Introduction to System Software

- Hardware is a physical entity of a machine at abstract level.

Characteristics of Machine
1. Stable
2. Predictable
3. Structured

- Languages { High level, ASL [Assembly], Machine level }

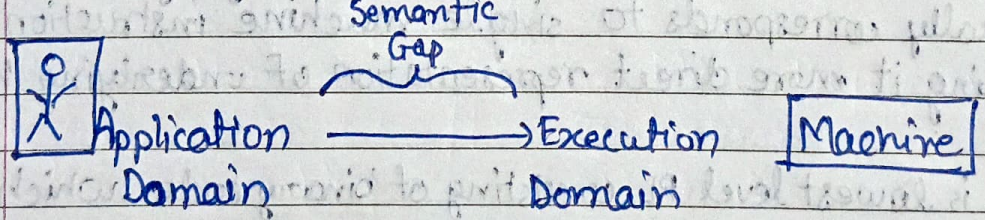
1. HHL are designed to be more user-friendly and abstract, providing a higher level of abstraction from the hardware.
Characteristics: Readability, Abstraction, Portability

2. ASL is a low level language that is symbolic repr. of machine code, specific to a particular computer architecture.
Characteristics
- Symbolic representation: Instead of using binary code directly, assembly language uses mnemonic codes and symbols to represent machine instructions.
- One-to-one mapping: Each assembly language instruction typically corresponds to single machine instruction, making it more direct representation of underlying hardware.

3. MSL is lowest level PL consisting of binary code, which is directly executed by Computer's CPU.
Characteristics:
- Binary representation
- Hardware specific [less portable]

Feature	HLL	ASL	MLL
① Abstract Level	High	Low to Medium	Low
② Readability	High	Moderate	Low
③ Portability	High	Low to Medium	Low
④ Development speed	Fast	Moderate	Slow
⑤ Programmer's Productivity	High	Moderate	Low
⑥ Syntax	Human Friendly		
⑦ Memory Management	Automatic	Manual	Manual
⑧ Hardware Dependency	Low	Moderate	High
⑨ Flexibility	High	Moderate	Low
⑩ Error Handling	Automatic	Minimal	Limited
⑪ Examples	C, Java, Python	asm for specific architecture	Binary code specific to processor

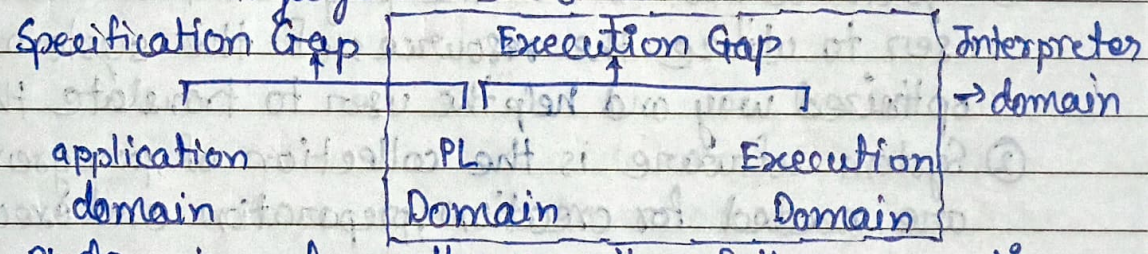
- Semantic Gap



The semantic gap between application and execution domain is breached by a logical entity [set of programs] called the software.

Specification gap is bridged by software development Team.
Execution gap is bridged by designers of programming language processors.

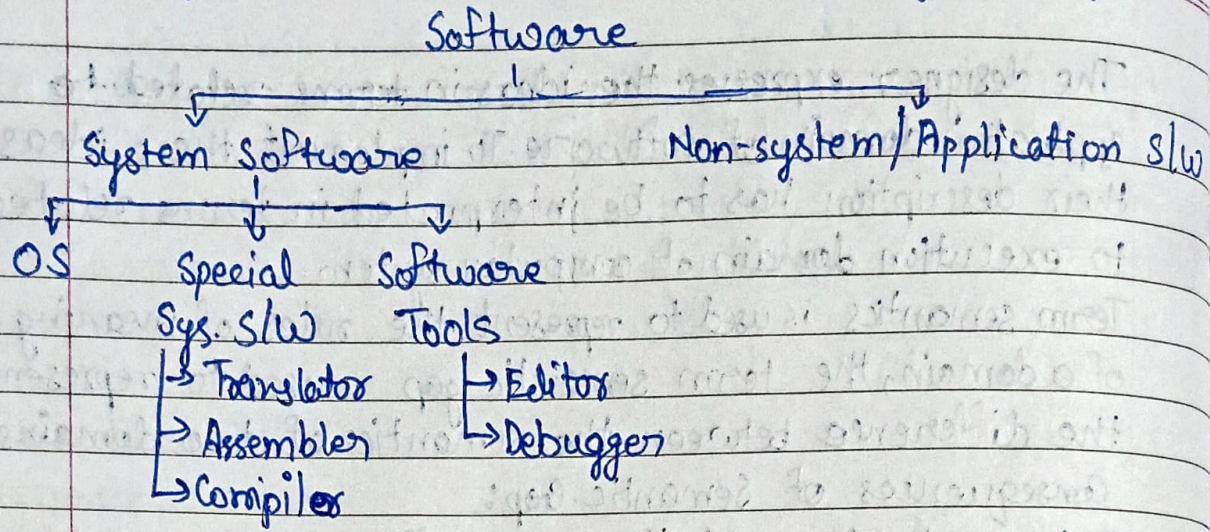
The designer expresses the ideas in terms related to application domain of software. To implement these ideas, their description has to be interpreted in terms related to execution domain of computer system.
Term semantics is used to represent the rules of meaning of a domain, the term semantic gap is used to represent the difference between the semantics of two domains
Consequences of Semantic Gap:
① Large development times and efforts.
② Poor quality of software.
Tackle using → Software implementation using Programming Language [PL] Domain.



Using PL domain reduces the severity of the semantic gap. Language processor provides a diagnostic capability which detects and indicates errors in its input. This helps in improving quality of software.

Language processor is a software with bridges a specification or execution gap. A interpreter is a language processor which bridges an execution gap without generating a machine language program.
Language processing activities arise due to difference between the manner in which a software designer describes the ideas concerning the behaviour of software and the manner in which these ideas are implemented in computer sys

Specification gap is the semantic gap between two specifications of the same task. Execution gap is gap between the semantics of programs written in different programming languages.

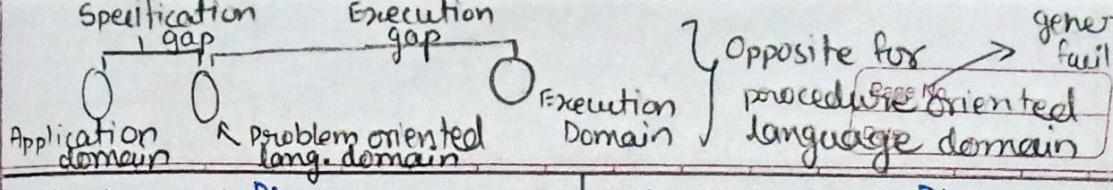


- Software: System Software

* ① System softwares are the programs that allows the user to utilize the resources of the system in the optimized way and help the user to translate there programs
② System software is the collection of programs which are needed for creation, preparation and execution of user programs.
③ System Software is a program which aids in effective execution of general users; computational requirements on a computer system.

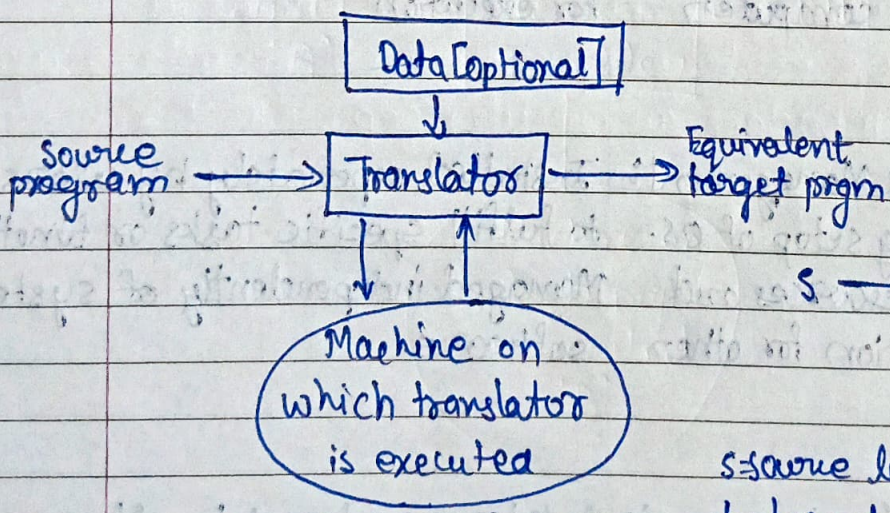
- List of Translators

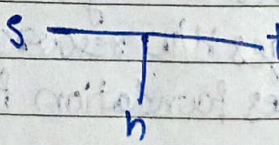
① Assembler
② Compiler
③ Cross-assembler
④ Cross-compiler
⑤ Preprocessor
⑥ Micro-Preprocessor
⑦ Linker
⑧ Loader
⑨ Interpreter.
⑩



System Software	Application Software
① Purpose: Manages & controls computer hardware and provides a platform for running application s/w.	Performs specific tasks for end-user, addressing their needs and requirements.
② Examples: OS, device drivers, utilities	Word processors, spreadsheets, Browsers, games, etc.
③ Interaction: Primarily interacts with computer hardware and other sys s/w - Less interaction with end-users.	Directly interacts with end-user, providing functionality for specific tasks or activities.
④ Functionality: Provides a platform for running applications, manages memory, handle I/O, and ensures hardware resources are utilized efficiently	Tailored to specific tasks or functions, providing features such as document editing, data analysis, gaming, etc.
⑤ Dependency: Necessary for functioning of computer system.	Dependent on system software for execution.
⑥ Installation and Management: Installed during setup of OS. Manages H/w resources and provides foundation for other softwares.	Installed seperately by users to fulfill specific tasks or functions. Managed independently of system software.
⑦ Regular updates & maintenance is critical for security and performance enhancement.	Updates are released to add new features, improve performance and address bugs. Maintenance is often user-driven.

- Steps to convert source file to executable file.





s=source language
t=target language
h=Platform on which translator is executed.

- Editor → Special type of System Software
  Tools → OS, Special system software, Software Tools [Editor / Debugger]

x.HLL
↓
x.obj
↓
x.MLL
↓
x.exe
↓
called as linker module which is given to Loader
Assumes prgm starts at 0000 location

Editor
↓
Pre-processor
↓
Compiler → o/p is obj file
↓
Assembler
↓
Linker
↓
Loader

sometimes they are built-in

x.HLL
x.S
x.obj
linker module
loader module
executables

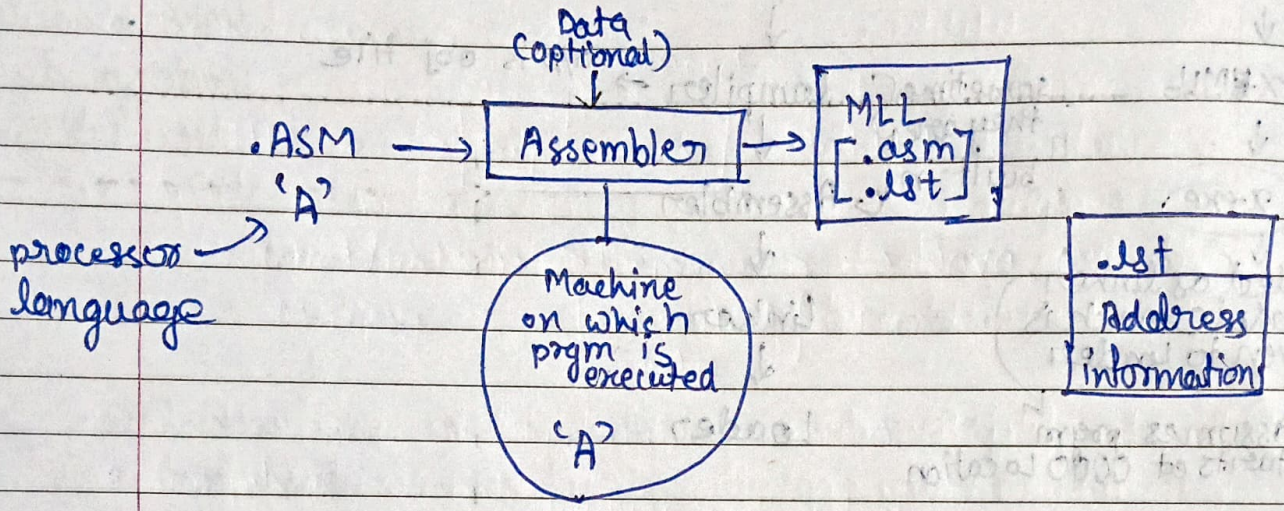
Loader ⇒
① Consults to Memory Module of OS.
OS returns starting address and length of program address.
② Reallocate Memory addresses.
③ It will give load module which is not in main memory. Load file [exe] from secondary into main/primary memory.

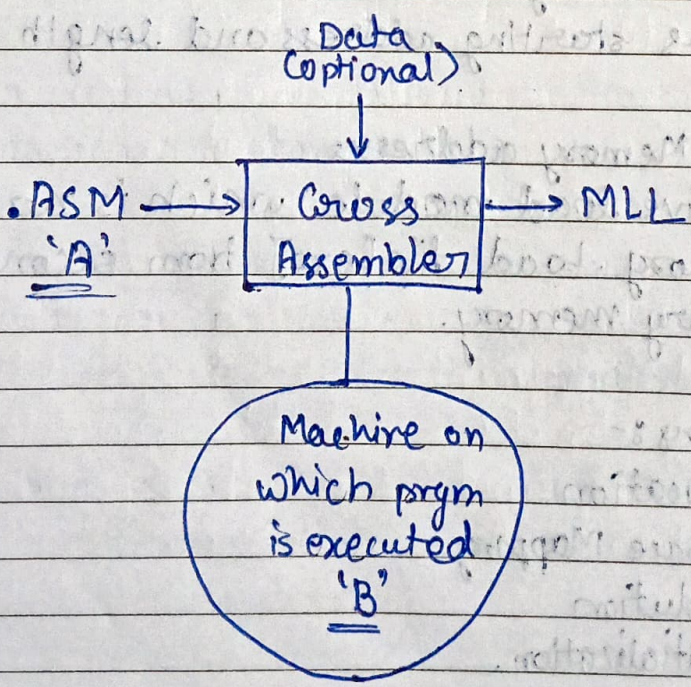
Main functions :-
① Memory allocation
② Address Space Mapping
③ Symbol Resolution
④ Program initialization.

- How many system calls are required for loader?

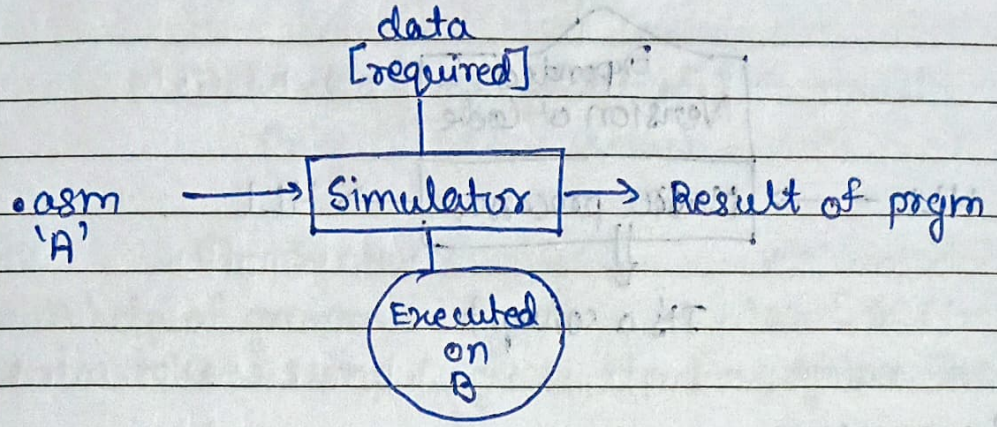
→ Depends on OS.
Linux and ~~BSD~~ OpenBSD have over 300.
NetBSD and FreeBSD have over 500.
Windows has close to 2000.

- Assembler





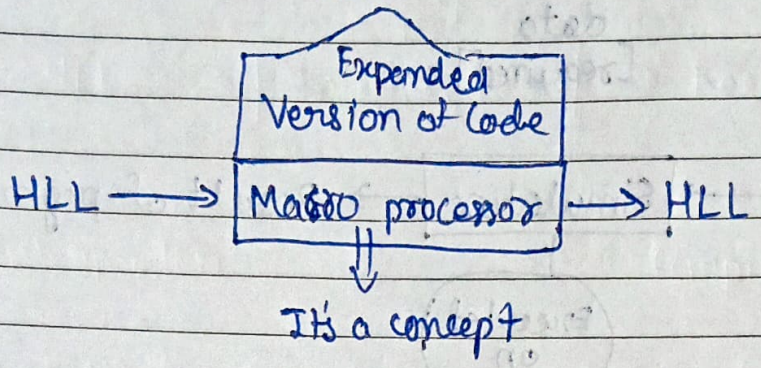
- Simulator [Translator]



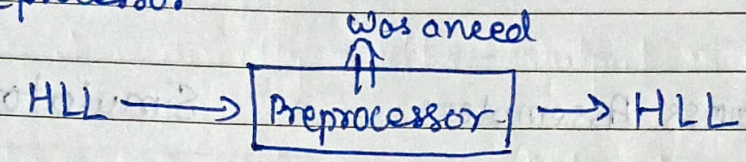
Parameter	Cross Assembler	Simulator
①	Totally Hardwired	Software
② Data	Optional	Required
③ Knowledge	To design, knowledge of both sides is required.	To design, no knowledge need of both simulators, only needs architecture of prgm. like [8086] which can be executed anywhere
④ Purpose	Converts assembly lang. code for one architecture to machine code for diff. archit.	Simulates execution of program without running it on actual hardware
⑤ Input	Src. code in asm ~~or HLL~~	asm or HLL code as input.
⑥ Output	Outputs ~~simulation~~ machine code or obj. code for target architecture	Outputs simulated execution results, such as register values, memory contents and prgm. flow.
⑦ Platform dependency	Platform independent	Platform dependent. (simulators compatability with target machine)
⑧ Example	NASM, GNU assemblers	QEMU

Page No.
Date

- Macro Processor

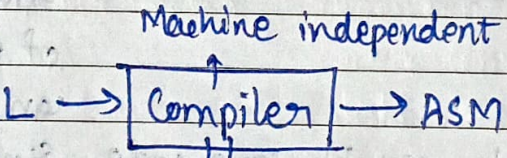


- Preprocessor



Cobol/Fortran → Preprocessor → Java, other langs.
[Converted into]

- Compilers



If platform changes then cross-compiler

Phases:

Src Code

↓

① Lexical analyzer [LA]

[Only Tokenization]

↓

② Syntax Analysis

[With help of grammar checkings syntax [CFGs]
Receive tokens as input, gives output as syntax Trees]

↓

③ Semantic Analysis

[logical errors are checked, output is Parse Tree]

↓

④ Immediate Code Generation

[Machine independent code is generated]

↓

⑤ Code Optimization

[Optimize Machine Code, Less Memory, More efficient]

↓

⑥ Code Generation

[.asm file with address information (.lst)]

①, ②, ③ ⇒ Analysis Phase

④, ⑤, ⑥ ⇒ Synthesis Phase.

lexical analyzer → Tokenization

Syntax analysis → CFGs used, Syntax Trees are generated

Semantic analysis → Logical errors checked, Parse Tree generated

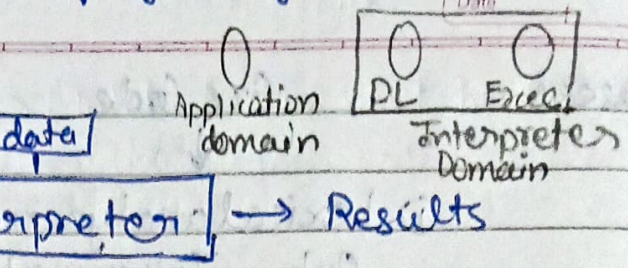
Intermediate O/P generation → independent machine code

Code Optimization → Optimize

Code Generation → .asm file

A language processor which bridges an execution gap without generating a machine language program.

- Interpreter



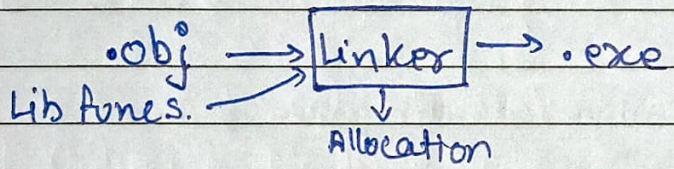
Interpreter	Compiler
① Line by line execution	Whole code executed together
② Output is directly result.	Output is machine code.
③ Data is required.	Data not required.
④ Less Memory	More Memory.
⑤ Source code is directly portable but requires an interpreter for each platform	Compiled code is platform-dependent, but same src code can be compiled for different platforms
⑥ Slower speed as it interprets code during runtime.	Faster as translation is done before runtime
⑦ Easy to debug due to ability to execute code line by line.	Debugging is more challenging as it involves working with compiled machine code.
⑧ Ex. Python, JS, Ruby	C, C++, Rust.

- First interpreter is BASIC developed by Bill Gates.
Final stage of Booting is command interpreter.

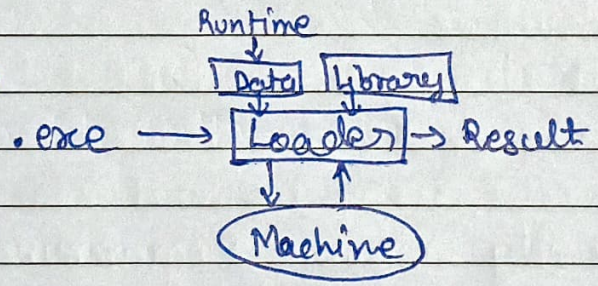
- Java is Compiled or interpreted?

→ Java is both compiled and interpreted.
Java source is first compiled into bytecode by Java compiler. The bytecode is platform-independent intermediate representation of Java Program.
JVM then, executes Java bytecode. The JVM is a platform independent program.
The bytecode is stored in a file .class extension.
When we run a Java prgm, JVM loads bytecode from .class file and interprets it.

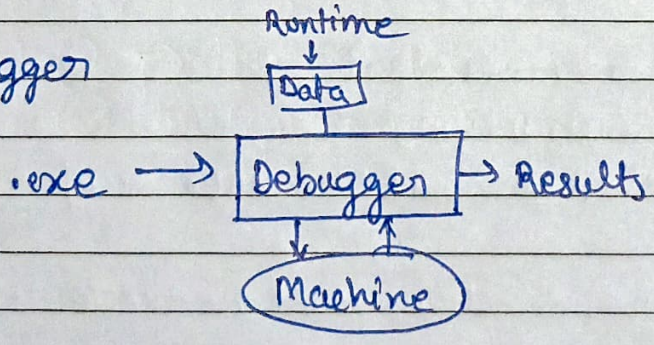
- Linker



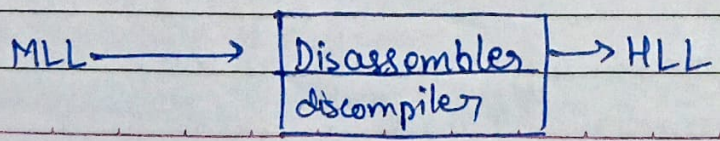
- Loader



- Debugger



-



UNIT 2: ASSEMBLER

- Assembly language provides 3 basic features:
  1. Mnemonic Operation Codes
  2. Symbolic Operands
  3. Data declarations

- Assembler
  1. Machine Dependent: Knowledge of Microprocessor is necessary.
     - Architecture [No. of address, data lines]
     - Registers [Accumulator, General Purpose registers]
     - Instruction set
     - Addressing Modes

- Instruction set for Hypothetical machines
  1. Data Transfer
  2. Arithmetic instructions
  3. Input / Output
  4. Branching

- Skeleton of Assembly Language Programming [ALP]

  1. Header → directives, start
  2. Body → Instructions
  3. Footer → End of Prgm [END, HALT, STOP], Execution

In header of prgm, directives must be there, telling that this is beginning of the program.

Directives ⇒ start → 1st location of user area
org → specify location from where to start

- In Body of Program : 3 types of statements
  1. Assembler directives
  2. Instructions
     - Mnemonics
     - Operands
  3. Comments
     - Start on new line
     - At end of instruction on same line
     - At end of instruction and on next line also

- Normal format
  [label] mnemonics [operand]

- Format of ALP

Address	Opcode	Label	Mnemonic	Comments

We can get the instruction length / size of mnemonic by MOT ['Mnemonic + Opcode'] Table.

Mnemonic	Opcode	No. of Operands	length of instruction	Optional: Meaning	Optional: Example

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

- MOT Table.

Mnemonic	OPCODE	No. of Operands	Length of instruction	Meaning	Example.
ADD	1	1	1+1 = 2 bytes		
SUB	2	1	2		
MUL	3	1	2		
JMP	4	1	2		
JMEG	5	1	2		
JPOS	6	1	2		
JZ	7	1	2		
LOAD	8	1			
STORE	9	1			
READ	10	1			
WRITE	11	1			
STOP	12	1			
HLT	13	1	1		
END	14				

Page No.
Date

- Steps assembler has to follow:
  ① Read statement line-by-line till end of file
  ② Analyze statement whether it is comment, pseudo opcode (Directive used in program) or mnemonics

  - MOT → List of mnemonics
  - POT → List of pseudo opcodes. [List of directives with no. of arguments]
  - ST [Symbol Table]

  ③ If statement is comment then ignore it
  ④ If pseudo opcode then take corresponding action.
  ⑤ If mnemonics then replace it with machine opcode and replace symbols with their relative addresses.

For doing this activity, sometimes one pass may not be enough to get all the information, so is some cases 2nd / more passes are necessary.
In 2 pass assembler, the object file is opened for writing in 2nd pass.

Pass → A complete scan of code.
Phase → Total activity of assembler is divided in 2 Phases:-
① Analysis ② Synthesis

- Analysis Phase:
  ① Isolate labels, mnemonics, operands and comments of statements.
  ② Check validity of mnemonics by consulting MOT
  ③ Check no. of operands, required for an instruction by consulting MOT

8086 have 16 columns in Symbol Table

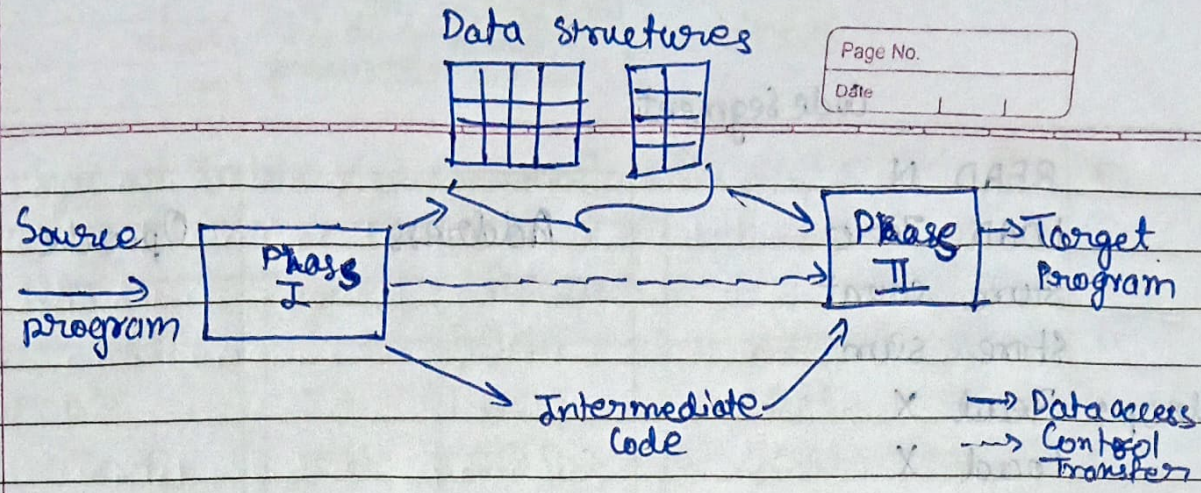
④ Process labels and symbols appropriately [with due considerations to duplicate definition of some labels or symbols] and fill into symbol table.
⑤ Update location counter appropriately by consulting MOT for length of instruction
⑥ Ignore comments
⑦ Take proper action for pseudo opcodes for consulting POT

- Synthesis
① Obtain Machine Opcode Corresponding to mnemonics by consulting MOT
② Fill addresses of symbols or labels by consulting Symbol Table.
③ Write above information in object file.

- Symbol Table have : Name of symbol, Type, Address

- Different data structures are needed for implementation of Table:
3 pointers for MOT, POT and Symbol Table.

- Two Pass Translation
Can handle forward referencing easily. LC processing is performed in 1st pass and symbols defined in the program are entered into symbol table. The 2nd pass synthesizes target form using addresses found in symbol table.
1st pass → Analysis of source program
2nd pass → Synthesis of target program



- Single pass translation

The problem of forward references is tackled using a process called backpatching. 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. This is indicated by adding an entry to Table of Incomplete Instructions [TII]

address where instr. is

101 ⟶ MOVER BREG, ONE → Forward referencing

TII pair → (<instr. address>, <symbol>) (101, ONE)

By the time END stmt is processed, assembler goes through each entry from TII, consults symbol table and inserts the address.

- Design of 2 Pass Assembler.

Pass I-

① Seperate symbol, mnemonic opcode and operand fields.
② Building symbol table.
③ Perform LC processing
④ Construct intermediate representation

Pass II - Synthesis target program

used before defⁿ is called forward referencing

Page No.
Date

Code Segment

```
①  READ N
②  LOAD Zero
3  Store count
4  Store sum →
loop: 5 Read X
6  Load X
7  Add sum
8  Store sum
9  Load count
10 add one
11 store count
12 sub N
13 JZ outer
14 Jump loop
outer: 15 write sum
16 Stop
17 EndP → In POT        (End of code)
```

Data Segment

```
Zero   count
one    count
sum    db
count  db
N      db
X      db
End →                   (End of 1st Pass, go to 2nd Pass)
```

Location Counter → 0 initially.

Address	Opcode
0000	10
0001	0035
0002	8
3	–
4	9
5	–
6	9
7	–
loop 8	10
9	–
10	8
11	–
12	1
13	–
14	9
15	–
16	8
17	–
18	1
19	–
0020	9
0021	–
23	2
23	–
24	7
25	– outer
26	9
27	–
outer → 28	11
29	–
30	12

If in POT, do not increment address

Page No.
Date

Symbol Table [no duplicates]

	Name	Type	Address	
0000	N	Id	-	0035
0002	zero	Id	-	0031 → In DS.
0004	count	Id	-	0034
0006	sum	Id	-	0033
0008	loop	Label	0008	
	X	Id	-	0036
	outer	Id [1st Pass]	-	
		Label	0028	
	one			0032

0031	zero
32	one
33	sum
34	count
35	N
36	X

Directive

- org 1000
  load A
  add one
  store A
  ENDP

DS⇒ DB A - 1
Const one 1
END

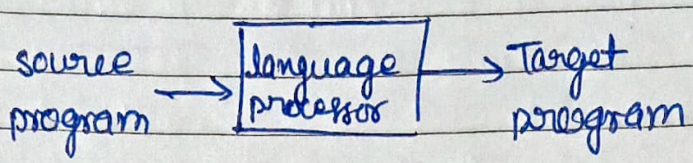
Address	Opcode/Add
1000	8
1	- 1006
2	1
3	- 1007
4	9
5	- 1006
6	
7	
8	

ST

Name	Type	Add.
A	Id.	- 6
one	Const	- 7

- Handling Constants. → 3 Methods

- A language generator processor is assumed to have diagnostic capability implicitly. It typically abandons generation of target program if it detects errors in source program.



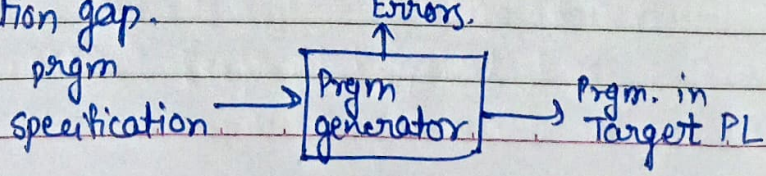
① A language translator bridges an execution gap to machine language of a computer system.
An assembler is a language translator whose src. lang. is assembly language.
A compiler is any language translator which is not an assembler.

② A detranslator bridges the same execution gap as lang. translator, but in reverse direction.

③ A preprocessor is a lang. processor which bridges an execution gap but is not a lang. translator.

④ A language migrator bridges specification gap between two PLs.

- Program Generator: a system s/w
Accepts specification of a program to be generated, and generate prgm in PL [target]. It introduces new domain between application and PL domains which is program generator domain. This gap is now called specification gap.



Page No.
Date

Linker

Execution of program steps:

1. Translation → Translator
2. Linking → Linker Work
3. Relocation → Linker Work
4. Loading → Loader

Linker Tasks:

1. Origin of program may have to be changed.
   Names for start time address:
   i) Translation time address [by Translator]
   ii) Linked ~~origin~~ address
   iii) Load ~~origin~~ address

EXTERN and ENTRY statements
↓ Symbols to which external refs. are made
→ List public definitions which can be used further in prgm. units

Linking is the process of binding an external reference to the correct link time address.
Linker converts object modules in set of program units SP into a binary program.

Object Module of a prgm contains all information necessary to relocate and ilink prgm with other prgms.
It consists of 4 components:-

1. Header
2. Program
3. Relocation Table
4. Linking Table

ASSEMBLER

- Types of statements:

① Imperative - an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction. IL

② Declarative - DL

Syntax: [Label] DS <constant>
[Label] DC 'value'

Ex: A DS 1
G DS 200

1st statement reserves a memory of 1 word and associates the name A with it. 2nd stmt reserves a block of 200 memory words associated with G. Other words in the block can be accessed using offset.
6th word ⇒ G+5

DS → short for declare storage
DC → short for declare constant.

ONE DC '1'

③ Assembler directives - AD
Instructs assembler to perform certain actions during the assembly of program.

START <constant>
ORIGIN
EQU
LTORG
END [<operand spec>]

① Symbol Table → Two fields: Name and address
Table is built in analysis phase

② Mnemonic Table → Mnemonic and Opcode
Used in Synthesis Phase.

③ Location counter → Implement memory allocation.
LC is always made to contain the address of next memory word in target prgm. It is initialized to constant specified by START statement.

DATA STRUCTURES FOR LANGUAGE PROCESSING

- Data structure used in language processing can be classified on the basis of following criterias -
  ① Nature of DS → Linear / Non-Linear
  ② Purpose of DS → search / allocation DS
  ③ Lifetime of DS → whether used during language processing or during target program execution.

① Linear data structure:
- requires contiguous memory
- size of data structure is difficult to predict.
- if forced to overestimate, it will lead to memory wastage

② Non-linear Data Structure
- Elements are accessed using pointers
- Not occupy contiguous areas of memory.
- Search efficiency is low.

③ Search Data Structures:-
- Used during language processing to maintain attribute information concerning different entities in source program.
- Entry for entity is created only once, but searched for large number of times.

④ Allocation Data Structures:-
- Address of memory area allocated to an entity known to user(s) of that entity.

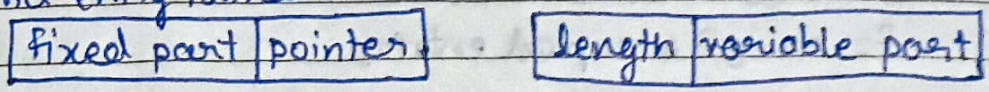
- Language processor uses both search & allocation DS:-
  - Search DS used to ~~handle prgms~~ constitutes various tables of information.
  - Allocation DS used to handle prgms with nested structures of some kind.
- Target Prgm rarely uses Search DS, but it may use allocation DS.

- Search Data Structure
Set of entries, each ~~entity~~ entity accomodating the info. concerning one entity. Each entry is assumed to contain a key field which forms basis for search

Entry format → ① Fixed Part - Value of tag field determines the information to be stored in variant part. Fixed and variant parts can be nested.

Symbol Table entries ⇒ Fixed = Symbol, class
Variant = variable, procedurename, functionname, label.

- Fixed entry format
ii Variable entry format
iii Hybrid entry format



- Operation on search data structure
① Add
② Search
③ Delete

- Generic Search Procedure - Algorithm
① Make a prediction concerning the entry of search data structure which symbol 's' may be occupying. Let this be entry 'e'.
② Let 's_e' be symbol occupying e^{th} entry. Compare 's' with 's_e'. Exit with success if the two matches.
③ Repeat steps 1 and 2 till it can be concluded that the symbol does not exist in search DS.

Each comparison at Step 2 is called probe.
p_s = No. of probes in a successful search
p_u = ——— unsuccessful

- Table Organization
① Linear DS.
② A particular location of entry can indicate next and prev entry of table.
③ Fixed length. Positional determinacy.
Address of e^{th} entry = $a+(e-1).l$
a = address of 1st entry, l = length of entry

Page No.
Date

- Sequential search Organization

n = no. of entries in table
f = no. of occupied entries

$p_s = f/2$ for successful search
$p_u = f$ for unsuccessful search.

Following an unsuccessful search, symbol may be entered into table using add operation.

- Add Operation
Value added to 1st free entry; then value of f is updated. [Symbol]

- Delete Operation
* Two ways
① Physical: deleted by erasing or by overwriting
if dth entry is to be deleted, f-d entries [d+1 to f] can be shifted up by 1 entry each. Efficient way to do this is move fth entry to dth position.
But this changes entry nos. of symbol table, which interferes representation of symbol in IC.

② Logical: Adding some information of entry to indicate it is deleted. This can be implemented by introducing a field to indicate whether an entry is active or deleted.

Active/deleted	Symbol	Other info

Page No.
Date / /

- Binary Search Organization
All entries of table are assumed to ~~satisfy~~ satisfy an ordering relation. Suitable only for table containing a fixed set of symbols.

- Hash Table Organization.
e is a function of s.
3 Possibilities:
① entry may be occupied by s.
② entry may be occupied by some other symbol.
③ entry may be empty.
In ②, i.e. $s \neq s_e$ is called collision.

Hashing function.
Two popular classes of hashing functions are:
① Multiplication functions
② Division functions

Collision Handling methods:
① Rehashing ② Overflow Chaining

⊙ Allocation Data Structures

- **Stack -**

① Stack is linear DS.
② LIFO, only last entry is accessible.
③ A pointer stack base [SB] points to first word of stack area.
④ A pointer Top of stack [TOS] to point to last entry allocated in stack.

⑤ When entry is pushed, TOS is incremented by d.
[d = size of entry]
When popped, TOS is decremented by d.
If TOS = SB - 1 → Stack empty.

- **Extended Stack Model**

Entities may not be of same size
Two new pointers:
① Record base pointer - 1st word of last record in stack.
② First word of each record is reserved pointer.

Page No.
Date / /

- Heap
Non-linear data structure, permits allocation and deallocation of entities in a random order.

⊙ Memory Management
Identifying the free memory areas and reusing them while making fresh allocations.
Two popular techniques:-
① Reference Counts
② Garbage Collection

- Reference Count → Garbage Collection
System associates a reference count with each memory area to indicate no. of its active users. In latter technique, system performs garbage collection which makes 2 passes over memory to identify unused areas. In 1st pass it traverses all pointers pointed to allocated areas and marks the memory areas which are in use. The 2nd pass finds all unmarked areas and declares them to be free. Garbage collection overheads are not incremental, they are incurred everytime the system runs out of memory to allocate to fresh requests.

- Reuse of Memory
① First Fit
② Best Fit

MACRO PROCESSORS

- Reasons of using MACROs.
  1. Modularity
  2. Readability
  3. Maintainability

A macro is a unit of specification for program generation through expansion.
A macro consists of a name, a set of formal parameters and a body code.
The use of a macro name with a set of actual parameters is replaced by some code generated from its body.
This is called macro expansion.

1. Lexical expansion -
   Replacement of a character string by another character string during program generation. Lexical expansion is typically employed to replace occurrences of formal parameters by corresponding actual parameters.

2. Semantic expansion
   Generation of instructions tailored to the requirements of a specific usage.
   Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in number, sequence and opcodes of instructions.

Page No.
Date

- Macro Definition

A macro definition is enclosed between a MACRO HEADER and a MACRO END

MACRO definition consists:-

① Macro Prototype statement
② One or more model statements
③ Macro preprocessor statements

① Macro prototype statement declares the name of a macro and the names and kinds of its parameters.
② A model statement is a statement from which an assembly language statement may be generated during macro expansion.
③ A preprocessor statement is used to perform auxilary functions during macro expansion.

- Macro Call

A macro is called by writing the macro name in the mnemonic field of an assembly statement.

<macro_name> [<actual parameter spec>[,....]]

parameters of default kind: positional parameters

- MACRO
  INCR &MEM_VAL, &INCR_VAL, &REG
  MOVER &REG, &MEM_VAL
  ADD &REG, &INCR_VAL
  MOVEM &REG, &MEM_VAL
  MEND

Model statements

To differentiate between original statements of program and the statements resulting from macro expansion, each expanded statements is marked with '+' preceding its label field.
Two key notation of macro expansion are:

① Expansion time control flow-
This determines the order in which model statements are visited during macro expansion.

② Lexical substitution-
lexical substitution is used to generate an assembly statement from a model statement.

- Flow of control during expansion
  ① Default flow is sequential.
  ② A preprocessor statement can alter flow of control during expansion such that some model statements are either never visited during expansion, or are repeatedly visited during expansion.

The flow of control during macro expansion is implemented using a macro expansion counter [MEC].

① MEC := statement num. of 1st stmt following the prototype statement.
② While stmt pointed by MEC is not a MEND stmt
  ⓐ If a model stmt then
    i) Expand stmt
    ii) MEC := MEC+1
  ⓑ Else [i.e. Preprocessor stmt]
    i) MEC := New value specified in stmt.
③ Exit from macro expansion

- Lexical Substitution

A model statement consists of 3 types of strings

① An ordinary string, which stands for itself.

② The name of a formal parameter which is preceded by character '&'

③ Name of preprocessor variable, also preceded by '&'

During lexical expansion, strings of Type1 are retained without substitution. Strings of Type 2 and 3 are replaced by 'values' of the formal parameters or preprocessor variables.

- Positional Parameters

&<parameter-name>

```
INCR  A, B, AREG
```

Foll. the rule of positional association, values of formal params. are:-

formal param.	value
MEM_VAL	A
INCR_VAL	B
REG	AREG

lexical expansion of model stmts. leads to code:-

```
+   MOVER    AREG, A
+   ADD      AREG, B
+   MOVEM    AREG, A
```

- Value of positional formal param. XYZ is determined by 'Positional assoc.'

① Find ordinal position of XYZ in list of formal parameters in the macro prototype stmt.

② Find actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call stmt.

- Keyword Parameters

<formal_parameter_name> = <ordinary string>

The value of Formal parameter XYZ is determined by rule of keyword association as follows:-

① Find actual param. specification which has the form XYZ = <ordinary_string>

② Let ordinary string be ABC, then the value of XYZ is ABC.

- Default specifications of parameters

& <parameter name> [<parameter kind>]
[<default value>]

- Nested Macro Calls

LIFO Rule.

COMPUTE X,Y

+ MOVEM BREG,TMP (1)

+ INCR_D X,Y
- (2) + MOVER BREG,X
- (3) + ADD BREG,Y
- (4) + MOVEM BREG,X

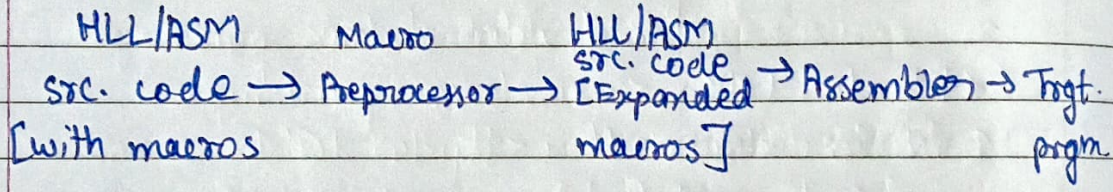
+ MOVER BREG,TMP (5)

- **Macro-Preprocessor**

Macro is a concept/theme/abstractions can be conceived as a shortcut way of doing regular but long and hectic activity.

A macro is abbreviation for a group of instructions. Since macro is a concept, it is not proprietary for any language.

The macropreprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions or calls



- **List of tasks involved in macro expansion**

① Identify macro calls in prgm. } MNT table is used to check if stmt is macro call.
② Determine values of formal parameters
③ Maintain values of expansion time variables declared in a macro.
④ Organize expansion time control flow.
⑤ Determine values of sequencing symbols.
⑥ Perform expansion of a model statement.

Page No.
Date

- Macro Structure

```
                    → pseudo code
1   MACRO     xyz   param1, param2, --- paramn   } Macro Header
2   -
3   -
|   -
|   -
8   MEND      (xyz) → optional                   } Macro Footer
|   |
21  xy        1, 2, ---n                         } Macro call.
                    params
```

Tasks:- IDMODP

(1) Identify macro call

Macro name Table [MNT]

A macro name is entered in this table when a macro def. is processed. While processing a statement in src. prgm. the preprocessor compares the string found in its Mnemonic field with macro names in MNT. If match, then its macro call.

(2) Determine values of formal parameters.

Actual Parameter [APT] is designed to hold the values of formal parameters during the expansion of a macro call.

(<formal param name>, <value>)

Two items of information are needed to construct this table, names of formal param. and default values of keyword params.

Parameter Default Table [PDT] is used for each macro

(<formal param name>, <default value>)

If macro call statement does not specify a value for some param, it's default value would be copied from PDT to APT.

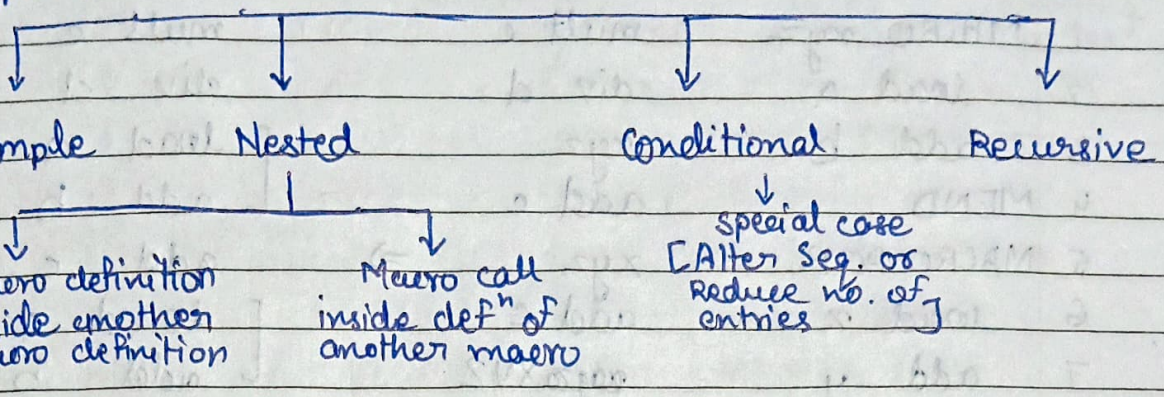
③ Maintain expansion time variable
Expansion Time variables Table [EVT]
(<EV name>, <value>)

④ Organize expansion time control flow
Body of a macro is to be stored in a table called Macro Definition Table [MDT]
The flow of control during macro expansion determines when a model statement is to be visited for expansion.
MEC [Macro Expension Counter] is used.

⑤ Determine values of sequencing symbols
Sequencing Symbols Table [SST]
(<sequencing symbol name>, <MDT entry #>)
where <MDT entry #> is the number of the MDT entry which contains model statement defining seq. symbol. This entry is made on encountering a statement seq. symbol in its label field [back ref.] or on encountering a ref. prior to its definition [forward ref.]

⑥ Perform expansion of a model statement
- MEC points to MDT entry containing model stmt.
- values of formal parameters and EV's are available in APT and EVT
- Model stmt defining a seq. symbol can be identified from SST.

- Types of macros



Simple → Either list of parameters or empty
Nested → —u—
Conditiond → Must be list of parameters

- Simple Macro

```
1   MACRO xyz
2   load  a
3   add   b
4   store c
5   MEND
6   —
7   —
8   —
9   xyz
10  —
11  xyz
```

⟹ O/P

```
6
7
8
9   load  a
10  add   b
11  store c
12  —
13  load  a
14  add   b
15  store c
```

```
1  MACRO xyz
2    load a
3    add b
4  MEND
5  MACRO PQR
6    load x
7    add y
8  MEND
9  MACRO ABC
10   load p
11   add q
12 MEND
```

```
mult c
div d
xyz
add c
xyz
add c
PQR
Store m
PQR
END
```

⇒

```
mult c
div d
load a
add b
add c
load a
add b
add c
load a
add b
load p
add q
store m
load x
add y
END
```

① MNT ← MNTPTR

Name of macro	No. of param	Start index	End index
xyz	0	1	2
PQR	0	3	4
ABC	0	5	6

② MDT ← MDTPTR

1	load a
2	add b
3	load x
4	add y
5	load p
6	add q

```
macro xyz (&a, &b) → formal param
  load &a
  load &b
MEND
load m
XYZ 3,4 → Actual param
store b
END
```

MNT

Name	No. of param	Starting index
xyz	2	01

MDT

1	load &a
2	load &b
3	MEND

O/P

```
load m
load 3
load 4
store b
END
```

Data Structure

formal vs positional

param	Position
&a	#1
&b	#2

actual vs Position

3	#1
4	#2

After use delete these tables

MDT ⇒

```
load #1
load #2
MEND
```

```
1  MACRO xyz (&a, &b)
2    load &a
3    add &b
4  MEND
5  MACRO ABC
6    load p
7    add q
8  MEND
9  MAIRO xyz(&a,&b,&c)
10   load x
11   add &a
12   sub &c
13   mult y
14   store &b
15 MEND
16 load m
17 ABC
18 XY2 3,4
19 xyz 3,4,5
20 END
```

MNT

xyz	2	1
ABC	0	4
xyz	3	7

① MDT

```
1  load &a => load #1
2  add &b => add #2
3  MEND
4  load p
5  load q
6  MEND
7  load x
8  add &a
9  sub &c
10 mult y
11 store &c
12 MEND
```

Formal vs pos

&a	#1
&b	#2

actual vs pos

XY2	3	#1
	4	#2
XYZ	3	#1
	4	#2
	5	#3

O/P File

```
load m
load p ]
add q  ]
load 3 ]
add 4  ]
load x ]
add 3  |
sub 5  |
mul y  |
store 4]
END
```

d=
0
1
0
1

e=
0 → Normal code line
0 → Macro defn
1 → Macro expansion
1 → Recursive macro

Page No.
Date

```
1  Load a
2  store b
3  MACRO ADD1 &arg
4  Load 1, &arg
5  Add 1, F'1'
6  Store 1, &arg
7  MEND
8  MACRO ADD5 &A1 &A2
9  Load c
10 store d
11 ADD1  5
12 ADD1  &A1
13 Load  &A2
14 load  &A3
15 MEND
16 ADD5 d1 d2 d3
17 ENDP
```

MNT

Name	No. of params	Start
ADD1	1	3
ADD5	2	5

MDT

```
1  Load 1, &arg → load 1, #1
2  Add 1, F'1'
3  Store 1, &arg → store 1, #1
4  MEND
5  load c
6  store d
7  ADD1 5
8  ADD1 &A1 → ADD1 #1
9  load &A2 → load #2
10 load &A3 → load #3
11 MEND
12
```

Page No.
Date

- Conditional Macro

① we must pass different parameter value as a part of macro call.

② Macro body must contain the segment which will come out with diff. seq. dependencies on the value being passed in macro call.

Instead of AIF we use AIF

AIF (condition) . label

AGO . label

- macro vary &arg0, &count, &arg1, &arg2, &arg3
  &arg0 : A1 &arg1
  AIF (&count == 1) . fini
  A2 &arg2
  AIF (&count EQ 2) . fini
  A3 &arg3

fini: AGO .over
over: MEND

vary loop1, 3, d1, d2, d3
loop1: A1 d1
A2 d2
A3 d3

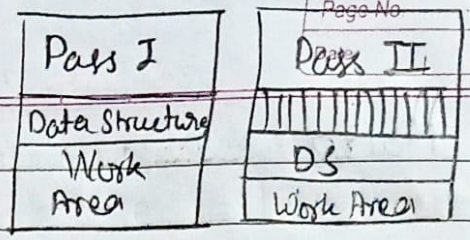
vary loop2, d1, d2
loop2 : A1 d1
A2 d2

ANOP

	Macro	Function
①	Macros are preprocessed	Functions are compiled.
②	No type checking is done.	Type checking is done.
③	Using macro increases code length	Using function keeps code lenght unaffected.
④	Lead to side effect later.	No side effects
⑤	Speed of execution is faster	Speed of execution is slower
⑥	Before compilation macro name is replaced with macro value.	During function call, transfer of control takes place.
⑦	Macro is important when small code is repeated many times	Functions are useful when large code is to be written.
⑧	Macro does not check any compile time errors.	Functions check compile time errors.

ASSEMBLER

Variant I



* Intermediate Code Forms

Two criteria: ① Processing Efficiency
② Memory Economy

Intermediate code consists of a set of IC units, each IC unit consisting of following three fields:-

Address	Opcode	Operands

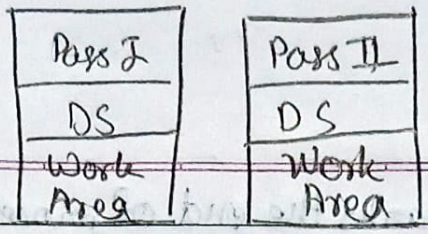
Mnemonic opcode field
(stmt class, code)
↓ ↓
IS, DL, AD : For IL, code is instruction opcode
For DL, AD:- it is ordinal number within class.

START ⇒ (AD,01)

- Variant I

1st operand is represented by single digit which is code of register (1-4) [AREG-DREG] or condition code itself [1-6 for LT-ANY].
2nd operand is a memory operand
(operand class, code)
operand class is one of Constant, Symbol or Literal.
For constant, code field contains internal repr. of const. itself.
For symbol and literal, code contains ordinal number of operand's entry in SYMTAB or LITTAB.
We assumed that entry in SYMTAB is already made but while processing a forward reference ——

Variant II



overall memory requirement of assembler is lower.

MOVER AREG, A

At this point, address and lenght fields of A's entry cannot be determined. This implies that two kinds of entries may exist in SYMTAB at any time. for defined and for forward references.

```
        START   200         (AD,01)   (C,200)
        READ    A           (IS,09)   (S,01)
LOOP    MOVER   AREG,A      (IS,04)   1(S,01)
        |
        LTORG               (DL,05)
```

- Variant II

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

```
        START   200         (AD,01)   (C,200)
        READ    A           (IS,09)   A
LOOP    MOVER   AREG,A      (IS,04)   AREG,A
```

- Differences

Variant I requires extra work at pass I as operand fields are completely processed. This processing simplies tasks of Pass II. IC is compact.

Variant II reduces work of Pass I passing burden on Pass II IC is less compact, since memory operand of typical imperative statement is in the source form itself. Pass II performs more work, funcs and memory requirements of two pass get better balanced.

Page No.
Date

- ENDP → Directive marks the end of procedure.
  END → Main directive marks the last time of program to be assembled.

- END [End of Program]
  Marks end of assembly language program. When assembler comes across END directive, it avoids source lines available later on. So after END statement no useful program statement should lie in the file

- ENDP [End of Procedure]
  In ASL, subroutines are called procedures.

```
Procedure P1
    :
P1 ENDP
```

- EQU : Equate
  It is used to assign a label with a value of symbol

```
LABEL EQU 5000H
ADDITION EQU ADD
```

- EXTRN
  Informs the assembler that the names, procedures and labels declared after this directive have been already defined in some other AL Modules. While in other module, where names, procedures and labels are actually declared, they must be declared public using the PUBLIC directive.

Page No.
Date

```
MODULE1 SEGMENT
PUBLIC FACT FAR
MODULE1 ENDS

MODULE2 SEGMENT
EXTRN FACT FAR
MODULE2 END
```

- LTORG
  It allows programmers to specify where literal should be placed in memory.

<, ≤, =, >, ≥, ANY
1 2 3 4 5 6

DC=1 DS=2

```
        START 500
        MOVER AREG ='25'
        SUB   AREG ,K
JMP1    MULT  AREG ,='81'
        ORIGIN 300
        LTORG
        ORIGIN 503
LOOP1   MOVER CREG, COUNT
        ADD   CREG, ='1'
        MOVEM CREG, COUNT
        ADD   AREG, COUNT
        SUMB  AREG, ='7'
        MOVEM AREG, SUM1
        BC    ANY, LOOP
        PRINT SUM
        STOP
A       DS    1
K       DS    1
COUNT   DS    1
SUM     DS    1
        END
```

Variant - 7

LC	(AD,01) (C,200)
500	(IS,04) (1) (L,01)
501	(IS,02) (1) (S,01)
502	(IS,03) (1) (L,02)
-	(AD,05)
300	(AD,03)
	(AD,05)
503	(IS,04) (3) (S,04)
504	(IS,01) (3) (L,03)
505	(IS,05) (3) (S,04)
506	(IS,01) (1) (S,04)
507	(IS,02) (1) (L,4)
508	(IS,05) (1) (S,5)
509	(IS,07) (6) (S,3)
510	(IS,10) (S,05)
511	(IS,00)
512	(DL,02) (C,1)
513	(DL,02) (C,1)
514	(DL,02) (C,1)
515	(DL,02) (C,1)
-	(AD,02)

SYMTAB

	Symbol	Add.	Length
①	K	513	1
②	JMP1	502	1
③	LOOP	503	1
④	COUNT	514	1
⑤	SUM	515	1
⑥	A	512	1

LITTAB

	Literal	Address	
①	='25'	300	1st pool
②	='81'	301	
③	='1'	516	2nd pool
④	='7'	517	

POOLTAB

Lit No
#1
#3

Page No.
Date

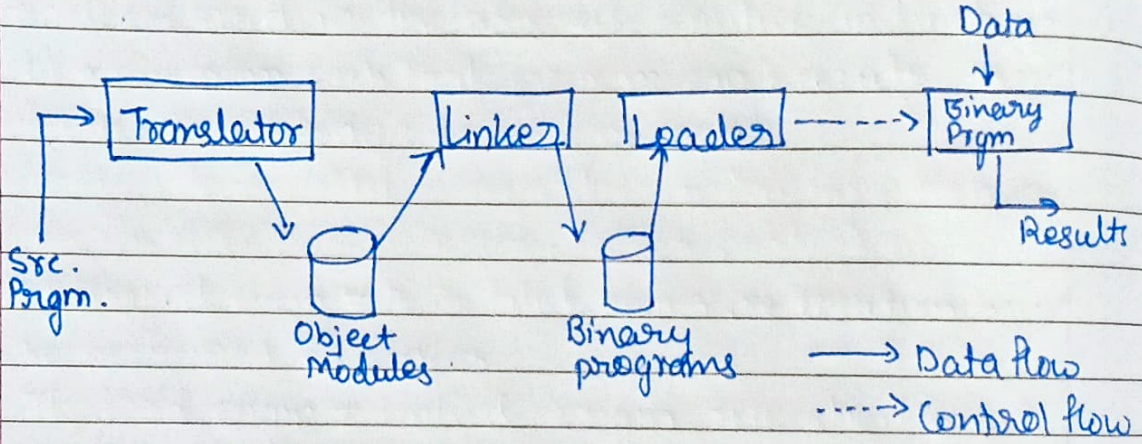
	Variant-I		Variant-II	
	LC		LC	
START 101	101	(AD,01) (C,101)	101	(AD,01) (C,01)
READ A	102	(IS,09) (S,01)	102	(IS,09) A
READ B	103	(IS,09) (S,02)	103	(IS,09) B
MOVER BREG,A	104	(IS,04) (1) (S,01)	104	(IS,04) BREG,A
MULT BREG,B	105	(IS,03) (2) (S,02)	105	(IS,03) BREG,B
MOVEM BREG,D	106	(IS,005) (2) (S,03)	106	(IS,05) BREG,D
STOP	107	(IS,00)	107	(IS,00)
A DS 1	108	(DL,01) (C,1)	108	(DL,01) (C,1)
B DS 1	109	(DL,02) (C,1)	109	(DL,02) (C,1)
D DS 1	110	(DL,03) (C,1)	110	(DL,03) (C,1)
END	111	(AD,02)	111	(AD,02)

SYMTAB

	Symbol	Type	Add
①	A	Id	108
②	B	Id	109
③	D	Id	110

LINKERS and LOADERS

- Execution of Program
  1. Translation of program
  2. Linking of program with other programs needed for its execution.
  3. Relocation of program to execute from specific memory area allocated to it.
  4. Loading of program ~~to execute from specific memory area~~ in memory for purpose of Execution.



- Translation time (or translated) address - Address assigned by translator
  Translated origin - Address of origin assumed by Translator. This is the address specified by programmer in ORIGIN statement.

- Linked address - Address assigned by linker
  Linker origin - Address of origin assigned by linker while producing a binary program.

- Load time address - Address assigned by loader.
  Load origin - Address of origin assigned by the loader while loading program for execution.

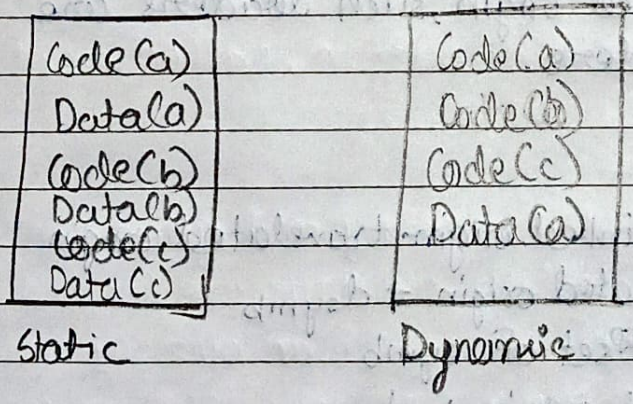
Page No.
Date

Binding

A binding is a association of an attribute of a program with a value. Each program entity has a set of variables associated with it.
Binding time is a time at which binding of an attribute with a value is performed.

- **Static Binding**
  Binding performed before execution of program begins.
  Static memory allocation, memory is allocated to variable before program execution. It is typically performed during compilation. No memory allocation and deallocation is performed during program execution.

- **Dynamic Binding**
  Performed during execution of program.
  Dynamic memory allocation, memory bindings are established and destroyed during execution of program.



Page No.
Date

Relocation and Linking Concepts.

- **Program Relocation**

If instructions in a program refer to absolute address i.e. the program assumes it's instruction and data to occupy memory words with specific address, such programs are called address sensitive programs.

- An address sensitive program can execute correctly only if the start address allocated in memory to it same as its translated address.
- Program relocation is process of modifying the address used in the address sensitive instructions of a program such that program can execute correctly from designated area and memory.

If linked origin ≠ translated origin, relocation must be performed by linker.
If load origin ≠ linked origin, relocation must be performed by loaders.
If load origin = linked origin, such loaders are called absolute loaders.

relocation factor = linked origin - translated origin

t_{symb} = translated origin + d_{symb}

d_{symb} = offset of symb

l_{symb} = linked origin + d_{symb}

∴ relocation factor = t_{origin} + relocation factor + d_{symb}

= t_{origin} + d_{symb} + relocation factor

= t_{symb} + relocation factor.

Page No.
Date

ENTRY and EXTRN statement

- ENTRY statement lists the public definitions of a program unit, i.e. it lists those symbols defined in the program unit which may be referenced in other program units.
- EXTRN statement lists the symbols to which external references are made in program unit. Assembler does not know the address of external symbols. Hence it puts zeroes in the address fields of instructions corresponding to statement. If EXTRN does not exists, assembler would have flagged the refs as error.

- Linking is the process of binding an external reference to correct link time address.

Binary Program

A binary program is a machine language program comprising a set of program units SP such that

$$\forall P_i \in SP$$

P_i has been relocated to memory area starting at its link origin.
Linking has been performed for each ~~external~~ external reference in P_i.

linker <link origin>, <object module names>.
[, <execution start address>]

Linker converts object modules in set of program units SP into binary program

Page No.
Date

- Object Module.

4 components :-

① Header - Translated origin, size, execution start addr.
② Program - Machine language program
③ Relocation Table - Describes IARp. Each RELOCTAB entry contains a single field.
④ Linking Table [LINKTAB] - Information concerning the public definitions and external references in P. (symbol, type, translated address) (PD/EXT)

- Linking for Overlay.

Overlay - An overlay is a part of program which as same load origin as some other part(s) of program. Overlays are used to reduce main memory requirement of a program.

Overlay structured program.

Consists :-

① A permanently resident portion, called root.
② A set of overlays.

Start by, root is loaded in memory and given control for purpose of execution. Other overlays are loaded as and when needed. The loading of an overlay overwrites a previously loaded overlay with same load origin. The overlay structure of program is designed by identifying mutually exclusive modules - i.e. they do not call each other.

The overlay structure of object program is specified in linker command.

Page No.
Date

- LINKER uses two pass strategy. In first pass, the object modules are processed to collect information concerning segments and public definitions. The 2nd pass performs relocation and linking.

- (1) Direct Linking Loader
  (2) Dynamic Linking Loader
  (3) Absolute Loader

A relocating loader performs relocation while loading a program for execution. This permits a program to be executed in different parts of memory.

- MS-DOS
  Support 2 object program forms.
  (1) .COM → non relocatable object prgm.
  → Absolute loader is involved to simply load object program into memory.
  (2) .EXE → relocatable object program.
  → Relocating loader relocates the prgm. to designated load area before passing control to it for execution.

Program EXE2BIN ⇒ .EXE → .COM

Page No.
Date

- Types of Errors
① Syntax Errors
② Semantic Errors
③ Linker Error
④ Runtine Error
⑤ Compilation Error.

- Types of Debuggers
Categorized into two main types based on User Interface provided during debugging process.
① Console based debugger
Executes in Text Mode.
GDB used for C++, C.
Purpose is to allow you to see what is going on "inside" another program while it executes or what another program doing at moment it crashed.
- Specify that might affect your program
- Stop on particular condition
- Examine what has happened, when your prgm stopped
- Experiment to correct a bug and move on another.

② Visual Debugger
Great for displaying real life objects.
Visual data.
Most implementations of visual debugging requires seperate application program running on specific hw or development platforms. They are generally language specific.
DDD [Data Display Debugger] [Inferior debugger]
- ddd + gdb for Java bytecode programs
- ddd + pydb for Python
Support Front End for Command Line Debugger.

Page No.
Date

Types of Editors.

1. Line Editor - One line at a time. You cannot have a free-flowing sequence of characters. E.g. Teleprinter.

2. Stream Editor - File is treated as contiguous flow of sequence of characters instead of line numbers.
Eg. Sed Editor on UNIX.

3. Screen Editor - User is able to use and see the cursor on screen and can copy, paste, cut operations easily. It is very easy to use mouse cursor.
Eg. Notepad, vi, emacs.

4. Word Processor - Overcome limitations of screen editor by allowing one to use some format to insert images, files, videos, etc. features. Majorly focuses on Natural Language. Ex. OpenOffice Writer.

5. Structure Editor - Focuses on Programming languages. It provides features to write and edit source code.
Eg. NetBeans IDE, gEdit.

- YACC

Yet Another Compiler Compiler.
Used for compiler construction and parsing.
YACC assists in generation of parsers for programming languages or other formal grammars.

① Parser Generation
② Parsing Technique
LALR [Look-Ahead, ~~look~~ Left-to-Right, Rightmost derivation] parsing. Bottom-Up Parsing.
constructs ~~the~~ parse tree for input text starting from the leaves (tokens) and working up towards the root.
③ Rule based specification-
set of production rules
④ Error Handling
⑤ Integration with Lex/Flex
Generates lexical analyzers for tokenizing input text
lexical analyzers produced by Flex/Lex tokens
to parse generator by YACC, ^Feed
allowing complete parsing of input Text.

Page No.
Date

- LEX

Generates lexical analyzer. Used with YACC parser generator.

Transforms an input stream into a sequence of tokens.

lex s&c ⟶ Lex Compiler ⟶ lex.yy.c

lex.yy.c ⟶ C compiler ⟶ a.out

input stream ⟶ a.out ⟶ output stream [Seq. of Tokens]

{definitions}

%%

{Rules}

%%

{user subroutines}

```
%{
#include<stdio.h>
int i; int ne=0, no=0;
%}
%%
[0-9]+ { i=atoi(yytext);
        if(i%2)
            ne++;
        else
            no++;
       }
%%

int yywrap(){
    return 1;
}

int main(){
    yylex();
    return 0;
}
```

```
%{
#include<stdio.h>
int v=0, c=0;
%}

%%
[\t \n]+
[aeiouAEIOU] {v++;}
[^aeiouAEIOU] {c++;}
%%

int main(){
    printf("                    ");
    yylex();
    printf("%d", v);
    printf("%d", c);
}

int yywrap(){
  return 1;
}
```

```
lex file.l
gcc lex.yy.c
./a.out
```