Assignment 1 Project Report

Li Jiaqi   120090727   School of Data Science

1. Overview

This project aims to construct a MIPS assembler, which translates the assembly language file into a file containing binary instructions (machine codes) used by mips hardware.

This project, in a big picture, consists of 2 phases.

- Phase 1 scans through the assembly language file, assigns memory location for each valid instruction, retrieves and stores the labels with their corresponding addresses.
- In phase 2 we're supposed to scan through the assembly language file second time and match the instruction types, registers, and labels with their corresponding machine code formats. To prevent repetitive jobs, in this procedure a processed temporary file generated in Phase 1 will be used.

Additionally, the step of conversion from assembly code to machine code requires 2 tables: a **label table** that looks up a label's address, a **register table** that can convert a register name like "$zero" to number. And because different instructions have different complicated formats, each instruction should be separately configured.

2. High-level Implemetations

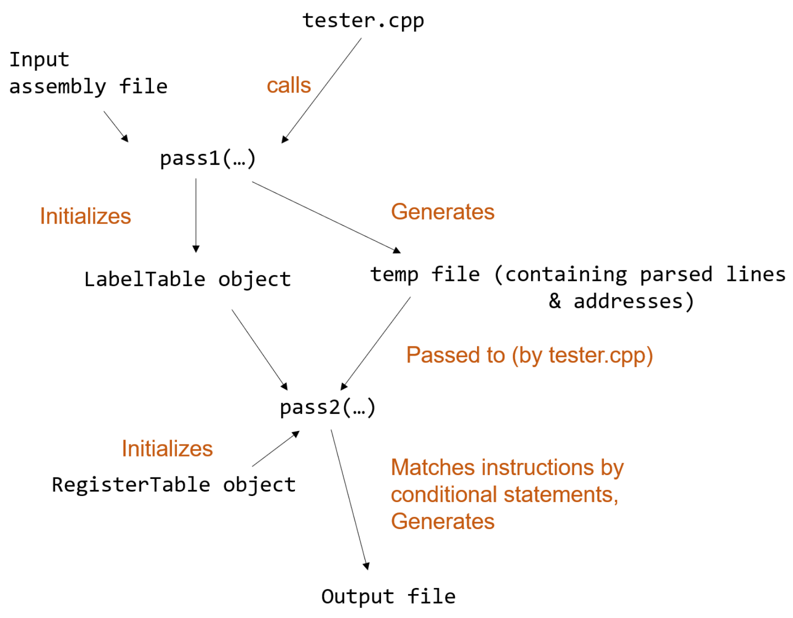
tester.cpp
Input assembly file
calls
pass1(…)
Initializes
Generates
LabelTable object
temp file (containing parsed lines & addresses)
Passed to (by tester.cpp)
pass2(…)
Initializes
RegisterTable object
Matches instructions by conditional statements, Generates
Output file

Figure 1. Hierarchical representation of high-level implementation

Procedures: `Pass1` and `Pass2`

Based on previous reasoning, the project is divided into the 2 phases, each represented in file `Phase1.cpp` and `Phase2.cpp`. In each file the problem is broken down into a major function and several auxilary functions that help the major function work.

- In `Phase1.cpp` the major function is `pass1()`. It opens the assembly language file, scans through all the labels and returns a `LabelTable` object storing all the labels and their addresses.
- In `Phase2.cpp` the major function is `pass2()`. It gets an output filename, a temporary file name. By scanning the temporary file (generated in Phase 1), it matches the instructions, registers and labels with their corresponding formats or codes. The machine code output is written in the ouput file.

Data Structures: `LabelTable` and `RegisterTable`

Apart from the 2 phases for execution, 2 additional classes are set up for managing data structures.

- The `LabelTable` class is a table for storing and retrieving the labels with their corresponding addresses. It has an `add()` function for adding an item. After adding labels, the `LabelTable` object can be passed from `pass1` to `pass2`. A label's address can be retrieved by calling the `get()` method.

- The `RegisterTable` class is a table for storing all the register names and their register numbers. Similar to `LabelTable`, it should have an `add()` and a `get()` method. A difference from the `LabelTable` class is that the mips register names and numbers are already known, so the `RegisterTable` is initialized with all the data.

Converting assembly code to machine code

Tokens

To process an assembly language instruction, each instruction is separated into tokens.

Before processing:

> addi $sp, $sp, -12

After:

> {"addi", "$sp", "$sp", "-12"}

In this way, we can match the first token for the instruction type, and then match every other token.

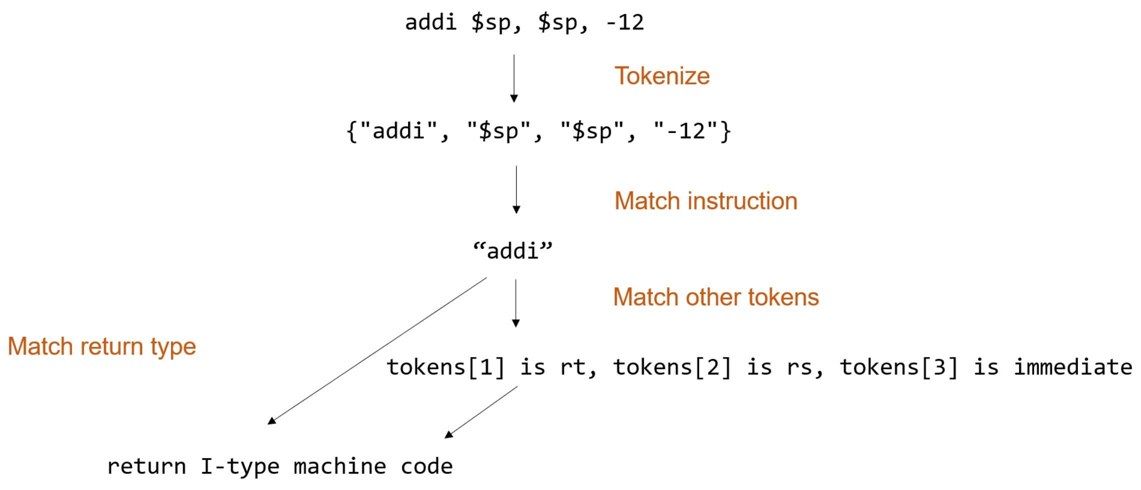
addi $sp, $sp, -12
Tokenize
{"addi", "$sp", "$sp", "-12"}
Match instruction
"addi"
Match other tokens
Match return type
tokens[1] is rt, tokens[2] is rs, tokens[3] is immediate
return I-type machine code

Figure 2. Workflow of processing assembly code

3. Implementation Details

Implementing `LabelTable`

- The `LabelTable` structure is implemented with a **linked list**. An `Item` structcture is defined for each item, containing the label name, its address and a link pointer to the next `Item`.
- For quick accessing, a pointer named `firstItem` points to the first item, a pointer named `lastItem` points to the last added item. The class comes with two methods: `add()` and `get()`. When calling `add()`, it initializes a new `Item` linked to NULL, and links the current `lastItem` to it, and replaces the new `lastItem`. When calling `get()`, it traverses each of the `Item` in the table to match the label.

Implementing `RegisterTable`

- The `RegisterTable` structure is implemented with the **vector** class in STL, with the index in the vector being the register number.

Implementing `pass1()`

- As mentioned before, the `pass1()` function opens the assembly language file, scans through all the labels and returns a `LabelTable` object storing all the labels and their addresses.
- The most challenging task in pass 1 is to split a line of assembly code into useful blocks of tokens. To achieve this, an auxilary function to pass 1 is `parseLineToTokens()`. The function scans each character in the scanned line. The criterion of spliting the line is byidentifying the "separator", like space and comma. Its return value is a `vector<std::string>` object.

Implementing `pass2()`

- The `pass2()` function matches the instructions, registers and labels and generates machine code output to the output file. This whole process is complicated and 7 auxilary functions are implemented for it: `getRTypeCode()`, 2 overloaded `getITypeCode()`, `getJTypeCode`, `stringToInt()`, and `toBinary()`. `getRTypeCode()`, `getITypeCode()` and `getJTypeCode()` does simple concatenation jobs to generate different types of machine codes.
- Because each mips instruction may correspond to very different token order; and mips has 3 types of machine codes. To easily control the handling of each instruction, a pile of "if" statements are used to check the instruction type and customize handling.

For example:

```
if (instrc == "addi"){
        opcd = 0x08;
        rt = regTable.get(tokens[1]);
        rs = regTable.get(tokens[2]);
        immediate = stringToInt(tokens[3]);
        return getITypeCode(opcd, rs, rt, immediate);
}
```

About tester.cpp

This file is a slight modification of the original tester.c. Firstly, its file extension is changed to .cpp for consistency in the project. Some c-string file name arguments are converted to std::string.