Computer Organisation

ASSEMBLER – PROJECT 1 REPORT

Suchet Aggarwal
2018105

Ujjwal Singh
2018113

Implementation

Our implementation of the assembler is written in Python, and the main code comprises of two passes, where the input code (written in assembly) is given in as input, and the corresponding machine code is produced as a file named output.out Simple Implementation of an Assembler for an Accumulator Architecture Machine that handles basic conversion and variable addressing

How the Assembler is Run:

```
C:\Users\shash\Desktop>python3 assembler.py input.txt
```

Instruction Format:

1. **The program must begin with a START statement that may or may not specify the starting address of the location counter; in case not specified, the starting address of the location counter is taken as 0.**
2. **The Instructions that follow should contain opcodes (and their corresponding operands) in case the opcode is incorrect; the assembler throws an error.**
3. **The labels should be defined by using a semicolon (:) and the syntax for the same is:**

<label> : <opcode> <operand>

*****There is a whitespace between the label and the semicolon as well as between the semicolon and the opcode***

4. **The variables used in the program are to be explicitly defined, and since these have a fixed length of one word these are defined using the DW directive**
5. **The program must also have an END statement signaling the end of the code**
6. **Anything below the END statement is ignored.**

The OUTPUT format:

1. The assembler writes off the output in the file named “output.out,” while the rest of the temporary files, symbol tables, and label tables are all dumped in temporary files that are removed during the clean-up process once the assembly process is finished.
2. The “output.out” file contains the binary equivalent of the code given in assembly language.
3. The first string of bits in each of the lines depicts the value of the address in virtual address space
4. The virtual address is followed by a semi-colon (:)
5. The string following the semi-colon (:) is the direct translation of the assembly equivalent given in as the input.
6. At each point, wherever a label/variable occurs, it is replaced by the corresponding address it will have in the virtual address space created by the assembler.

```
1  00001010 : 0000
2  00001110 : 1000 10000110
3  00011010 : 1000 10010010
4  00100110 : 0001 10000110
5  00110010 : 0100 10010010
6  00111110 : 0110 01100110
7  01001010 : 1001 10000110
8  01010110 : 0000
9  01011010 : 0101 10000010
10 01100110 : 1001 10010010
11 01110010 : 0000
12 01110110 : 0101 10000010
13 10000010 : 1100
14 10000110 : 01111000
15 10010010 : 01111101
```

How the assembler works:

The given code implements a two-pass assembler,

Working of the PASS ONE

1. **The location counter is initialized**
2. **Label table and Symbol Tables are initialized**
3. **The input program is processed line by line, wherein only the non-commented lines are processed, and the comments are ignored**
4. **If the given line:**
    a. **Is a Start Statement, then the address given/specified is used starting value of location counter**
    b. **Has a label then it is put in the label table**
    c. **Has a Variable then it is put in the symbol table**
    d. **Has opcode, then the opcode is verified whether it is valid, if not an error is thrown, and assembly process is terminated.**
    e. **If the given opcode is correct, then its corresponding operand is read (depending on whether it takes in any operand or not), else an error is thrown.**
    f. **If it is a variable declarative, then its symbol is checked whether it is present in the symbol table,**
    g. **If it is a label, then it is checked whether it is present in the symbol table,**
5. **If the code has variables/ symbols that have not been defined but have been used, then the assembler throws an error**
6. **If the code does not encounter any END statement, then the code throws an error, and if it does have any then everything below the END statement is ignored.**
7. **The code prepares several temporary files,**
    a. **Label/Variable Address Table**

```
1  L1 118
2  L2 154
3  A 120
4  B 125
```

b. A file that contains all the lines except the comments

```
10 CLA None
22 INP A
34 INP B
46 LAC A
58 SUB B
70 BRN L1
82 DSP A
94 CLA None
106 BRZ L2
118 DSP B
130 CLA None
142 BRZ L2
154 STP None
166 A DW 120
178 B DW 125
```

c. Symbol Table

```
A 22
B 34
L1 70
L2 106
```

Working of the PASS TWO

1. Since the assembler runs this pass only if all the symbols, labels, and opcodes have been declared the code can directly processed.
2. The Temporary file created during the first pass is now processed line by line
3. The labels and variables are replaced by their corresponding addresses
4. The opcodes and the addresses are converted to their binary equivalents, and all changes are written off to output.out
5. Clean-Up is done, removing all temporary files and symbols and label tables.

Errors Handled:

All Errors are reported along with the line number in which that error occurs on the terminal/Command Prompt in which the given python script is run.

1. **Invalid opcode used: the opcode specified does not exist or is immediately preceded/followed by any other symbol (mostly a semi-colon (:))**
2. **A line contains multiple opcodes: a line contains more than one opcode which is not permissible**
3. **An Opcode is supplied with too many/ too few arguments (more than one in our case)**
4. **An Opcode that does not take any operand is supplied with an argument/operand. Thus an error is raised signaling opcode supplied with too many arguments**
5. **The START statement contains an address beyond 256 (maximum value for 8-bit address)**
6. **Multiple Declaration of a Label/ Variable.**
7. **No initial value provided at the time of variable declaration**
8. **Missing END statement**
9. **Missing START statement**
10. **Symbol/Variable has been used but has not been defined in the program.**