
    2. Accumulation with a partial sum (c).
- The design uses FIFOs and registers to manage data flow and ensure synchronization.

2. Top-Level Module: mkSystolic

The mkSystolic module implements the core systolic array using the Ifc_systolic interface. It manages data propagation, computation, and output collection for matrix multiplication.

- Creates a 4×4 grid of MAC (Multiply-Accumulate) units, where each element is an instance of Ifc_mac_systolic.
- Each MAC unit operates independently but is connected to its neighbors for data flow.
- Matrix Registers:
    - a_matrix: Stores one row of the A matrix.
    - b_matrix: Stores one column of the B matrix.

Key Functionalities

1. Matrix Initialization

The module allows the user to input the A and B matrices and initializes their respective values in the systolic array:

- set_A_matrix: Sends the row of the A matrix to the first column of the systolic array.
- set_B_matrix: Sends the column of the B matrix to the first row of the systolic array.
- set_C_row1: Initializes the first row of partial sums (C) with a constant value.
- initialise_systolic: Resets the entire systolic array, setting C values to zero and clearing A values in all columns except the first.

These methods ensure that the systolic array starts in a clean state and has the correct initial inputs.

2. Data Propagation

The systolic array relies on a pipelined data flow where:

- A Matrix:
    - Propagates horizontally across rows.
    - Rules like shift_a_column1, shift_a_column2, and shift_a_column3 ensure A values move step-by-step from one column to the next.
- B Matrix:
    - Propagates vertically across columns.
    - Rules like shift_b_row1, shift_b_row2, and shift_b_row3 transfer B values from one row to the next.

- Partial Results (C):
    - Intermediate results flow vertically down columns.
    - Rules like shift_c_row1, shift_c_row2, and shift_c_row3 ensure that computed partial sums are passed to the next row for accumulation.

These propagation rules ensure that data flows synchronously through the systolic array, enabling matrix multiplication.

3. Computation Control

The systolic array is controlled using the following mechanisms:

- State Control (set_state_systolic):
    - Enables or disables the computation of the systolic array by setting a global state for all MAC units.
- Cycle Counter (count_up):
    - Tracks the current cycle and allows timed execution of rules for precise control over data movement.

These controls ensure the systolic array executes operations in a synchronized and predictable manner.

4. Result Collection

Once the matrix multiplication is complete, the module collects the results:

- get_result: Retrieves the final computed values from the last row of the systolic array.
    - Each column in the last row contains one element of the resulting matrix.
- The results are returned as a 4×1 vector, representing the final outputs after all computations.

2. **Module:** mkTopSystolic

The mkTopSystolic module is the top-level controller for a systolic array designed to perform matrix multiplication. It manages the data flow between input matrices, intermediate computations, and the retrieval of results.

Workflow of the Module

1. **Initialization**:
    - The input matrices A, B, and C are loaded into their respective registers.
    - The mode of operation is set using a control signal.
2. **Data Propagation**:
    - Rows of B are sent into the systolic array sequentially from top to bottom.
    - Columns of A are sent into the array in a staggered manner from left to right.
    - Dummy data is injected after all valid inputs to allow the systolic array to flush and complete its computations.
3. **Computation**:
    - The systolic array processes the inputs, performing multiply-accumulate operations in a pipelined manner.
4. **Result Collection**:
    - Computed results are retrieved from the systolic array row by row and stored in the C matrix.
    - The staggered results are mapped to a standard 4×4 matrix format for final output.
5. **Output**:
    - The module provides the computed 4×4 matrix multiplication result to the external interface.

VERIFICATION

We have created a Python script that implements a verification framework for testing the functionality of systolic array hardware using the cocotb framework. It focuses on verifying the operations of Multiply-Accumulate (MAC) modules and systolic arrays by generating exhaustive input patterns and comparing results with a reference model.

Core Objectives

1. **Functional Verification**:
    - Validate the MAC operations (multiplication and accumulation) in **int8** and **bfloat16** data formats.
    - Verify the systolic array's matrix multiplication for correctness and edge cases.
2. **Coverage-Driven Testing**:
    - Ensure all input combinations, including corner cases, are tested systematically.
    - Use coverage points and cross-coverage metrics for comprehensive analysis.

Key Components

1. Test Pattern Generation

To exercise a wide range of inputs, the framework defines several test pattern generators and systematically combines them into test cases.

- **Test Patterns**:
    - Walking ones and zeros.
    - Alternating ones and zeros.
    - Powers of two minus one.
    - Sliding patterns (zeros and ones).
    - Random binary patterns.
- **Coverage Variables**:
    - Sign bits: '1' (negative) and '0' (positive).
    - Exponent patterns: Includes special cases (11111111 for infinity/NaN and 00000000 for subnormal/zero).
    - Mantissa patterns: Defined separately for bfloat16 (7 bits) and FP32 (23 bits).
- **Process**:
    - Patterns are generated for each combination of sign, exponent, and mantissa.
    - Separate patterns are created for integers A, B (16 bits), and C (32 bits).

2. Coverage Points

The framework defines coverage points to track input combinations and ensure that all scenarios are tested.

Coverage Points:

- A, B, and C inputs are tracked as separate coverage points, each with bins for 32 unique patterns.

Cross Coverage:

- Tracks combinations of all A, B and C values to ensure every possible interaction is tested.

Verification Flow

1. **Test Pattern Generation**:
    - Exhaustively generate inputs for A, B, and C using systematic patterns and random tests.
2. **Coverage Tracking**:
    - Track input coverage and ensure cross-combinations are exercised.
3. **Reference Model Validation**:
    - Use mac_int8 and mac_fp32 models for functional correctness checks.
4. **Matrix-Level Testing**:
    - Validate systolic array-level 4×4 matrix multiplication for both integer and floating-point inputs.