CSE112
CO ASSEMBLER
PROJECT-01

GROUP MEMBERS :
TARUN KHANNA 2019339
RAGHAV BHALLA 2019379

INTRODUCTION

In this project we have designed a 2-Pass Assembler using the programming language PYTHON. In this project we have taken input using a text file and used our designed 2-Pass Assembler to generate a machine code to the corresponding input text file.

ASSUMPTIONS

For this 2-Pass Assembler we have the following Assumptions :

1. CLA and STP take no operands/arguments

2. LAC, SAC, INP, DSP take 1 operand which is an address

3. ADD, SUB, MUL, DIV take 1 operand which can be a literal or an address

4. BRZ, BRN, BRP take 1 operand which should be a valid label

5. If DIV is called, the quotient and remainder get stored in Variables R1 and R2 which can be used in the code later. If DIV is called again, then the value in R1 and R2 will get overwritten.

6. SAC and INP can take undefined addresses as an argument. Rest of the opcodes cannot. (If no valid value is present in an address, it is treated as an undefined address)

GENERAL

1. Clear accumulator (CLA) is used to reset the accumulator. No address or value is present in the accumulator.

2. Execution of the assembly program stops after the end statement. The code written after the end statement shall not be executed.

3. START loads the instructions from the specified address. If no address is specified, the instruction set is loaded normally.

4. Label name **cannot** be an opcode name.

5. Comments can be added with the help of **; or /** .

6. Literals & Constants can be specified between single quotes "'". Example ADD '3'

7. Variable names and label names can be alpha-numeric. But, the same variable cannot be used as a label name and vice versa.

8. Literals can be of any value, contiguous memory spaces are allotted.

9. If the size of the literal uses more than 1 memory space, then only the first is written in the machine code. Exact memory mapping can be seen from the LiteralTable.

10. The maximum memory size is 256 words, which can be stored using 8 bit addresses. If the program size is greater than this, the assembler will throw an error and terminate.

11. Total instruction length is 12 bit: 8 bit instruction address + 4 bit opcode

METHODS & WORKING OF THE CODE

FIRST PASS

- Open the Input Text file instruction by instruction and an Output Text file for writing machine code .

1. We first check for comments and remove them using string slicing and removecomment(instruction) function.

```
def removecomment(instruction)
'''Input : it takes an instruction as an argument.
Return : it removes comments from the instruction and returns it..''
```

.

1. First go instruction by instruction and check for labels using **checklabel(instruction)** and append the label to the label table if it has not returned -1.

```
def checklabel(instruction)
'''Input : it takes an instruction as an argument.
Return : if the instruction contains the label it returns the instruction else it returns -1.''
```

1. Then, it checks for the instruction if it contains a literal if not label using **checkliteral(instruction)** and appends it to the literal table if it has not returned -1 **.**

```
def checkliteral(instruction)
'''Input : it takes an instruction as an argument.
Return : if the instruction contains the literal it returns the instruction else it returns -1.''
```

1. If it is neither a label nor a literal it is either an address or a symbol **.** So it appends it to the symbol table.

SECOND PASS

- Open the Input Text file instruction by instruction.

1. We first removed all the comments before the first pass of the assembler using **removecomment(instruction).**

1. First go instruction by instruction and check for labels using **checklabel(instruction)** and find the address of the label from the label table if it has not returned -1.

1. Then, it checks for the instruction if it contains a literal if not label using **checkliteral(instruction)** and finds the address of the literal from the literal table if it has not returned -1 **.**

1. If it is neither a label nor a literal it is either an address or a symbol **.** So we find the address from the symbol table.

ERROR HANDLING

For this 2-Pass Assembler we have assumed the during execution the following Errors can occur :

1. **DUPLICATE LABEL ERROR**

   If a duplicate label has been given then an error would be generated.

2. **LABEL NOT DEFINED ERROR**

   If a Label is not defined then, LABEL NOT DEFINED ERROR generated

3. **MANY OPERANDS ERROR**

   If an opcode is provided with many operands then this error would be generated.

4. **START ERROR**

   If a multiple start statements are given then an error would be generated.

5. **ENDPOINT NOT FOUND ERROR**

   If the STP or END command is not found then an error would be generated.

6. **VARIABLE VALUE ERROR**

   If the variable value is used but not defined this error would be generated.

7. **OPCODE NOT FOUND ERROR**

   If an opcode of a given instruction is not found then an error would be generated.

8. **LITERAL ERROR**

   If find literal where variable or label is needed, then an error would be generated.

9. **LITERAL ERROR**

   If opcode is found with lesser number of operands, then an error would be generated.

SAMPLE INPUT :

START 10

ADD ‘10’

SUB ‘20’

CLA

MUL A

L1 : DIV 10

END

OUTPUT :

LITERAL TABLE

LITERAL	MEMORT ADDRESS (in Binary)
‘10’	00010000
‘20’	00010001

SYMBOL/LABEL TABLE

SYMBOL/LABEL	MEMORT ADDRESS (in Binary)
“A”	00010011
“L1”	00010100

GENERATED MACHINE CODE: