Computer Organization and Assembly Language Programming   
Project 1: RISC-V RV321

short line

Project Report

1. Implementation Description

Our project is a simple RISC-V simulator implemented in Python. It simulates the execution of RISC-V instructions by emulating a subset of the RISC-V architecture. The simulator supports basic instructions such as arithmetic and logical operations, memory operations, and control flow instructions. In our implementation, we have defined a Memory class that represents the memory of the simulated system. It consists of a collection of pages, each of which has a page number and a data array representing the content of the page. The memory class provides methods for reading and writing data at specific addresses. We also have a Page class that represents a single page in memory. Each page has a page number and a data array, which is initialized to a size of 4KB (4096 bytes). The simulator uses a dictionary to store the register values. The keys of the dictionary are the register names, and the values are the corresponding register values. We have also implemented a ConstantDict class that extends the dictionary class to prevent modification of the zero register.

The main logic of the simulator is implemented in the form of functions for each RISC-V instruction. For example, we have functions like add, sub, AND, OR, XOR, etc., to handle arithmetic and logical operations. These functions update the register values and the program counter accordingly. To handle memory operations, we have functions like LBU, LHU, LWU, LDU, LB, LH, LW, and LD, which read data from memory and store it in registers. We also have functions like sll, srl, sra, which perform logical shift operations on registers. Control flow instructions such as beq, bne, bge, bgeu, blt, bltu are implemented as functions that modify the program counter based on the condition specified.

In addition to the components mentioned earlier, our RISC-V simulator implementation includes several functions and a main simulation loop that orchestrates the execution of instructions. We have functions to parse the instruction code and extract relevant fields such as opcode, operands, and immediate values. These functions help in decoding the instruction and determining the appropriate action to take. The main simulation loop fetches instructions from memory based on the program counter and uses the opcode to identify the instruction type. It then calls the corresponding function to execute the instruction, passing the operands and immediate values as arguments. During the execution of each instruction, the simulator updates the register values and the program counter according to the instruction's semantics. For example, an arithmetic instruction like add will add two register values and store the result in a destination register.

To facilitate debugging and monitoring, we have included print statements at key points in the simulation loop to display the instruction being executed, the register values before and after the instruction, and any relevant status flags or memory modifications.

In addition to the command-line interface, we have enhanced our RISC-V simulator implementation by incorporating a graphical user interface (GUI) using Flask, a popular web framework in Python. The GUI provides a more user-friendly and interactive experience for running and visualizing the execution of RISC-V programs. It allows users to input their RISC-V code, step through instructions, and observe the state of registers and memory during program execution. We have designed the GUI to have a code editor where users can write or paste their RISC-V assembly code. The code editor provides syntax highlighting and basic code editing features to aid in code development. Once the code is entered, the user can click a "Run" button to start the execution of the program. The simulator processes the instructions and updates the register and memory values accordingly. The GUI displays the current instruction being executed, the contents of registers, and the memory state.

2. Design Decisions and Assumptions

In our simulator, we made the following design decisions and assumptions. We implemented the simulator in Python as it is a widely used and accessible programming language with extensive support for data manipulation and scripting. We chose to represent memory as a collection of pages, with each page having a fixed size of 4KB (4096 bytes). This allows us to simulate memory access and provide a basic memory management mechanism.

We decided to implement each RISC-V instruction as a separate function to improve code modularity and maintainability. This allows us to easily extend the simulator to support additional instructions in the future. We assumed a simplified version of the RISC-V architecture and implemented a subset of instructions. This subset covers basic arithmetic and logical operations, memory operations, and control flow instructions. We did not implement more complex instructions or features like pipelining or caching. We assumed a little-endian memory organization, where the least significant byte is stored at the lowest address.

3. Known Bugs or Issues

Needs to rerun the entire program after each code to work

4. A user guide showing how to guide compile and run your simulator including a full simulation example step-bystep with screenshots

5. A list of programs (and associated data if any) you simulated. You should at least provide 3 programs. The programs must cover all instructions supported and one of them at least must have a loop.

Test case #1:

"""addi a2, zero, 3

addi a3, zero, 3

bne a2, a3, Else

add a0, a1, a2

beq zero, zero, Exit

Else: sub a0, a1, a2

Exit: addi s0, s0, 1"""

Test case #2:

“li a2, 5

li a3, 3

beq a2, a3, Equal

add a0, a1, a2

jal ra, Exit

ecall

Equal:sub a0, a1, a2

Exit: addi s0, s0, 1

jalr zero, 0(ra)

“

Test Case #3:

“li a2, 10

li a3, 7

bgeu a2, a3, GreaterThanOrEqual

addi a0, a1, 5

jal ra, Exit

ecall

GreaterThanOrEqual:

sub a0, a1, 3

Exit:

addi s0, s0, 1

“

6. Optionally, you can include a section about your experience working on this project. This section will NOT affect your grade in any way.

Working on this project has been a valuable learning experience that has deepened our understanding of computer architecture and simulation. Developing a RISC-V simulator from scratch has allowed us to explore the inner workings of a processor and gain insights into how instructions are executed.

Throughout the project, we encountered various challenges that required problem-solving and critical thinking skills. Designing the memory structure, implementing the instruction set, and managing the program flow all demanded careful consideration and attention to detail. Debugging and testing the simulator also played a crucial role in ensuring its accuracy and functionality.

One aspect of the project that enhanced its usability and user experience was the integration of a graphical user interface (GUI) using Flask. The GUI provided a user-friendly interface for interacting with the simulator, allowing users to easily load and run their RISC-V programs. The addition of the GUI made the simulator more accessible to individuals without a deep understanding of the underlying architecture.

Overall, working on this project has not only reinforced my technical skills but also fostered a greater appreciation for the complexities involved in developing a functional simulator. It has provided me with hands-on experience in implementing computer architectures and has expanded my knowledge in the field of processor simulation. I am proud of the accomplishments achieved during this project and look forward to further exploring and expanding my skills in the realm of computer architecture and simulation.

To use a GUI, there are several conditions that must be met. Firstly, the code must run at least once without any GUI implementation. This is important to ensure that the code works correctly before any GUI functionality is added.

Secondly, the code should be entered without any line breaks. This is necessary because line breaks can cause issues with the GUI implementation and can make it difficult to read and understand the code.

Thirdly, tables should be placed on the same line as the next instruction. This is important for readability and to ensure that the code is properly structured.

Finally, when entering data into the GUI, it should be in the format of "address, value". This is important to ensure that the data is properly formatted and can be correctly interpreted by the code. By following these conditions, you can ensure that your code is suitable for a GUI implementation and will work correctly.