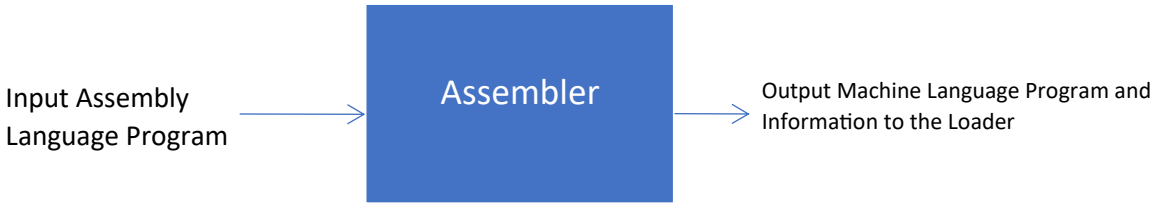
Module-2

Assemblers

Syllabus

2		**Assemblers**	7
	2.1	Elements of Assembly Language programming, Assembly scheme, pass structure of assembler, Assembler Design: Two pass assembler Design and single pass Assembler Design for X86 processor, data structures used.	

- An assembly language is machine dependent, low level programming language



- **Basic Features of Assembly Language Program**

1. Mnemonic operation code
2. Symbolic operands
3. Data declaration

Elements of Assembly Language Programming

→ An assembly language is a machine dependent.
→ It is a low level programming language which is specific to a certain computer system.
→ It provides three basic features which simplify programming:

1) Mnemonic operation codes: a) It eliminates the need to memorize numeric operation codes.
b) It enables the assembler to provide helpful diagnostics (example - Misspelt operation codes)
2) Symbolic operands: Symbolic names can be associated with data or instructions.
a) Symbolic names can be used as operands in assembly statements
b) The assembler performs memory bindings to these names
c) The programmer need not ~~remember~~ know any details of the memory bindings performed by the assembler.
3) Data declaration: Can be done using decimal notation.

Statement format:

[Label] <opcode> <operand spec>[,<operand spec>...]

(<operand spec> is optional)

If a label is specified in a statement, it is associated as a symbolic name with the memory words generated for the statement.

Syntax for <operand spec>

<symbolic name> [+<displacement>] [<index register>]

Examples of operand forms AREA, AREA+5, AREA(4) and AREA+5(4)

a. AREA → It refers to the memory word with which the name AREA is associated

b. AREA+5 → It refers to the memory word 5 words away from the word with the name AREA. Here, 5 is the displacement or offset from AREA

c. AREA(4) → The operand address is obtained by adding the contents of index register 4 to the address of AREA.

AREA+5(4) → It is the combination of b and c

A Simple assembly language

→ Each statement has two operands.
→ First operand is always a register which can be any one of AREG, BREG, CREG, and DREG.
→ Second operand is ~~alwa~~ refers to a memory word using a symbolic name and an optional displacement
→ Indexing is not permitted.

Mnemonic operation codes

Instruction opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	First operand is modified and condition code is set
02	SUB	
03	MULT	
04	MOVER	Register ← Memory move
05	MOVEM	Memory ← Register move
06	COMP	sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	First operand is not used
10	PRINT	

MOVER : The second operand is the source operand and the first operand is the target operand

MOVEM : The first operand is the source operand and the target is the second operand

All arithmetic is performed in a register i.e the result replaces the contents of a reegister and sets a condition code.

COMP : Comparison instruction sets a condition code analogous to a subtract instruction without affecting the values of its operands.

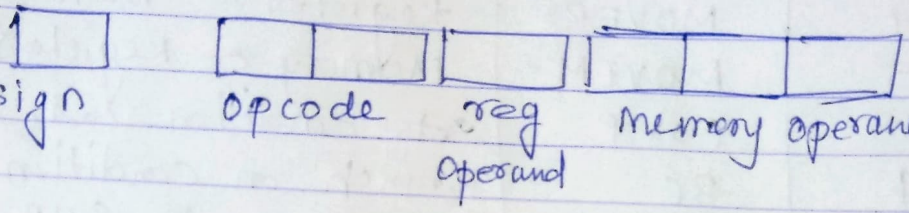
BC : The condition code can be tested by a branch on condition instruction.
The assembly statement :

BC <condition code spec>, <memory address>

It transfer control to the memory word with the <memory address> if the current value of condition code matches <condition code spec>

In machine language program, we show all addresses and constants in decimal rather than in octal or hexadecimal

Instruction Format



Sign is not a part of the instruction.

The condition code in BC statement is encoded into the first operand position using the codes 1 to 6: ~~for~~ for LT, LE etc.
LT=1, LE=2, EQ=3, GT=4, GE=5, ANY=6

	Assembly Language Program		Machine Language Program			
	START	101				
	READ	N	101)	+ 09	0	113
	MOVER	BREG, ONE	102)	+ 04	2	115
	MOVEM	BREG, TERM	103)	+ 05	2	116
AGAIN	MULT	BREG, TERM	104)	+ 03	2	116
	MOVER	CREG, TERM	105)	+ 04	3	116
	ADD	CREG, ONE	106)	+ 01	3	115
	MOVEM	CREG, TERM	107)	+ 05	3	116
	COMP	CREG, N	108)	+ 06	3	113
	BC	LE, AGAIN	109)	+ 07	2	104
	MOVEM	BREG, RESULT	110)	+ 05	2	114
	PRINT	RESULT	111)	+ 10	0	114
	STOP		112)	+ 00	0	000
N	DS	1	113)			
RESULT	DS	1	114)			
ONE	DC	'1'	115)	+ 00	0	001
TERM	DS	1	116)			
	END					

Assembly Language Statements

An assembly program consists of three kinds of statements

1. Imperative statements
2. Declaration statements
3. Assembler Directives

1. Imperative Statements

→ It indicates the action to be performed during the execution of the assembled program.

→ Each imperative statement translates into one machine instruction.

e.g. →

```
      START  101
      READ   N
```

↓ Imperative statement in assembly language

```
101) + 09 03 113
```

↓ Machine Instr.

2. Declaration Statements

Syntax

```
[Label]   DS   <constant>
[Label]   DC   '<value>'
```

DS → Declare storage : It reserves areas of memory and associates the names with them.

example: →

```
A   DS   1
G   DS   200
```

The first statement reserves a memory area of 1 word and associates the name A with it.

The second statement reserves a block of 200 memory words. The first word name G is associated with first word of the block other words of the block can be accessed through offsets from G. like G+5 is the sixth word.

DC → Declare constant: It constructs memory words containing constants.

e.g. → ONE DC '1'

This statement associates the name ONE with a memory word containing the value '1'

→ The program can declare constants in different forms like decimal, binary, hexadecimal.
→ The assembler converts them to appropriate internal form.

Literals

It is an operand with the syntax ='<value>'

Difference bet^n const and literal:

Location of a literal can not be specified in the assembly program.
Hence, its value is not changed during the execution of the program.

example: ADD AREG,='5' ⇒ ADD AREG, FIVE

use of literal 5

FIVE DC '5'

The assembler allocates a memory word to contain the value of the literal and replaces the use of literal in a statement by an operand expression referring to this word.
The name and address of this word is not known to the programmer hence the value of the literal is protected.

Assembler Directives

Assembler directives instruct the assembler to perform certain actions during the assembly of a program.

e.g. → START <constant>

This directive indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <constant>

Design specification of an Assembler

Four steps:

1. Identify the information necessary to perform a task
2. Design a suitable data structure to record the information
3. Determine the processing necessary to obtain and maintain the information
4. Determine the processing necessary to perform the task.

Two phases:

1. Synthesis Phase
2. Analysis Phase

Synthesis Phase

.. MOVER BREG, ONE

To synthesize the machine instruction from this statement the following information is required:

i) Address of the memory word with which name ONE is associated
ii) Machine operation code corresponding to the mnemonic MOVER

→ This information is made available by the Analysis Phase as it depends on source program.

→ This information can be determined by the synthesis phase because it is not dependent on the source program

Two data structures are used :-

Data Structures

1. Symbol Table 2. Mnemonics Table

Symbol Table

→ Each entry has two primary fields : i) name and ii) address
→ The table is built by analysis phase

Mnemonics Table

→ An entry has two primary fields : i) Mnemonics and ii) opcode.
→ The synthesis phase uses these tables to obtain the machine address with which a name is associated and the machine opcode corresponding to a mnemonic respectively

Analysis Phase

→ The main function of these phase is the building of the symbol table.
→ For this purpose, it must determine the addresses with which the symbolic names used in the program are associated.
→ It is possible to determine some addresses directly, like the address of the first instruction in the program but other addresses must be inferred.
→ To determine the address of N, we must fix the address of all program element preceding it. This function is called memory allocation.

Data structure Used

1) Location Counter (LC)

→ It is used to implement memory allocation.
→ It contains the address of the next memory word in the target program.
→ It is initialized to the constant specified in the START statement.

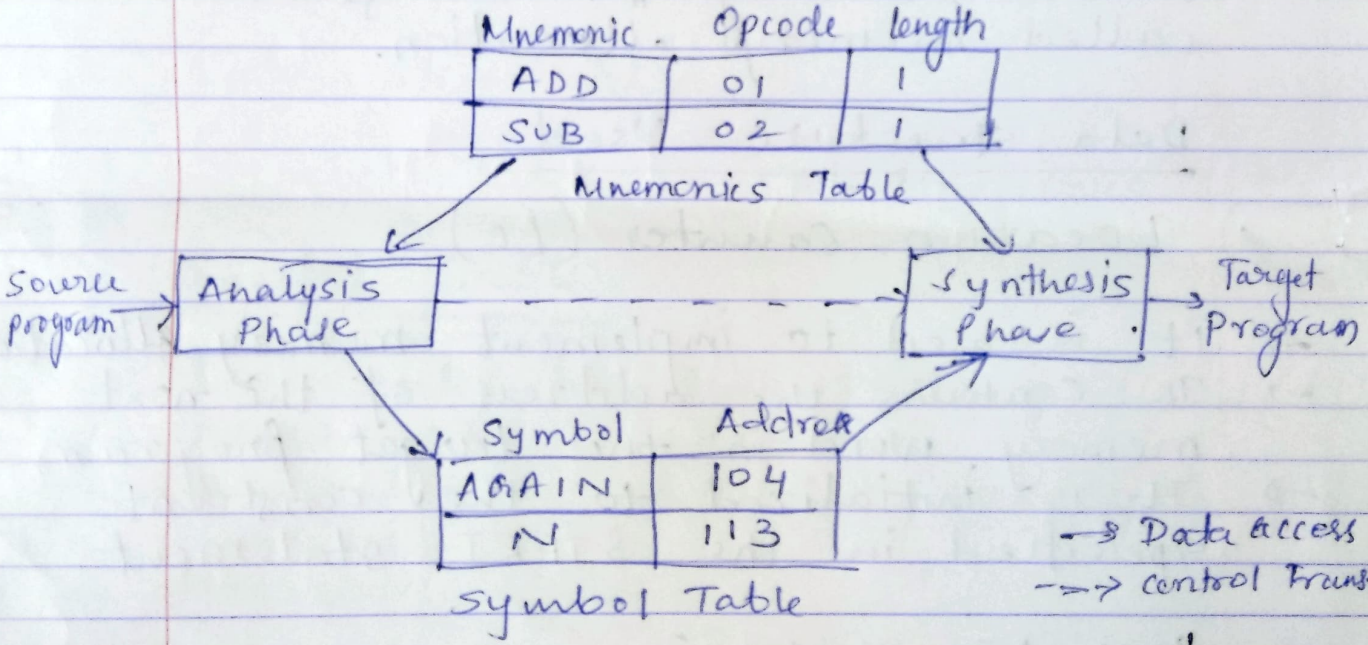
How LC works ?

Whenever the analysis phase sees a label in an assembly statement, it enters it and the contents of LC as a new entry in the symbol table. Then the no. of memory words required by the assembly statements is found and the contents of LC is updated.
This ensures that LC points to the next memory word in the target program.

LC processing

To update the contents of LC, analysis phase needs to know the lengths of different instructions. This information depends on the assembly language.
The Mnemonics table can have a field as 'length' which can hold the length of the instructions.
This process of maintaining the LC is known as 'LC processing'.

Example of data structure used



Mnemonics Table is accessed by both analysis and synthesis phase.

Symbol Table is constructed during analysis phase and used during synthesis phase

Single Pass Translation:

1. LC processing is performed
2. Symbols defined in the program are entered in the symbol table
3. The operand field of an instruction containing a forward reference is left blank initially.
4. The address of the forward referenced symbol is put into this field when its definition is encountered.

This is the forward reference problem

example: MOVER BREG, ONE

The instruction corresponding to the statement can be only partially synthesized ~~once~~ since ONE is a forward reference.
Hence the instruction opcode and address of BREG will be assembled to reside in location 101.

Two pass translation

→ It can handle forward references easily

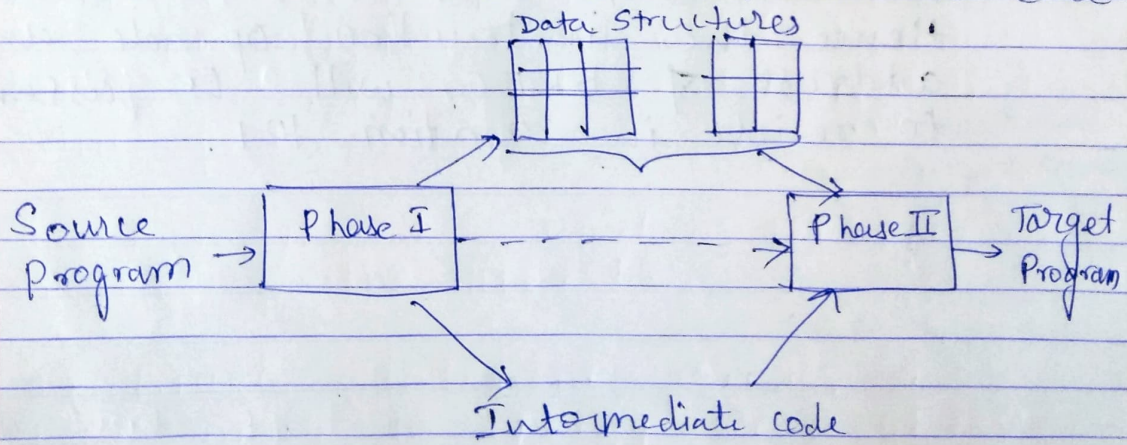
→ Pass 1 :

→ i) LC processing is performed in first pass
ii) Symbols defined in the program are entered into the Symbol table.

Pass 2 :

→ This pass synthesizes the target form using the address information found in the Symbol table.

The first pass constructs an intermediate representation of the source program for use by second pass.
This intermediate representation consists of two main components :-
i) Data structures - Symbol table
ii) Processed form of the source program which is called as intermediate code.



Design of a two pass assembler

Pass 1 performs analysis of the source program and synthesis of intermediate representation.
Pass 2 processes the intermediate representation to synthesize the target program.

Tasks performed by two pass assembler

Pass I:
1. Separate the symbol, mnemonic opcode and operand fields
2. Build the symbol table
3. Perform LC processing
4. Construct Intermediate representation

Pass II: Synthesize the target program.

To understant the design details of assembler passes, advanced assembler directives likes ORIGIN, LTORG needs to be understood and its influence on LC processing is also required

ORIGIN

Syntax: ORIGIN <address spec>

<address spec> is an <operand spec> or <constant>
This directive indicates that LC should be set to the address given by <address spec>
If the target program does not consist of consecutive memory words then ORIGIN statement is useful because of its ability to use <operand spec> which allows to do LC processing in relative manner rather than absolute manner.

Example:

```
1        START   200
4   LOOP
17  LAST  STOP                    216
18        ORIGIN  LOOP+2          204
20        ORIGIN  LAST+1          217
```

The start address of the program is 200
Statement No. 18 address i.e ORIGIN LOOP+2, sets LC value to 204 because LOOP in statement no. 4 has address 202 (202+2=204)
Statement No. 20, ORIGIN LAST+1 sets LC value to 217 because LAST is statement no. 17 has address 216 (216+1=217)

LTORG

i) A literal can be treated as a <value> if it is in a DC (declare constant) statement i.e a memory word containing the value of the literal is formed

ii) This memory word is used as the operand in place of the literal

→ The LTORG statement permits a programmer to specify where literals should be placed.
→ By default, assembler places the literals after END statement.
→ The assembler allocates memory to the literals of a literal pool.
→ The pool contains all literals used in the program since the start or since last LTORG statement.

Example.

```
1     START   200
2     MOVER   AREG, ='5'        200)
:       :                        :
6     ADD     CREG, ='1'        204)
:       :
13    LTORG
          ='5'                  211)
          ='1'                  212)
:       :                        :
15    NEXT SUB  AREG, ='1'      214)
:       :
24    END
25          ='1'                219)
```

The literals are present in line no. 2 and 6 i.e ='5' and ='1' (those same) These are added to the literal pool in statement i.e. 2 and 6 respectively. The first LTORG statement in line no. 13 allocates the addresses 211 ~~to the value~~ '5' and 212 to the value '1'.
A new literal pool is now started. The value ='1' is put in the pool in line no. 15. This value is allocated the address 219 while processing the END statement

Example of Forward Reference

The literal ='1' in line no. 15 refers to location 219 of the second pool of literals rather than location 212 of first pool ~~Thus all~~ Same is the case with ='5' in line no. 2 (location 211) and ='1' in line no. 6 (location 212)
Thus all references to literals are forward references by definition.

EQU

Syntax: <symbol> EQU <address spec>

where <address spec> is an <operand spec> or <constant>

The EQU statement defines the symbol to represent <address spec>.

EQU associates the name <symbol> with <address space>

Example:

```
1     START  200
2     :
4  LOOP MOVER  AREG, A
12    BC     ANY_NEXT
16    BC     LT, BACK
:
22 BACK EQU  LOOP
```

200)
202) +04 1 217
210) +07 6 214
215) +07 1 202

In line no. 22, EQU associates the name BACK to represent the symbol, LOOP ~~whose reference is in line 16~~
So, line no. 16 is assembled as

+07 1 202

Instruction opcode ↙ ↓ ↘
location of BC Register opcode location of LOOP in line no. 4
=AREG, =1
~~AREGET~~

The following data structures are used:

i) OPTAB → It is a table of Mnemonic opcodes and related information
ii) SYMTAB → Symbol table
iii) LITTAB → A table of literals used in the program.

Example

Sample Program:

```
1.  -   START  200
2       MOVER  AREG, ='5'
3       MOVEM  AREG, A
4   LOOP MOVER AREG, A
5       MOVER  CREG, B
6       ADD    CREG, ='1'
-   -   -  -  -
12      BC     ANY, NEXT
13      LTORG
            ='5'
            ='1'
-   -   -  -  -
15  NEXT SUB   AREG, ='1'
16      BC     LT, BACK
17  LAST STOP
18      ORIGIN LOOP+2
19      MULT   CREG, B
20      ORIGIN LAST+1
21  A   DS     1
22  BACK EQU   LOOP
23  B   DS     1
24      END
25          ='1'
```

```
200)  +04  1  211
201)  +05  1  217
202)  +04  1  217
203)  +05  3  218
204)  +01  3  212
-   -   -  -  -
210)  +07  6  214

211)  +00  0  005
212)  +00  0  001
-   -   -  -  -
214)  +02  1  219
215)  +07  1  202
216)  +00  0  000

204)  +03  3  218

217)

218)

219)  +00  0  001
```

The tables or datastructures are as fo

1) OPTAB : It contains the fields mnemonic opcode, class, mnemonic info

a) Mnemonic opcode → contains Instruction mnemonics
b) Class → It indicates whether the ~~mnemonic~~ mnemonic opcode corresponds to an imperative statement (IS) or a declaration statement (DL) or an assembler directive (AD).
c) Mnemonic info → For IS:- the mnemonic info consists of the pair machine opcode, instruction length.
→ For DL :- This field contains id of the routine to handle the declaration statement
→ For AD :- or assembler directive.

Mnemonic opcode	class	mnemonic info
MOVER	IS	(04, 1)
DS	DL	#R#7
START	AD	R#11
—	—	—

OPTAB

2) SYMTAB :- It contains address and length. alongwith the symbol field

Symbol	Address	Length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

SYMTAB

3) LITTAB: It contains literal and address field

LITTAB

literal	address
='5'	
='6,'	
='6,'	

Awareness of different literals pools is maintained using the auxiliary table known as POOLTAB

POOLTAB: It consist of literal no. of starting literal of each pool.

literal no.
#1
#3

Processing

1. Processing starts with label field.
2. If it contains a symbol, the symbol and the value in LC is copied into a new entry of Symtab.
3. The functioning of Pass 1 centers around the interpretation of the OPTAB entry for the mnemonic

4 i) If the class field contains IS then length of the machine instruction is added to LC

ii) This length is also entered in Symtab. for Symbol entry of the symbol defined in the IS statement

5. i) If the class field contains DL or AD then the routine mentioned in the mnemonic info field is called to perform appropriate processing of the statement.

ii) For DL, this routine processes the operand field of the statement to determine the amount

of memory required by the DL statement and appropriately updates the LC and symtab entry of the symbol defined in the statement.

iii) For AD, the called routine would perform appropriate processing which affects the value in LC

6. Next, LITTAB and POOLTAB are used.
7. At any stage, the current literal pool is the last pool in LITTAB.
8. On encountering an LTORG statement, or END statement, literals in the current pool are allocated addresses starting with the current value in LC and LC is appropriately incremented.

~~Flowchart~~ Flowchart ~~2~~ Algorithm of Assembler pass 1 is given in Pg-100

For use in a two pass assembler, intermediate code forms are important

Intermediate Code Forms

Two criteria to choose intermediate code :-
i) Processing efficiency
ii) Memory economy

Intermediate code consists of a set of IC units, each IC unit consisting of the following three fields :-
i) Address
ii) Representation of Mnemonic opcode
iii) Representation of operands

Address	opcode	operands

Due to tradeoff between processing efficiency and memory economy, two variants of intermediate codes are there:

Variant I and Variant II

Assumption: The address field is assumed to contain identical information in both variants.

Variant I

→ The first operand is represented by a single digit number which is a code for a register. (1 - AREG, 2 - BREG, 3..., 4...) or the condition code itself (1-6 for LT-ANY)

→ The second operand is a memory operand which is represented by a pair of the form (operand class, code) where operand class is constant (C), symbol (S) or literal (L). The code field is the internal representation of the constant (if operand class is C)

example

	START 200	(AD, 01)	(C, 200)
	READ A	(IS, 09)	(S, 01)
LOOP	MOVER AREG, A	(IS, 04)	(1) (S, 01)
	⋮	⋮	
	SUB AREG, '=1'	(IS, 02)	(1) (L, 01)
	BC GT, LOOP	(IS, 07)	(4) (S, 02)
	STOP	(IS, 00)	

Variant II

→ Operand fields of the source statements are selectively replaced by their processed forms.

→ For declarative statements and assembler directives, processing of operand fields is essential to support LC processing

→ For imperative statements, the operand field is processed only to identify literal referen

Example:

	START	200	(AD, 01)	(C, 200)
	READ	A	(IS, 09)	
LOOP	MOVER	AREG, A	(IS, 04)	AREG, A
	:		:	
	SUB	AREG, ='1'	(IS, 02)	AREG, (L, 01)
	BC	GT, LOOP	(IS, 07)	GT, LOOP
	STOP		(IS, 00)	
A	DS	1	(DL, 02)	(C, 1)
	LTORG		(AD, 05)	
	=			

Algorithm for Pass II of the assembler is given in page -107 of the book.

In Pass II algorithm it is assumed that the target code is to be assembled in the area named code_area

There might be some minor changes to suit the intermediate code forms being used i.e Variant I or Variant II

Design of Single Pass Assembler

Intel 8088 Assembler

- The LC processing differs from pass 1 of two pass assembler.
- Here, the unit of memory allocation is a byte
- → LC alignment concept is used. For example start address must be even address i.e a word requires alignment on event boundaries.
- → Allocation of memory for a statement and entering its label in the symbol table is performed after LC alignment.

Data Structures used

1. Mnemonics Table (MOT)
   - → It is hash organized
   - → It contains the following fields: mnemonic opcode, machine opcode, alignment/format info and routine id.
   - → Alignment/format info is specific to a give routine
   - → Routine id field of an entry specifies the routine which processes that Opcode.

Mnemonic Opcode	Machine Opcode	Alignment/format Info	Routine id
JNE	75H	00H	R2

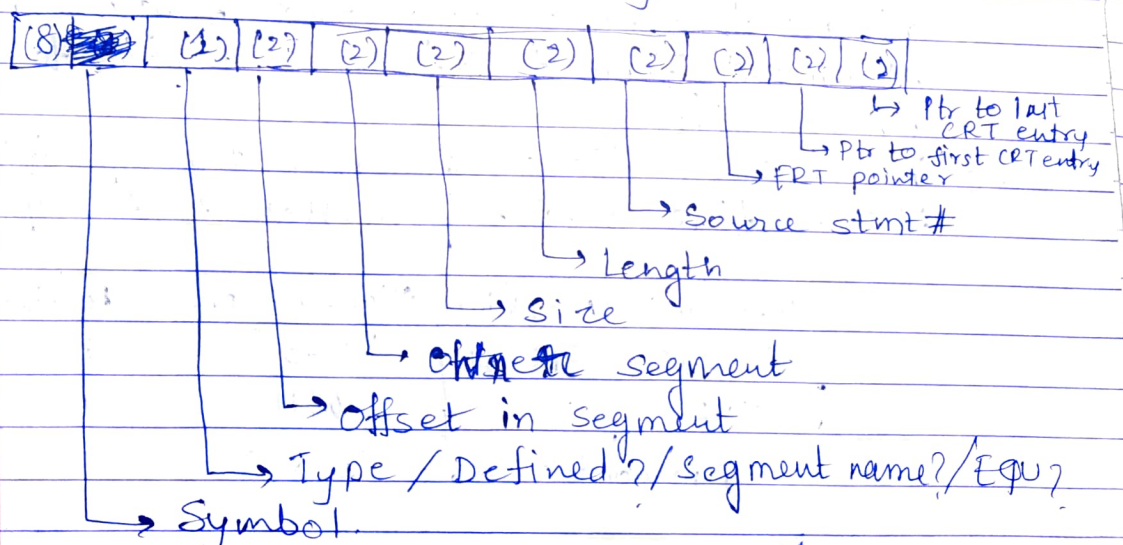
Mnemonics table (MOT)

The code 00H for routine R2 implies that only one instruction format with self-relative displacement is supported.
If the format info is FFH for the same routine then it implies that all formats are supported. Hence the routine must decide which ~~formats~~ machine opcode to use.

2. Symbol table (SYMTAB)

→ It is also hash organized
→ It contains all relevant information about symbols defined and used in the source program.
→ The fields are: i) Symbol
ii) Owner segment - It indicates id of the segment in which a symbol is defined.
iii) Type - For non EQU, the TYPE field indicates the alignment information.
- For EQU symbol, this field indicates whether the symbol is numeric value or text value.

iv) Offset - This field is used to accomodate the value.
v) Size vi) Length vii) Source stmt #
viii) FRT pointer ix) Ptr to first CRT entry
x) Pointer to last CRT entry



Symbol Table (SYMTAB)

3. SRTAB (Segment Register Table Array) — ARRAY

→ It contains upto four entries, one for each register.
→ The current SRTAB exists in the last entry of SRTAB_ARRAY
→ SRTAB's are accessed by their entry number in SRTAB_ARRAY

~~4. Forward Reference Table (FRT)~~

Segment Register (1)	Segment Name (2)	
DO(ES)	23	SRTAB #1
		SRTAB #2
⋮		

Segment Register Table Array (SRTAB_ARRAY)

4. Forward Reference Table (FRT)

→ It contains a set of linked lists
→ Information concerning forward references to a symbol is organized in these linked l
→ The FRT pointer of SYMTAB entry points to head of this list.
→ Each FRT entry contains SRTAB# to be used to assemble the forward reference
→ FRT also contains Usage code, instructior address which indicates where and how the reference is assembled

Pointer (2)	SRTAB# (1)	Instruction Address (2)	Usage code (1)	Source stmt # (2)

Forward Reference Table (FRT)

5. Cross Reference Table (CRT)

→ A cross reference directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order of the statement numbers

→ The CRT is used by the assembler to collect the information concerning references to all symbols in the program.

→ Each SYMTAB entry points to the head and tail of a linked list in the CRT.

Pointer (2)	Source stmt # (2)

Cross Reference Table (CRT)

Algorithm for Single pass assembler is given in page no. 128 of the book