System Programming

Assembler

Prepared By,

Prof. Rohan C Prajapati

IT, GIT,Gandhinagar

Introduction

- There are two main classes of programming languages: *high level* (e.g., C, Pascal) and *low level*.
- *Assembly Language* is a low level programming language. Programmers code symbolic instructions, each of which generates machine instructions.
- An *assembler* is a program that accepts as input an assembly language program (source) and produces its machine language equivalent (object code) along with the information for the loader.

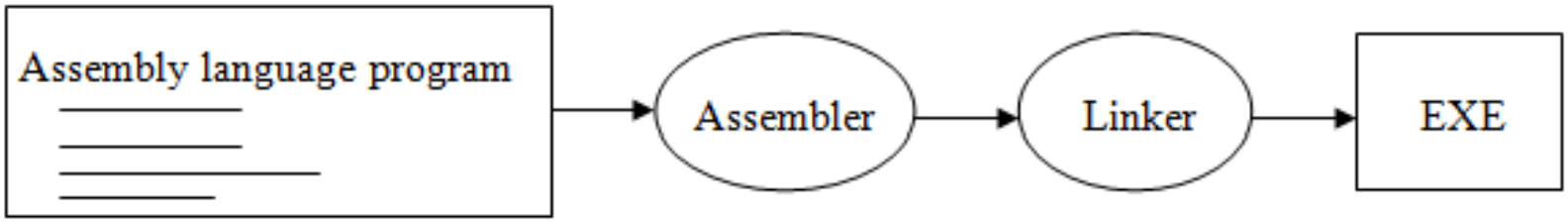


Figure 1. Executable program generation from an assembly source code

Advantages of coding in assembly language are:

- Provides more control over handling particular hardware components
- May generate smaller, more compact executable modules
- Often results in faster execution

Disadvantages:

- Not portable
- More complex
- Requires understanding of hardware details (interfaces)

- **Assembler:**

- An assembler does the following:

- 1. Generate machine instructions
  - evaluate the mnemonics to produce their machine code
  - evaluate the symbols, literals, addresses to produce their equivalent machine addresses
  - convert the data constants into their machine representations

- 2. Process pseudo operations

2. Two Pass Assembler

- A two-pass assembler performs two sequential scans over the source code:
- Pass 1: symbols and literals are defined
- Pass 2: object program is generated
- *Parsing*: moving in program lines to pull out op-codes and operands

- **Data Structures:**
- *Location counter (LC):* points to the next location where the code will be placed
- *Op-code translation table*: contains symbolic instructions, their lengths and their op-codes (or subroutine to use for translation)
- *Symbol table (ST):* contains labels and their values
- *String storage buffer (SSB):* contains ASCII characters for the strings
- *Forward references table (FRT)*: contains pointer to the string in SSB and offset where its value will be inserted in the object code

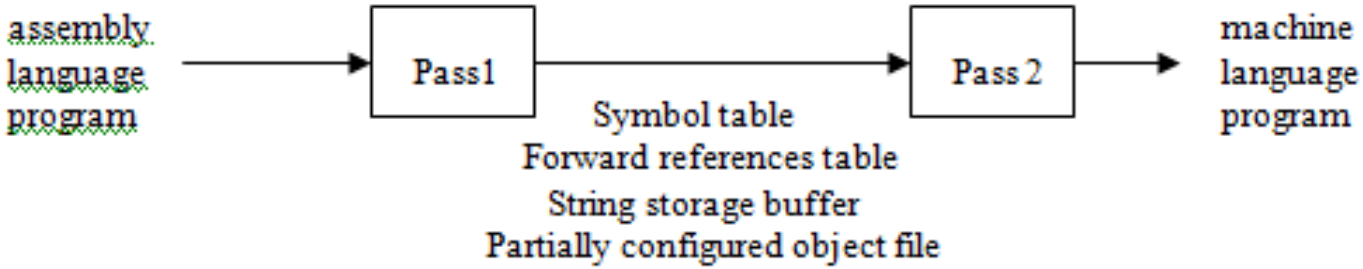


Figure 2. *A simple two pass assembler.*

Example 1: Decrement number 5 by 1 until it is equal to zero.

	Assembly language Program		memory address	object code in memory	
	START	0100H			
	LDA	#5	0100	01	
			0101	00	
			0102	05	
LOOP:	SUB	#1	0103	1D	
			0104	00	
			0105	01	
	COMP	#0	0106	29	
			0107	00	
			0108	00	
	JGT	LOOP	0109	34	
			010A	01	*placed in Pass 1*
			010B	03	
	RSUB		010C	4C	
			010D	00	
			010E	00	
	END				

Op-code Table

Mnemonic	*Addressing mode*	*Opcode*
LDA	immediate	01
SUB	immediate	1D
COMP	immediate	29
LDX	immediate	05
ADD	indexed	18
TIX	direct	2C
JLT	direct	38
JGT	direct	34
RSUB	implied	4C

Symbol Table

Symbol	*Value*
LOOP	0103

Example 2: Sum 6 elements of a list which starts at location 200.

	Assembly language Program		memory address	object code in memory	
	START	0100H			
	LDA	#0	0100	01	
			0101	00	
			0102	00	
	LDX	#0	0103	05	
			0104	00	
			0105	00	
LOOP:	ADD	LIST, X	0106	18	
			0107	01	*placed in Pass 2*
			0108	12	
	TIX	COUNT	0109	2C	
			010A	01	*placed in Pass 2*
			010B	15	
	JLT	LOOP	010C	38	
			010D	01	*placed in Pass 1*
			010E	06	
	RSUB		010F	4C	
			0110	00	
			0111	00	
LIST:	WORD	200	0112	00	
			0113	02	
			0114	00	
COUNT:	WORD	6	0115	00	
			0116	00	
			0117	06	
	END				

Symbol Table

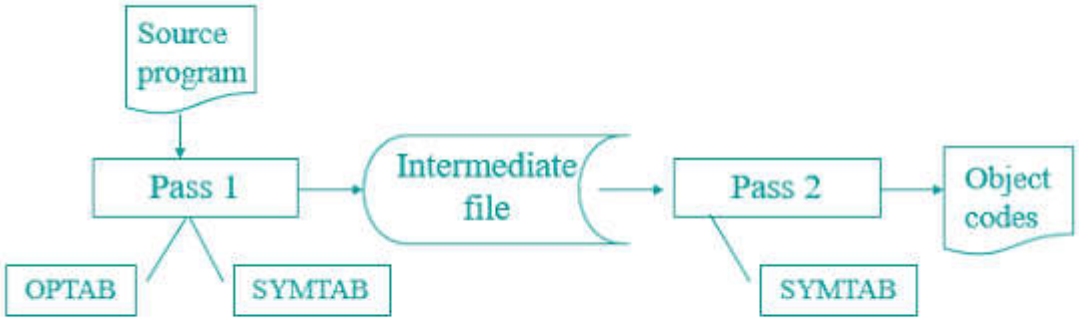
Symbol	*Address*
LOOP	0106
LIST	0112
COUNT	0115

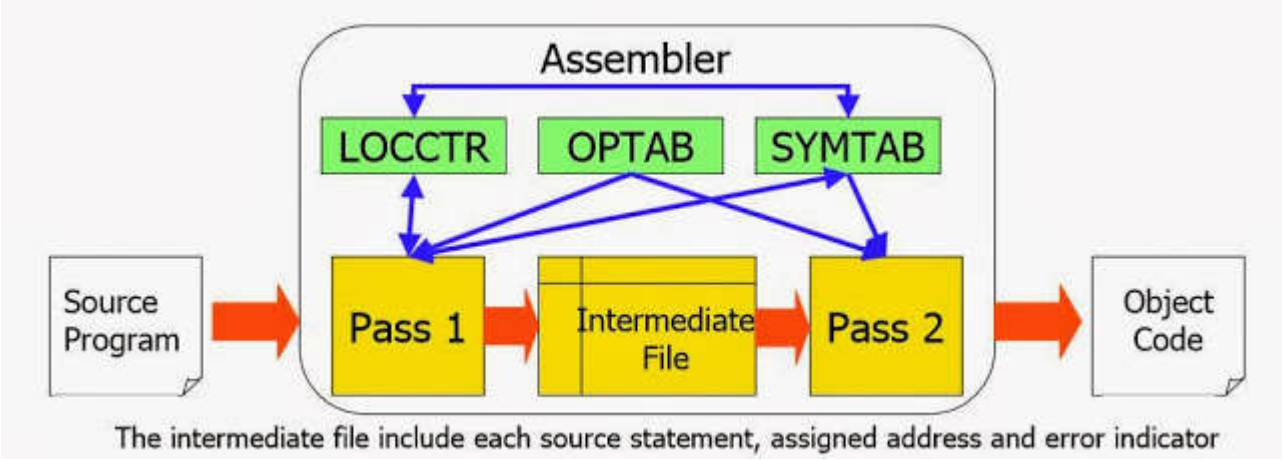
Forward References Table

Offset	*SSB pointer for the symbol*
0007	DC00
000A	DC05

	SSB	
DC00	4CH	ASCII for L,I,S,T
DC01	49H	
DC02	53H	
DC03	54H	
DC04	5EH	ASCII for separation character
DC05		

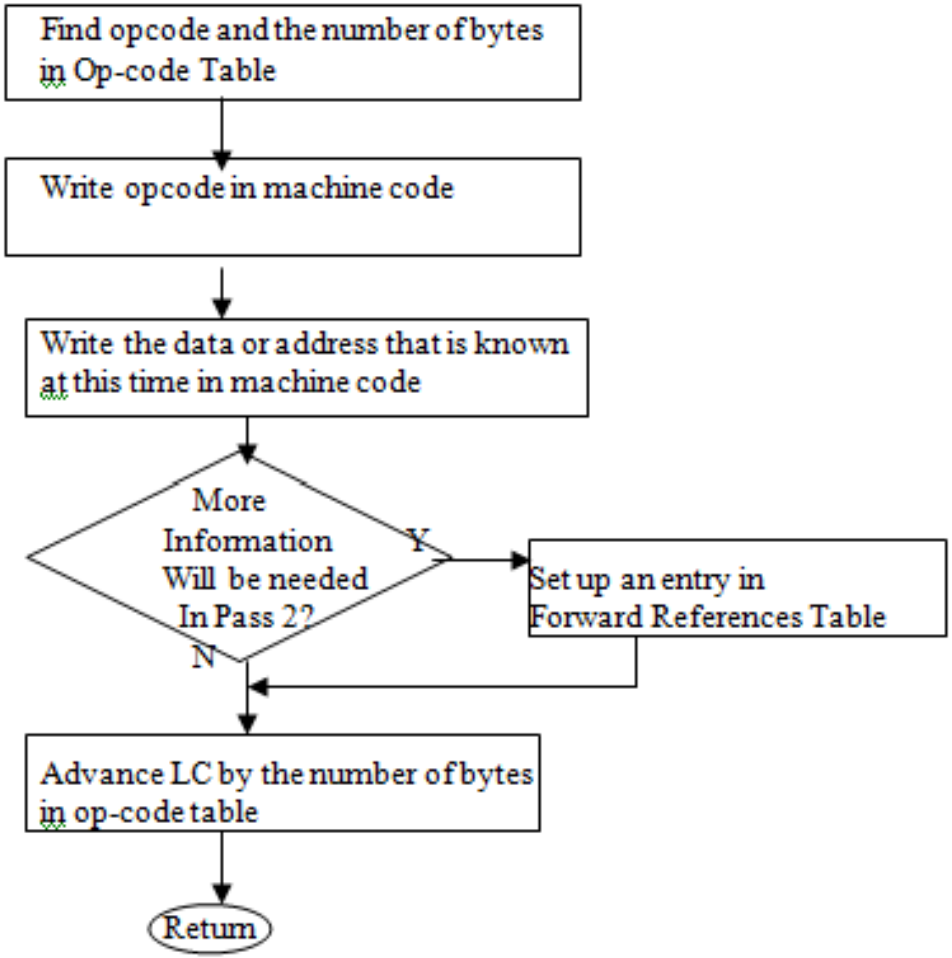
- **Pass1**
- All symbols are identified and put in ST
- All op-codes are translated
- Missing symbol values are marked





The intermediate file include each source statement, assigned address and error indicator

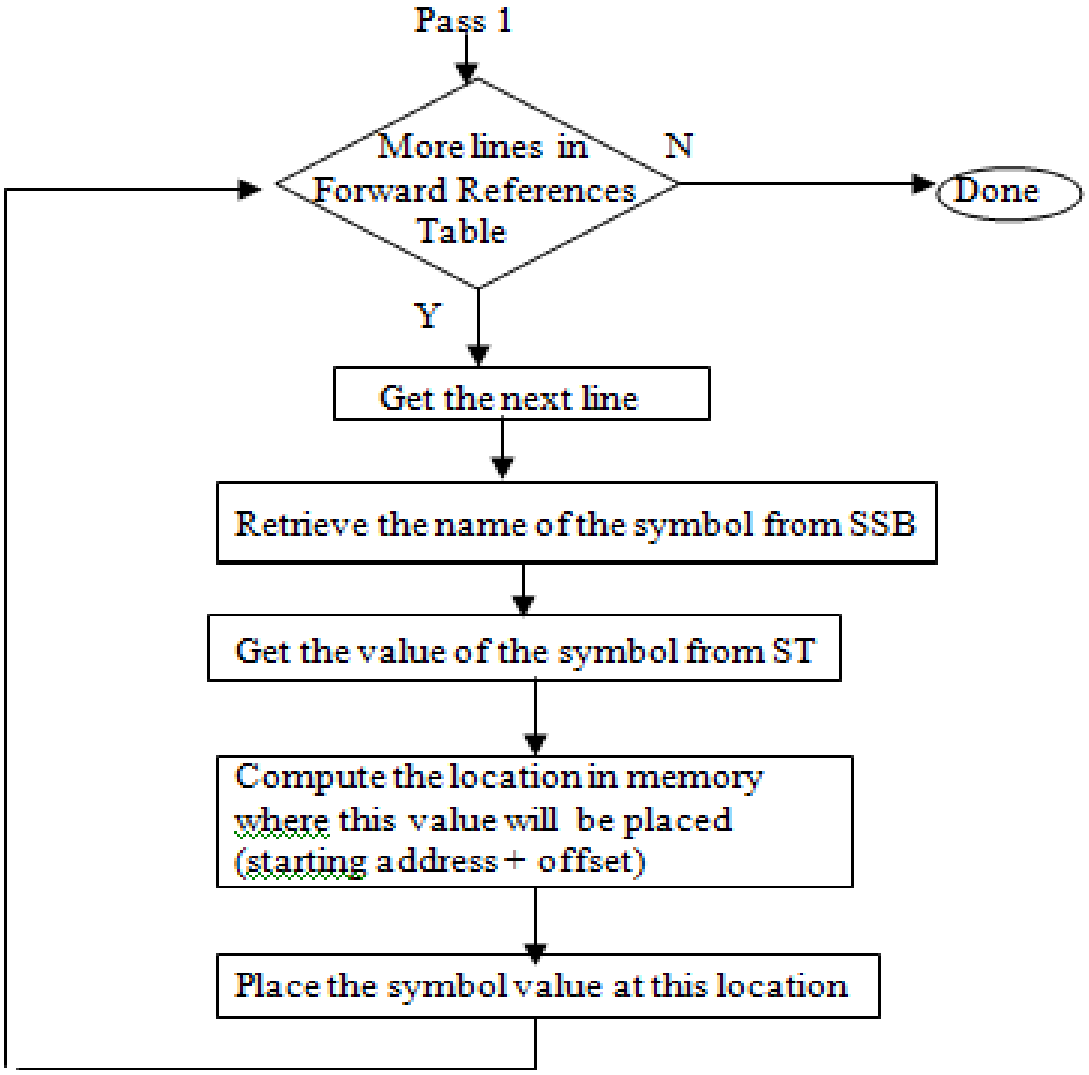
Translator Routine



Flowchart of a translator routine

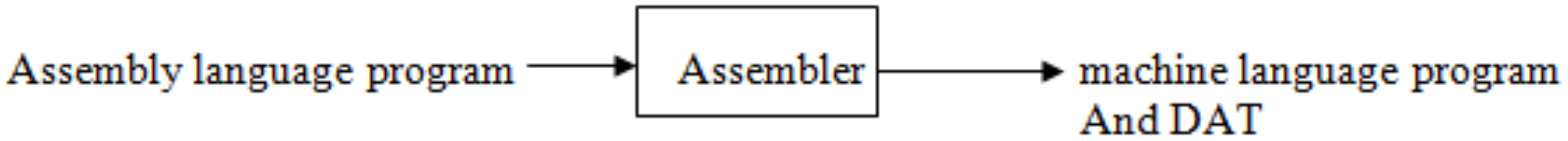
Pass 2

- Fills addresses and data that was unknown during Pass 1.



Second pass of a two-pass assembler.

- **Relocatable Code**
- Producing an object code, which can be placed to any specific area in memory?
- *Direct Address Table (DAT):* contains offset locations of all direct addresses in the program (e.g., 8080 instructions that specify direct addresses are LDA, STA, all conditional jumps...). To relocate the program, the loader adds the loading point to all these locations.



Assembler output for a relocatable code.

DAT

0007
000A
000D

Example 3: Following relocatable object code and DAT are generated for Example 1.

	Assembly language Program		memory address	object code in memory
	START			
	LDA	#0	0000	01
			0001	00
			0002	00
	LDX	#0	0003	05
			0004	00
			0005	00
LOOP:	ADD	LIST, X	0006	18
			0007	00
			0008	12
	TIX	COUNT	0009	2C
			000A	00
			000B	15
	JLT	LOOP	000C	38
			000D	00
			000E	06
	RSUB		000F	4C
			0010	00
			0011	00
LIST:	WORD	200	0012	00
			0013	02
			0014	00
COUNT:	WORD	6	0015	00
			0016	00
			0017	06
	END			

- Forward and backward references in the machine code are generated relative to address 0000. To relocate the code, the loader adds the new load-point to the references in the machine code which are pointed by the DAT.
- **<u>One-Pass Assemblers</u>**
- **Two methods can be used:**
- **- Eliminating forward references**
- Either all labels used in forward references are defined in the source program before they are referenced, or forward references to data items are prohibited.
- - **Generating the object code in memory**
- No object program is written out and no loader is needed. The program needs to be re-assembled every time.

Multi-Pass Assemblers

Make as many passes as needed to process the definitions of symbols.

- **Example 3:**

```
A       EQU     B
B       EQU     C
C       DS      1
```

- 3 passes are required to find the address for A.

Such references can also be solved in two passes: entering symbol definitions that involve forward references in the symbol table. Symbol table also indicates which symbols are dependent on the values of others.

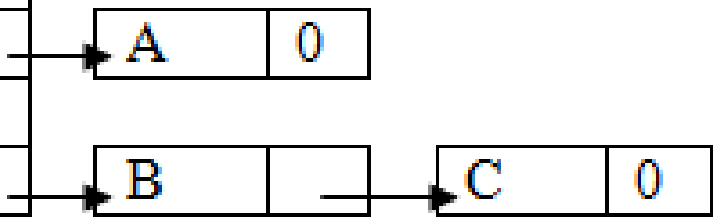
Example 4:

```
A       EQU     B
B       EQU     D
C       EQU     D
D       DS      1
```

At the end of Pass1:

Symbol Table

A	&1	B	0
B	&1	D	
C	&1	D	0
D		200	



After evaluating dependencies:

Symbol Table

A		200	0
B		200	0
C		200	0
D		200	0

THANK YOU...