LAB3: RISC-V Machine-Level Instructions Report

Kartikeya Mandapati
CS22BTECH11032

October 11, 2023

1 Introduction

This report presents a program that converts machine-level instructions into RISC-V assembly code. The tool deciphers binary instructions, making them human-readable and aiding in code analysis. The report outlines the program's design, implementation and practical usage.

2 Code Overview

The disassembler is implemented in C++ and follows a modular structure. It can handle various types of RISC-V instructions, including R, I, S, B, U, and J formats. Each instruction has its own function to decode to Assembly level. Here's an overview of the code structure:

2.1 Main Program

The main program reads machine-level instructions from an input file, disassembles them, and displays the corresponding assembly instructions. The input file is set as "input.txt" in the main. All the instructions are stored into a vector of strings and are looped one by one to convert into assembly level. The input is converted to binary string in the main function. For the first time the instructions are looped through and labels are created. It is looped through again this time calling the respective functions. The opcode is also separated and based on the opcode, the function for the respective instruction is called with the binary string or integer converted value based on its requirement.

2.2 Instruction Formats and Functions

The code is organized into functions for different instruction formats, including R-format, I-format, S-format, B-format, U-format, and J-format. Each function handles the disassembly of instructions in its respective format. In each of the functions the **rs1,rd,imm,funct3,funct7** are separated based on the instruction format and the function is identified and at the enf the converted instruction is printed using the function and other parameters that have been used. The functions use **substr** or logical operators like $\ll$,$\gg$,& based on the complexity of instruction format. The functions are **R_format,I_format1,I_format2,J_format,S_format,B_format,U_format**.
The J type and B type are just a bit different from others as they have a return value.The code follows the following sequence

1. Adding labels to the instructions.
2. Instruction format Classification using opcode.
3. Other fields calculation.
4. Instruction identification using funct3.
5. Ouput in sequence.

2.3 Other Functions

I have made other functions for converting **hexToBinary** and **binaryStringToInt** to use them based on the requirement.

2.4 Label Handling

The code can also handle branch and jump instructions that require label resolution. It dynamically generates labels for branch targets and jump addresses. I used an unordered map that contains the sudo **PC** and respective dynamic label. When ever the code comes accross a B-type or J-type, it creates a label for the dest PC. And for each instruction it checks if its PC matches with any of them that are meant to have labels and a label is printed in the front based on that. Labels are created in the first loop as there could be the possibility that label is created after the instruction has been disassembled.

3 Usage

The program's usage is well-documented and provided in the README file. To use the disassembler program:

1. Prepare an input file containing machine-level instructions in hexadecimal format.
2. Compile the C++ program.
3. Run the program, providing the input file name chnaged to that of yours.
4. The program will disassemble the instructions and display the corresponding assembly instructions.

4 Results

The disassembler program successfully converts machine-level instructions into human-readable RISC-V assembly instructions. Below are sample disassembled instructions:

Listing 1: Sample Input

```
007201b3
00720863
00c0006f
00533623
100004b7
00c50493
```

Listing 2: Sample output

```
add x3 x4 x7
beq x4 x7 L1
jal x0 L1
sd x5 12(x6)
lui x9 0x10000
L1:addi x9 x10 12
```

5 Correctness

The correctness of the code has been verified both logically and also by testing with various input. I have verified with the output of RIPES simulatior and also tested edge cases and each instruction independently.

6 Conclusion

The code is easy to use and efficient. The test cases verified are attached in the zip file.