Index

How to use?

Easy !

Run the ‘assemblerMainCode.py ’after placing the ‘sourceCode.txt ’containing the source Assembly program in the same directory. Then check the new files created for the output.

Important: ‘assemblerMainCode.py’ and ‘sourceCode.txt’ should be in the same directory along with ‘opcodeSymbolTable.txt’.

New files created as outputs in the same directory are:

1- ‘symbolTable.txt’
2- ‘literalTable.txt’
3- ‘labelTable.txt’
4- ‘opcodeTable.txt’
5- ‘machineCode.txt’

Please ensure to follow the Instruction formats for no errors. However, any mistakes in the source code, will be printed on the python terminal without hindering the code conversion.

Instruction Formats

Comments

Comments are not read by the assembler. They should be placed only in the starting of each assembly instruction line starting with '@'. Also the value of Location Counter is not incremented.

Example:

```
@ This is a comment !
@ Manas and Prachi made this awesome project !!!
```

Operation Codes

Type 1: no operands required

```
[opcode]
```

Example:

```
CLA
STP
```

Note: After STP is encountered, no instruction will be converted by the assembler to the binary code.

Type 2: requires 1 operand

```
[opcode] [operand]
```

Example:

```
MUL R1
INP R7
```

Note: The operand can be a register, an immediate value, symbol or a literal.

In case of branch statements, it should be a label.

Example:

```
BRP loop
BRZ here
```

Pseudo Opcodes

Not converted into machine Code.

START: It indicates beginning of source code program. It is followed by the value of the Location Counter(LC). If no value is given, then Location Counter is set to default as 0.

Example:

```
START         // LC = 0
START 21      // LC = 21
START 0       // LC = 0
```

LTORG: It indicates Literal origin. It is followed by initialization of a literal in order of its appearance in the source code. If no initialization of literal is followed, nothing happens.

Note that the declaration of a literal begins with an asterisk '*'.

There are two ways to declare a literal:

Type 1: Decimal form

In this case the value is preceded by a 'd'.**

Also if the value is not preceded by a character, then it is by default taken as a decimal.

Example:

```
LTORG
* ='10'        //decimal value of ='10' is 10
* =d'10'       //decimal value of =d'10 'is 10
```

Type 2: Hexadecimal form

In this case the value is preceded by 'h'.**

Example:

```
LTORG
* =h'a9'       //decimal value of =h'a9 'is 169
```

**** characters used for initialization of decimal or hexadecimal as well as characters used in hexadecimal numbers are case sensitive and should be lowercase only.**

DC: Indicates declaration of a symbol/variable and the value is taken in decimal form.
Example:

```
Z DC ‘1’        //decimal value of Z is 1
```

DS: Indicates declaration of a symbol/variable and the value is taken in hexadecimal form.
Example:

```
A DS ‘f9’       //decimal value of A is 249.
```

Immediate values:

Send the value of the operand itself.
Example:

```
MUL #45         //multiply by 45
SUB #3          //subtract 3
```

Immediate values can be between 0 and 63 only. (Limits Inclusive)

Note: In case of any empty line in the source program, the location counter is not incremented.

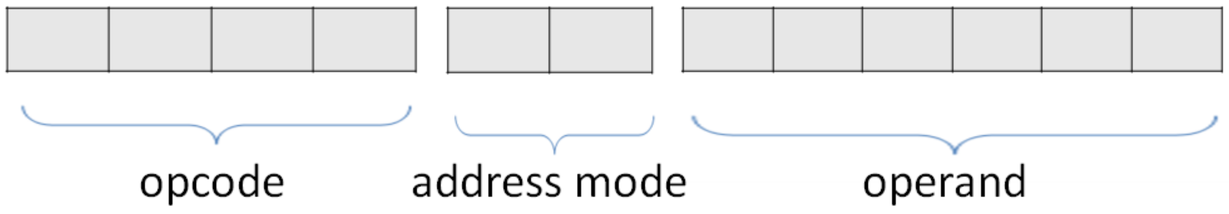
General Architecture

The assembler is built for a 12 bit accumulator architecture consisting of 16 registers and 64 memory locations.

The 16 registers are named as follows-
'R0', 'R1', 'R2', 'R3', 'R4', 'R5', 'R6', 'R7', 'R8', 'R9', 'R10', 'R11', 'R12', 'R13', 'R14', 'R15'.

Each memory location has length 1 word where 1 word = 12 bits.

Assembly instruction to Binary Encoding



- immediate addressing - 00
- direct addressing - 01
- register addressing - 10

- Immediate value
- Register address
- Symbol
- Literal

Instruction size is constant, i.e. 12 bits for every Instruction.

For type 1 operation codes - 'CLA' and 'STP', instruction is 12 bits long as well (no exceptions).

For 'CLA':

The first 4 bits denote the opcode of CLA - 0000.
The next 2 bits denote the immediate addressing mode (to denote that the value of Accumulator Register has been set to 0).
The next 6 bits denote the value of Accumulator Register which is set to 0.
Thus, the machine code becomes 0000 00 000000.

For 'STP':

The first 4 bits denote the opcode of STP - 1100.
To keep the Instruction length constant, the next 2 + 6 bits are stored as a string of 0s.
Thus, the machine code becomes 1100 00 000000.

List of all Operation Codes (OpCodes)

Operation Code	Instruction	Mnemonic
0000	Clear accumulator	CLA
0001	Load into accumulator from address	LAC
0010	Store accumulator contents into address	SAC
0011	Add address contents to accumulator contents	ADD
0100	Subtract address contents from accumulator contents	SUB
0101	Branch to address if accumulator contains zero	BRZ
0110	Branch to address if accumulator contains negative value	BRN
0111	Branch to address if accumulator contains positive value	BRP
1000	Read from terminal and put in address	INP
1001	Display value in address on terminal	DSP
1010	Multiply accumulator and address contents	MUL
1011	Divide accumulator contents by address content. Quotient in R1 and remainder in R2	DIV
1100	Stop execution	STP

Error Handling

IMPORTANT: If the assembler encounters an error, it will report the error on the python IDE along with the location counter and will leave a blank line in the machine code output.
Important: When an error occurs, location counter is decremented. So in total, errors will not increment Location counter.

<u>Zero Division Error:</u>
When programmer sends 0 to the accumulator as immediate value with DIV opcode.

<u>Declaration Error:</u>
It is of two types-
Type 1:
Symbol/literal/label has been declared multiple times.
Type 2:
Symbol/literal/label has not been declared but has been used in the code.

<u>Declarative Statement Error:</u>
If a pseudo opcode like 'LTORG' or 'START' or an opcode such as 'STP' has been used multiple times in the code.
Important Note:
In case of LTORG, error is reported only when LTORG is again used just after initial use of LTORG, rendering the second LTORG redundant.

<u>Exceeded Memory Limit Error:</u>
6 bits have been allocated to store memory address in the Location Counter(LC).
For every instruction, the LC increments by 1 since the size of every instruction is constant ie 12 bits.
Therefore, the maximum value that the LC can store is 63.
Thus if the value of LC exceeds 63, an exceeded memory limit error pops up.

<u>Word Limit Exceeded Error:</u>
Since 6 bits have been allocated to store the operand, so if the binary value of symbol/ literal/immediate value exceeds 6 bits, a word limit exceeded error pops up.

Invalid Immediate Value:

The 2 types of error handled are:

Type 1:

If there are no characters after # or characters that are not valid (such as alphabets), an invalid immediate error pops up.

Type 2:

If the immediate value does not lie in the range [0,63].

Illegal Opcode:

Following errors have been handled in this category;

Type 1:

If the opcode used is not valid, i.e. it is not part of the instruction set, then and invalid opcode error pops up.

Type 2:

If opcodes of Type/Class 1 (which aren't supposed to have operands, eg: 'CLA', 'STP') are followed by an operand.

Type 3:

If opcode of Type/Class 2 (which have 1 operand, eg: 'MUL', 'DIV') are followed by either no operand or more than one operands.

Illegal Register Address:

The registers have predefined addresses that lie in the range [0,15]. If a register address doesn't lie in the aforementioned range, an illegal register address error pops up.

START/STP Missing:

The START pseudo opcode indicates the beginning of an assembly code and sets the value of LC whereas the STP opcode indicates the end of an assembly code. If these statements are missing, an error pops up.

Illegal use of INP:

The assembler can allow the accumulator to take input only into registers. Use of INP with any other type of operand will lead to error display.

Illegal use of Literals:

The assembler can allow the accumulator to take input only into registers. Use of INP with any other type of operand will lead to error.

Source Example

```
@ Program for FACTORIAL of a number !
@ updating Y with the result of factorial.
@ STORING 1 IN Z.
START
INP R1
INP R4
CLA
LOOP: LAC R2
MUL R1
SAC R2
LAC R1
SUB #5
SUB =d'7'
SUB =h'a'
SAC R1
BRP LOOP
LTORG
Z DC '1'
LTORG
* =h'a'
* =d'7'
STP
BRP LOOP
```

// corresponding binary output is given on next page.

Machine Output in Binary Format:

```
1000 10 000001
1000 10 000100
0000 00 000000
0001 10 000010
1010 10 000001
0010 10 000010
0001 10 000001
0100 00 000101
0100 01 010001
0100 01 010000
0010 10 000001
0111 01 000100
1100 00 000000
```

Workflow of the Assembler

Assembler would first initialize the following tables:

OpcodeSymboltable from ‘opcodeSymbolTable.txt’ to get a list of opcodes.
Literal Table - store literals, their values and address
Symbol Table - store symbols, their values and address
Opcode Table – store mnemonic opcode, opcode value, operand and opcode type
Label Table- store label and their address

Assembler then initiates pass 1 and stores the respective values in the respective tables as encountered.

In pass 2, all the instructions are converted to corresponding binary instructions with the help of values from the previous tables.

If any error is encountered during any pass, it is immediately shown on the python terminal.

Project by –

Prachi Goyal
Manas Gupta