Instruction Set Design for RISC-V

Student Name
Student ID

November 30, 2024

Abstract

This project focuses on designing a custom instruction set architecture (ISA) for a RISC-V CPU, incorporating essential components such as the Program Counter (PC), Arithmetic Logic Unit (ALU), Register File, Control Unit, and Memory Units. The custom instruction set design includes a range of operations, including arithmetic, logical, load/store, and branch instructions. The goal is to document the design, explain the control signals and data flow, and provide a sample program to demonstrate the functionality of the custom RISC-V CPU.

Contents

1 Introduction

This report presents the design and implementation of a custom instruction set for a RISC-V CPU, based on a specific data path and control unit layout. The CPU architecture facilitates arithmetic, logical, and branching operations, and integrates key components such as instruction memory, arithmetic and logic unit, register file, and data memory.

2 Custom Instruction Set Design

2.1 Supported Instructions

The following RISC-V instruction types are supported:

- **Arithmetic Instructions (R-type)**:
  - add: adds the values of two registers and storing the result into a third register
  - sub: subtracts values of two registers and storing the result into a third register
- **Logical Instructions (R-type)**:
  - and - logical bit-wise AND between the values from two registers
- **Branch Instructions (B-type)**:
  - beq: branches to a specified address if the values in two registers are equal.
  - blt: branches to a specified address if the value in the first register is less than the value in the second register.
- **Load/Store Instructions (I-type, S-type)**:
  - lw: loads a word from memory into a register.
  - sw: stores a word from a register into memory.
  - addi: adds immediate to a value from register and stores in specified register

2.2 Instruction Format

Each instruction type is aligned with the RISC-V instruction formats (R-type, I-type, S-type, B-type), utilizing opcode, funct3, and funct7 fields for decoding.

3 CPU Architecture and Datapath Components

The central processing unit (CPU) is the core component of a computer system, responsible for executing instructions and managing the flow of data. The architecture of a CPU defines its structure, organization, and the way it processes information. This section provides an overview of the main components and functions of a CPU architecture.

3.1 Program Counter (PC)

The program counter manages instruction sequencing by holding the address of the current instruction and incrementing it 4 bytes after each cycle.

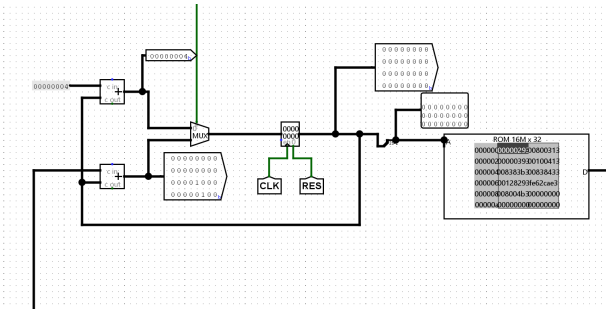

Figure 1: Program Counter as a part of the Instruction Fetch Stage

3.2 ALU

Performs arithmetic and logical operations based on control signals. It handles addition, subtraction, and logical operations like AND and OR.

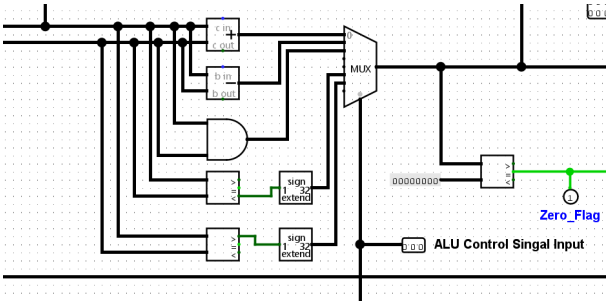

Figure 2: ALU

The first important input is the ALUOp signal from the control unit. Based on its value, we can determine the ALU Control Signal output using a reference table which is processed by the ALU Contol Unit.

- When ALUOp is 10 (binary 2), there is no exact ALU Control Signal value, so we use the second row of multiplexers based on the func3 value.
- Similarly, if func3 is 0, the ALU Control Signal is determined by the 5th bit of the main instruction opcode and func7.

3.3 Register File

Stores temporary values and supports dual-read and single-write operations for instruction execution.

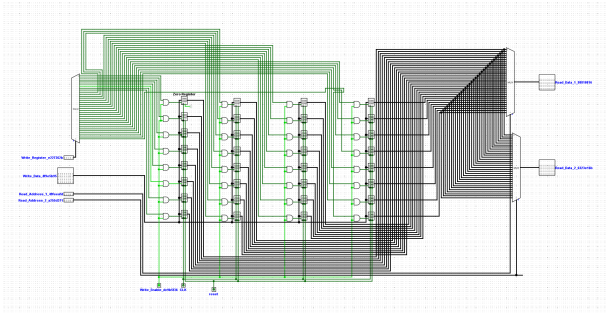

Figure 3: What the Register file looks like in Logisim

3.4 Control Unit

Decodes opcodes and generates control signals required for instruction execution, , such as whether an instruction requires a register write, a memory read/write, or an ALU operation.

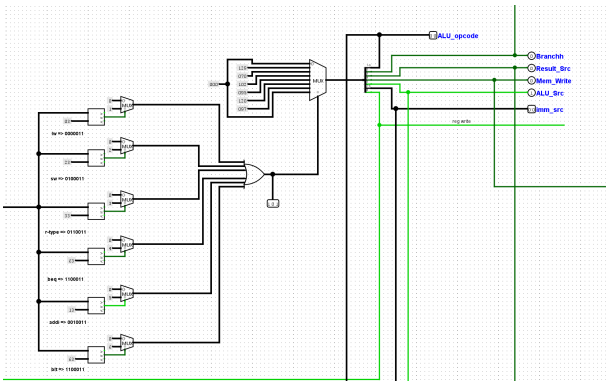

Figure 4: Control Unit

4 Control Signals and Instruction Mapping

4.1 Arithmetic Instructions

Example: `add x1, x2, x3`

- **Control signals:** ALUSrc = 0, RegWrite = 1
- **Data flow:** The values stored in registers `x2` and `x3` are input into the ALU, and the resulting value is written to `x1`.

Example: `sub x4, x5, x6`

- **Control signals:** ALUSrc = 0, RegWrite = 1
- **Data flow:** The value in register `x6` is subtracted from the value in register `x5`, and the result is stored in `x4`.

4.2 Load and Store Instructions

Example: `lw x5, 0(x6)`

- **Control signals:** ALUSrc = 1, MemRead = 1, RegWrite = 1
- **Data flow:** The immediate value `0` is added to the value in register `x6` to compute the memory address. The data at that memory address is then loaded into `x5`.

Example: `sw x7, 4(x8)`

- **Control signals:** ALUSrc = 1, MemWrite = 1
- **Data flow:** The immediate value `4` is added to the value in register `x8` to form the memory address. The data in register `x7` is stored at that address.

4.3 Branch Instructions

Example: `beq x1, x2, label`

- **Control signals:** PCSrc determined by the Zero flag from the ALU
- **Data flow:** The ALU compares the values in registers `x1` and `x2`. If they are equal (`Zero = 1`), the PC is updated to branch to the specified label.

Example: `blt x3, x4, label`

- **Control signals:** PCSrc determined by the comparison result
- **Data flow:** The ALU compares the values in registers `x3` and `x4`. If `x3` is less than `x4`, the PC is updated to branch to the specified label.

4.4 Logical Instructions

- **Control signals:** ALUSrc = 0, RegWrite = 1
- **Data flow:** The values in registers `x8` and `x9` are fed into the ALU for a bitwise AND operation, and the result is stored in `x7`.

5 Instruction Execution Plan

5.1 Execution Stages

Each instruction executes in five stages:
Instruction Fetch (IF):

- The CPU retrieves the instruction from memory at the address indicated by the Program Counter (PC).
- The PC is incremented by 4 to point to the subsequent instruction.

Instruction Decode (ID):

- The fetched instruction is interpreted to determine the operation the source registers, and the destination register. For immediate instructions, the immediate value is also extracted.

Execute (EX):

- The Arithmetic Logic Unit (ALU) performs the specified arithmetic operation using the values from the source registers (or a source register and an immediate value in the case of addi). The result is temporarily stored for the next stage.

Memory Access (MEM):

- Data is read from data memory using the address calculated in the EX stage.
- This stage is skipped for arithmetic instructions since they do not involve memory reads or writes.

Writeback (WB):

- The result produced by the ALU is written back to the destination register (`rd`), completing the instruction cycle.

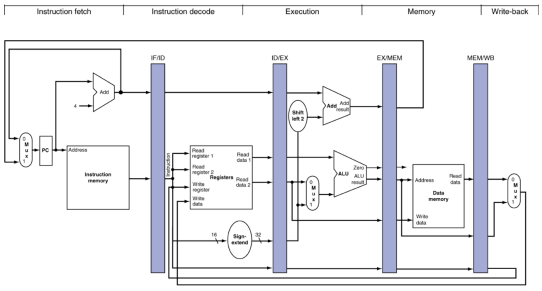

Figure 5: Single Cycle CPU

6 Sample Program and Demonstration

```
add t0, x0, t0        # t0 is counter
addi t1, t1, 8        # t1 is 8 so 8 repetitions
addi x7, x0, 0        # x7 is set to 0, first element of fibonacci
addi x8, x0, 1        # x8 is set to 1, second element of fibonacci

main_loop:
    add x7, x7, x8            # x7 += x8
    add x8 x7, x8             # x8 += x7
    addi t0, t0, 1            # t0++
    blt t0, t1, main_loop     # if t0 < t1, repeat main_loop

add x9, x0, x8        # x9 = x8 (last fibonacci number)
```

6.1 Program Explanation

Initialization:

- `addi t0, x0, 0`: Sets register `t0` to 0, which acts as a counter for iterations.
- `addi t1, t1, 8`: Sets register `t1` to 8, which specifies the number of iterations (8).
- `addi x7, x0, 0`: Sets register `x7` to 0, representing the first element of the Fibonacci sequence.
- `addi x8, x0, 1`: Sets register `x8` to 1, representing the second element of the Fibonacci sequence.

Main Loop:

- `main_loop:` Marks the beginning of the main loop.
- `add x7, x7, x8`: Adds the value of `x8` to `x7`, effectively calculating the next Fibonacci number.
- `add x8, x7, x8`: Updates `x8` with the newly calculated Fibonacci number.
- `addi t0, t0, 1`: Increments the loop counter `t0`.
- `blt t0, t1, main_loop`: Branches back to the beginning of the loop if `t0` is less than `t1`. This ensures that the loop runs for 8 iterations.

7 Design and Implementation Tools

7.1 Logisim Evolution

Used to design and simulate the CPU datapath and control unit.

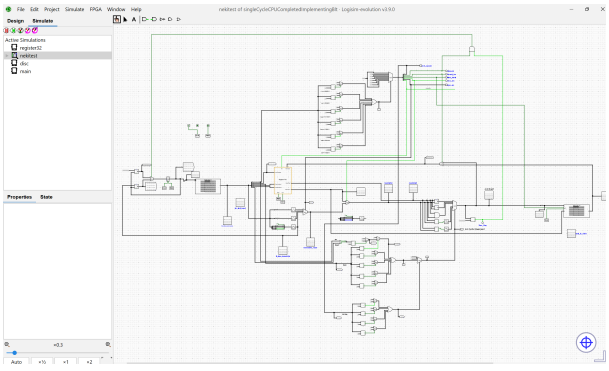

Figure 6: Logisim Environment

7.2 RARS (RISC-V Simulator)

Used to write and test the custom RISC-V instructions.

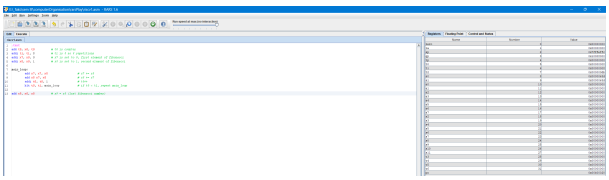

Figure 7: RARS Environment

7.3 GDB Debugging

Employed for in-depth analysis and debugging of complex assembly code.

8 Conclusion

This project provided valuable insights into CPU architecture, instruction set design, and control signal mapping. Design choices made throughout the project were guided by RISC-V architecture principles, with an emphasis on simplicity and efficiency.

Key outcomes include developing a functional instruction set that supports basic RISC-V operations, implementing a CPU design that effectively controls different instruction types, and trying demonstrate a strong understanding of computer organization principles through practical application. In the end this project provided practical experience in CPU design and implementation.

9 References

- Patterson, D. A., Hennessy, J. L. (2021). Computer Organization and Design RISC-V Edition: The Hardware/Software Interface.
- RISC-V Foundation. (2024). RISC-V Instruction Set Manual. Retrieved from `https://riscv.org/specifications/`
- Logisim Documentation. (2024). Retrieved from `http://www.cburch.com/logisim/`