CS2323 - Lab 3 - RiscV Assembler Report

Shiven Bajpai - AI24BTECH11030

Contents

I. Preface

This Report outlines the process of creation of this Assembler. The aim was to have it work with all base class instructions and have it be useable for a GUI based RISCV simulator. Pseudo instructions were left out of this assembler. There is however a section at the end briefly detailing how one may add pseudo instructions to this assembler.

II. Breaking down the problem

At its core, parsing assembly instructions involves 3 simple steps:

1) Read the instruction
2) Convert it to binary
3) Write the binary code to the output file/buffer

Since every instruction is processed sequentially, The entire process looks as thought it can be implemented in a single loop. However it is not so. Say there is a jump or branch instruction with a Label that is declared on a later line, if we are reading and converting line by line, we wouldn't have reached that label yet and thus wont be able to convert the current instruction to binary.
To solve this issue we use two passes. The first pass looks for Labels and notes down the address to which each label points. Then in the second pass we actually convert the instructions, which can be now be down without any issues since we already know the position of all labels in the code.
For convenience, we also remove excess whitespace in the first pass.

Theoretically this could be also be done in a single pass by deferring the conversion of all jump/branch statements to the end of the pass. However my assembler goes with two passes for simplicity
The rough control flow can be written as follows:

1) Load the file
2) Read every line, for every line
   a) If there is a flag declared, note down the flag's position
   b) Remove whitespace/flags and write it to a temporary file
3) Write this temporary cleaned up code to a file 'cleaned.s'
4) Read every line from the temporary file, for every line
   a) Read the name of the instruction
   b) If the instruction is invalid, exit gracefully
   c) Based on the instruction, try to read the arguments and check for validity
   d) If the arguments are invalid, exit gracefully
   e) Convert to binary and write to a temporary buffer
5) Write the full binary code from the temporary buffer to the output file 'out.hex'

Although step 3 is not necessary for the objective of this program, it is included since the assembler is designed with the intention of being integrated with a GUI simulator for the next project of this course so having a cleaned up version of the code would be useful for then. The temporary buffer between the second pass and final writing to binary is to avoid a scenario where an incomplete binary is produced as a result of a program being partially assembled and then having the assembler encounter an error causing it to stop and leave behind a binary with unpredictable contents.

III. Code Implementation

This section only touches on the parts key to assembly to binary conversion, and leaves out mundane details like file handling and the fine details of memory management.
For simplifying the code, 2 data structures have been defined `vec` and `label_index`. vec is essentially a glorified array that handles expanding its memory allocation on its own. `label_index` is a collection of label-position pairs that are later searched when trying to resolve labels during the second pass

A. The First Pass

In this pass the code goes through the input charachter by charachter. A series of different flags are used to keep track of state. We essentially just remove excess whitespace and copy the rest to the output buffer. If a colon is encountered, the program discards the text from the current line before the colon, treating it as a label and noting down its position in a `label_index`. Additionally, the program builds up an array of the line number on which each instruction was written, this helps it know which line number to report in an error message if an error is encountered in the second pass, since the line number of an instruction in the output of the first pass may be different from the line number in the input due to removal of whitespace

B. The Second Pass

1) Parsing the instruction: Thanks to handling unnecessary whitespace in the first pass, to read an instruction the second pass can simply keep reading chars into a buffer until it encounters a space. The program then tries to look-up information about this instruction in the list of instruction it has. If it does not find a matching instruction, it gives an error and exits.unpredictable If the program finds a matching instruction, It is able to look-up information regarding the encoding format of the instruction, the format of the arguments as well as the constant parts of the instruction's enconding (opcode, funct3, funct7 etc.)

2) Parsing the arguments: On performing the lookup the program gets a value of type `enum instruction_type`'. This is matched in a switch-case block and a parser function for that format of arguments is called. This function just calls another generalized argument parser with the right parameters, performs any extra validation if applicable (like 12-bit limit for immediate values of many instructions) And then it encodes the arguments as per the format and returns that as a long int.

3) Calculating binary: The encoded arguments from the parser function is just combined with the constant part from looking up the instruction in section III-B1. The two numbers contain different parts of the full encoded instruction and are combind with a bitwise OR operation. After this the program writes the output to a buffer that holds the output until all instrutions have been encoded.

C. A note on logistics

A lot of function calls occur during this process. For the sake of efficiency, instead of creating buffers to send the arguments to the argument parsing functions, we pass a pointer to the file pointer reading the intermediate 'cleaned.s' file. The argument parser simply increments the file pointer as it reads the arguments, after which the pointer ends up at the next line ready for the next iteration of the loop to attempt to read another instruction.

IV. Testing