Chapter 1:

INTRODUCTION

Computer software, or simply **software**, is a collection of data or computer instructions that tell the computer how to work. There are two main types of software

- System software
- Application software

System software is software designed to provide a platform for other software. This software makes it possible for the user to focus on an application or other problem to be solved.

Application software is a program or group of programs designed for end users. They include programs such as web browsers, office software etc.

In System software one of the most important components is the Assembler.

An **assembler** program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an operation code ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities.

An assembler takes source program and translates it into object code, then this object code is converted into executable program using linker and then the memory is allocated with the help of loader. These steps are illustrated using pass1 and pass2 of 2-Pass assembler.

Pass 1 of 2-Pass assembler generates symbol table. During Pass 1, OPTAB is used to look up and validate operation opcode in source program. During the Pass 1 of assembler, the labels are entered into symbol table as they are encountered in the source program along with their assigned address.

Pass 2 of 2-Pass assembler generates object code and assembles the object codes into the object program.

1.1 SIC/XE MACHINE ARCHITECTURE

The Simplified Instructional Computer (also abbreviated SIC) is a hypothetical computer system. Due to the fact that most modern microprocessors include subtle, complex functions for the purposes of efficiency, it can be difficult to learn systems programming using a real-world system. The Simplified Instructional Computer solves this by abstracting away these complex behaviors in favor of an architecture that is clear and accessible for those wanting to learn systems programming. SIC/XE stands for Simplified Instructional Computer Extra Equipment or Extra Expensive. This computer is an advance version of SIC. Both SIC and SIC/XE are closely related to each other that's why they are Upward Compatible.

1.2 Memory:

Memory consists of 8 bit-bytes and the memory size is 1 megabytes (2 power 20 bytes). Standard SIC memory size is very small. This change in the memory size leads to change in the instruction formats as well as addressing modes. 3 consecutive bytes form a word (24 bits) in SIC/XE architecture.

1.3 Registers:

SIC machines have several registers, each 24 bits long and having both a numeric and character representation.

A (0): Accumulator register used for arithmetic operations.

X (1): Index register used to store and calculate addresses.

L (2): Linkage register used for jumping to specific memory addresses and storing return addresses.

PC (8): Program counter register contains the address of the next instruction to execute.

SW (9): Status word register contains a variety of information, such as carry or overflow flags.

In addition to the standard SIC registers, there are also four additional general-purpose registers specific to the SIC/XE machine.

B (3): Base register used for addressing.

S (4): General purpose register.

T (5): General purpose register.

F (6): Floating point accumulator register is of 48-bits instead of 24.

1.4 Instruction Formats:

In SIC/XE architecture there are 4 types of instruction formats. Format 1 consists of 8 bits of allocated memory to store instructions, format 2 consists of 16 bits of allocated memory to store 8 bits of instructions and two 4-bits segments to store operands, format 3 consists of 6 bits to store an instruction, 6 bits of flag values, and 12 bits of displacement, format 4 consists of the same elements as format 3, but instead of a 12-bit displacement, stores a 20-bit address.

Both format 3 and format 4 have six-bit flag values in them, consisting of the following flag bits:

n: Indirect addressing flag

i: Immediate addressing flag

x: Indexed addressing flag

b: Base address-relative flag

p: Program counter-relative flag

e: Format 4 instruction flag

Format 1(1 byte):

opcode (8 bits)

Format 2(2 bytes):

opcode (8 bits)	r1(4 bits)	r2(4 bits)

Format 3(3 bytes):

opcode(6bits)	n(1bit)	i(1bit)	x(1bit)	b(1bit)	p(1bit)	e(1bit)	Displacement (12 bits)

Format 4(4 bytes):

opcode(6bits)	n(1bit)	i(1bit)	x(1bit)	b(1bit)	p(1bit)	e(1bit)	Address (20 bits)

1.5 Addressing Modes:

Two new relative addressing modes are available for use with instructions assembled using format 3.

- Base relative addressing mode's target address (TA) is calculated by TA=(B)+ displacement where (0<= displacement <= 4095)
- Program counter relative addressing mode's target address (TA) is calculated by TA=(PC) + displacement where (-2048<= displacement <=2047)

Some rules are to be followed while writing a object code for the instruction.

e = 0 : format 3

e = 1 : format 4

format 3:

b = 1, p = 0 (base relative)

b = 0, p = 1 (pc relative)

b = 0, p = 0 (direct addressing)

format 4:

b = 0, p = 0 (direct addressing)

x = 1 (index)

i = 1, n = 0 (immediate)

i = 0, n = 1 (indirect)

i = 0, n = 0 (SIC)

i = 1, n = 1 (SIC/XE for SIC compatible)

Chapter 2:

2-PASS ASSEMBLER

2.1 2 Pass Assembler

Programmers uses a high level language which is not understood by the computer. Hence assemblers translates this high level language into machine code so that programming can be done effectively.

A pass is said to be a onetime scan of the entire code. As the name suggests 2 pass assemblers scan the code twice. In the first pass the assembler checks for any syntax errors and updates the symbol table and in the second pass the object program is generated.

2.2 Pass One

As mentioned earlier in pass 1 the assembler checks for any errors. By errors we mean any syntactic errors. If an error is generated the assembly of the program terminates by flagging an error.

Errors are checked by checking the mnemonics in the program with the mnemonics in the object code table. The variable names that are declared in the variable segment are entered in the symbol table. In the code segment the variable name wherever used is cross checked with the entries in the symbol table. If any mismatch occured then an error is flagged.

The goals of pass 1 are mentioned below:

- Assign addresses to all the statements in the program.
- Addresses and the variable names are updated in the symbol table.

2.3 Pass Two

Pass two phase assembles the code into an object program. The input of pass 2 is the output of pass 1.

In pass two each instruction is fetched and the object code for each instruction is generated. The opcode for each mnemonic is available in the object table and the address of the variable is available in the symbol table.

Assembler directives will not be translated into machine instructions. They only provide instruction/direction/information to the assembler. Some The basic assembler directives are **START** to specify name and starting address for the program, **END** to indicate the end of the source program.

Based on the instruction formats the assembler assembles the code into its respective object program.

The goal of Pass 2 is:

- Translate the opcode and generating object program.

2.4 Data Structures Used

Three simple data structures are used. They are the Symbol table, Operation code table and the location counter.

OPTAB: This table consists of the representations of the various mnemonics that are used the source code. This table is mainly used to validate that correctness of the mnemonic used and to fetch the operation code of the mnemonic used.

This table is usually organized as a hash table.

SYMTAB: This table consists of the name and the address of the various variables used in the source code. This table is mainly used to check is declared variables are used in the code segment. In the second pass the address of each variable is retrieved in order to assemble the object program.

LOCCTR: This data structure is used to point at the succeeding instruction of the current instruction that is being processed.

Chapter 3:

LITERATURE SURVEY

3.1 TYPES OF ASSEMBLER

- **Macro assembler[1]:** Includes a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code.

- **Cross assembler[1]:** Assembler that is run on a computer or operating system (the host system) of a different type from the system on which the resulting code is to run (the target system).

- **High-level assembler[1]:** Program that provides language abstractions more often associated with high-level languages, such as advanced control structures (IF/THEN/ELSE, DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.

- **Micro-assembler[1]:** Program that helps prepare a microprogram, called firmware, to control the low level operation of a computer.

- **Meta-assembler[1]:** Term used in some circles for a program that accepts the syntactic and semantic description of an assembly language and generates an assembler for that language.

3.2 BASED ON PASSES

There are two types of assemblers based on how many passes through the source are needed (how many times the assembler reads the source) to produce the object file.

- **One-pass assemblers[1]**: Go through the source code once. Any symbol used before it is defined will require correction at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.
- **Multi-pass assemblers[1]**: Create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

Chapter 4:

DESIGN OF ASSEMBLER

- Design and Analysis of Algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology.

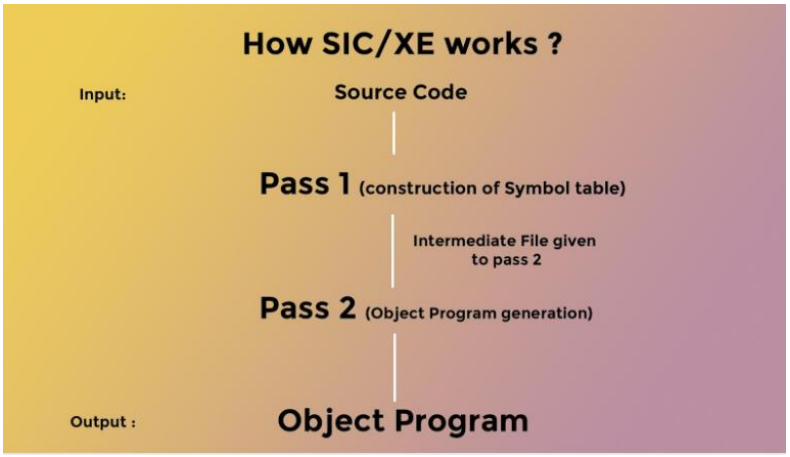

Fig 4.1: Working flow of 2-Pass assembler

- Python language is used to implement the SIC/XE assembler. The input SIC/XE instructions are given in a list in the python code. In Pass 1 the symbol table generated containing labels and addresses which is stored as a dictionary of the Python. This dictionary is used by pass 2 to generate object codes which are assembled into the object program. This object program is the final output.

Chapter 5:

ALGORITHMS

5.1 Pass 1 Algorithm

1.Set location counter value to the address specified in the “START” instruction. Read a line from the given assembly code and check if label is present or not. If it contains label add that label and the current location counter’s value to the symbol table.

2.Increment the location counter’s value based on the instruction format or opcode.

3.Continue 1 and 2 till the end of assembly code is reached.

5.2 Pass 2 Algorithm

1. Read the start statement and get assembly program name and first executable line’s address. Write header record.

2. Initiate text record.

3. Scan the next statement from the intermediate file(Here dictionary is used instead of file).

4. Get the opcode of that statement by a function named “oc”, based on the instruction format, addressing mode. Based on the displacement value choose PC relative or BASE relative addressing mode.

5. Add the opcode to the text record. If the opcode field of text record exceeded 60 characters, print the text record and initiate new text record.

6. If end statement is not the next line, go to step 3. Else write the modification records for format 4 instructions using the starting location of the address field and the length of the address field to be modified, end record using program length and first executable statement of the line.

Chapter 6:

IMPLEMENTATION

Pass1 function is called in the main program by sending dictionary containing of SIC/XE instructions as argument. Pass1 function will work on this dictionary to give the symbols and their addresses in a dictionary called sytab. Then the pass1 function calls the pass2 by sending the SIC/XE instructions dictionary and the length as arguments to pass2 function.

```
def pass1(code):
    loc=int(code[0].split()[2],16)
    for i in range(1,len(code)):
        if(code[i].split()[0]!='END'):
                if(len(code[i].split())==3):
                    sytab[code[i].split()[0]]=hex(loc)[2:]
                    if(code[i].split()[1][0]=='+'):
                        add.append(hex(loc)[2:]+' '+str(4))
                        loc=loc+4
                    elif(code[i].split()[1]=='RESW'):
                        add.append(hex(loc)[2:]+' '+str(3*int(code[i].split()[2])))
                        loc=loc+(3*int(code[i].split()[2]))
                    elif(code[i].split()[1]=='RESB'):
                        add.append(hex(loc)[2:]+' '+str(int(code[i].split()[2])))
                        loc=loc+int(code[i].split()[2])
                    elif(code[i].split()[1]=='WORD'):
                        add.append(hex(loc)[2:]+' '+str(3))
                        loc=loc+3
                    elif(code[i].split()[1]=='BYTE'):
                        if(code[i].split()[2][0]=='X'):
                            add.append(hex(loc)[2:]+' '+str(1))
                            loc=loc+1
                        if(code[i].split()[2][0]=='C'):
                            add.append(hex(loc)[2:]+' '+str(len(code[i].split()[2])-3))
                            loc=loc+len(code[i].split()[2])-3
                    elif(code[i].split()[1] in op2sy):
                        add.append(hex(loc)[2:]+' '+str(2))
                        loc=loc+2
                    else:
                        add.append(hex(loc)[2:]+' '+str(3))
                        loc=loc+3
                else:
                    if(code[i].split()[0]!='BASE'):
                        if(code[i].split()[0][0]=='+'):
                            add.append(hex(loc)[2:]+' '+str(4))
                            loc=loc+4
                        elif(code[i].split()[0] in op2sy):
                            add.append(hex(loc)[2:]+' '+str(2))
                            loc=loc+2
                        else:
                            add.append(hex(loc)[2:]+' '+str(3))
                            loc=loc+3
                    else:
                       add.append('BASE')
                       base=code[i].split()[1]
                if(len(code[i].split())!=1):
                    if(code[i].split()[-2][-3:]=='LDB'):
                        base=code[i].split()[-1]
                        if(base[0]=='#'):
                           base=base[1:]

    prglen=hex((loc)-int(code[0].split()[2],16))[2:]
    print(sytab)
    print('')
    print('')
    #print(prglen,"{in hexadecimal}")
    pass2(prglen,code,base)
```

Fig 6.1: Code snippet for Pass 1

For the case of simplicity modend function is used for generating records. modend function called in pass2 with the dictionary containing of SIC/XE instructions as argument passed. In this function, modification and end records are generated. Modification records contain the starting location of the address field to be modified and the length of the address field to be modified in half bytes. End record consists of address of first executable instruction.

```
def modend(code):
    for q in range(1,len(add)):
        if(add[q].split()[-1]=='4'):
            tomod=hex(int(add[q].split()[0],16)+1)[2:].rjust(6,'0')
            bit='5'.rjust(2,'0')
            print("%s%s%s%s"%('M^',tomod,'^',bit))
    print("E^"+hex(int(code[0].split()[-1]))[2:].rjust(6,'0'))
```

Fig 6.2: Code snippet for Modification and End records

negbi function is used to convert the negative number obtained to the positive hexadecimal.

```
def negbi(x):
    x=bin(int(x,2)-1)[2:].rjust(12,'0')
    y=''
    for i in range (len(x)):
        y=y+str(int(x[i])^1)
    return hex(int(y,2))
```

Fig 6.3: Code snippet for converting negative hexadecimal

In the program, pass2 function is called in the pass1 after all the instructions of pass1 have been executed. In the pass2 Header records and the text records are generated and for the modification and end record modend function is called at the end of pass2.Header record consists of program name, starting address and the object program length. Text records consist of the starting address for object code in that record, length of object codes in that record and the object codes.

```
def pass2(prglen,code,base):
    print("H^%-6s^%s^%s"%(code[0].split()[1],code[0].split()[2].rjust(6,'0'),prglen.rjust(6,'0')))
    t='T^'
    leng=0;
    for i in range(1,len(code)):
        if(code[i].split()[0]!='END'and code[i].split()[0]!='BASE'):

            opcode=oc(code[i],add[i].split()[1],add[i].split()[0],base)

            if(opcode==-1):
                if(len(t)!=2):
                    textr=t.split("  ")
                    print(textr[0]+hex(leng)[2:].rjust(2,'0')+textr[1])
                    #print(t)
                    t='T^'
                    leng=0
            else:
                if(len(t)==2):
                    t=t+add[i].split()[0].rjust(6,'0')+str('^')+'  ^'+opcode
                    leng=leng+int(add[i].split()[1])
                elif(len(t)+len(opcode)+1<=81):
                    leng=leng+int(add[i].split()[1])
                    t=t+str('^')+opcode
                else:
                    textr=t.split("  ")
                    print(textr[0]+hex(leng)[2:].rjust(2,'0')+textr[1])
                    #print(t)
                    t='T^'
                    leng=0
                    t=t+add[i].split()[0].rjust(6,'0')+str('^')+'  ^'+opcode
                    leng=leng+int(add[i].split()[1])
        else:
            if(len(t)!=2 and code[i].split()[0]!='BASE'):
                    #print(t)
                    textr=t.split("  ")
                    print(textr[0]+hex(leng)[2:].rjust(2,'0')+textr[1])
    modend(code)
```

Fig 6.4: Code snippet for Pass 2

Chapter 7:

RESULTS

Input for pass 1:

The instructions of the SIC/XE program is given in the Python code as input. The instructions of the SIC/XE program is stored in the Dictionary created in the Python code. Pass 1 uses this dictionary as its input.

Output of Pass 1:

Symbol table as dictionary is generated.

```
{'FIRST': '0', 'CLOOP': '6', 'ENDFIL': '1a', 'EOF': '2d', 'RETADR': '30',
'LENGTH': '33', 'BUFFER': '36', 'RDREC': '1036', 'RLOOP': '1040', 'EXIT': '1056',
'INPUT': '105c', 'WRREC': '105d', 'WLOOP': '1062', 'OUTPUT': '1076'}
```

Fig 7.1: Symbol Table contents in Dictionary of Python

Input of Pass 2:

The Symbol table generated in pass 1 is used in pass 2 for the labels and their respective addresses for the calculation of the object codes. The program length and the SIC/XE instructions stored in the dictionary are given as input to Pass 2.

Output of Pass 2:

Object program corresponding to SIC/XE instructions is generated in pass 2.

```
H^START ^000000^001077
T^000000^1d^17202d^69202d^4b101036^032026^290000^332007^4b10105d^3f2fec^032010
T^00001d^13^0f2016^010003^0f200d^4b10105d^3e2003^454f46
T^001036^1d^b410^b400^b440^75101000^e32019^332ffa^db2013^a004^332008^57c003^b850
T^001053^1d^3b2fea^134000^4f0000^F1^b410^774000^e32011^332ffa^53c003^df2008^b850
T^001070^07^3b2fef^4f0000^05
M^000007^05
M^000014^05
M^000027^05
M^00103d^05
E^000000
```

Fig 7.2: SIC/XE Object Program

Chapter 8:

TESTING

Table No 8.1: Testcases

Serial No.	Testcases	Expected result	Actual result	Result status
1.	Run the program with the SIC/XE instructions.	Symbol table along with object program is to be generated.	Got the expected output.	Success
2.	Valid SIC/XE program where instructions exceed PC Relative addressing mode and Base addressing mode.	Object program with ‘error’ is to be printed at the place of object code for that particular instructions.	Object program with ‘error’ is printed at the place of object code for that particular instructions.	Success
3.	The SIC/XE Program should start with label ‘START’.	If the label is not ‘START’ no output is generated.	Output is not generated hence the starting label is not ‘START’.	Success

CONCLUSION

The project “2-Pass assembler” primarily focuses on generating a Symbol Table and the object program for the SIC/XE program. With respect to the programming language which we give as input to the assembler we can create our own procedure to implement and design the data structures and also generate object/machine code that the system can understand.

This project helped us a lot in knowing about the different types of Assemblers, how an Assembler can be used to convert an Assembly level program into a machine level code that a system can understand.

We want to conclude that our team got good hands on experience on implementing an assembler. This helped our team to gain a lot of knowledge about implementing an assembler with Python language.

.

BIBILOGRAPHY

Online Source:

1. https://en.wikipedia.org/wiki/Assembly_language

2. https://www.geeksforgeeks.org/introduction-of-assembler/

Offline Source:

1. System Software: An Introduction to systems Programming, Leland L Beck, Manjula D, 3rd Edition