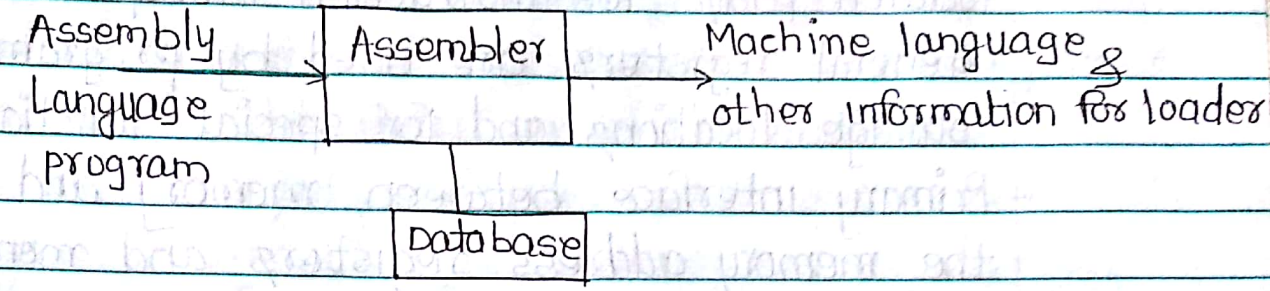
Assemblers

[System Programming & operating system by Dhamdhere]



General Design Procedure:-

steps to design any software is as follow.

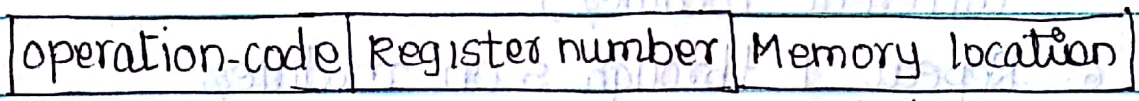
1. Specify the problem.
2. Specify data structure.
3. Define format of data structure.
4. Specify algorithm.
5. Look for modularity (Divide program functionality into different modules)
6. Repeat 1 through 5 on modules.

Assembly Language Basics:- (IBM system 360 and 370)

General machine structure:-

- The CPU consist of an instruction interpreter, a locati counter (LC), an instruction register & various working & general registers.
- Instruction interpreter:- feteches the instruction from mer
- Location counter (LC) - denotes the location of the curr instruction being executed.
- Instruction Register:- holds the copy of the current instruction.

- Working registers are memory devices that serves as scratch pads' for instruction interpreter.
- General registers are used by programmers as storage locations and for special functions.
- Primary interface between memory and CPU is via the memory address registers and memory buffer registers.
- Memory address register (MAR) contains the address of the memory location that is to be read from or stored into.
- Memory Buffer Register (MBR) contain a copy of the designated memory location specified by the MAR after a read or after the write.
- The basic instruction format is as follow.



Memory :-

- The basic unit of memory in the 360 is a byte - eight bits of information.
- That is, each addressable position in memory can contain eight bits of information.

Registers :-

- The 360 has 16 general-purpose registers consisting of 32 bits each.

- In addition there are 4 floating point registers consisting of 64 bits each.

When General purpose registers used as base registers, they aid in the formation of the address.

eg. Index register

A 1,901(2,15)

ADD offset Base register.

Types of instruction:-

1] Arithmetic
2] Logical
3] Control
4] Special interrupt.

Types of operands:-

1] Register operands:- operands are stored in general registers & provide faster data acc

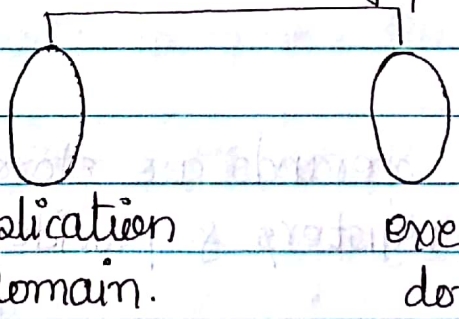
2] storage operands:- data stored in core memory & the length depends on the data types.

3] Immediate operands:- are single byte of data stored as a part of instruction.

D.M. Dhamdhere.

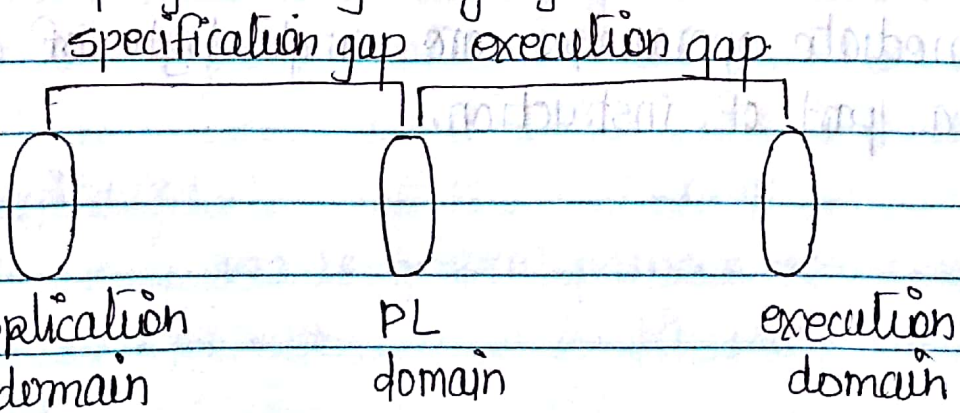
Why Language Processing activities arise?
due to the difference between the manner in which a software designer describes the ideas about the software behaviour and the manner in which these ideas are implemented in a computer system.

- The designer express the ideas in terms related to the application domain.
- To implement these ideas, their description has to be interpreted in terms related to the execution domain.



- The issue of semantic gap has been tackled by S/W engineering through the use of methodology and programming languages (PLs).

specification
design
coding



Language Processor :- is a software that convert source program into target lange - and finally execute it.

source and Target languages :- The languages in which programs are written are called source language & target language respectively.

forward reference :-

A forward reference of a program entity is a reference to the entity which get used or define first & then it is declared.

ex :-

```
add = a + 10;   → forward reference
---
---
int a;
```

Language processing pass :- represent the processing of every statement in a source program by language processor.

Assembly Lanaguage Basic conti....

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

- Three basic features

1. Mnemonic operation codes :- (Mnemonic opcodes)
   - This codes represent the different operations in assembly lanaguage.
   - Use of mnemonic opcode for machine instruction eliminates the need to memorize numeric operation codes.

2. Symbol operands:
   - Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements.
   - The assembler performs memory bindings to these names, the programmer need not know any details of the memory bindings performed by the assembler.

3. Data declarations :-
   - Data can be declared in a variety of notations, including the decimal notation.

Statement Format :-

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

Where,

[..] - 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(s) generated for the statement.

<operand spec> has the following syntax :-

<Symbolic Name> [+<displacement>] [(<index register>)]

Ex 1] AREA :- refer to the memory word with which the name AREA is associated.

2] AREA+5 - refer to the memory word 5 words away from the word with the name AREA. here 5 is displacement or offset from AREA.

3] AREA(4) = implies indexing with index register 4 - that is, the operand address is obtained by adding the contents of index register 4 to the address of AREA.

4] AREA+5(4)

In this language, each statement has two operands, the first operand is always a register which can be any one of

	M/C code
AREG	1
BREG	2
CREG	3
DREG	4

The second operand refers to a memory word using a symbolic name & an optional displacement.

Instruction code	Assembly Mnemonic	Remarks
00	STOP	stop execution
01	ADD	First operand is modified 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	
09	READ	First operand is not used
10	PRINT	

BC instruction is as follow.

BC <condition code spec>, <memory address>

LT, LE, EQ, GT, GE & ANY.

if matches go to

The opcode, register operand & memory operand occupy 2, 1 and 3 digits, respectively.

	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				

Types of Assembly Language statement:-

1] Imperative statements:- indicates an action to be performed during the execution of the assembled program.

ex ADD AREG, TWO.

2] Declarative statement:-

The syntax of declaration statement is as follow.

[label] DS <constant>

[label] DC '<value>'

DS - Declare storage statement reserve areas of memory & associate names with them.

ex.

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

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

The second statement reserves a block of 200 memory words. The name G is associated with the first word of the block.

The DC (declare constant) statement constructs memory words containing constant.

ex.

```
ONE    DC    '1'
```

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

Use of Constants : can be declared in two ways

1] immediate operands :-

ex. ADD AREG, FIVE

- - -

FIVE DC '5'

2] ADD AREG ='5'

A literal is an operand with the syntax ='<value>'
It differs from a constant because its location cannot be specified in the assembly program. This helps to ensure that its value is not changed during execution of a program.

3] Assembler Directives :-

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

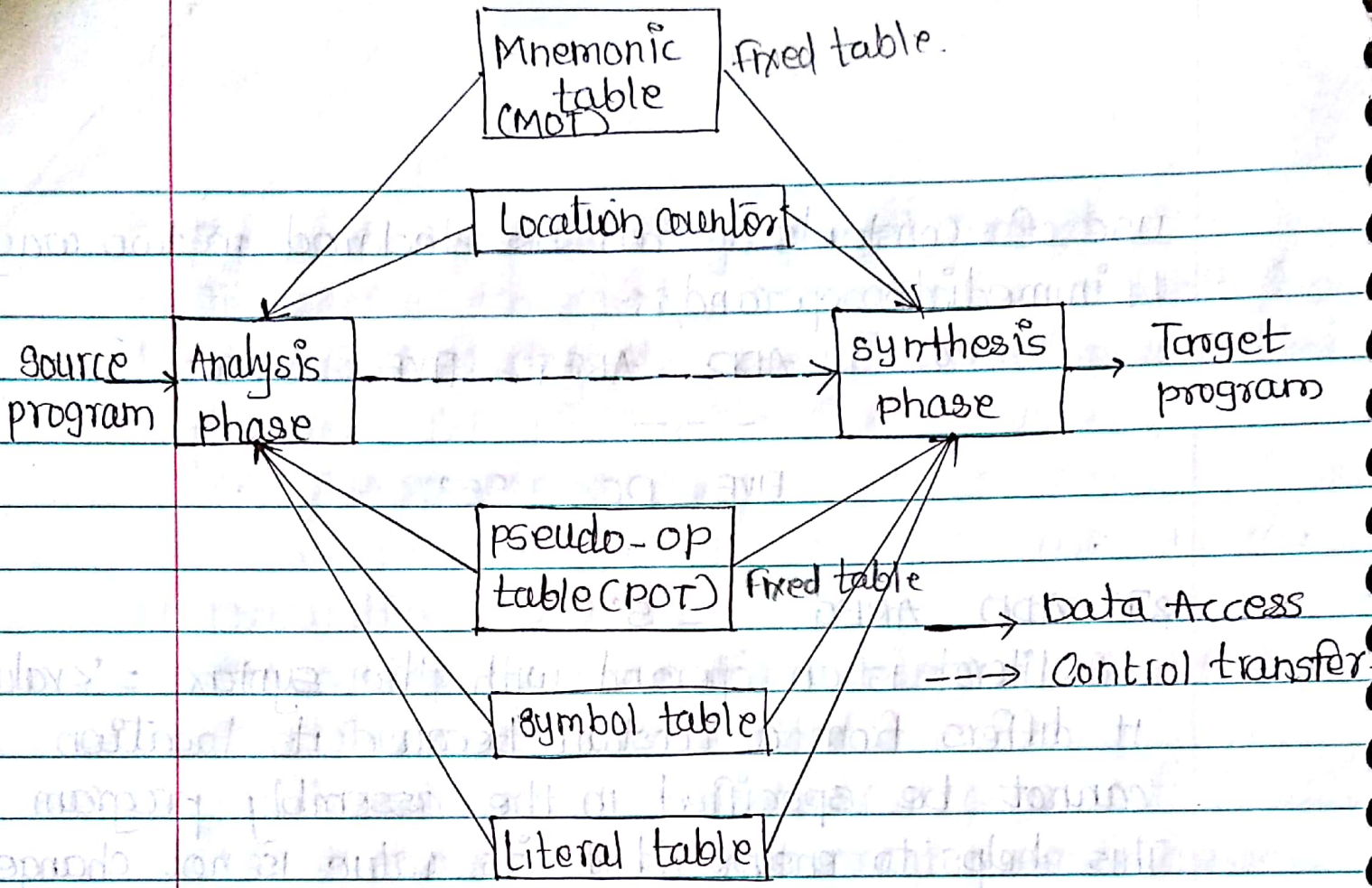
ex

START <constant>

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

END -

This directives indicates the end of the source program.



Analysis phase:-

The primary function performed by the analysis phase is the building of the symbol table.

steps 1] Isolate the label, mnemonic opcode & operand fields of a statement.

2] If a label is present, enter the pair (symbol, <LC contents>) in a new entry of symbol table.

3] Check validity of the mnemonic opcode through a look-up in the Mnemonics table.

4] Perform LC processing i.e. update the value contained in LC by considering the opcode & operands of the statement.

Synthesis phase :- convert the assembly statem into machine instruction with the help of data structures constructed in analysis phase.

steps 1] obtain the machine opcode corresponding to the mnemonic from the mnemonics table.
2] obtain address of a memory operand from the symbol table.
3] Synthesize a machine instruction

Data structure format:

mnemonic opcode	class	Binary Machine op code/Routine	length

Mnemonic opcode Table.

2] Symbol table

Symbol	address	length

Literal table

	literal	address
1	='5'	
2	='1'	
3	='1'	

3] A table, the pseudo-operation table (POT), that indica the symbolic mnemonic & action to be taken for each pseudo-op in pass 1.

literal no
#1
#3
-

POOHtab.

Pass structure of Assembler :-

1] Two-Pass translation -
- It can handle forward reference easily.
- Location counter (LC) processing is done in pass 1 & symbols defined in the program are entered into symbol table.
- The second pass uses this address info. to generate target program for the loader.
- In this scheme, pass-1 constructs an intermediate representation of source program used by pass2 This is called as intermediate code (IC)

2] Single pass translation
- Here problem of forward references is resolved using the technique of back patching where the operand that is forward referenced is left blank initially and then is later filled.
- These blank fields are kept in a special table called as table of incomplete instruction (TII)
- Each entry in TII is of the form
  (<Instruction address>, <symbol>)
- ex MOVER BREG, ONE.
- This statement can be partially synthesized since ONE is a forward reference. Hence entry in TII will be of the form (1001, ONE)

- By the time we process END statement, TII will contain information about all forward references.
- Assembler can process this entry by checking out the symbol table & finding the addresses of each forward reference.

Tasks performed by the 2-pass Assembler:-

Pass I -

1] Seperate symbol, mnemonic & operand fields.

2] Build the symbol table.

3] Perform LC processing.

4] Construct IC: (intermediate code)

Pass-II

1] Synthesize the target program

Advanced Assembler Directives:-

1] origin → Syntax - ORIGIN <address>
- This directive indicates that LC should be set to the address given by the address specification in origin.

- This statement is useful when target program does not consist of consecutive memory words.

- The ability to use operand in the origin statement helps to perform LC processing in a relative rather than absolute manner
eg. origin Loop+2.
sets LC to the value of address of label Loop+2.

- This facility is absent in START statement.

2] EQU - EQU defines the symbol to represent the address.
syntax -
<symbol> EQU <address>

3] LTORG - (memory allocation)
- By default, assembler places literal after END statement

- At every LTORG statement, as well as END statement, assembler allocates memory to the literals of a literal pool.

- LTORG statement however permits a programmer to specify where literals should be placed.

- If this statement is not used, every literal will be placed at the end of program.

4] DC statement :-

- For DC/DS, statement required storage has to be allocated either in byte, word or full word hence LC is incremented accordingly.

Intermediate Code :-

In IC each mnemonic representation contains a pair of the form.

statement	class	code.
	IS	instruction code
	DS	DC - 01
		DS - 02
	AD	START - 01
		END - 02
		ORIGIN 03
		EQU 04
		LTORG 05

IC for operands :-

The first operand is represented by a single digit number, which can represent a register or a condition code.

ex.

1 - 4 for AREG to DREG registers &
1 - 6 for LT - ANY condition codes.

- The second operand i.e. storage operand is represented as.

Operand class	code
C (Constant)	direct constant value.
S (Symbol) } L (Literal) }	entry number of the storage operand in ST. or Literal table.

Listing & error reporting :-

Listing means printing source program along with error reports if any.

While designing error reporting scheme it is necessary to concentrate on.

1] effectiveness of error reporting
2] speed
3] Memory requirement.

There are two phases, ~~tohere~~ after which errors can be reported.

1] After phase I :-

Advantages of error reporting & listing after pass - I.

a) Source program need not be preserved till pass II
b) conserves memory.
c) Avoids some amount of duplicate processing.

Disadvantages :-

1] A listing produced in pass I can report only certain errors in the most relevant place, that is against the source statement itself.

ex of such errors are syntax errors like missing commas or parenthesis and semantic errors like duplicate definitions of symbols.

Other errors like references to undefined variables can only be reported at the end of the source program

2] It is difficult to locate the target code corresponding to a source statement. All these factors make debugging difficult.

ex.

Sr No.	Statement		Address.	
001	START	200		
002	MOVER AREG, A		200	target code is not generated.
003	:			
009	MOVR 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

2] After pass-II :- (effective error reporting)

Advantages :- 1] It reports all errors against the erroneous statement itself.

2] Possible to print error reports as well as the target code against each source statement.

Disadvantages :-

1] Extra memory requirement to preserved source program for pass-II.

ex

Sr No	Statement	Address	Instruction
001	START 200		
002	MOVER AREG, A	200	04 1 209
003	⋮		
009	MOVR BREG, A	207	-/- 2 209
	**error ** Invalid opcode.		
010	ADD BREG, B	208	01 2 ---
	**error **Undefined symbol B in operand field.		
014	A DS 1	209	
015	⋮		
021	A DC '5'	227	00 0 005
	**error ** Duplicate definition of symbol A.		
022	⋮		
035	END.		

For effective error reporting, it is necessary to report all errors against the erroneous statement itself.

Ex

```
        START 200
        MOVER AREG, FIRST
        ADD   AREG, SECOND
        MOVEM AREG, RESULT
        PRINT RESULT
RESULT  DS    1
FIRST   DC    '5'
SECOND  DC    '7'
        END
```

Q) Using Two Pass Assembler. content of symbol table.

Symbol	Address/updated
FIRST	~~200~~ 205
SECOND	~~201~~ 206
RESULT	~~202~~ 204

content of MOT, POT is fixed.
There is no literal in source program.
Intermediate code after pass-I

Address	IC	
~~200~~	(AD, 01)	(C, 200)
200	(IS, 04)	1 (S, 01)
201	(IS, 01)	1 (S, 02)

202	(IS, 05)	1 (S, 03)
203	(IS, 10)	0 (S, 04)
204	(DL, 02)	(C, 1)
205	(DL, 01)	(C, 5)
206	(DL, 01)	(C, 7)
207	(AD, 02)	

Machine code after pass-II

–

200	04	1	205
201	01	1	206
202	05	1	204
203	10	0	204
204			
205	00	0	005
206	00	0	007

Using single pass Assembler:-
ST, MOT, POT, Literal table contents are same as in 2Pass assembler.

Sr No	Table of Incomplete instruction.
01	<200, FIRST>
02	<201, SECOND>
03	<202, RESULT>
04	<203, RESULT>

200	04	1	205
201	01	1	206
202	05	1	204
203	10	0	204
204			
205	00	0	005
206	00	0	007

Algorithms

1] ~~Assembler First Pass~~

1] loc_cntr = 0 (default value)
pooltab_ptr = 1;
POOLTAB[1] = 1;
literal_ptr = 1.

2] While next statement is not an END statement
(a) IF label is present then
this_label = Symbol in label field
Enter (this_label, loc_cntr) in Symbol table(ST)

(b) If an LTORG statement then
~~put literals into literal Table, update the values of literal_ptr accordeng & also update value of pooltab_ptr~~

process literals i.e allocate memory & put the address in the address field. Update Loc_cntr accordingly.

(c) If a START or ORIGIN statement then.
loc_cntr = value specified in operand field.
generate IC as (AD, code) (C, value)

(d) If an EQU statement then.
i) this_addr = value of <address> in operand field
ii) Correct the symbol entry for this_label to (this_label, this_addr)

(e) If a declaration statement.
i) code = code of declaration statement
ex DC = 1 & DS = 2
ii) size = size of memory required by DC/DS
iii) loc_cntr = loc_cntr + size
iv) Generate IC '(DL, code) ....'

(f) If an imperative statement then.
i) code = machine opcode from MOT.
ii) loc_cntr = loc_cntr + instruction length from MOT
iii) If operand is a literal
this_literal = literal in operand field
LITTAB[littab_ptr] = this_literal.
littab_ptr = littab_ptr + 1;
else (i.e operand is symbol)
this_entry = symbol table entry number of operand
iv) Generate IC as '(IS, code) (S, this_entry)
'(IS, code) OR '(L, this_literal_entryno)

3] (processing of END statement)
ⓐ Perform step 2(b)
ⓑ Generate IC as (AD, 02)
ⓒ Go to pass II.

2] Assembler second pass :-
It has been assumed that the target code is to be assembled in the area named code_area

Steps
1] Code_area_address = Address of code_area
pooltab_ptr = 1
loc_cntr = 0;

2] While next statement is not an END statement.
ⓐ clear machine_code_buffer.

ⓑ IF an LTORG statement
i) process literals
ii) Assemble the literals in machine_code_address
iii) size = size of memory area required for literals
iv) pooltab_ptr = pooltab_ptr + 1

ⓒ IF a START or ORIGIN statement then
i) loc_cntr = value specified in operand field
ii) size = 0

(d) If a declaration statement
   i) If a DC statement then
      Assemble the constant in machine_code_buffer
   ii) size = size of memory area required by DC/DS.

(e) If an imperative statement
   i) Get operand address from symbol table or Literal table
   ii) Assemble instruction in machine_code_buffer
   iii) size = size of instruction.

(f) if size ≠ 0 then
   i) Move contents of machine_code_buffer to the address code_area_address + loc_cntr.
   ii) loc_cntr = loc_cntr + size.

3] (processing of END statement)
   (a) Perform steps 2(b) & 2(f)
   (b) Write code_area into output file.

relative addressing
offset (index register, Base register)

ex — Name of Program.

```
JOHN  START  0
      using *, 15   — relative addressing is used.
      L      1, FIVE
      A      1, FOUR
      ST     1, TEMP.
FOUR  DC     F'4'
FIVE  DC     F'5'
TEMP  DS     1F
      END
```

using — This tells the assembler that register 15 is the base register & at execution time will contain the address of the first instruction of the program.

For solving above example, assume every instruction require 4 word of memory

Pass I

Relative address	IC	
—	(AD, 01)	(C, 0)
0	(IS, 04)	1, —(0,15)
4	(IS, 01)	1, —(0,15)
8	(IS, 05)	1, —(0,15)

12 (DL,01) (C,4)
16 (DL,01) (C,5)
20 (DL,02) (C,1)
24 (AD,02)

symbol	address
FIVE	16
FOUR	12
TEMP	20

~~Pass-II~~

Relative address

0	04	1 , 16(0,15)
4	01	1 , 12(0,15)
8	05	1 , 20(0,15)
12	00	0 004
16	00	0 005
20		