The George Washington Universityshort line



**Computer System Architecture (CSCI 6461\_10)**

Project Part 0 Design Notes

**Submitted By:**

Group 9

Abhiyan Sainju

Boxin Yang

Udaykiran Kalyanapu

**Submitted to:**

Professor Morris Lancaster

# Project Part 0: Assembler Project Documentation

## Project Overview

The part 0 (assembler) project aims to convert each instruction into its 16-bit representation. This document outlines the design notes, methods used, and the functionality of each component in the project.

## Design Notes

The part 0 (assembler) project aims to convert each instruction into its 16-bit representation. This document outlines the design notes, methods used, and the functionality of each component in the project.

* **The opcode to Binary Map**: We created and used a hash table for storing opcodes and mapping opcodes to their binary representation. Using a hash table facilitates quick changes during assembly.
* **Reading Instruction**: In our program, instructions are read from a file, parsing from right to left. We use space used to distinguish and split instructions and operands.
* **Binary to Octal Method**: Our program has a method named *BinaryToOctal* which converts a binary input to its octal representation from the smallest digit. But our approach may not be perfect for all situations.
* **Format and Convert**: The *formatAndConvertInstruction* function handles opcode and other parts, converting them into binary strings with padding zeros as necessary, then converting to octal.
* **Handling Different Instructions**: We have categorized instructions into various categories according to the project description (e.g., LoadStore, Transfer). Each category has its handling logic within separate functions.
* **Address in Parsing**: Parsing starts by checking if the first instruction is "LOC." If not, the address is set to 6, reserving the first five addresses.
* **Writing to Files**: Our program uses two methods to handle writing results to files: *writeListingFile* for a list of instructions with original lines, and *writeLoadFile* for the load file.

## Functions Overview

The following is an elaborate description of the functions used in the Assembler project.

1. **readFile**

The *readFile()* method is used to take a text file as input and read each line from the given text file. The method then adds a line as an instruction to an *ArrayList* named *Instructions* until a null line is found. If any error occurs during this process, the program throws an *IOexception* error.

Why did we use ArrayList to store instructions?

* Because it is easy to manipulate - add, get, remove.
* It provides dynamic sizing and indexed access

1. **loadDict**

The function *loadDict()* is used to load a HashMap named Ins2Bi which stores all the instructions and their respective opcodes in a key-value pair. Both key and value are strings. The key is the instruction and the value is its opcode.

Eg: Ins2Bi.put("STR","000010"); → STR is key whose value is 000010 in the HashMap

1. **parse**

Then we have a method named *parse()* which parses instructions and stores them into addresses. We have reserved the first 5 addresses for special cases described in the Project Description. These addresses are:

**Reserved Locations**:

Memory Address                Usage

0                              Reserved for the Trap instruction for Part III.

1                              Reserved for a machine fault (see below).

2                              Store PC for Trap

3                              Not Used

4                              Store PC for Machine Fault

5                              Not Used

1. The first thing we do in the *parse()* method is to check if the first instruction in the input file is “LOC”. If yes, we initialize the address = 6 (as the first 5 addresses are reserved Locations)
2. Then we iterate through every instruction in the ArrayList **instructions**.

For each instruction, we split the instruction whenever whitespace is found in the instruction using the split function in JAVA to convert the instruction which is a string type to an array. For example, if the instruction is "ADD R1, R2", calling split("\\s+") on this string will result in the array ["ADD", "R1,", "R2"], because the whitespace character between words "ADD", "R1,", and "R2" is used as the delimiter.

For each instruction, we check if the instruction is a LOC. We do this by checking the first array element we created using split. I.e. tempArray[0]. If yes, we set the address as given by the instruction. We get the array using the second element in the tempArray i.e. tempArray[1]. For example, “LOC 9” instruction sets the address to 9

Else, If the instruction is “Data”,

1. We check if the instruction's second element, i.e., tempArray[1], is END. If yes, we set the second element of the array to “01024” instead of END. (END has value “01024”)
2. For all the instructions, we add them to an ArrayList named **parsedInstru** with type String**.** As the instruction is Data, it will have two parts, the first will be the address in octal, and the second will be the value to be stored in that address.
3. We first format the address as **String.format("%06o", address)**. This formats the address variable as a string with a width of at least 6 characters, padded with leading zeros (%06), and represented in octal format (%o). So, it ensures that the address is represented as a string of 6 characters in octal format. This is the first part of the instruction.
4. Secondly, we we format the tempArray[1] which gives the value to be stored in the address in the same way. String.format("%06o", Integer.parseInt(tmpArr[1])): This part parses the second token of the instruction (tmpArr[1]) as an integer using Integer.parseInt(). Then, it formats this integer as a string with a width of at least 6 characters, padded with leading zeros (%06), and represented in octal format (%o). This ensures that the data value is represented as a string of 6 characters in octal format.
5. Otherwise, if the instruction is not data or loc, we determine which category is the instruction. The categories are:

* Miscellaneous instructions
* Load and Store instructions
* Transfer instructions
* Arithmetic and Logical instructions
* Register Operation instructions
* Shift and Rotate instructions
* I/O instructions
* Floating Point and Vector instructions

Each category has its logic to handle instructions according to the project description.

1. **writeListingFile**

Outputs the ListingFile as a text file.

1. **writeLoadFile**

Outputs the LoadFile as a text file.

## Conclusion

The assembler project (project part 0) provides a comprehensive solution for converting instructions into binary representations, addressing various aspects of parsing, categorization, and output generation.