RISC-V Assembler

AI24BTECH11031 - Shivram S

August 31, 2024

Contents

1 Introduction

As part of the CS2323 Computer Architecure course, we had to develop an assembler for the RISC-V instruction set architecture. This report outlines the design of the assembler and the major design decisions taken during its development.

2 Planning

Before I could begin writing code, I had to outline the structure of the assembler and get a high-level understanding of what it would look like.

2.1 Parsing

The first step in converting assembly code to machine code is to read the program and parse it into semantic elements such as instructions and labels.

Since there problem statement makes several assumptions about the code (such as having one instruction per line, one space after each comma, etc.), it is possible to parse the code using only simple functions like `scanf`.

However, I decided to do for a more structured approach, using separate tokenization and parsing steps, to give practical utility to the assembler.

2.2 One-pass versus Multi-pass

Assemblers can be broadly divided into two categories - one-pass assemblers and multi-pass assemblers.

A one-pass assembler processes the source code once, and patches earlier output based on later definitions of symbols. On the other hand, multi-pass assemblers create a symbol table in the first pass, and then use it to generate code in the second pass.

It is common to see assembly code that refers to labels defined later in the program:

```
    beq x6, x0, Exit
    ; Some code
Exit:
    add x0, x0, x0
```

While scanning the program, when we reach line 1, we have no idea where `Exit` is defined and what address it corresponds to.

If the assembler followed a single-pass design, we would need to maintain a table of locations where a label is referenced, and then update the earlier location when we encountered the label.

To avoid this complication, I decided to use a two-pass design. The first pass creates a list of labels and their addresses, and the second pass emits the machine code for each instruction, substituting labels for their addresses.

2.3 Error Handling

The program that the assembler receives as input need not be correct. It can have syntactic and semantic errors, and the assembler is expected to find and report these errors.

Unlike other popular languages such as C++ and Java, C lacks an inbuilt exception-Handling system. There is no single way to handle errors

3 Design

3.1 File Strucure

3.2 File Strucure

3.3 File Strucure

4 Features

5