

Assembler

Project submitted to the
SRM University – AP, Andhra Pradesh
for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology/Master of Technology

In

**Computer Science and Engineering
School of Engineering and Sciences**

Submitted by

B.O.S.V. SUBHASH

(AP20110010324)



Under the Guidance of

Arnab Mitra

**SRM University–AP
Neerukonda, Mangalagiri, Guntur
Andhra Pradesh – 522 240
[December, 2022]**

Certificate

Date: 12/5/2022

This is to certify that the work present in this Project entitled “**Assembler**” has been carried out by B.O.S.V.SUBHASH under my/our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

(Signature)

Prof. / Dr.

[Name]

Designation,

Affiliation.

Co-supervisor

(Signature)

Prof. / Dr.

[Name]

Designation,

Affiliation

Acknowledgements

I would like to express my special thanks of gratitude to my teacher Mr. Arnab Mitra, who gave me the golden opportunity to do this wonderful project of Compiler design, who also helped me in completing my project. I came to know about so many new things, I am really thankful to them.

Secondly I would also like to thank my friends who have helped me a lot finalizing this project within limited time frame.

B.O.S.V. SUBHASH
CSE Section -E

Abstract:

The relationship between a computer's instruction set architecture and its assembly language instruction set and uses simple addressing modes to write a simple assembly language program. The chapter examines how byte addressing is implemented. It discusses two methods of packing bytes into 16-bit memory words: little endian and big endian, and the ambiguity of the big endian versus little endian. It provides a description of the condition code register (CCR). The CCR contains a set of five condition bits: X, N, Z, V, and C whose value can change with the result of each instruction being executed. The chapter discusses the basic organization of assembly language instructions and explains how the opcode word is interpreted as a part of a machine language instruction. It deals with the fundamentals of creating an assembly language program and uses pseudo opcodes to control the operation of the assembler program. It also analyzes a simple assembly language program.

Introduction:

To convert a program into machine level from assembly level language the instruction written in machine assembly level language must be first get converted to its corresponding opcode in hexadecimal format simultaneously keeping track of memory associated with each instruction.

To do so each instruction and its corresponding opcode should be known in advance so that as they are encountered in the program they can be converted to their corresponding opcode. The 8085-assembly level language defines a set of instructions with their opcode and also the memory size associated to them. Using those tables an assembler can be designed. The instructions can be mapped to the corresponding size and operands can be mapped to its corresponding opcode available in hexadecimal format.

The label should be tracked with their corresponding line no so that in case of any jump statement the program execution can be shifted to corresponding label. The jump statements may be unconditional or conditional. The conditional jump checks for flag so the flag condition must be checked each time we encounter jump statement to decide the flow of the execution of the program.

Keeping these conditions a assembler can be designed using the concept of hash map, hashing through the implementation of dictionary and certain other data structures.

Methodology:

The assembler is a program which converts the program written in assembly language into machine level language.

Each instruction which comprises of operands and registers is associated with certain opcode . Also

The dictionary was used to store these key value pairs.

Similarly, each instruction was associated with certain amount of memory which was occupied by it.

In each 8085 processor there are generally three type of instructions based on memory size that it occupies . Memory occupied by each instructions is either 1 Byte or 2 Byte or 3 Byte.

These integer values were chosen as key and they were mapped to list of corresponding operations which takes that much amount of memory.

The idea to choose dictionary as a data structure was its efficiency with respect to time for searching . The internal hashing is performed to store the keys in case of dictionary . The time complexity to search any value corresponding to key is $O(n)$ if values are atomic and $O(\log n)$ when values are in the form of collection of objects such as list which is very efficient.

The whole file was read once to get all the labels and their corresponding line no and these were stored in dictionary .The labels act as key and line numbers are the values corresponding to each key level. This helps to set the pointer at particular line whenever commanded.

As soon the file is read completely the pointer again is set to the beginning of the file and file is again read line by line. A line contains some instruction which further helps program identify the operation and operands. Program control is then passes to different function according to the number of the operands in the instruction, these function then fetches the corresponding Hexadecimal values from the module file. At the same time the memory size of the instruction is also calculated which is given to the instruction to store its value. Furthermore, the output file is appended with values (memory, hexadecimal code).

In case of a conditional jump instruction the assembler should check the flags and jump according to the flags to levelled position, but this program interrupts the program as soon the jump statement is encounter, and the user input is considered for jump execution. This process keeps on until the last line of the input file if no instruction found incorrect.

Abbreviations:

Addr	16-bit address of the memory location.
Data	8-bit data
data 16	16-bit data

r, r1, r2	One of the registers A, B, C, D, E, H or L
A, B, C, D, H, L	8-bit register
A	Accumulator
H-L	Register pair H-L
B-C	Register pair B-C
D-E	Register pair D-E
PSW	Program Status Word
M	Memory whose address is in H-L pair
H	Appearing at the end of the group of digits specifies hexadecimal, e.g. 2500H
Rp	One of the register pairs.
Rh	The high order register of a register pair
RI	The low order register of a register pair
PC	16 bit program counter, PCH is high order 8 bits and PCL low order 8 bits of register PC.
CS	Carry Status
[]	The contents of the register identified within bracket
[[]]	The content of the memory location whose address is in the register pair identified within brackets
^	AND operation
∨	OR operation
⊕ or ∇	Exclusive OR
←	Move data in the direction of arrow
↔	Exchange contents

Program for the Assembler :

```
# TWO OPERAND
MOV={ 'A':{ 'A': '7F', 'B': '78', 'C': '79', 'D': '7A', 'E': '7B', 'H': '7C', 'L': '7D', 'M': '7E' },
      'B':{ 'A': '47', 'B': '40', 'C': '41', 'D': '42', 'E': '43', 'H': '44', 'L': '45', 'M': '46' },
      'C':{ 'A': '4F', 'B': '4A', 'C': '49', 'D': '4A', 'E': '4B', 'H': '4C', 'L': '4D', 'M': '4E' },
      'D':{ 'A': '57', 'B': '50', 'C': '51', 'D': '52', 'E': '53', 'H': '54', 'L': '55', 'M': '56' },
      'E':{ 'A': '5F', 'B': '5A', 'C': '59', 'D': '5A', 'E': '5B', 'H': '5C', 'L': '5D', 'M': '5E' },
      'H':{ 'A': '67', 'B': '60', 'C': '61', 'D': '62', 'E': '63', 'H': '64', 'L': '65', 'M': '66' },
      'L':{ 'A': '6F', 'B': '6A', 'C': '69', 'D': '6A', 'E': '6B', 'H': '6C', 'L': '6D', 'M': '6E' },
      'M':{ 'A': '77', 'B': '70', 'C': '71', 'D': '72', 'E': '73', 'H': '74', 'L': '75', 'M': '??' },
      'MVI':{ 'A': '3E', 'B': '06', 'C': '0E', 'D': '16', 'E': '1E', 'H': '26', 'L': '2E', 'M': '36' } }

# ONE OPERAND
JMPS={ 'J':{ 'MP': 'C3', 'NZ': 'C2', 'Z': 'CA', 'C': 'DA', 'NC': 'D2', 'PE': 'EA', 'PO': 'E2', 'M': 'FA', 'P': 'F2' },
       'C':{ 'ALL': 'CD', 'NZ': 'C4', 'Z': 'CC', 'C': 'DC', 'NC': 'D4', 'PE': 'EC', 'PO': 'E4', 'M': 'FC', 'P': 'F4' },
       'R':{ 'ET': 'C9', 'NZ': 'C0', 'Z': 'CA', 'C': 'DA', 'NC': 'D0', 'PE': 'EA', 'PO': 'E0', 'M': 'FA', 'P': 'F0' } }

# ONE OPERAND
AMOP={ 'ADD':{ 'A': 'A7', 'B': 'A0', 'C': 'A1', 'D': 'A2', 'E': 'A3', 'H': 'A4', 'L': 'A5', 'M': 'A6' }, #AMOP=OP
       'ADC':{ 'A': 'AF', 'B': 'AA', 'C': 'A9', 'D': 'AA', 'E': 'AB', 'H': 'AC', 'L': 'AD', 'M': 'AE' },
       'SUB':{ 'A': '97', 'B': '90', 'C': '91', 'D': '92', 'E': '93', 'H': '94', 'L': '95', 'M': '96' },
       'SBB':{ 'A': '9F', 'B': '9A', 'C': '99', 'D': '9A', 'E': '9B', 'H': '9C', 'L': '9D', 'M': '9E' } }

# ONE OPERAND ON ACC
BIT={ 'ANA':{ 'A': 'A7', 'B': 'A0', 'C': 'A1', 'D': 'A2', 'E': 'A3', 'H': 'A4', 'L': 'A5', 'M': 'A6' },
      'XRA':{ 'A': 'AF', 'B': 'A8', 'C': 'A9', 'D': 'AA', 'E': 'AB', 'H': 'AC', 'L': 'AD', 'M': 'AE' },
      'ORA':{ 'A': 'B7', 'B': 'B0', 'C': 'B1', 'D': 'B2', 'E': 'B3', 'H': 'B4', 'L': 'B5', 'M': 'B6' },
      'CMP':{ 'A': 'BF', 'B': 'B8', 'C': 'B9', 'D': 'BA', 'E': 'BB', 'H': 'BC', 'L': 'BD', 'M': 'BE' }, }

# TWO OPERANDS
LEFT={ 'LXI':{ 'B': '01', 'D': '11', 'H': '21', 'SP': '31' } }
```

Fig 1.1

```
# TWO OPERANDS
LEFT={ 'LXI':{ 'B': '01', 'D': '11', 'H': '21', 'SP': '31' } }

# ONE OPERAND
OPST={ 'LDA': '3A', 'STA': '32',
       'OUT': 'D3', 'IN': 'DB',
       'LHLD': '2A', 'SHLD': '22', 'ADI': 'C6', 'ACI': 'CE',
       'SUI': 'D6', 'SBI': 'DE', 'ANI': 'E6', 'XRI': 'EE',
       'ORI': 'F6', 'CPI': 'FE', 'XTHL': 'E3', 'SPHL': 'F9' }

OPSTR={ 'INR':{ 'A': '3C', 'B': '04', 'C': '0C', 'D': '14', 'E': '1C', 'H': '24', 'L': '2C', 'M': '34' },
        'DCR':{ 'A': '3D', 'B': '04', 'C': '0D', 'D': '15', 'E': '1D', 'H': '25', 'L': '2D', 'M': '35' },
        'INX':{ 'B': '03', 'D': '13', 'H': '23', 'SP': '33' },
        'DCX':{ 'B': '0B', 'D': '1B', 'H': '2B', 'SP': '3B' },
        'DAD':{ 'B': '09', 'D': '19', 'H': '29', 'SP': '39' },
        'LDAX':{ 'B': '0A', 'D': '1A' },
        'STAX':{ 'B': '02', 'D': '12', 'SP': 'PSW' },
        'PUSH':{ 'B': 'C5', 'D': 'D5', 'H': 'E5', 'SP': 'F5' },
        'POP':{ 'B': 'C1', 'D': 'D1', 'H': 'E1', 'SP': 'F13' } }

# ZERO OPERAND
ACCOP={ 'XCHG': 'EB', 'CMA': '2F', 'CMC': '3F', 'STC': '37', 'RLC': '07',
        'RRC': '0F', 'RAL': '17', 'RAR': '1F', 'DAA': '27' }

# ZERO OPERAND
ICPC={ 'EI': 'FB', 'DI': 'F3', 'RIM': '20', 'SIM': '30', 'NOP': '00', 'HLT': '76' }

# OUT IN

# def tohexa(t):
```

Fig 1.2

Fig 1.1 and Fig 1.2 shows the Opcode values for the assembly command


```

from opcodefile import *

def twooprnd(opt, oprnd):
    dest,srt=oprnd[0],oprnd[1]
    if(opt=='MOV' ):
        # print(MOV[dest][srt])
        f_write.write(MOV[dest][srt])
        f_write.write('\n')
    elif(opt=='MVI'):
        f_write.write(MOV[opt][dest])
        f_write.write('\n')
    elif(opt=='LXI'):
        f_write.write(LEFT[opt][dest])
        f_write.write('\n')
    else:
        # print(opt,dest,srt)
        print(opt,'Operation not valid')
        print('Program status: Exit')
        exit(1)

def oneoprnd(opt,oprnd):
    dest=oprnd[0]
    if(opt in OPST):
        f_write.write(OPST[opt])
        f_write.write('\n')
    elif(opt in OPSTR):
        f_write.write(OPSTR[opt][dest])
        f_write.write('\n')
    elif(opt in AMOP ):
        f_write.write(AMOP[opt][dest])
        f_write.write('\n')
    elif(opt[:1] in JMPS):

```

Fig 1.3

```

if(oprnd[0] in checks):
    print("Jump statement encountered")
    print('Enter 1 to continue and 0 to ignore ')
    if(int(input())):
        f_input.seek(checks[oprnd[0]],0)
    else:
        print('Jump address not found')
        print('Program status: Exit')
        exit(1)
    f_write.write(JMPS[opt[:1]][opt[1:]])
    f_write.write('\n')
elif(opt in BIT):
    f_write.write(BIT[opt][dest])
    f_write.write('\n')

else:
    print(opt,'has wrong instructions' )

def nooprnd(opt):
    if(opt in ACCOP):
        f_write.write(ACCOP[opt])
        f_write.write('\n')
    elif(opt in ICPC):
        f_write.write(ICPC[opt])
        f_write.write('\n')
    else:
        print(opt,'Operation not valid!!')
        print('Program status: Exit')
        exit(1)

```

Fig 1.4

```

checks,ln={},0

f_input=open('/home/ssundarkars/Documents/vscode documents/Labs/Cmpl dgn/ASCtoHEXconverter/input.txt','r+')
for line in f_input:
    ln+=1
    line=line.rstrip()
    if(line[-1]==':'):
        checks.update({line[-1]:ln})
f_input.seek(0)
f_write=open('/home/ssundarkars/Documents/vscode documents/Labs/Cmpl dgn/ASCtoHEXconverter/output.txt','a')
memory=int(input('Enter Index Memory address: '))

for line in f_input:
    try:
        opt,oprnd = line.split()
        oprnd=oprnd.split(',')
        # print(memory,end='\t\t\t')
        f_write.write(str(memory))
        f_write.write('\t\t\t')
        memory+=len(oprnd)+1
        # ln+=1
    except ValueError:
        line=line.strip()
        if(':')==line[-1] ):
            # if(line not in checks):
            #     ln+=1
            checks.update({line[:-1]:ln})
            continue

```

Fig 1.5

```

        else:
            opt,oprnd=line.strip(),''
            # print(memory,end='\t\t\t')
            f_write.write(str(memory))
            f_write.write('\t\t\t')
            memory+=len(oprnd)+1
            # ln+=1

            if(len(oprnd)==2):
                twooprnd(opt,oprnd) #
            elif(len(oprnd)==1):
                oneoprnd(opt,oprnd)
            elif(len(oprnd)==0):
                nooprnd(opt)
f_input.close()
f_write.close()

```

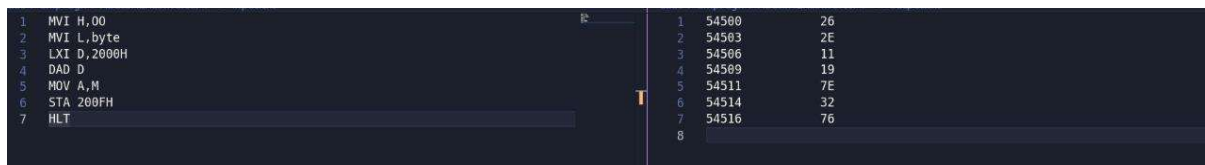
Fig 1.6

The below mentioned figures is one example how our program will work

```
MVI B,04H
LXI H,2000H
SUB A
LOOP:
INX H
DCR B
STA 2004H
HLT
STOP:
```

Fig 1.7

Fig 1.7 shows the assembly language that we are giving to the code



1	MVI H,00	1	54500	26
2	MVI L,byte	2	54503	2E
3	LXI D,2000H	3	54506	11
4	DAD D	4	54509	19
5	MOV A,M	5	54511	7E
6	STA 200FH	6	54514	32
7	HLT	7	54516	76
		8		

Fig 1.8

Fig 1.8 shows the output of the Assembler

Conclusion:

As you can see, we are able to design an assembler which converts the assembly level language to its equivalent address code where for every instruction from input is mapped to an address location. From this assembler we can easily generate address code for the instructions and use them in evaluating the instructions.