UNIT 6 SYSTEM SOFTWARE AND ITS IMPORTANCE

System Software -

The collection of programs that manages & control the operations of computer's hardware. This includes OS, compilers, assemblers, loaders, linkers & other utilities that enable hardware & software to communicate effectively.
System software acts as a bridge bet^n physical components of computer & application software that performs specific tasks.

Need

↳ Hardware communication ↳ S/w provides UI ↳ Application support
↳ Resource management ↳ security features ↳ Error detection & handling

Components / Types of system software

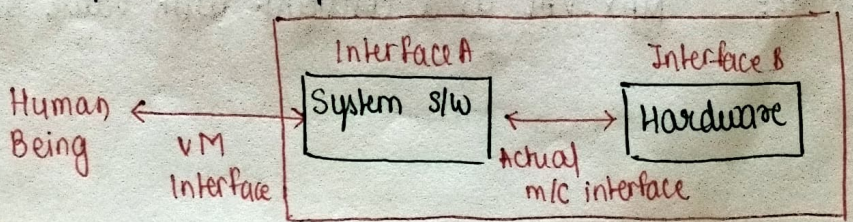
① OS
Manages hardware resources & provides services to run application software.
Eg. Windows, Linux, Os
features - file mangement, memory mainagement, process scheduling & UI

② BI/Os (Basic input output System)
Performs hardware initialization during boot-up & provides runtime services for the OS.
Acts as a firmware interface bet^n H/w & OS.

③ Device Drivers
Enables communication bet^n OS & H/w devices like printers, keyboards
Eg. Priter driver translates os instruction into commands that the printer understands.

④ Utility Programs
Perform maintainance task to optimize system performance.
Eg. Disk cleanup, antivirus software & system backup tools.
Enhances system efficiency & security.

⑤ Language translator
Convert programming lang code into machine language
Types - compiler Interpreter Assembler



Role of a S/w system.

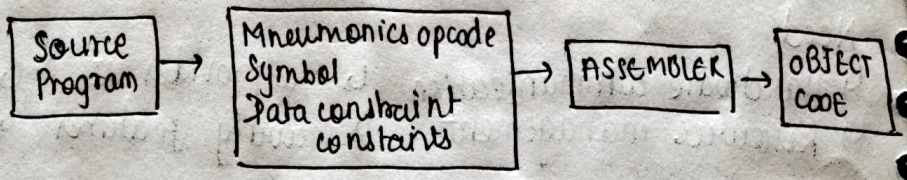
Assemblers

Assembly language programming involves using low level, human-readable instructions to communicate directly with computer's hardware.

Assembler translates assembly code into machine learning code that the CPU can execute.

Elements of Assembly Language Programming :—

Instruction Set
Registers
Memory Addressing



Assembly language consist of 3 types of Statements :—

① Imperative Statement

These are the instructions that tell the processor what action to perform. They correspond to operation executed by CPU, such as arithmetic operations, data movement or control flow.

Role - Define logic/flow of program

Eg : MOV AX, BX - moves data from BX reg. to AX
ADD AX, 5 - adds 5 to AX reg.

② Declarative Statement

These are non executable statements. They declare constants/storage areas in program.

Eg: X DS 2 - declare storage of 1 word & assign that to X.
NUM DB 10 - Allocates byte & initialize it to 10.

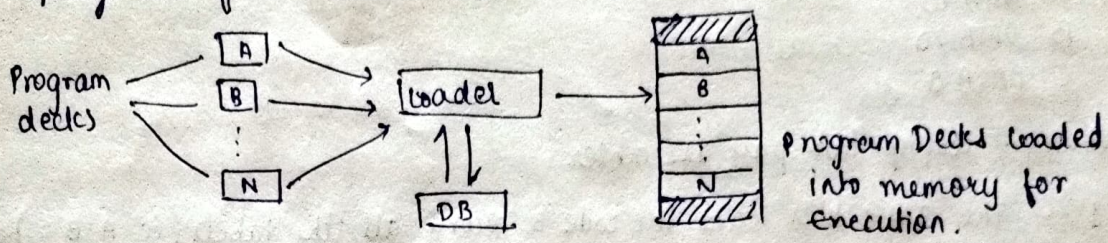
③ Assembly Directives

These are provided to assembler, not the processor. It instructs the assembler to perform certain action during assembly of a program.

Eg. ORG 100H - sets the code start address to 100H.
MAX_VAL EQU 255 - MAX_VAL is a constant with value of 255

Loaders

Loads the programs & libraries into memory, prepares them for execution. A loader is a program that accepts the object program and prepares these programs for execution by loading them into memory.



Program Decks loaded into memory for execution.

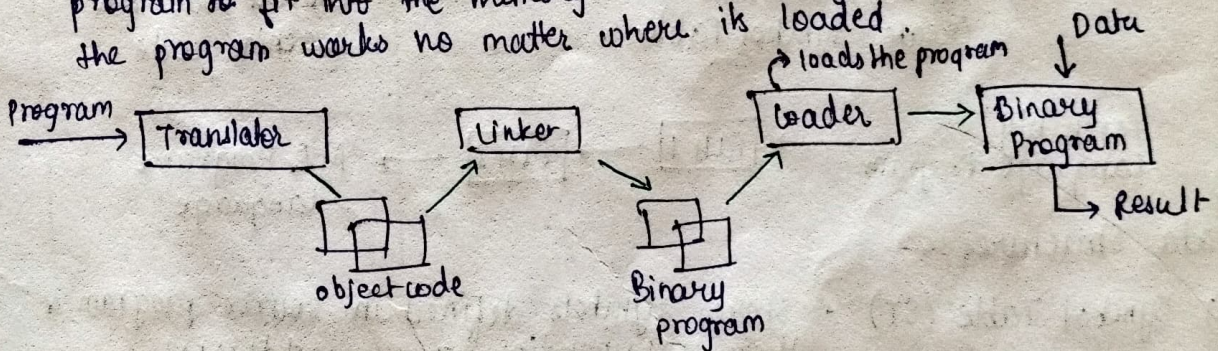
Functions :-

① Loading - load executable file into memory. This makes the program accessible to the computer for execution.

② Allocation - The loader figures out how much memory the program needs & reserves the appropriate amount of space in computer's memory. This ensures that the program has enough space to run without interference.

③ Linking - It links two/more separate object codes & provides the necessary info to allow reference bet them. This ensures the program can access everything it needs to run properly.

④ Relocation - If the program was written with specific memory address, but those addresses are already occupied, the loader adjusts the program to fit into the memory area it was given. This ensures the program works no matter where it's loaded.



Forward Reference Problem

When a program refers to variable, func or label before it has been defined, is called as forward reference.

In Assembly language, the code is processed sequentially. When the assembler encounters a reference to an identifier that has not been defined yet, it doesn't know where to find it / how to resolve it.

This is called as forward Reference Problem.

This problem is solved by making different passes (One pass, Two pass, Multi pass)

Eg: START:

```
    LOAD A ; Attempting to load value of A before it is defined
    ADD B
    STORE C
A: DATA 10
B: DATA 20
C: DATA 0
```

Solution : Using two-pass assembler

Pass 1 :- The assembler scans the code & records all the labels (ie A B C) along with their memory addresses in a symbol table.

Pass 2 :- The assembler revisits the code & replace the references (LOAD A) with correct memory addresses using the symbol table created in Pass 1.
This resolves the problem

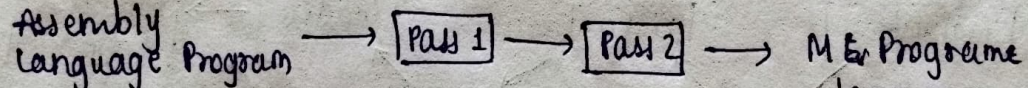
The FRP is common in low-level language programming & demonstrates why modern programming languages enforce declaration before usage.

Two Pass Assembler Design

Means more than 1 pass used by assembler. Eg IBM 360/370 processor

First pass → Scan the code.
Valide the tokens.
Create Symbol Table.

Second pass → solve FRP.
Convert code into machine code.



Data Structures :—

① Symbol Table (ST) - stores symbols defined in source program & their corresponding memory addresses

Symbol Name	Value	length	Relocation
"JOHN␣␣␣␣"	0000	01	"R"
"FIVE␣␣␣␣"	0010	04	"R"

- Implemented as hash table / directory for fast lookups
- key value pairs.

② Literal Table (LT) - stores literals (constants) used in program along with their memory locations..

Literal	Mem. addr
=10	2000
=20	2004

(MOT)

⑤ Machine Opcode Table - fixed length table ie contents of these tables are not filled in / altered during assembly process.

Mnemonic opcode	Binary op-code	Instruction length	Instruction format	Not used in this design
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
⋮			⋮	

④ Pseudo opcode Table (POT) - fixed length table

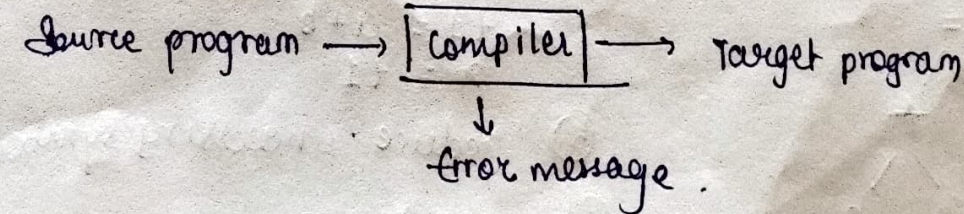
Pseudo opcode	Address of routine to process Pseudo - opcode
"DROPbb"	PIDROP
"ENDbb"	PIEND
⋮	

③ Base Table (BT) - used to generate the proper base register reference in machine ~~code~~ instructions & its correct offsets.

Available indicator	Designed relative-addr contents of base reg
"N"	—
"N"	—
⋮	⋮
"Y"	00 00 00

Compiler & Phases of Compiler

Compiler is a software program that translates high-level programming language/ code written in programming language into machine code that can be executed by computer's hardware.



Phases of Compiler

Analysis Phase

Breaks down the source code to understand its structure & meaning.
ie reads the code.

i/p - High level source code

eg - checks meaning of a+b*c

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

Synthesis Phase

Converts analysed code into machine language
. ie writes the code

i/p - Intermediate representation

eg - converts it into Machine Language

- Code optimization
- Code generation

① Lexical Analysis - breaks code into small parts called token.
Eg. int x = 5 ; token are : int | x | = | 5 | ;

② Syntax Analysis - checks if the arrangement of token follows the rules of programming language. (Do from TB pg 7-8)
Eg. int x = 5 ; is written correctly ?
* This is also known as Hierarchical / Parsing analysis.

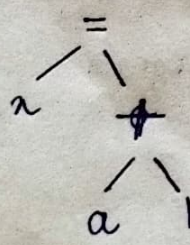
③ Semantic Analysis - ensures the code makes sense logically.
Eg. Checks if int x = "text" ; is valid (no because text is string)

④ code optimization - makes the code run faster / use less memory
Eg - Removes unecessary steps

⑤ code Generation - converts optimized code into machine instruction.

Eg; x = a + b;

① x | = | a | + | b

② 

③ check if x & b are integers

④ Intermediate code generation

t1 = a+b
x = t1

⑤ reduce unecessary symbol

⑥ LOAD a
ADD b
STORE x

Macro - is like a shortcut for writing code. Instead of typing the same code again & again, we use macro, to save time.

Macro call - invokes a macro ie when we need to use shortcut in program.

eg. `SQUARE(x) ((x) * (x))` ← this is our macro

`int result = SQUARE(5);` ← call

Macro Expansion - is what happen when shortcut gets turned into actual code.

So when you use `SQUARE(5)`, it changes to

`int result = ((5) * (5));`

Code simplification & reusability.