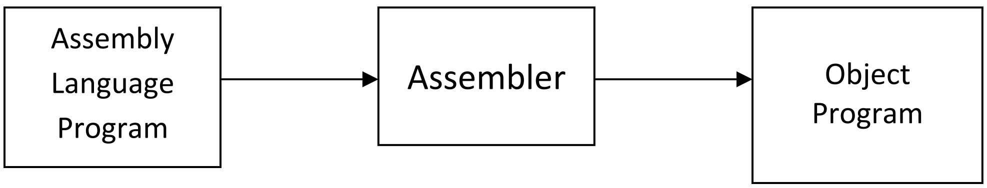
Prepared By
CHETAN D S
Asst. Prof. CSE
RNSIT

UNIT 2: ASSEMBLERS

The basic assembler functions are:

- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.



SIC Assembler Directive:

START: Specify name and starting address for the program

END: Indicate End of the program and (optionally) specify the first execution instruction in the program.

BYTE: Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.

WORD: Generate one-word integer constant.

RESB: Reserve the indicated number of bytes for a data area.

RESW: Reserve the indicated number of words for a data area.

A simple SIC Assembler

The design of assembler in other words:

1. Convert mnemonic operation codes to their machine language equivalents.

   Example: Translate LDA to 00.

2. Convert symbolic operands to their equivalent machine addresses.

   Example: Translate GAMMA to 400F

3. Build the machine instructions in the proper format.
4. Convert the data constants to internal machine representations.

   Example: ONE WORD 1 to 000001

5. Write the object program and the assembly listing

Two Pass Assembler

Pass-1

- Assign addresses to all the statements in the program
- Save the addresses assigned to all labels to be used in *Pass-2*
- Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
- Defines the symbols in the symbol table(generate the symbol table)

Pass-2

- Assemble the instructions (translating operation codes and looking up addresses).
- Generate data values defined by BYTE, WORD etc.
- Perform the processing of the assembler directives not done during *pass-1*.
- Write the object program and assembler listing.

Assembler Design:

The most important things which need to be concentrated is the generation of Symbol table and resolving *forward references*.

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

DELTA=GAMMA + INCR – 1

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	ARTH	START	4000	
4000		LDA	GAMMA	00400F
4003		ADD	INCR	184012
4006		SUB	ONE	1C4015
4009		STA	DELTA	0C400C
400C	DELTA	RESW	1	
400F	GAMMA	RESW	1	
4012	INCR	RESW	1	
4015	ONE	WORD	1	000001
4018		END		

OPTAB	
MNEMONIC	**OPCODE**
LDA	00
ADD	18
SUB	1C
STA	0C

SYMTAB	
LABEL	**ADDRESS**
DELTA	400C
GAMMA	400F
INCR	4012
ONE	4015

Figure 2.1: Assembly Language Program with object code

Object Code for Instruction

LDA GAMMA

Opcode	X	Address
0000 0000	0	100 0000 0000 1111

0 0 4 0 0 F

OBJECT PROGRAM

The simple object program contains three types of records: Header record, Text record and end record.

The header record contains the starting address and length.

Text record contains the translated instructions and data of the program, together with an indication of the addresses where these are to be loaded.

The end record marks the end of the object program and specifies the address where the execution is to begin.

Syntax

- Header record
  - Col. 1 H
  - Col. 2~7 Program name
  - Col. 8~13 Starting address of object program (hex)
  - Col. 14~19 Length of object program in bytes (hex)
- Text record
  - Col. 1 T
  - Col. 2~7 Starting address for object code in this record (hex)
  - Col. 8~9 Length of object code in this record in bytes (hex)
  - Col. 10~69 Object code, represented in hex (2 col. per byte)
- End record
  - Col.1 E
  - Col.2~7 Address of first executable instruction in object program (hex)

HARTH 004000000018

T0040000C00400F1840121C40150C400C

T00401503000001

E004000

Fig 2.2 Object program corresponding to Fig 2.1

Write the object program for the ALP given below

STR2 = STR1 where STR1=”HELLO”

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	COPY	START	2000	
2000		LDX	ZERO	042019
2003	MOVECH	LDCH	STR1,X	50A00F
2006		STCH	STR2,X	54A014
2009		TIX	FIVE	2C201C
200C		JLT	MOVECH	382003
200F	STR1	BYTE	C’HELLO’	48454C4C4F
2014	STR2	RESB	5	
2019	ZERO	WORD	0	000000
201C	FIVE	WORD	5	000005
201F		END		

OPTAB	
MNEMONIC	**OPCODE**
LDX	04
LDCH	50
STCH	54
TIX	2C
JLT	38

SYMTAB	
LABEL	**ADDRESS**
MOVECH	2003
STR1	200F
STR2	2014
ZERO	2019
FIVE	201C

Object Code for the instruction

LDCH STR1,X

Opcode	X	Address
0101 0000	1	010 0000 0000 1111

5 0 A 0 0 F

HCOPY 00200000001F

T0020001404201950A00F54A0142C201C38200348454C4C4F

T002019060000000000005

E0020000

Algorithms and Data structure

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (SYMTAB).

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.
- Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2.
- A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

LOCCTR:

LOCCTR is an important variable which helps in the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

The Algorithm for Pass 1:

```
Begin
  read first input line
  if OPCODE = 'START' then begin
     save #[Operand] as starting address
     initialize LOCCTR to starting address
     write line to intermediate file
     read next input line
     end( if START)
   else
     initialize LOCCTR to 0
     While OPCODE != 'END' do
       begin
          if this is not a comment line then
             begin
               if there is a symbol in the LABEL field then
                  begin
                    search SYMTAB for LABEL
                     if found then
                        set error flag (duplicate symbol)
                     else
                        (if symbol)
```

```
search OPTAB for OPCODE
if found then
   add 3 (instr length) to LOCCTR
else if OPCODE = ‘WORD’ then
    add 3 to LOCCTR
else if OPCODE = ‘RESW’ then
      add 3 * #[OPERAND] to
            LOCCTR
else if OPCODE = ‘RESB’ then
      add #[OPERAND] to LOCCTR
else if OPCODE = ‘BYTE’ then
begin
      find length of constant in bytes
      add length to LOCCTR
end
else
set error flag (invalid operation code)
end (if not a comment)
write line to intermediate file
read next input line
end { while not END}
```

write last line to intermediate file

Save (LOCCTR – starting address) as program length

End {pass 1}

- The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line.
- If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value.
- If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol.
- It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction.
- If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds a value one to the LOCCTR, if RESB it adds number of bytes.
- If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

The Algorithm for Pass 2:

Begin

read 1st input line

if OPCODE = 'START' then

begin

write listing line

```
      read next input line
    end
  write Header record to object program
  initialize 1st Text record
while OPCODE != 'END' do
   begin
     if this is not comment line then
       begin
         search OPTAB for OPCODE
         if found then
           begin
             if there is a symbol in OPERAND field then
               begin
                 search SYMTAB for OPERAND field then
                 if found then
                 begin
  store symbol value as operand address
  else
    begin
  store 0 as operand address
                       set error flag (undefined symbol)
                   end
```

```
                end (if symbol)
              else store 0 as operand address
                  assemble the object code instruction
              else if OPCODE = 'BYTE' or 'WORD" then
      convert constant to object code
              if object code doesn't fit into current Text record then
                begin
                  Write text record to object code
                  initialize new Text record
                end
            add object code to Text record
        end {if not comment}
        write listing line
      read next input line
      end
    write listing line read
    next input line write
    last listing line
  End {Pass 2}
```

Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file. A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code.

If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets added to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.

If the opcode is BYTE or WORD, then the constant value is converted to its equivalent object code(for example, for character EOF, its equivalent hexadecimal value '454f46' is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assembled and when the END directive is encountered, the End record is written.

Generate the complete object program for the following assembly level program

LOCCTR	SOURCE STATEMENT			OBJECT CODE
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	69101788
		BASE	TOTAL	
0009	LOOP	ADD	TABLE,X	1BA00C
000C		TIX	COUNT	2F2006
000F		JLT	LOOP	3B2FF7
0012		STA	TOTAL	0F4000
0015	COUNT	RESW	1	
0018	TABLE	RESW	2000	
1788	TOTAL	RESW	1	
178B		END	FIRST	

Program Length= LOCCTR – STARTING ADDRESS=178B-0=178BH

OPTAB	
MNEMONIC	**OPCODE**
LDA	00
LDB	68
ADD	18
TIX	2C
JLT	38
STA	0C
CLEAR	B4

SYMTAB	
LABEL	**ADDRESS**
FIRST	0000
LOOP	0009
COUNT	0015
TABLE	0018
TOTAL	1788

The Object code for the instruction

+LDB #TOTAL

Opcode	N	I	X	B	P	E	Address
0110 10	0	1	0	0	0	1	0000 0001 0111 1000 1000
6	9				1		0 1 7 8 8

STA TOTAL

Opcode	N	I	X	B	P	E	Displacement
0000 11	1	1	0	1	0	0	0000 0000 0000
0	F			4			0 0 0

The instruction cannot be assembled by using Program Counter Relative Addressing Mode because the Displacement what we calculate can not fit into 12 bit displacement. So, Base Relative addressing mode is used.

Displacement = TA – (B)

= 1788-1788=0

Object Program

HSUM 00000000178B

T00000015B410010000691017881BA00C2F20063B2FF70F4000

E000000

Program Relocation

Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

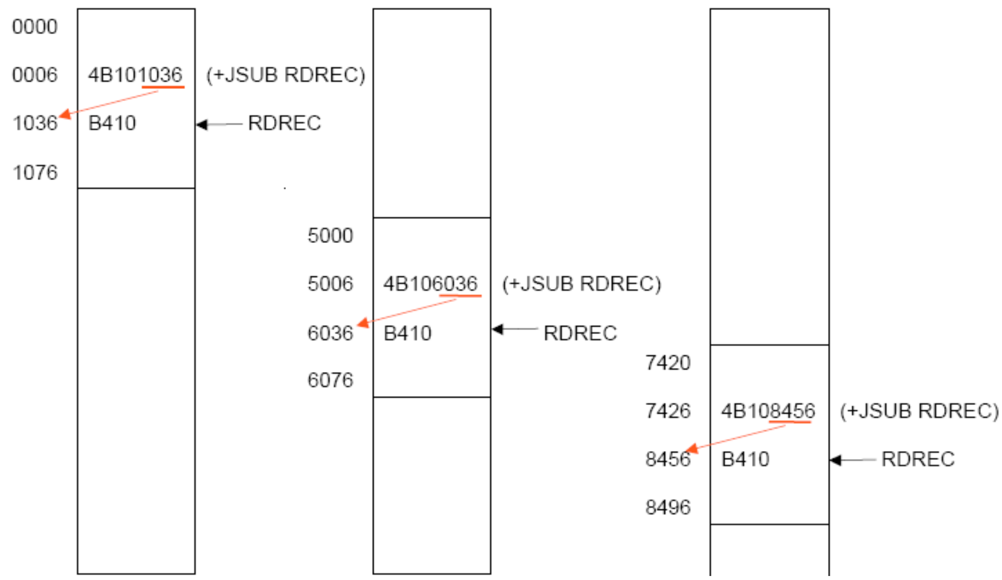


Fig: Examples of Program Relocation

The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.

- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000.
- The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- The only part of the program that require modification at load time are those that specify direct addresses. The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.

- From the object program, it is not possible to distinguish the address and constant The assembler must keep some information to tell the loader. The object program that contains the modification record is called a relocatable program.
- For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

Modification record

Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9	Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified. The length is stored in half-bytes (4 bits). The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

The Modification Record for

    +JSUB   RDREC

instruction is

    M00000705

000007 is the starting location of the address field to be modified by the loader for proper execution of the program.

05 is the length of the address field to be modified , in half bytes.

Design and Implementation Issues

Some of the features in the program depend on the architecture of the machine. If the program is for SIC machine, then we have only limited instruction formats and hence limited addressing modes. We have only single operand instructions. The operand is always a memory reference. Anything to be fetched from memory requires more time. Hence the improved version of SIC/XE machine provides more instruction formats and hence more addressing modes. The moment we change the machine architecture the availability of number of instruction formats and the addressing modes changes. Therefore the design usually requires considering two things: Machine-dependent features and Machine-independent features.

***