In the name of God

16-Bit Instruction Set Architecture (ISA) Implementation with Matrix Multiplication

Computer Architecture Project - Spring 2023
Arsalan Jabbari - Soroush Eskandari

---

1. Introduction:

The goal of this project is to design and implement a 16-bit Instruction Set Architecture (ISA) using the Verilog hardware description language. The project aims to develop modules for fundamental operations such as arithmetic, logical, memory, and control instructions. Additionally, a matrix multiplication operation will be implemented to showcase the practical application of the designed ISA.

2. Overview:

The project will consist of the following components:

a. Instruction Set Architecture (ISA) implementation:

- Design and implementation of an instruction set supporting **16-bit instructions**.
- Define opcode **partitioning**, register usage, and immediate value representation for the instructions, and maybe the direction for shift/rotate.
- Implement the following instructions:
    - **add**: Addition operation between two registers.
    - **addi**: Addition operation with immediate value and a register.
    - **shift**: Bitwise shift operation.
    - **rotate**: Bitwise rotation operation.
    - **beq**: Branch if equal to zero.
    - **sw**: Store word operation.
    - **lw**: Load word operation.
    - **jmp**: Unconditional jump instruction.

b. ALU and Sign-Extend Module:

- Design and implementation of the Arithmetic Logic Unit (ALU) module capable of performing arithmetic and logical operations on **16-bit** data.
- Development of a **sign-extend module** to handle sign extension of immediate values.

c. Test Bench Development:

- Creation of individual test benches for each module to verify their functionality.
- Writing test cases to ensure the correct behavior of instructions, including edge cases and error scenarios.
- Simulation and debugging of the modules using test benches.
- **For each module, you have to provide a test bench file, with a pattern name like “module_name_TB.v”.**

d. Matrix Multiplication:

- Implementation of a matrix multiplication operation **using the defined ISA**.
- **Design and implementation of the necessary modules** to handle matrix multiplication.
- **Verification of matrix multiplication functionality using appropriate test benches**.
- **You are not allowed to use multiply as an instruction.**
- **Matrix** dimensions is (1-N) * (N-1).
- Matrix values will be given. You should set them in a specific part of data memory and then implement your algorithm using the array's starting address.
- **The last result** is a single **Integer** that will evaluate that your Processor is working correctly.

3. Modules to implement:

Here is a list of the Verilog modules that you need to implement for the project:

- ***Instruction Memory***: This module contains your program machine code instruction that needs to be executed.
- ***PC***: This module is a 16-bit register that gets a new value in every clock's rising edge.
- ***ALU (Arithmetic Logic Unit)***: This module performs arithmetic and logical operations on 16-bit data. It should support operations like addition, shifting, rotating, and comparisons.
- ***ALU Control***: is not necessary.
- ***Register File***: This module represents a register file that stores values in registers. It should have read and write functionalities.
- ***Data Memory Unit***: This module emulates the memory and supports load (lw) and store (sw) operations. It can use a simple memory array or a more complex memory hierarchy depending on the desired complexity of the project.
- ***Sign-Extend Module***: This module takes a **5-bit immediate value and extends it to 16 bits** by sign extension.
- ***Control Unit***: This module controls the execution flow based on the opcode and other control signals. It generates control signals for different modules based on the current instruction.
- .***Shift left 1, Mux2_1(16 bit), 16BitAdder***: You can implement them using an assign statement and it is not necessary to create independent modules for them.
- ***Test Bench for each module***: Students should create individual test benches for each module to verify their functionality. These test benches should include various test cases covering different scenarios and edge cases.

Note: Depending on the design choices made by you, they might need additional helper modules or modules to handle data formatting, branching logic, or timing considerations.
Follow modular design principles and ensure that each module is thoroughly tested using the respective test benches before integration.

Good Luck!