Report on the assembler-like Python program (Bonus Project) CSE 311: Computer Organization

Sara Basheer Mohamed
ID: 120220035
CSE 311 - Computer Organization
Dr. Hossam Kassem

January 7, 2025

1 Introduction

Here is an explanation of my implementation of the assembler, following the concepts learned in the course. The goal of this project is to convert assembly code into machine code, specifically the symbolic instructions and data into binary format. The assembler follows a two-pass process: the first pass generates a symbol table, and the second pass translates the code into binary instructions.

2 Program Overview

The program consists of two main passes:

- **First Pass**:The program reads through the assembly code to identify labels and build a symbol table. It also tracks the location of each instruction and data.
- **Second Pass**: Here, the actual machine code is generated. The assembler translates the instructions into their binary equivalents and replaces any labels with the correct memory addresses.
- **Instructions included**: The instructions expected to appear in the assembly code are the ones shown in the following table

The assembler is designed to read assembly code from a file named `asm.txt` and output the resulting machine code to a file named `Machine_Code.txt`.

Symbol	Hexadecimal code $I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Figure 1: Basic Computer Architecture. Adapted from *Computer System Architecture* by Morris Mano, 3rd Edition, Prentice Hall, 2006.

3 Detailed Explanation of the Code

3.1 Assembler Class

The code is short, but for organization and readability purposes, I preferred to use OOP approach. The main class is `Assembler`. It contains several methods that handle different parts of the assembly process:

- **Initialization**: The `init` method (constructor) initializes the symbol table, location counter, and a dictionary of instructions with their binary representations. The dictionary is just the mapping of the symbolic instructions to the corresponding binary code as in Table 1.

- **First Pass**: The `first_pass` method performs an initial scan of the assembly code to build a symbol table and determine the memory layout. During this phase, it performs the following operations:

- **Initialization**: The location counter is initialized to 0. This counter is used to track the memory address of each instruction or data item.
- **Processing Each Line**: The method reads the assembly code line by line, ignoring comments and blank lines.
- **Handling the ORG Directive**: If an `ORG` directive is encountered, the location counter is updated to the specified hexadecimal address. This determines where subsequent instructions or data will be placed in memory.
- **Identifying Labels**: If a line contains a label (indicated by a comma, e.g., `LABEL,`), the method extracts the label, associates it with the current value of the location counter, and stores this mapping in the symbol table. The label's address is stored as a 12-bit binary value.
- **Tracking Memory Locations**:
  - If the line defines a data declaration (e.g., `DEC`), the location counter is incremented to account for the space required to store the data.
  - For instruction lines, the location counter is incremented to account for the memory needed to store the instruction.
- **Handling the END Directive**: If an `END` directive is encountered, the method stops processing further lines, as the assembly code has been fully read.

The symbol table created during this pass serves as a lookup for label addresses in the second pass, ensuring all symbolic references in the code are resolved to concrete memory locations. .

- **Decimal Conversion**: The `convert_decimal` method converts decimal values (including negative numbers) to 16-bit binary format to handle the pseudo instructions of DEC.

- **Second Pass**: The `second_pass` method processes the assembly code a second time. Its primary purpose is to generate binary machine code for each instruction or data item. During this phase, it performs the following operations:
  - **Initialization**: A list is created to store the final binary instructions, and the location counter is reset to 0.
  - **Handling the ORG Directive**: If an `ORG` directive is encountered, it sets the location counter to the specified hexadecimal address. This determines where subsequent instructions or data are placed in memory.
  - **Processing Labels and Data Declarations**: If a line contains a data declaration (e.g., `DEC`), the value is extracted and converted into its 16-bit binary representation using the `convert_decimal` method. These are paired with their respective memory addresses and stored.
  - **Converting Instructions to Binary**:
    - The method reads each instruction and separates the opcode from its operand (if present).
    - The opcode is converted to a 4-bit binary string using the instruction dictionary.

    - If the instruction uses indirect addressing (indicated by the `I` prefix), the second hex digit of the opcode is used, and the `I` is removed from the operand.
    - For instructions with operands, the operand is resolved to its memory address using the symbol table. If the operand cannot be found, a default address of `000000000000` is used.
    - The binary opcode and the resolved operand address are concatenated to form the complete binary instruction.
  - **Incrementing the Location Counter**: After processing each line, the location counter is incremented to point to the next memory address.
  - **Returning the Machine Code**: The method outputs a list of tuples, where each tuple contains the memory address (as a 12-bit binary value) and its corresponding binary instruction or data.

  This phase ensures that all instructions are properly encoded in binary, with labels replaced by their actual memory addresses as defined in the symbol table during the first pass.

4 Conclusion

This assembler program successfully converts assembly code into machine code. It handles both instructions and data, ensuring that labels are correctly mapped to memory addresses. The code is structured to allow for easy extension and modification, however, it is tailored to the 24 instructions of the Basic Computer we were studying throughout the semester; 6 for memory-reference instructions, 12 for register-reference instructions, and 6 for input/output instructions.