Project Report

Risc V Simulator

Adityanath Madishetti

4th September, 2024

Note :

1. Minimalistic Risc V simulator
2. Supports basic instructions
3. 0x0 - 0x10000 text section
4. 0x10000 - 0x50000 Data
5. 32 general purpose registers
6. Memory here is byte addressable

Table of contents

Introduction

This project aims to develop a basic and minimalistic Risc V 64 simulator

Key-points

> It's a terminal based interactive simulator

> It has specific commands to simulate the RISC V assembly code

> It is built upon the previous project RISCV ASSEMBLER

> C++ 17 is used to build this simulator

> A make file is provided for the usage of this simulator

> This report outlines the approach, challenges, and experiences during the implementation of the Risc V simulator.

Implementation Details

- Uses c++ 17 features like std : : optional and string_veiw to accomplish the task
- Utilized data structures like maps (for label management),vectors (for instruction storage),

and strings (for parsing).

- String stream and filestream are used to collect information from file and parse it and generate Hexcode
- The generated hex code is collected from the Assembler object and some meta information which is useful for further development is stored in some variables
- An Infinite loop works as terminal to collect inputs and interact with users
- Cpp types like uint8_t , int64_t are used to mimic Byte memory and registers
- Operator overloading is done for arithmetic on Registers
- Register class and memory class encapsulate memory

COMMANDS

→ load <filename>

This loads the file into simulator and fills the text section and sets all the memory locations to o by default

→ run

This executes the binary code and gives the output

→ step

This runs one instruction at a time .

→ break <line_no> and del break <line_no>

This allows debugging the code it stops the code execution at specific instructions and allows us to check variables at that point and we can delete breakpoint also

→ regs

This command prints all the register values at taht instant in hexadecimal format

→ mem <address> <count>

This allows us to view memory at any instant of time

It start from <address> and prints count number of memory location

→ show-stack

This commands keep track of function class from jal and returning from function jalr in form of function stack

→ exit

This is to gracefully exit from the simulator after its use

APPROACH :

Following information try to address the approach used to build the simulator in cpp

1) Lexical analysis

→Upon loading a parser object is instantiated and it encodes instructions to the binary.

→ these instructions are collected by simulator
In a vector which contains these binary lines and their original line number in Input file

→ Now Loop starts and waits for user input command

2) Parsing and simulating code :

These binary lines are sent to a function called identify_and_run .This deduces the information type from the first 7 bits (LSB) and sends to the respective decoding functions

This functions extract

- Source reg
- Destination reg
- Func bits
- Immideates

This way actual instruction is found

3) Processing the Instructions

→ once the exact instruction is identified

The arithmetic required for it is done and required effects may be applied to memory and registers

4) Labels and jump instructions

→ Pc a special variable is maintained it is Incremented by 4 because each instruction is 4 bytes

Based on immediate (Branch instruction) the pc is changed and control reaches to that instruction. Jalr instruction makes pc as the value of source register

3) CODE DIVISION

→ code is divided into many files as to differentiate by use of the code and maintain modularity

→ Namespaces are used to wrap and abstract the unwanteds things from user

The code is divided Header and Source Files

→ code clarity but also optimized the compilation process by ensuring that changes to function logic didn't require recompiling the entire project.

Experiences :

→ The process of simulator development enhanced my logic building and language skills

→I explored various C++ libraries, especially for handling string parsing and file input/output, which streamlined the tokenization process.

→I became aware C++17’s features like STRING VIEW for better memory management

→ got a glance how compilation and os works while simulating the code

→ go to know working of many instructions and reason and purpose behind their use

→ became aware of internals of a simulator of some assembly language

Limitations

Parsing Flexibility :

Currently the working of the simulator strongly depends on the format in which its return . Treating commas and spaces equally would make the parsing process more flexible and user-friendly.

Pseudo Instructions :

Pseudo instructions make the assembly lab=nguage more simplified and easy to use .Current version of assembler and simulator does not support pseudo instructions

7.Comment Handling:

The assembler does not detect comments that are placed on the same line as an instruction. This can lead to undefined behavior, as the comment is not stripped before processing the instruction

Conclusion

The project not only deepened my understanding of RISC-V but also improved my skills in parsing, data handling, and low-level programming. Future improvements could focus on optimizing the simulators' performance, enhancing error reporting, and expanding support for more complex RISC-V instructions.

REFERENCES

1. CPP REFERENCE
2. RISC-V MANUAL
3. Geeks-for-Geeks
4. JACARD-SIMILARITY