System Programming

For BCA 3rd Semester

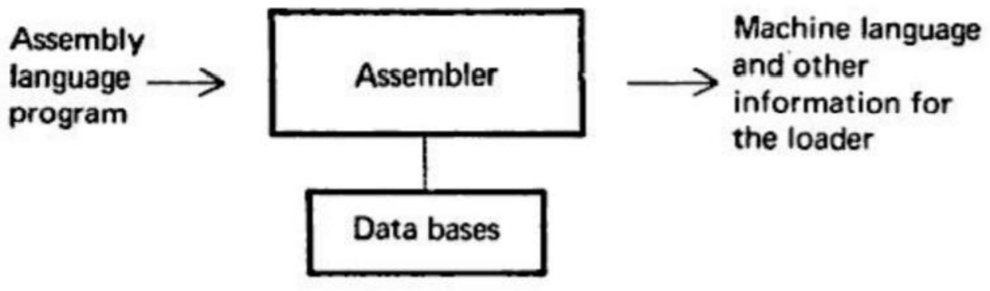
Unit 2

Compiled
By
Sakhi Bandyopadhyay
Dept. of Computer Science & BCA,
Kharagpur College,
Kharagpur 721305

Assembler:

• An assembler is a program that takes computer instruction and converts them into a pattern of bits that the computer processor can use to perform its basic operation.

• The assembler is responsible for translating the assembly language program into machine code. When the source program language is essentially a symbolic representation for a numerical machine language, the translator is called assembler and the source language is called an assembly language.


Assembly language program
Assembler
Machine language and other information for the loader
Data bases

Elements of Assembly Language Programming:

An assembly language provides the following three basic facilities that simplify programming:

1. Mnemonic operation codes: The mnemonic operation codes for machine instructions (also called mnemonic opcodes) are easier to remember and use than numeric operation codes. Their use also enables the assembler to detect use of invalid operation codes in a program.

2. Symbolic operands: A programmer can associate symbolic names with data or instructions and use these symbolic names as operands in assembly statements. This facility frees the programmer from having to think of numeric addresses in a program. We use the term symbolic name only in formal contexts; elsewhere we simply say name.

3. Data declarations: Data can be declared in a variety of notations, including the decimal notation. It avoids the need to manually specify constants in representations that a computer can understand, for example, specify -5 as (11111011)2 in the two's complement representation.

Statement format:

An assembly language statement has the following format:

[Label] <Opcode> <operand specification>[,<operand specification>..]

where the notation [..] indicates that the enclosed specification is optional. If a label is specified in a statement, it is associated as a symbolic name with the memory word generated for the statement. If more than one memory word is generated for a statement, the label would be associated with the first of these memory words.

<operand specification> has the following syntax:

<symbolic name> [± <displacement>] [(<index register>)]

Thus, some possible operand forms are as follows:

- The operand AREA refers to the memory word with which the name AREA is associated.
- The operand AREA+5 refers to the memory word that is 5 words away from the word with the name AREA. Here '5' is the displacement or offset from AREA.
- The operand AREA(4) implies indexing the operand AREA with index register 4—that is, the operand address is obtained by adding the contents of index register 4 to the address of AREA.
- The operand AREA+5 (4) is a combination of the previous two specifications.

Types of Assembly Statements:

1. Imperative statement

- An imperative statement indicates an action to be performed during the execution of the assembled statement.
- Each imperative statement typically translates into one machine instruction.
- These are executable statements.
- Some example of imperative statement are given below
  MOVER BREG,X
  STOP
  READ X
  PRINT Y
  ADD AREG,Z

2. Declaration statement

- Declaration statements are for reserving memory for variables.
- The syntax of declaration statement is as follow:
  - [Label] DS <constant>
  - *Label+ DC '<value>'

DS: stands for Declare storage, DC: stands for Declare constant.

The DS statement reserves area of memory and associates name with them.

A DS 10

Above statement reserves 10 word of memory for variable A.

- The DC statement constructs memory words containing constants.
  ONE DC ‘1’
  Above statement associates the name ONE with a memory word containing the value ‘1’
- Any assembly program can use constant in two ways- as immediate operands, and as literals.
- Many machine support immediate operands in machine instruction. Ex: ADD AREG, 5
- But hypothetical machine does not support immediate operands as a part of the machine instruction. It can still handle literals.
- A literal is an operand with the syntax=’<value>’. EX: ADD AREG,=’5’
- It differs from constant because its location cannot be specified in assembly program.

3. Assembler Directive

- Assembler directives instruct the assembler to perform certain action during the assembly program.

  a. START
  - This directive indicates that first word of machine should be placed in the memory word with address <constant>.
  - START <Constant>
  - Ex: START 500
  - First word of the target program is stored from memory location 500 onwards.

  b. END
  - This directive indicates end of the source program.
  - The operand indicates address of the instruction where the execution of program should begin.
  - By default it is first instruction of the program.
  - END <operand 2>
  - Execution control should transfer to label given in operand field.

<u>Basic function of Assembler:</u>

• Translate mnemonics opcodes to machine language.

• Convert symbolic operands to their machine addresses.

• Build machine instructions in the proper format

• Convert data constants into machine representation.

• Error checking is provided.

• Changes can be quickly and easily incorporated with a reassembly.

• Variables are represented by symbolic names, not as memory locations.

• Assembly language statements are written one per line. A machine code program thus consists of a sequence of assembly language statements, where each statement contains a mnemonics.

<u>Advantages:</u>

• Reduced errors

• Faster translation times

• Changes could be made easier and faster.

• Addresses are symbolic, not absolute

• Easy to remember

Disadvantages:

• Assembler language are unique to specific types of computer

• Program is not portable to the computer.

• Many instructions are required to achieve small tasks

• Programmer required knowledge of the processor architecture and instruction set.

Translation phase of Assembler:

The six steps that should be followed by the designer

1. Specify the problem
2. Specify data structure
3. Define format of data structure
4. Specify algorithm
5. Look for modularity
6. Repeat 1 through 5 on modules

Functions / Purpose of Assembler:

An assembler must do the following

1. Generate instruction

a. Evaluate the mnemonics in the operation field to produce the machine code

b. Evaluate the subfield-fine the value of each symbol. Process literals and assign addresses.

2. Process pseudo ops

a. Pass 1 (Define symbol and literals)

i. Determine length of machine instruction (MOTGET)

ii. Keep track of location counter (LC)

iii. Remember value of symbol until pass 2 (STSTO)

iv. Process some pseudo ops(POTGET1)

v. Remember literal (LITSTO)

b. Pass 2 (Generate object Program)

i. Look up value of symbol (STGET)

ii. Generate instruction (MOTGET2)

iii. Generate data (for DS, DC and Literal)

iv. Process pseudo ops (POTGET2)

Design data structure for assembler design in Pass-1 and Pass-2 with flow chart:

Pass -1

1. Input source program

2. A location counter used to keep track of each instruction location.

3. A table, the machine –operation table (MOT) that indicate the symbolic mnemonics for each instruction and its length (tow, four or six bytes)

4. A table, the pseudo operation table (POT) that indicate the symbolic mnemonics and action to be taken for each pseudo-op in pass-1

5. A table, the literal table (LT) that is used to store each literal encounter and its corresponding assigned location.

6. A table, the symbol table (ST) that is used to store each label and its corresponding value.

7. A copy of the input to be used later by Pass-2. This may be stored in a secondary storage device.

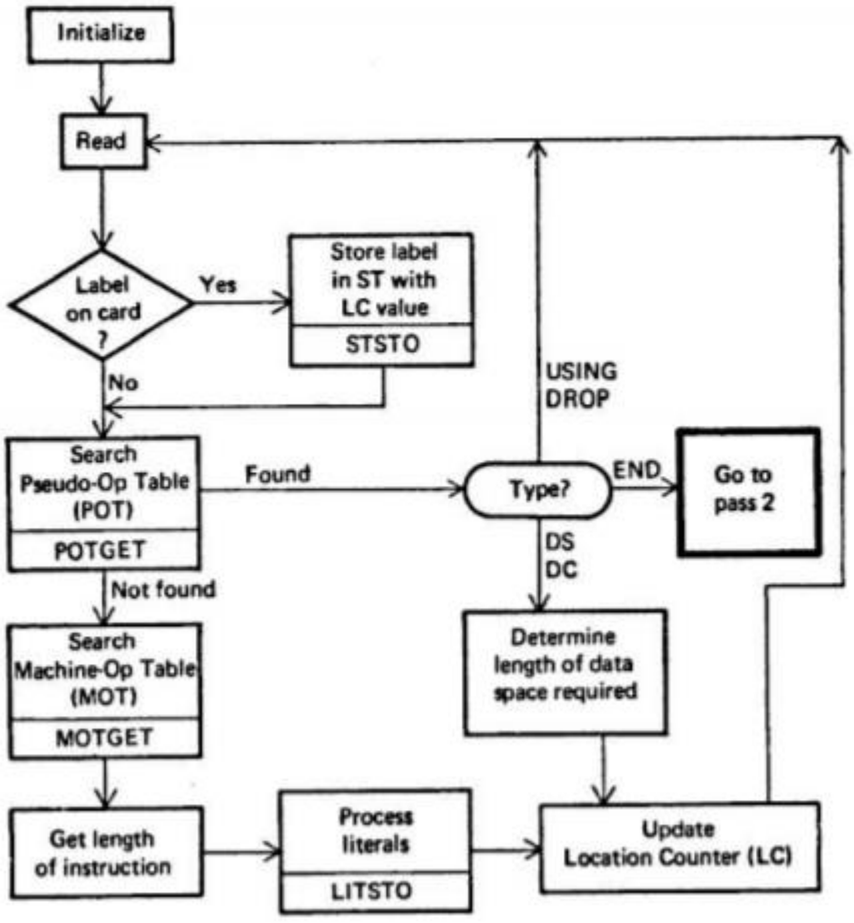

Initialize
Read
Label on card ?
Yes
Store label in ST with LC value
STSTO
No
USING
DROP
Search Pseudo-Op Table (POT)
POTGET
Found
Type?
END
Go to pass 2
DS
DC
Not found
Search Machine-Op Table (MOT)
MOTGET
Determine length of data space required
Get length of instruction
Process literals
LITSTO
Update Location Counter (LC)

FIGURE 3.3 Pass 1 overview: define symbols

Pass-2

1. Copy of source program input to pass-1

2. Location counter

3. A table the MOT that indicates for each instruction

a. Symbolic

b. Mnemonics

c. Length

d. Binary machine op-code

e. Format (RR, RS, RX, SI, SS)

4. A table the POT that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in Pass-2

5. The ST prepare by Pass-1, containing each label and its corresponding value.

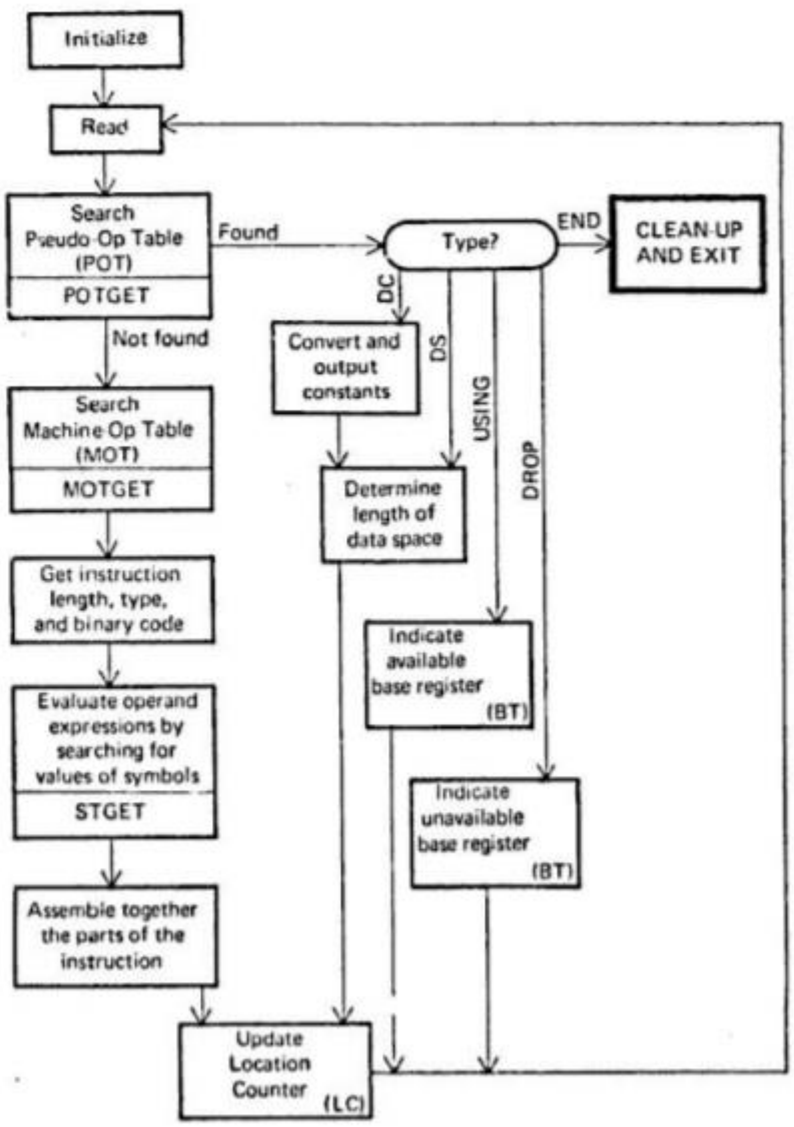
6. A table, BT that indicate which register are currently specified by base register by USING pseudo-ops and what are the specified contents of these register.

7. A work space INSR that is to hold each instruction as its various parts are being assembled together.

8. A workspace PRINT LINE used to produce a printed listing

9. A workspace PUNCH CARD used prior to actual outputting for converting assembled instruction into the format needed by the loader.

10. An output deck of assembled instruction in the format needed by the loader.



Initialize
Read
Search Pseudo-Op Table (POT)
POTGET
Found
Type?
END
CLEAN-UP AND EXIT
Not found
DC
DS
USING
DROP
Convert and output constants
Search Machine-Op Table (MOT)
MOTGET
Determine length of data space
Get instruction length, type, and binary code
Indicate available base register (BT)
Evaluate operand expressions by searching for values of symbols
STGET
Indicate unavailable base register (BT)
Assemble together the parts of the instruction
Update Location Counter (LC)

FIGURE 3.4 Pass 2 overview: evaluate fields and generate code