Instruction Set Design for RISC-V

Selma Karasoftić and Amina Brković

November 2024

Abstract

This project focuses on the design of a custom instruction set for a RISC-V CPU, which includes the implementation of main components: Program Counter, ALU, Register File, Control Unit, and Memory Units. The designed custom instruction set supports arithmetic, some logical, and some branching operations. The key goals here are to document the design of such components, describe associated control signals, show data flow, and include a sample program illustrating the functionality of a custom CPU. The project also covers basics of implementing instructions in RARS.

Contents

1 Introduction

We are designing a custom instruction set for a CPU based on the RISC-V architecture. We chose RISC-V for its simplicity and modularity, which make it ideal for educational and experimental purposes. For this project, we will build upon an existing datapath and control unit.

Our focus will be on understanding and implementing essential instruction types, including arithmetic, some logical, and some branch operations. We will ensure these instructions integrate seamlessly with core components such as instruction memory, register files, the ALU, data memory, and the control unit. Our goal is to develop an efficient and functional CPU design.

2 Custom Instruction Set Design

2.1 Supported Instructions

The custom RISC-V CPU will support the following instruction types:

- **Arithmetic Instructions (R-type)**: addition, subtraction, multiplication and devision.
- **Logical Instructions (R-type)**: and, or, not
- **Branch Instructions (B-type)**: beq, bne
- **Load/Store Instructions (I-type, S-type)**: lw, sw

2.2 Instruction Format

Each instruction type adheres to the standard RISC-V instruction formats (R-type, I-type, S-type, B-type), employing opcode, funct3, and funct7 fields for accurate decoding and execution.

2.2.1 Instruction Format 1

- Operations: Arithmetic, Logical, Shift
- Example instructions: ADD (add rd, rs1, rs2), OR (or rd, rs1, rs2), SLL (sll rd, rs1, rs2) etc.

	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
R-type	funct3	rs2	rs1	funct3	rd	opcode

2.2.2 Instruction Format 2

- Operations: Arithmetic, Logical, Shift, Comparison, Load
- Example instructions: ADDI (addi rd, rs1, imm), LW (lw rd, imm(rs1)), ANDI (andi rd, rs1, imm) etc.

	7 bits 5 bits	5 bits	3 bits	5 bits	7 bits
I-type	immediate[11:0]	rs1	funct3	rd	opcode

2.2.3 Instruction Format 3

- Operations: Store
- Example instructions: SW (sw rs2, imm(rs1)), SB (sb rs2, imm(rs1)), SH (sh rs2, imm(rs1)) etc.

	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

2.2.4 Instruction Format 4

- Operations: Conditional Branching
- Example instructions: BEQ (beq rs1, rs2, imm), BLT (blt rs1, rs2, imm), BGE (bge rs1, rs2, imm) etc.

	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
B-type	imm[12,10:5]	rs2	rs1	funct3	rd	opcode

2.2.5 Instruction Format 5

- Operations: Conditional Branching
- Example instructions: BNE (bne rs1, rs2, imm), BLTU (bltu rs1, rs2, imm), BGEU (bgeu rs1, rs2, imm) etc.

	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
SB-type	imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode

2.2.6 Instruction Format 5

- Operations: Uncoditionals Jumps
- Example instructions: JAL (jal rd, imm), JALR (jalr rd, imm(rs1)) etc.

	20 bits	5 bits	7 bits
UJ-type	immediate[20,10:1,11,19:12]	rd	opcode

2.2.7 Instruction Format 6

- Operations: Load upper immediate
- Example instructions: LUI (lui rd, imm), AUIPC (auipc rd, imm) etc.

	20 bits	5 bits	7 bits
U-type	immediate[31:12]	rd	opcode

3 CPU Architecture and Datapath Components

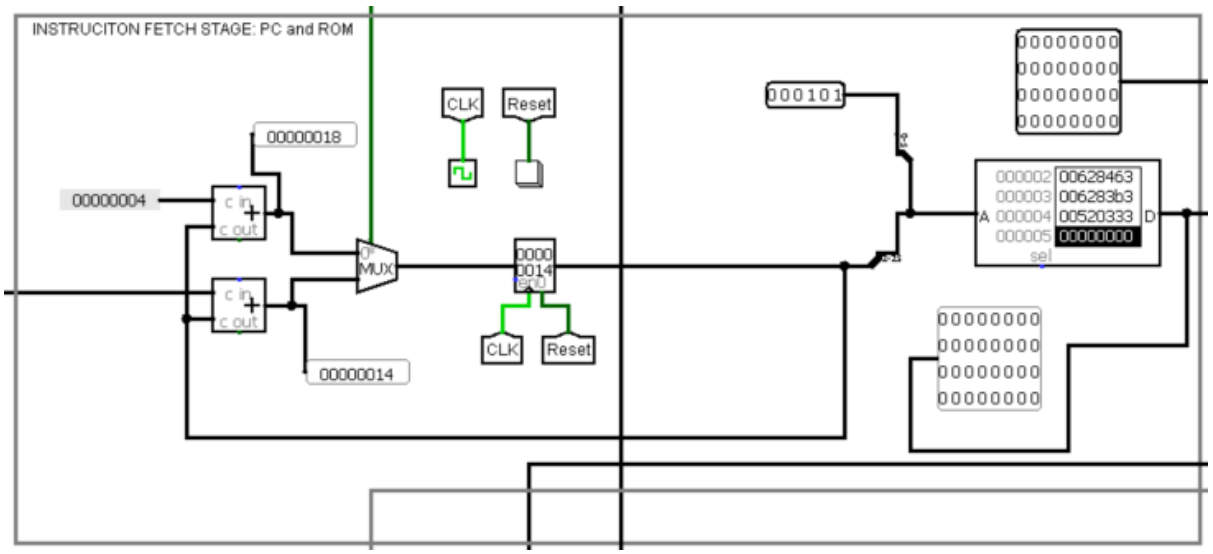

Figure 1: Instruction Fetch Stage: PC and ROM

3.1 ROM memory

Read-Only Memory (ROM) is a form of non-volatile memory utilized in processors and computing systems to hold data that remains constant during regular operation. ROM preserves its data even when the system is turned off, making it suitable for storing firmware, bootloader code, and other essential instructions needed for system initialization and functionality.

3.2 Program Counter (PC)

In the RISC-V architecture, the Program Counter is a key register that keeps track of the memory address of the instruction that is currently executing for proper program flow. The PC is initialized from a known address (like 0x00000000 or a bootloader entry) upon system reset and automatically increments by 4 after each instruction fetch, since RISC-V instructions are 4 bytes and aligned on 4-byte boundaries. This will align the data properly in memory and avoid misaligned instruction exceptions. The control flow instructions, which include branches-beq and bne-jumps-jal and jalr-and function calls, all change the PC in order to enable dynamic execution patterns such as loops, conditional logic, and subroutines. The PC also plays an important role in exception handling and interrupts by saving its value to allow resumption of execution after the handler completes. In some implementations, the program counter may be directly accessed or modified by particular instructions to allow for finer control over program flow. The PC also interacts with modern CPU features such as pipelining and branch prediction that ensure efficient and accurate instruction sequencing even in complex execution scenarios.

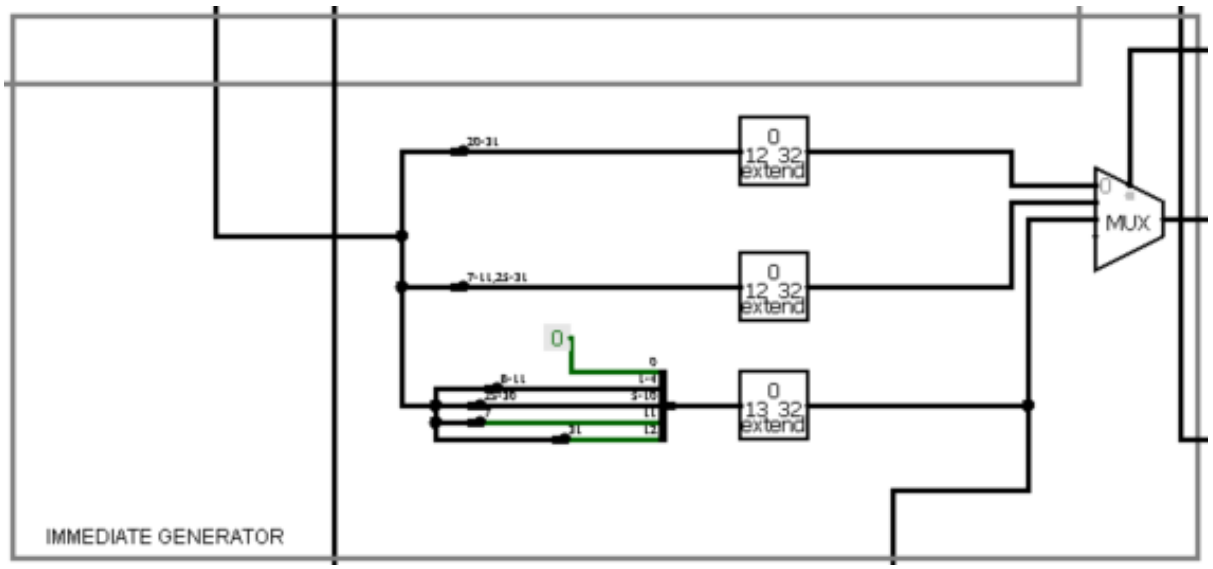

Figure 2: Instruction Decode Stage: Immediate Generator

3.3 Immediate Generator

The Immediate Generator is one important module in the RISC-V architecture that works by extracting, arranging, and sign-extending the immediate values embedded within the instructions for use by the ALU during execution. This immediate value can be seen in instructions such as ‘addi‘, ‘lw‘, ‘sw‘, and branch instructions, and they represent constants encoded directly in the instruction. The Immediate Generator decodes these values by referencing specific bit fields of the instruction, as defined by its format. For I-type instructions-for example, ‘addi‘, ‘lw‘-the immediate is in bits 31-20 and represents a 12-bit signed integer. In S-type instructions-for example, ‘sw‘-the immediate is in bits 31-25 and 11-7, requiring reassembly and sign-extension. For B-type instructions, the immediate is spread out across bits 31, 30–25, 11, and 7; when rearranged, it forms a signed 13-bit value. This whole process ensures that the extracted immediate values are in the proper format for arithmetic or memory operations. An implementation in Logisim will thus require strict adherence to the specification in the RISC-V Instruction Set Specification for I-, S-, and B-type instructions to correctly decode and extend immediate values.

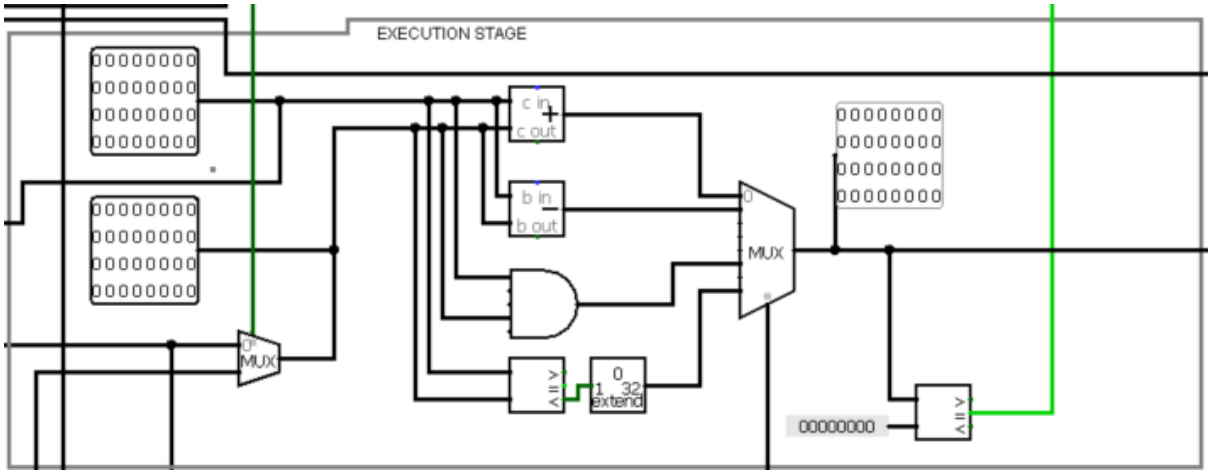

Figure 3: Execution Phase: ALU (ALU control signals, ALUSrc, ALUop

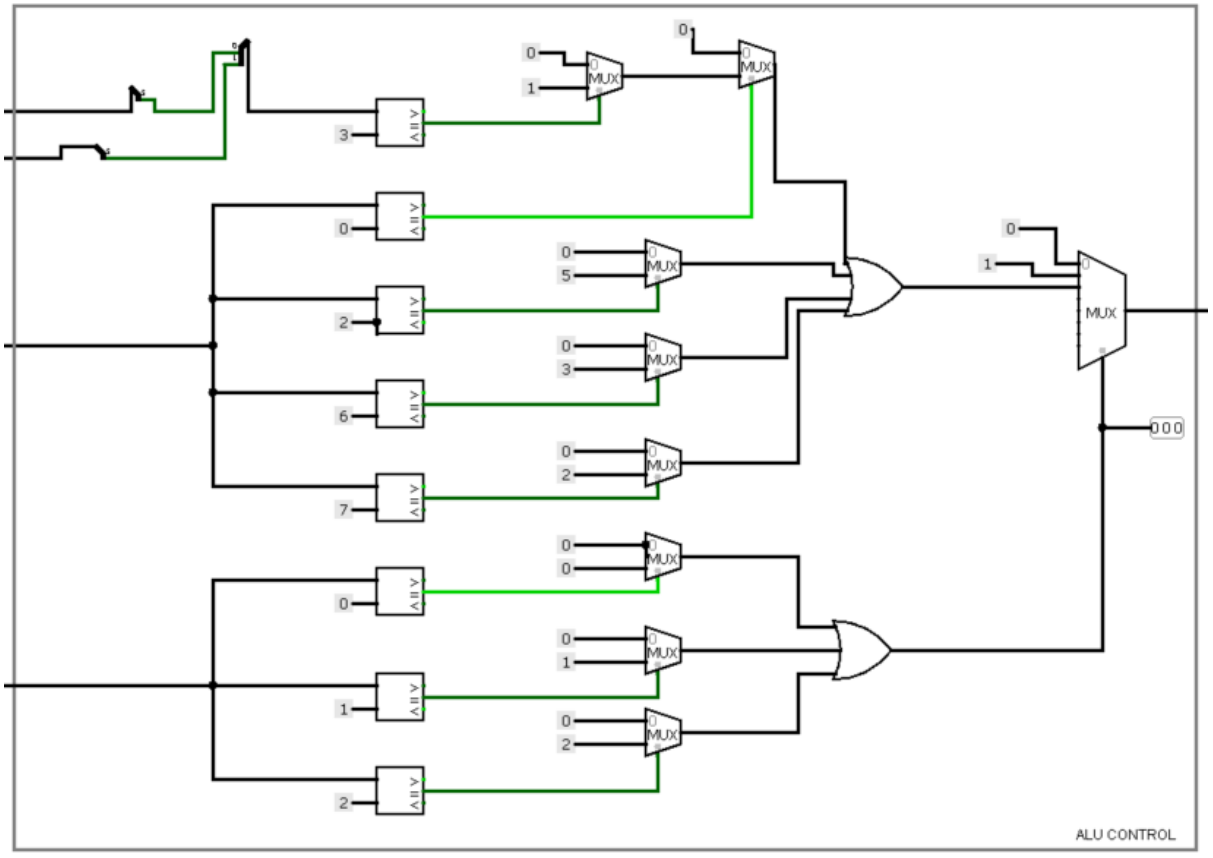

Figure 4: Instruction Fetch Stage: ALU control

3.4 ALU

The ALU Control in the RISC-V architecture plays a major role in setting the Arithmetic Logic Unit (ALU) for the proper operation to be performed depending on the instruction being executed. While the main control unit generates broad control signals, the ALU

Control unit refines these signals into specific instructions for the ALU. That involves the interpretation of the ‘funct3‘ and ‘funct7‘ fields of the instruction, which provide the detailed operation information with the ‘ALUOp‘ signal from the main controlling unit. The ‘ALUOp‘ provides a higher-order instruction on the nature of the operation to be carried out, such as addition or logical, while ‘funct3‘ and ‘funct7‘ are to specify the exact operation for, say, add, subtract, AND, OR, and set less than. For example, from an R-type instruction, funct3 can specify that the operation is an addition, while funct7 can select between a normal addition or subtraction. After decoding those fields, the ALU Control unit generates an appropriate control signal to set the ALU to perform the accurate execution of the operation concerned. This modular approach separates the concerns of general control and ALU-specific operation, simplifying design and enhancing flexibility in instruction handling.

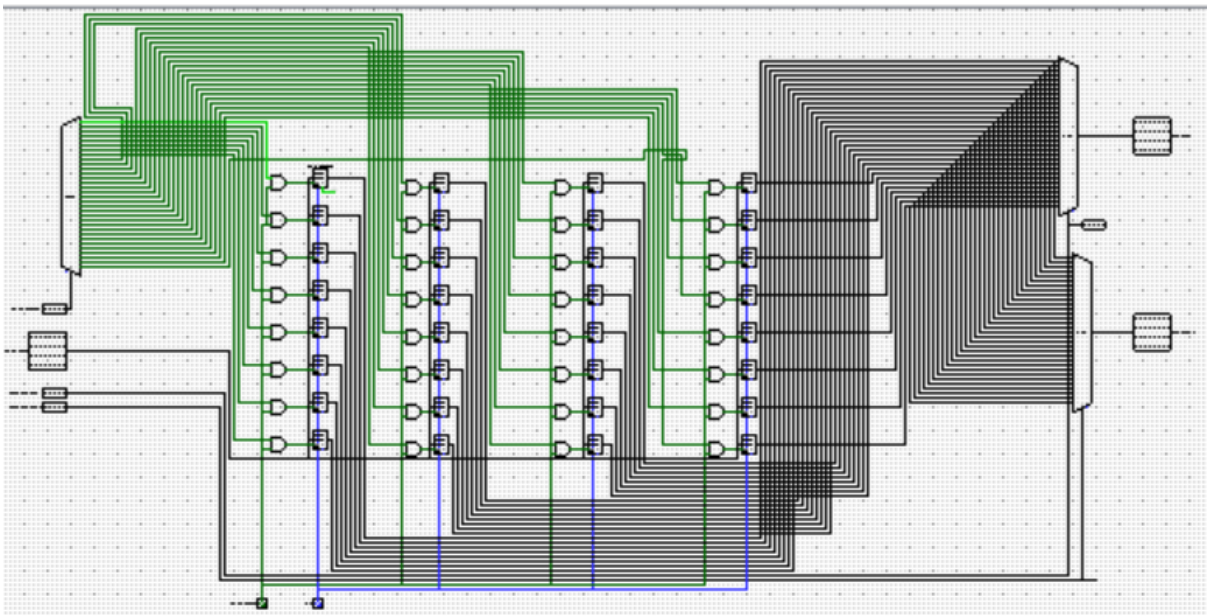

Figure 5: Instruction Decode Stage: Register

3.5 Register File

The RISC-V register file includes all 32 general-purpose registers, x0 through x31, and each holds 32-bit data. In the decode stage, it reads the values of source registers specified by the instruction-the contents of ‘rs1‘ and ‘rs2‘ provide data for arithmetic, logical, or memory access operations later in the pipeline. It allows reading and writing simultaneously to provide good data flow during the execution of instructions. Register x0 is hardwired to always return zero, regardless of writes, simplifying operations requiring a constant zero. This design ensures high-performance execution and seamless integration into the pipeline.

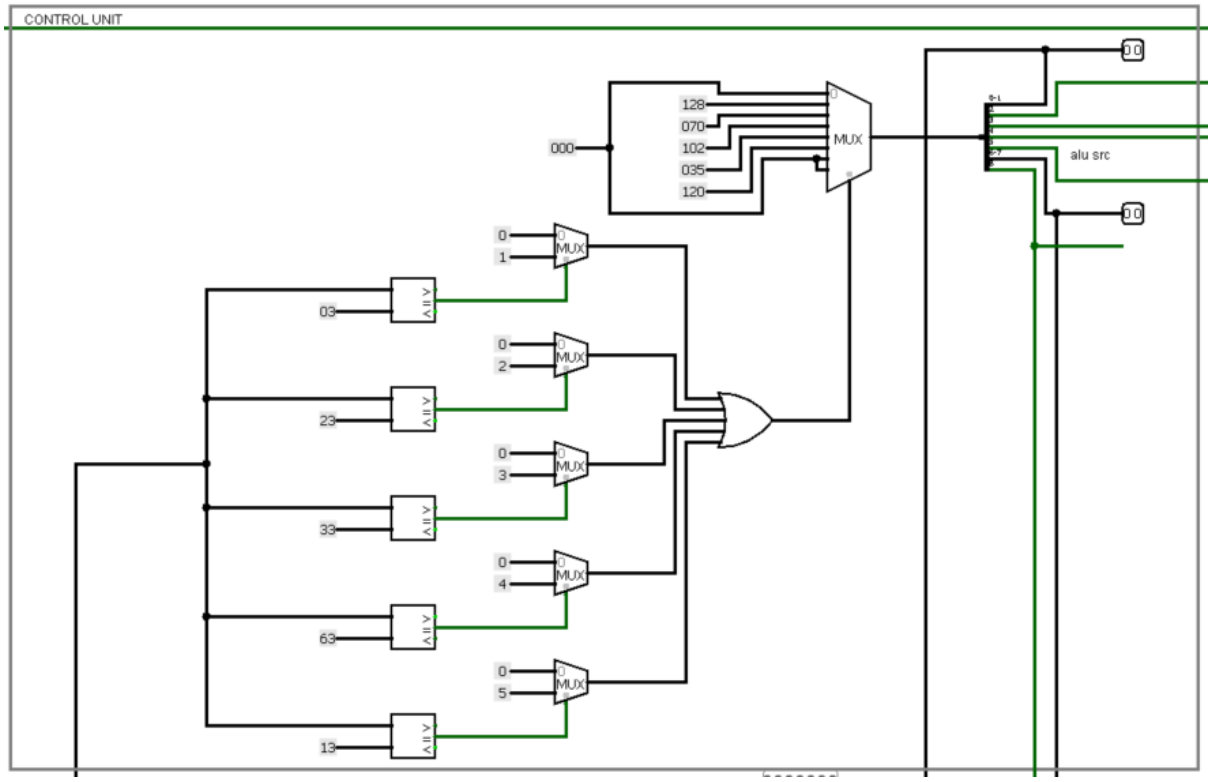

Figure 6: Instruction Decode Stage: Control Unit

3.6 Control Unit

The control unit is the centerpiece of the RISC-V CPU, and plays a crucial role in the ID stage by interpreting the opcode of the fetched instruction. Based on the opcode, it generates a set of control signals that guide the operation of various CPU components, determining how data flows through the processor. These signals identify whether the instruction needs to write into a register, read or write from memory, or execute an operation in the ALU. As an example, the control unit recognizes whether the instruction is a load (‘lw‘), a store (‘sw‘), a branch (‘beq‘), or an ALU operation like ‘add‘ or ‘sub‘, and then it enables accordingly: register file, memory access, or ALU. The control unit, by generating appropriate control signals, makes sure that the instructions are executed in coordination. Therefore, it enables the CPU to perform a wide range of tasks with great ease and efficiency. This simplifies the decoding process and makes it RISC-V compatible.

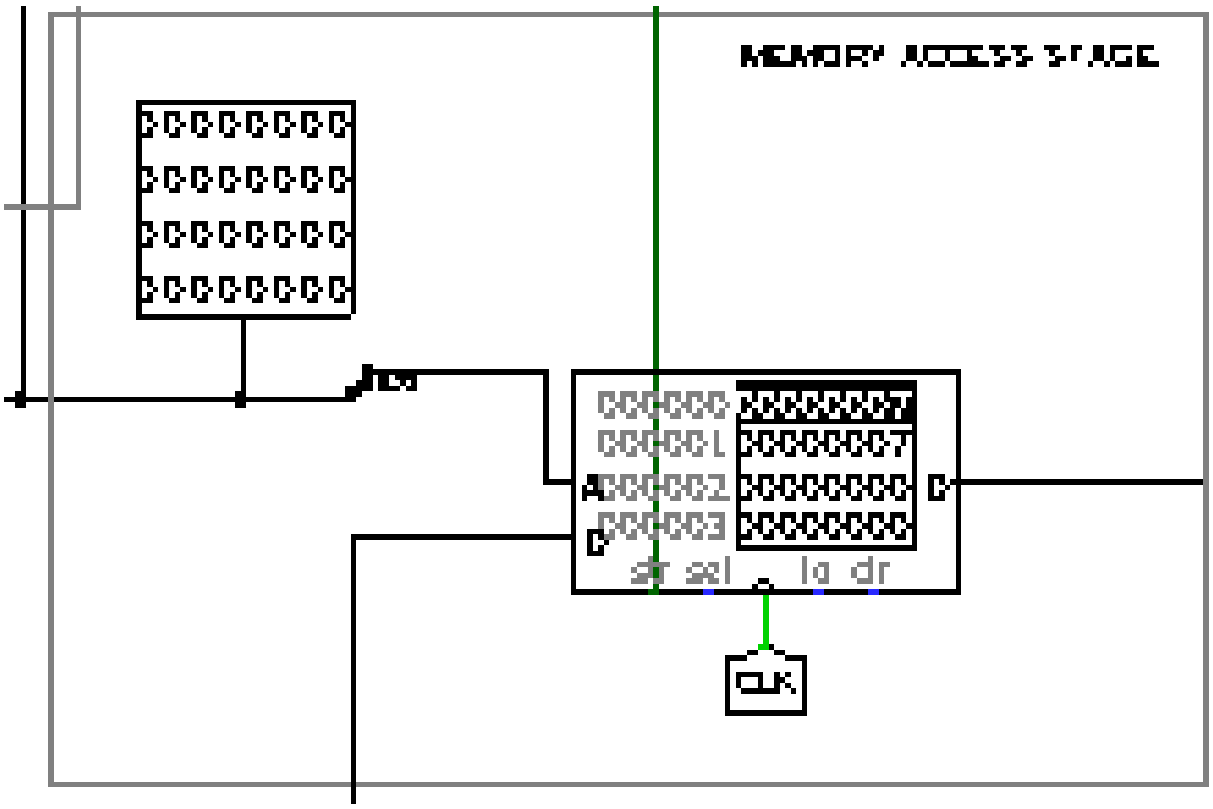

Figure 7: Memory Access Stage: RAM

3.7 RAM Memory

In the memory (MEM) stage of instruction execution, the CPU uses Random Access Memory, or RAM, to perform either a read or a write of data dynamically. This is quite different from the instruction fetch stage, which simply fetches instructions through Read-Only Memory. Here, RAM is important because it will support both reading and writing. This is how the CPU can load data from memory into the registers for processing or store results from the registers back into memory. For example, in load ('lw'), the CPU reads a particular memory address and retrieves data from that address for use, whereas in a store instruction ('sw'), the CPU writes data from some register into a memory address. In either case, the address is normally computed during the execution (EX) stage of the ALU and then supplied to the data memory in the MEM stage. The dual-read/write capability of RAM allows for the flexibility and functionality needed in dynamic data handling, making it an essential part of the CPU's datapath.

3.8 Whole CPU

A datapath in a CPU is the part of the architecture responsible for executing instructions. It contains components such as registers, multiplexers, ALUs (Arithmetic Logic Units), memory access units, and control logic. These components work together to carry out instructions that we entered in our ROM memory. We added 4 instructions which our program executed. To give a clearer perspective, we added some probes which are only showing us the value at that point during the instruction execution.

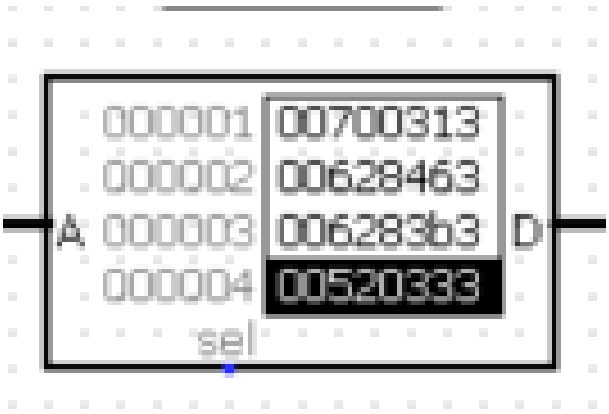

Figure 8: instructions

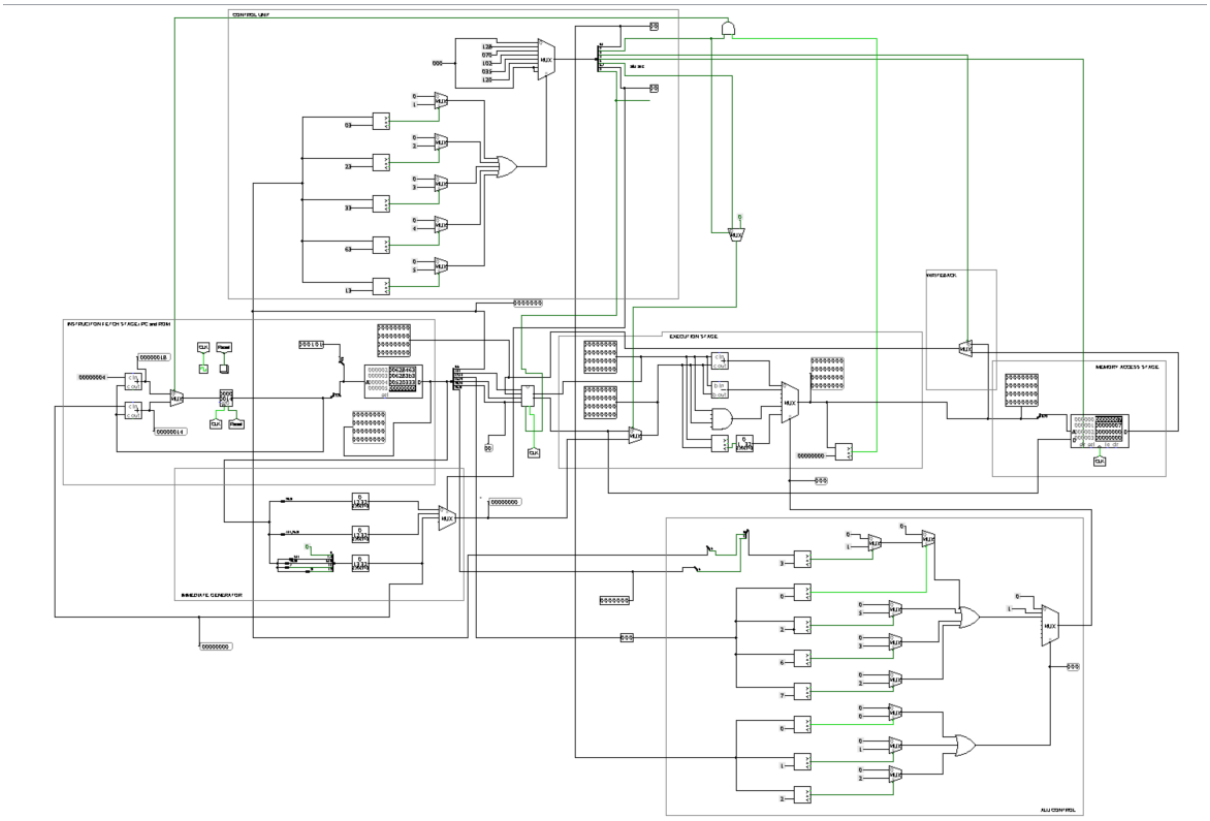

Figure 9: CPU containing all of the components listed above

Data saved in register:

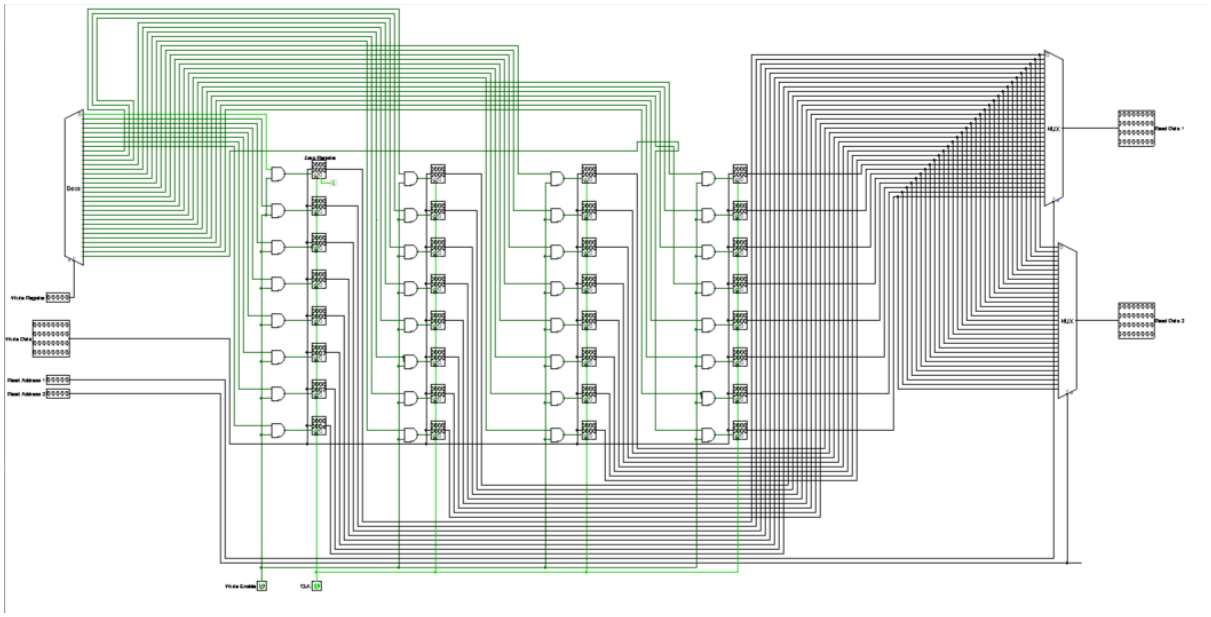

Figure 10: Registered with stored values

Close up view:

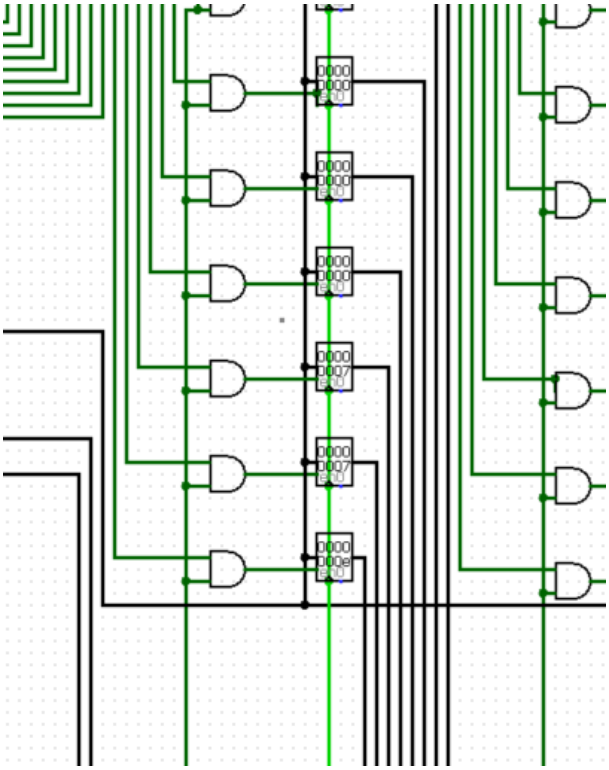

Figure 11: Data Stored in Registers

4 Control Signals and Instruction Mapping

4.1 Arithmetic Instructions

- **Example**: `addi x1, x0, 7`
  - **Control signals**: `ALUSrc = 1`, `RegWrite = 1`
  - **Data flow**: Immediate value `7` is used as one operand; `x0` (value `0`) is used as the other. The ALU performs the addition (`0 + 7`), and the result (`7`) is stored in `x1`.
- **Example**: `addi x2, x0, 9`
  - **Control signals**: `ALUSrc = 1`, `RegWrite = 1`
  - **Data flow**: Data flow: Immediate value 9 is used as one operand; x0 (value 0) is used as the other. The ALU performs the addition (0 + 9), and the result (9) is stored in x2.
- **Example**: `sub x3, x2, x1`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: Values in registers `x2` and `x1` are fed into the ALU. The ALU subtracts `x1` from `x2` (`9 - 7`), and the result (`2`) is stored in `x3`.
- **Example**: `add x5, x3, x4`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: Values in registers `x3` (`2`) and `x4` (`12`) are fed into the ALU. The ALU adds them (`2 + 12`), and the result (`14`) is stored in `x5`.
- **Example**: `mul x6, x4, x2`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: Values in registers `x4` (`12`) and `x2` (`9`) are fed into the ALU, which multiplies them (`12 * 9`). The result (`108`) is stored in `x6`.
- **Example**: `div x7, x6, x1`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: Values in registers `x6` (`108`) and `x1` (`7`) are fed into the ALU. The ALU performs integer division (`108 / 7`), and the result (`15`) is stored in `x7`.

4.2 Load and Store Instructions

- **Example**: `la x2, value`
  - **Control signals**: (pseudo-instruction, resolved during assembly)
  - **Data flow**: Loads the address of `value` into register `x2`.
- **Example**: `lw x4, 0(x2)`
  - **Control signals**: `ALUSrc = 1`, `MemRead = 1`, `RegWrite = 1`
  - **Data flow**: The value `0` is added to the base address in `x2` to form the effective address. The data is then read from memory at that address and stored in register `x4` (`x4 = 42`).

4.3 Immediate Arithmetic Instructions

- **Example**: `addi x4, x1, 5`
  - **Control signals**: `ALUSrc = 1`, `RegWrite = 1`
  - **Data flow**: The value in `x1` (`7`) is added to the immediate value `5` by the ALU. The result (`12`) is stored in `x4`.
- **Example**: `addi x8, x5, 2`
  - **Control signals**: `ALUSrc = 1`, `RegWrite = 1`
  - **Data flow**: The value in `x5` (`14`) is added to the immediate value `2` by the ALU. The result (`16`) is stored in `x8`.

4.4 Logical Instructions

- **Example**: `and x9, x5, x3`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: The values in `x5` (`14`) and `x3` (`2`) are fed into the ALU, which performs a bitwise AND operation. The result (`2`) is stored in `x9`.
- **Example**: `or x10, x4, x2`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: The values in `x4` (`12`) and `x2` (`9`) are fed into the ALU, which performs a bitwise OR operation. The result (`13`) is stored in `x10`.
- **Example**: `xor x11, x1, x2`
  - **Control signals**: `ALUSrc = 0`, `RegWrite = 1`
  - **Data flow**: The values in `x1` (`7`) and `x2` (`9`) are fed into the ALU, which performs a bitwise XOR operation. The result (`14`) is stored in `x11`.

4.5 Branch Instructions

Example: `beq x3, x1, label`

- **Control signals**: `PCSrc` determined by the Zero signal from the ALU
- **Data flow**: The ALU compares the values in `x3` (2) and `x1` (7). If the result is zero (indicating the values are equal), the program counter (`PC`) is updated to branch to the address indicated by `label`. In this case, `x3` is not equal to `x1`, so no branch occurs.

4.6 Additional Instructions

Example: `addi x10, x0, 4`

- **Control signals**: `ALUSrc = 1`, `RegWrite = 1`
- **Data flow**: The immediate value `4` is added to the value in `x0` (`0`). The result (`4`) is stored in `x10`.

4.7 Jump Instructions

Example: j end

- **Control signals**: `PCSrc = 1` (updates the Program Counter to the target address).
- **Data flow**: The target address of the `end` label is computed during assembly. When the instruction is executed, the Program Counter (PC) is updated to point to the address of the `end` label, unconditionally redirecting program flow to that location.

5 Instruction Execution Plan

5.1 Execution Stages

Each instruction executes in five stages:

- **Instruction Fetch (IF)**: This is the initial step in the execution of an instruction, where data is fetched from memory. The key components involved in this stage are the Program Counter (PC) and ROM memory. The PC stores the address of the current instruction, and once an instruction is executed, the PC is incremented by four, corresponding to the number of bytes per instruction. This automatic incrementation allows the CPU to process instructions sequentially unless control flow instructions, like "beq," dictate otherwise. Instruction addresses are stored in ROM (Read-Only Memory), which contains the instruction code in a fixed format that does not require modification during execution. The fetched instruction is then passed on to the next stage of execution.

- **Instruction Decode (ID)**: This is the second stage, which involves more components responsible for interpreting and preparing data for the next phase. The key component in this stage is the Control Unit, which plays a central role by decoding the instruction's opcode to generate control signals. These signals dictate how data moves through the CPU and specify the operations to be performed, whether accessing the register file, performing memory read/write, or executing an ALU operation. Another critical component is the Register File, which contains all general-purpose registers. It retrieves the values of the source registers from the instruction and prepares the data for ALU operations. Additionally, the Immediate Generator and ALU Control are vital in this phase. The Immediate Generator extracts specific bits from the instruction, arranges them based on the instruction format, and sign-extends them as necessary to prepare the data for the ALU. The ALU Control unit, on the other hand, uses funct3 and funct7 from the instruction along with the OP signal from the Control Unit to produce the signals that configure the ALU for the exact operation required.

- **Execute (EX)**: In this phase, the actual computation occurs. Arithmetic and logical operations are carried out, memory addresses for load and store instructions are calculated, and branch conditions are evaluated. The primary components in this stage are the ALU and multiplexers. Multiplexers select the appropriate data inputs for the ALU based on control signals. Key signals in this phase include: **ALU Control Signals**: These specify the operation the ALU should perform, such as addition, subtraction, bitwise AND, OR, and comparison operations for branches, using funct3 and funct7. **ALU Source**: This signal controls a multiplexer that determines the ALU's second input. For arithmetic instructions, the ALU uses two register values. For instructions like `addi`, `lw`, or `sw`, this signal selects the immediate value as the second input instead of a second register. **Branch Signal**: This indicates whether an instruction involves branching. It works with the zero flag to decide if the program should update to a branch target address or continue sequentially. **Result Source**: For some instructions, the ALU result is not directly used. This signal controls a multiplexer that determines whether the result comes from the ALU output or from memory.

- **Memory Access (MEM)**: Load and store operations interact with the data memory. In this phase, the CPU reads data from or writes data to memory using the effective address calculated during the execution phase. Key signals and components in this phase include: **Data Memory (RAM)**: Unlike the Instruction Fetch phase, which uses ROM, the CPU utilizes RAM to dynamically load data into registers or store data from registers to memory locations. **Memory Address**: This is the effective address calculated by the ALU during the execution phase. For `lw` instructions, it specifies the address from which the CPU retrieves data, while for `sw` instructions, it indicates the address where data should be stored. **Memory Write Signal**: This signal, generated by the control unit, is activated for store instructions. When MemWrite is set to 1, the CPU writes data from the source register into RAM at the specified address. For load instructions, MemWrite is set to 0 to prevent unintended writes. **Data to Store**: This input is used for all memory instructions. It represents the `rs2` data, which is the value read from the second register in the instruction.

- **Writeback (WB)**: In the final stage, the results of specific operations are stored back into the CPU's register file. This step is particularly crucial for arithmetic operations and load instructions, as it writes the results back to the register file. Key components and signals in this stage include: **Register File**: The operation's result is written to the destination register specified in the instruction (`rd`). The register file is a collection of general-purpose registers that store data required by the CPU during execution. Writing results back to the registers ensures that the CPU can use this data in subsequent instructions.

  Write Data Source: The data to be written back varies depending on the instruction type. For arithmetic and logical instructions, the data comes from the ALU's result. For load instructions, the data is retrieved from RAM during the memory phase (`MEM`).

6 Sample Program and Demonstration

Listing 1: Sample RISC-V Assembly Program

```
    # Sample program to demonstrate arithmetic,
    # load/store, and branch operations
.data
value:   .word 42       # Define a word in memory
                        # initialized with the value 42

.text
.globl main
main:

    la x2, value        # Load the memory address
                        # of 'value' (which contains 42) into register x2

    lw x4, 0(x2)        # Load the word at the memory address in x2 into
                        # register x4 (x4 now holds 42)

    addi x1, x0, 7      # add immediate value 7 into register x1 (x1 = 7)
    addi x2, x0, 9      # add immediate value 9 into register x2 (x2 = 9)
    sub x3, x2, x1      # Subtract x1 from x2 (9 - 7),
                        # store result in x3 (x3 = 2)

    addi x4, x1, 5      # Add immediate value 5 to x1 (7 + 5),
                        # store result in x4 (x4 = 12)

    add x5, x3, x4      # Add the values in x3 and x4 (2 + 12),
                        # store result in x5 (x5 = 14)

    mul x6, x4, x2      # Multiply the values in x4 and x2 (12 * 9),
                        # store result in x6 (x6 = 108)
    div x7, x6, x1      # Divide the value in x6 by the value in x1 (108/7),
                        # store the quotient in x7 (x7 = 15)
    addi x8, x5, 2      # Add immediate value 2 to x5 (14 + 2),
                        # store result in x8 (x8 = 16)

    and x9, x5, x3      # Perform bitwise AND between x5 (14) and x3 (2),
                    # store result in x9 (x9 = 2), bitwise AND of
                    # 14 (0000 1110) and
                    # 2 (0000 0010) results in 2 (0000 0010)

    or x10, x4, x2      # Perform bitwise OR between x4 (12) and x2 (9),
                    # store result in x10 (x10 = 13), bitwise OR
                    # of 12 (0000 1100) and 9 (0000 1001)
                    # results in 13 (0000 1101)

    xor x11, x1, x2     # Perform bitwise XOR between x1 (7) and x2 (9),
```

```
                    # store result in x11 (x11 = 14),
                    # bitwise XOR of 7 (0000 0111)
                    # and 9 (0000 1001) results in 14 (0000 1110)

    beq x3, x1, label # Compare x3 and x1; if x3 equals x1,
                      # branch to 'label' (in this case, x3 != x1,
                      # so no branch)

    j end             # Explicit jump to avoid falling through to label

label:
    addi x10, x0, 4   # Load immediate value 4 into register x10 (x10 = 4)

end:
    li a7, 10         # Load the exit system call number into a7
    ecall             # System call to exit the program
```

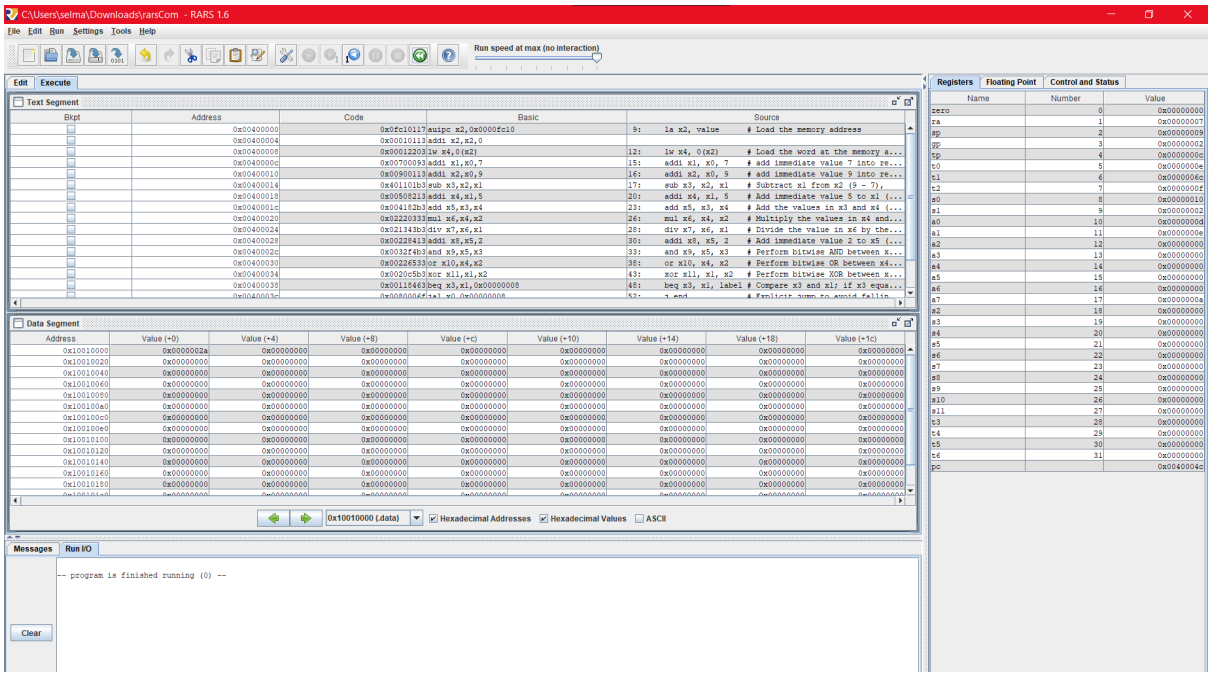

Figure 12: RARS program

6.1 Program Explanation

The program illustrates the execution of different RISC-V operations, highlighting key features of the assembly language. It incorporates arithmetic operations such as addition (addi, add), subtraction (sub), multiplication (mul), and division (div) to modify register values. A memory load operation (lw) fetches data from a specified memory location, demonstrating the interaction between memory and registers. Logical operations like bitwise AND (and), OR (or), and XOR (xor) are executed to showcase fundamental logical computations. A conditional branch operation (beq) compares specific register

values for equality and branches to a labeled section if the condition holds true, while an unconditional jump (j) facilitates smooth program flow. The code employs a label (label) as a branching target and performs various register operations, including immediate value loading, arithmetic computations, and control flow management. Each instruction is supported by detailed comments, ensuring the program is straightforward to understand and follow step by step. This program provides a comprehensive demonstration of how registers, memory, and control flow instructions collaborate in RISC-V assembly.

7 Design and Implementation Tools

7.1 Logisim

Logisim is a graphical design and simulation tool widely used for teaching and designing digital logic circuits. In this project, Logisim was utilized to design and simulate the CPU's datapath and control unit, providing a visual approach to understanding how these components function. It allowed us to create digital building blocks necessary for a functional CPU design.

7.2 RARS (RISC-V Simulator)

RARS, which stands for RISC-V Assembler and Runtime Simulator, was a crucial tool in the project for writing, testing, and debugging custom RISC-V instructions. This lightweight simulator allowed us to implement assembly-level programs tailored to the custom instruction set we developed. It provided detailed insights into how instructions were executed, including the step-by-step changes to registers, memory, and program control flow. RARS also offered a user-friendly interface to monitor the behavior of our instructions in real-time. Key features, such as the ability to visualize register contents and memory states, made it easier to identify and correct issues during testing.

8 Conclusion

This project gave great practical experience in CPU architecture and the design of the instruction set, emphasizing how components like the Program Counter, ALU, Register File, and Control Unit all integrate together. This helped us to actually understand what is happening in the core of our processor. The custom-designed instruction set well supported the key operations of RISC-V. The design and testing of the instruction set allowed us to understand how instructions are executed, what the role of control signals is, and how CPU components can be coordinated efficiently. In this way, practical work bridged theoretical knowledge and its application, allowing a deeper understanding of how complex and challenging modern processor design is.

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/`
- Lecture Notes from IT 208
- LAB Notes from IT 208