CS2323 - Lab 3 - RiscV Assembler Report

Shiven Bajpai - AI24BTECH11030

August 31, 2024

Contents

1 Preface

In this project I aimed to create an assembler that was fully RV64I Compatible. I wanted to implement pseudo instructions as well but due to time constraints I decided not to. I have learnt a lot over the course of this project and as a fresher I must say it was probably the most exciting thing I have worked on in college so far.

In the end I had an assembler which had the following capabilities:

- Supports all RV64I Base Integer Instructions
- Is able to correctly handle whitespace and comments (comments may begin with # or ;)
- Is capable of handling literals in Decimal, Hex, Octal and Binary formats (Follows C convention for syntax)
- Gives clear error reporting including the Line number as well as type of issue.
- Has basic command line options for specifying custom file names and enabling debug information

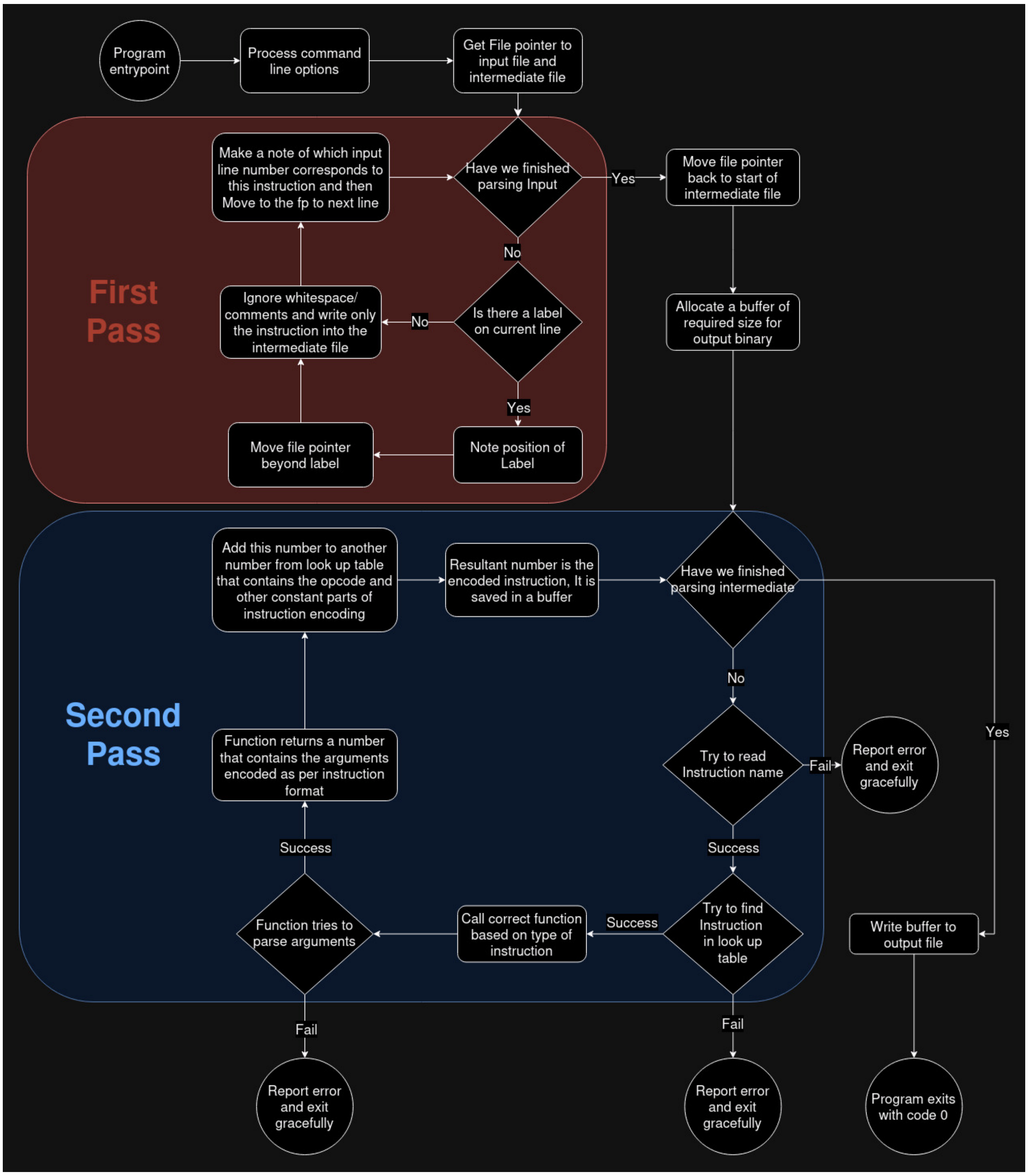
2 Approach to the Problem

The way I approached the problem was that I identified 3 simple steps that had to be performed for every instruction in the input:

1. Read the instruction
2. Encode it as per applicable format
3. Write the binary code to the output file/buffer

At first It seemed as though I could achieve this in a single loop iterating over all instructions. However there was the issue that if I were parsing the input sequentially it would not be possible for me to resolve a Label in one instruction that is declared on a later line. To remedy this problem I could've chosen to defer the processing of all instructions including labels to happen after processing all other instructions, however I chose a simpler approach where I performed two passes. One where I identified labels and the other where I actually processed the instructions.

I also chose to remove unnecessary whitespace in the first pass as well. Not only does this simplify the processing in the second pass, but also it provided me with an opportunity to create an intermediate cleaned up version of the code without labels and extra whitespace. The assembler writes this intermediate version out to a separate file called "cleaned.s". The motivation for this is that, inspired by the RIPES simulator, I wanted to have my simulator (in the next project) capable of showing a cleaned up version of the code separated by labels just like RIPES has in the middle pane of the Editor tab.



Program entrypoint
Process command line options
Get File pointer to input file and intermediate file
First Pass
Have we finished parsing Input
Yes
Move file pointer back to start of intermediate file
No
Make a note of which input line number corresponds to this instruction and then Move to the fp to next line
Is there a label on current line
No
Ignore whitespace/ comments and write only the instruction into the intermediate file
Yes
Note position of Label
Move file pointer beyond label
Allocate a buffer of required size for output binary
Second Pass
Have we finished parsing intermediate
Yes
Add this number to another number from look up table that contains the opcode and other constant parts of instruction encoding
Resultant number is the encoded instruction, It is saved in a buffer
No
Try to read Instruction name
Fail
Report error and exit gracefully
Function returns a number that contains the arguments encoded as per instruction format
Success
Success
Function tries to parse arguments
Call correct function based on type of instruction
Success
Try to find Instruction in look up table
Write buffer to output file
Fail
Fail
Report error and exit gracefully
Report error and exit gracefully
Program exits with code 0

Figure 3: Program Flowchart

3 Code Implementation

The overall control flow of my program can be summarized as follows:

Note that to keep it clean, some checks for overflowing buffer size while reading instruction/Label names have been omitted from the flowchart. Essentially, if an instruction/label name is longer than 128 chars the program will declare it invalid and exit gracefully.

3.1 Noting Position of labels

In the first pass, to store information about labels as they are encountered, I use a struct called label_index which is defined in `index.c`. This struct is essentially a data structure that manages a pair of key and value arrays. Some helper functions are defined that take care of memory management and the function `int label_to_position` performs a linear search over the data to convert labels to addresses

3.2 Instruction lookup tables

Information about the instructions is hard coded into the program in an array of structs containing the instruction name, a number which has just the constant bits set, and an enum value indicating which function is to be called to parse arguments. While parsing instructions this enum value is checked via a switch statement to call the right function. The lookup table is present in `translator.c`

I would like to note that for a large set of instructions it would be considerably faster to use a hashmap for such lookup operations, but since the size of instructions here was relatively small I chose to skip using a hashmap for simplicity.

3.3 Parsing Arguments

Arguments are parsed by individual functions that are uniquely defined for different groups of instructions which have different formats of arguments, each of these unique functions simply calls a generalized function `int* parse_args` with the right parameters based on the format, then performs some extra checks and transformations on the returned values to encode the arguments as per the instruction format and then returns this number.

3.4 Error Handling

Whenever an error is encountered, the program has two aims: Free memory, report the error and exit gracefully

The error reporting is done immediately in whichever function the error is encountered. Then although it would be sufficient to simply exit the program then and there and let the OS de-allocate memory, It is best practice to always deallocate memory, In my program all functions return and indicate to the caller in some way or another that the function failed, which causes to caller to also free memory and return until the main function is reached which is where the program actually always exits.

Since we are reading a cleaned up version of the input in the second pass, the line numbers may not match those in the input, so there is also a struct called `line_mapping` of custom type `vec` that is passed around that contains the mapping between instruction numbers and the line on which they appear in the input file

4 Testing

For testing, I used a small set of test cases comprising of:

- The examples given with the problem statement

- My previous Lab assignments under this course
- A single program containing all instructions

Testing is managed by a simple test harness written as a bash script in the tests directory. You can run all tests by running `make run` in the project root
To have a "correct" output to test against, I needed to use another known correct compiler. Since RIPES does not allow you to save the hex output I used an online riscv assembler at https://riscvasm.lucasteske.dev/ to generate a correct version to test against To run the tests one may use the 'make run' command from the root directory.

Appendices

A Usage Guide

How To build

1. Run ‘make build‘ to compile the project, binary is produced in `/bin` called `riscv_asm`
2. Run the binary. A temporary file called ‘cleaned.s‘ will be created and the machine code output shall be in ‘output.hex‘ in the same directory by default. See the next section for how to specify input/output files

Command line switches

```
‘-d‘
‘--debug‘
```

Runs the assembler in debug mode, providing more verbose output as it runs

```
‘-i <filename>‘
‘--input <filename>‘
```

Specifies a file for input, defaults to input.s if not specified

```
‘-o <filename>‘
‘--output <filename>‘
```

Specifies a file for output, defaults to output.hex if not specified

B Project File Structure

```
.
+-- root
    +-- bin/                    (Contains output binary when built)
    +-- build/                  (Object files are placed here)
    +-- report/
    |   +-- AI24BTECH11030.tex  (The latex file for project report)
    +-- src/
    |   +-- index.c             (Defines label_index struct)
    |   +-- index.h
    |   +-- main.c              (Main file)
    |   +-- translator.c        (Contains most functions and hard-coded data
    |   |                        regarding instruction parsing)
    |   +-- translator.h
    |   +-- vec.c               (Defines the vec struct)
    |   +--  vec.h
    +-- tests/
    |   +-- cases/              (The actual tests cases are stored here)
    |   |   +-- (...)
    |   +-- outputs/            (Program output from running tests goes here)
    |   +-- test.bash           (Basic Test harness used for running tests)
    +-- .gitignore
    +-- LICENSE
    +-- README.md
    +-- report.pdf              (Project report)
    +-- Makefile
```