

MET's Institute of Engineering

Bhujbal Knowledge City, Adgaon, Nashik.

Department of Computer Engineering

“INTRODUCTION OF SP”

Prepared By

Prof. Anand N. Gharu

(Assistant Professor)

METIOE Computer Dept.

CLASS : TE COMPUTER 2019

SUBJECT : SPOS (SEM-I)

01 JULY 2021

UNIT : I

SYLLABUS :-

Introduction to Systems Programming, Need of Systems Programming, Software Hierarchy, Types of software: system software and application software, Machine structure.

Evolution of components of Systems Programming: Text Editors, Assembler, Macros, Compiler, Interpreter, Loader, Linker, Debugger, Device Drivers, Operating System.

Elements of Assembly Language Programming: Assembly Language statements, Benefits of Assembly Language, A simple Assembly scheme, Pass Structure of Assembler.

Design of two pass Assembler: Processing of declaration statements, Assembler Directives and imperative statements, Advanced Assembler Directives, Intermediate code forms, Pass I and Pass II of two pass Assembler.

CONTENTS :-

1. Introduction to Systems Programming

- Need of Systems Programming
- Software Hierarchy
- Types of software: system software and application software.

2. Evolution of components of Systems Programming:

- Text Editors
- Assembler
- Macros
- Compiler
- Interpreter

CONTENTS :-

2. Evolution of components of Systems Programming: -

- Loader
- Linker
- Debugger
- Device Driver
- Operating System.

3. Elements of Assembly Language Programming:

- Machine structure
- Assembly Language statements, Benefits of Assembly Language
- A simple Assembly scheme, Pass Structure of

CONTENTS :-

4. Design of two pass Assembler:

- Processing of declaration statements
- Assembler Directives and imperative statements
- Advanced Assembler Directives
- Intermediate code forms
- Pass I and Pass II of two pass Assembler.

CONTENTS :-

Sample videos :

1. Preprocessor :

<https://youtu.be/JZkPEl8JjZo?list=PLhb7SOmGNUc6Fg7zmBOOS3yN2AG0JiREp>

2. Compiler execution stages :

<https://youtu.be/cJDRShqtTbk?list=PLhb7SOmGNUc6Fg7zmBOOS3yN2AG0JiREp>

Introduction

Computer : A programmable device that can store, retrieve, and process data.(Combination of H/w & S/w)

Hardware : things which we can touch.

Software : things which we cannot touch.(Can only see)

Programming: A programming language is a set of commands, instructions, and other syntax use to create a software program.

Data : Information in a form a computer can use

Information : Any knowledge that can be communicated

Introduction

Computer program : Data type specifications and instructions for carrying out operations that are used by a computer to solve a problem.

Machine language : The language, made up of binary coded instructions, that is used directly by the computer

Assembly language : A low-level programming language in which a mnemonic is used to represent each of the machine language instructions for a particular computer

Introduction

Source code : Program or set of instructions written in a high-level programming language

Object code : A machine language version of source code.

Target code : program output in Machine code (binary form)

Introduction

Preprocessor :

a preprocessor is a program that processes its input data to produce output that is used as input to another program.

Editor : A text editor is a type of computer program that edits plain text. Such programs are sometimes known as "notepad" software

Compiler : A program that translates a program written in a high-level language into machine code

Assembler : A program that translates an assembly language program into machine code

-

Introduction

Loader:-

- A loader is a program used by an operating system to load programs from a secondary to main memory so as to be executed.

Linker :

a linker is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program

Debugger :

Debugger is program which is used to test program or execute program in single step execution.

Introduction

- **What is System?**
 - System is the collection of various components
- Ex:- College is a system.
- College is a system because it consist of various components like various departments, classrooms, faculties and students.
- **What is Programming?**
 - Art of designing and implementing the programs.

Introduction

- **What is System Programming?**
 - Systems programming involves the development of the individual pieces of software that allow the entire system to function as a single unit.
 - Systems programming involves many layers such as the operating system (OS), firmware, and the development environment.

Introduction

- In college system, what is **program**?
- A **LECTURE** can be a program. Because it has input and output.
- **Input**-> The information that teacher is delivering.
- **Output**-> The knowledge student has been received.

So **system programming** is an art of designing and implementing system Programs.

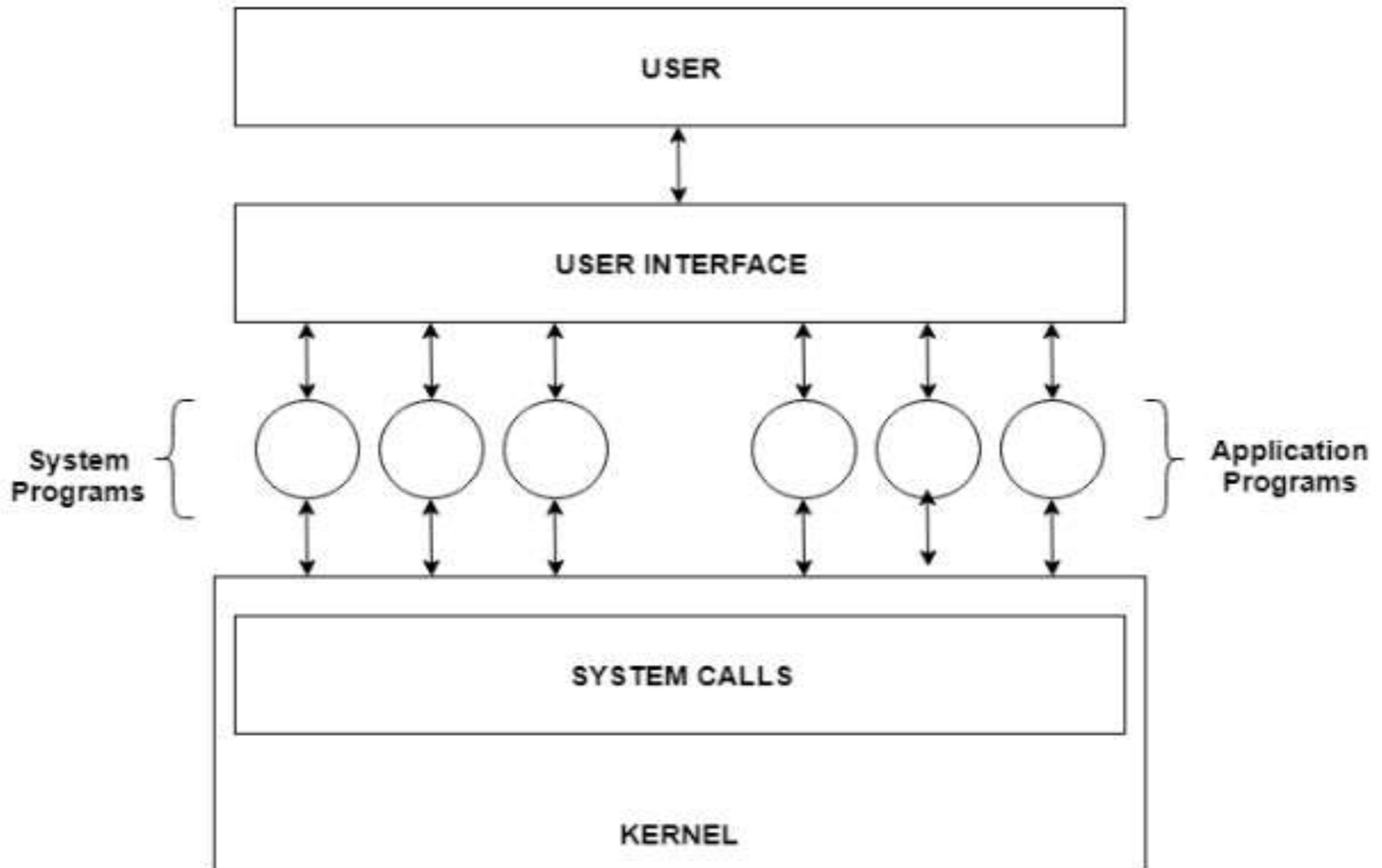
Need of System Programming

- System programs provide an environment where programs can be developed and executed.
- In the simplest sense, system programs also provide a bridge between the user interface and system calls.
- In reality, they are much more complex. For example, a compiler is a complex system program.

Need of System Programming

- The system program serves as a part of the operating system. It traditionally lies between the user interface and the system calls.
- The user view of the system is actually defined by system programs and not system calls because that is what they interact with and system programs are closer to the user interface.

Need of System Programming



Need of System Programming

- In the above image, system programs as well as application programs form a bridge between the user interface and the system calls. So, from the user view the operating system observed is actually the system programs and not the system calls

Need of System Programming

- To achieve effective performance of the system.
- To make effective execution of general user program.
- To make effective utilization of human resources.
- To make available new, better facilities.

Software Hierarchy

A software hierarchy is the combination of product, version, and release (or feature) that represents an item of software in a database or knowledge base. The product is the root of the hierarchy.

- **Software**

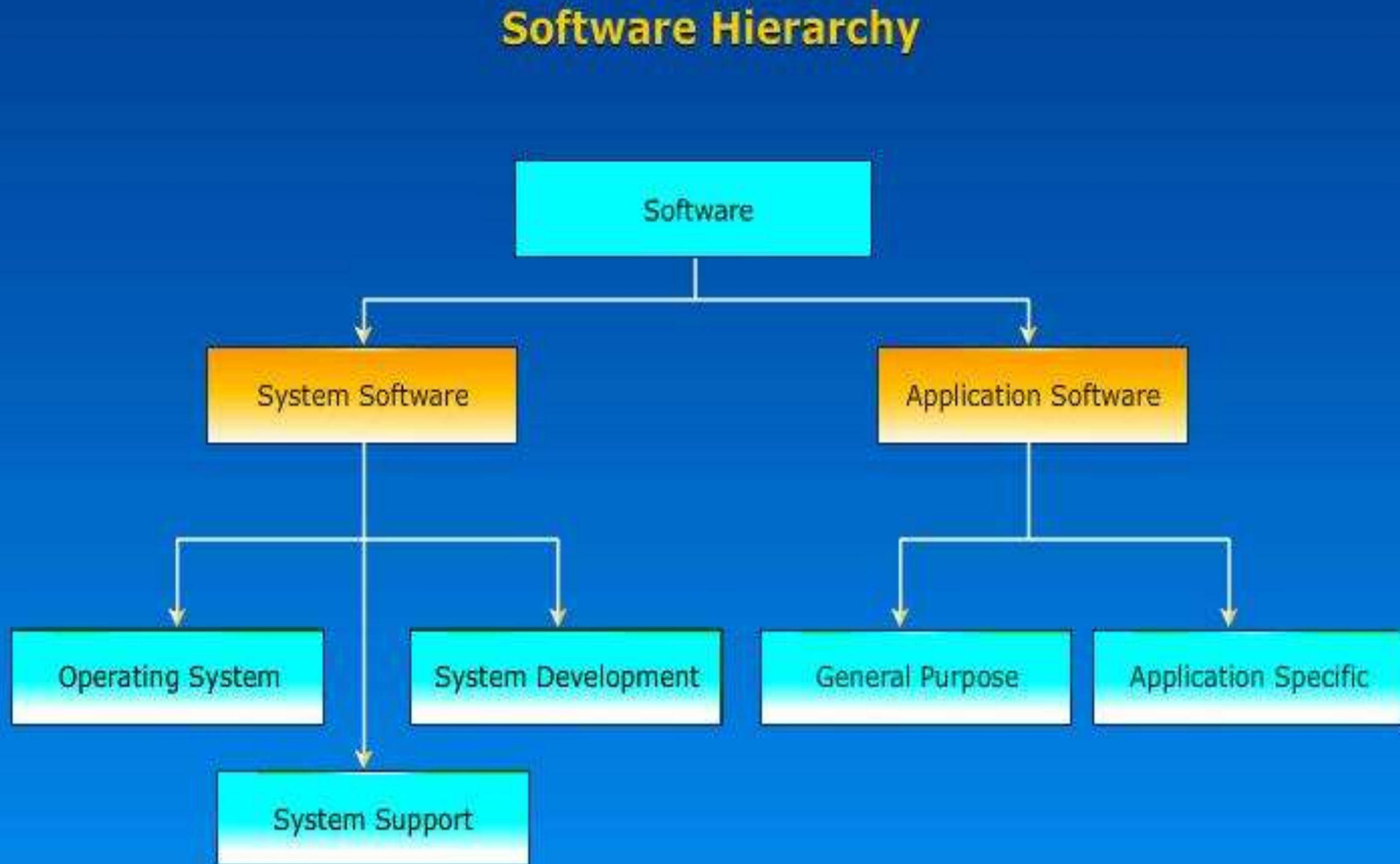
- 1. System Software**

- Operating System
- System Support
- System Development

- 2. Application Software**

- General Purpose
- Application Specific

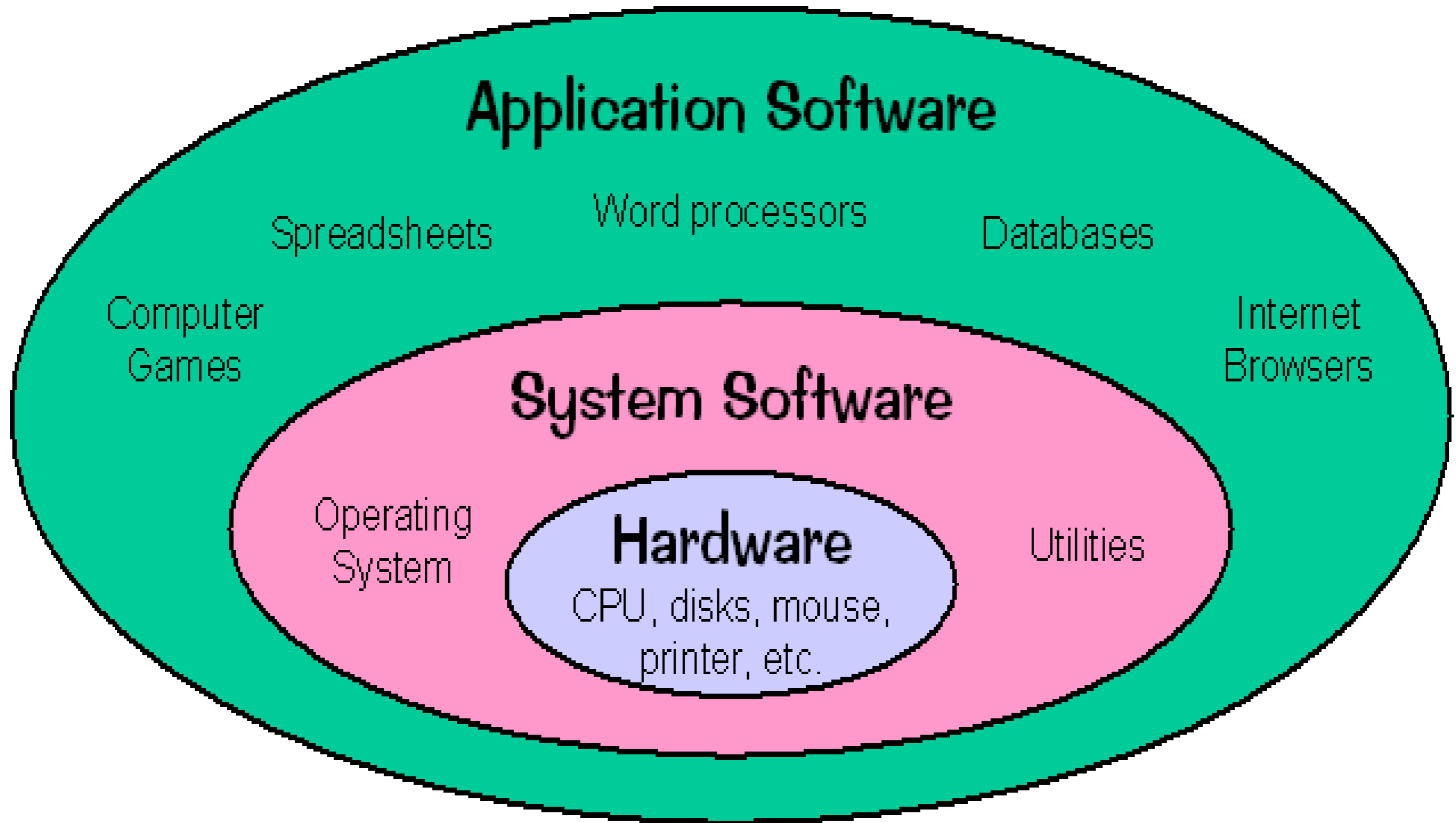
Software Hierarchy



What is Software ?

- Software is collection of many programs
- **Two types of software**
 - **System software**: These programs assist general user application programs
 - Ex:- Operating System , Assembler etc.
 - **Application software**
 - These are the software developed for the specific goal.
 - Ex. Media Player, Adobe Reader etc

System software Vs Application software



System software Vs Application software

Sr. No	System Software	Application Software
1.	System software is used for operating computer hardware.	Application software is used by user to perform specific task.
2.	System softwares are installed on the computer when operating system is installed.	Application softwares are installed according to user's requirements.
3.	In general, the user does not interact with system software because it works in the background.	In general, the user interacts with application softwares.
4.	System software can run independently. It provides platform for running application softwares.	Application software can't run independently. They can't run without the presence of system software.
5.	Some examples of system softwares are compiler, assembler, debugger, driver, etc.	Some examples of application softwares are word processor, web browser, media player, etc.

System software Vs Application software

Sr. No	System Software	Application Software
1.	It is general purpose software	It is specific purpose software
2.	Written in low level language	Written in High level language
3.	Small in size	Large in size
4.	Complex to design and implement	Easy to design and implement
5.	Some examples of system softwares are compiler, assembler, debugger, driver, etc.	Some examples of application softwares are word processor, web browser, media player, etc.

System Programming Components

- Text Editors
- Assembler
- Macros
- Compiler
- Interpreter
- Loader
- Linker
- Debugger
- Device Driver
- Operating System.

Text Editors

- Editor is a computer program that allows a user to create and revise a document..
- A **text editor** is a type of program used for editing plain text files.
- With the help of text editor you can write your program(e.g. C Program or Java Program).
- Text editor's example is Notepad.

Types of Text Editors

- 1. Line editor :** This code editor edits the file line by line. You cannot work on a stream of lines using the line editor. Example of a line editor is teleprinter.
- 2. Stream editor :** In this type of editors, the file is treated as continuous flow or sequence of characters instead of line numbers, which means here you can type paragraphs.

Ex : Sed editor in UNIX

Types of Text Editors

3. Screen editors : In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer.

Ex : vi, emacs, Notepad

4. Word Processor : Overcoming the limitations of screen editors, it allows one to use some format to insert images, files, videos, use font, size, style features. It majorly focuses on Natural language.

Loaders

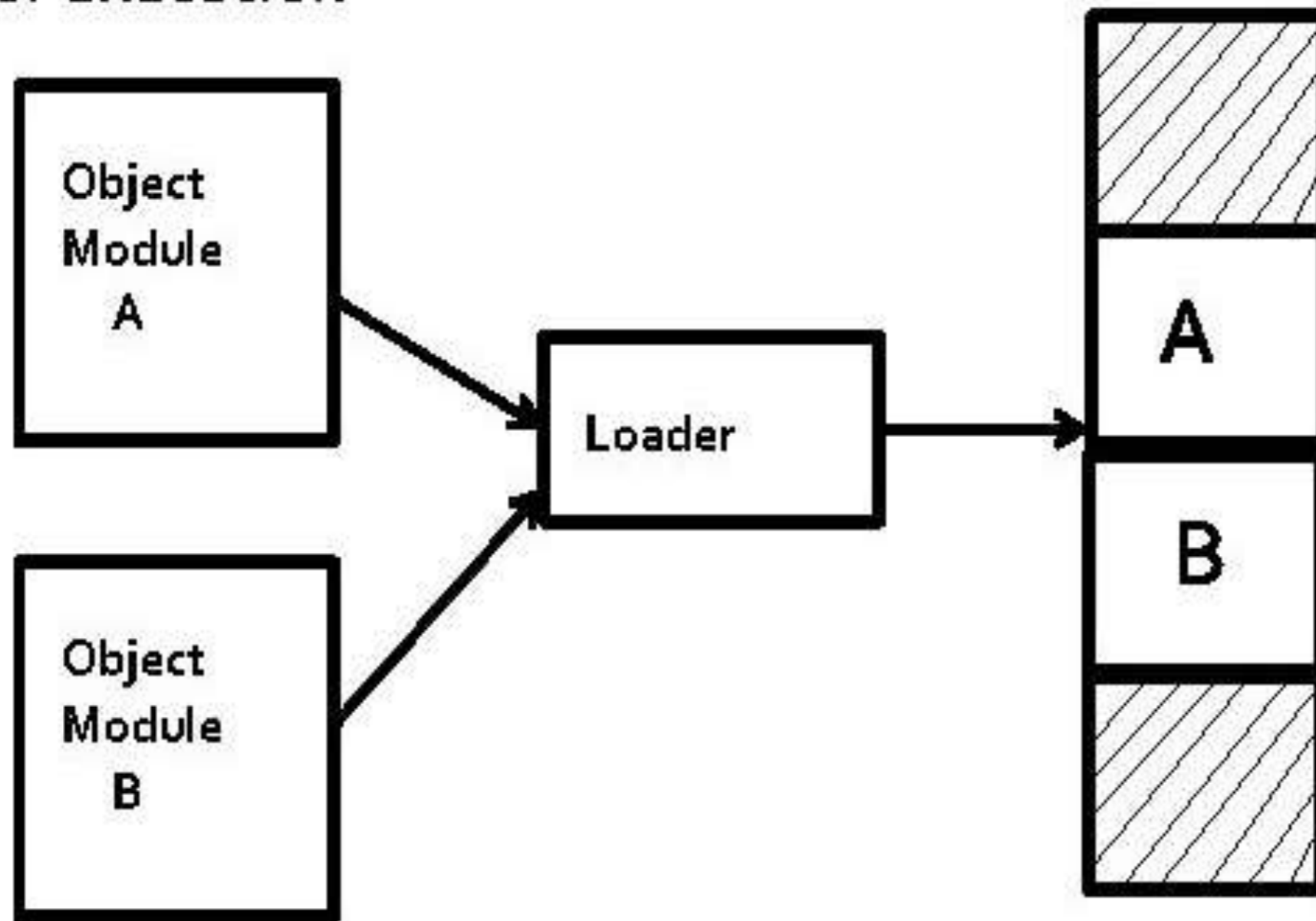
- A loader is a program that takes object code as input and prepares them for execution.
- It initiates the execution.
- **Functions:**
 1. Allocation
 2. Linking
 3. Relocation
 4. Loading

Functions of Loader

- 1. Allocation :** Loader allocates space for programs in main memory
- 2. Linking-** which combines two or more separate object programs (by Linker)
- 3. Relocation -** which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader) and data into main memory
- 4. Loading -** which allocates memory location and brings the object program into memory for execution - (Loader)

GENERAL LOADING SCHEME

Program modules A and B are loaded in memory after linking. It is ready for execution



RELOCATION

Case I: Address assigned to Prog. A and F1 () when they translated to memory

Drawback- a lot of storage area is wasted

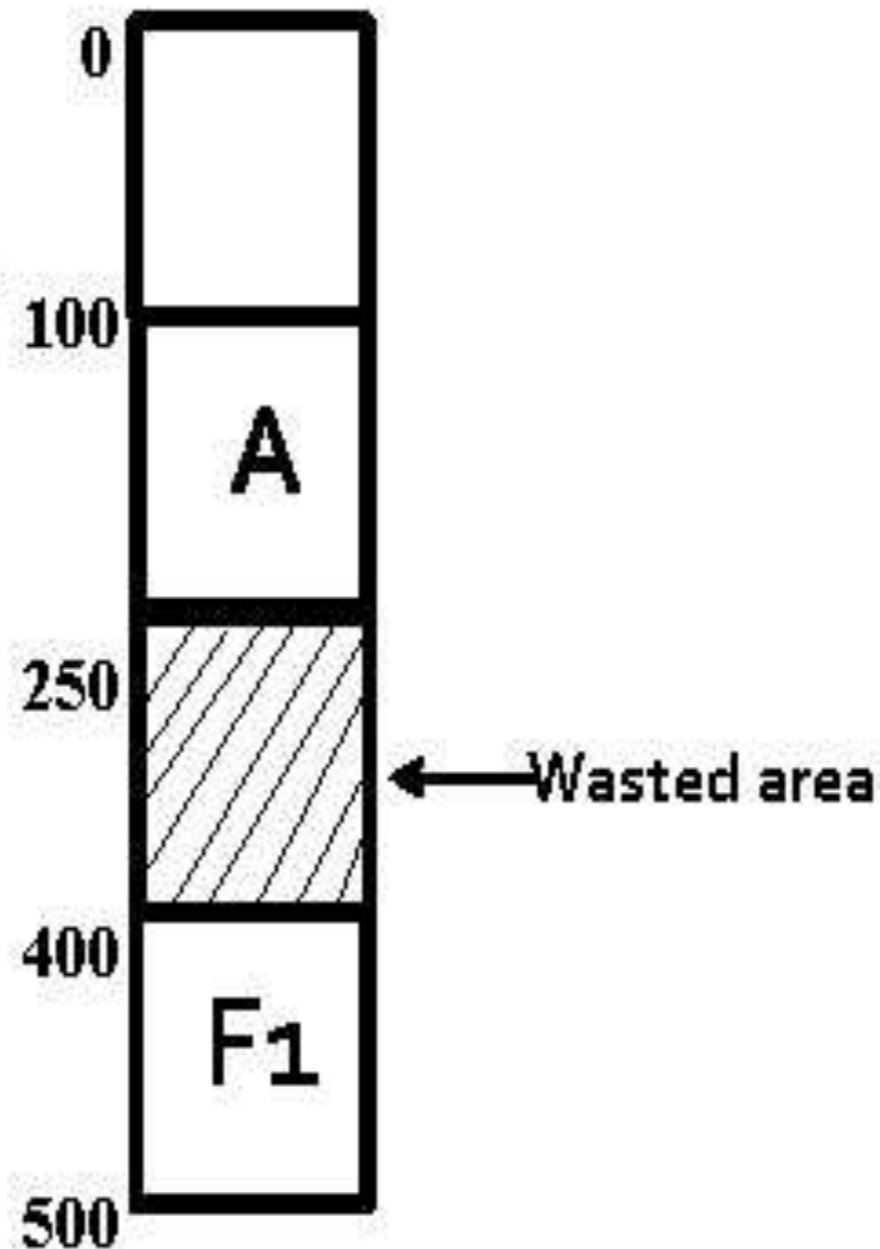


Fig.: General Loading scheme

RELOCATION

Case II: These two module cannot co-exist at same storage locations.

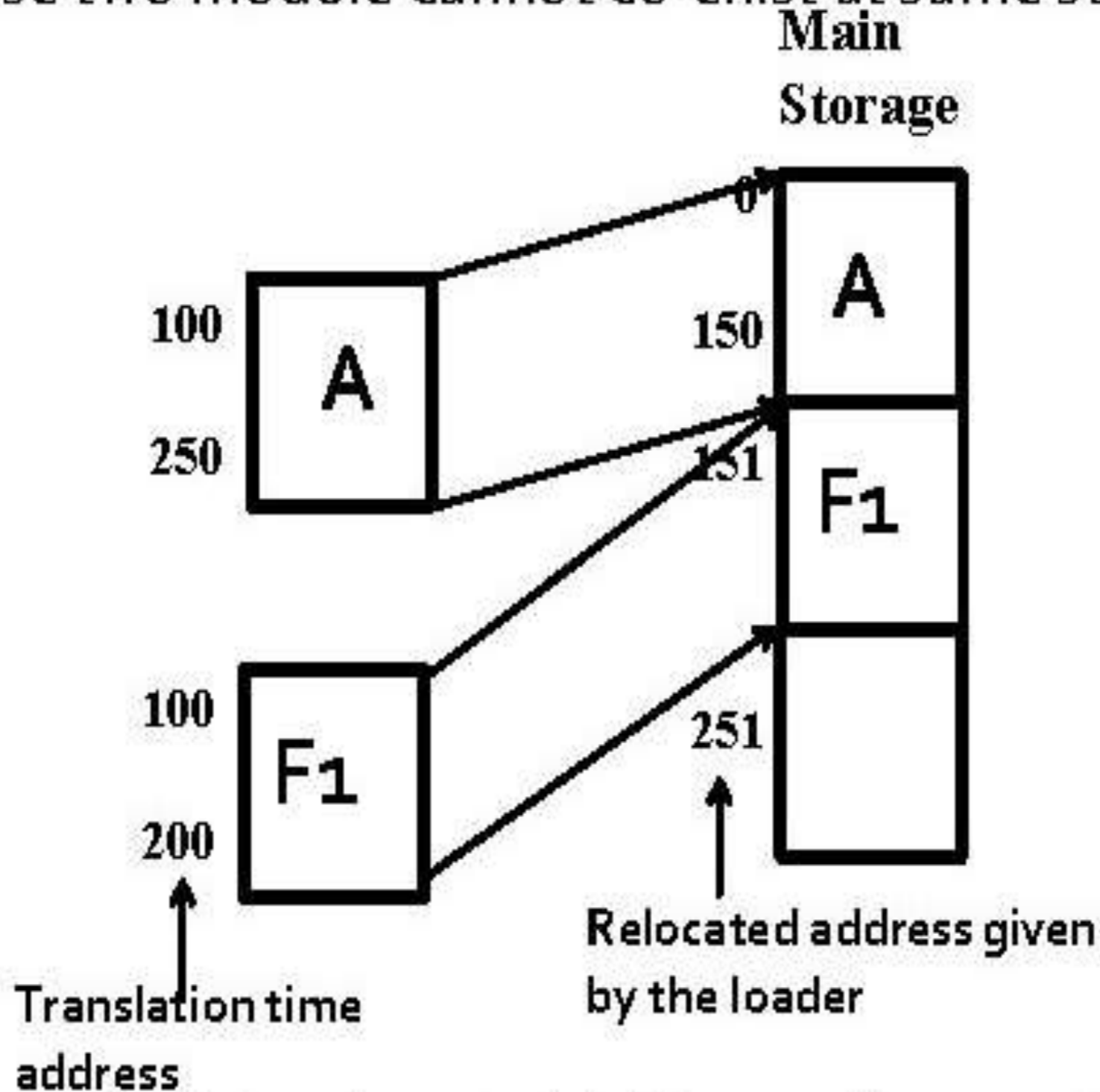
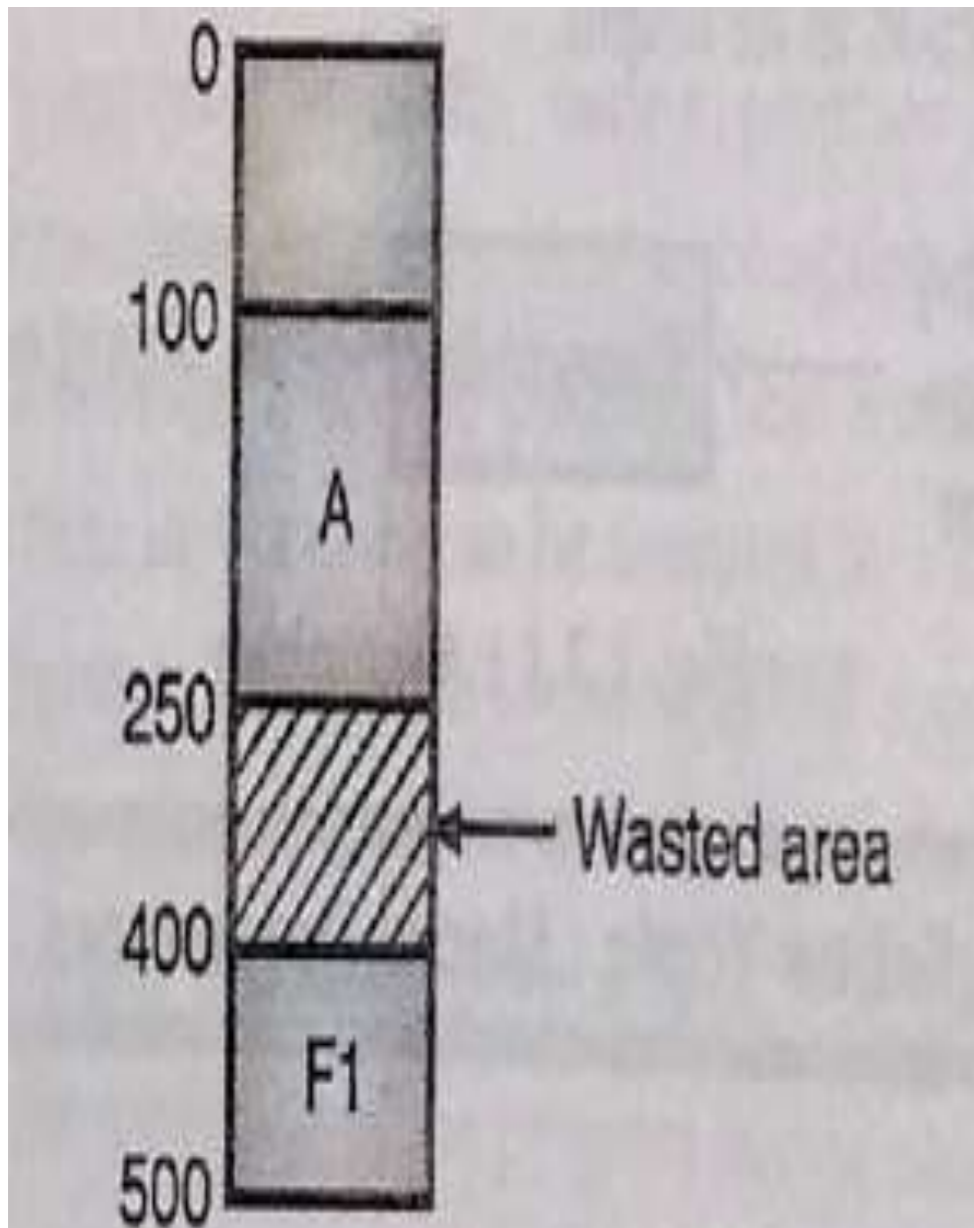


Fig.: Relocation to avoid address conflict or storage waste

RELOCATION



(S5.2) Fig. 1.2.2 : Case I of relocation

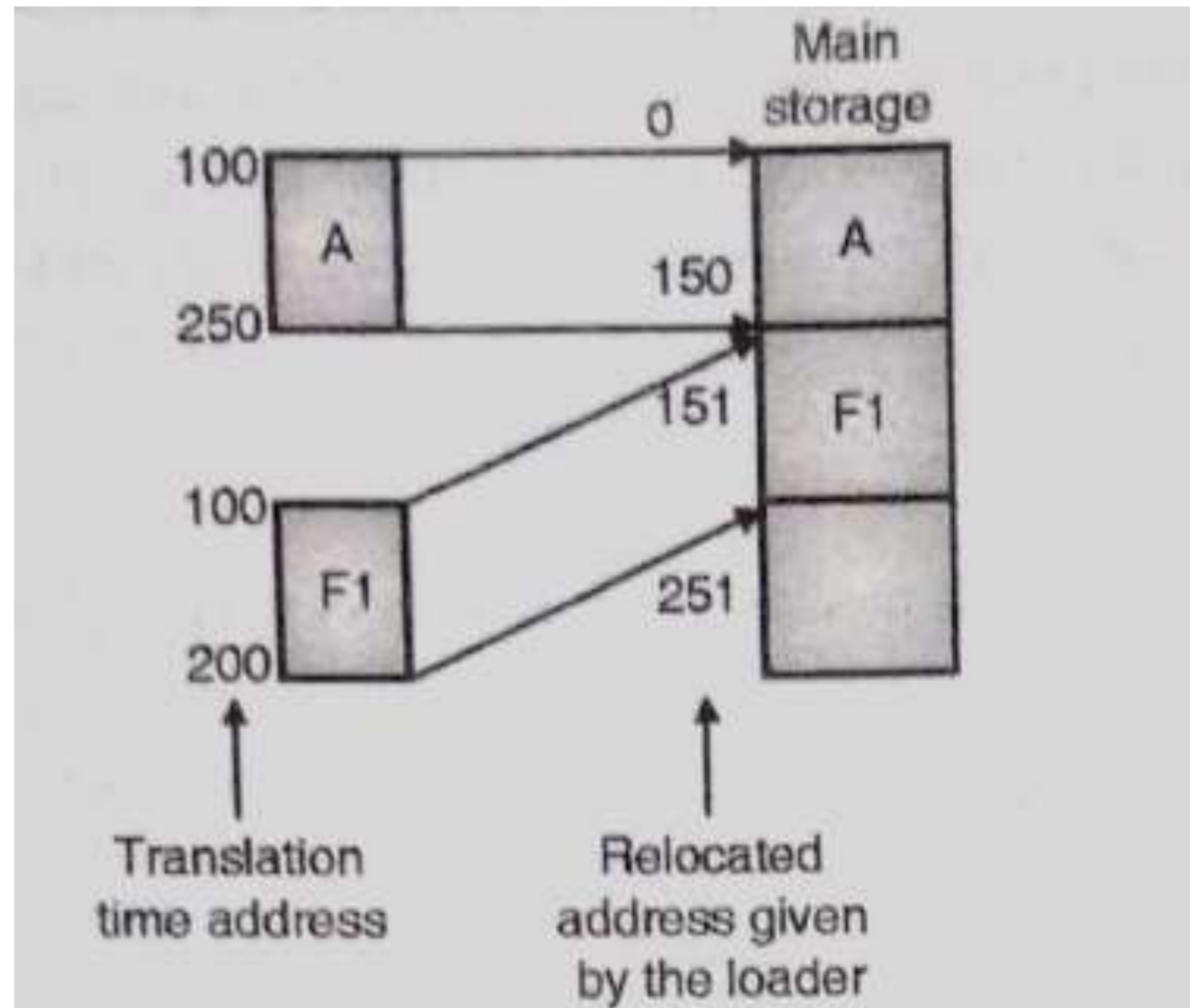
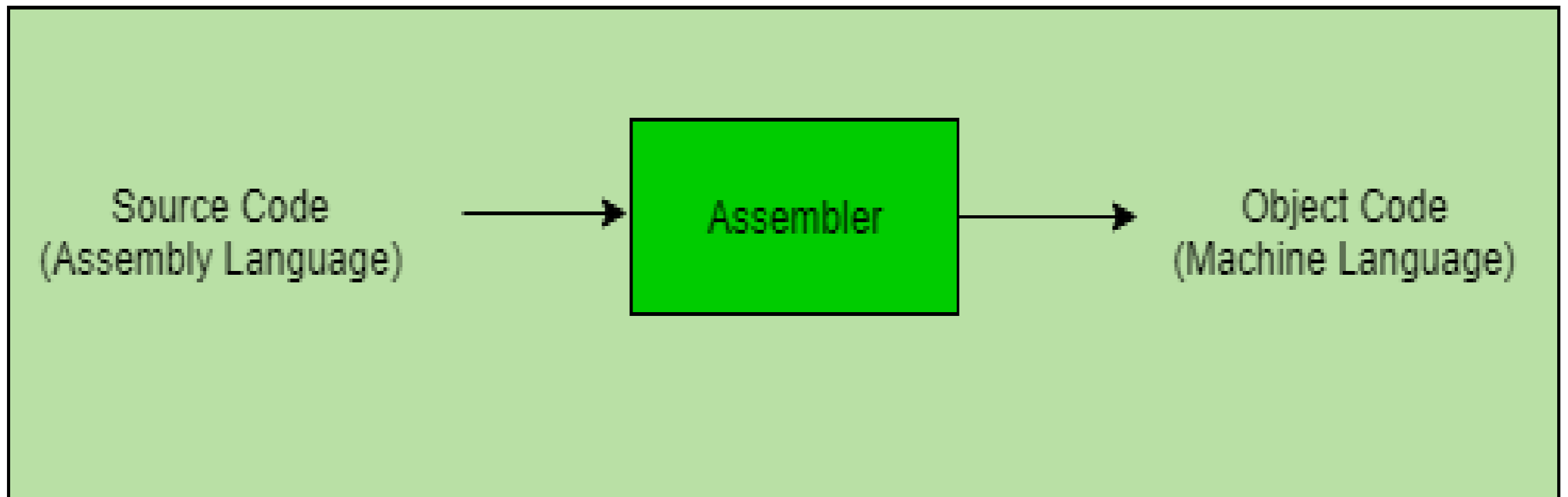


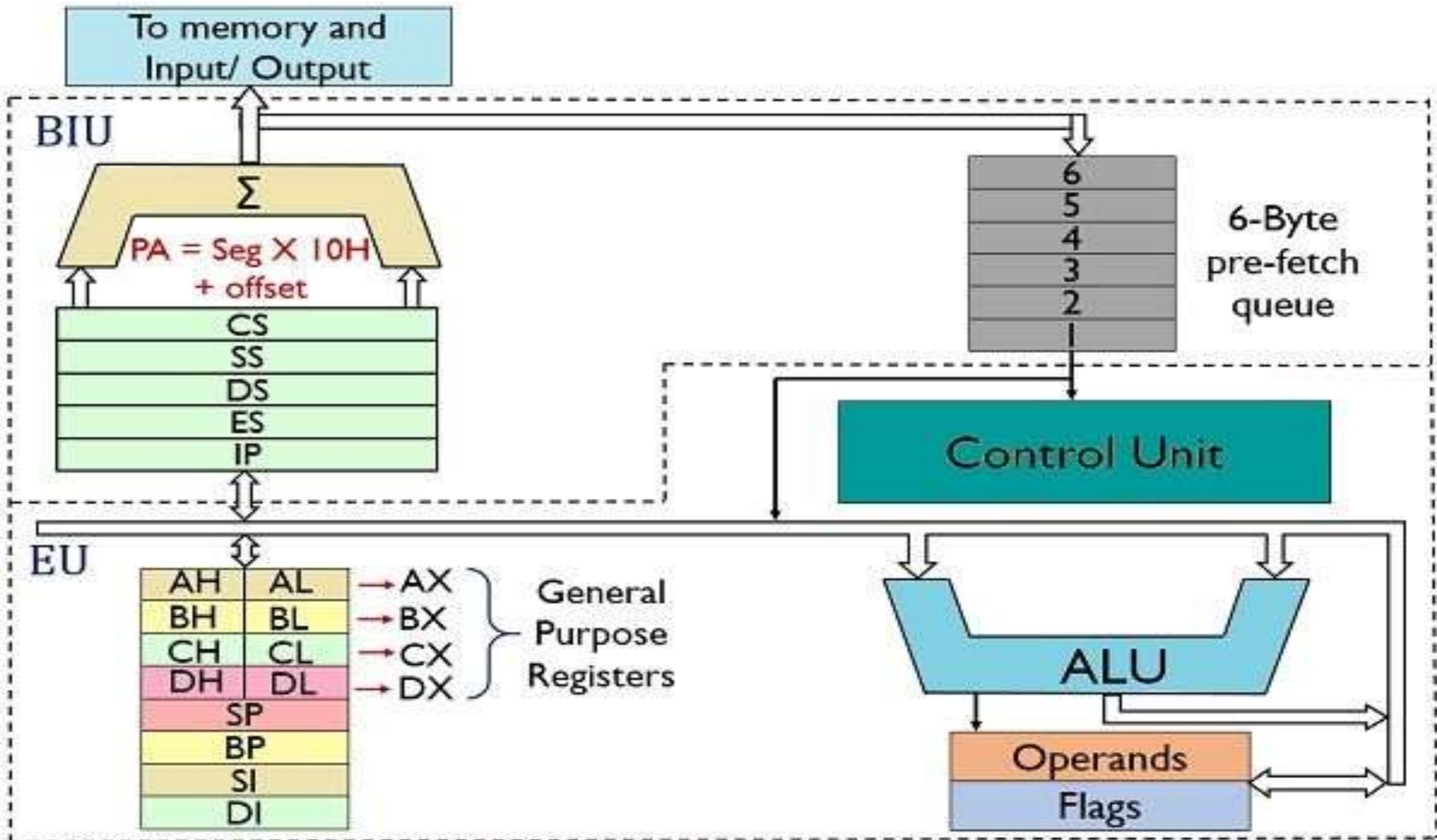
Fig. 1.2.3 : Relocation to avoid address conflict or storage waste

Assembler

- Assembler is a translator which translates assembly language program into machine language.



Microprocessor



Block Diagram of 8086 Microprocessor

Macro & Macroprocessor

- **Macro** allows a sequence of source language code to be defined once and then referred many times.
- **“ A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program ”**
- In NASM, macros are defined with `%macro` and `%endmacro` directives.
- The macro begins with the `%macro` directive and ends with the `%endmacro` directive.
- **Syntax:**
`%macro macro_name number_of_params`
`<macro body>`
`%endmacro`

Macro & Macroprocessor

- A **macro processor** takes a source with macro definition and macro calls and replaces each macro call with its body.

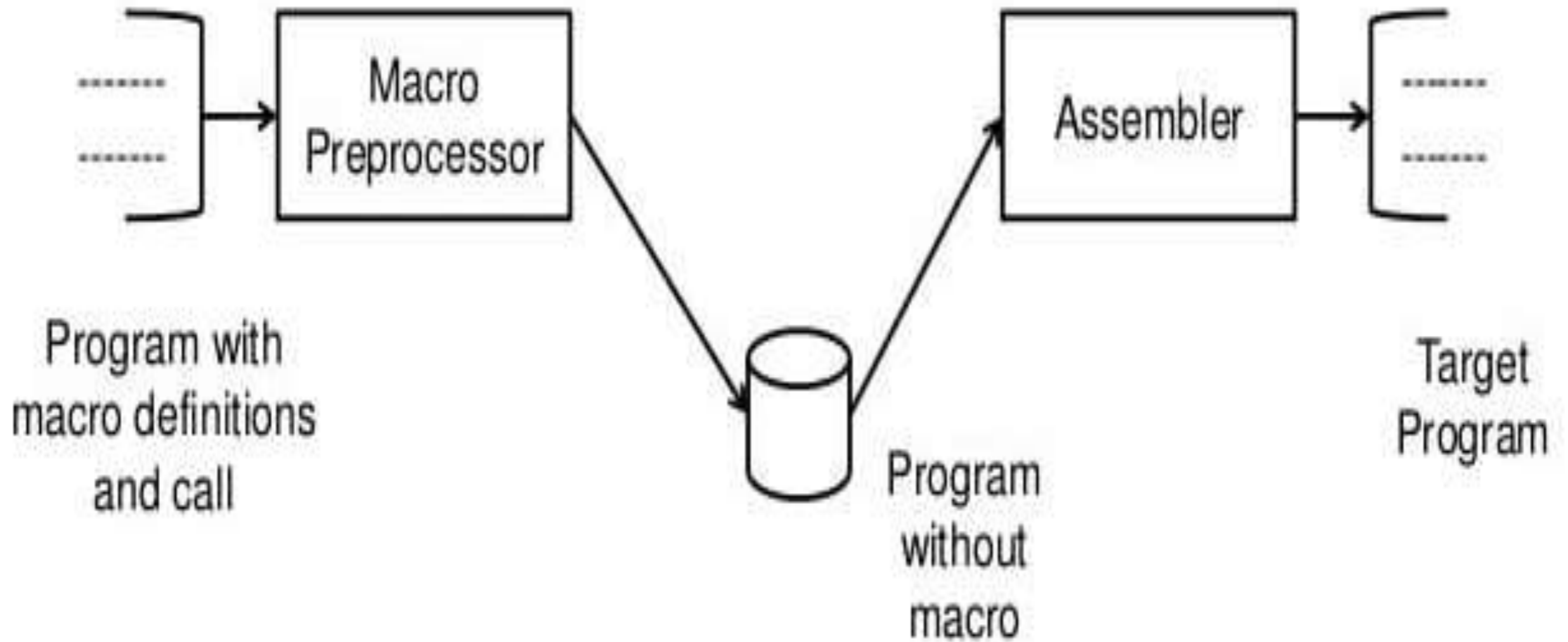


Figure: A Schematic of a macro preprocessor
MR. ANAND GHARU

Macro & Macroprocessor

- How Macroprocessor works :

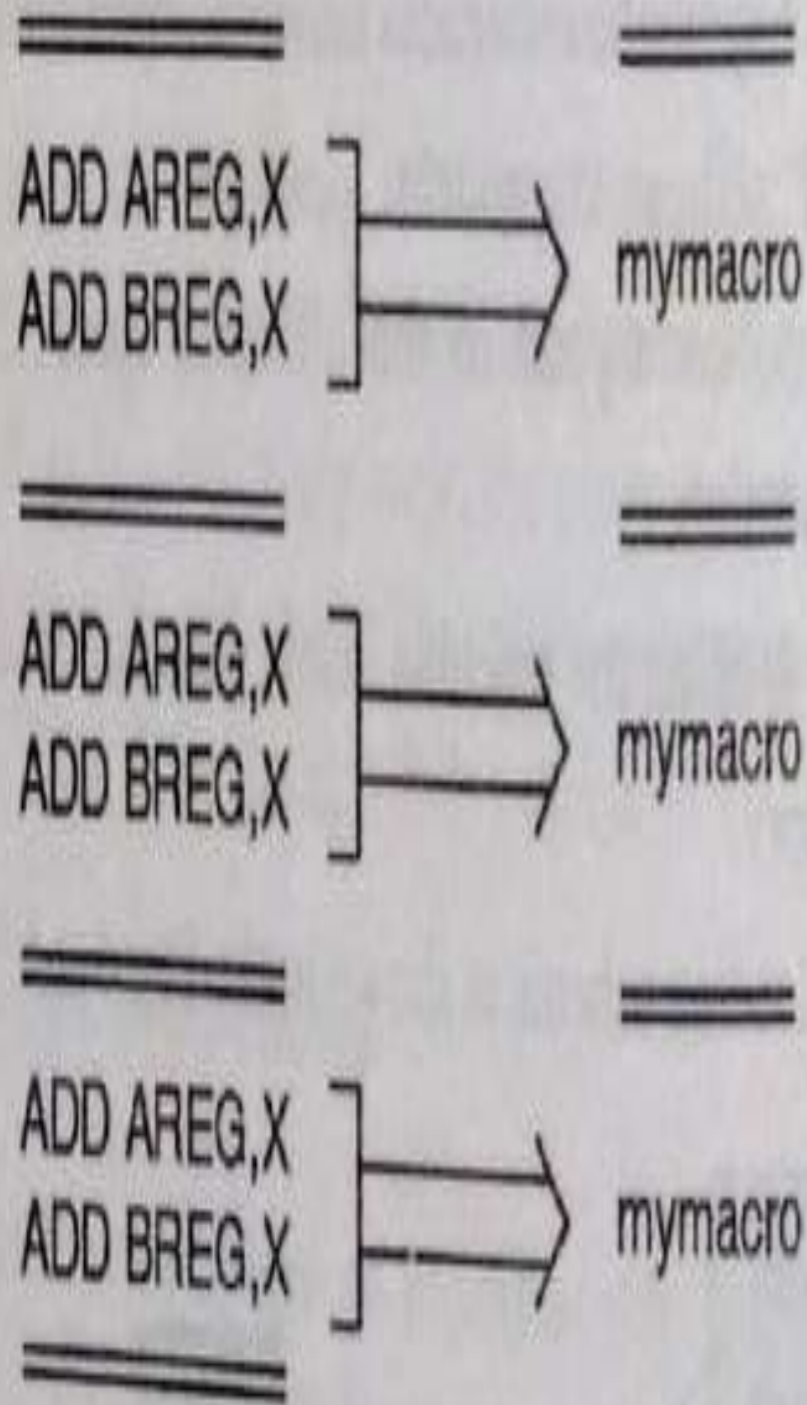
```
ADD AREG, X
ADD BREG, X
=====
ADD AREG, X
ADD BREG, X
=====
ADD AREG, X
ADD BREG, X
=====
```

In the above program, the sequence
ADD AREG, X
ADD BREG, X
occurs three times. A macro allows us to attach a name to this sequence and to use this name in its place.

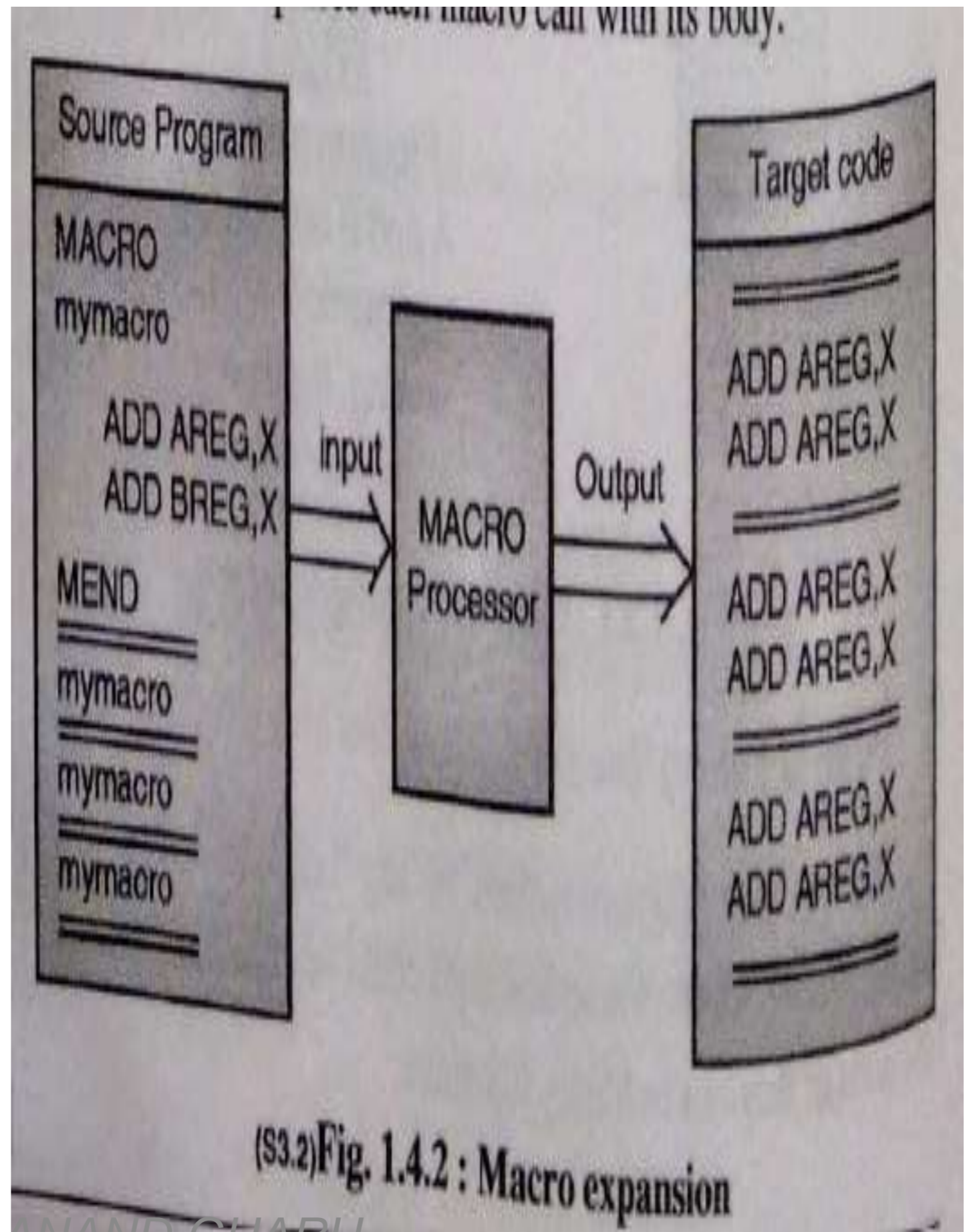
Start of definition	MACRO
macro name	mymacro
macro body	[ADD AREG, X ADD BREG, X]
End of macro definition	MEND

Original Program

Program with macro

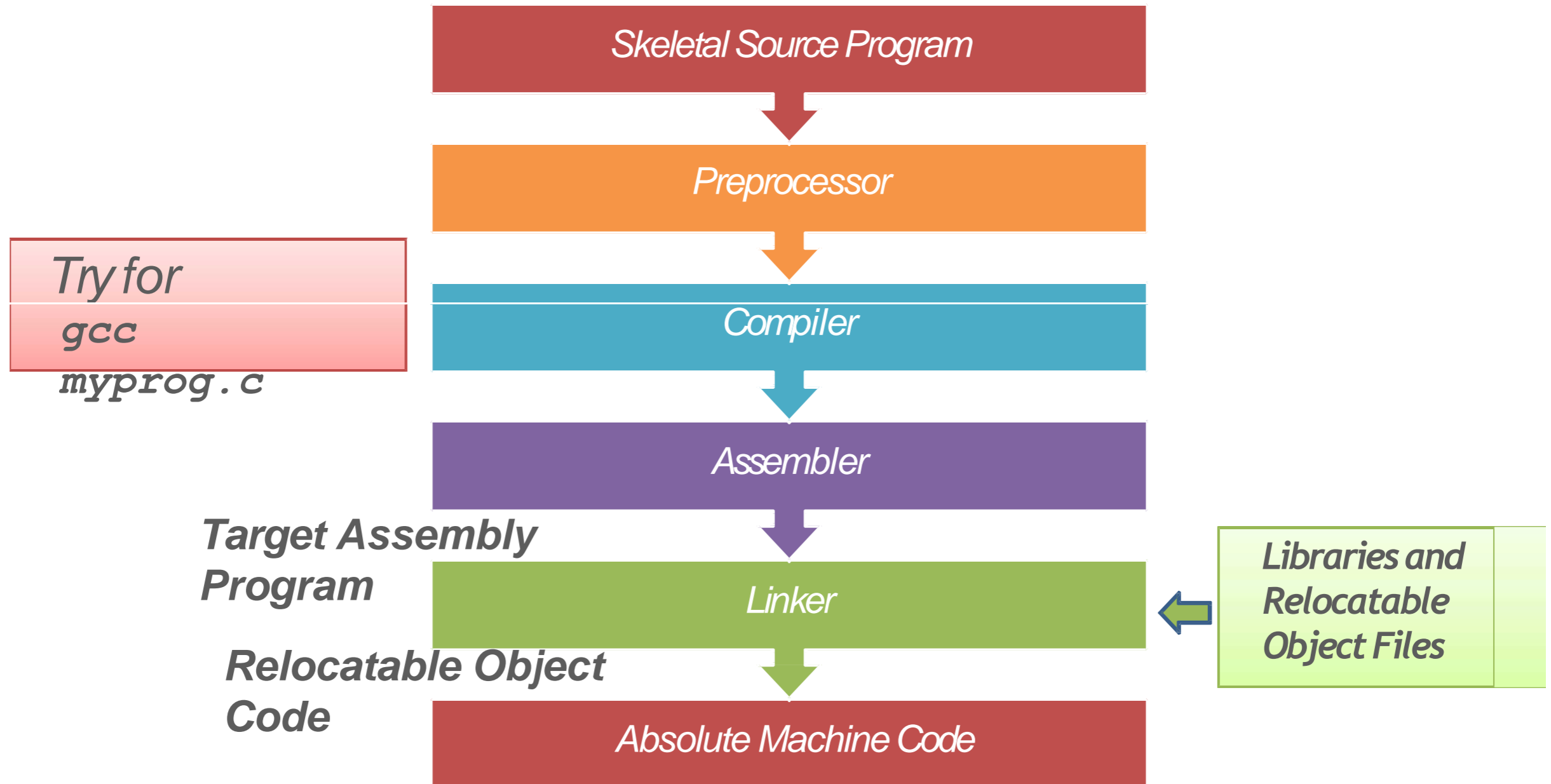


(S3.1) Fig. 1.4.1 : Re-writing a program with macros



(S3.2) Fig. 1.4.2 : Macro expansion

PREPROCESSORS, COMPILERS, ASSEMBLERS, AND LINKERS



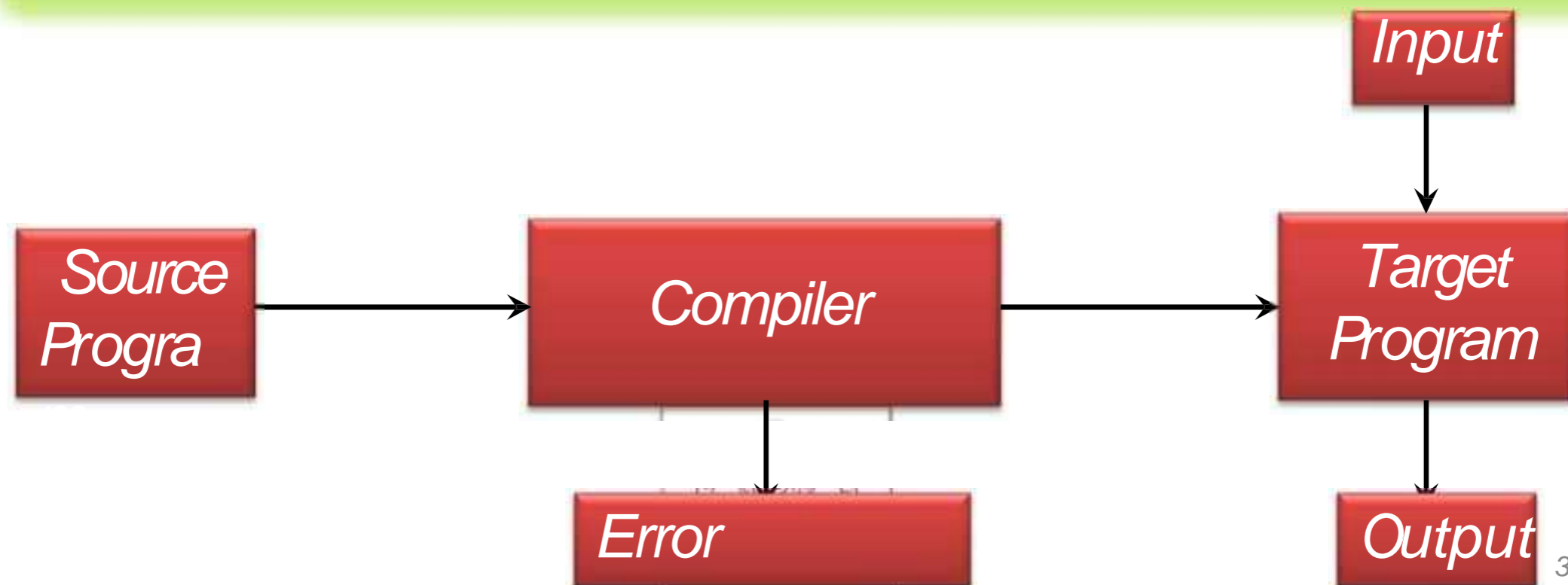
Compiler

- Compiler is a translator which converts the high level language into low level language.
- Benefits of writing a program in a high level language :
- **Increases productivity:** It is very easy to write a program in a high level language.
- **Machine Independence:** A program written in a high level language is machine independent.

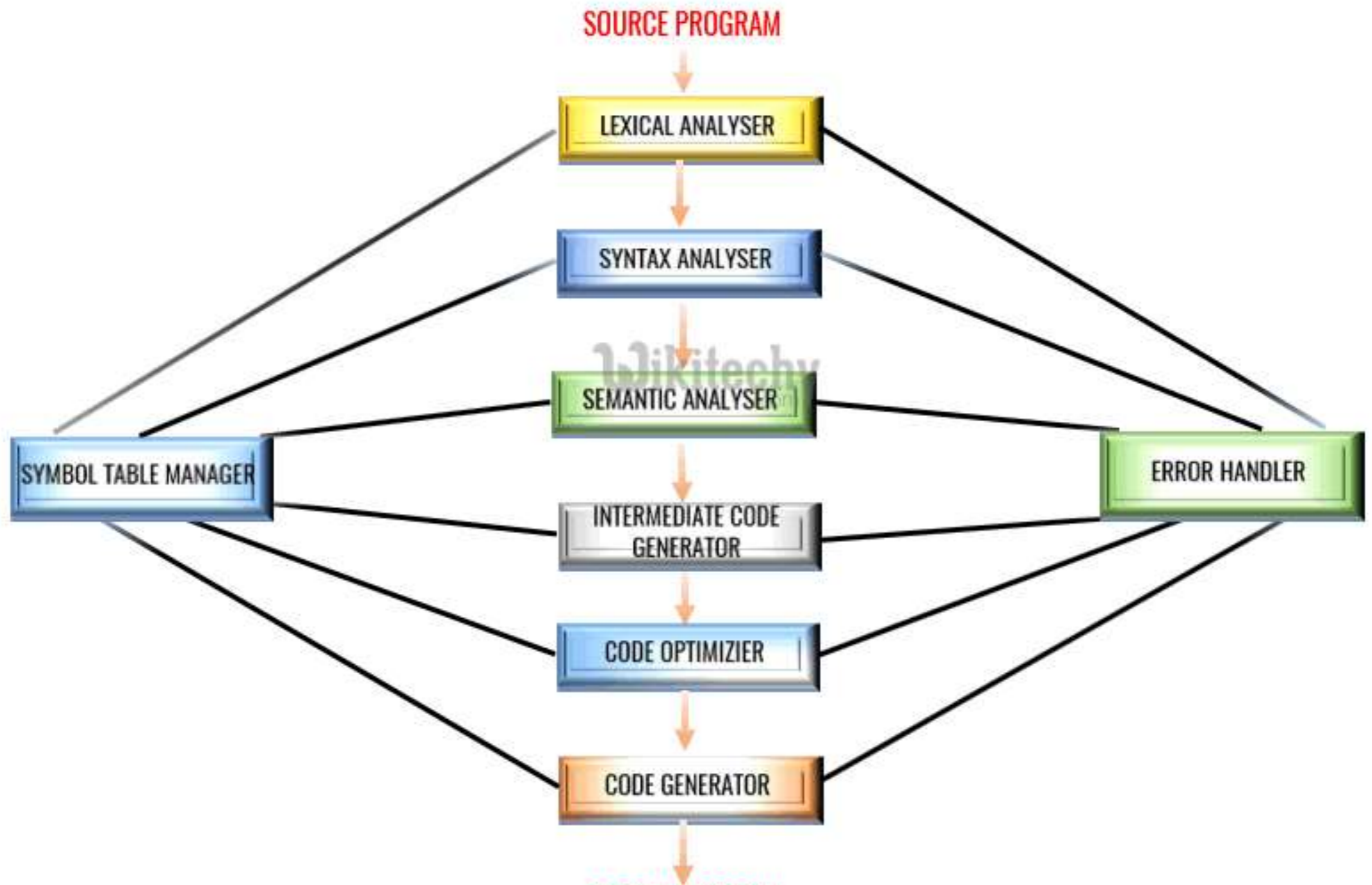


COMPILERS

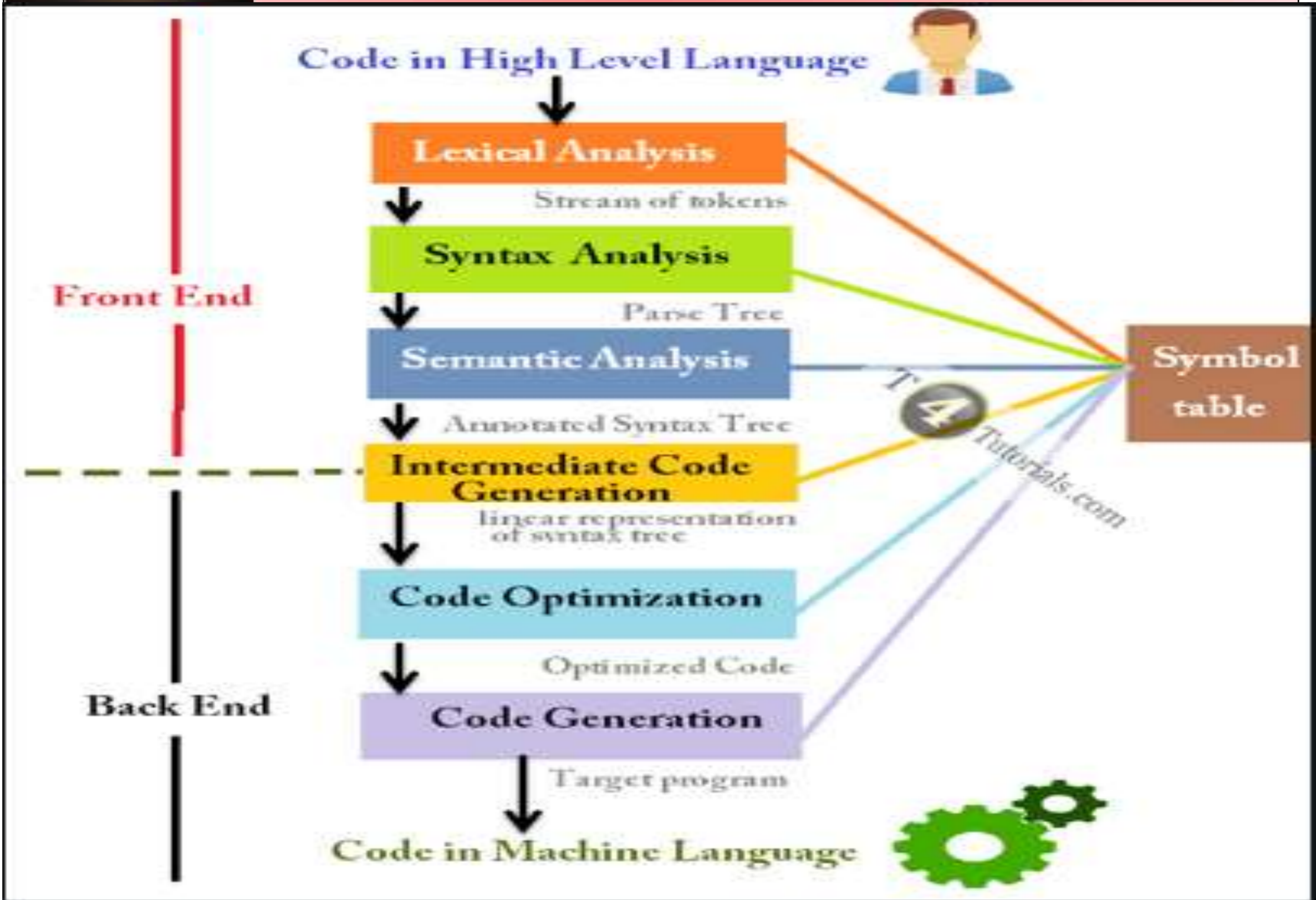
- *“Compilation”*
 - *Translation of a program written in a source language into a semantically equivalent program written in a target language*



Phases of Compiler

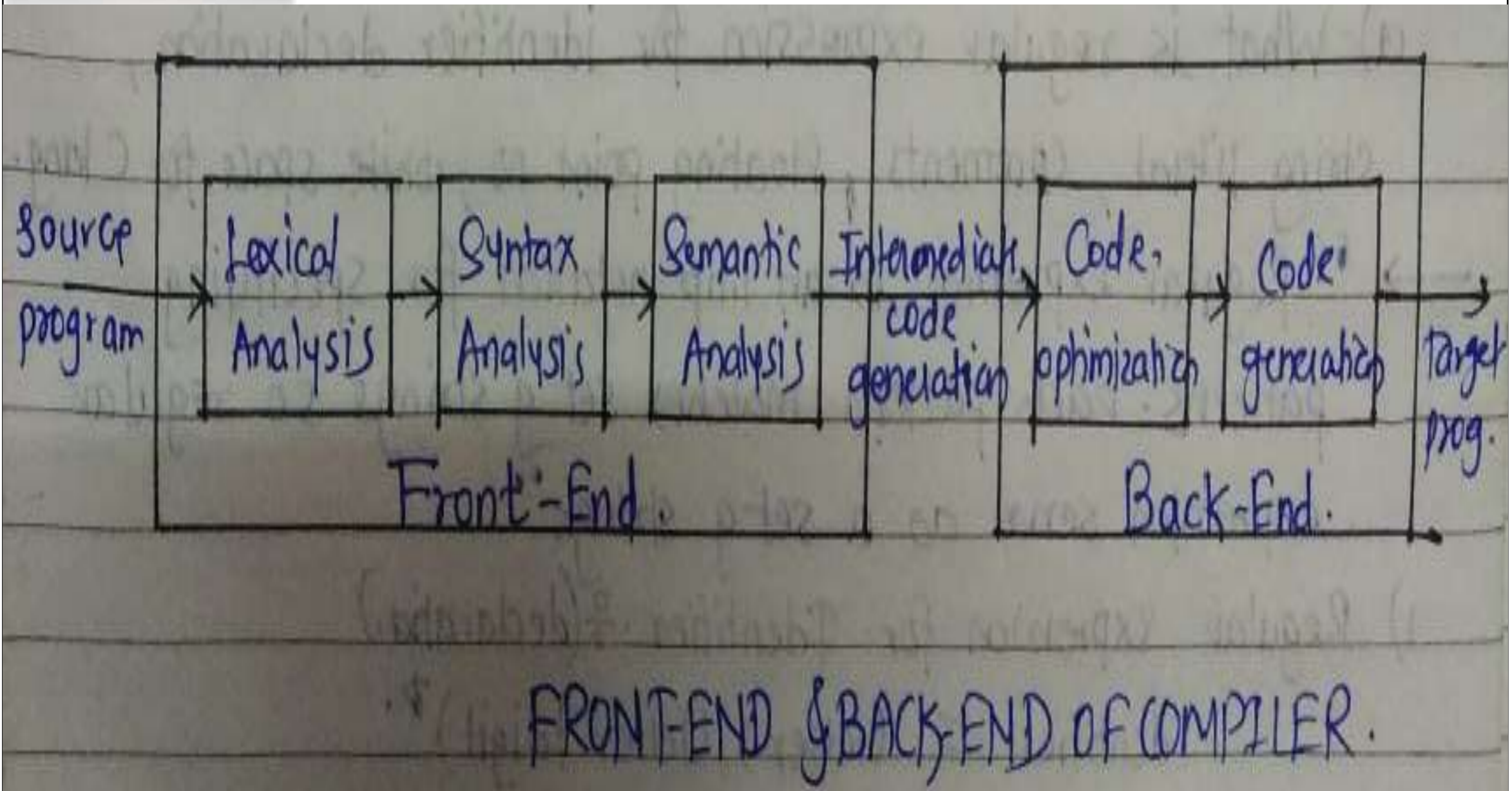


PHASES OF COMPILER





FRONT END & BACK END OF COMPILER



Phases of Compiler

- **Lexical Analysis :**

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

- **Syntax Analysis :**

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Phases of Compiler

- **Semantic Analysis :**

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

- **Intermediate Code Generation :**

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Phases of Compiler

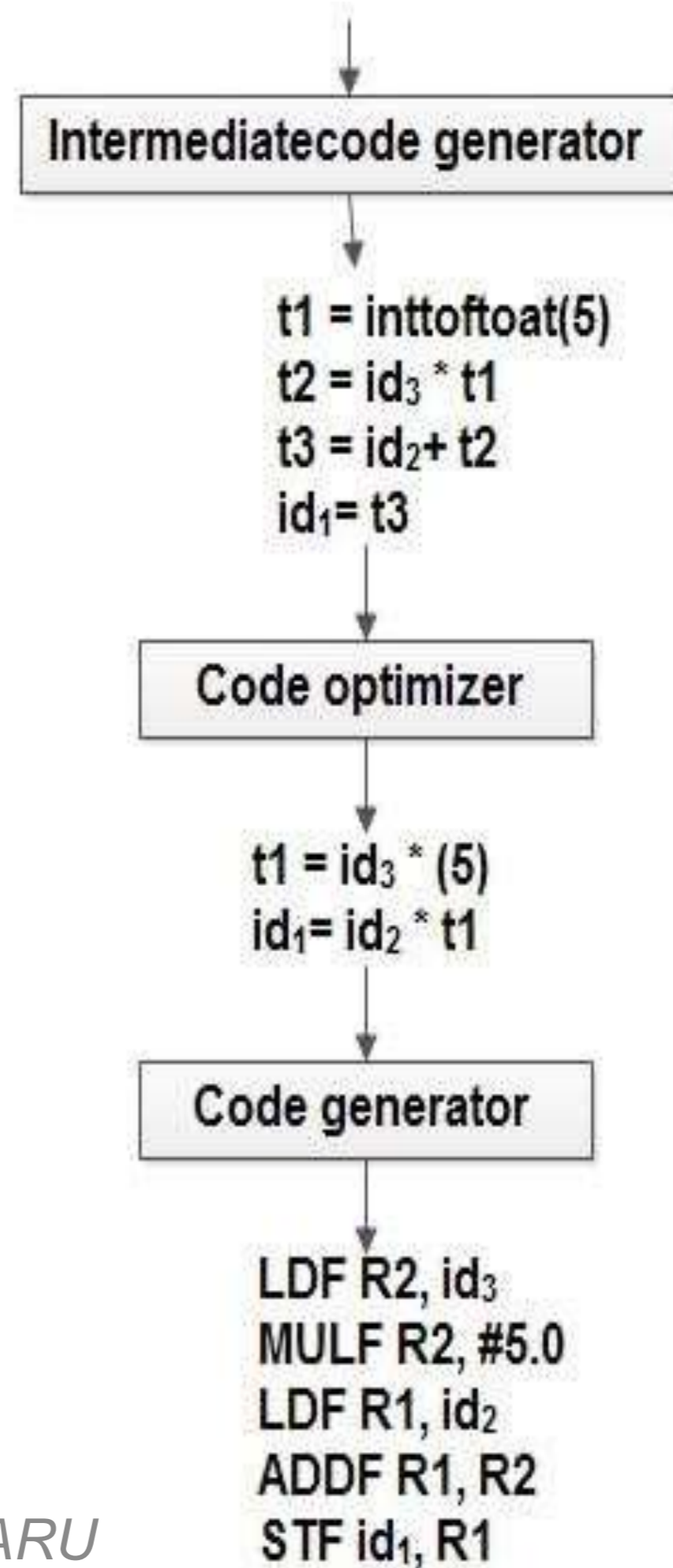
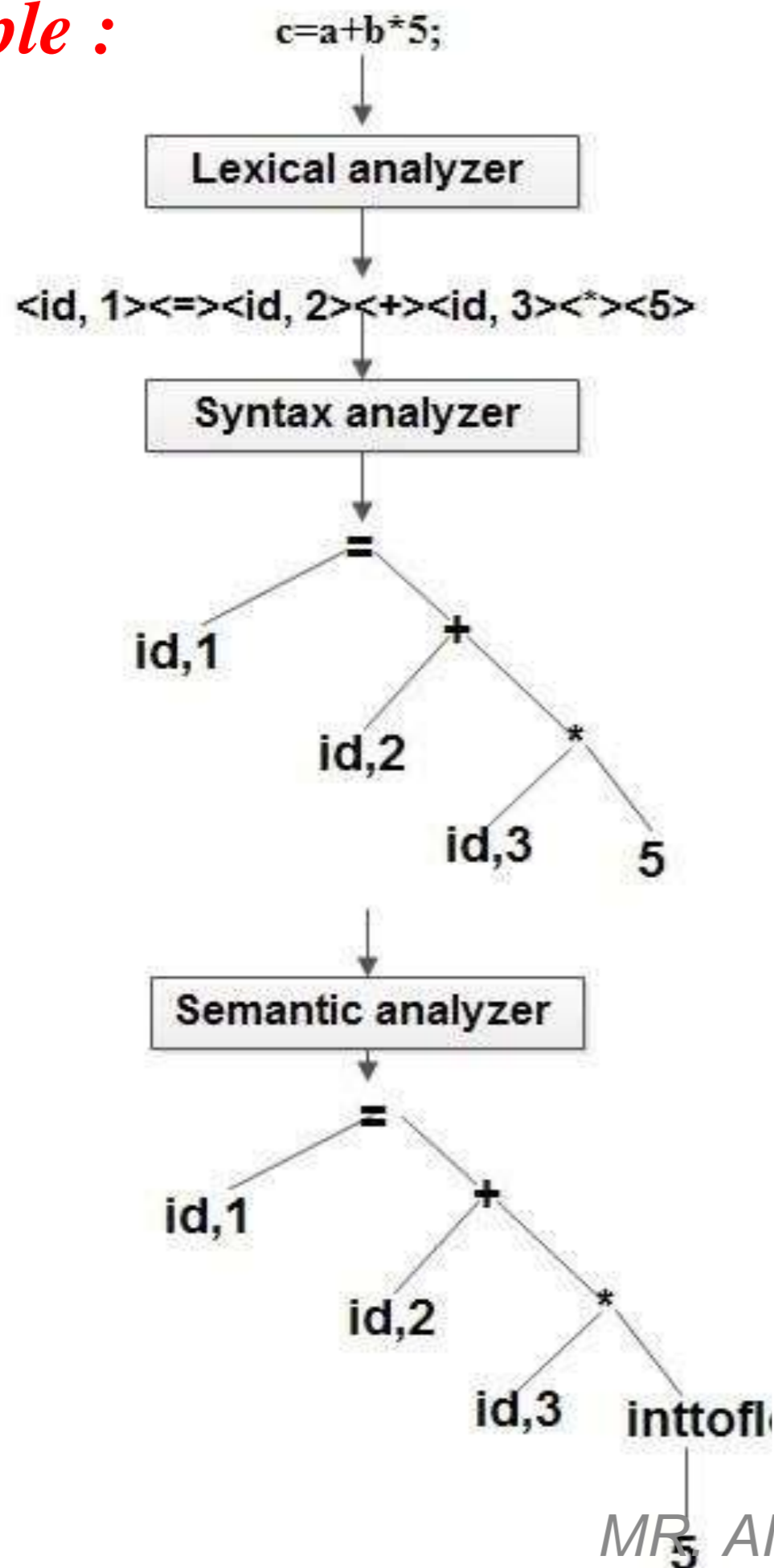
- **Code Optimization :**

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

- **Code Generation :**

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

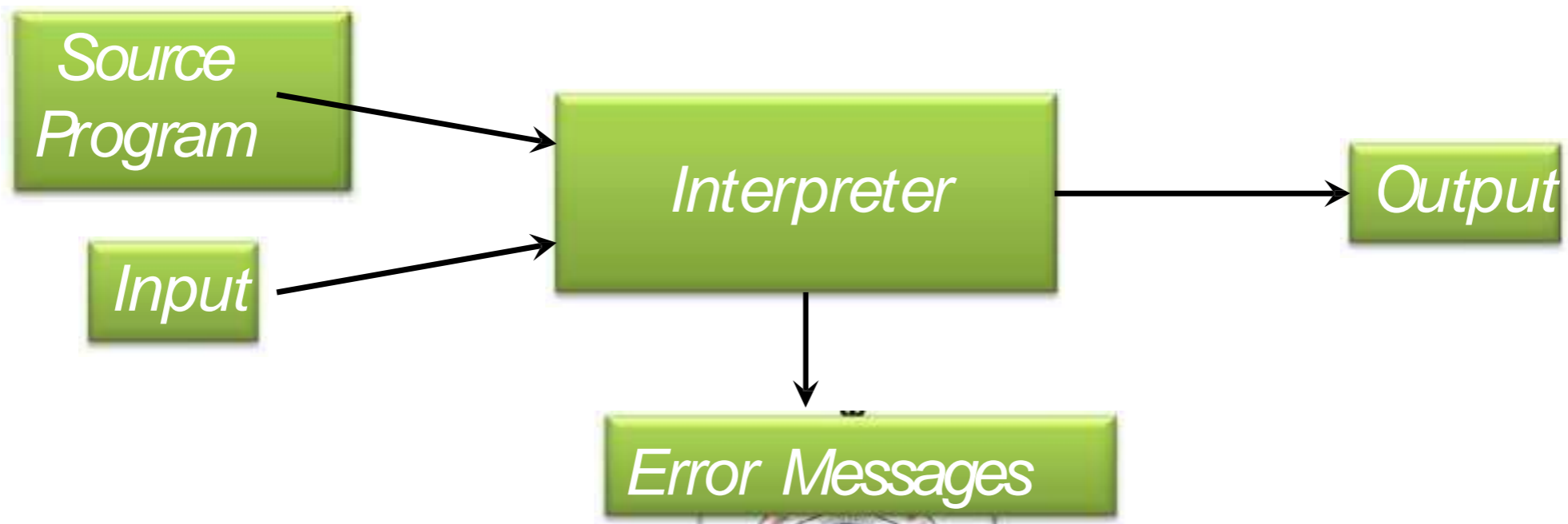
Example :

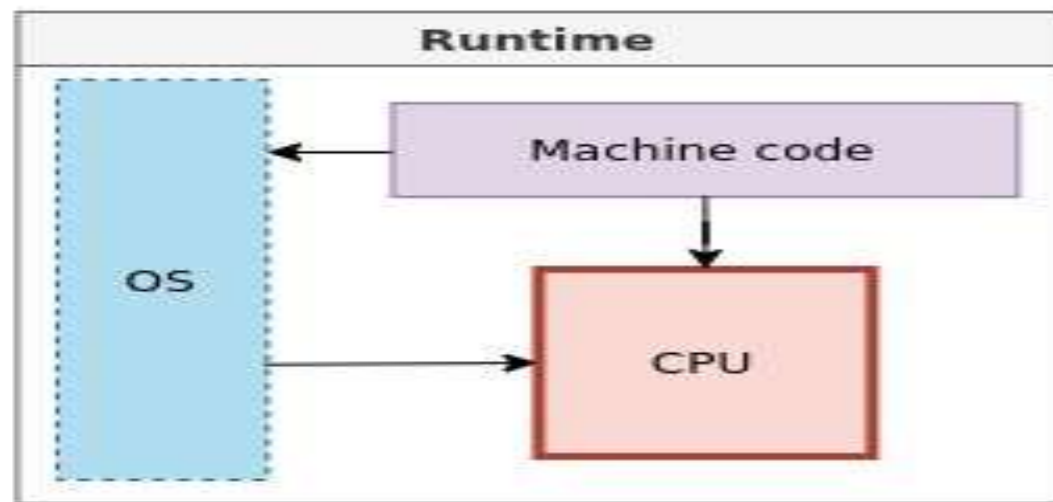
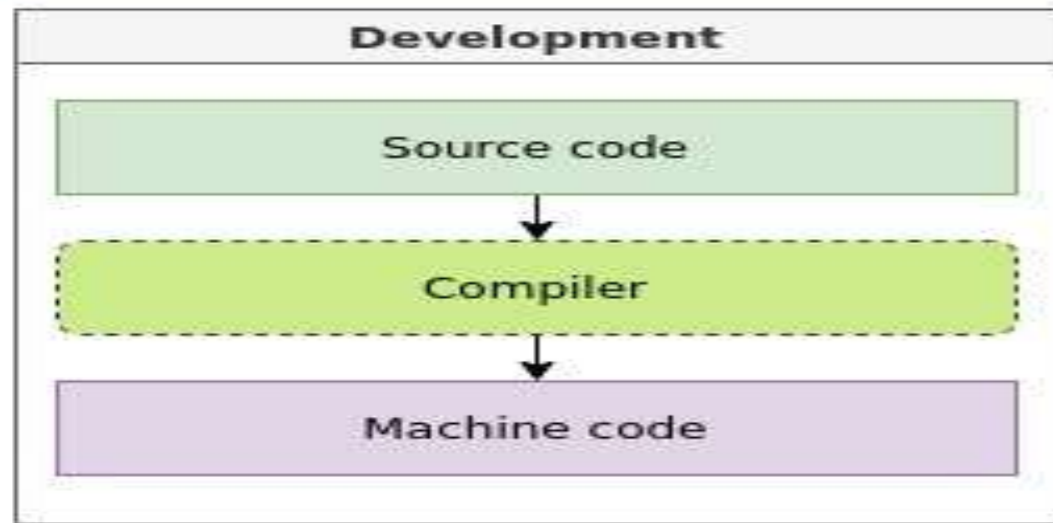




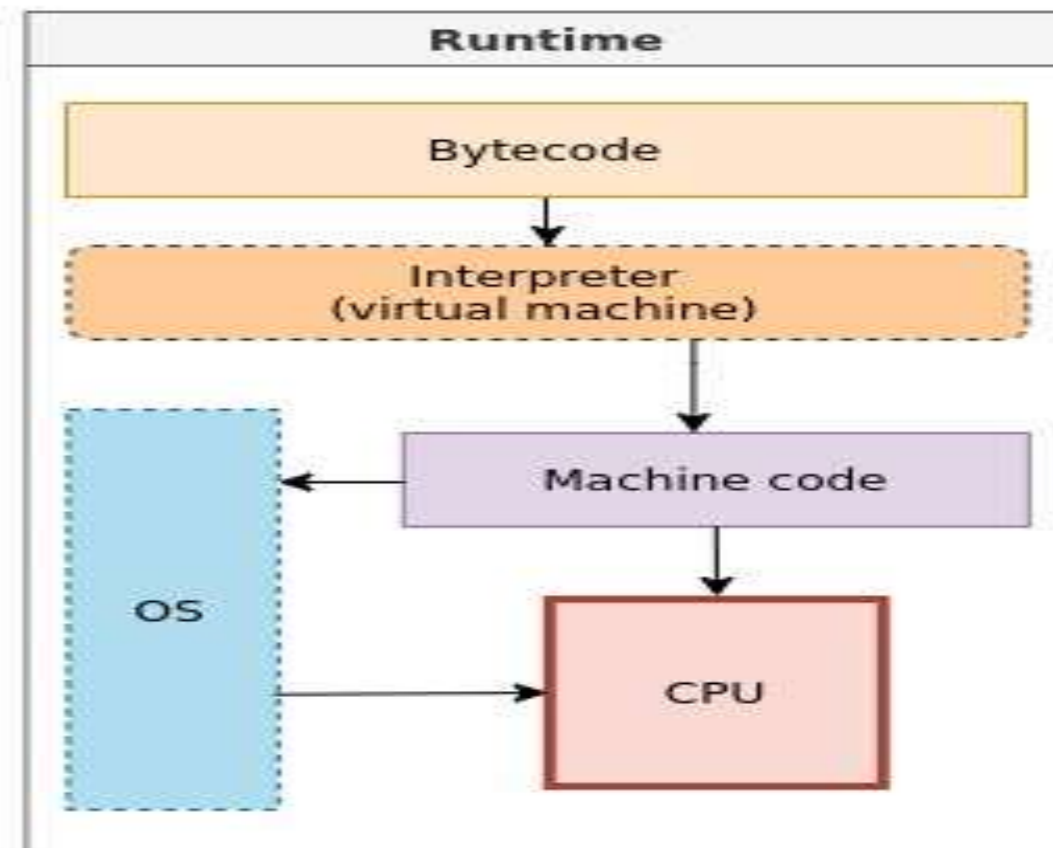
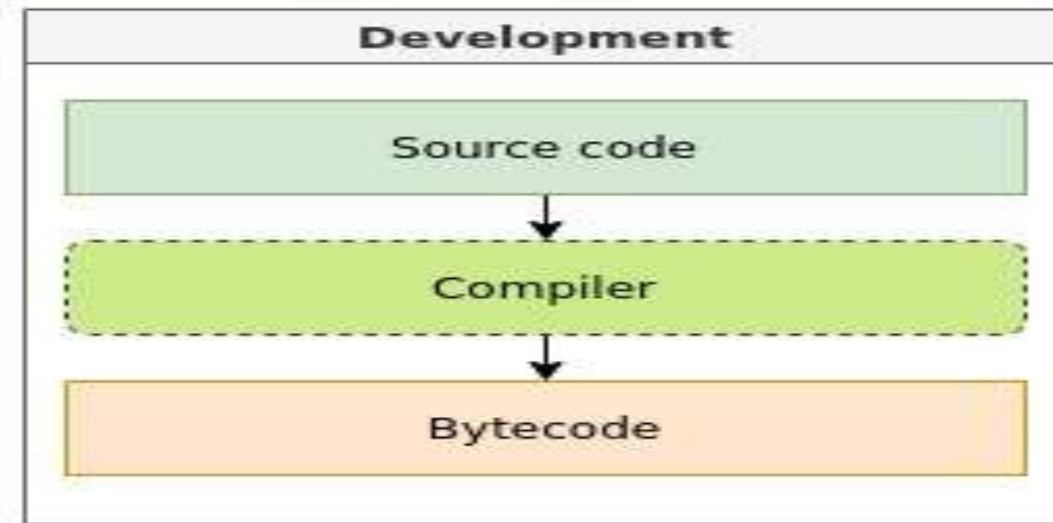
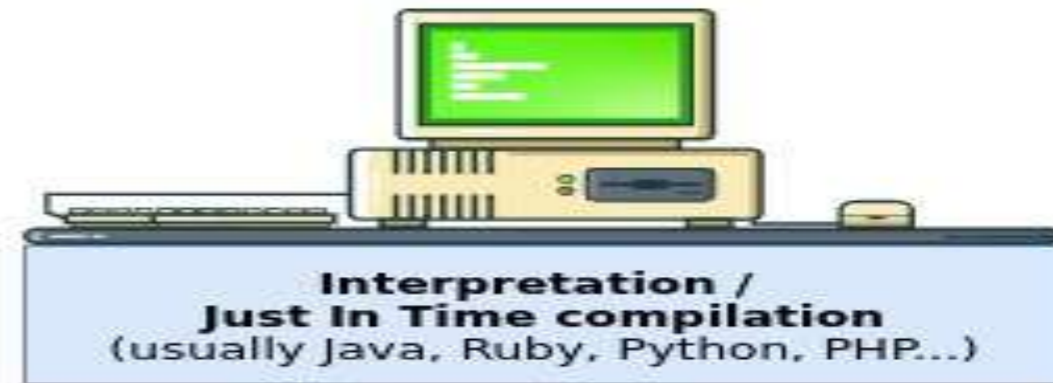
INTERPRETERS

- *“interpretation”*
 - *Performing the operations implied by the source program*





→ Input / Output



Difference between Compiler and Interpreter

No	Compiler	Interpreter
1	Compiler Takes Entire program as input	Interpreter Takes Single instruction as input .
2	Intermediate Object Code is Generated	No Intermediate Object Code is Generated
3	Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
4	Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
5	Program need not be compiled every time	Every time higher level program is converted into lower level program
6	Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
7	Example : C Compiler	Example : BASIC

Debugger

- Debugging tool helps programmer for testing and debugging programs.
- It provides some facilities:
- **Setting breakpoints.**
- **Displaying values of variables.**

Device driver

- Debugging tool helps programmer for testing and debugging programs.
- It provides some facilities:
- **Setting breakpoints.**
- **Displaying values of variables.**

Operating system

- It is system software which provides interface between user and hardware(computer system)

Assembly Language

MR. ANAND GHARU

ASSEMBLY LANGUAGE PROGRAMMING



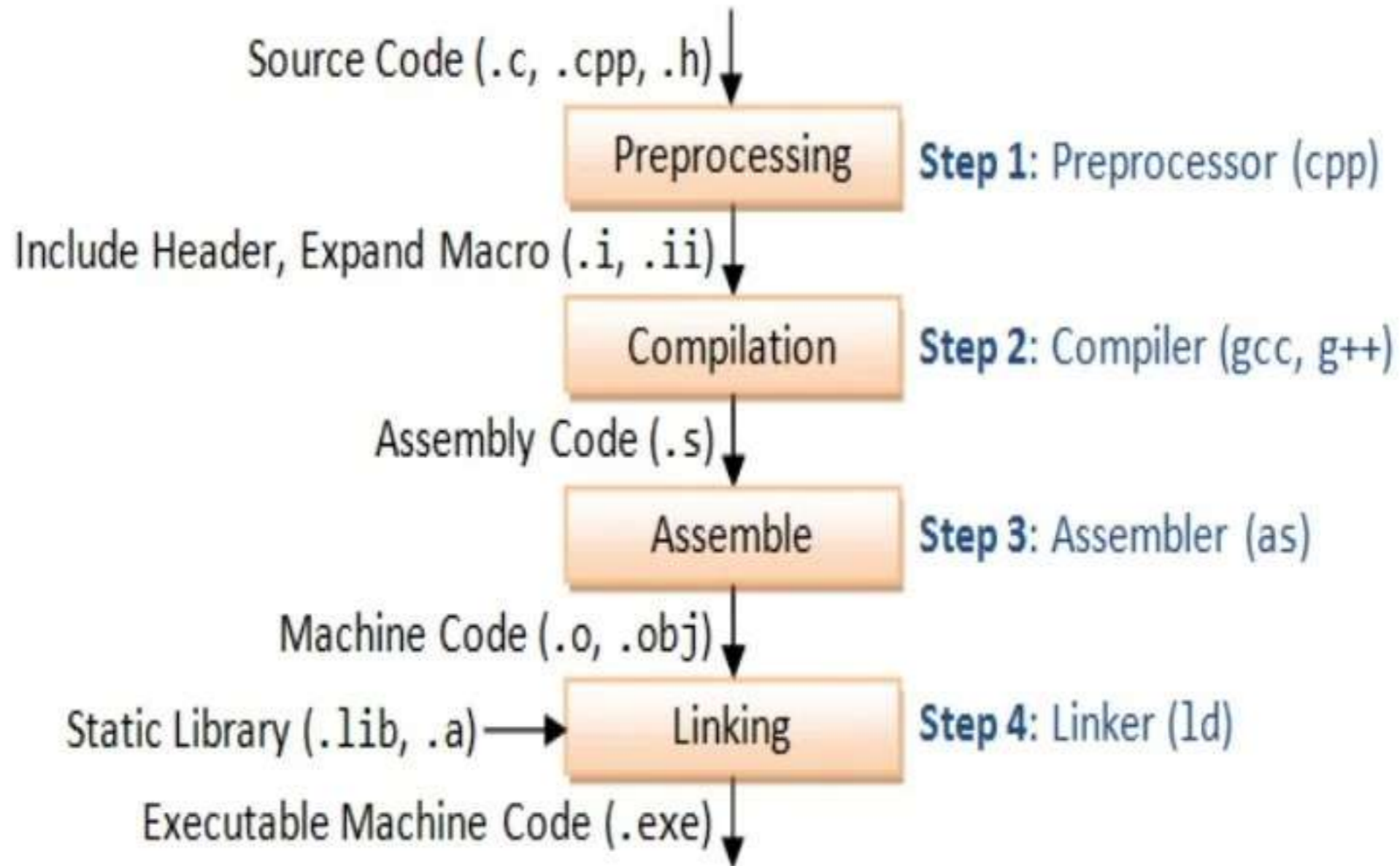
Assembly Language

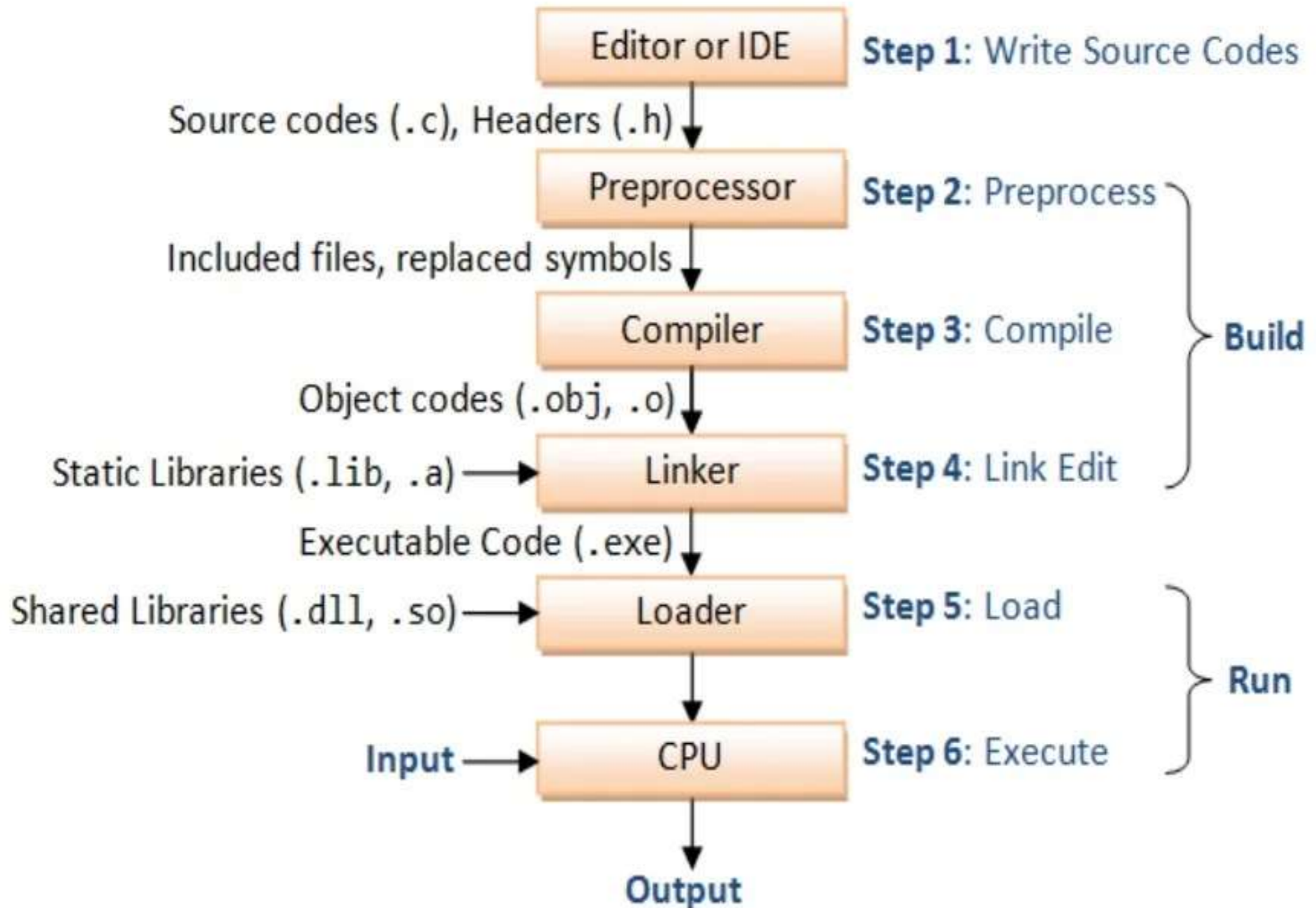
- Assembly language is low level language.
- An assembly language is machine dependent.
- It differs from computer to computer.
- Writing programs in assembly language is very easy as compared to machine(binary) language.

Assembly Language Programming (ALP)

- Assembly language is a kind of low level programming language, which uses symbolic codes or mnemonics as instruction.
- Some examples of mnemonics include ADD, SUB, LDA, and STA that stand for addition, subtraction, load accumulator, and store accumulator, respectively.
- For processing of an assembly language program we need a language translator called assembler
- Assembler- Assembler is a Translator which translates assembly language code into machine code

Position of Assembler





Applications of Assembly Language

- **assembly language** is used for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues.
- Typical **uses** are device drivers (CD, HDD), low-level embedded systems (Keyboard, water tank indicator) and real-time systems (computer, notepad).

Advantage and Disadvantages of ALP

- **Advantages-**

- Due to use of symbolic codes (mnemonics), an assembly program can be written faster.
- It makes the programmer free from the burden of remembering the operation codes and addresses of memory location.
- It is easier to debug.

- **Disadvantages-**

- it is a machine oriented language, it requires familiarity with machine architecture and understanding of available instruction set.
- Execution in an assembly language program is comparatively time consuming compared to machine language. The reason is that a separate language translator program is needed to translate assembly program into binary machine code

Machine Instruction in Assembly Language

- **Machine Instructions are commands or programs written in machine code of a machine (computer) that it can recognize and execute.**
1. A machine instruction consists of several bytes in memory that tells the processor to perform one machine operation.
 2. The processor looks at machine instructions in main memory one after another, and performs one machine operation for each machine instruction.
 3. The collection of machine instructions in main memory is called a machine language program.

Machine Instruction in Assembly Language

The general format of a **machine instruction** is

[Label:] Mnemonic [Operand, Operand] [; Comments]

1. Brackets indicate that a field is optional
2. Label is an identifier that is assigned the address of the first byte of the instruction in which it appears. It must be followed by “:”
3. Inclusion of spaces is arbitrary, except that at least one space must be inserted; no space would lead to an ambiguity.
4. Comment field begins with a semicolon “ ; ”

Example:

Here: MOV R5,#25H; load 25H into R5

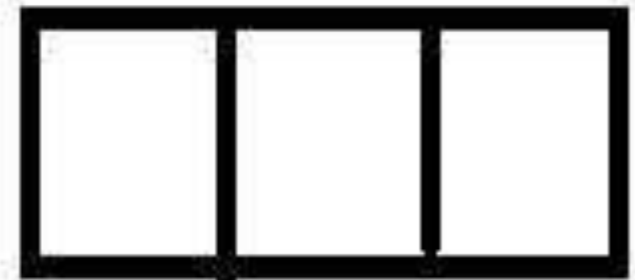
- **Machine instruction**
Format:



opcode



register operand



memory operand

Assembly language programming

Terms:

- **Location Counter:** (LC) points to the next instruction.
- **Literals:** constant values
- **Symbols:** name of variables and labels
- **Procedures:** methods/ functions

Assembly language Statements:

- Imperative Statements:
- Declarative/Declaration Statements:
- Assembler Directive:

Imperative Statements

- Imperative means mnemonics
- These are executable statements.
- Each imperative statement indicates an action to be taken during execution of the program.
- E.g
 - MOVER
 - BREG, X
 - STOP
 - READ X
 - ADD AREG, Z

Declarative Statements

- Declaration statements are for reserving memory for variables.
- We can specify the initial value of a variable.
- It has two types:
- DS // Declare Storage
- DC // Declare Constant

DS(Declare Storage):

- Syntax:
- [Label] DS <Constant specifying size>
- E.g. X DS 1

DC (Declare Constant):

Syntax:

[Label] DC <constant specifying value>

E.g Y DC '5'

Assembler Directive

- Assembler directive instruct the assembler to perform certain actions during assembly of a program.
- Some assembler directive are:
- `START <address constant>`
- `END`

Advanced Assembler Directives

- 1. ORIGIN
- 2. EQU
- 3. LTORG

Sample Assembly Code

1. `START 100` **It is an AD statement because it has Assembler directive START**
2. `MOVER AREG, X` **It is an IS because it starts with mnemonic.**
3. `MOVER BREG, Y`
4. `ADD AREG, Y`
5. `MOVEM AREG, X`
6. `X DC '10'` **It is an DS/ DL statement because it has DC**
7. `Y DS 1` **It is an DS/ DL statement because it has DS**
8. `END`

Identify the types of

State.No	IS	DS	AD
1			
2			
3			
4			
5			
6			
7			
8	<i>MR. ANAND GHARU</i>		

Identify the types of

State.No	IS	DS	AD
1			AD
2	IS		
3	IS		
4	IS		
5	IS		
6		DS	
7		DS	
8	<i>MR. ANAND GHARU</i>		AD

Advanced Assembler Directives

- ORIGIN
- EQU
- LTORG

Advanced Assembler Directives

Origin :

The origin directive tells the assembler where to load instructions and data into memory.

SYNTAX :

ORIGIN

Advanced Assembler Directives

Equate:

The EQU directive is used to equate a name with an expression, symbolic address, or number. Whenever this name is used as a symbol, it is replaced.

We might do something, such as the following, which makes the symbol R12 to be equal to 12, and replaced by that value when the assembler is run.

E.G. R12 EQU 12

Advanced Assembler Directives

LTORG :

The LTOrg directive instructs the assembler to assemble the current literal pool immediately.

The Literal Pool contains a collection of anonymous constant definitions, which are generated by the assembler.

The LTOrg directive defines the start of a literal pool.

Syntax :

LTORG

How LC Operates?

Sr. NO		LC
1	START 100	
2	MOVER AREG, X	
3	MOVER BREG, Y	
4	ADD AREG, BREG	
5	MOVEM AREG, X	
6	X DC '10'	
7	Y DC '15'	
8	END	

How LC Operates?

Sr. NO		LC
1	START 100	
2	MOVER AREG, X	100
3	MOVER BREG, Y	101
4	ADD AREG, BREG	102
5	MOVEM AREG, X	103
6	X DC '10'	104
7	Y DC '15'	105
8	END	

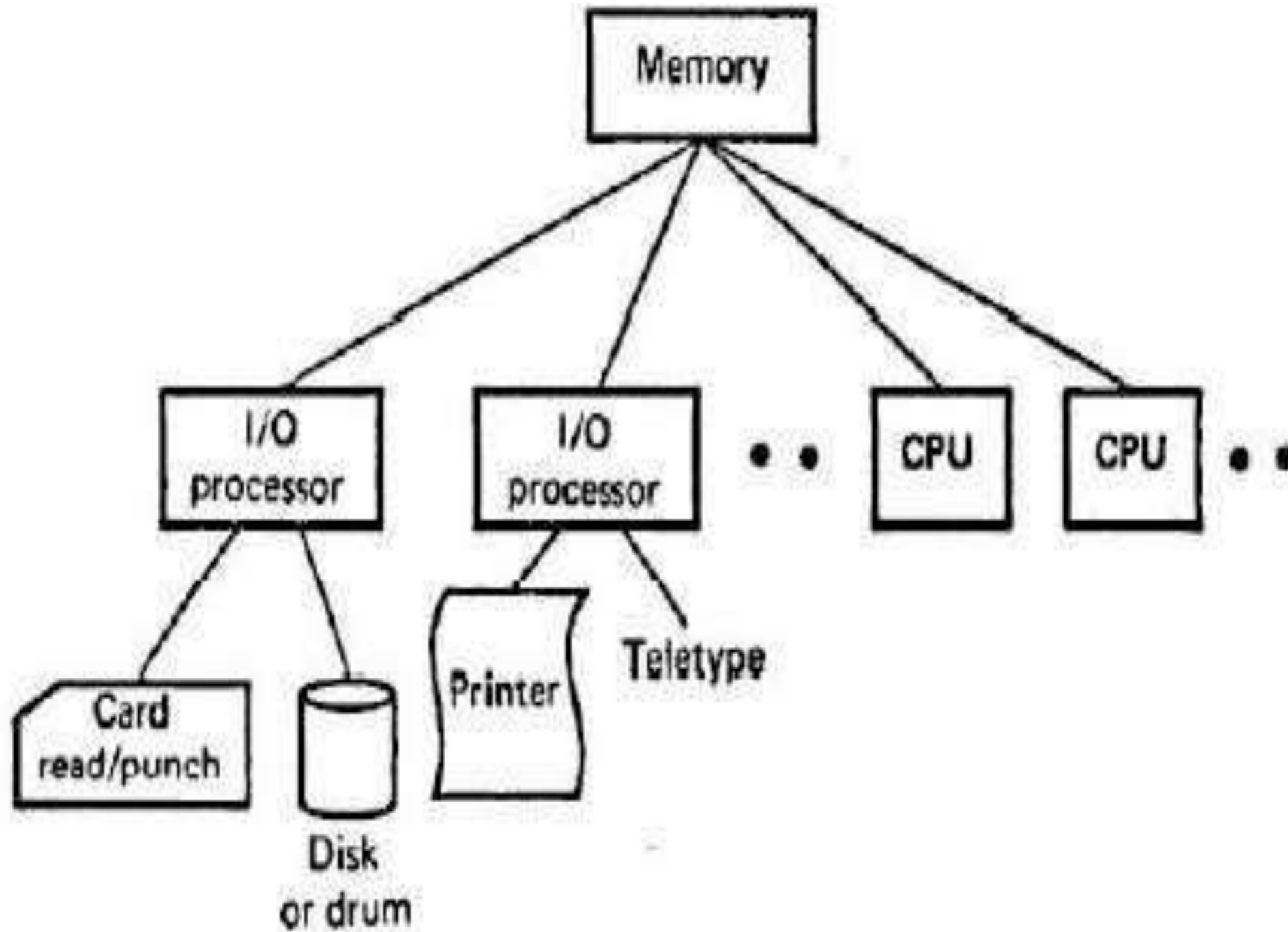
Identify symbol, literals, AD, DS, IS , Symbol, Literal Label

- START 100
- MOVER BREG, ='2'
- LOOP MOVER AREG, N
- ADD BREG, ='1'
- ORIGIN LOOP+5
- LORG
- ORIGIN NEXT +2
- LAST STOP
- N DC '5'
- END

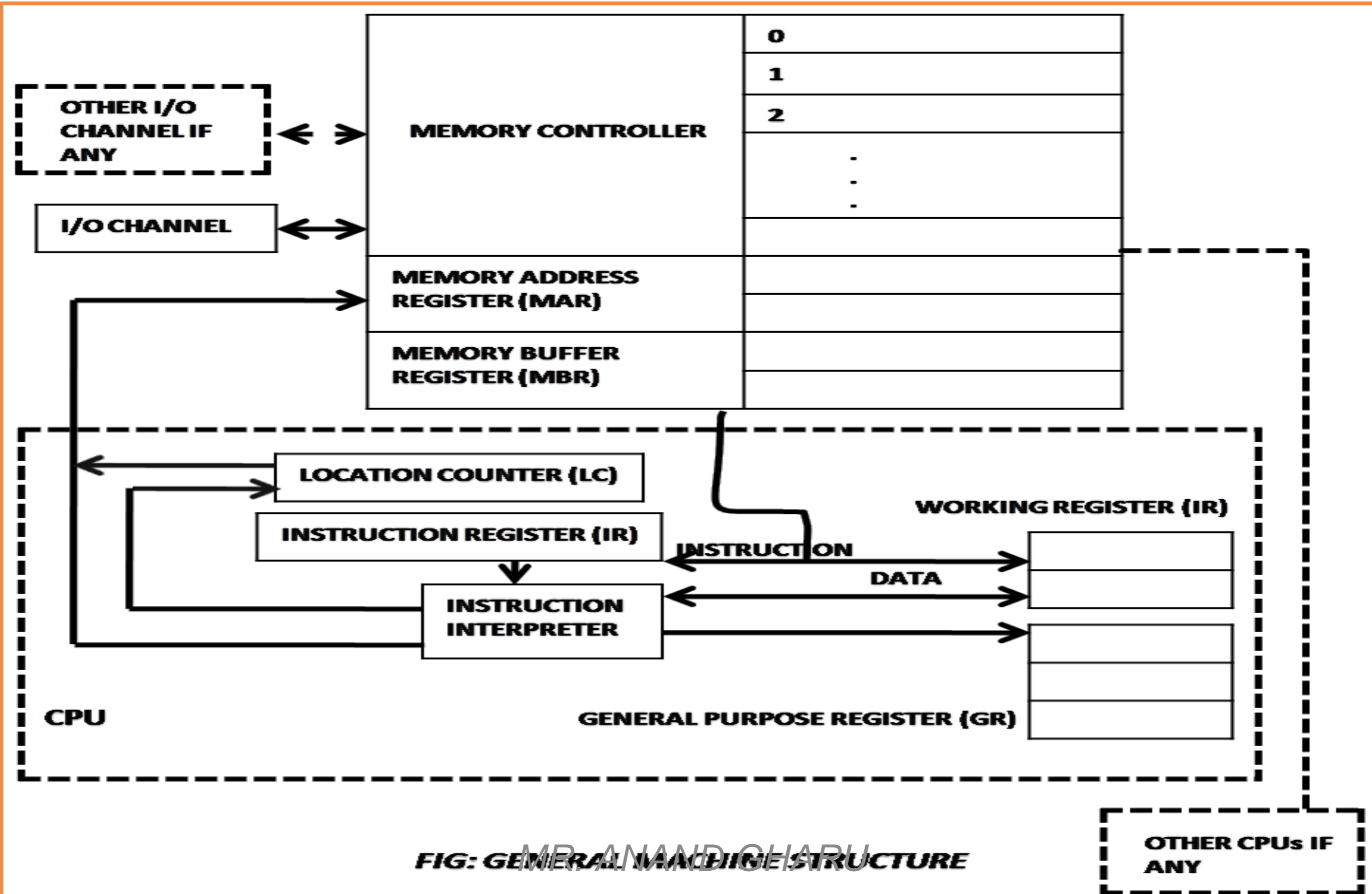
Solution (From Previous Example)

Sr. No	AD	DS	IS	Symb ol	Literal	Label
1	AD					
2			IS		=2	
3			IS	N		LOOP
4			IS		=1	
5	AD					
6	AD					
7	AD					
8			IS			LAST
9		DS				
10	AD					

Machine Structure

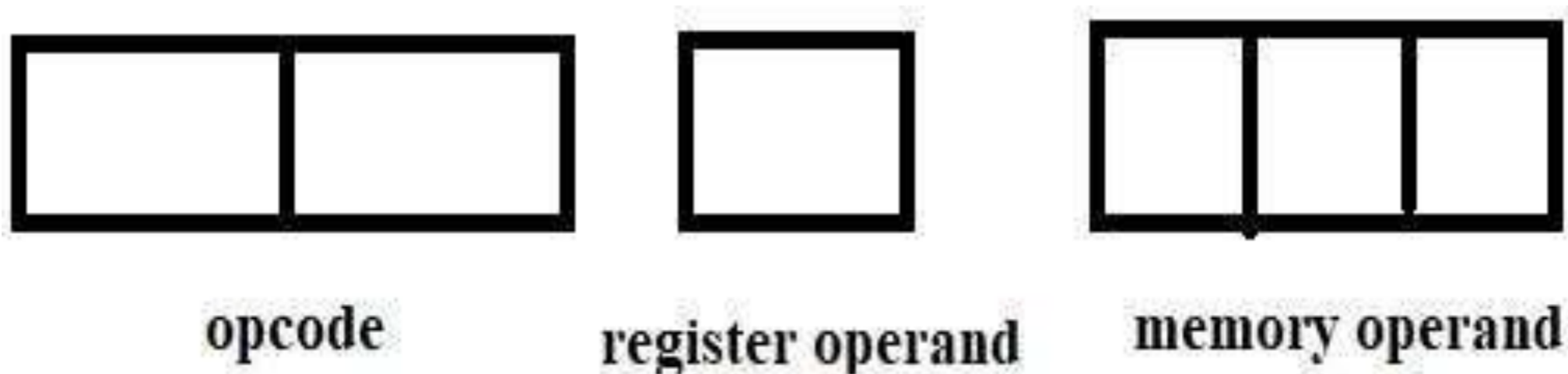


Machine Structure



- Consider any hypothetical assembly language.
- It supports three registers:
- **AREG**
- **BREG**
- **CREG**

- **Machine instruction**
Format:



- It supports 11 different OPERATIONS.
- **STOP**
- **ADD**
- **SUB**
- **MULT**
- **MOVER**
- **MOVEM**
- **COMP**
- **BC**
- **DIV**
- **READ**
- **PRINT**

- In this hypothetical machine,
- First operand is always a **CPU register**.
- Second operand is always **memory operand**.
- READ and PRINT instructions do not use **first operand**.
- The STOP instruction has **no operand**.

- Each symbolic opcode is associated with machine opcode.
- These details are stored in machine opcode table(MOT).
- MOT contains:
 - 1. Opcode in mnemonic form
 - 2. Machine code associated with the opcode.

Symbolic Opcode	Machine Code for opcode	Size of instruction (in number of words)
STOP	00	1
ADD	01	1
SUB	02	1
MULT	03	1
MOVER	04	1
MOVEM	05	1
COMP	06	1
BC	07	1
DIV	08	1
READ	09	1
PRINT	10	1

Symbolic Opcode	Machine Code for opcode
START	01
END	02
LTORG	03
ORIGIN	04
EQU	05

Sr. NO	Declarative Statement	Machine Opcode
01	DS	01
02	DC	02

MR. ANAND GHARU

Sr. No	Symbolic opcode	Machine opcode
1	AREG	01
2	BREG	02
3	CREG	03

MR. ANAND GHARU

ASSEMBLER

- An assembly language program can be translated into machine language.
- It involves following steps:
 - 1. Find addresses of variable.
 - 2. Replace symbolic addresses by numeric addresses.
 - 3. Replace symbolic opcodes by machine operation codes.
 - 4. Reserve storage for data.

Step 1

- We can find out addresses of variable using LC.
- First identify all variables in your program.
- **START 100**
- **MOVER AREG, X**
- **MOVER BREG, Y**
- **ADD AREG, X**
- **MOVEM AREG, X**
- **X DC '10'**
- **Y DC '15'**
- **END**

Step 1

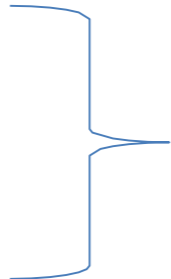
Sr. NO		LC
1	START 100	
2	MOVER AREG, X	100
3	MOVER BREG, Y	101
4	ADD AREG, X	102
5	MOVEM AREG, X	103
6	X DC '10'	104
7	Y DC '15'	105
8	END	

MR. ANAND GHARU

Sr. No	Name of Variable(Symbol)	Address
1	X	104
2	Y	105

MR. ANAND GHARU

Step2: Replace all symbolic address with numeric address.

- **START 100**
 - **MOVER AREG, 104**
 - **MOVER BREG, 105**
 - **ADD AREG, 104**
 - **MOVEM AREG, 104**
 - **X DC '10'**
 - **Y DC '15'**
 - **END**
- 
- Memory is reserved but no code is generated.

Step3: Replace symbolic opcodes by machine operation codes.

LC	Assembly Instruction	Machine Code
101	MOVER AREG, 104	04 01 104
102	MOVER BREG, 105	04 02 105
103	ADD AREG, 104	01 01 104
104	MOVEM AREG, 104	05 01 104
105		
106		
107		

Question For U

```
START 102  
READ X  
READ Y  
MOVER AREG, X  
ADD AREG, Y  
MOVEM AREG, RESULT  
PRINT RESULT  
STOP  
X DS 1  
Y DS 1  
RESULT DS 1  
END
```

Question For u

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

Assembler

- An Assembler is a translator which translates assembly language code into machine language with help of data structure.
- It has two types
- Pass 1 Assembler.
- Pass 2 Assembler.

General design procedure of assembler

- Statement of Problem
- Data Structure
- Format of databases
- Algorithms
- Look for modularity.

Statement of Problem

- We want to convert assembly language program into machine language.

Data Structure Used

- Data Structure used are as follows:
- Symbol table
- Literal Table
- Mnemonic Opcode Table
- Pool Table

Format of Databases

- Symbol Table:

Name of Symbol	address

- Literal Table:

Literal	address

- MOT:

Mnemonic	Machine Opcode	Class	Length

- Pool Table:

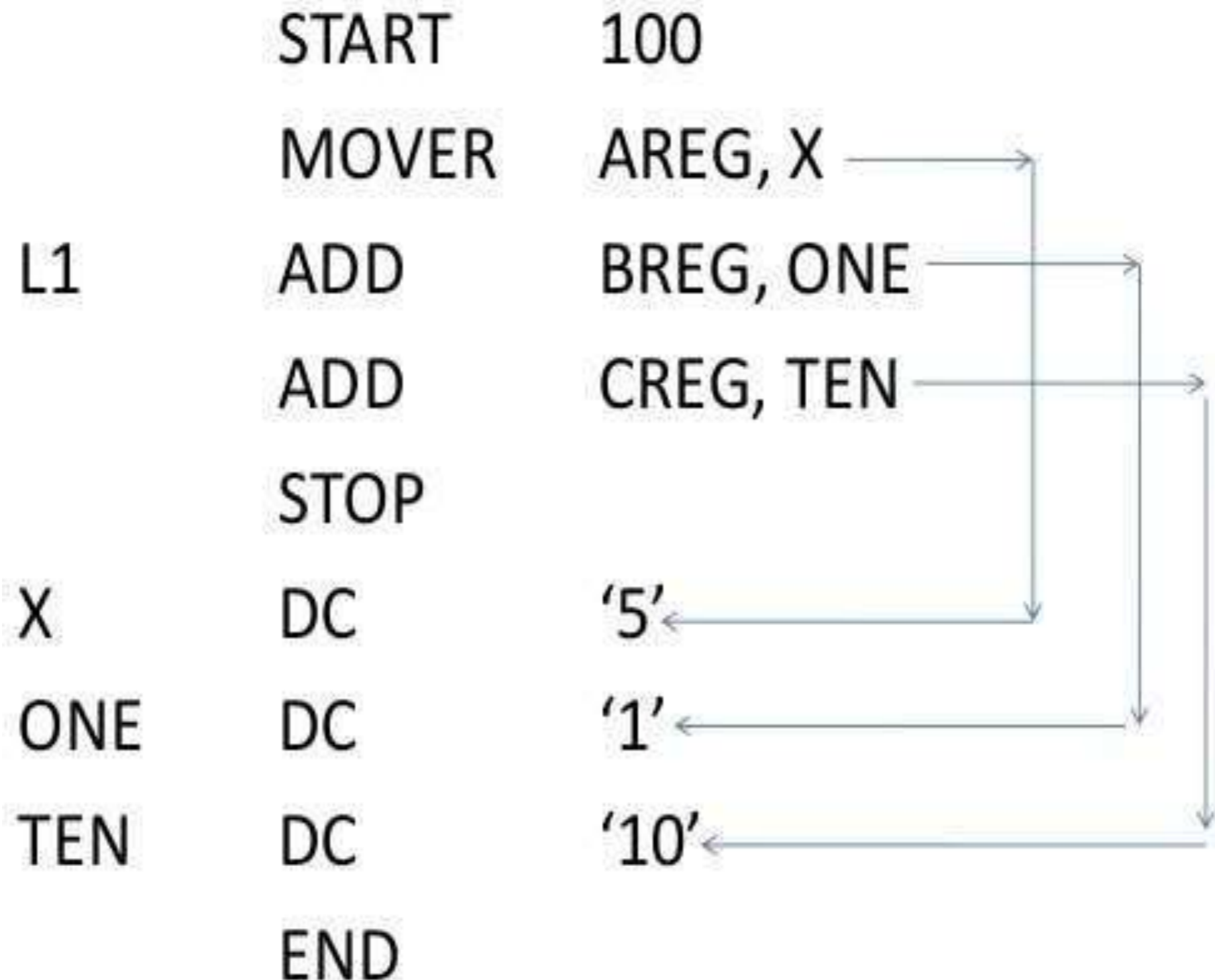
Literal Number

Forward Reference Problem

- Using a variable before its definition is called as forward reference problem.
- E.g.
- START 100
- MOVEM AREG, X
- MOVER BREG, Y
- ADD AREG, Y
- X DC '4'
- Y DC '5'
- END

- In example variable X, Y are making forward reference.
- So, We can solve it by using back patching.

Consider another example



Apply LC

	START	100	
	MOVER	AREG, X	100
L1	ADD	BREG, ONE	101
	ADD	CREG, TEN	102
	STOP		103
X	DC	'5'	104
ONE	DC	'1'	105
TEN	DC	'10'	106
	END		

Try to convert into machine code

	START	100			
	MOVER	AREG, X	100	04	1 ----
L1	ADD	BREG, ONE	101	01	2 ----
	ADD	CREG, TEN	102	06	3 ----
	STOP		103	00	0 000
X	DC	'5'	104		
ONE	DC	'1'	105		
TEN	DC	'10'	106		
	END				

Backpatching

- The operand field of instruction containing a forward reference is left blank initially.
- Step 1: Construct TII(Table of incomplete instruction)

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

- **Step 2** : After encountering END statement symbol table would contain the address of all symbols defined in the source program.

SYMBOL	NAME	ADDRESS
	X	104
	ONE	105
	TEN	106

- Now we can generate machine code...

04	1	104
01	2	105
01	03	106
00	0	000

Pass 1 Assembler

- **Pass 1 assembler** separate the labels , mnemonic opcode table, and operand fields.
- Determine storage requirement for every assembly language statement and update the location counter.
- Build the symbol table. Symbol table is used to store each label and each variable and its corresponding address.
- **Pass 2 Assembler:** Generate the machine code

How pass 1 assembler works?

- Pass I uses following data structures.
 - 1. Machine opcode table.(MOT)
 - 2. Symbol Table(ST)
 - 3. Literal Table(LT)
 - 4. Pool Table(PT)
- Contents of MOT are fixed for an assembler.

Observe Following Program

```
START 200
MOVER AREG, ='5'
MOVEM AREG, X
L1  MOVER BREG, ='2'
    ORIGIN L1+3
    LTOrg

NEXT ADD AREG, ='1'
     SUB BREG, ='2'
     BC LT, BACK
     LTOrg

BACK EQU L1
ORIGIN NEXT+5
MULT CREG, ='4'
STOP
X DS 1
END
```

Apply LC

START 200

MOVER AREG, ='5' 200

MOVEM AREG, X 201

L1 MOVER BREG, ='2' 202

ORIGIN L1+3

LTORG

= '5' 205

= '2' 206

NEXT ADD AREG, ='1' 207

SUB BREG, ='2' 208

BC LT, BACK 209

LTORG

= '1' 210

= '2' 211

BACK EQU L1

ORIGIN NEXT+5

MULT CREG, ='4' 212

STOP 213

X DS 1 214

END

= '4'

215 MR. ANAND GHARU

Construct Symbol table

index	Symbol Name	Address
0	X	214
1	L1	202
2	NEXT	207
3	BACK	202

Construct Literal Table

index	Literal	Address
0	5	205
1	2	206
2	1	210
3	2	211
4	4	215

Pool Table.

- Pool table contains starting literal(index) of each pool.

Literal number
0
2
4

NOW CONSTRUCT INTERMEDIATE CODE/MACHINE CODE

- For constructing intermediate code we need MOT.

Variant I

Declaration statement	Instruction code
DC	01
DS	02

Assembler directive	Instruction code
START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

Imperative statements (mnemonics)	Instruction code
STOP	00
ADD	01
SUB	02
MULT	03
MOVER	04
MOVEM	05
COMP	06
BC	07
DIV	08
READ	09
PRINT	10
JUMP	11

Variant I

Condition	Instruction code
LT	1
LE	2
EQ	3
GT	4
GE	5
ANY	6

Register	Instruction code
AREG	1
BREG	2
CREG	3
DREG	4

- First operand is represented by a single digit number which is a code for a register or the condition code
- The second operand, which is a memory operand, is represented by a pair of the form (operand class, code)
- Where operand class is one of the C, S and L standing for constant, symbol and literal. For a constant, the code field contains the internal representation of the constant itself. Ex: the operand descriptor for the statement START 200 is (C,200). For a symbol or literal, the code field contains the ordinal number of the operand's entry in SYMTAB or LITAB

Enhanced Machine opcode Table

Table 1.10.1 : An enhanced machine opcode table (MOT)

	Mnemonic opcode	Class	Opcode	Length
0	STOP	IS	00	1
1	ADD	IS	01	1
2	SUB	IS	02	1
3	MULT	IS	03	1
4	MOVER	IS	04	1
5	MOVEM	IS	05	1
6	COMP	IS	06	1
7	BC	IS	07	1
8	DIV	IS	08	1
9	READ	IS	09	1
10	PRINT	IS	10	1
11	START	AD	01	-
12	END	AD	02	-
13	ORIGIN	AD	03	-
14	EQU	AD	04	-
15	LTORG	AD	05	-
16	DS	DL	01	-
17	DC	DL	02	1
18	AREG	RG	01	-
19	BREG	RG	02	-
20	CREG	RG	03	-
21	EQ	PC	01	-

Mnemonic opcode	Class	Opcode	Length
LT	CC	02	-
GT	CC	03	-
LE	CC	04	-
GE	CC	05	-
NE	CC	06	-
ANY	CC	07	-

Example No.2

```
START 205
MOVER AREG, ='6'
MOVEM AREG, A
LOOP  MOVER AREG, A
      MOVER CREG, B
      ADD CREG, ='2'
      BC ANY , NEXT
      LTOrg
      ADD BREG, B
NEXT  SUB AREG, ='1'
      BC LT, BACK
LAST  STOP
      ORIGIN LOOP+2
      MULT CREG, B
      ORIGIN LAST+1
A     DS      1
BACK  EQU     LOOP
B     DS      1
END
```

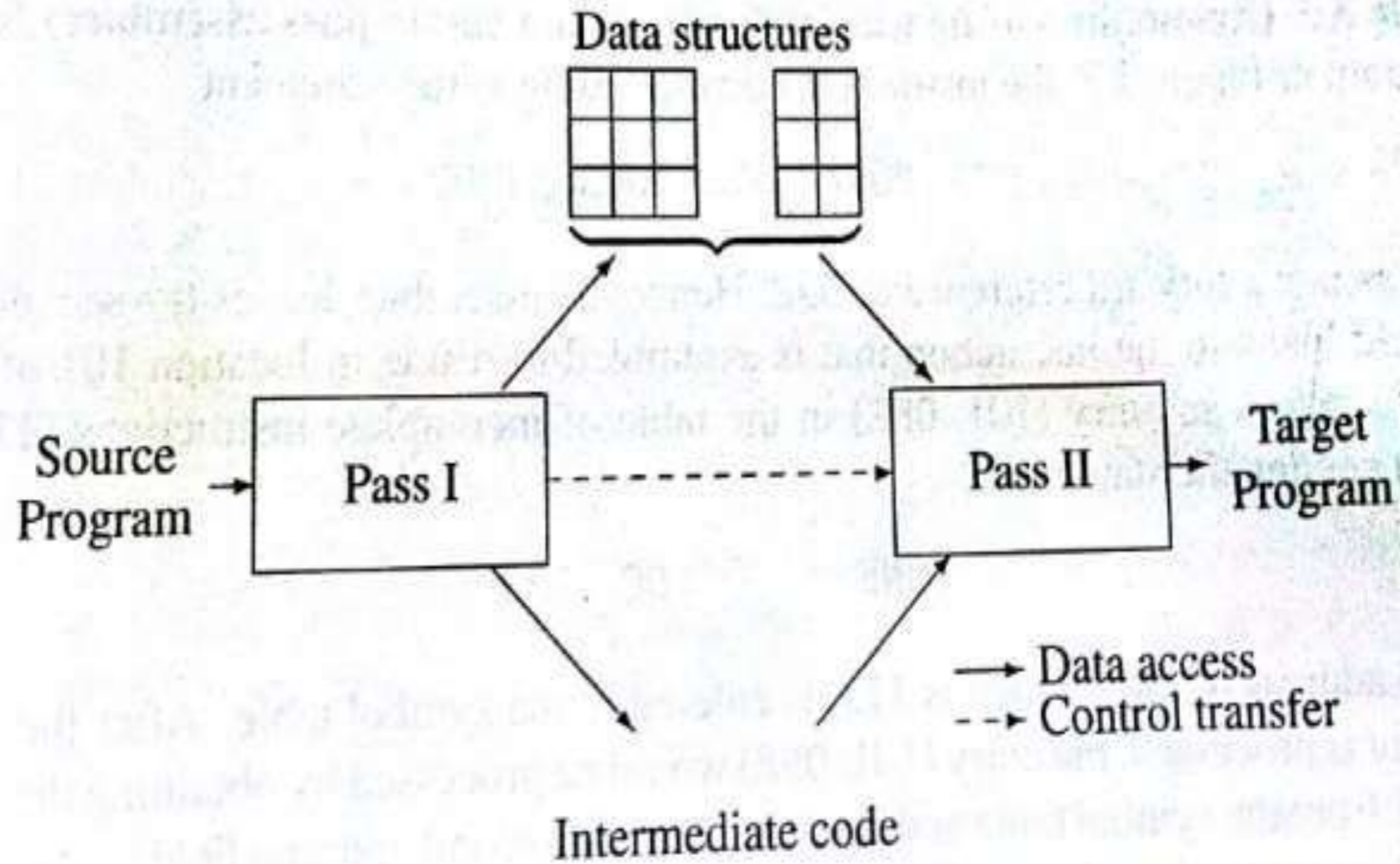
MR. ANAND GHARU

- PASS 2 assembler requires two scans of program to generate machine code.
- It uses data structures defined by pass 1. like symbol table, MOT, LT.

Design of two pass assembler

- Tasks performed by the passes of a two pass assembler are as follows:
- **Pass 1:**
 1. Separate the symbol, mnemonic opcode, and operand fields.
 2. Build the symbol table.
 3. Perform LC processing.
 4. Construct intermediate representation(or IC).
- **Pass 2:**
 1. Synthesize the target program.

Two Pass Assembler



Analysis Phase Vs. Synthesis Phase

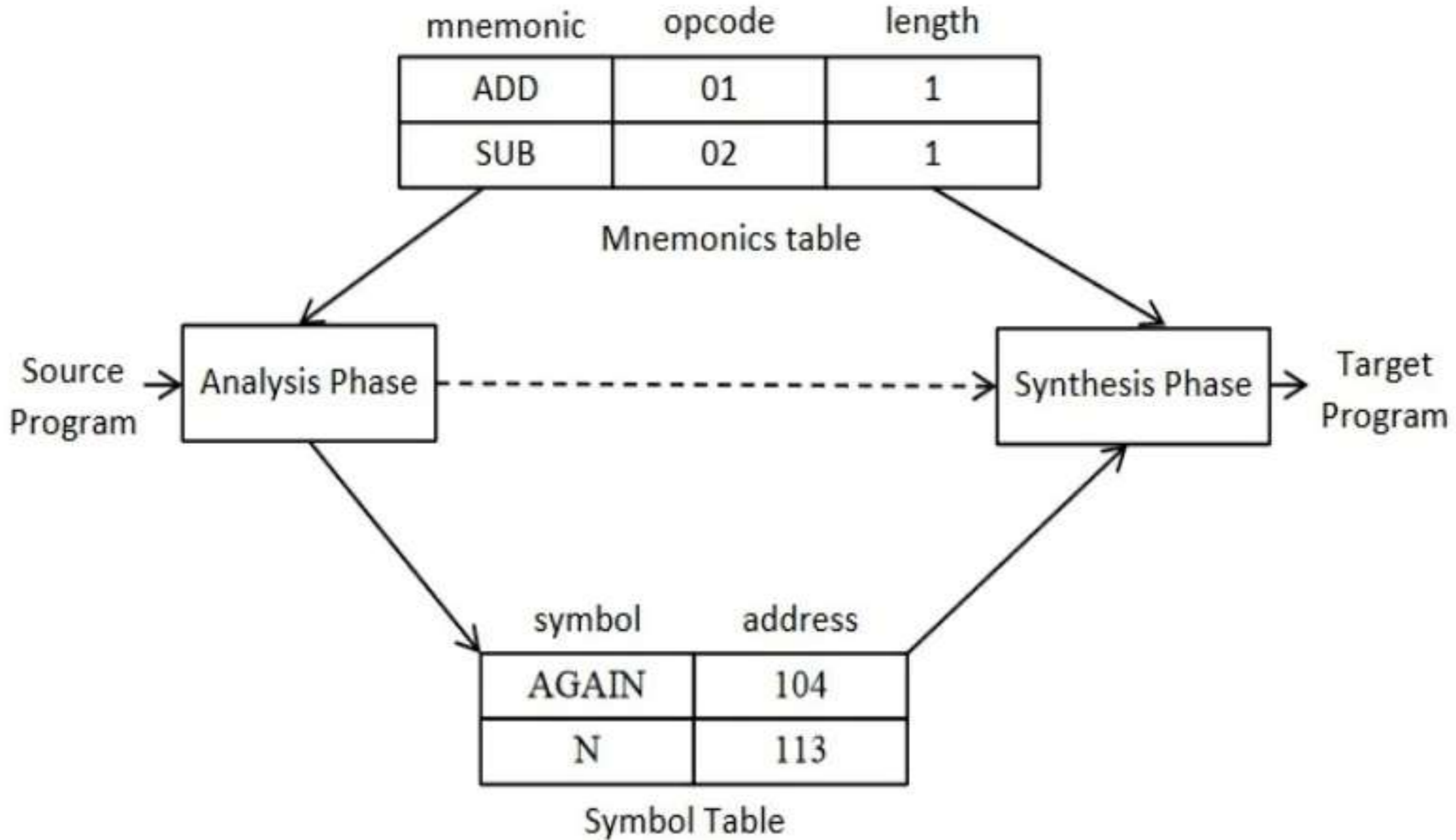


Fig.-Design of assembler

Comparison between Pass 1 and Pass2

Sr. No	Pass 1	Pass 2
01	It requires only one scan to generate machine code	It requires two scan to generate machine code.
02	It has forward reference problem.	It don't have forward reference problem.
03	It performs analysis of source program and synthesis of the intermediate code.	It process the IC to synthesize the target program.
04	It is faster than pass 2.	It is slow as compared to pass 1.

Pass 1 output and pass 2 output

- Pass 1 assembler generates **Intermediate code.**
- Pass 2 assembler generates **Machine code.**

INTERMEDIATE CODE

MR. ANAND GHARU


INTERMEDIATE CODE

- **Format for intermediate code:**
- For every line of assembly statement, one line of intermediate code is generated.
- Each mnemonic field is represented as
- **(statement class, and machine code)**

- Statement class can be:
- 1. IS
- 2. DL/DS
- 3. AD

• **E.g. MOVER AREG, X**


Mnemonic field


Operand field

- So, IC for mnemonic field of above line is,
- **(statement class, machine code)**
- (IS, 04)from MOT

- Operand Field:
- Each operand field is represented as

**(operand class,
reference)**

- The operand class can be:
- 1. C: Constant
- 2. S: Symbol
- 3. L: Literal
- 4. RG: Register
- 5. CC: Condition codes

- E.g. MOVER AREG, X
- **For a symbol or literal the reference field contains the index of the operands entry in symbol table or literal table.**
-
- So IC for above line is:
- **(IS, 04) (RG, 01) (S, 0)**

- For example...
- START 200
- IC: (AD, 01) (C, 200)

Consider following example

START 200

MOVER AREG, ='5' 200

MOVEM AREG, X 201

L1 MOVER BREG, ='2' 202

ORIGIN L1+3

LTORG

= '5' 205

= '2' 206

NEXT ADD AREG, ='1' 207

SUB BREG, ='2' 208

BC LT, BACK 209

LTORG

= '1' 210

= '2' 211

BACK EQU L1

ORIGIN NEXT+5

MULT CREG, ='4' 212

STOP 213

X DS 1 214

END

= '4'

215 MR. ANAND GHARU

Symbol Table and Literal Table

index	Symbol Name	Address
0	X	214
1	L1	202
2	NEXT	207
3	BACK	202

index	Literal	Address
0	5	205
1	2	206
2	1	210
3	2	211
4	4	215

Intermediate Code

(AD, 01) (C, 200)

200 (IS, 04) (RG,01) (L, 0)

201 (IS, 05) (RG,01) (S,0)

202 (IS, 04) (RG,02) (L,1)

203 (AD, 03) (C, 205)

205 (DL, 02) (C,5)

206 (DL, 02) (C, 2)

207 (IS,01) (RG, 01) (L, 2)

208 (IS, 02) (RG, 02) (L,3)
209 (IS, 07) (CC, 02) (S, 3)
210 (DL,02) (C,1)
211 (DL,02) (C,2)
212 (AD, 04) (C, 202)
212 (AD, 03) (C, 212)
212 (IS, 03) (RG, 03)(L, 4)
213 (IS, 00)

214 (DL, 01, C, 1)

215 (AD, 02)

215 (DL, 02) (C,4)

Intermediate Code and Machine code

I.C	LC	Machine Code
(AD, 01) (C, 200)		
(IS, 04) (RG,01) (L, 0)	200	04 01 205
(IS, 05) (RG,01) (S,0)	201	05 01 214
(IS, 04) (RG,02) (L,1)	202	04 02 206
(AD, 03) (C, 205)	203	
(DL, 02) (C,5)	205	00 00 005
(DL, 02) (C, 2)	206	00 00 002
(IS,01) (RG, 01) (L, 2)	207	01 01 210

MR. ANAND GHARU

I.C	LC	Machine Code
(IS, 02) (RG, 02) (L,3)	208	02 02 211
(IS, 07) (CC, 02) (S, 3)	209	07 02 202
(DL,02) (C,1)	210	00 00 001
(DL,02) (C,2)	211	00 00 002
(AD, 04) (C, 202)	212	
(AD, 03) (C, 212)	212	
(IS, 03) (RG, 03)(L, 4)	212	03 03 215
(IS, 00)	213	00 00 000

I.C	LC	Machine Code
(DL, 01, C, 1)	214	
(AD, 02)	215	
(DL, 02) (C,4)	215	00 00 004

MR. ANAND GHARU

Variants of Intermediate Code.

- There are two variants of I.C.:
- Variant I
- Variant II.

Variant I

- In Variant I, each operand is represented by a pair of the form (operand class, code).
- The operand class is one of:
 - 1. S for symbol**
 - 2. L for literal**
 - 3. C for constant**
 - 4. RG for register.**

Variant

	START	100	(AD, 01) (C, 100)
L1	READ	A	(IS, 09) (S, 1)
	SUB	AREG, = '5'	(IS, 02) (RG, 01) (L, 0)
	BC	GT, L1	(IS, 07) (CC, 03) (S, 0)
	STOP		(IS, 00)
A	DS	1	(DL, 01) (C, 1)
	-		-
	-		-
	-		-

Variant II

- In variant II, operands are processed selectively.
- Constants and literals are processed. Symbols, condition codes and CPU registers are not processed.

Variant II

A sample intermediate code

Fig. 1.10.6.

	START	100	(AD, 01) (C, 100)
L1	READ	A	(IS, 09) A
	SUB	AREG, = '5'	(SI, 02) AREG, (L, 0)
	BC	GT, L1	(IS, 07) GT, L1
	STOP		(SI, 00)
A	DS	1	(DL, 01) (C, 1)
	-		-
	-		-
	-		-

Fig. 1.10.6 : Intermediate code using variant II

Error Reporting

- An assembly program may contain errors.
- It may be necessary to report these errors effectively.
- Some errors can be reported at the end of the source program.
- Some of the typical programs include:
 - Syntax errors like missing commas...
 - Invalid opcode
 - Duplicate definition of a symbol.
 - Undefined symbol
 - Missing START statement.

Example

- START 100
- MOVER AREG, X
- ADDER BREG, X
- ADD AREG, Y
- X DC '2'
- X DC '3'
- Z DC '3'
- END

- START 100
- MOVER AREG, X
- **ADDER BREG, X** **Invalid opcode**
- **ADD AREG, Y** **Undefined symbol Y**
- **X DC '2'**
- **X DC '3'** **duplicate definition of Symbol X.**
- **Z DC '3'**
- **END**

PASS-1 OF TWO PASS ASSEMBLER

```
LC = 0; // Location counter loop until the end of the program  
{
```

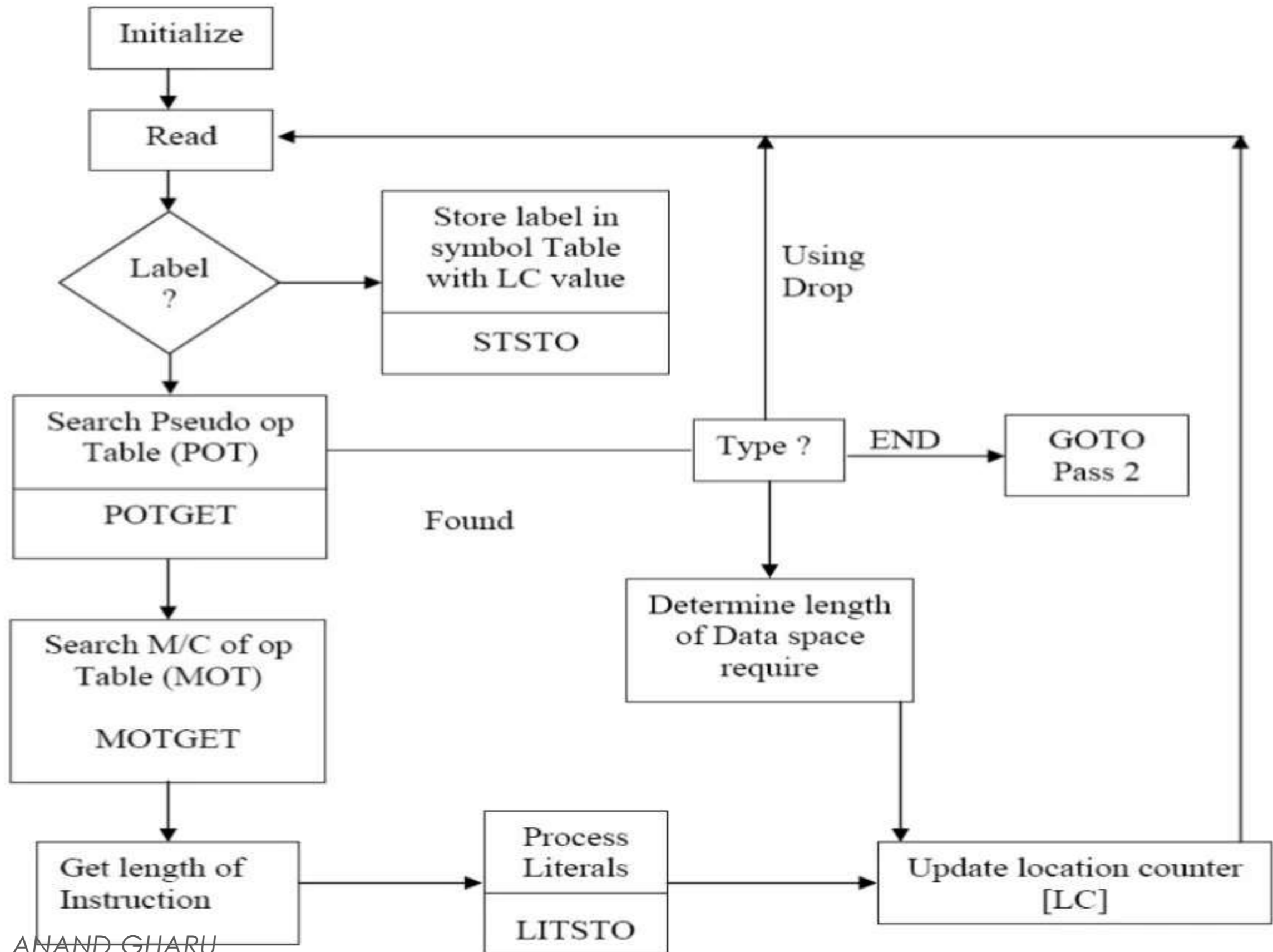
1. Read in a line of assembly code.
 - a. If there is label then insert the label with LC in the symbol table.
 - b. If assembler directive then process it
 - c. If executable instruction then generate machine code. If the instruction contains a symbol with forward reference then enter the same in the Backpatch list.

```
}
```

Process the Backpatch list.

NOTE : Draw pass-1 and pass-2 algorithm and flowchart in your own understanding.

PASS-1 OF TWO PASS ASSEMBLER

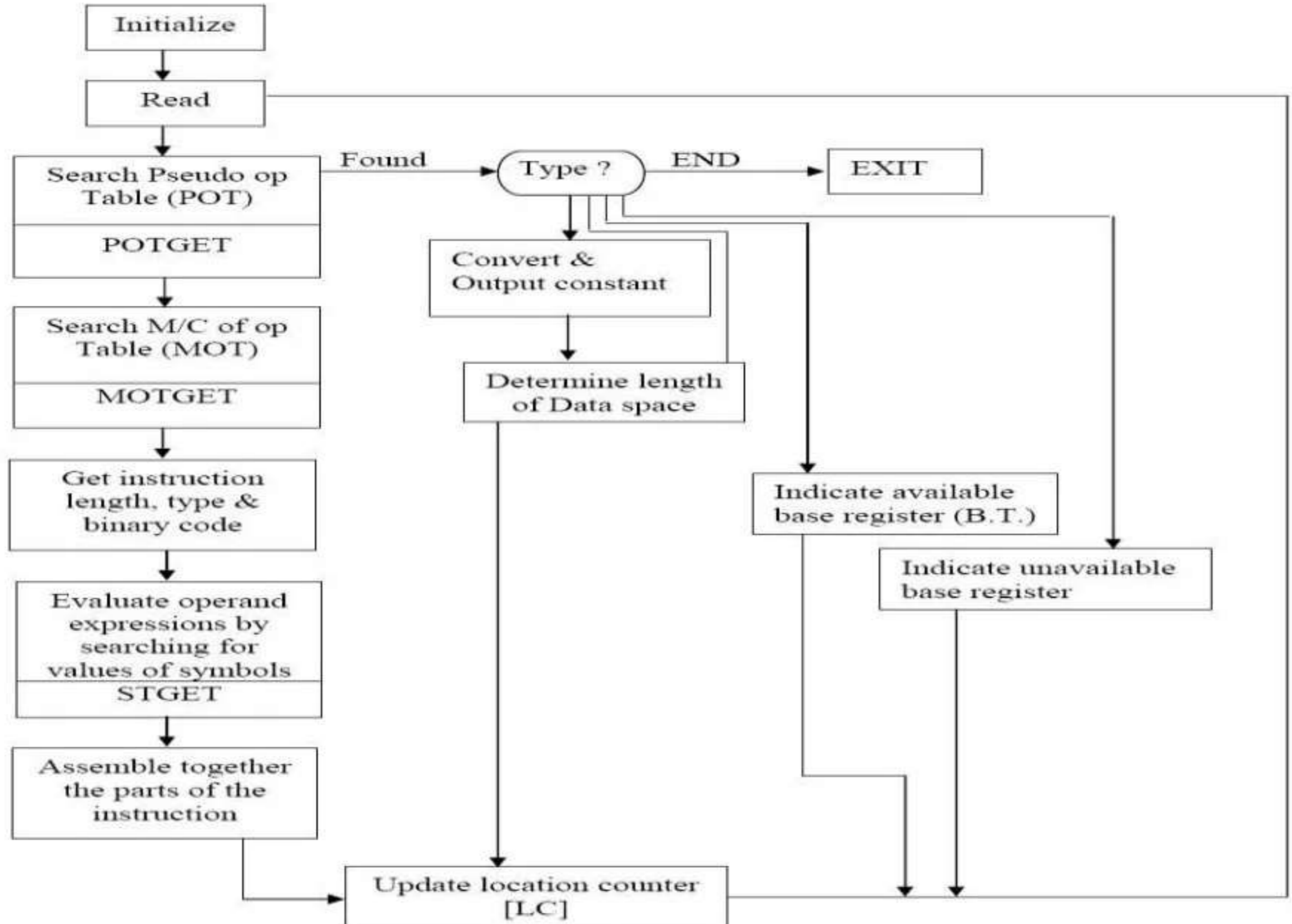


PASS-2 OF TWO PASS ASSEMBLER

Algorithm

1. Code_area_address = address of Code_area
2. For each entry in IC[]
 - {
 - a) If an imperative statement
 - (i) Read LC
 - (ii) Get opcode
 - (iii) Get operand / literal address from the symbol / literal table.
 - (iv) Assemble instruction in machine_code_buffer.
 - (v) Move contents of machine_code_buffer in code_area at the address LC + code_area_address.
 - b) If a DC statement then
 - (i) Read LC
 - (ii) Assemble the constant in machine_code_buffer.
 - (iii) Move contents of machine_code_buffer in code_area at the address LC + code_area_address.
3. Write code_area into output file.

PASS-2 OF TWO PASS ASSEMBLER



THANK YOU!!!

My Blog : <https://anandgharu.wordpress.com/>

Email : gharu.anand@gmail.com