Unit 2

2.1 Basics of assembly language: Assembly language statements, statement format, simple set of instruction

2.2Types of assembly language statements

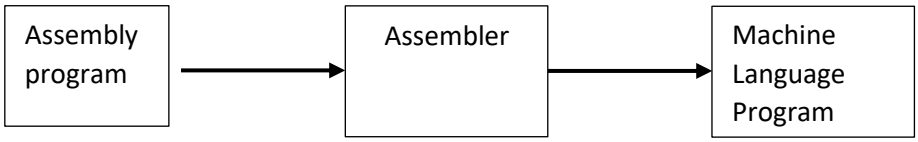
2.3 Assembler

2.4 Design of Assembler

- Problem Statement
- Databases Required for designing of assembler
- Pass Structure of assembler
- Pass I of assembler: working, algorithm
- Pass II of assembler: working, algorithm

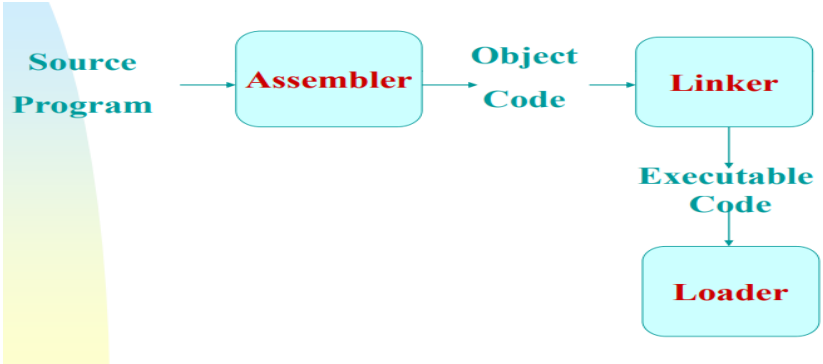
- **Assembler:**

- It is a Language processor or system software which translate program written in assembly language into machine language.
- A program that accepts an assembly language program as input and A program that accepts an assembly language program as input and produces its machine language equivalent along with information for the loader



(eg. Mov id3,r1 Error message(if any) (0001 1010 1100

Add r2,r1) 1101 0011 1010)



- **Assembly Language:**

- Assembly language is a kind of low level programming language which uses symbolic codes or Mnemonics as instruction.
- Assembly language program is converted into executable machine code by a utility program referred to as an assembler like Nasam, Masam etc
- Designed for a specific family of processors that represents various instruction in symbolic code
- An assembly program can be divided into 3 section
    1. **The data section-**used for declaring initialized data or constants
    2. **The bss section-** used for declaring variables
    3. **The text section**- used for keeping the actual code.

- **Features of Assembly Language:**
- Machine dependent
- Low level programming language
- Each element in an assembly program is
  -an instruction
  -declarative statement or directive
  -provide mnemonic operation code or instruction
    - ✓ Symbolic names for data or instruction
    - ✓ Facility for data declaration
    - ✓ Assembler directive to provide useful auxiliary facilities

- **Application of Assembly Language**
- It is used for direct hardware manipulation access to specialized processor instructions or to address critical performance issues.
- Typical use are device drivers(cd, HDD) low level embedded system(keybord, water tank indicator) & real time systems(computer, notepad)

- **Element of Assembly Language(Feature):**

1. **Mnemonic Operation Code or Mnemonic opcode**:

- Instead of using numeric opcodes(0 & 1), mnemonics are used (ADD, SUB, MOV, JMP)
- It is used for machine instruction eliminates the need to remember numeric operation codes. It also enables the assembler to provide helpful diagnostics for example indication of misspelt operation codes.
- Example: ADDSUB,MOVE,LDA(load accumulator),STA(store accumulator) etc

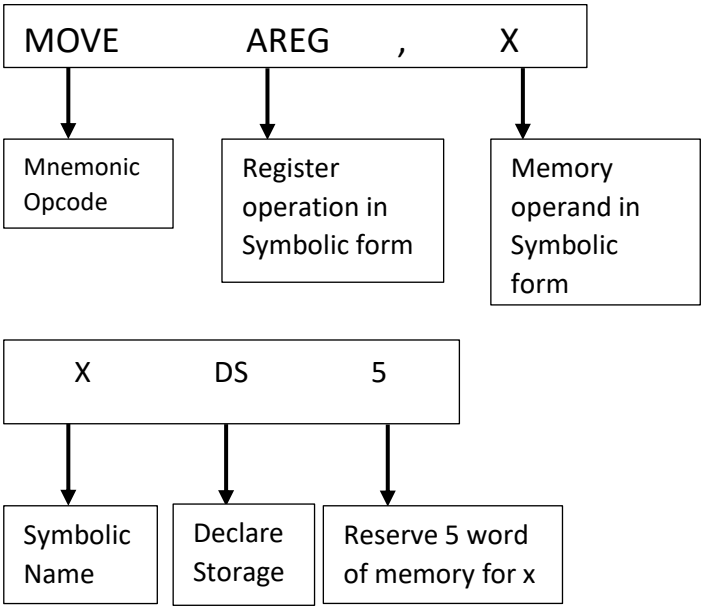
2. **Symbolic Operand:**

- Symbolic names can be associated with data or instruction.

- These Symbolic Name can be used as Operands in Assembly Statements.
- The Assembler performs memory binding to these names; the programmer need to know any details of the memory binding performed by the assembler.
- Example:- ADD R1,R2,R3

3. **Data declarations:**

- Data can be declared in variety of notations including the decimal notation.
  - Example NUM1 03 OR NUM1 0011H



- **Assembly Language Statement Format:**
  [Label]<opcode><operand specification>[,<operand specification..>]

  Or

  [Label]<opcode><operand1><operand2>

  Where the notation[..] indicates that the enclosed specification is optional.

  Example:

  ```
  ADD   AREG,SUM
  Max   DS, 1
  ```

  - **Label:**
    - Marks the address of an instruction,
    - Must have a colon :
    - Used to transfer program execution to a labeled instruction

  - **Mnemonic Opcode**

- Identifies the operation

Instruction Opcode	Assembly Mnemonics	Remarks
00	STOP	Stop execution
01	ADD	Perform addition
02	SUB	Perform Subtraction
03	MULT	Perform Multiplication
04	MOVER	Memory to register move
05	MOVEM	Register to memory move
06	COMP	Compare and set condition code
07	BC	Branch on Condition
08	DIV	Perform Division
09	READ	Read into Register
10	PRINT	Print Content of register

- **Operand**
  - Specify the data required by the operation
  - Executable instruction can have Zero to Three operands
  - Operands can be register, memory variable or constant

- **<operand specification> syntax:-**

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

  - <symbolic name>:
    Example: AREA- refers to memory word with which name AREA is associated

  - [+<displacement>]:
    Example: AREA +5 : refers a memory word of 5 word way from the word with name AREA. 5 is displacement.
  - [(<index register>)]
    Example: AREA(4): It implies indexing the operand AREA with index register that is operand address is obtained by adding the content of index register 4 to the address of AREA.

- Types of Assembly Language Statement

1. Imperative Statement
   - Indicates an action to be taken or performed during execution of program
   - Translate into one machine instruction.
   - Example: MOVE, ADD, MULT etc
2. Declarative Statement
   - They Declare storage or constant
   - To reserve memory for variable
     [Label] DS <constant> e.g X DS 5

[Label] DC<value> e.g X DC 3

❖ Constant and Literals:

- The Dc statement initialize memory words to given values
- The values of constant is changed by moving a new value into the memory word.
- The literal is an operand with the syntax:- =’<value>’
- Literal is different from constant because its location cannot be specified in assembly program
- Literals cannot be changed during program execution.
- Literal is more safe and protected than constant.
- Literals appear as a part of the instruction
- Example ADD AREG,=’5’

3. Assembler Directives

- Assemble directives can’t generate machine code. They are only used to instruct assembler to perform certain actions
- **START < constant >:-**
  This directive indicates that the first word of the target program will start on ROM memory location with address <constant>. START < 200> ROM location will be 200 where first machine code will reside.
- **END Directive :-**This directive indicates the end of the source program.

❖ Advanced Assembler Directives

1. ORIGIN:-
   - Syntax:

     ORIGIN <address specification>

     Where <address specification> is an <operand specification> or <constant>

   - This directive instructs the assembler to put the address given by <address specification> in the location counter or indicates that LC should be set to the address given by <address specification>

   - Example: ORIGIN Loop + 2

2. EQU:
   - Syntax

     <symbol> EQU <address Specification>

     Where <address specification> is an <operand specification > or <constant>
   - This statement simply associates the name <symbol> with the address specified by<address specification>. However the address in the location counter is not affected.
   - Example: Back EQU Loop

3. LTORG:

- The LTORG directive, which stands for 'origin for literals' allows a programmer to specify where literals should be placed- if a program does not use an LTORG statement, the assembler would enter all literals used in the program into a single pool and allocate memory to them when it encountered the END statement.
- Literals can be handled in 2 step:
    1. Literals is treated as if it is a<value> in a DC statement-memory word containing the value of the literal is formed.

    2. This memory word is used as the operand in place of the literals

Design specification for an Assembler

- 4 step approach to develop a design Specification for an Assembler:
    i. Specify the problem and identify the Information necessary to perform a Task
    ii. Specify the data structure to be used.
    iii. Define the format of the data structure required to record the information
    iv. Specify the algorithm to be used to obtain and maintain the information.

Design of working of Assembler:

Analysis Phase:

- Primary function- to build symbol table for synthesis phase to proceed.
- For this purpose it must determine the address with which the symbolic names in program associated.
- Determine address of each symbols called as memory allocation
- Location counter used to hold address of next instruction
- Isolated label, mnemonic opcode, operands, constant etc.

Synthesis Phase:

- Used data structure generated by analysis phase.
- To build machine instructions for every assembly statement as per Mnemonic code and there address allocation.
- Synthesis machine instruction as per source code.
- Used 2 database:
    1. Symbol table: Each entry of symbol table has 2 primary field:-*name* and *address*
    2. Mnemonic table: An entry in the mnemonics table has 2 primary fields:- *mnemonic* and *opcode*

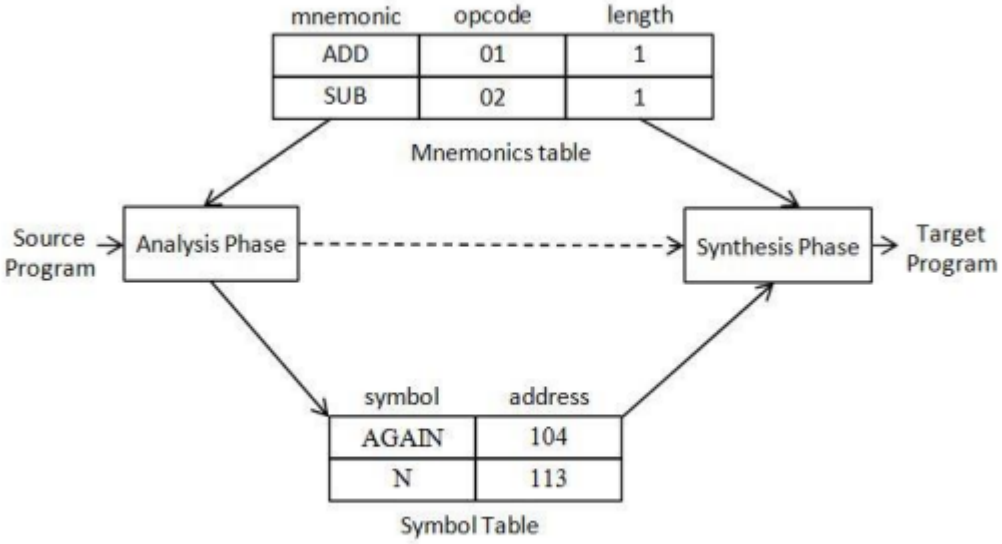


Figure 4.2.1: Design of Assembler

- The figure describe the use of the data structures by the analysis and synthesis phases.
- The Mnemonics table is fixed table which is only accessed by the analysis and synthesis phases
- The symbol table is constructed during analysis and used during synthesis
- The tasks performed by analysis and synthesis phases are as follows:

Analysis Phase:

1. Isolate the label, mnemonic opcode and operand fields of statement.
2. If a label is present, enter the pair (symbol, <LC counter>) in new entry of symbol table.
3. Check validity of the mnemonic opcode through a look-up in Mnemonic opcode
4. Perform LC processing-update the value contained in LC by considering the opcode and operand

Synthesis Phase:

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 or the machine form of constant

Example and program of Assembly Language:

[Label] [opcode] [operand]

Example:

M ADD R1 , ='3'

Where, m- label, ADD- symbolic opcode, R1- symbolic register operand, ='3' - literals

Label	Opcode	Operand	LC Value
A	START	200	
	MOVER	R1,='3'	200
	MOVER	R1,X	201
L1	MOVER	R2,='2'	202
	LTORG		203
X	DS	1	204
	END		205

Where A- name of the program passed to loader,
START-pseudo opcode
200- Instruction will be stored starting related memory location 200.

- **Databases required for designing of Assembler:**

1. **Symbol table(SYMTAB)**:
   - Symbol table is used to contain all the symbols used in the program
   - Types of symbol tables include variable, procedure defined constant, label etc.

Label	Address
A	200
L1	202
X	204

2. Literal table(LITTAB):
   - It contains the information about all the literal encountered in the assembly program
   - A LITTAB entry contains the field literal and address
   - The first pass use LITTAB to collect all literals used in a program

Index	Literal	Address
0	='3'	200
1	='2'	202

3. Mnemonics Table or Machine operation table(MOT):

- MOT is a static or fixed i.e. content cannot be changed during lifetime of the assembler.
- The mnemonic is the key for searching MOT and its value is the binary code that is used for generation of machine code.
- Indicates the symbolic mnemonic for each instruction along with its length(2,4or6 bytes)

Instruction Opcode	Assembly Mnemonic	Remark
00	STOP	Stop execution
01	ADD	OP1← OP1+OP2
02	SUB	OP1 ← OP1-OP2
03	MULT	OP1 ← OP1*OP2
04	MOVER	CPU register ← Memory
05	MOVEM	Memory ← CPU register
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	OP1 ← OP1/OP2
09	READ	Operand 2 ← Input value
10	PRINT	Output ← Operand 2

Instruction	Opcode	Length(Bytes)
MOVER	3	2
MOVEM	X	1
MOVER	2	2

4. Pseudo-opcode table(POT) or (OTTAB):

- Pseudo operation code table(POT) contains a field [mnemonic opcode, class and mnemonic information(R# routine number)]
- Indicates the symbolic mnemonics and action to be taken for each pseudo opcode in pass I.
- Class field-indicates whether the opcode corresponding to an imperative statement, declaration statement or Assembler Directives
- Routine –perform appropriate processing of the statement

Mnemonic opcode	Class	Mnemonics Information
START	AD	R#11
LTORG	AD	R#5
ORIGIN	AD	R#3
DS	DL	R#7
EQU	AD	R#4
END	AD	R#2
MOVER	IS	(04,1)

5. Location counter(LC):

- Location counter keeps the track of each instruction's location
- It is used to perform memory allocation

Variable	Address
L1	202
X	204

6. Pool table(POOLTAB):

- Whenever an assembly program contains multiple LTORG statement then it creates different literal pools.
- POOLTAB is used to record the details of all the different literal pools.
- This table contains the literal number of the starting literal of each literal pool
- At any stage the current literal pool is the last pool in the literal table
- On encountering an LTORG statement or END statement literal in the current pool are allocated address starting with the current value of LC and LC is appropriately incremented.

- **Pass Structure of Assembler:**
  - **A. Single pass Translation**
  - **B. Two pass Translation**

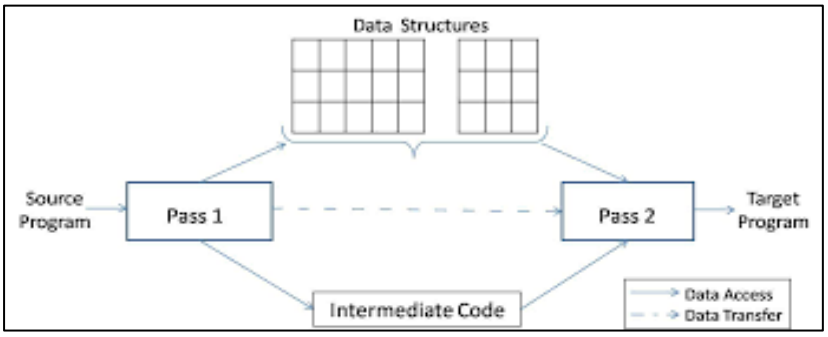


fig. pass structure of assembler

A. Single pass Translation:

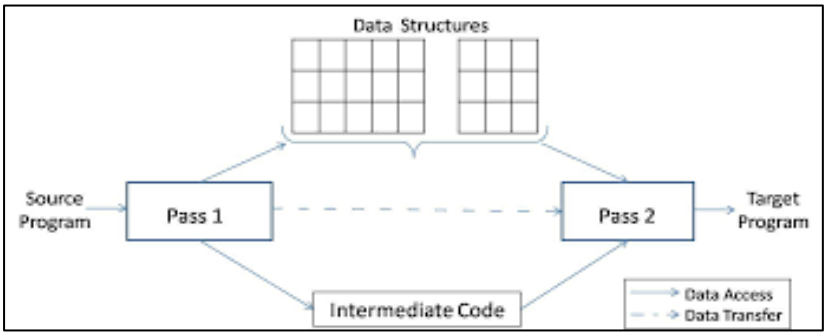
- Require one scan of the source program to generate machine code.
- LC processing, symbol table construction and target code generation proceed in pass II
- The issue of forward reference can be resolved using a process call back patching.

- Forward reference of a program entity is a reference to the entity which precedes its definition in the program.
- The operand field of an instruction containing a forward reference is left blank initially
- The address of forward referenced symbol is put into this field when its definition is encountered.
- It builds a Table of Incomplete Information about instruction whose operand fields were left blank(TII)
- TII contains ***instruction address, symbol.***
- When END statement is processed the symbol table would contains address of all symbol defined in source program.
- TII would contain information describing all forward reference
- The assembler can now process each entry in TII to complete the concerned instruction.

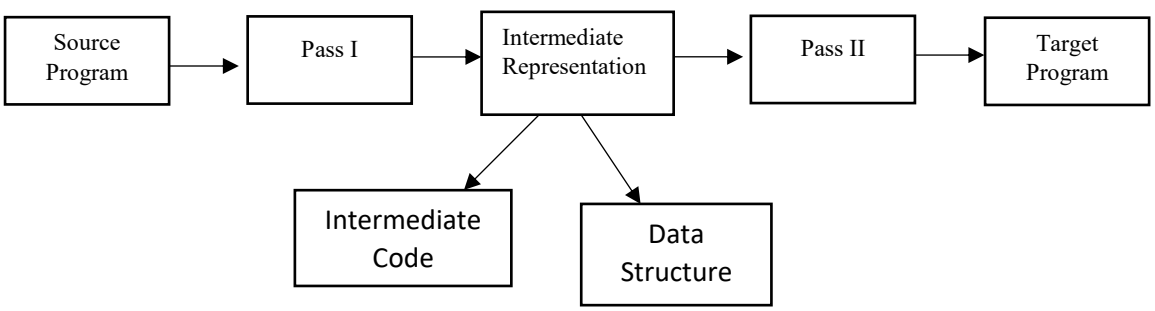
B. Two pass Translation:

- Pass II perform analysis of source program
- It perform location counter processing and records the address of symbols in symbol table
- It construct Intermediate Representation of source program
- Intermediate Representation consist of 2 component:
    i. Intermediate code
    ii. Data Structure
- Synthesizes the target program by using address information stored in symbol table
- It handles the forward reference to a symbol naturally because of the address of each symbol would be known before program synthesis program.

○ Design of Two pass Assembler:



Or



Refer explanation of single pass and two pass translation.

- Task performed by the pass of two pass assembler are follows:

Pass I:

i. Separate the symbol, mnemonic opcode and operand fields.
ii. Built the symbol table
iii. Perform LC processing
iv. Construct intermediate representation

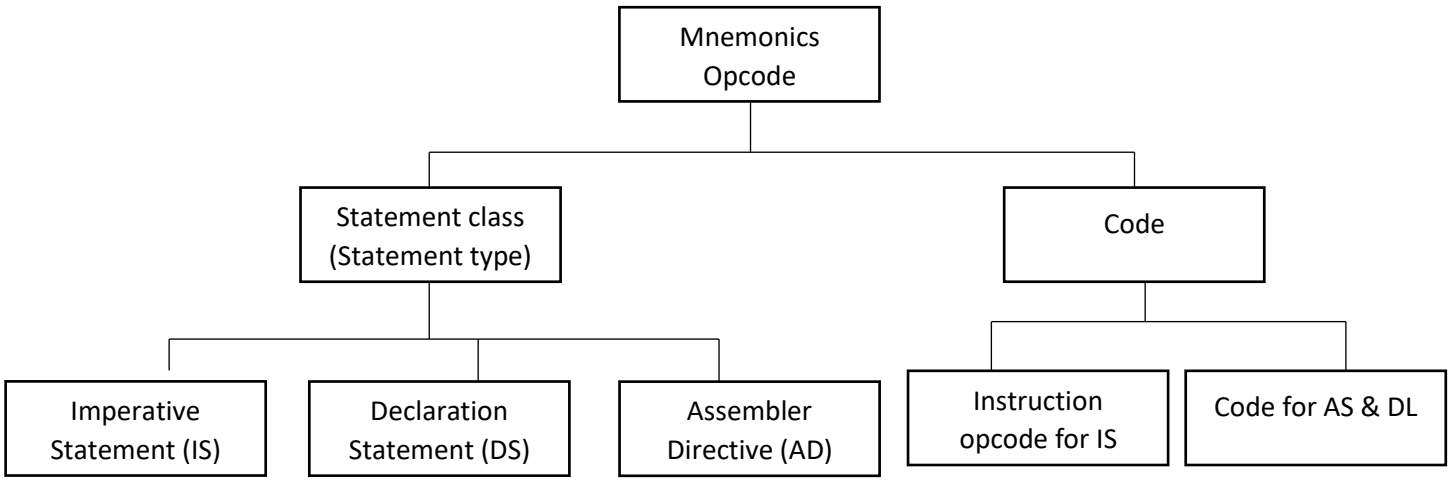
Pass II:

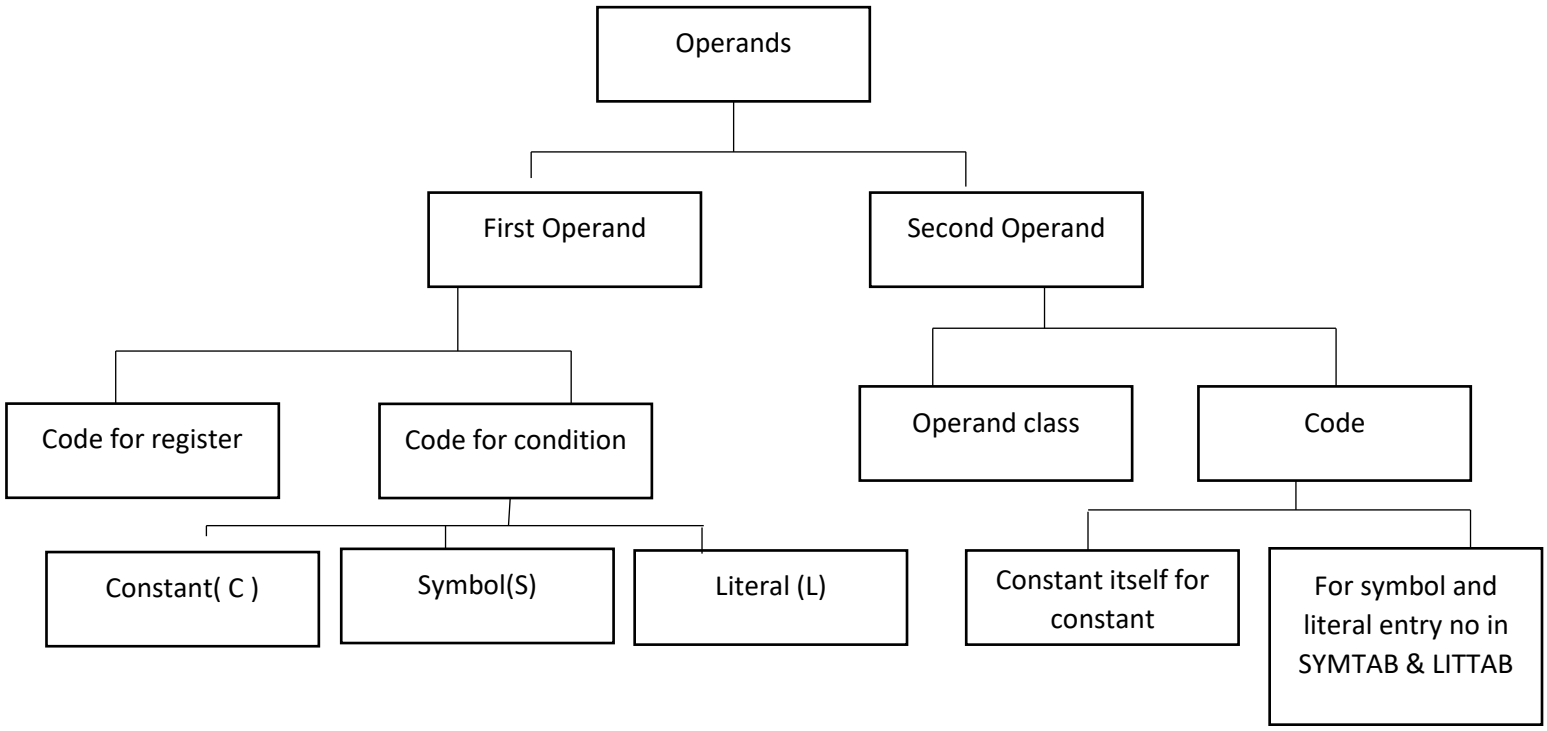
i. Synthesize the target program

- **Format of Intermediate code:**
  - The intermediate code consists of a set of IC units, each IC unit consisting of following 3 fields.
  - 1.Address
  - 2.Representation of the mnemonic opcode
  - Representation of opcode.

Syntax:

Address	Mnemonics opcodes	Operand





- Code for Address Directive(AD):

START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

- Code for Declaration Statement(DL):

DC	01
DS	02

- Code for Register:

AREG	01
BREG	02
CREG	03
DREG	04

- Code for condition

LT	01
LE	02
EQ	03
GT	04
GE	05
ANY	06

- **Pass I : working and Algorithm**

Pass I uses the following **data structure**:

1. **OPTAB:** A table of mnemonic opcodes and related information.
   It contains the field's mnemonic opcode, class and mnemonic information.
   The class field contains opcode corresponding to IS, DL and AD.
   The mnemonic information contain machine opcode and instruction length

2. **SYMTAB:** Symbol table
   It contains address and length.
3. **LITTAB:** A table of literals used in the program.
   It contains literal and address.
4. **Forward Reference:** symbol that are defined in later part of the program are called forward reference. There will not be ant address value for such symbols in symbol table in pass 1. This can be tackled using backpatching.

Working of Pass I:

Algorithm of Pass I:

1. loc_cntr=0;
   pooltab_ptr=0;POOLTAB[1]=1; littab=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 SYMTAB
   b) If an LTORG statement then
      i. Process literals to allocate memory and put the address in the address field.
         Update loc_cntr accordingly
      ii. pooltab-ptr=pooltab+1;
      iii. POOLTAB[pooltab-ptr]=littab-ptr;
   c) If a START or ORIGIN statement then
      Loc_cntr= va;ue specified in operand field;
   d) If an EQU statement then

i. This-addr=value specified in <address specification>
ii. Correct the SYMTAB entry for this_label to(this_label,this-addr)

e) If a declaration statement then
   i. Code=code of the declaration statement
   ii. Size=size of memory area 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 OPTAB;
   ii. Loc_cntr=loc_cntr+instruction length from OPTAB;
   iii. If operand is a literal then
   this-literal=literal in operand field
   LITTAB[litab-ptr]=this-literal;
   Littab-ptr=littab-ptr+1;
   Else
   This-entry=SYMTAB entry number of operand
   Generate IC'(IS,code)(S,this-entry);

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

- **Pass II: working and Algorithm**: Minor changes may be needed to suit the IC being used. It has been assumed that the target code is to be assembled in the area named code-area

1. code-area-address=address of code-area;
   pooltab-ptr=1;
   loc-cntr=0;
2. While next statement is not an END statement
   a) Clear machine-code-buffer;
   b) If an LTORG statement
      i. Process literals in LITTAB and assemble 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. 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 SYMTAB or LITTAB
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 end statement
a) Perform steps 2(b) and 2(f)
b) Write code-area into output file.