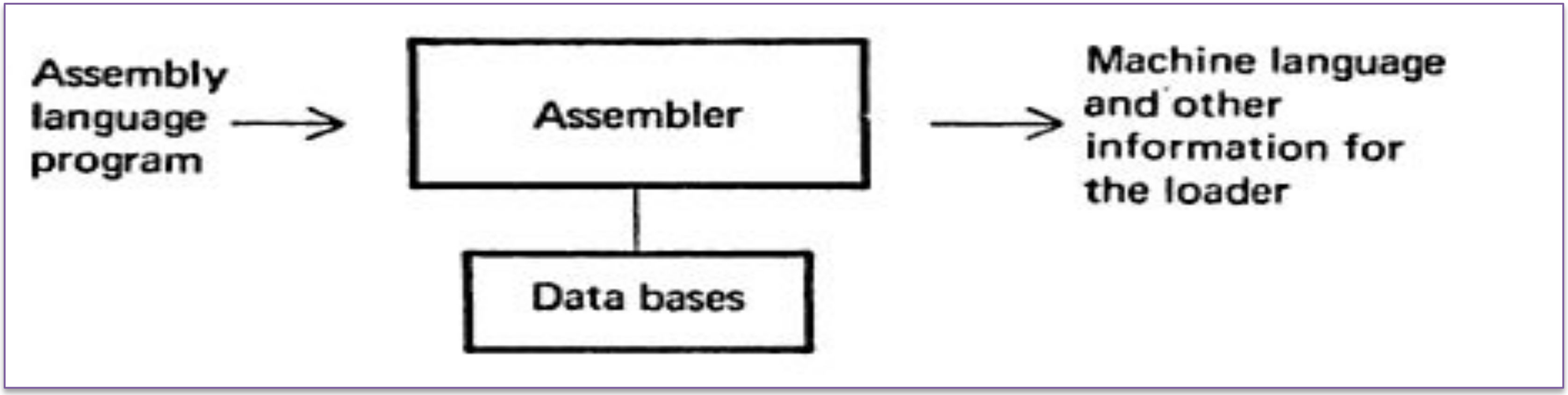
ASSEMBLER

Assembler



- **Basic Assembler functions:**
  - Translating mnemonic language to its equivalent object code.
  - Assigning machine addresses to symbolic labels.

ASSEMBLER

General Instruction Format :

[Label *]	[opcode]	[operand1]*, [operand2]*

1. Label:- Is optional.
2. Opcode:- Symbolic opcode
3. Operand:- Symbolic name (Register or Memory variable)

A simple assembly language

I am always a register..!!!

I am always a symbolic name..!!!

Mnemonic Operation Codes

Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stop Execution
01	ADD	Op1 Op1+ Op2
02	SUB	Op1 Op1 – Op2
03	MULT	Op1 Op1* Op2
04	MOVER	CPU Reg Memory operand
05	MOVEM	Memory CPU Reg
06	COMP	Sets Condition Code
07	BC	Branch on Condition
08	DIV	Op1 Op1/ Op2
09	READ	Operand 2 input Value
10	PRINT	Output Operand2

MOVER and MOVEM

- **MOVEM**

- **MOVER**

Assembler: Types of Staetments

Assembly language consist of three kinds of statements

1. Imperative statements

Imperative assembly language statements indicates actions to be performed during the execution of the assembled program. Hence each imperative statement translates into machine instruction.

Assembler: Types of Statements

1. Imperative statements

Example:

MOVEM	AREG ,	B
MOVER	AREG ,	C
ADD	AREG ,	ONE
COMP	AREG ,	='1'
SUB	BREG ,	='2'

Assembler: Types of Statements

2. Declarative statements:

A declarative statement assembly language statement declares constants or storage areas in a program.

e.g.

i) A DS 1 ii) G DS 200

These statements simply declares a storage area of 1 word and block of 200 words respectively.

Constants are declared using *Declare Constant (DC) statement.*

e.g. ONE DC '1'

Assembler: Types of Statements

3. Assembler Directives:

Assembler Directives neither represent machine instructions instead of that they direct the assembler to take certain actions during the process of assembling a program.

e.g. START 100

This statement indicates that the first word of the object program generated by the assembler should be placed in the memory location with address '100'

END

Indicates end of assembly language program

The Process of Translation

The general model for the translation process can be represented as follows:

$$\left\{ \text{Analysis of Source Text} \right\} + \left\{ \text{Synthesis of Source Text} \right\} = \left\{ \text{Translation from Source to Target Text} \right\}$$

Analysis Phase:

In the analysis phase , we are concerned with the determination of meaning of a source language text, for that we should know the grammar of source language . Further we should also know how to determine the meaning of a statement once its grammatical structure becomes known. The rules of grammar and rules of meanings are known as *syntax* and *semantics* of language respectively.

General Design Procedure of Assembler

Synthesis Phase

Assembly Language Program → Analysis Phase → Synthesis Phase → Equivalent Target Code

In the synthesis phase , the source language statements are replaced by target language statements.

General Design Procedure of Assembler

1. Analysis Phase.

In this phase following functions are performed.

i. Isolate the label, mnemonic operation code and operand fields of a statement

ii. Enter the symbol found in label field (if any) and its address into *Symbol Table.*

iii. Validate the mnemonic operation code by looking it up in the Mnemonic Table.

iv. Determine the storage requirements of the statement by considering the mnemonic operation code and operand fields of the statement. Calculate the address of the first machine word following the target code generated for this statement (*Location Counter processing*).

General Design Procedure of Assembler

2. Synthesis Phase.

In this phase following functions are performed.

i. Obtain the machine operation code corresponding to the mnemonic operation code by searching the *Mnemonic table.*

ii. Obtain address of the operand from *Symbol table.*

iii. Synthesis the machine instruction or the machine form of the constant, as the case may be.

Some Assembler Directives

1. ORIGIN

This directive sets the location counter to the given address.

e.g. ORIGIN 200

sets the location counter(LC) value to 200.

2. EQU

The EQU statement simply defines a new symbol and gives it the value indicated by operand expressions.

e.g. FIRTST EQU LAST

Some Assembler Directives

3. LTORG

While assembling, a reference to literal, the following care should be taken:

i) Allocation of machine location to contain the value of the literal during execution and

ii) Use of the address of this location as the operand address in the statement referencing the literal.

At every LTORG , statement assembler allocates all literals used in the program .

Sample program to find X+Y

Program to Find ……… numbers:

```
        START   101
        READ    X
        READ    Y
        MOVER   AREG, X
        ADD     AREG, Y
        MOVEM   AREG, RESULT
        PRINT   RESULT
        STOP
X       DS      1
Y       DS      1
```

Sample program to find X+Y

- *Find X+Y*

Opcode Register Memory operand

			LC				
	START	101					
	READ	X	101	+	09	0	108
	READ	Y	102	+	09	0	109
	MOVER	AREG, X	103	+	04	1	108
	ADD AREG, Y		104	+	01	1	109
	MOVEM	AREG, RESULT	105	+	05	0	110
	PRINT	RESULT	106	+	10	0	110
	STOP		107	+	00	0	000
X	DS 1		108				
Y	DS 1		109				
RESULT	DS 1		110				
	END						

Required M/C Code

LC	Opcode	Register	Address
101	09	0	108
102	09	0	109
103	04	1	108
104	01	1	109
105	05	0	110
106	10	0	110
107	00	0	000
108			
109			
110			
111			

Variable	Address
X	108
Y	109
RESULT	110

Literals & Constants

int x=5;

x = x + 5; Constant

1. Literal cannot be changed during program execution
2. Literal is more safe and protected than a constant.
3.

```
            START     101
            READ      N
            MOVER     BREG,  ONE
            MOVEM     BREG,  TERM
AGAIN       MULT      BREG,  TERM
            MOVER     CREG,  TERM
            ADD    CREG,  ONE
            MOVEM     CREG,  TERM
            COMP      CREG,  N
            BC     LE, AGAIN
            MOVEM     BREG,  RESULT
            PRINT     RESULT
            STOP
N           DS     1
RESULT   DS      1
ONE      DC      '1'
TERM        DS     1
            END
```

```
101)  + 09 0 113
102)  + 04 2 115
103)  + 05 2 116
104)  + 03 2 116
105)  + 04 3 116
106)  + 01 3 115
107)  + 05 3 116
108)  + 06 3 113
109)  + 07 2 104
110)  + 05 2 114
111)  + 10 0 114
112)  + 00 0 000
113)
114)
115)
116)
```

```
                START       101
                READ        N
                MOVER       BREG,   ONE
                MOVEM       BREG,   TERM
AGAIN           MULT        BREG,   TERM
                MOVER       CREG,   TERM
                ADD     CREG,   ONE
                MOVEM       CREG,   TERM
                COMP        CREG,   N
                BC      LE, AGAIN
                DIV     BREG,   TWO
                MOVEM       BREG,   RESULT
                PRINT       RESULT
                STOP
N               DS      1
RESULT  DS          1
ONE     DC          '1'
TERM            DS      1
TWO             DC      '2'
                END
```

```
101)    + 09 0 114
102)    + 04 2 116
103)    + 05 2 117
104)    + 03 2 117
105)    + 04 3 117
106)    + 01 3 116
107)    + 05 3 117
108)    + 06 3 114
109)    + 07 2 104
110)    + 08 2 118
111)    + 05 0 115
112)    + 10 0 115
113)    + 00 0 000
114)
115)
116)
117)
118)
```

Sample program

Program to Find …….numbers:

```
                START   101
                READ    N
                MOVER   BREG, ONE
                MOVEM   BREG, TERM
AGAIN           MULT    BREG, TERM
                MOVER   CREG, TERM
                ADD     CREG, ONE
                MOVEM   CREG, TERM
                COMP    CREG, N
                BC      LE, AGAIN
                MOVEM   BREG, RESULT
```

Types of Assembler

1. Multi-pass Assembler
2. Single-pass Assembler

An Assembler pass is one complete scan of the source program input an assembler.

1. Single-pass Assembler:

- Single-pass assembler have the advantage that every source statement processed only once.

- Single pass assembler would face a problem while translating *forward references.*

- This problem can be solved as below:

Single-pass Assembler

- Instructions containing forward references can be left incomplete until address of the referenced symbol becomes known. These incomplete instructions are placed into a table called as *Table of Incomplete Instructions (TII).*

- The problem of forward reference can be handled using a technique called as back patching.(In encountering its definition, its address can be filled into that instruction.)

```
        START   100
        MOVER   AREG, X
L1      ADD     BREG, ONE
        ADD     CREG, TEN
        STOP
X       DC      ‘5’
ONE     DC      ‘1’
TEN     DC      ‘10’
        END
```

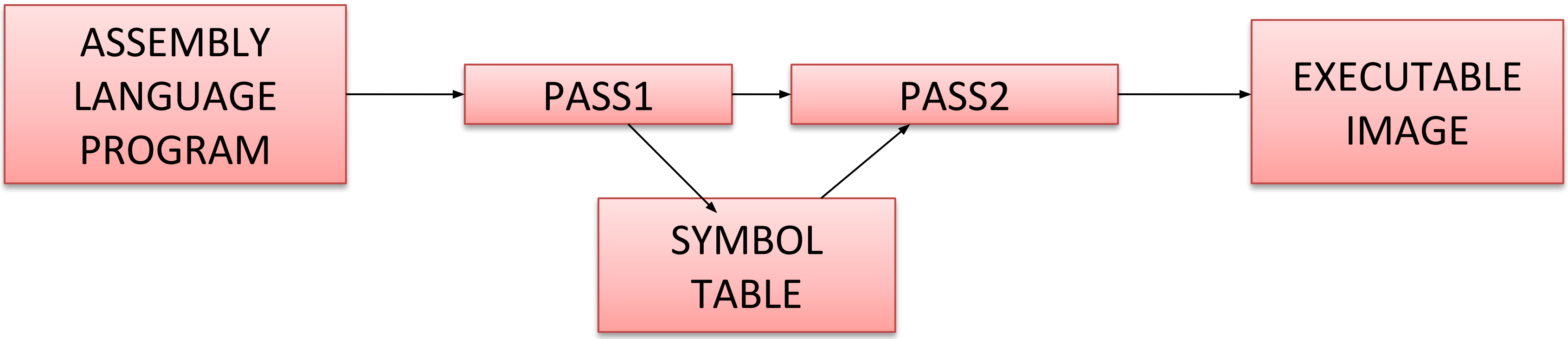
2. Multi-pass Assembler

- Multi pass assemble resolves the problem of forward reference by using more than one passes.
- In first pass analysis is takes place in which LC processing is performed and symbols defined in the program are entered into the symbol table.
- During the second pass, statements are processed or synthesized to generate machine instructions.

Two Pass Assembler

- Handles forward references easily.
- Requires 2 scans of the source program.
- LC processing is performed in the 1^{st} pass and symbols are stored in the symbol table.
- Second pass synthesis Target Program.

Two Pass Assembler



First Pass:

scan program file

find all labels and calculate the corresponding addresses;

this is called the *<u>symbol table</u>*

Second Pass:

convert instructions to machine language,

using information from symbol table

Pass structure of assembler

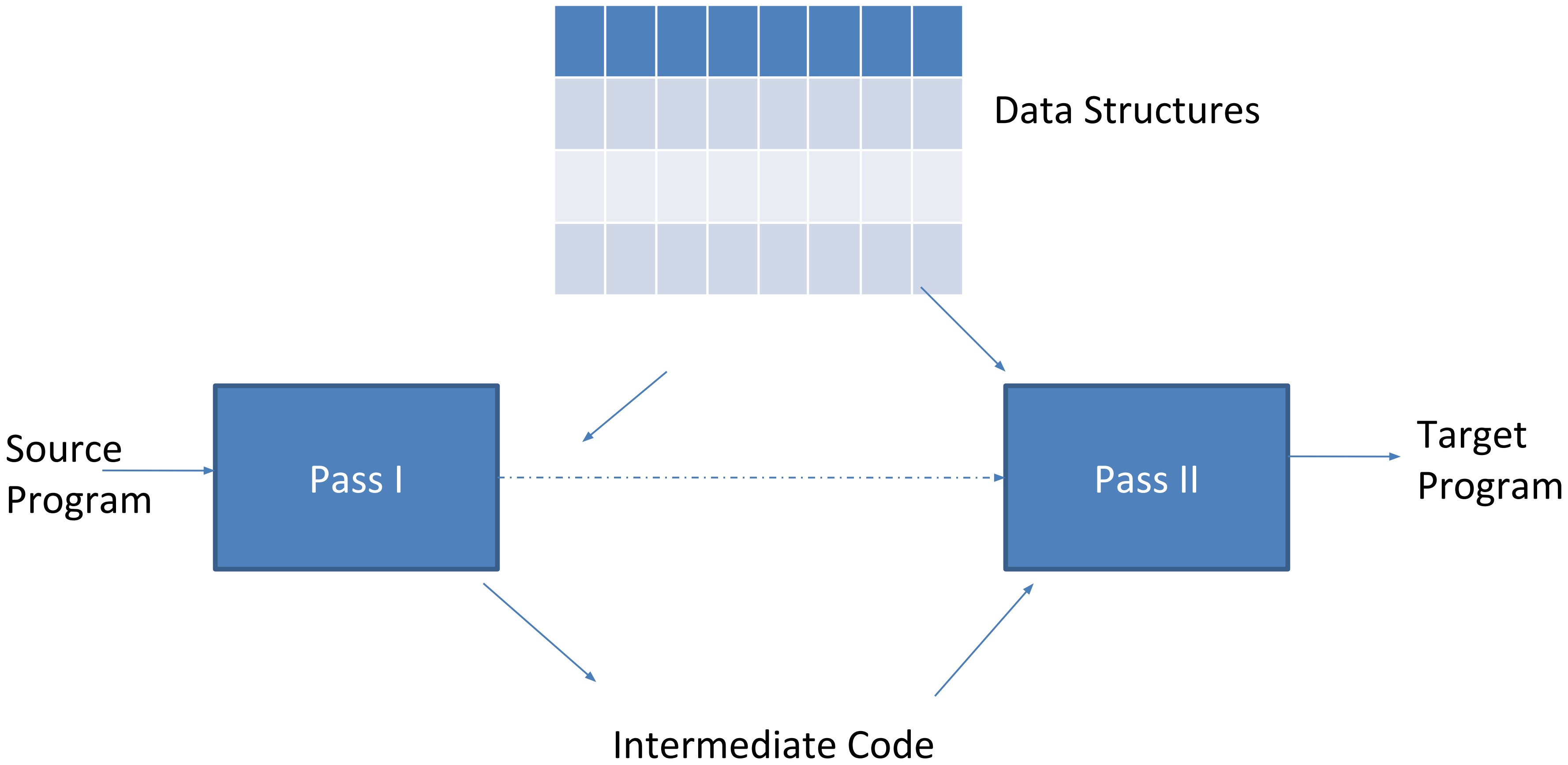


Figure: Overview of Two Pass Assembler

Data Structures Used in an Assembler

Following Data Structures are used in Pass-I of an Assembler:

1. OTAB (Opcode Table)
2. SYMTAB (Symbol Table)
3. LITTAB (Literal Table)
4. POOLTAB(Literals Pools)

Sample program ILLUSTRATING ORIGIN & LTORG DIRECTIVE

1 START 200

2 MOVER AREG, ='5'

3 MOVEM AREG, A

4 LOOP MOVER AREG, A

5 MOVER CREG, B

6 ADD CREG, ='1'

Sample program ILLUSTRATING ORIGIN & LTORG DIRECTIVE

1 START 200

2 MOVER AREG, ='5' 200) 04 1 211

3 MOVEM AREG, A 201) 05 1 217

4 LOOP MOVER AREG, A 202) 04 1 217

5 MOVER CREG, B

1. OPTAB (Opcode Table)

Mnemonic Opcode	Class	Machine Opcode/ Routine ID	Length
START	3 (Directive) AS	R#11	-
MOVER	1 (Imperative) IS	04	1
DS	2 (Declarative) DL	R#7	-
START	3 (Directive) AS	R#11	-
MOVEM	1 (Imperative) IS	05	1

2. SYMTAB (Symbol Table)

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

3. LITTAB (Literal Table)

Literal	Address
='5'	211
='1'	212
='1'	219

first	#literals
1	2
3	1
4	0

Next Free entry

Current Pool Pointer

Pass-I of an Assembler

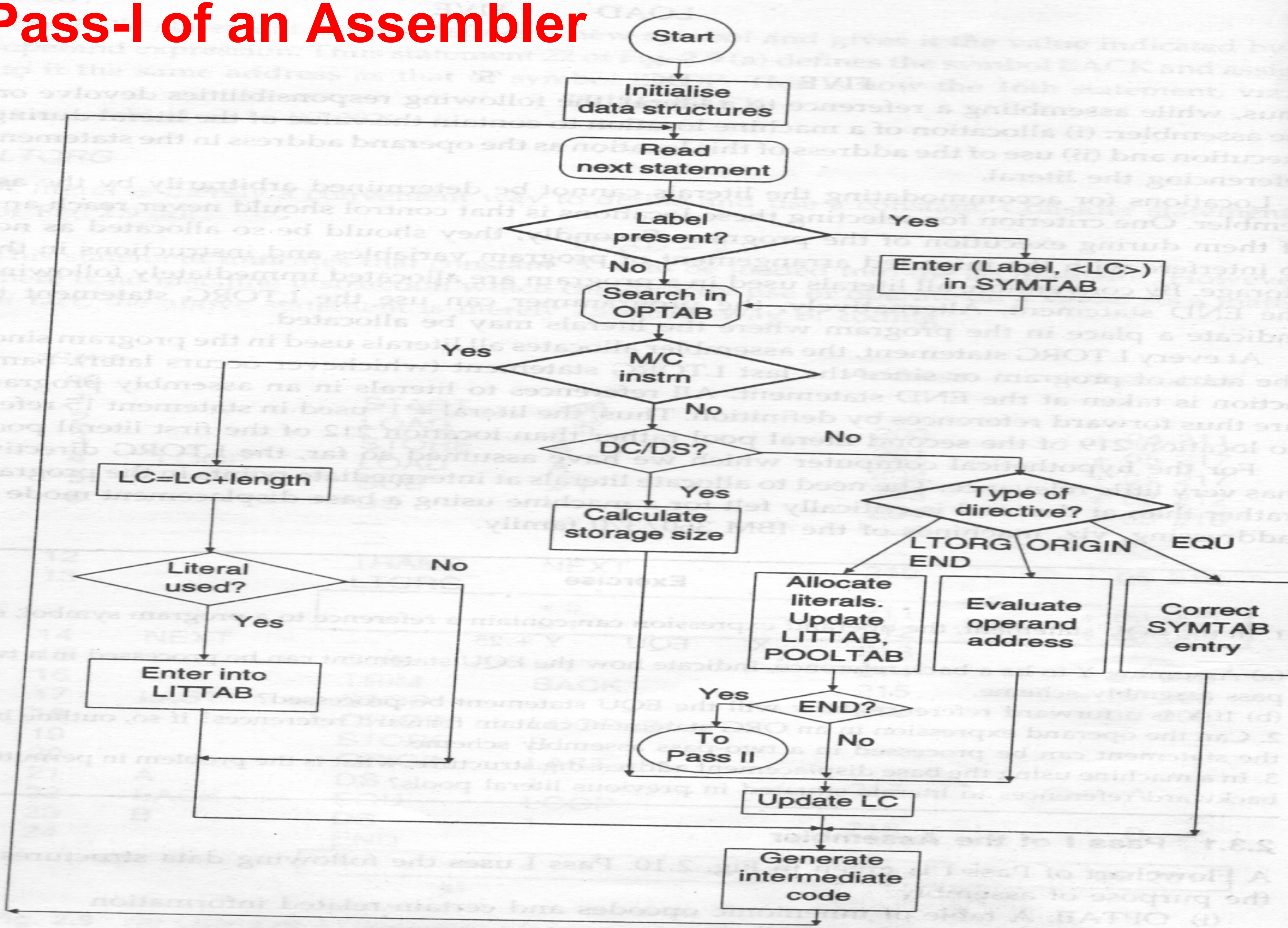


Fig. 2.10 *Pass I of the Assembler*

PASS-1 Algorithm

LC	:	Location counter
littab_ptr	:	Points to an entry in LITTAB
pooltab_ptr	:	Points to an entry in POOLTAB

1. LC := 0; (This is the default value)
   littab_ptr := 1;
   pooltab_ptr := 1;
   POOLTAB [1]. *first* := 1; POOLTAB [1]. *# literals* := 0;

2. While the next statement is not an END statement

   (a) If a symbol is present in the label field then
       this_label := symbol in the label field;
       Make an entry (*this_label*, <LC>, –) in SYMTAB.

(b) If an LTORG statement then

(i) If POOLTAB [*pooltab_ptr*]. # *literals* > 0 then

Process the entries LITTAB [POOLTAB [*pooltab_ptr*]. *first*] ... LITTAB [*littab_ptr* − 1] to allocate memory to the literal, put address of the allocated memory area in the *address* field of the LITTAB entry, and update the address contained in location counter accordingly.

(ii) *pooltab_ptr* := *pooltab_ptr* + 1;

(iii) POOLTAB [*pooltab_ptr*]. *first* := *littab_ptr*;
POOLTAB [*pooltab_ptr*]. # *literals* := 0;

(c) If a START or ORIGIN statement then
LC := value specified in operand field;

(d) If an EQU statement then

(i) *this_addr* := value of <*address specification*>;

(ii) Correct the SYMTAB entry for *this_label* to (*this_label*, *this_addr*, 1).

(e) If a declaration statement then

(i) Invoke the routine whose id is mentioned in the *mnemonic info* field. This routine returns *code* and *size*.

(ii) If a symbol is present in the label field, correct the symtab entry for *this_label* to (*this_label*, <LC>, *size*).

(iii) LC := LC + *size*;

(iv) Generate intermediate code for the declaration statement.

(f) If an imperative statement then

(i) *code* := machine opcode from the *mnemonic info* field of OPTAB;

(ii) LC := LC + instruction length from the *mnemonic info* field of OPTAB;

(iii) If operand is a literal then

this_literal := literal in operand field;

if POOLTAB [*pooltab_ptr*]. # *literals* = 0 or *this_literal* does not match any literal in the range LITTAB [POOLTAB [*pooltab_ptr*]] .*first* … LITTAB [*littab_ptr*−1] then

LITTAB [*littab_ptr*]. *value* := *this_literal*;

POOLTAB [*pooltab_ptr*]. # *literals* :=

POOLTAB [*pooltab_ptr*]. # *literals* +1;

littab_ptr := *littab_ptr* + 1;

else (i.e., operand is a symbol)

this_entry := SYMTAB entry number of operand;

Generate intermediate code for the imperative statement.

3. (Processing of the `END` statement)

(a) Perform actions (i)–(iii) of Step 2(b).

(b) Generate intermediate code for the `END` statement.

Pass 1 purpose: Define symbols and literals

- Determine length of machine instructions
- Keep track of location counter (LC)
- Remember values of symbols until pass 2
- Process some pseudo ops, e.g. EQU, DS
- Remember literals

Pass Structure of Assembler

- **Single Pass Translation Example**

```
            START     100
            MOVER     AREG, X
            ADD       BREG, ONE
            ADD       CREG, TEN
            STOP
X           DC    '5'
ONE              DC      '1'
TEN              DC      '10'
            END
```

Pass Structure of Assembler

- Single Pass Translation Example

```
      START  100
      MOVER  AREG, X
      ADD    BREG, ONE
      ADD    CREG, TEN
      STOP
X     DC  '5'
ONE       DC  '1'
TEN       DC  '10'
      END
```

```
100            04   1   _ _ _
101       01  2   _ _ _
102            06   3   _ _ _
103            00   0   000
104
105
106
```

Instruction Address	Symbol Making a forward reference
100	X
101	ONE
102	TEN

Figure : TII (Table of Incomplete Instruction)

Symbol	Address
X	104
ONE	105
TEN	105

Figure : Symbol Table

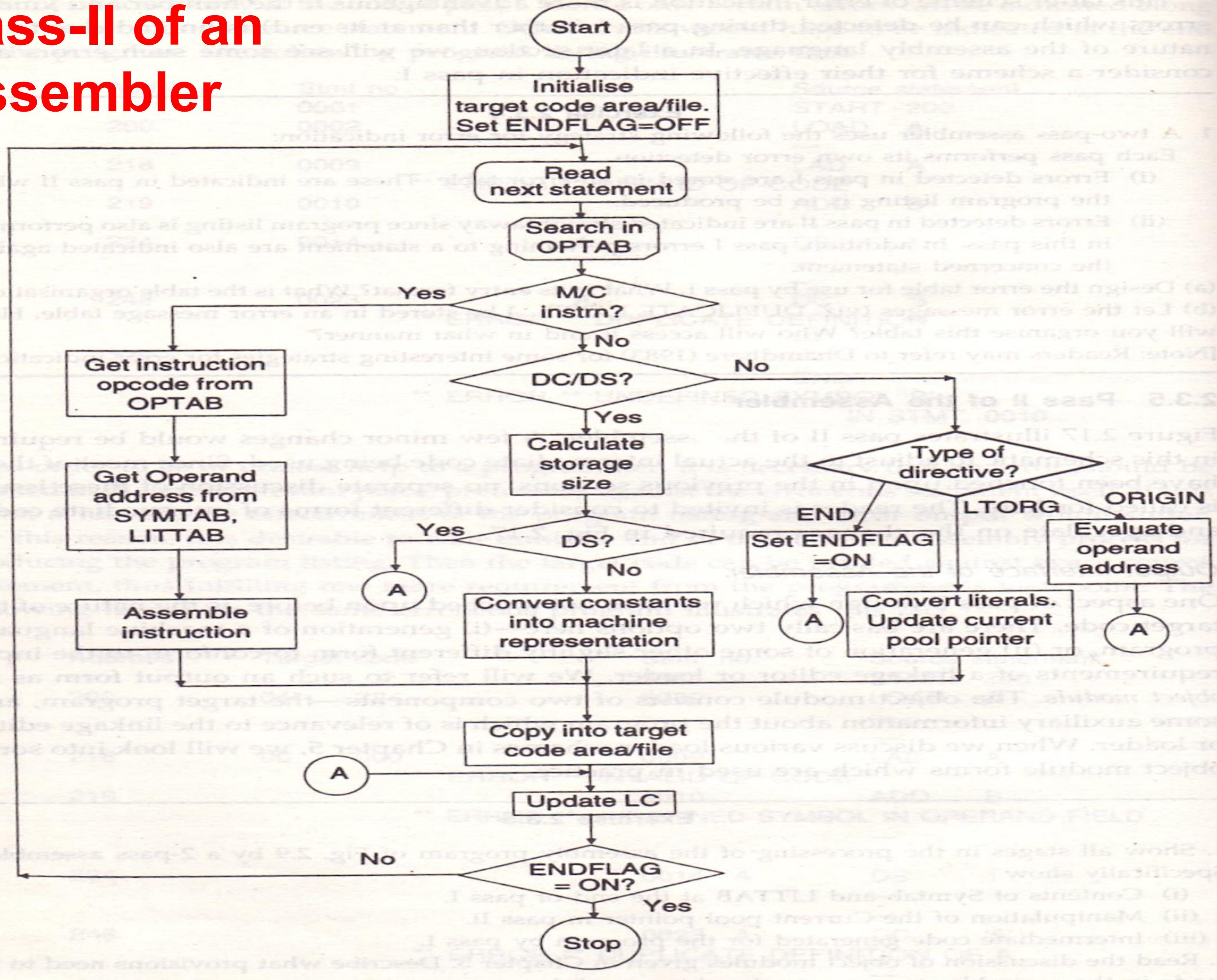
Pass Structure of Assembler

- Single Pass Translation Example

```
       START  100
       MOVER  AREG, X
       ADD    BREG, ONE
       ADD    CREG, TEN
       STOP
X      DC  '5'
ONE        DC  '1'
TEN        DC  '10'
       END
```

100	04	1	**104**
101	01	2	**105**
102	06	3	**106**
103	00	0	000
104			
105			
106			

Pass-II of an Assembler



PASS – 2 of assembler

SYMTAB, LITTAB and POOLTAB

LC	:	Location counter
littab_ptr	:	Points to an entry in LITTAB
pooltab_ptr	:	Points to an entry in POOLTAB
machine_code_buffer	:	Area for constructing code for one statement
code_area	:	Area for assembling the target program
code_area_address	:	Contains address of *code_area*

Algorithm 3.2 (Second pass of a two-pass assembler)

1. *code_area_address* := address of *code_area*;
   pooltab_ptr := 1;
   LC := 0;

2. While the next statement is not an END statement

   (a) Clear *machine_code_buffer*;

   (b) If an LTORG statement

      (i) If POOLTAB [*pooltab_ptr*]. *# literals* > 0 then

         Process literals in the entries LITTAB [POOLTAB [*pooltab_ptr*] .*first*] … LITTAB [POOLTAB [*pooltab_ptr*+1]−1] similar to processing of constants in a DC statement. It results in assembling the literals in *machine_code_buffer*.

      (ii) *size* := size of memory area required for literals;

      (iii) *pooltab_ptr* := *pooltab_ptr* + 1;

   (c) If a START or ORIGIN statement

      (i) LC := value specified in operand field;

   (d) If a declaration statement

      (i) If a DC statement then

         Assemble the constant in *machine_code_buffer*.

      (ii) *size* := size of the memory area required by the declaration statement;

   (e) If an imperative statement

      (i) Get address of the operand from its entry in SYMTAB or LITTAB, as the case may be.

      (ii) Assemble the instruction in *machine_code_buffer*.

      (iii) *size* := size of the instruction;

(f) If *size* $\neq 0$ then

(i) Move contents of *machine_code_buffer* to the memory word with the address *code_area_address* + <LC>;

(ii) LC := LC + *size*;

3. (Processing of the END statement)

   (a) Perform actions (i)–(iii) of Step 2(b).

   (b) Perform actions (i)–(ii) of Step 2(f).

   (c) Write *code_area* into the output file.

Pass 2 purpose: Generate object program

- Look up value of symbols
- Generate instructions
- Generate data (for DS, DC and literals)
- Process pseudo ops codes

Design of a two pass assembler

1. Separate label, opcode & operand

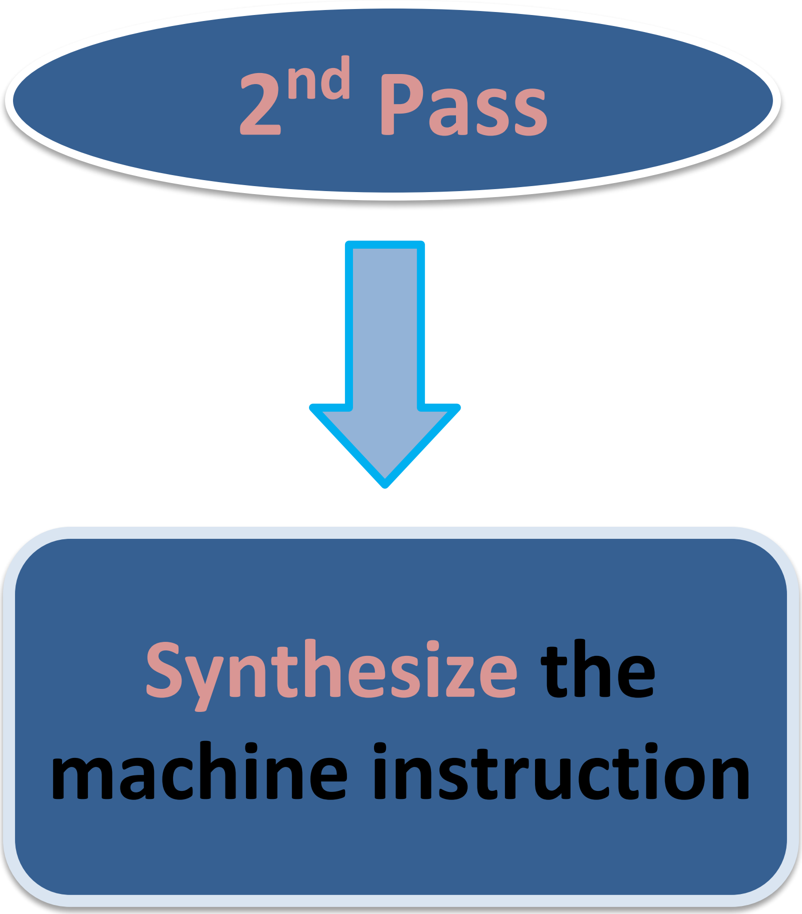
2. Build the symbol table

1st pass

3. Perform LC processing

4. Construct IC

2^{nd} Pass



Example

			LC
	START	200	
	MOVER	AREG, ='5'	200
	MOVEM	AREG, X	201
L1	MOVER	BREG, ='2'	202
	ORIGIN	L1+3	
	LTORG		205
			206
NEXT	ADD	AREG,='1'	207
	SUB	BREG,='2'	208
	BC	LT, BACK	209
	LTORG		210
			211
BACK	EQU	L1	212
	ORIGIN	NEXT+5	
	MULT	CREG,='4'	212
	STOP		213
X	DS	1	214
	END		

START 200

Symbol	Address

Literal	Address

First	#literal
1	0

MOVER AREG,='5' 200

Symbol	Address

Literal	Address
='5'	---

First	#literal
1	0

MOVEM AREG,X 201

Symbol	Address
X	----

Literal	Address
='5'	---

First	#literal
1	0

L1 MOVER BREG,='2' 202

Symbol	Address
X	----
L1	202

Literal	Address
='5'	---
='2'	---

First	#literal
1	0

ORIGIN L1+3 203

Symbol	Address
X	----
L1	202

Literal	Address
='5'	---
='2'	---

First	#literal
1	0

LTORG 205

206

Symbol	Address
X	----
L1	202

Literal	Address
='5'	205
='2'	206

First	#literal
1	2

NEXT ADD AREG, ='1' 207

Symbol	Address
X	----
L1	202
NEXT	207

Literal	Address
='5'	205
='2'	206
='1'	----

First	#literal
1	2
3	0

SUB BREG,='2' 208

Symbol	Address
X	----
L1	202
NEXT	207

Literal	Address
='5'	205
='2'	206
='1'	----
='2'	-----

First	#literal
1	2
3	0

BC LT, BACK 209

Symbol	Address
X	----
L1	202
NEXT	207
BACK	----

Literal	Address
='5'	205
='2'	206
='1'	----
='2'	----

	#literal
1	2
3	0

LTORG 210

211

Symbol	Address
X	----
L1	202
NEXT	207

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211

First	#literal
1	2
3	2
5	0

BACK EQU L1 212

Symbol	Address
X	----
L1	202
NEXT	207
BACK	202

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211

First	#literal
1	2
3	2
5	0

ORIGIN NEXT+5 213

Symbol	Address
X	----
L1	202
NEXT	207
BACK	202

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211

First	#literal
1	2
3	2
5	0

MULT CREG,='4' 212

Symbol	Address
X	----
L1	202
NEXT	207
BACK	202

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211
='4'	----

First	#literal
1	2
3	2
5	0

STOP 213

Symbol	Address
X	----
L1	202
NEXT	207
BACK	202

Literal	Address
='5'	205
='2'	206
='1'	210
='2'	211
='4'	----

First	#literal
1	2
3	2
5	0

X DS 1 214

Symbol	Address
X	214
L1	202
NEXT	207
BACK	202

Literal	Address
=‘5’	205
=‘2’	206
=‘1’	210
=‘2’	211
=‘4’	----

First	#literal
1	2
3	2
5	0

END

Symbol	Address
X	214
L1	202
NEXT	207
BACK	202

Literal	Address
=‘5’	205
=‘2’	206
=‘1’	210
=‘2’	211
=‘4’	215

First	#literal
1	2
3	2
5	1
6	0

General Design Procedure:

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

Intermediate code Form

1. Intermediate code for Declarative Statements
   - DC 01
   - DS 02
2. Intermediate code for Assembler directives
   - START 01
   - END 02
   - ORIGIN 03
   - EQU 04
   - LTORG 05

THE MNEMONICS OPCODE FIELD CONTAIN A PAIR IN THE FORM

(Statement Class, Code)

Intermediate code Form

3. Intermediate code for Imperative statements

Consist of two variant of IC code (Differs in information contained in the operand fields)

- Variant 1
- Variant 2

Intermediate code Form

3. Intermediate code for Imperative statements (Variant 1):

The first operand(CPU Register) is represented by single digit number. E.g. 1 represents AREG, 2 represent BREG e.t.c.

The second operand is a memory operand is represented by pair of form as follows:

(OPERAND CLASS, CODE)

OPERAND CLASS:

Class	Meaning
C	Constant
S	Symbol
L	Literals

Syntax for BC

BC	Condition code	Memory Address

LT, LE, EQ, GT, GE, ANY

What if we want to have an unconditional jump?

Intermediate code Form

3. Intermediate code for Imperative statements (Variant 1):

First operand

- Represented by single digit number.

Register	No
AREG	1
BREG	2
CREG	3
DREG	4

BC	No.
LT	1
LE	2
EQ	3
GT	4
GE	5
ANY	6

Second operand

- memory operand is represented by pair of

(OPERAND CLASS, CODE)

Operand class	
C	Constant
S	Symbol
L	Literal

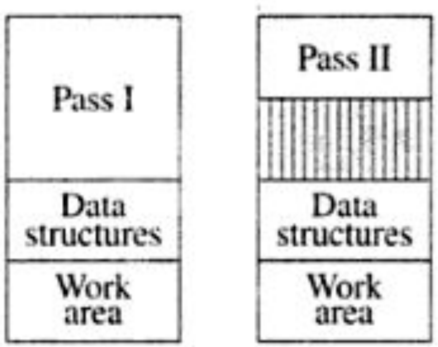
Intermediate code Form

Intermediate code for Imperative statements (Variant 1):

```
            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)
A           DS        1           (DL,02)  (C,1)
            LTORG                 (AD,05)
            ……………
```

Variants 1 of IC

```
        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)
A       DS      1                   (DL, 02)   (C,1)
        LTORG                       (DL,05)
        ...                         ...
```



(a)

Intermediate code Form

Intermediate code for Imperative statements (Variant 2):

Operand field of intermediate code may be same as variant 1 or In source form itself.

Declarative statement & Assembler directives has to process to support LC processing.

Literals are referenced & their entries are made in LITTAB.(L,m)

Intermediate code Form

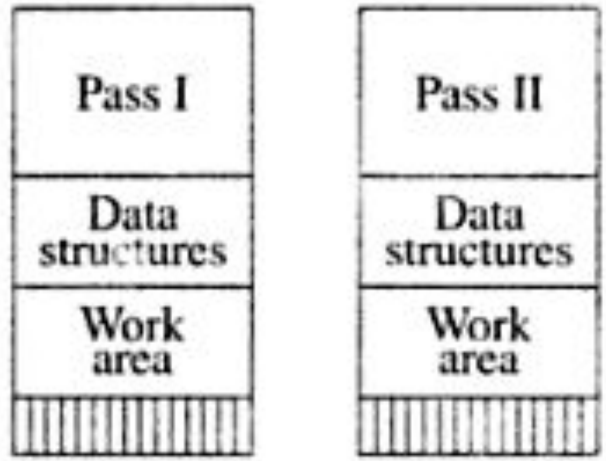
Intermediate code for Imperative statements (Variant 2):

	START	200	(AD,01)	(C,200)
	READ	A	(IS,09)	A
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		(DL,05)	
	……………			

Variant - 2 of IC

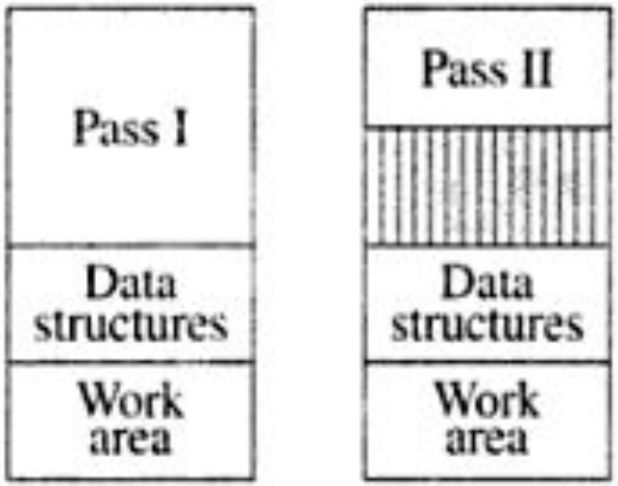
```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   A
LOOP    MOVER   AREG, A         (IS,04)   AREG, A
          :                        :
        SUB     AREG, ='1'      (IS,02)   AREG, (L,01)
        BC      GT, LOOP        (IS,07)   GT, LOOP
        STOP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

Variant-2 of IC

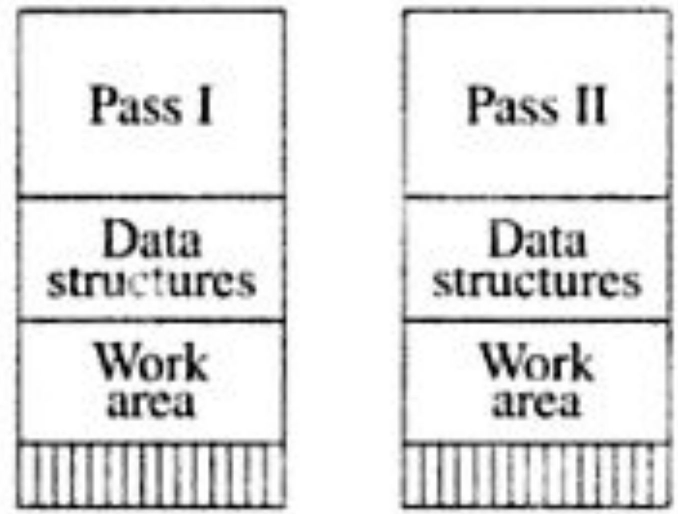


(b)

Variants of IC



(a)



(b)

- ✔Extra work in Pass I
- ✔Simplified Pass II
- ✔Pass I code occupies more memory than code of Pass II
- ✔Does not simplify the task of Pass II or save much memory in some situation.

- ✔IC is less compact
- ✔Memory required for two passes would be better balanced
- ✔So better memory utilization

Error Reporting

Sr. No.		Statement		Address
001		START	200	
002		MOVER	AREG, A	200
003		⋮		
009		MVER	BREG, A	207
	** error ** Invalid opcode			
010		ADD	BREG, B	208
014	A	DS	1	209
015		⋮		
021	A	DC	'5'	227
	** error ** Duplicate definition of symbol A			
022		⋮		
035		END		
	** error ** Undefined symbol B in statement 10			

Machine Dependent and Machine Independent features of Assembler

- M/C Dependent Features
  - **A] Instruction format & addr. mode:-**
  - **B] Program Relocation**
- Machine Independent Assembler Features
  - **1) Literals**
  - **2) Symbol defining statements**
  - **3) Expressions**

2. Literal

????
?

- **Operand** with syntax = ‘ value ’

What is the Difference between literal and constant..??

ADD AREG, =‘5’

The location of a literal cant be specified. So its value cant be changed like constant…

2. Literal

????
?

What is the Difference between literal and immediate operand..??

(Literal)
ADD AREG, ='5'

(Imm. operand)
ADD AREG,5

No Architectural provision is needed for literal like immediate operand..


Thank you ...


Thank you!