MIT-WPU
T.Y. B.Tech

System Software and Compilers

Course Objective & Course Outcomes

Course Objectives

1. To demonstrate the fundamentals of translator.
2. To explain preprocessing , linking and loading concepts.
3. To construct a scanner for high level languages.
4. To analyze the process of parsing and code generation for high level languages.

Course Outcomes

Students will be able:

1. To analyze and synthesize a translator.
2. To develop a preprocessor, linker and loader schemes.
3. To build a scanner using the LEX tool for any high-level language.
4. To make use of the YACC tool for compiler design.

Text Books & Reference Books

Text Books:

1. Dhamdhere D., "Systems Programming and Operating Systems", McGraw Hill, ISBN 0 - 07 -463579 – 4.
2. A V Aho, R Sethi, J D Ullman, \Compilers: Principles, Techniques, and Tools", Pearson Edition, ISBN 81-7758-590-8.
3. John Donovan, "System Programming", McGraw Hill, ISBN 978-0--07-460482-3.

Reference Books:

1. John. R. Levine, Tony Mason and Doug Brown, "Lex and Yacc", O'Reilly, 1998, ISBN: 1- 56592-000-7.
2. Leland L. Beck, "System Software An Introduction to Systems Programming" 3rd Edition, Person Education, ISBN 81-7808-036-2.
3. Adam Hoover, "System Programming with C and Unix", Pearson,2010

Unit I

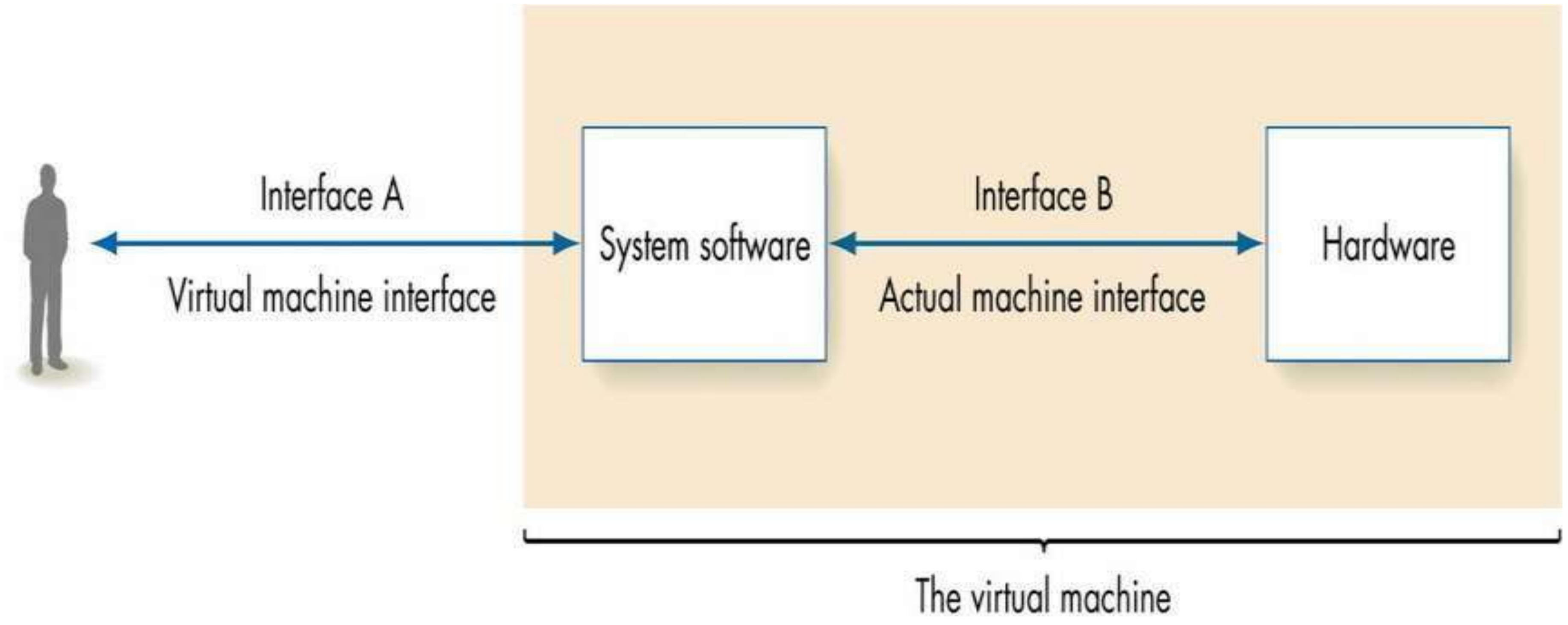
Introduction to System Software and Assembler Design

- Need and Components of system software:
- Assembler, Compiler, Interpreter,
- Macro processor, Linker, Loader,
- debugger, text editor,
- Microservices and containers.
- **Assembler**: Elements of Assembler language programming,
- Machine dependent and machine independent assembler features,
- Design of 2 pass Assembler.

System Software

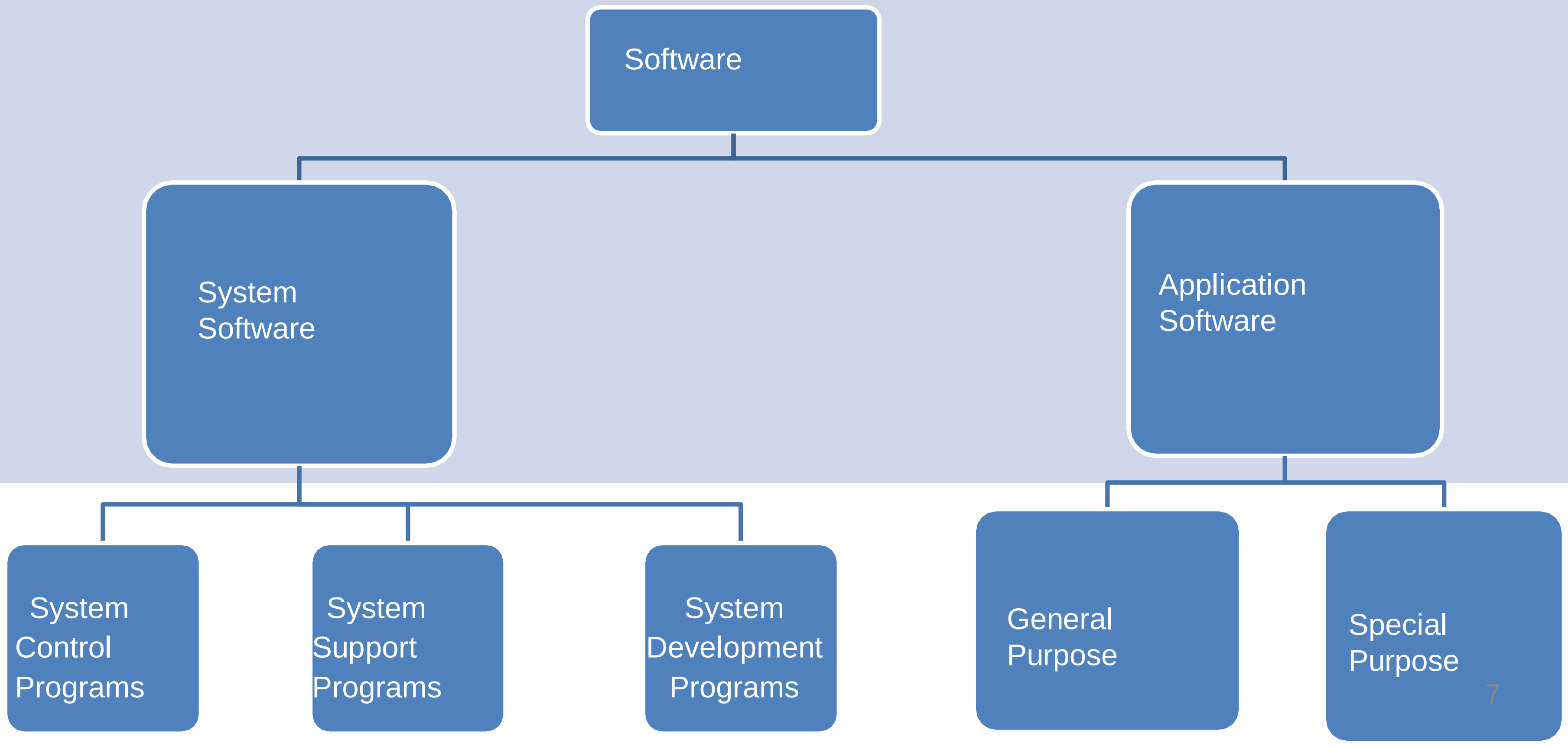
- Collection of programs
  - Designed to
    - Operate
    - Control
    - Extend the processing capabilities of the computer itself.
- Prepared by computer manufacturers.
  - Perform functions
    - File editing,
    - Storage management,
    - Resource accounting,
    - I/O management, etc.

Role of System Software



Introduction to System Software

Software is a set of computer programs which are designed and developed to perform specific task desired by the user or by the computer itself.



Types of System Software

1. **System Control Programs :**
   - They control the execution of programs
   - Manage the storage and processing resources of the computer
   - Perform other management and monitoring functions.
   - e.g., OS

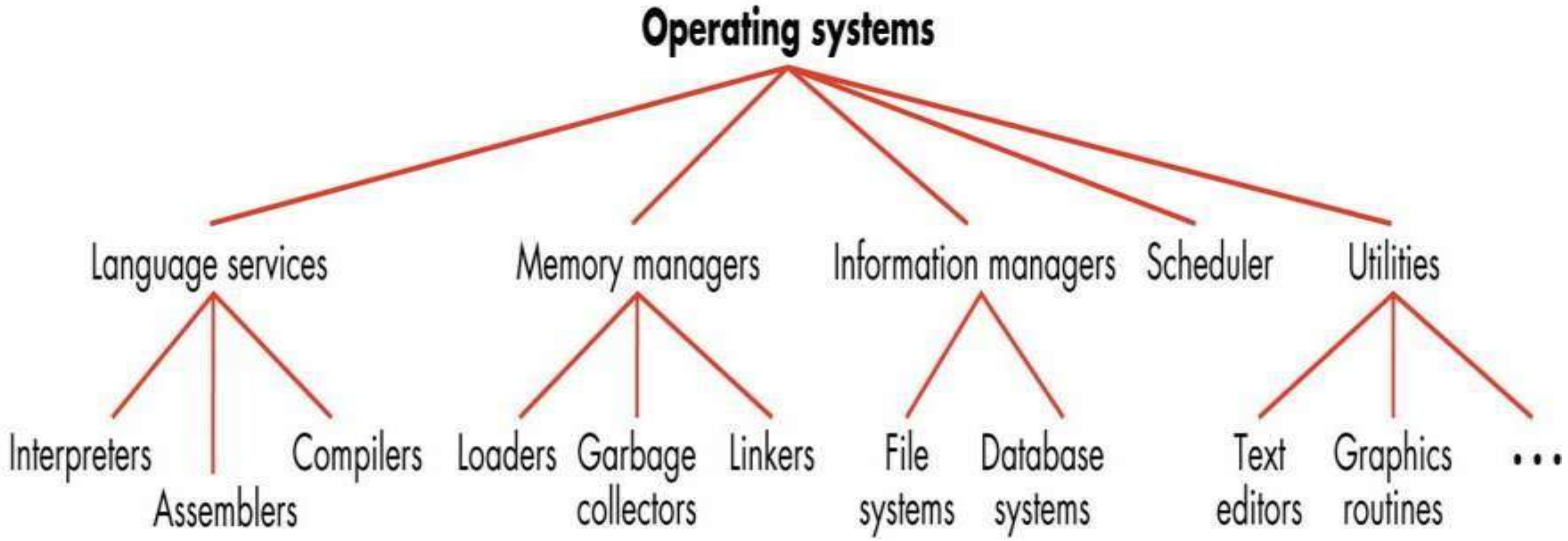
2. **System Support Programs :**
   - Provide routine service functions to other computer Programs and computer users.
   - e.g.Utility Programs

3. **System Development Programs :**
   - Assist in the creation of publication programs.

   e.g., Language translators like interpreters

System Control Programs-OS

- An OS is an integrated set of specialized programs that are used to manage overall resources of and operations of the computer.



System Development Programs-Language Translators

- Language translators are also called language processors.

- **Main functions** are :
- Translate high level language to low level language.
- Check for and identity syntax errors

- There are **3 types of translator programs-**

1. Assembler
2. Interpreter
3. Compiler

- An assembly language is a programming language that can be used to directly tell the computer what to do.
- Machine code, consisting of machine language instructions, is a low-level programming language used to directly control a computer's central processing unit (CPU).

Assemblers

Assembly Lang. Program ----------------□ Machine Language Program

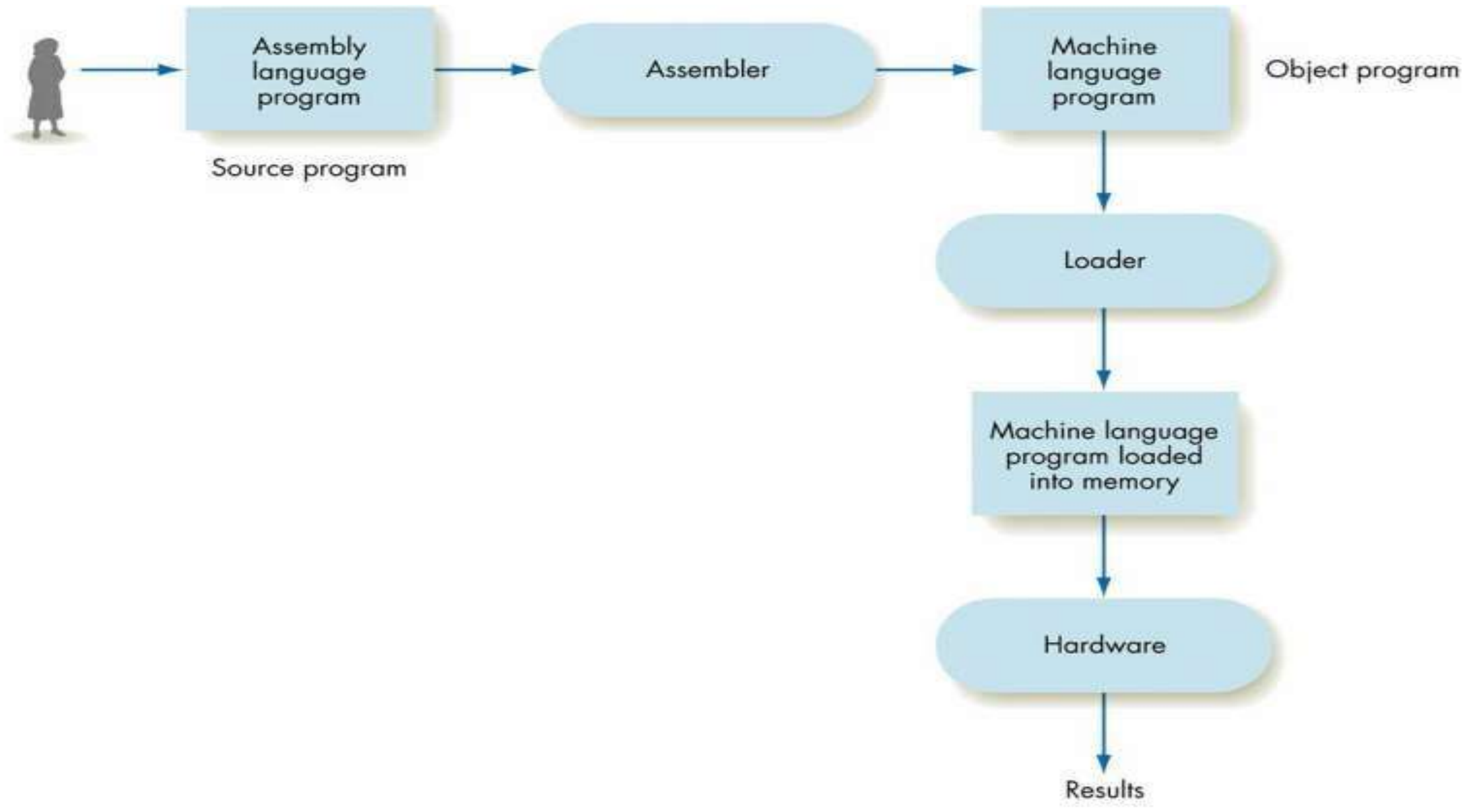
- Programs known as assembler is used to translate assembly language into machine language.
- The input to an assembler is called the source program and the output is a machine language translation(object program).

- **Assembler tasks:**

  - Convert symbolic op codes to binary
  - Convert symbolic addresses to binary
  - Perform assembler services requested by the pseudo-ops
  - Put translated instructions into a file for future use

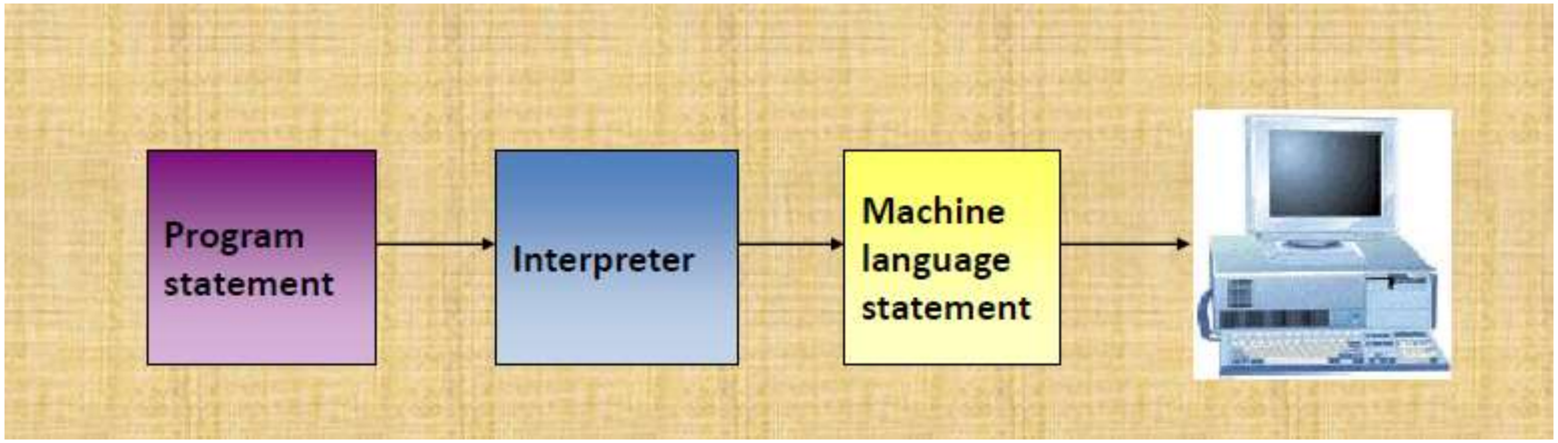
26TH March 2021

Assembler Task



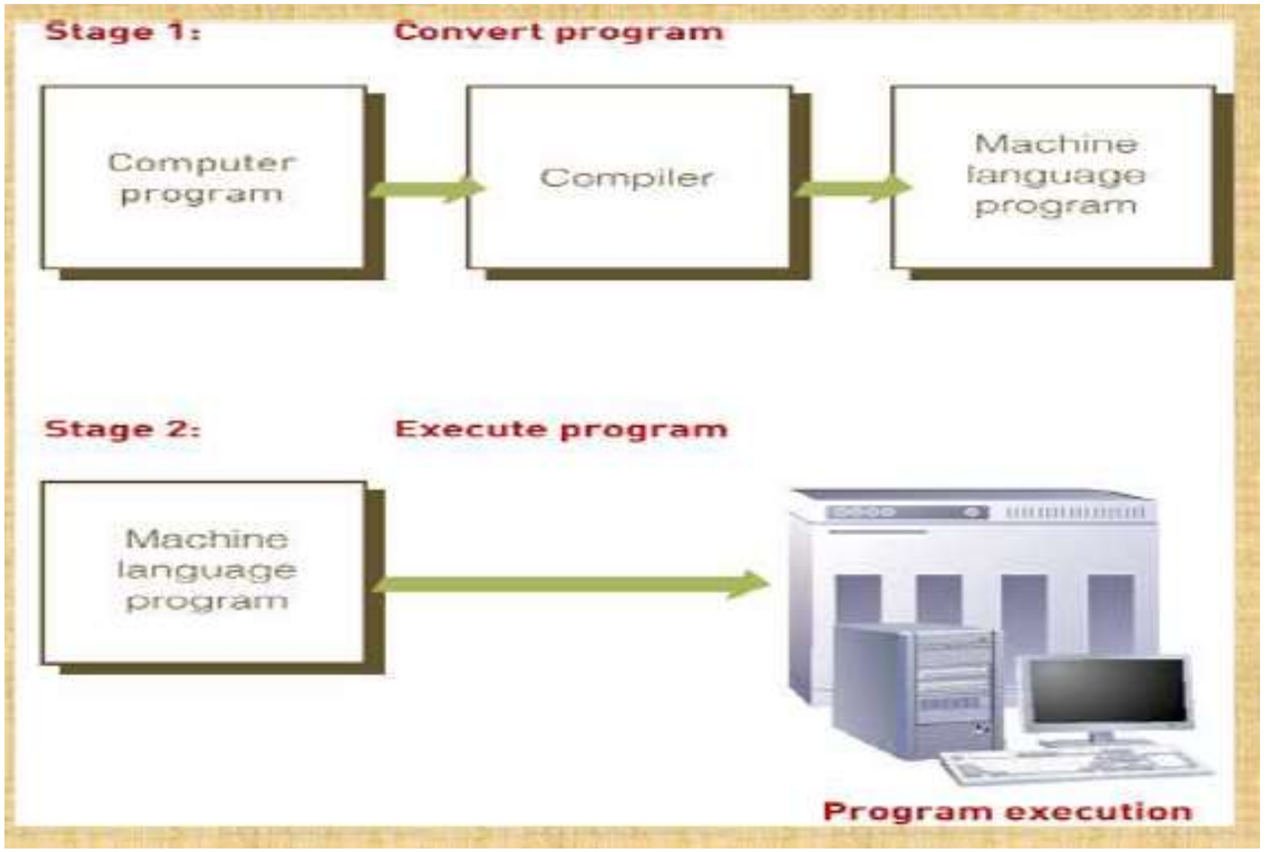
Interpreter

- A language translator that translates one program statement at a time into machine code.



Compiler

A language translators that converts a complete
Program into machine language to produce a program that the computer can
Process.



Macro Processor

- It permits the programmer to define an abbreviation for a part of his program and use the abbreviation in program.

- [A macro processor enables you to define and to use macros in your assembly programs. When you define a macro, you provide text (usually assembly code) that you want to associate with a macro name. Then, when you want to include the macro text in your assembly program, you provide the name of the macro.]

Linkers

- Linker is a program in a system which helps to link a object modules of program into a single object file. It performs the process of linking.

- Linker are also called link editors.

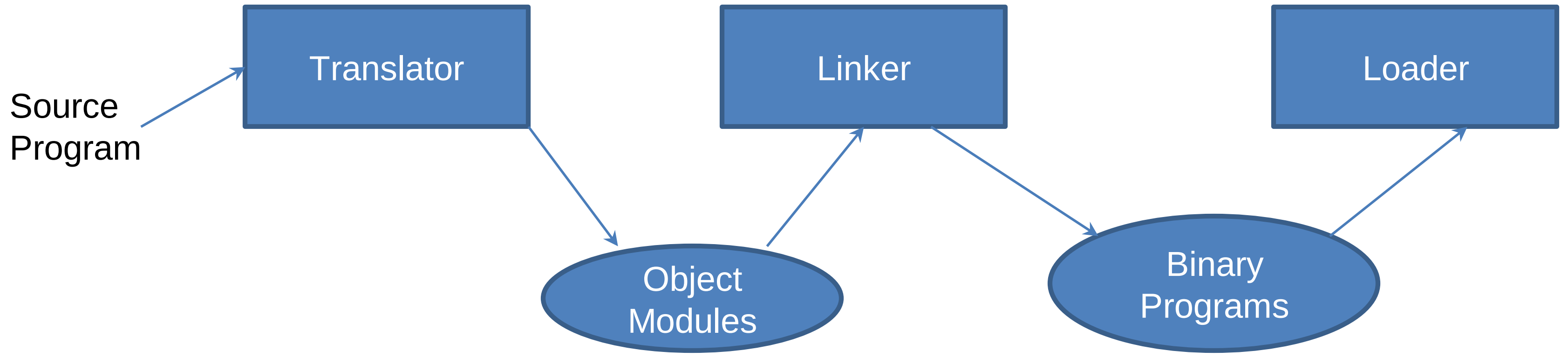
- Linking is process of collecting and maintaining piece of code and data into a single file.

- It takes object modules from assembler as input and forms an executable file as output for loader.

Linkers

Steps in program Execution

- Translation
- Linking
- Relocation
- Loading



Translated Address (Translated origin)

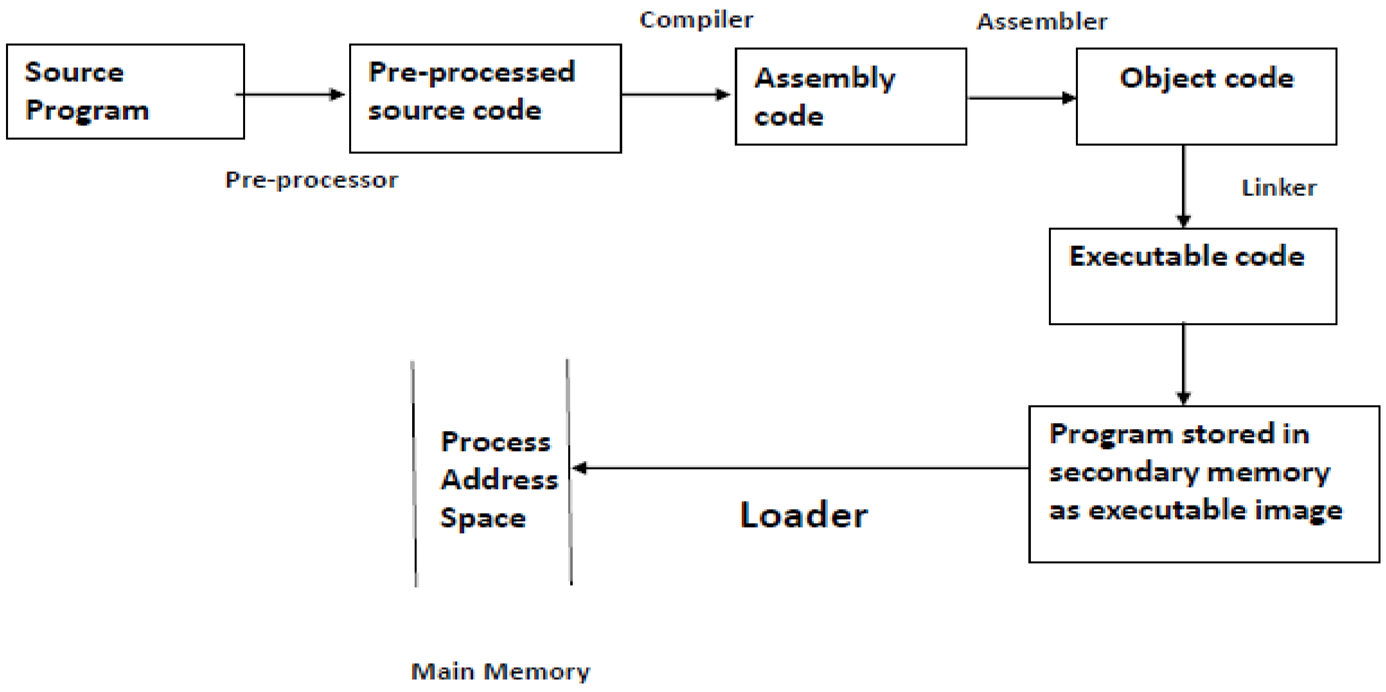
Linked Address (Linked origin)

Load Time Address (Load origin)

Loader

- Performs the loading function.
- Process of placing the program into memory for execution
- Responsible for initiating the execution of the process

Loader


Source Program
Pre-processor
Pre-processed source code
Compiler
Assembly code
Assembler
Object code
Linker
Executable code
Program stored in secondary memory as executable image
Loader
Process Address Space
Main Memory

Debugger

- A debugger is a software program used to test and find bugs (errors) in other programs .
- A debugger is also known as a debugging tool.
- There are two types of debuggers :
- CorDBG (command-line debugger) – in this , compilation of the original c# file using the debug switch is a must.
- DbgCLR (graphic debugger) – used by Visual Studio .NET

List of Debuggers

- Some widely used debuggers are:
- Firefox JavaScript debugger
- GDB - the GNU debugger
- LLDB
- Microsoft Visual Studio Debugger
- Valgrind
- WinDbg
- Eclipse debugger API used in a range of IDEs : 1. Eclipse IDE (Java) 2. Nodeclipse (JavaScript)
- WDW, the OpenWatcom debugger

Text Editor

- Primary interface to the computer for all type of “knowledge workers”.
- They compose, organize, study, and manipulate computer-based information.
- A text editor allows you to edit a text file (create, modify etc…).
- The common editing features are:
  - Moving the cursor,
  - Deleting
  - Replacing
  - Pasting
  - Searching
  - Searching and replacing,
  - Saving and loading, and,
  - Miscellaneous(e.g. quitting)

Examples

- Windows OS - Notepad, WordPad, Microsoft Word, and text editors.
- UNIX OS - vi, emacs, jed, pico.
- Gui based editors
- • Gedit gvim
- Nedit
- Tea
- subtime

Microservices

- **Microservices** are a software development technique
- Microservices - also known as the micro service architecture
- A variant of the service-oriented architecture (SOA) structural style that arranges an application as a collection of loosely coupled services.
- In a microservices architecture, services are fine-grained and the protocols are lightweight.
- It is an architectural style that structures an application as a collection of services that are:
  - Highly maintainable and testable
  - Loosely coupled
  - Independently deployable
  - Organized around business capabilities
  - Owned by a small team

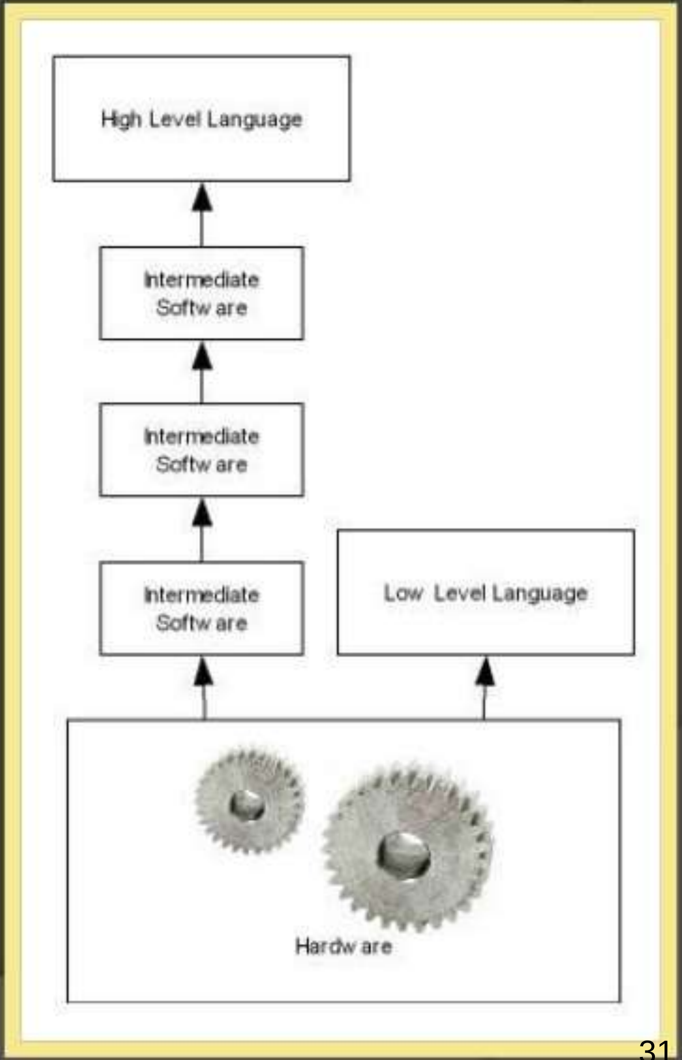
Containers

- A **microservice** is an application with a single function, such as routing network traffic, making an online payment or analyzing a medical result.
- **Containers** are easily packaged, lightweight and designed to run anywhere.
- Multiple **containers** can be deployed in a single VM.
- Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.

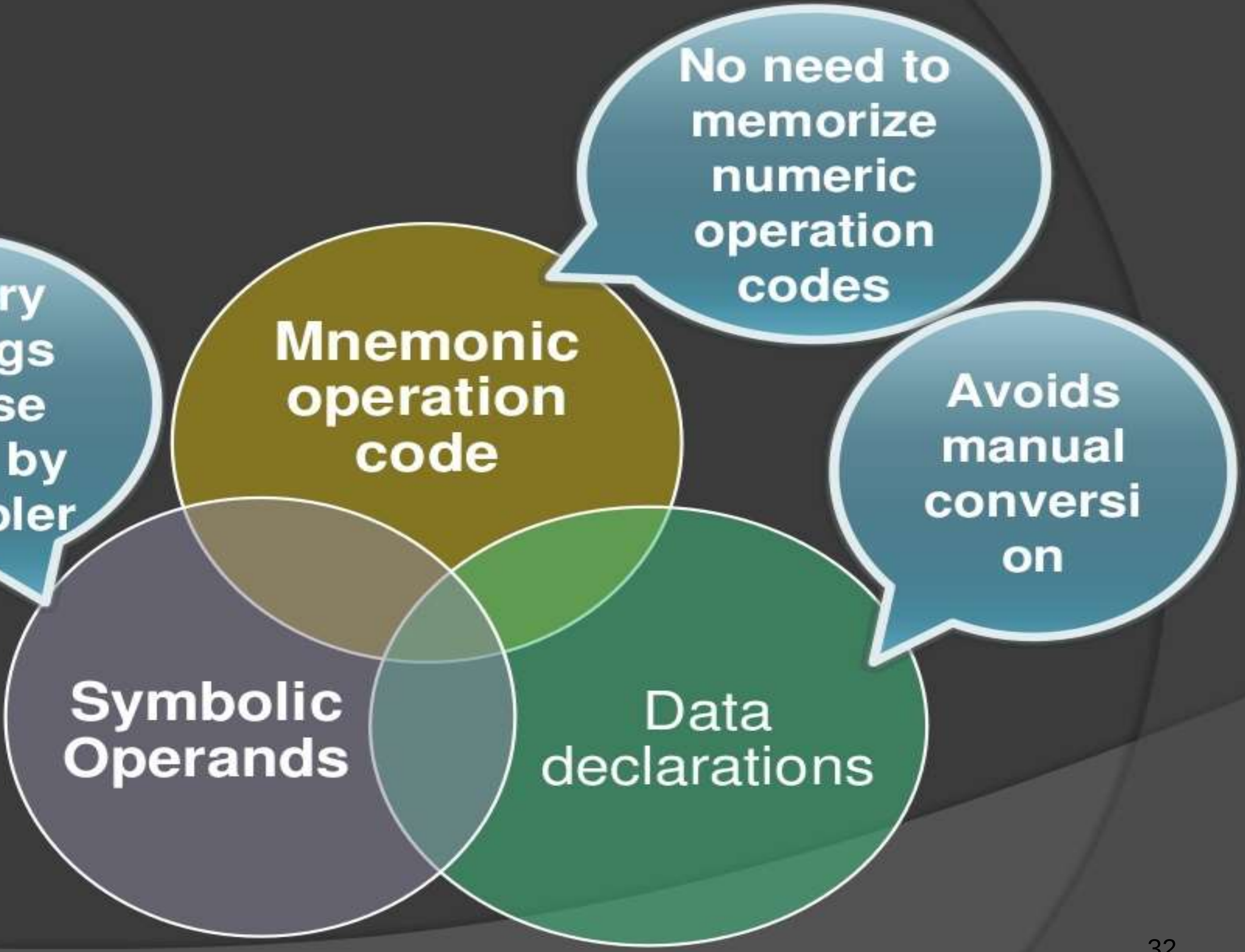
Assembler

Assembly language

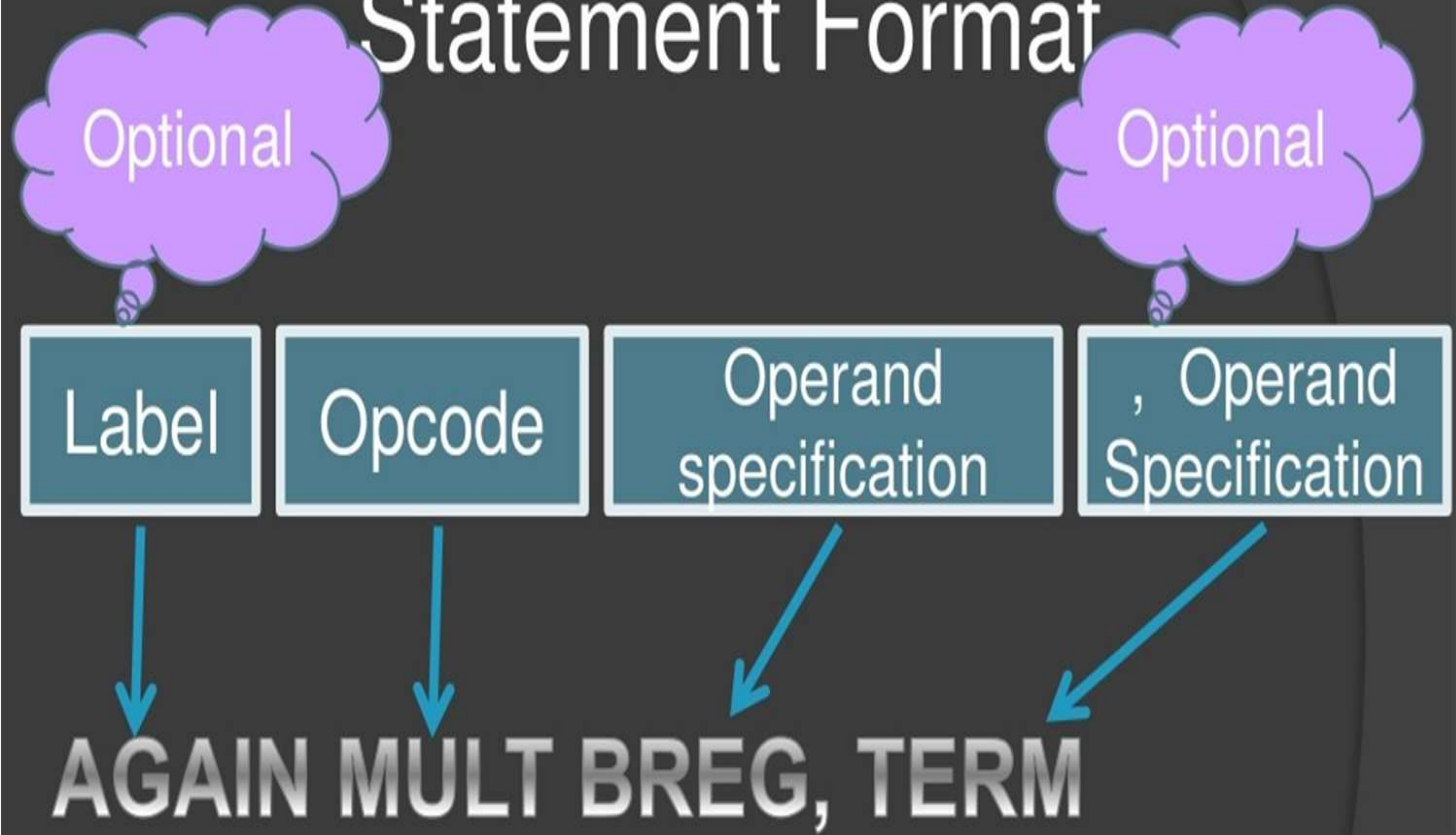
- Machine dependant
- Low level programming language



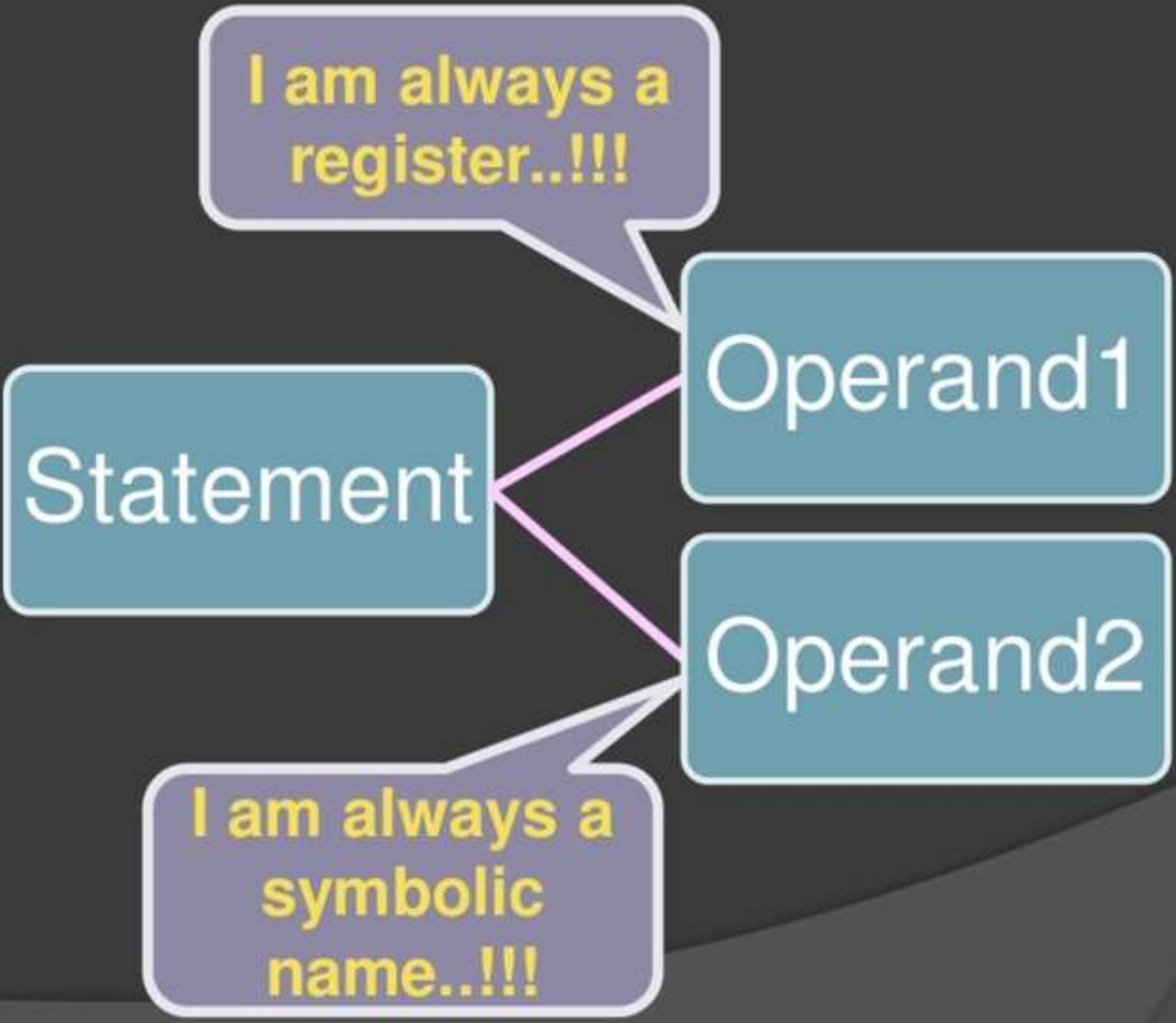
Elements Of ALP


No need to memorize numeric operation codes
Memory bindings to these names by assembler
Mnemonic operation code
Avoids manual conversi on
Symbolic Operands
Data declarations

Statement Format

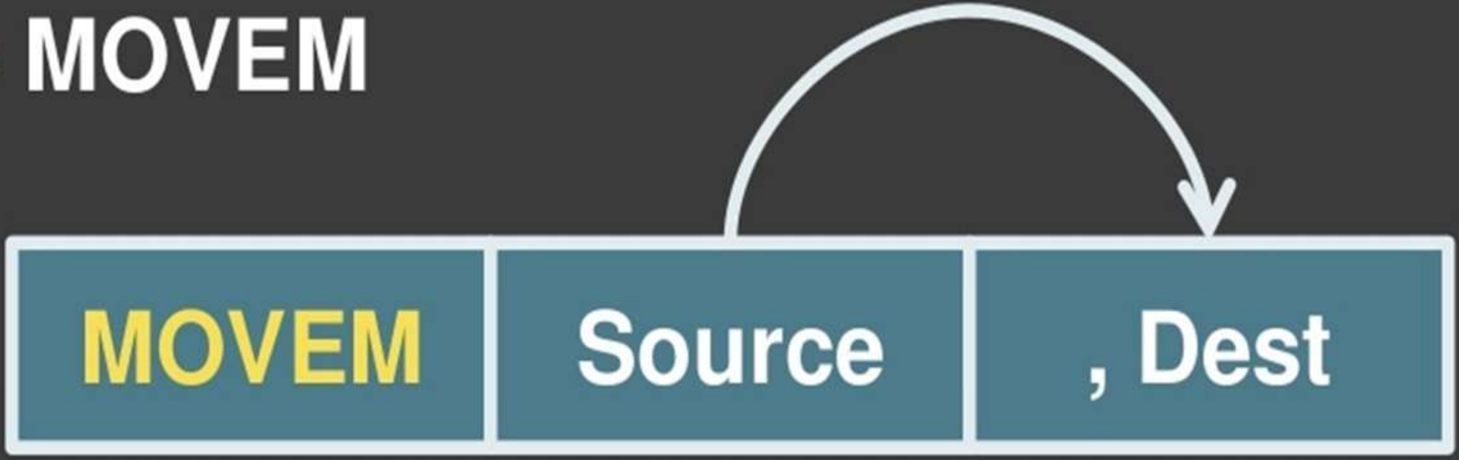


A simple assembly language

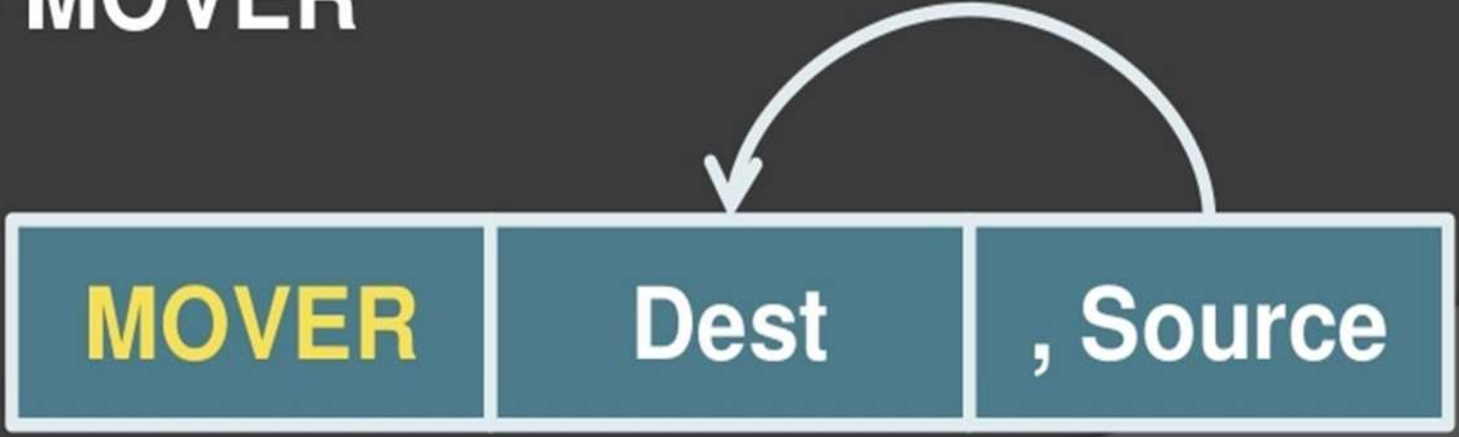


MOVER and MOVEM

- **MOVEM**



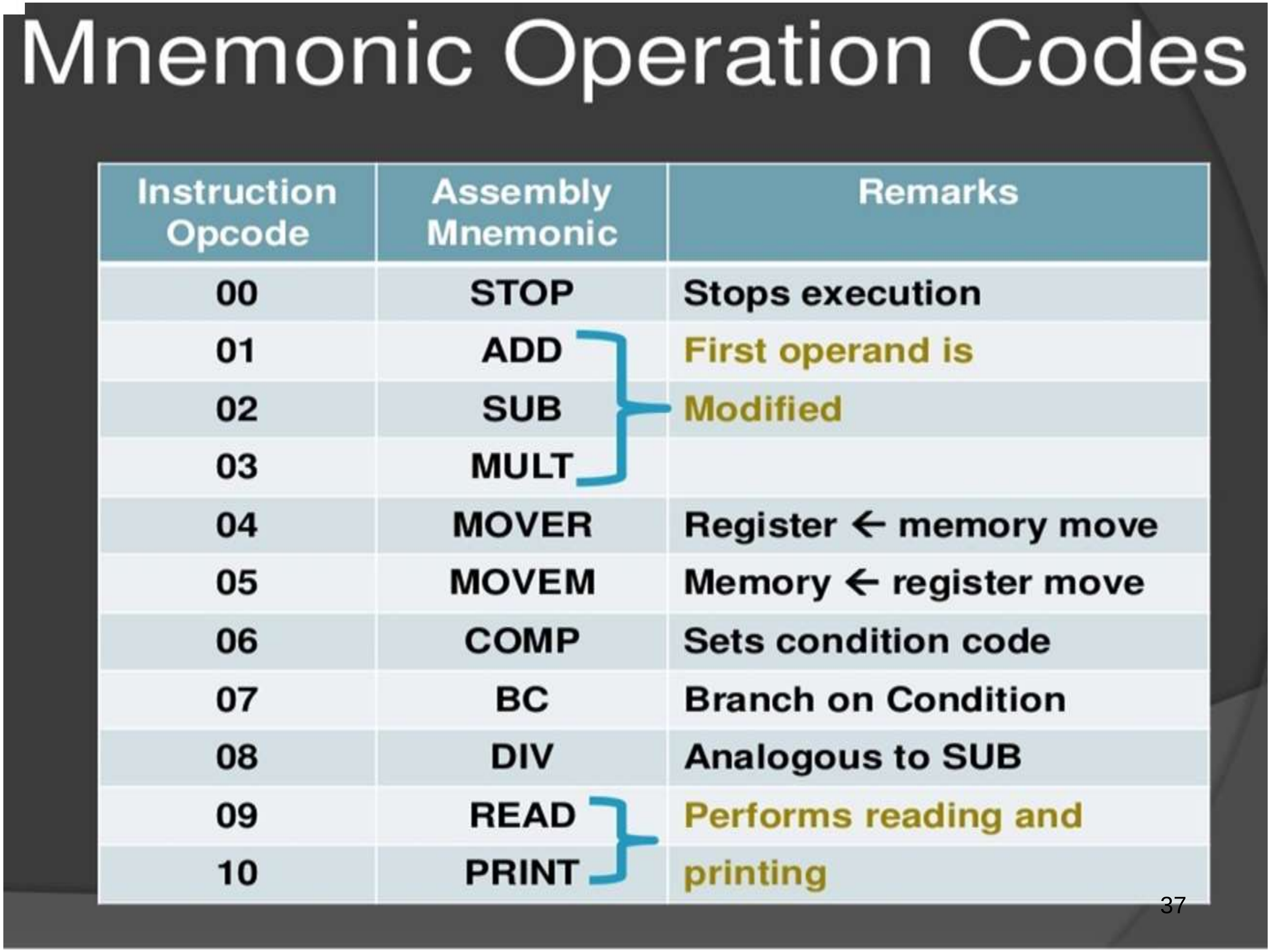
- **MOVER**



Operand specification

Symbolic name	+ (Displacement)	Index register

AREA+5(4)



Mnemonic Operation Codes

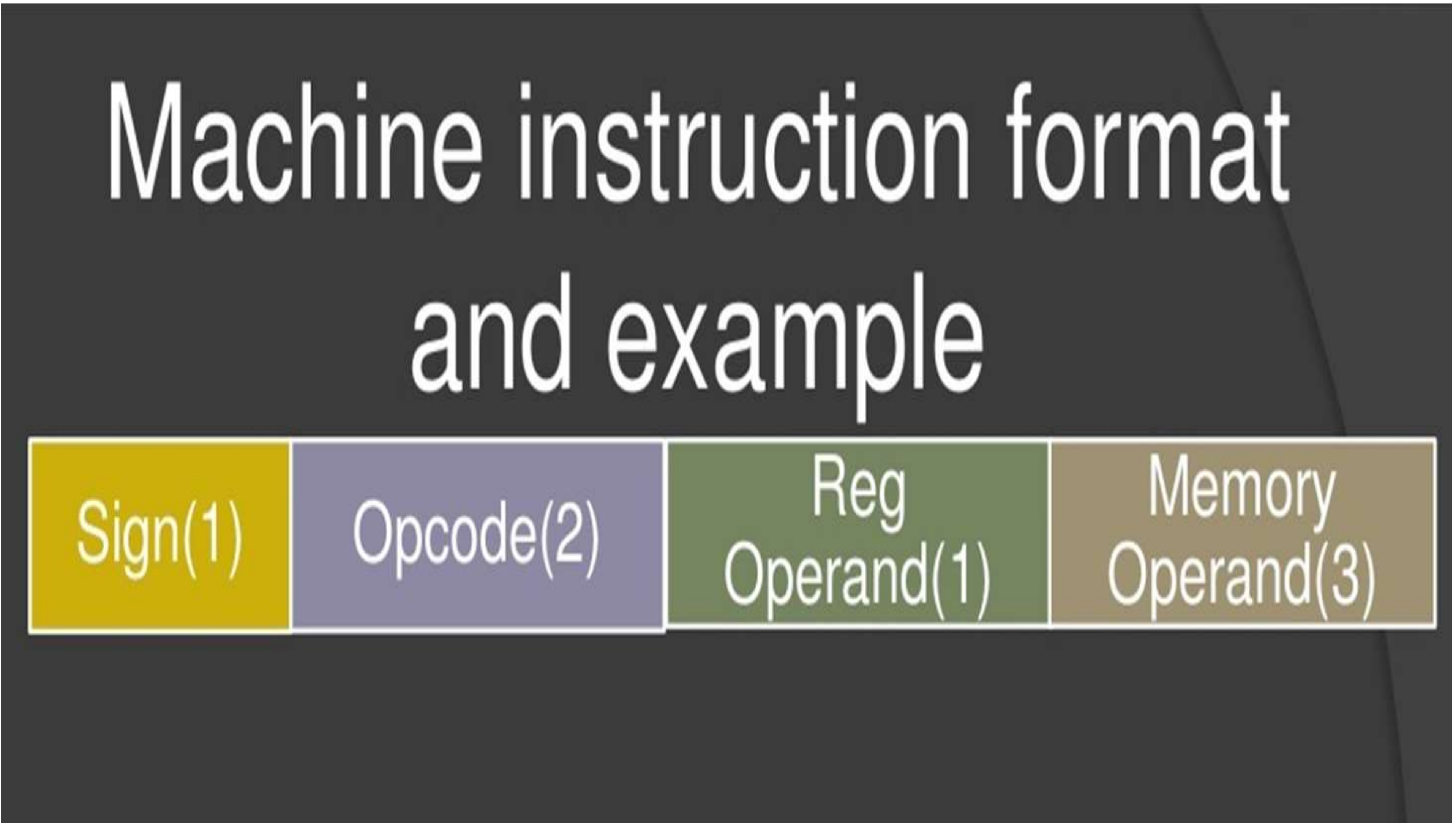
Instruction Opcode	Assembly Mnemonic	Remarks
00	STOP	Stops execution
01	ADD	First operand is
02	SUB	Modified
03	MULT	
04	MOVER	Register ← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on Condition
08	DIV	Analogous to SUB
09	READ	Performs reading and
10	PRINT	printing

Syntax for BC

BC	Condition code	Memory Address

LT, LE, EQ,
GT, GE, ANY

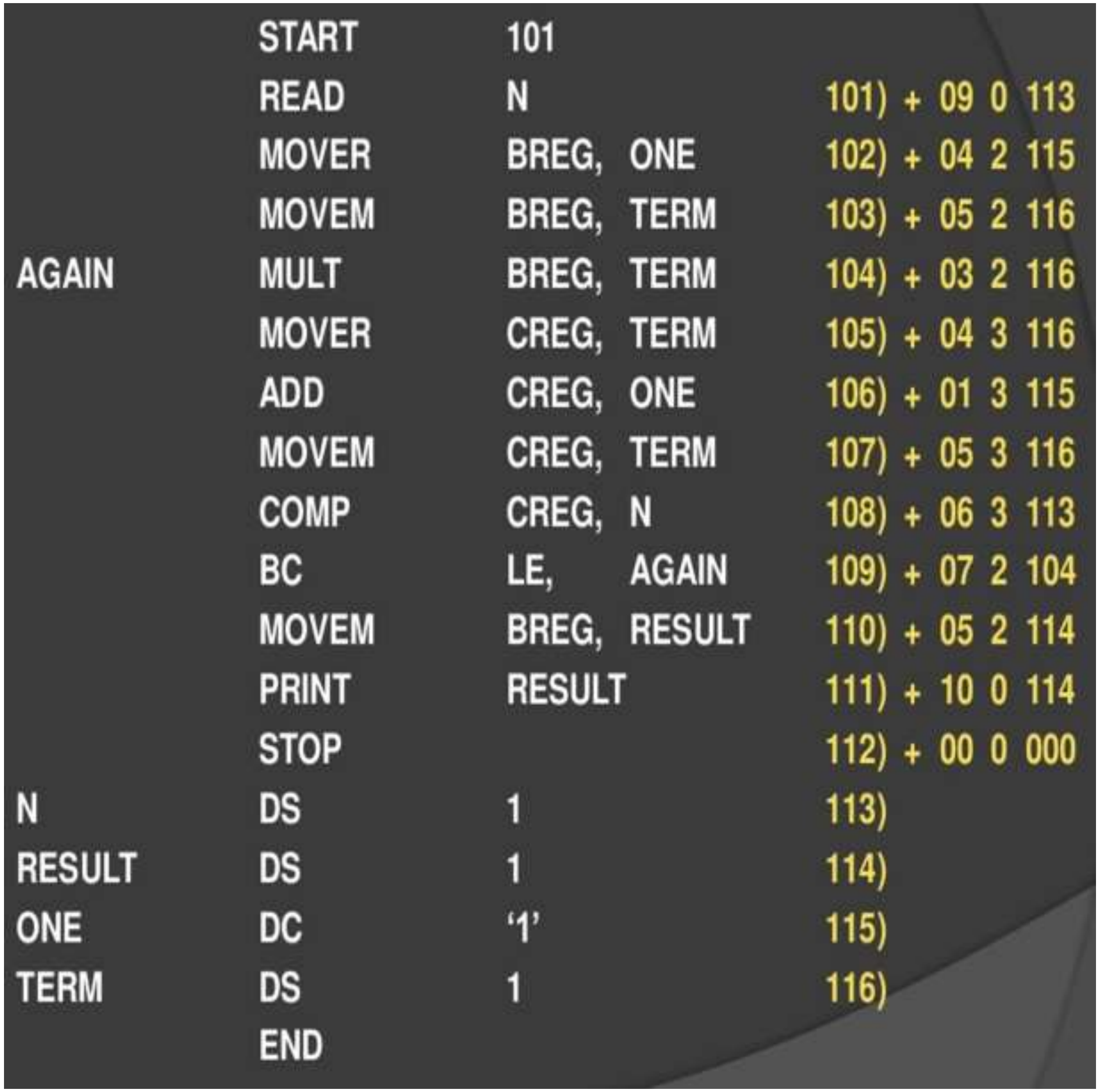
What if we want to have an unconditional jump?



Machine instruction format and example

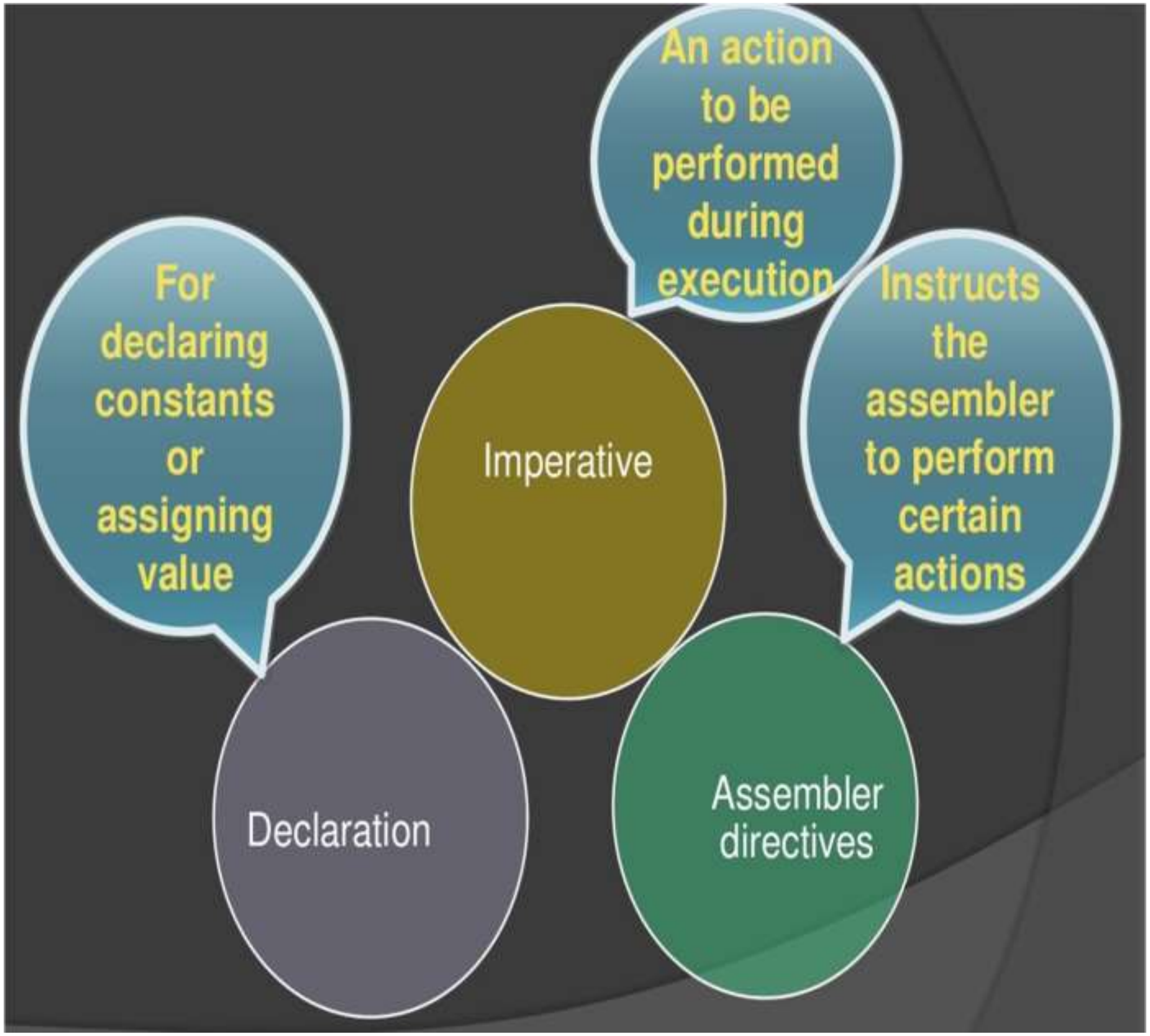
Sign(1)	Opcode(2)	Reg Operand(1)	Memory Operand(3)

Example ALP and its equivalent MLP



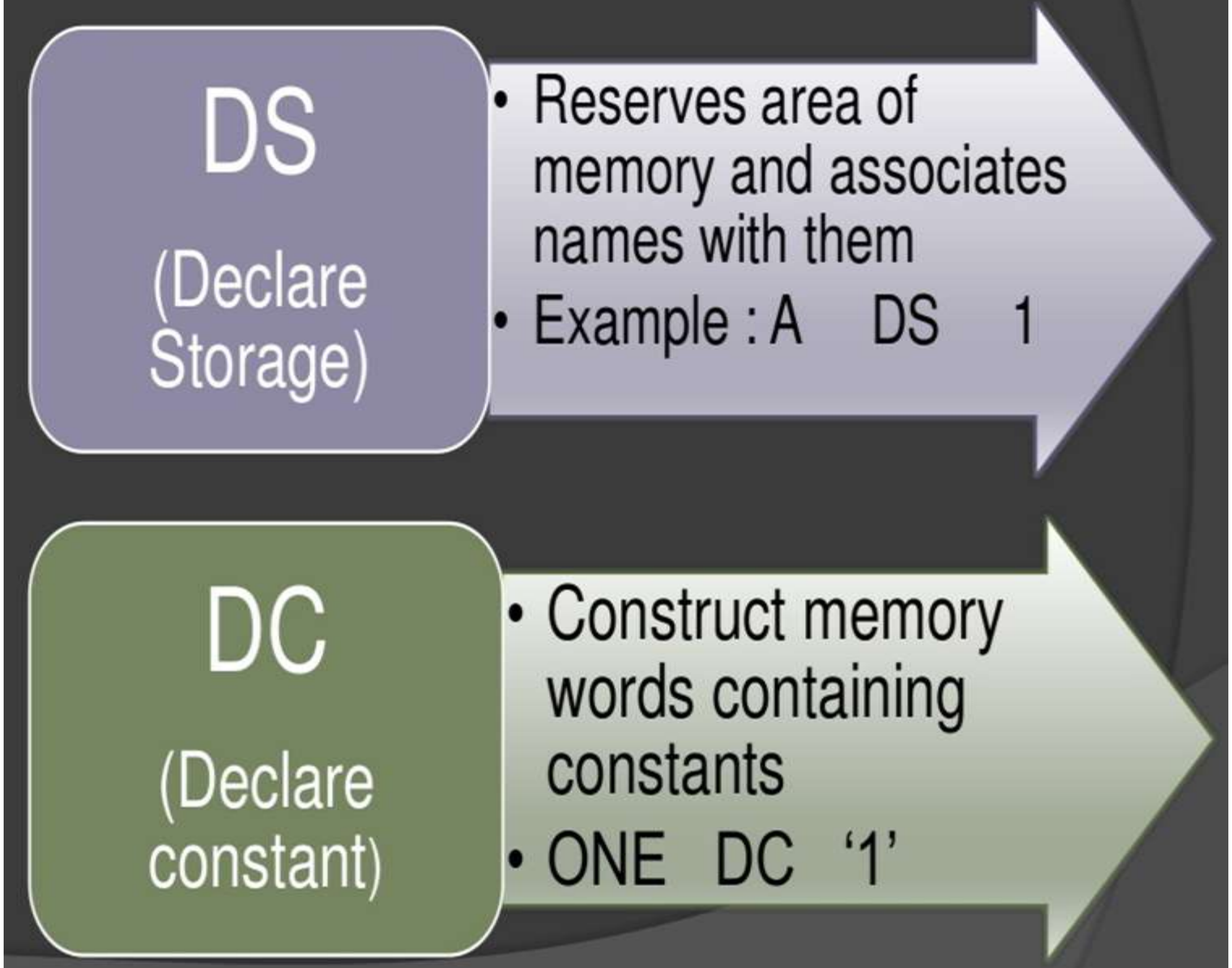
```
          START   101
          READ    N               101) + 09 0 113
          MOVER   BREG,  ONE      102) + 04 2 115
          MOVEM   BREG,  TERM     103) + 05 2 116
AGAIN     MULT    BREG,  TERM     104) + 03 2 116
          MOVER   CREG,  TERM     105) + 04 3 116
          ADD     CREG,  ONE      106) + 01 3 115
          MOVEM   CREG,  TERM     107) + 05 3 116
          COMP    CREG,  N        108) + 06 3 113
          BC      LE,    AGAIN    109) + 07 2 104
          MOVEM   BREG,  RESULT   110) + 05 2 114
          PRINT   RESULT          111) + 10 0 114
          STOP                    112) + 00 0 000
N         DS      1               113)
RESULT    DS      1               114)
ONE       DC      '1'             115)
TERM      DS      1               116)
          END
```

Statement Class for an ALP



For declaring constants or assigning value
An action to be performed during execution
Instructs the assembler to perform certain actions
Imperative
Declaration
Assembler directives

Declaration Statements



DS

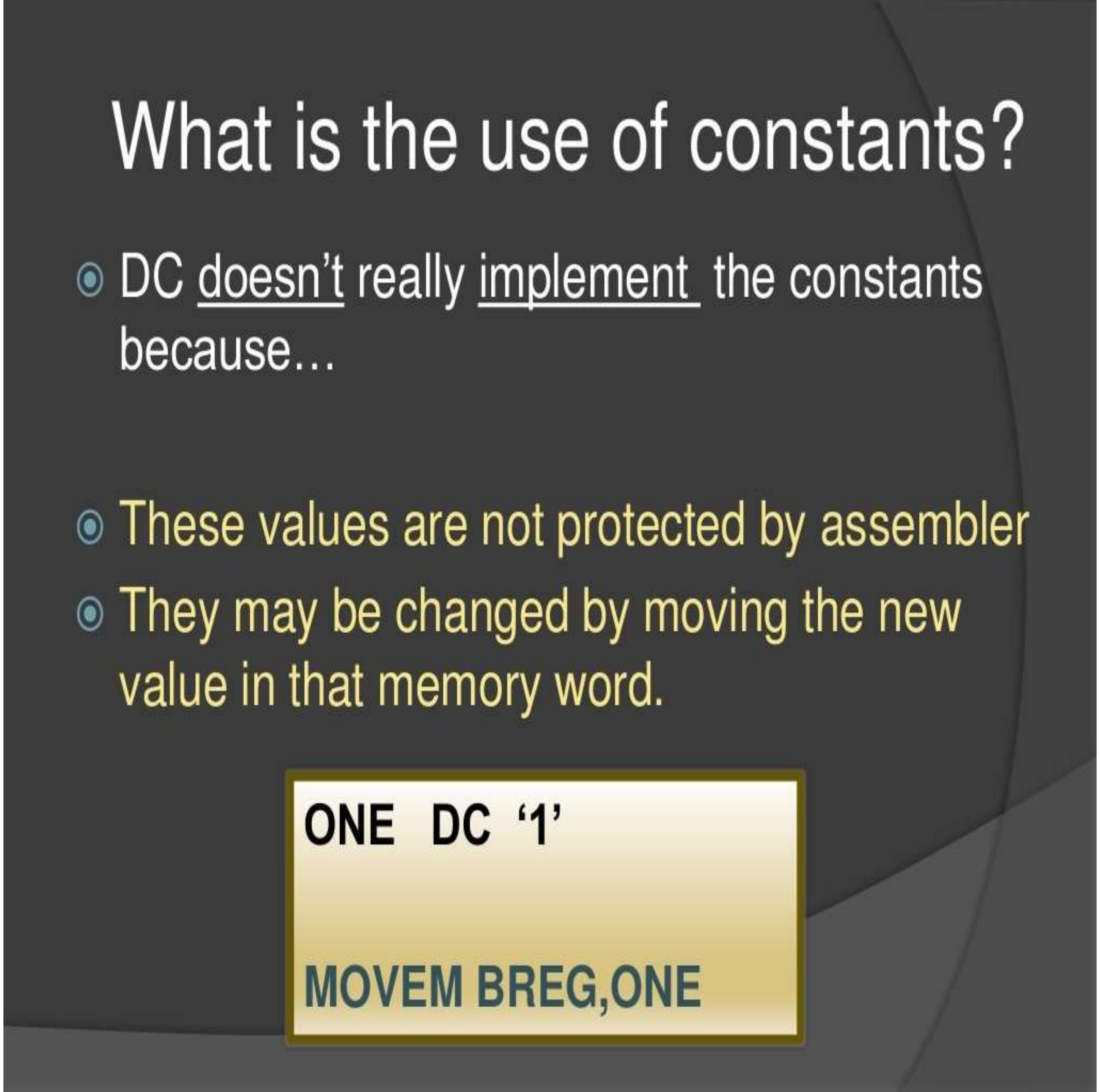
(Declare Storage)

- Reserves area of memory and associates names with them
- Example : A DS 1

DC

(Declare constant)

- Construct memory words containing constants
- ONE DC ‘1’



What is the use of constants?

- DC doesn't really implement the constants because…

- These values are not protected by assembler
- They may be changed by moving the new value in that memory word.

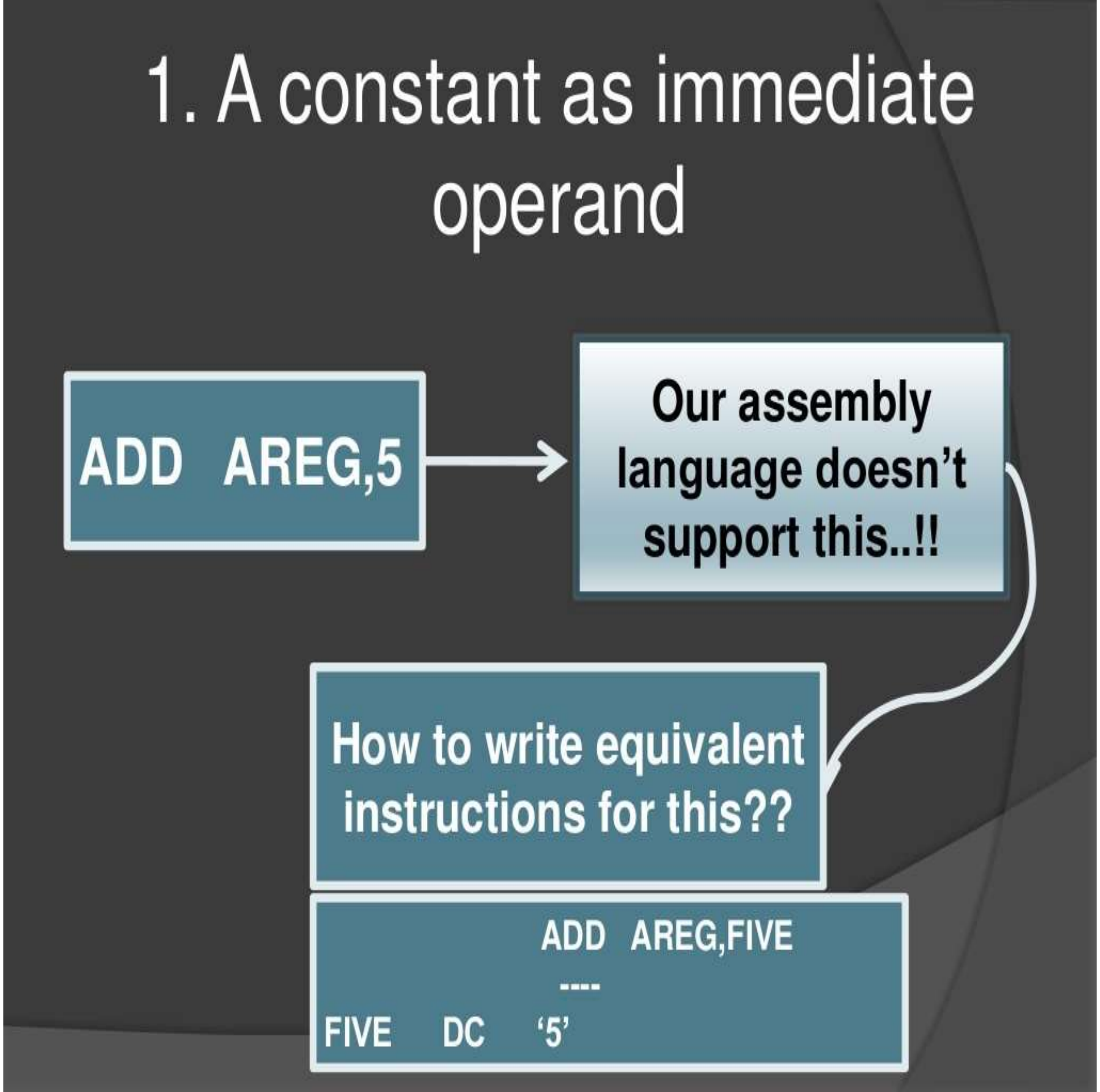
```
ONE   DC  '1'


MOVEM BREG,ONE
```

Similarities with the implementation of constants in HLL

1. Immediate Operands

2. literals



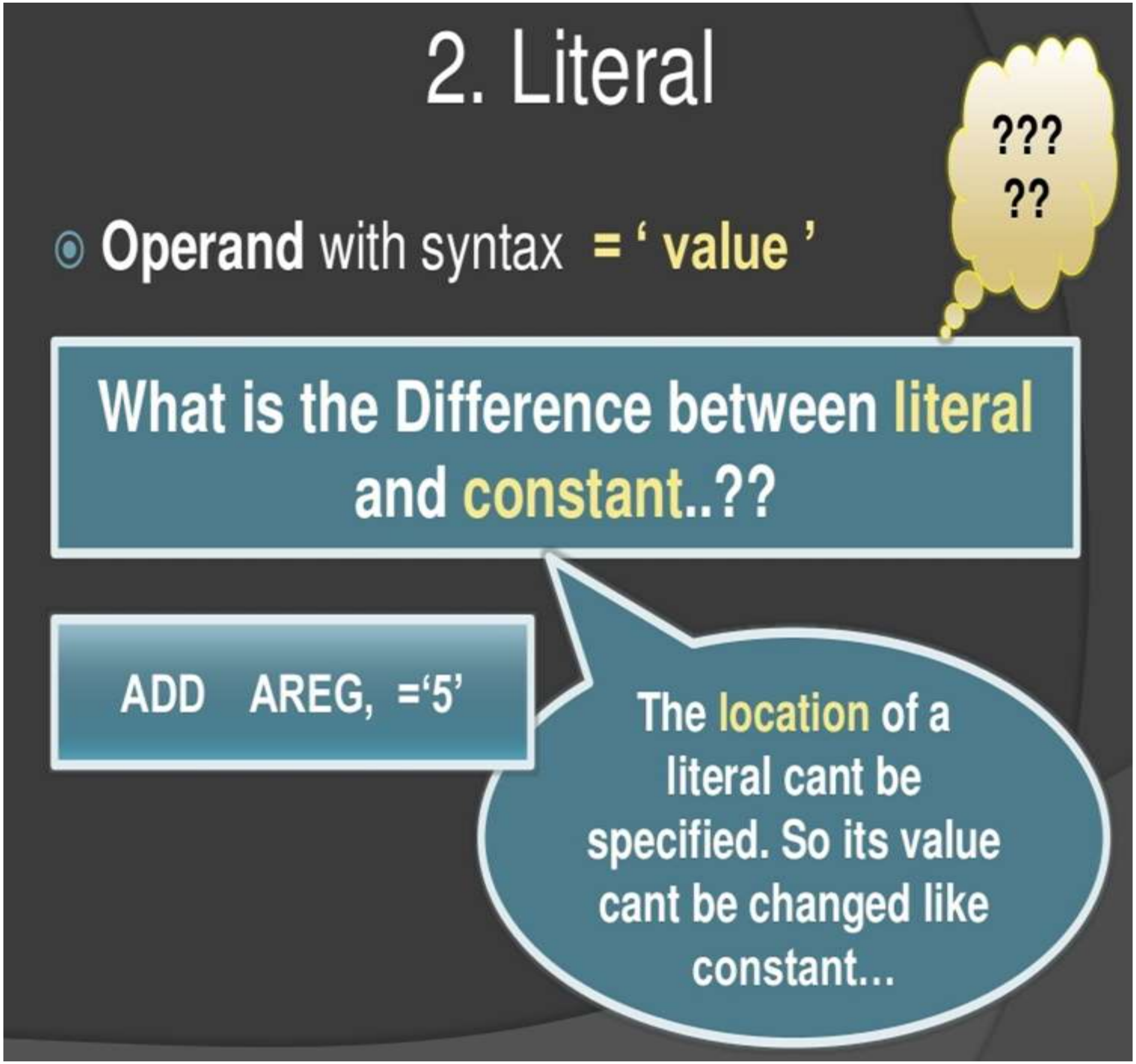
1. A constant as immediate operand

```
ADD   AREG,5
```

Our assembly language doesn't support this..!!

How to write equivalent instructions for this??

```
              ADD   AREG,FIVE
               ----
FIVE    DC    '5'
```



2. Literal

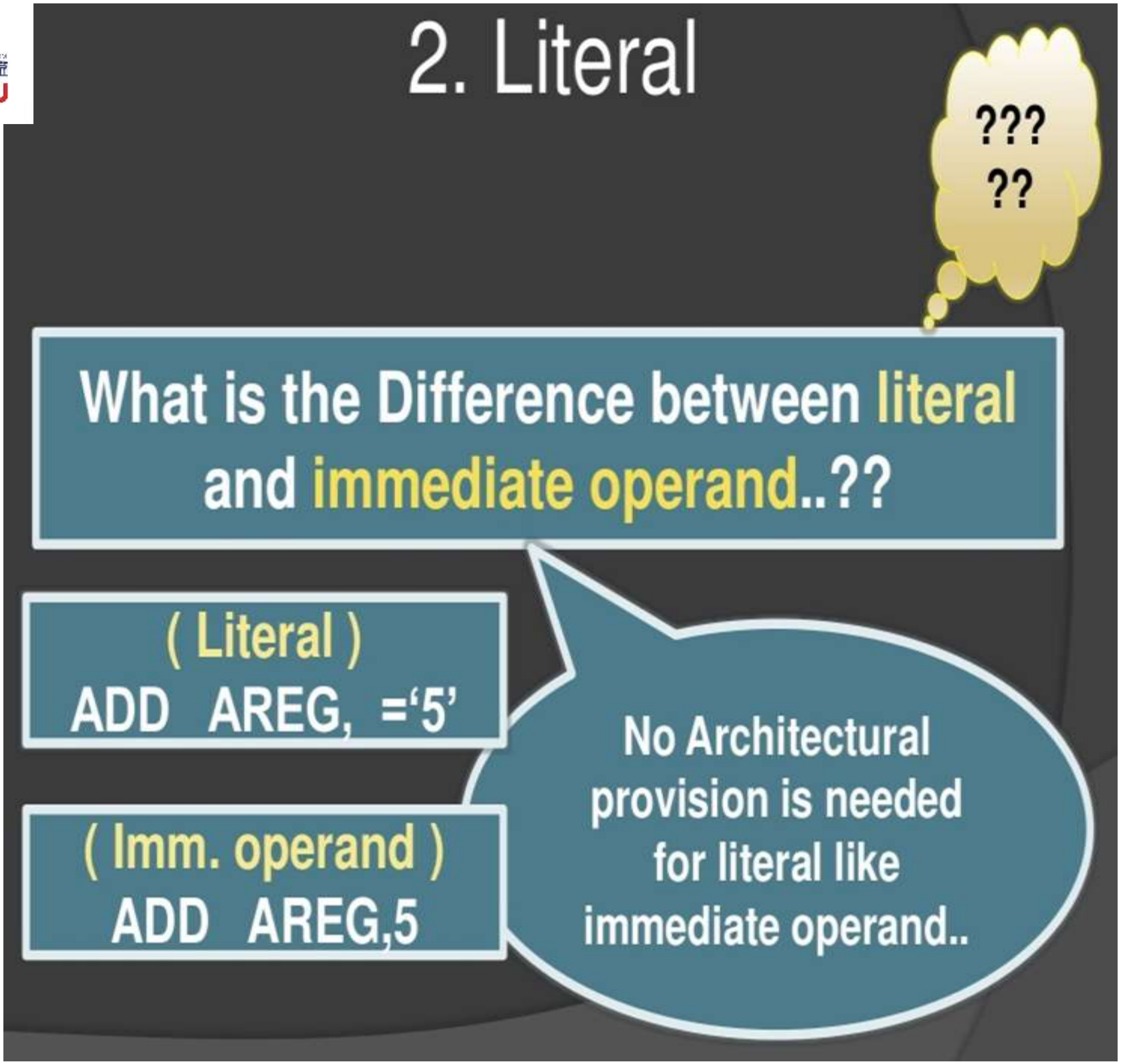
??? ??

- **Operand** with syntax **= ' value '**

What is the Difference between literal and constant..??

```
ADD   AREG, ='5'
```

The location of a literal cant be specified. So its value cant be changed like constant…



2. Literal

???
??

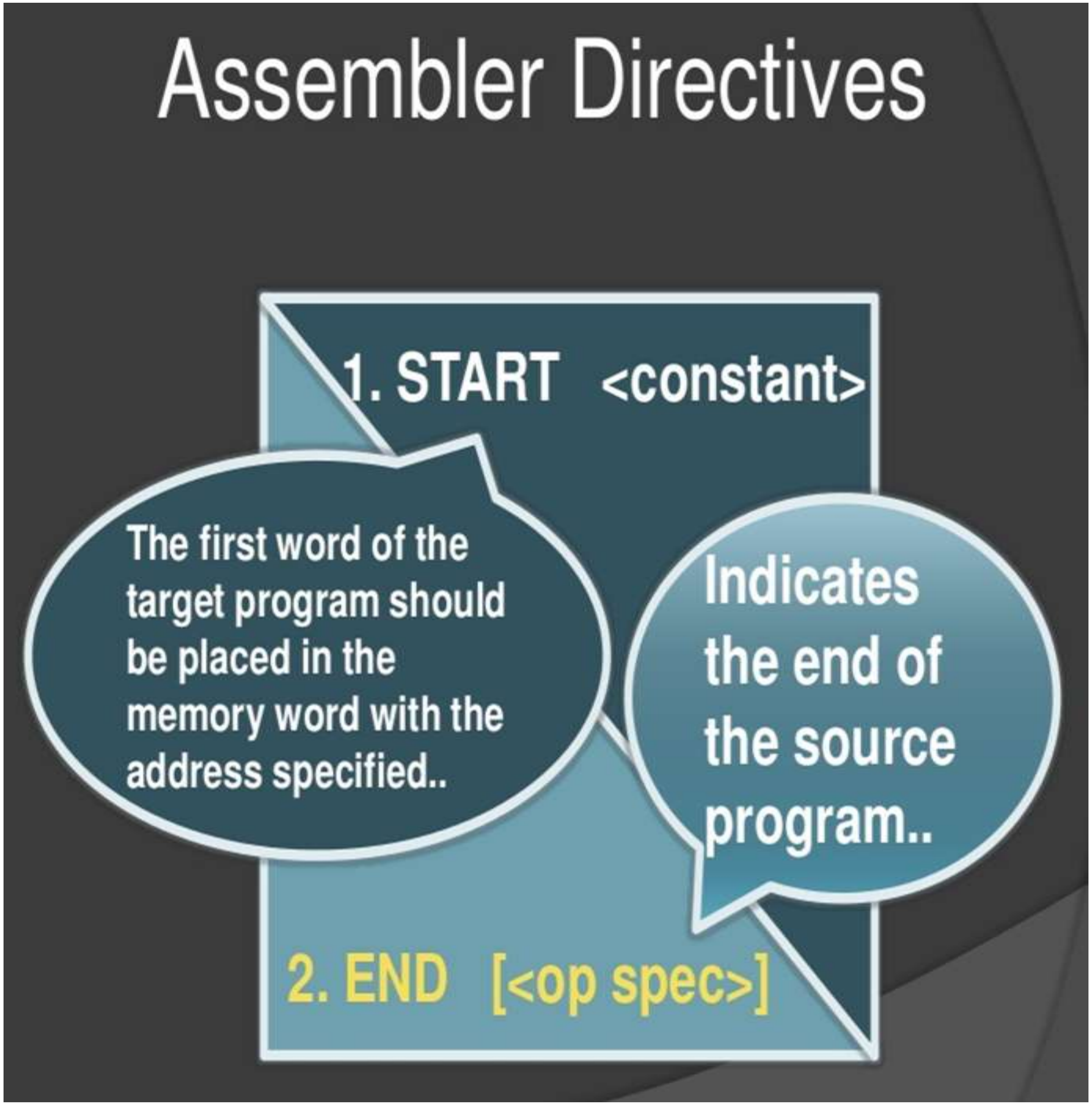
What is the Difference between literal and immediate operand..??

(Literal)
ADD AREG, ='5'

(Imm. operand)
ADD AREG,5

No Architectural provision is needed for literal like immediate operand..

Assembler Directives



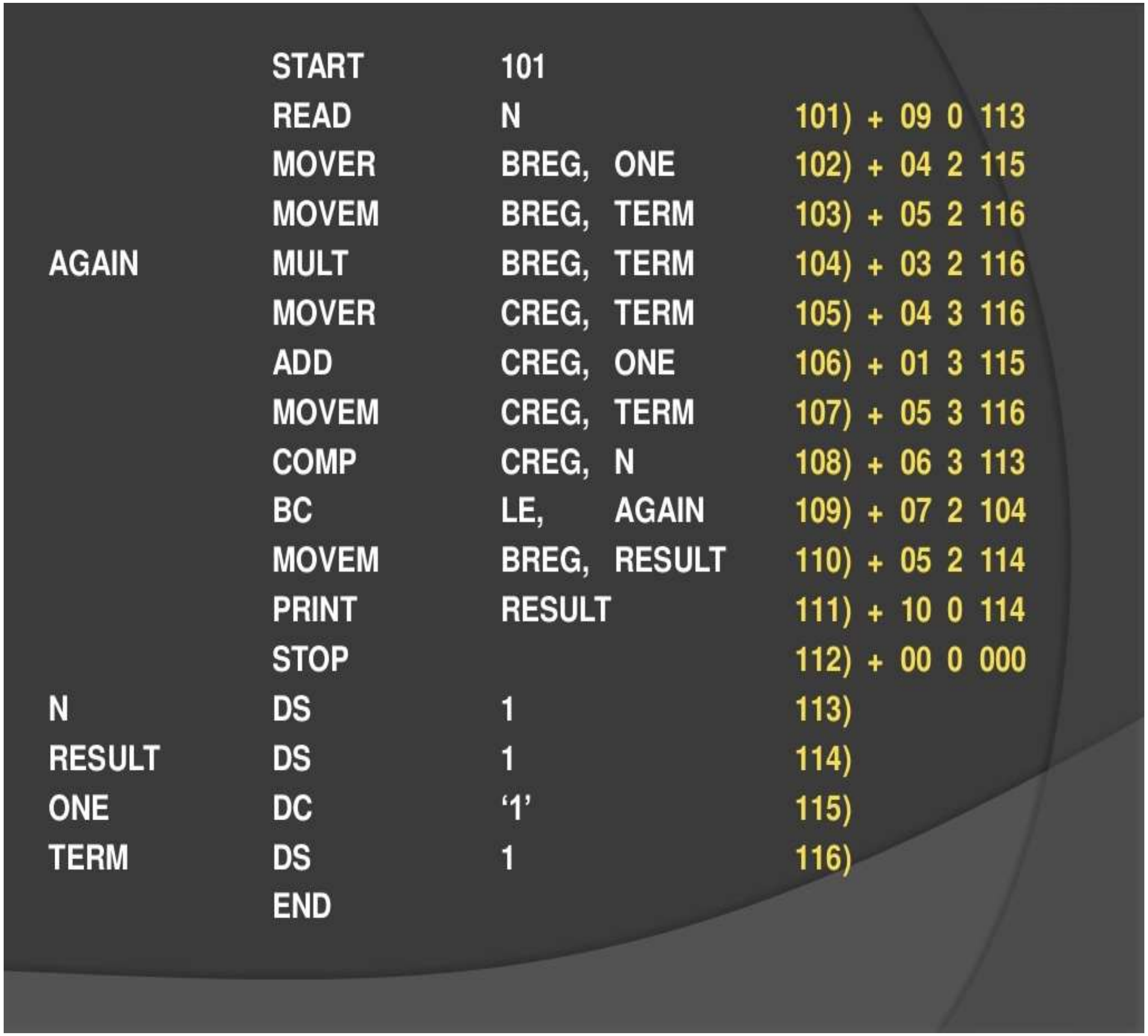
1. START <constant>

The first word of the target program should be placed in the memory word with the address specified..

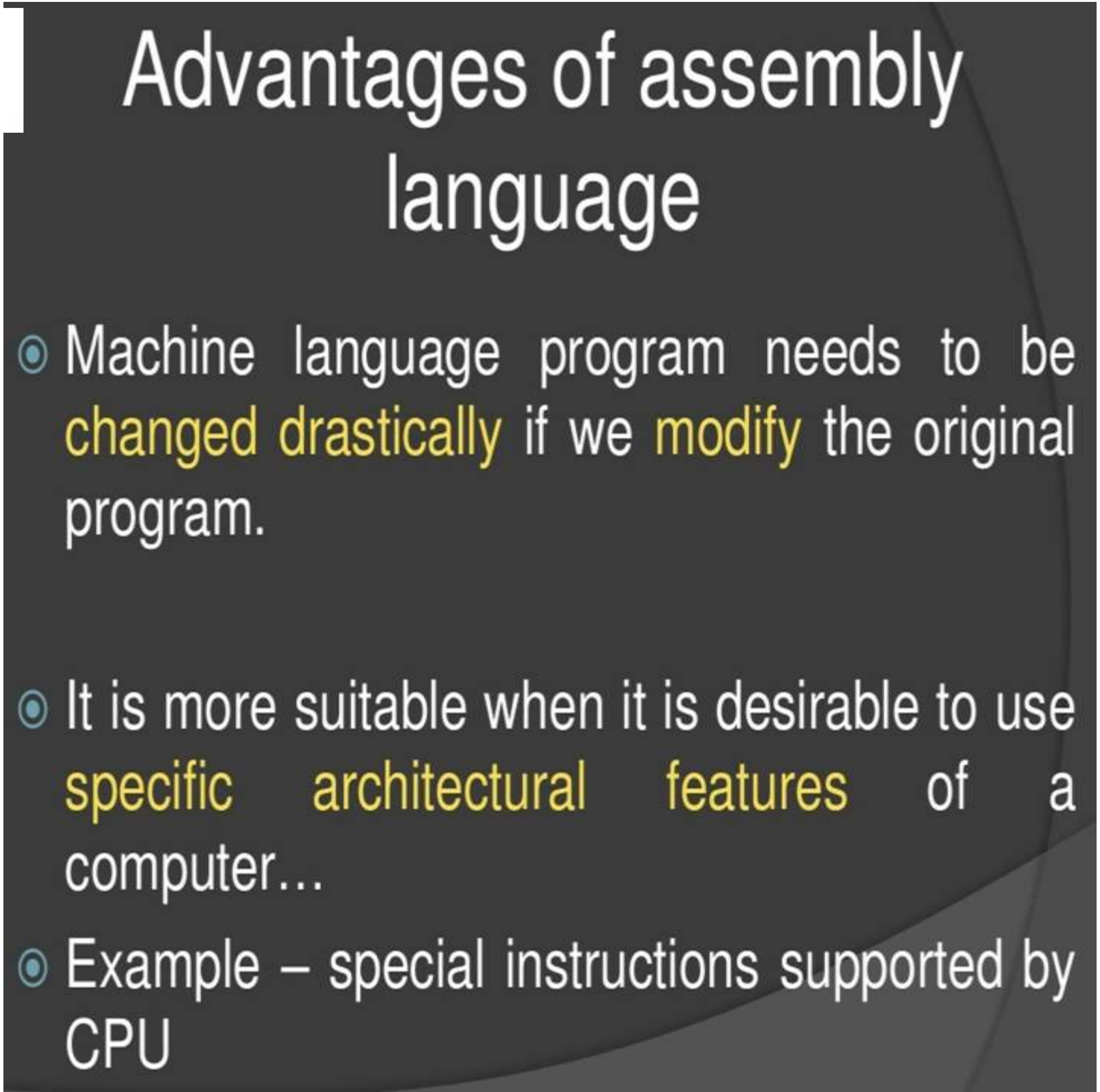
Indicates the end of the source program..

2. END [<op spec>]

Sample Assembly Language Program with its Machine Translation



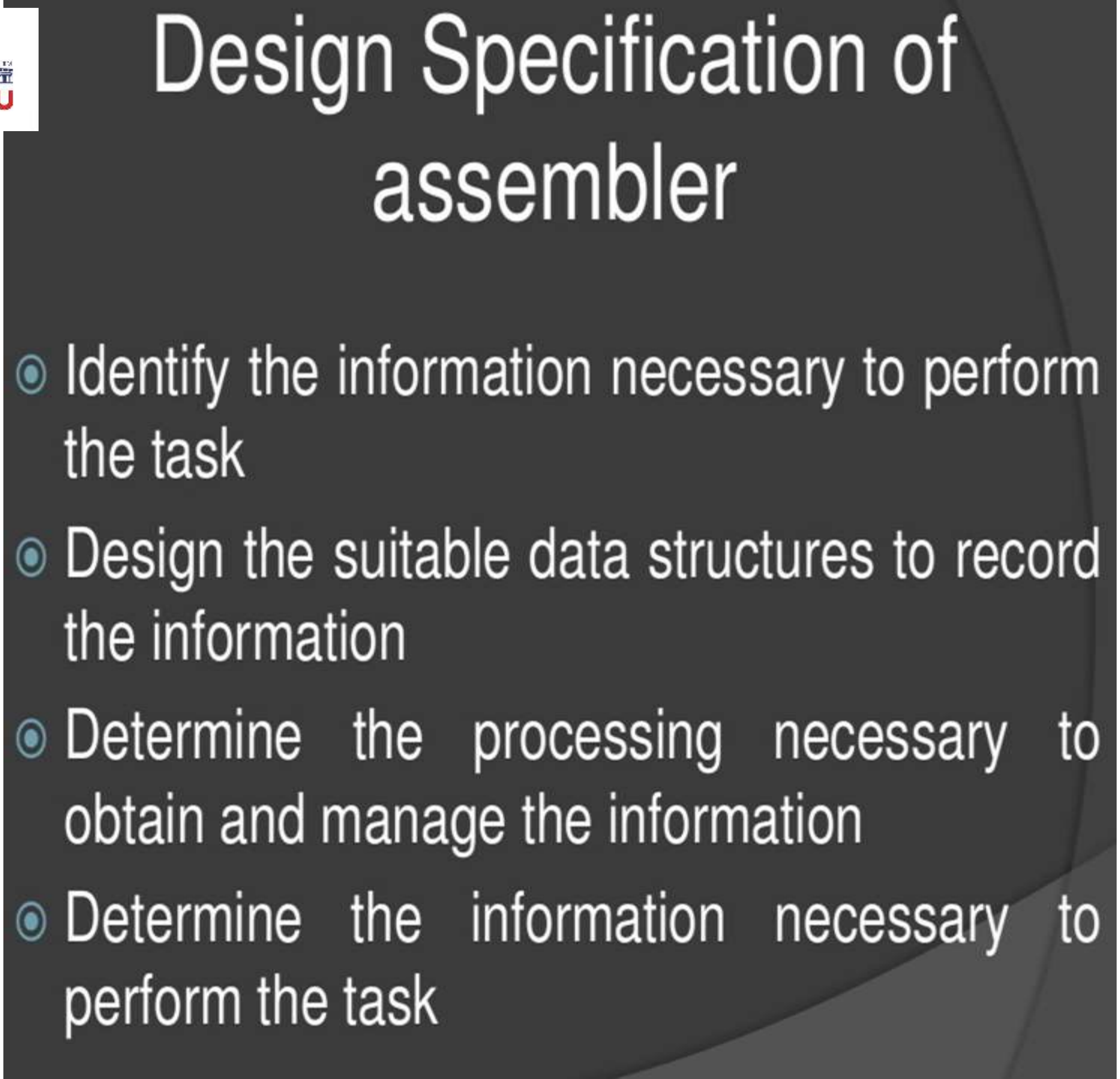
```
            START   101
            READ    N               101) + 09 0 113
            MOVER   BREG,  ONE      102) + 04 2 115
            MOVEM   BREG,  TERM     103) + 05 2 116
AGAIN       MULT    BREG,  TERM     104) + 03 2 116
            MOVER   CREG,  TERM     105) + 04 3 116
            ADD     CREG,  ONE      106) + 01 3 115
            MOVEM   CREG,  TERM     107) + 05 3 116
            COMP    CREG,  N        108) + 06 3 113
            BC      LE,    AGAIN    109) + 07 2 104
            MOVEM   BREG,  RESULT   110) + 05 2 114
            PRINT   RESULT          111) + 10 0 114
            STOP                    112) + 00 0 000
N           DS      1               113)
RESULT      DS      1               114)
ONE         DC      ‘1’             115)
TERM        DS      1               116)
            END
```



Advantages of assembly language

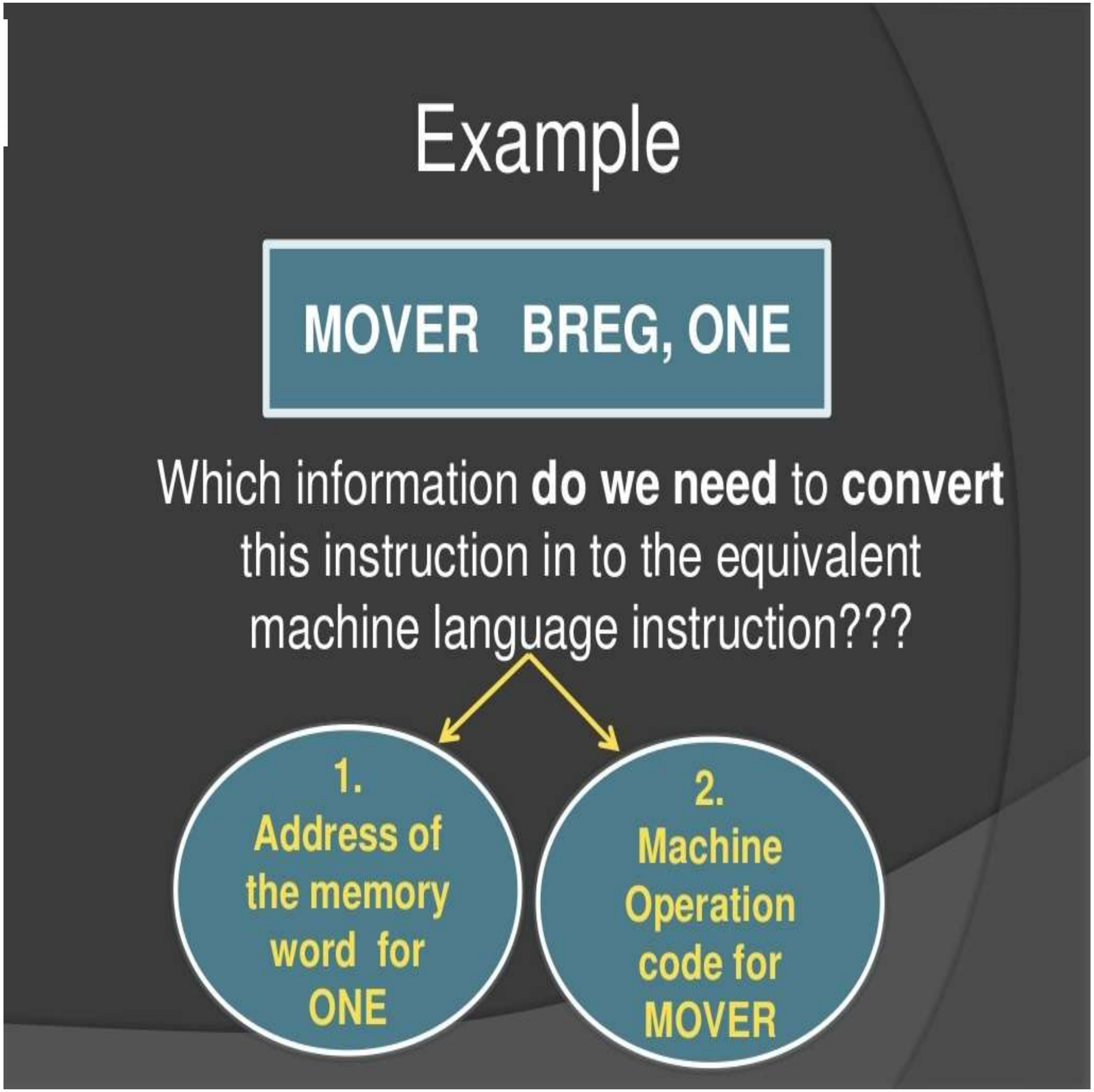
- Machine language program needs to be changed drastically if we modify the original program.

- It is more suitable when it is desirable to use specific architectural features of a computer…
- Example – special instructions supported by CPU



Design Specification of assembler

- Identify the information necessary to perform the task
- Design the suitable data structures to record the information
- Determine the processing necessary to obtain and manage the information
- Determine the information necessary to perform the task



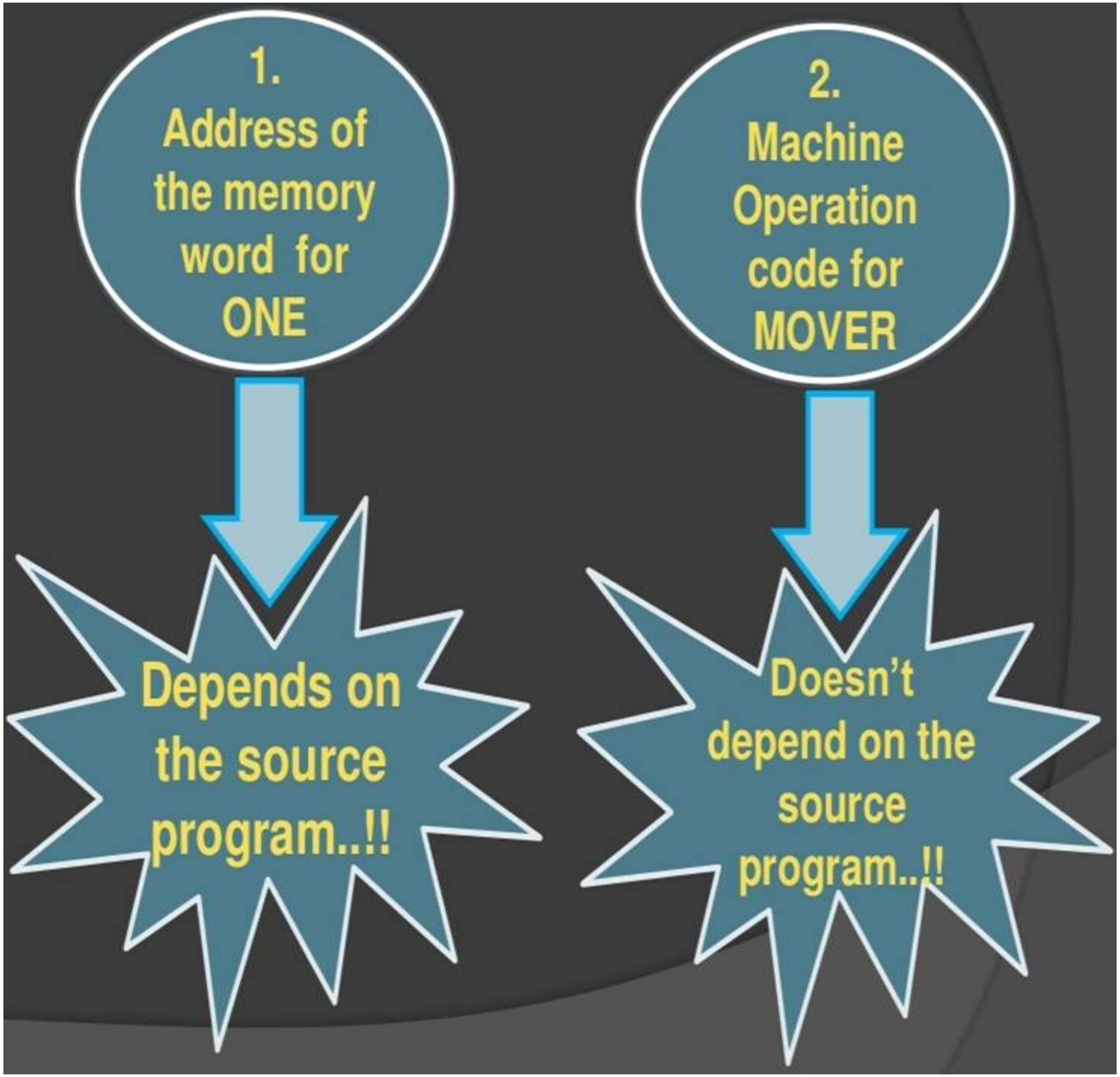
Example

MOVER BREG, ONE

Which information **do we need** to **convert** this instruction in to the equivalent machine language instruction???

1. Address of the memory word for ONE

2. Machine Operation code for MOVER



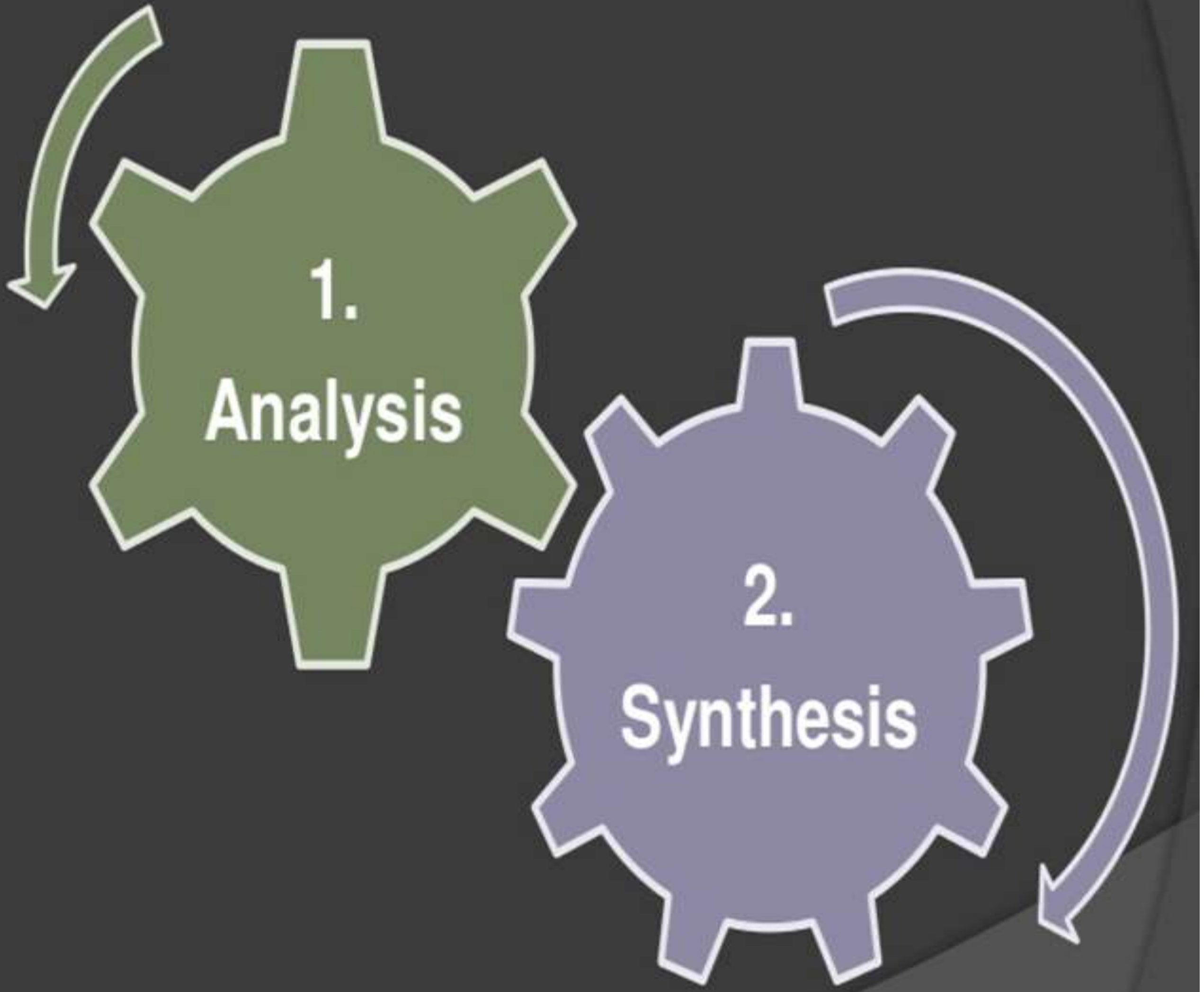
1. Address of the memory word for ONE

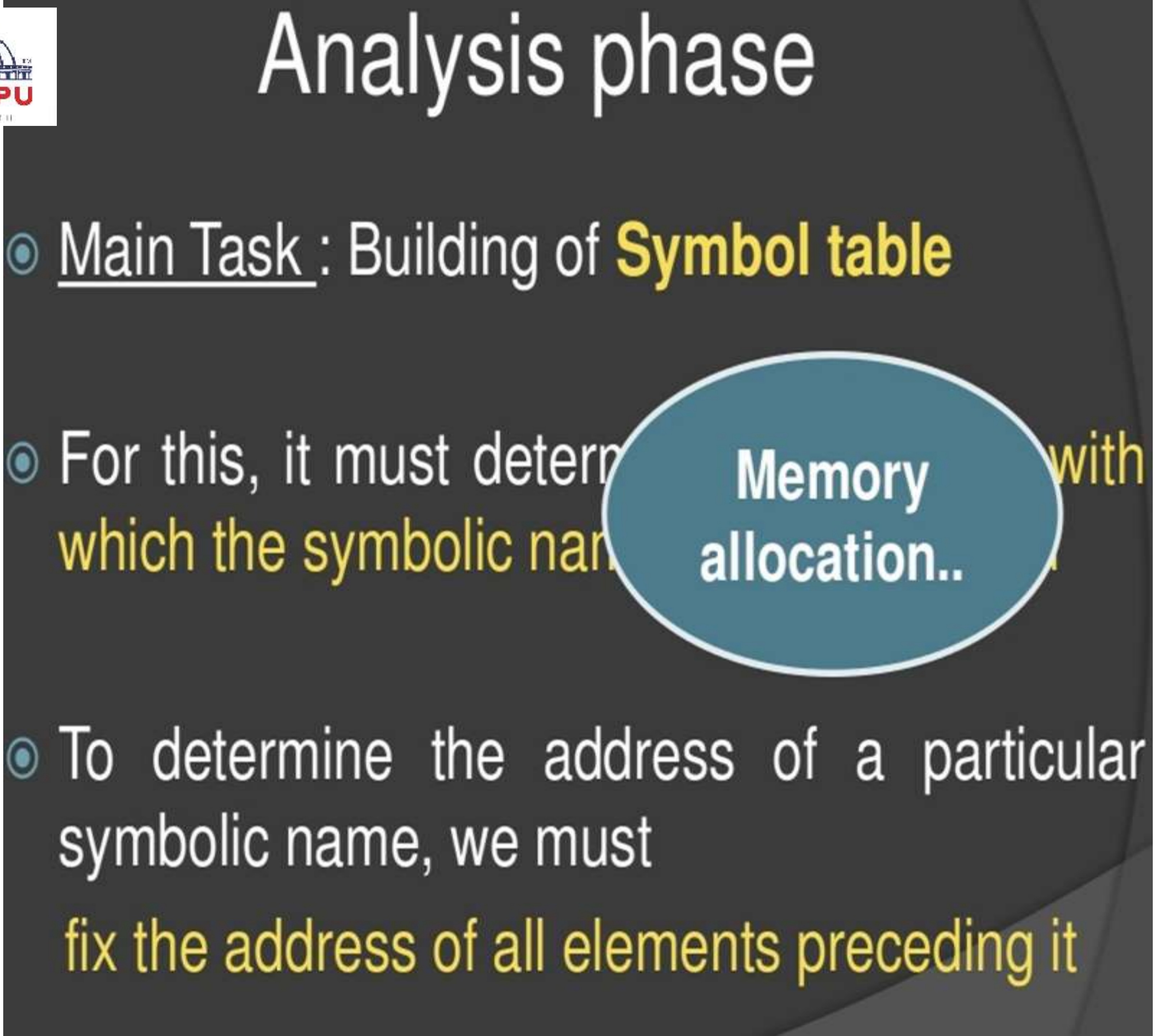
Depends on the source program..!!

2. Machine Operation code for MOVER

Doesn’t depend on the source program..!!

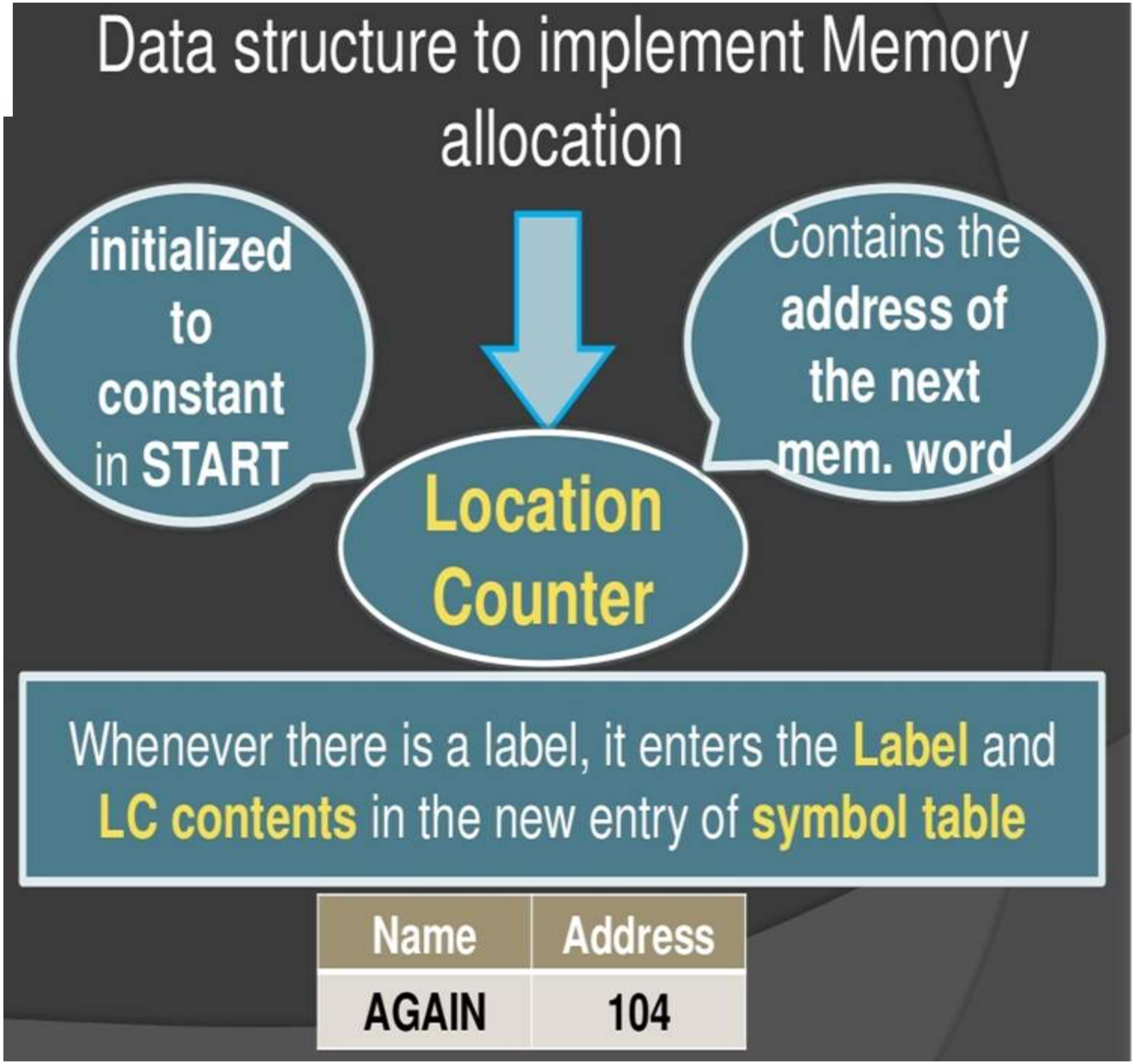
Two phases of assembler





Analysis phase

- <u>Main Task</u> : Building of **Symbol table**
- For this, it must deterr **Memory allocation..** with which the symbolic nan
- To determine the address of a particular symbolic name, we must fix the address of all elements preceding it



Data structure to implement Memory allocation

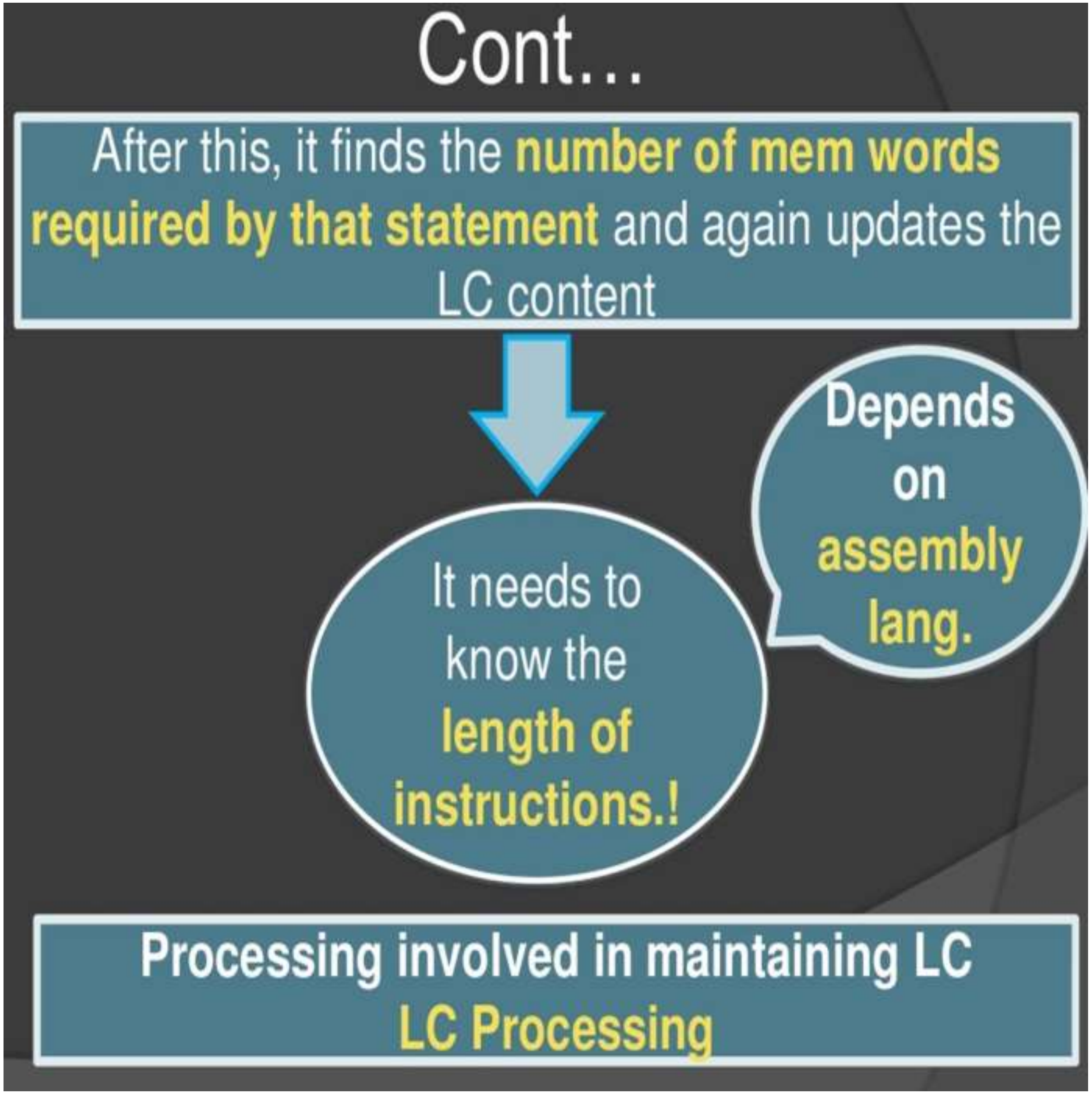
initialized to constant in **START**

Contains the **address of the next mem. word**

Location Counter

Whenever there is a label, it enters the **Label** and **LC contents** in the new entry of **symbol table**

Name	Address
AGAIN	104



Cont…

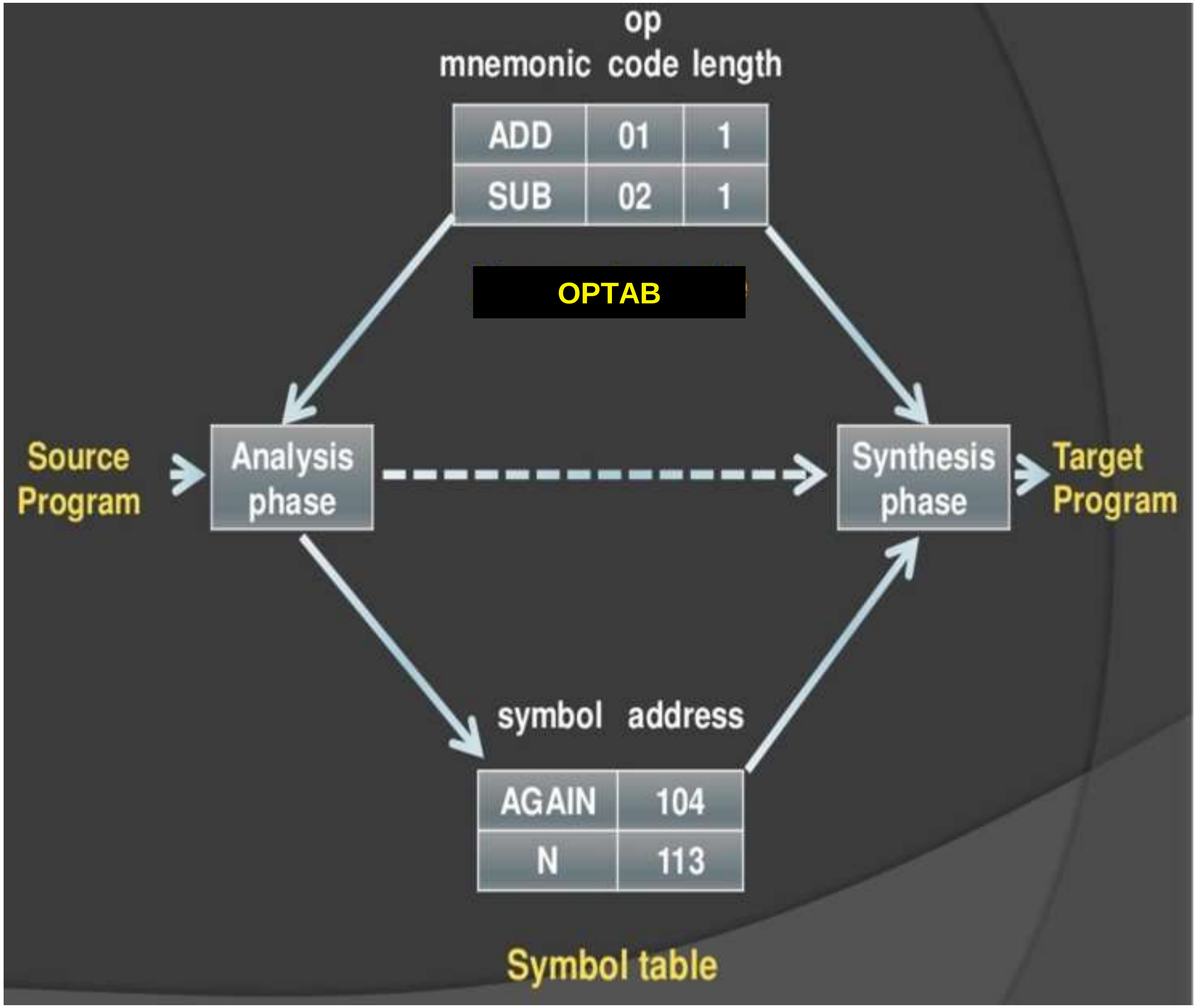
After this, it finds the **number of mem words required by that statement** and again updates the LC content

It needs to know the **length of instructions.!**

Depends on assembly lang.

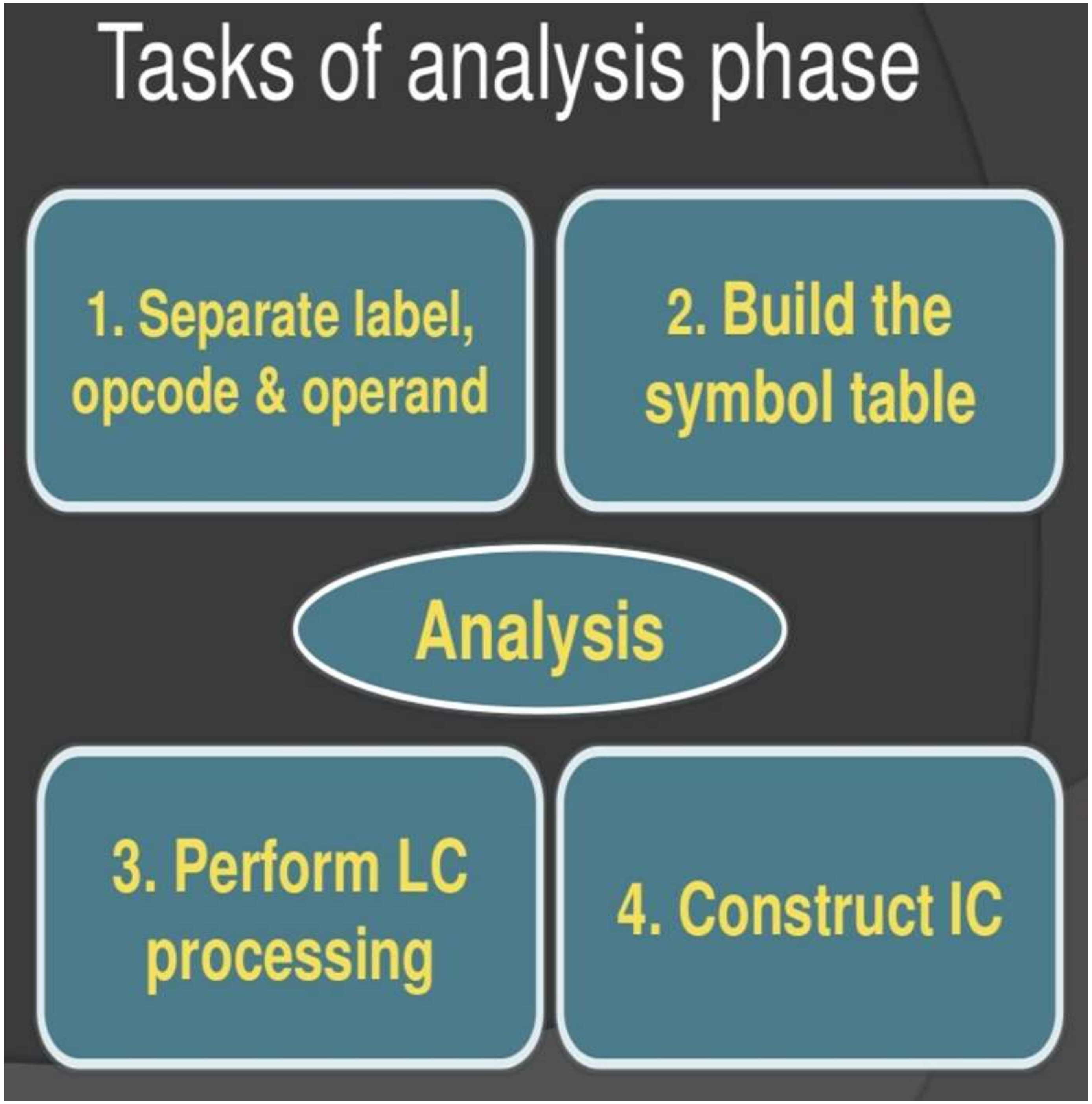
Processing involved in maintaining LC
LC Processing

Data Structures For Assembler



mnemonic	op code	length
ADD	01	1
SUB	02	1

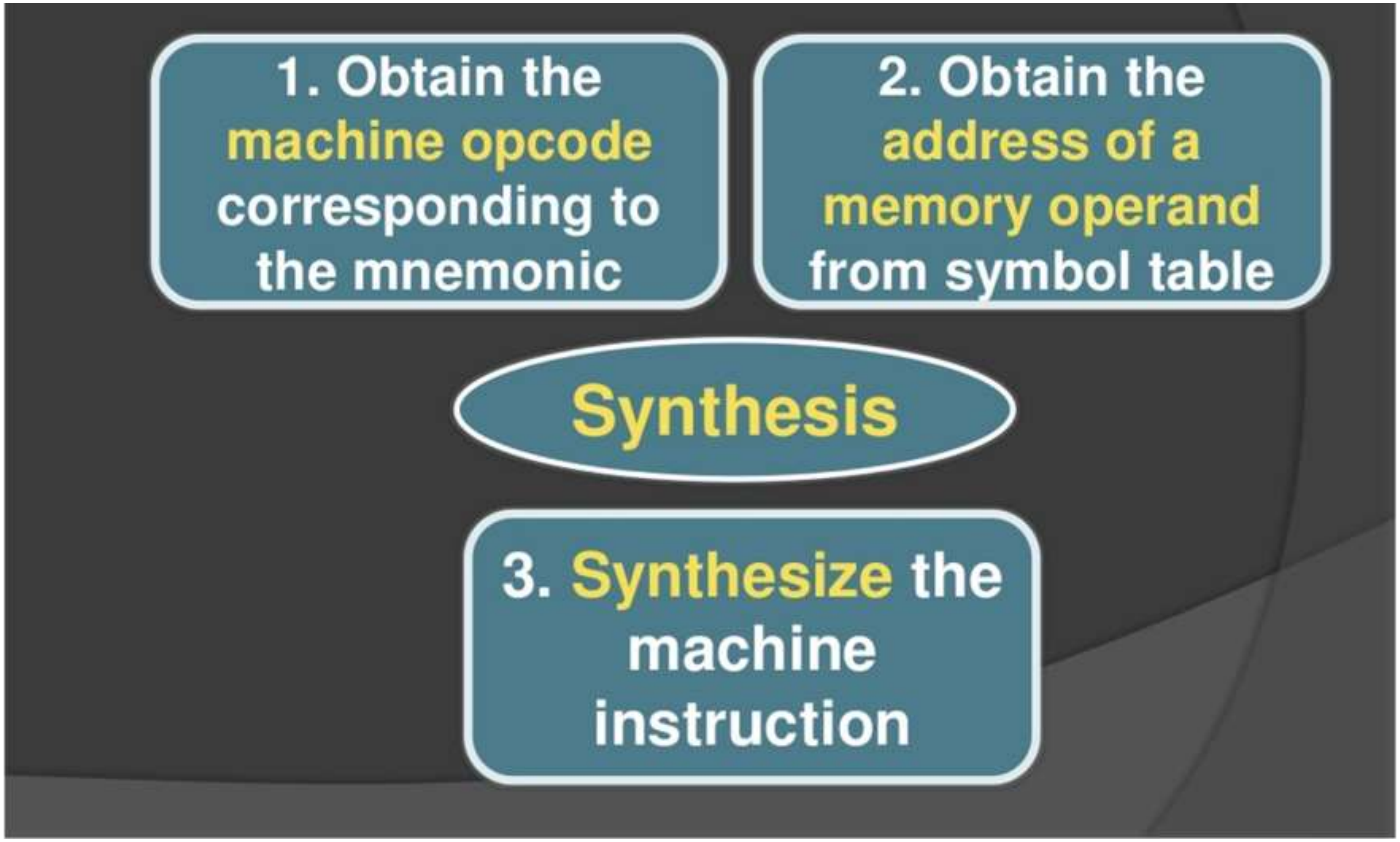
symbol	address
AGAIN	104
N	113



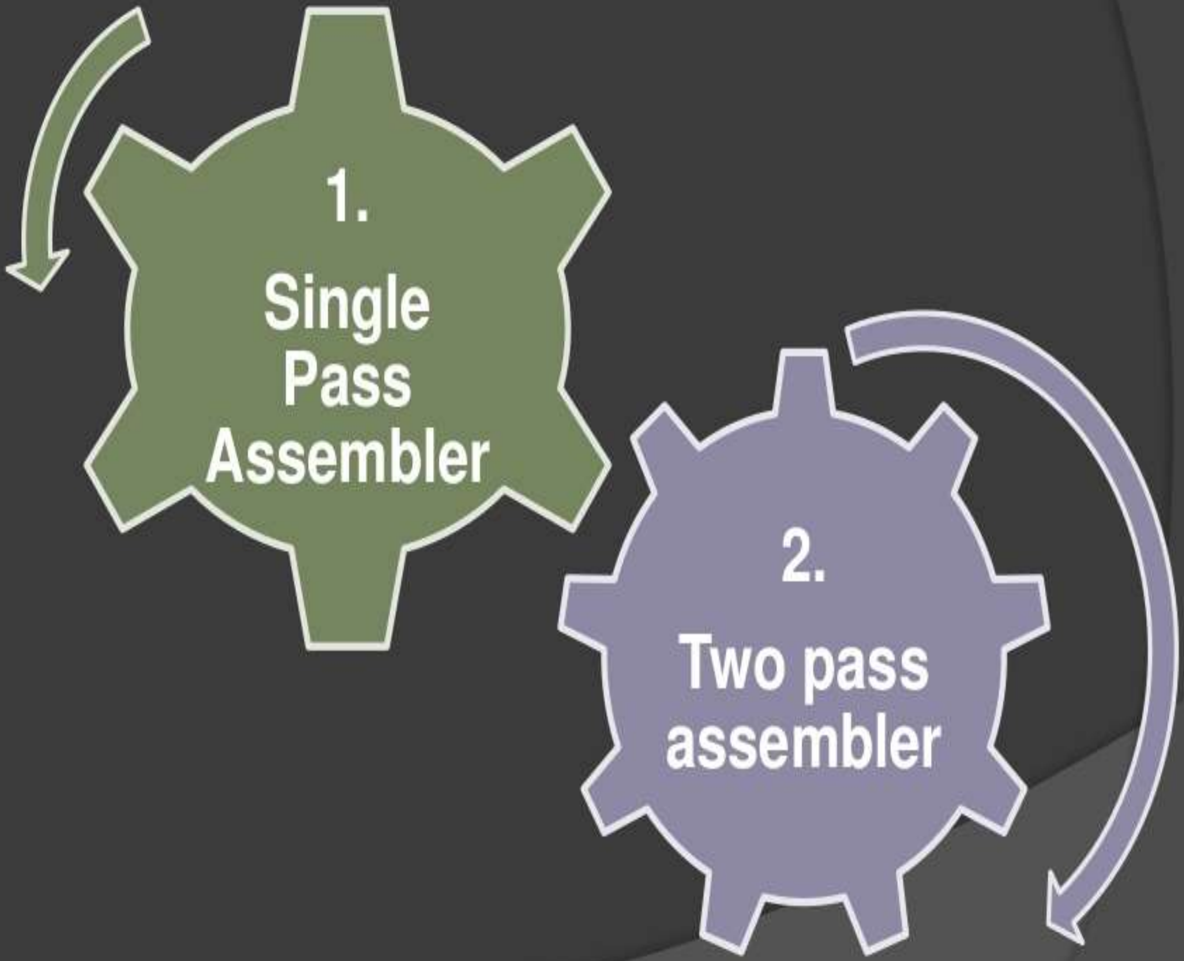
Tasks of analysis phase

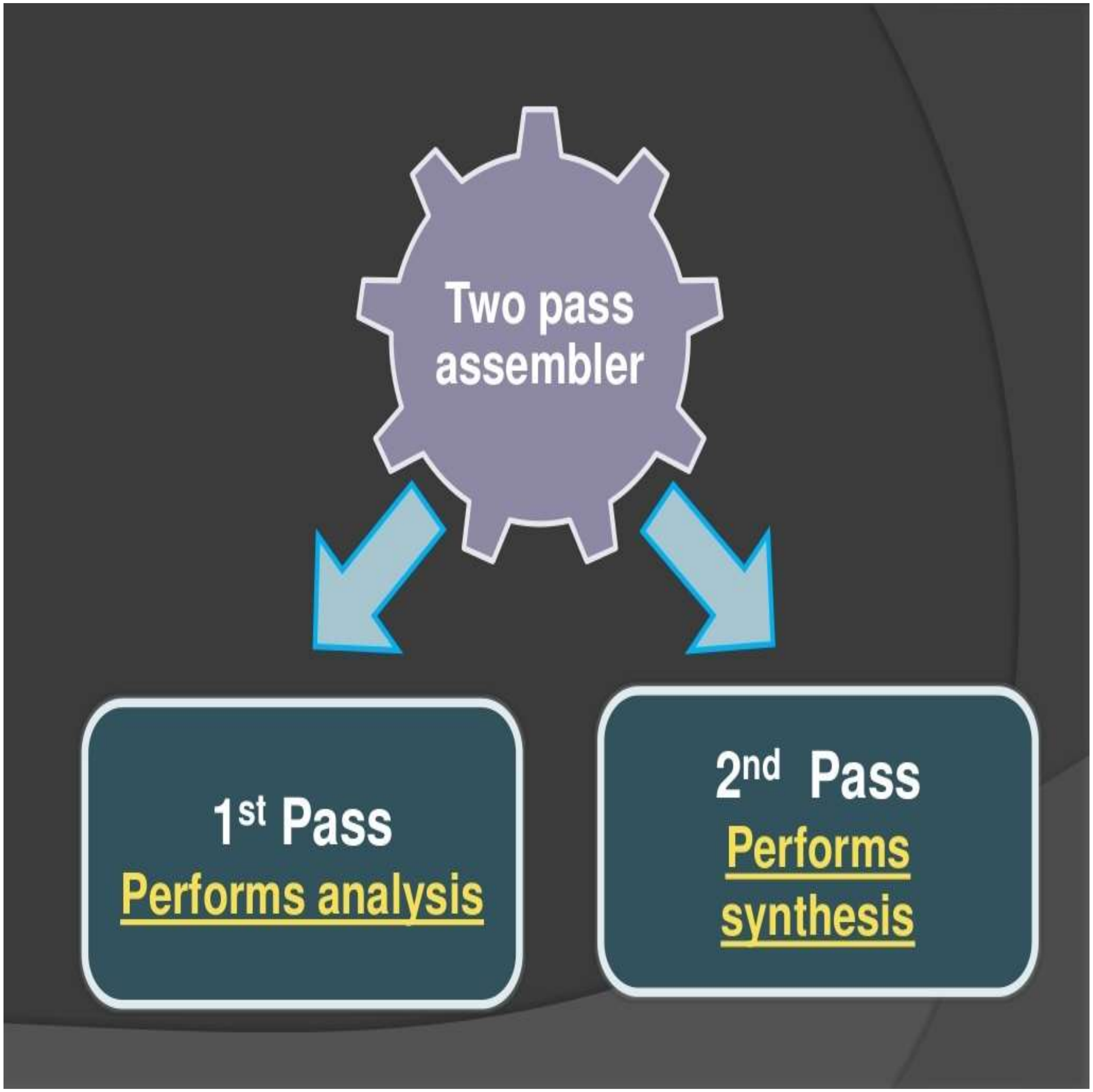
1. Separate label, opcode & operand
2. Build the symbol table
Analysis
3. Perform LC processing
4. Construct IC

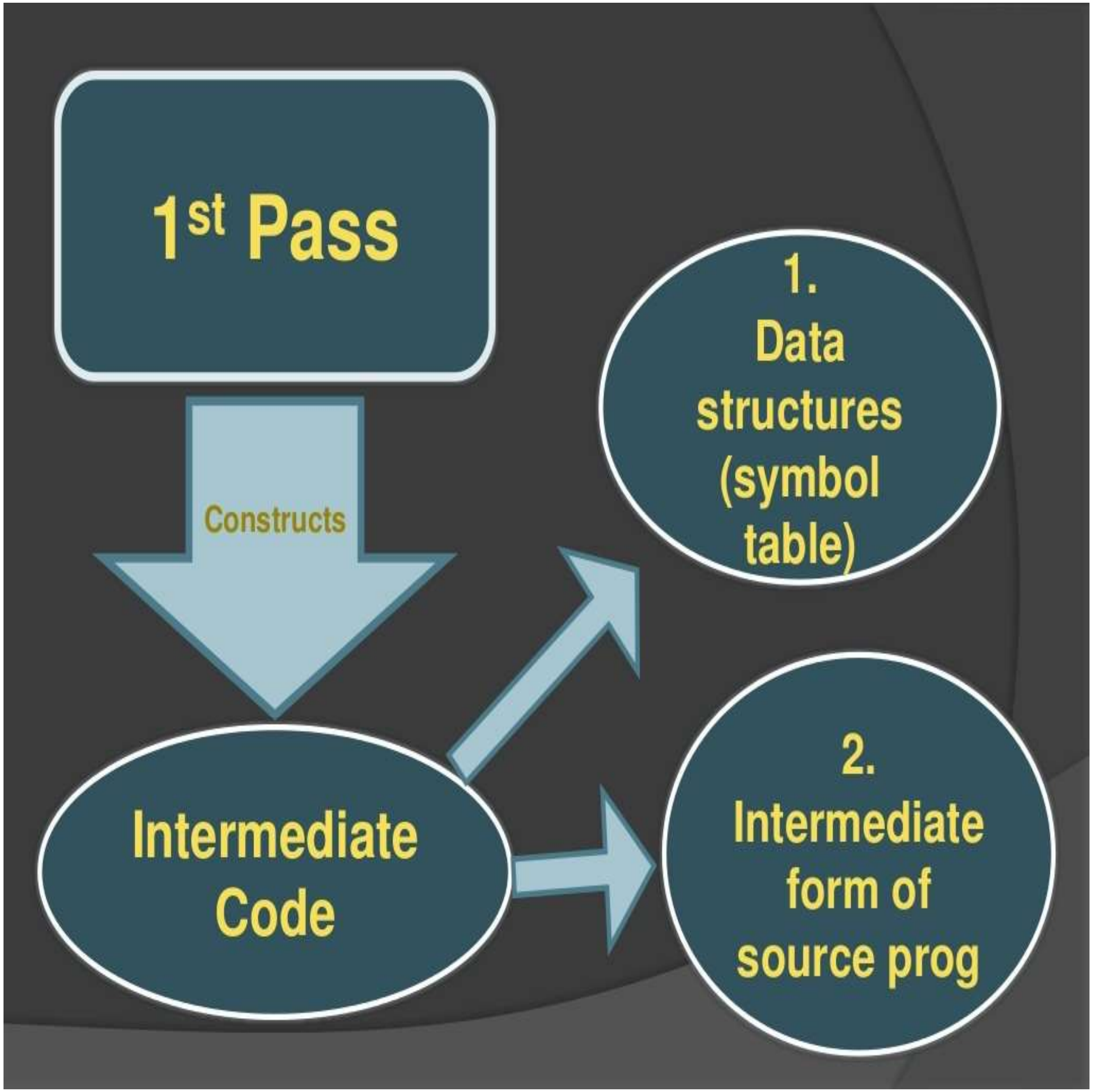
Tasks of Synthesis Phase

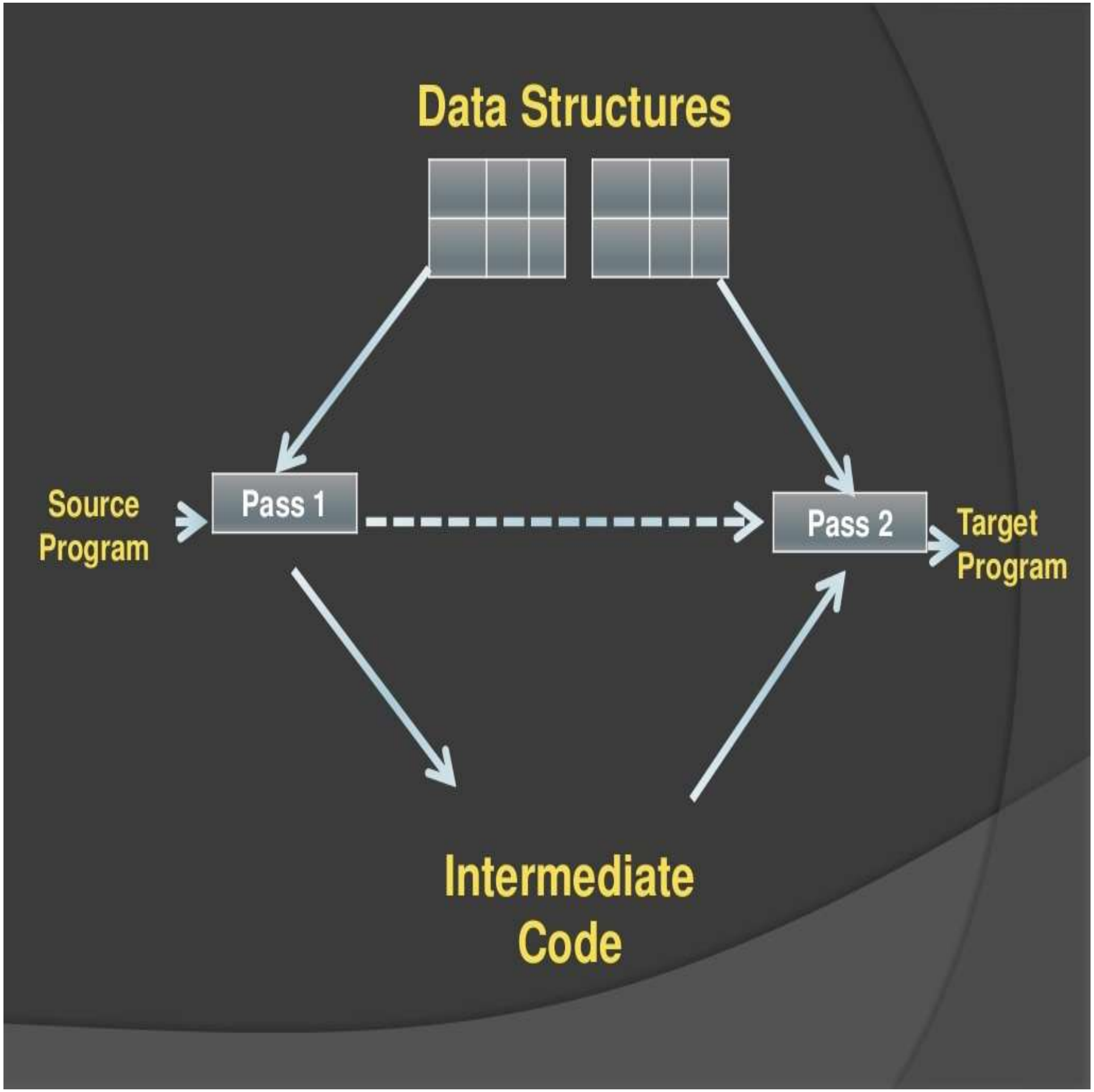

1. Obtain the machine opcode corresponding to the mnemonic
2. Obtain the address of a memory operand from symbol table
Synthesis
3. Synthesize the machine instruction

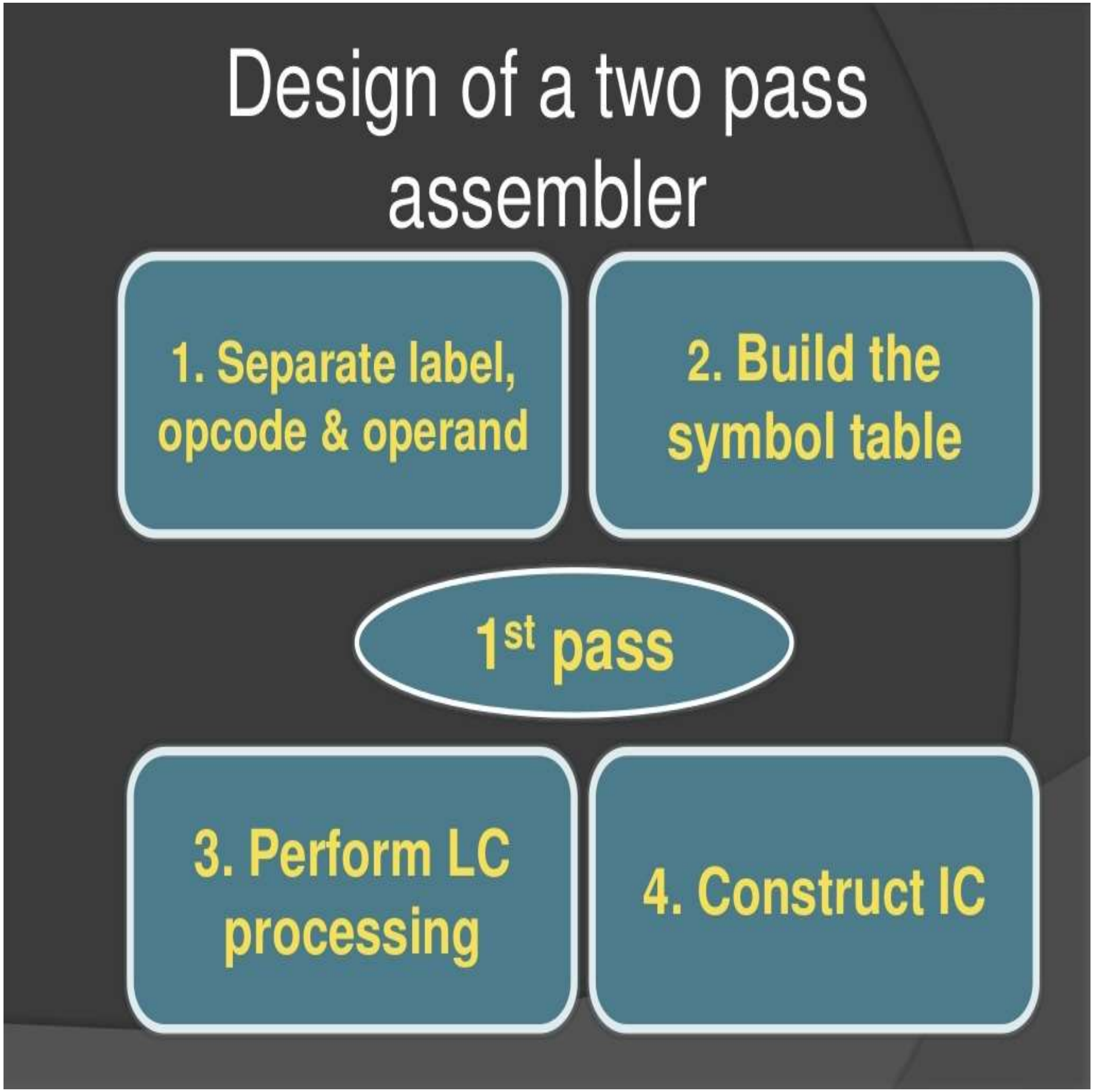
Pass structure of Assembler




Two pass assembler
1st Pass
Performs analysis
2nd Pass
Performs synthesis


1st Pass
Constructs
Intermediate Code
1. Data structures (symbol table)
2. Intermediate form of source prog


Data Structures
Source Program
Pass 1
Pass 2
Target Program
Intermediate Code



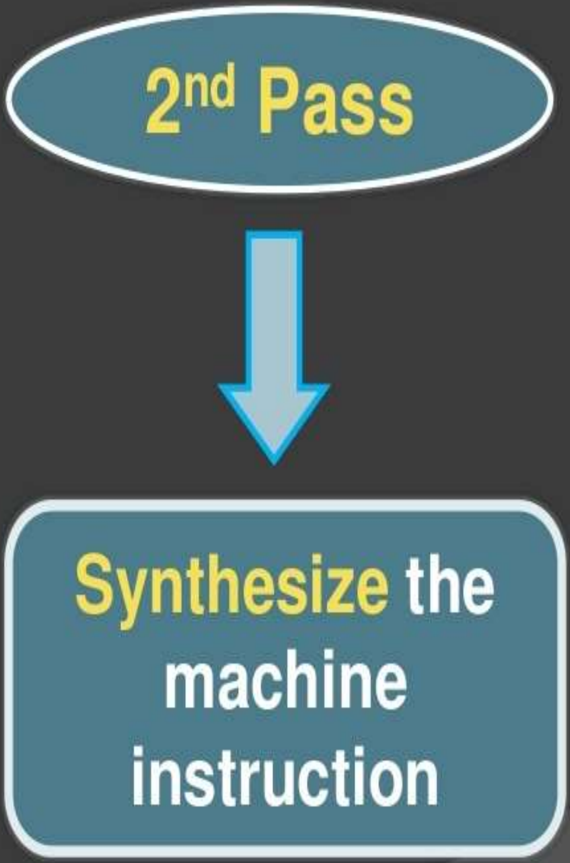
Design of a two pass assembler

1. Separate label, opcode & operand
2. Build the symbol table

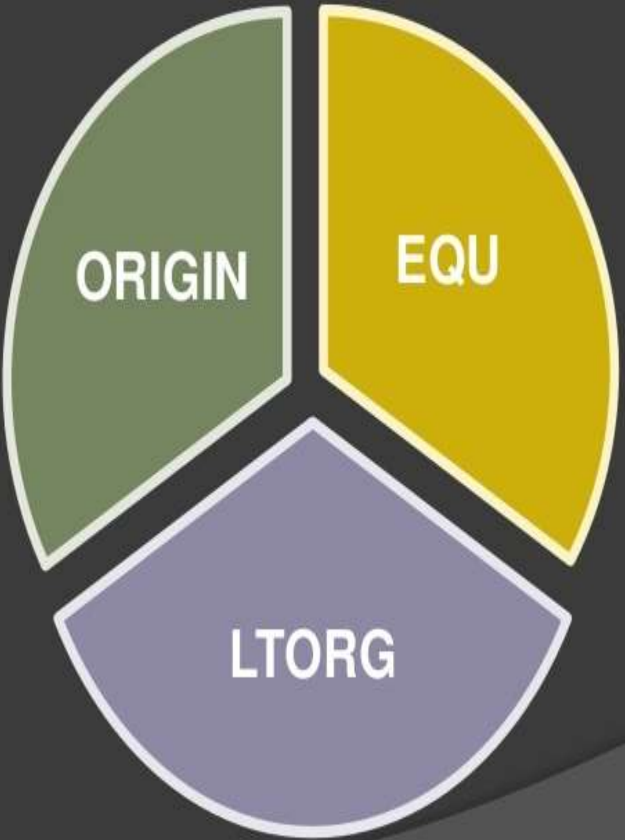
1st pass

3. Perform LC processing
4. Construct IC

2nd Pass

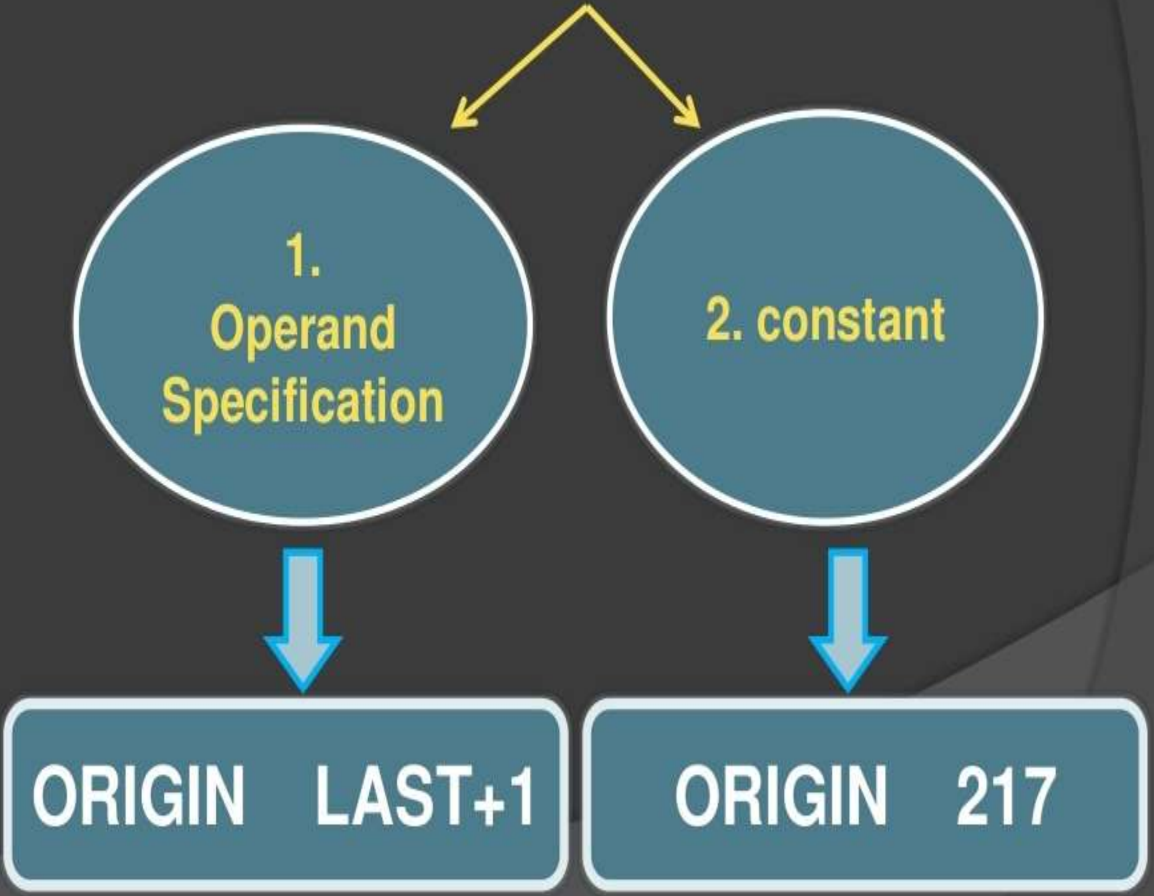


Advanced Assembler Directives



ORIGIN

- ORIGIN < Address Specification >

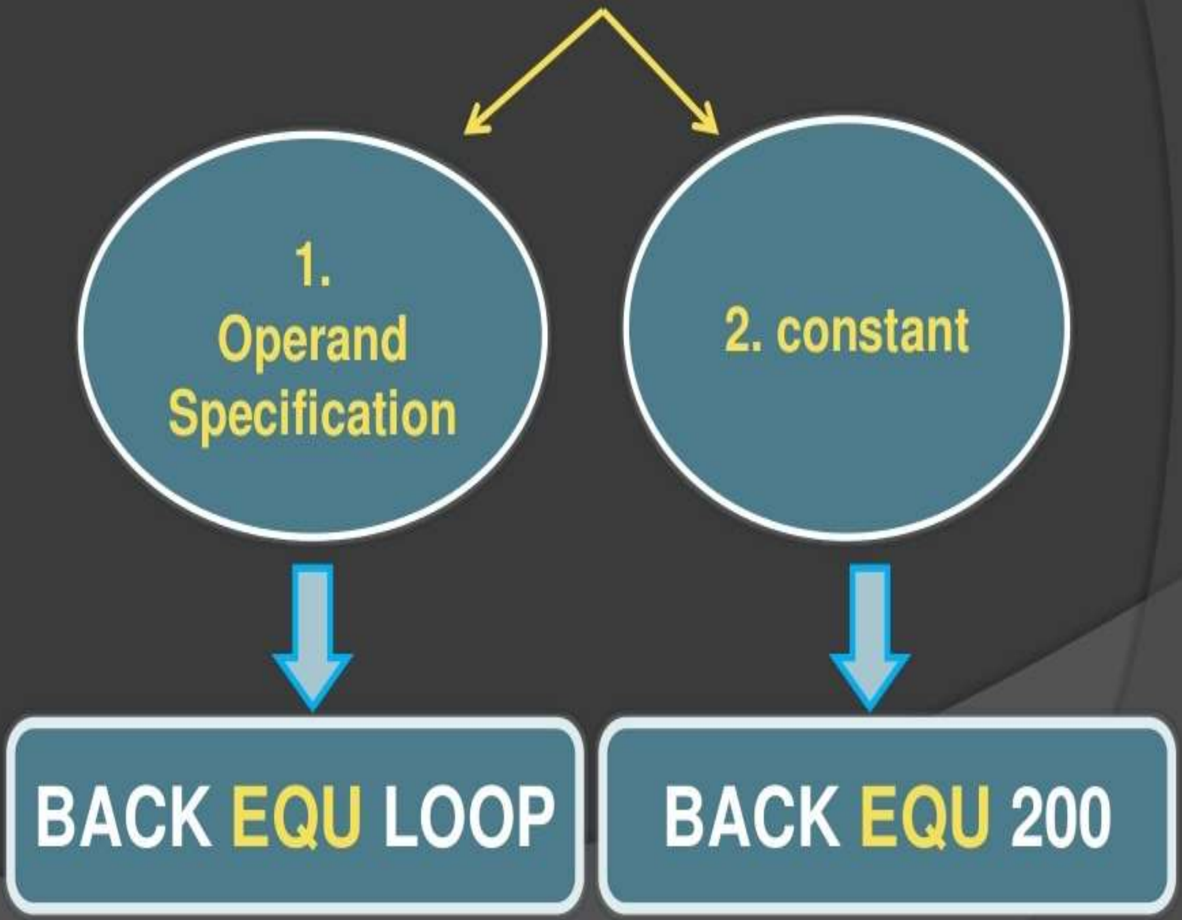


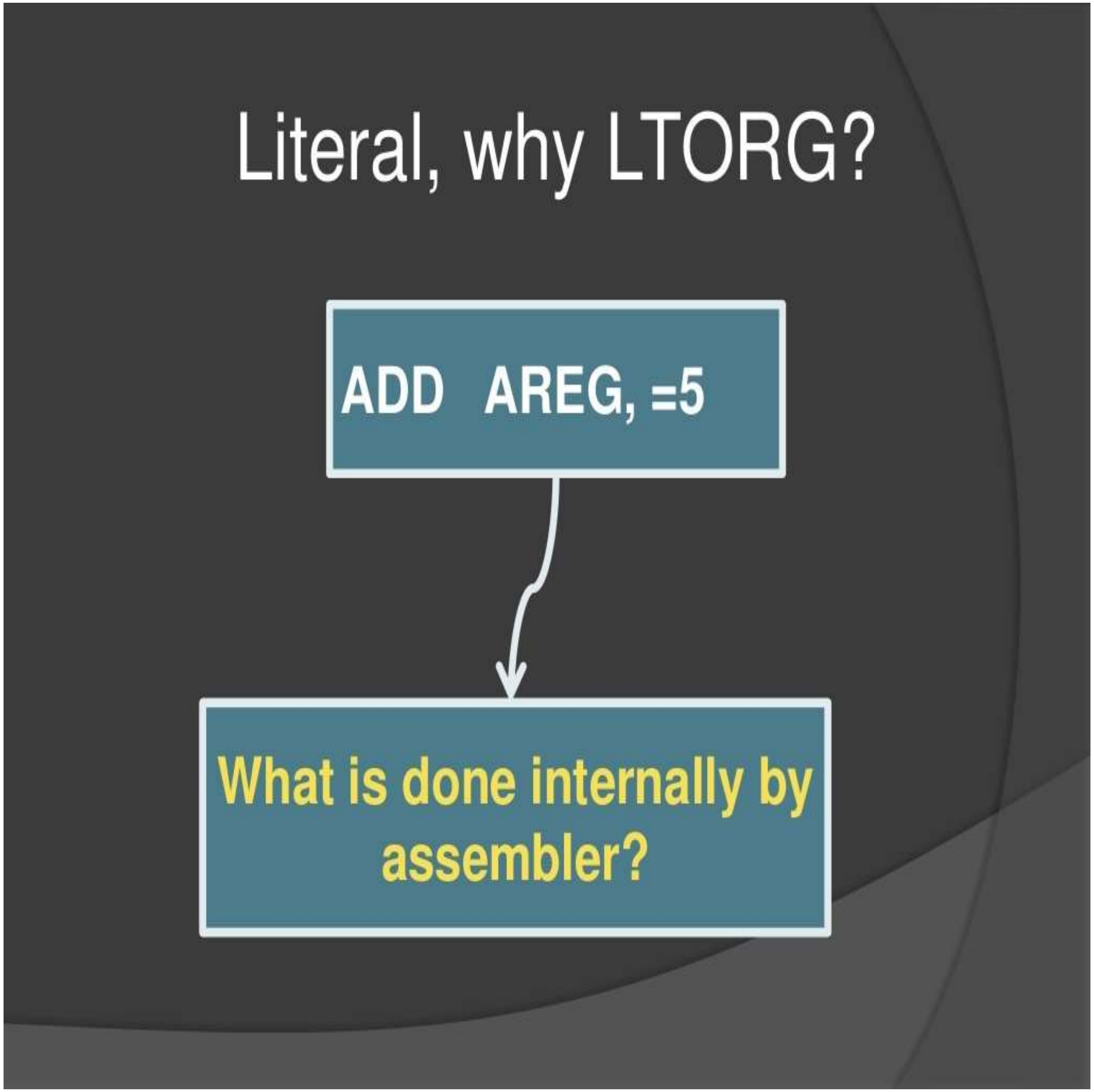
Cont…

- It is useful when your target program does not consist of **consecutive** memory words.
- Operand Specification – Ability to perform **Relative LC Processing**, not absolute.
- Difference between using both the options

EQU

- Defines the symbol to represent <add spec>
- **Symbol EQU < address specification >**





Literal, why LTORG?

```
ADD   AREG, =5
```

What is done internally by assembler?

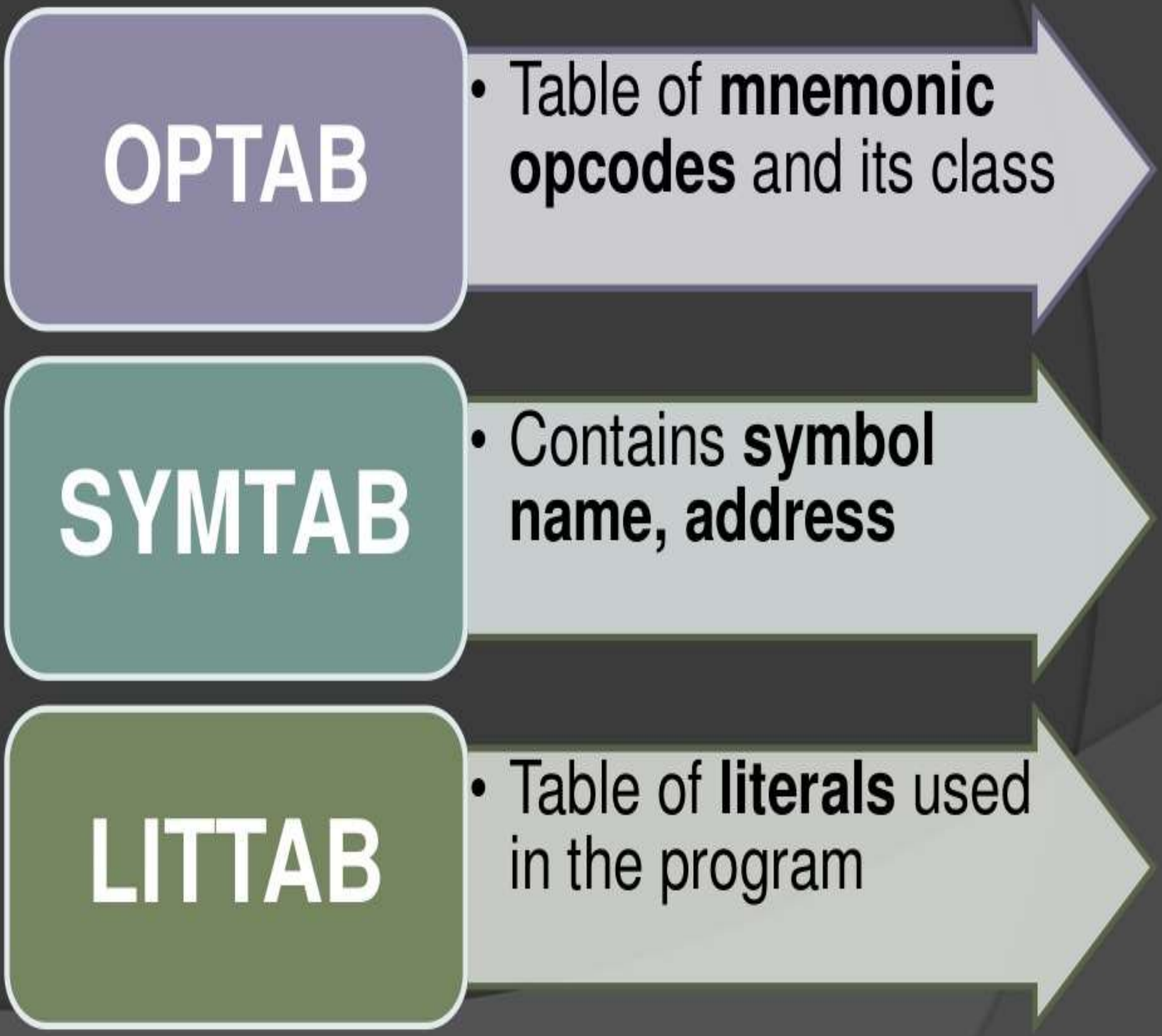
```
MOVER AREG,A          □     200
MOVEM BREG,=‘2’       □     201
ADD AREG,=‘2’         □     202
SUB BREG,=‘3’         □     203
LTORG                 □     204 (  for = ‘2’  )
                      □     205 (  for = ‘3’ )
MOVER AREG,=‘4 ’      □     206
ADD BREG,=‘2’         □     207
 A    DC   5          □     208
 END                  □     209   ( for =‘4 ‘ )
                      □     210   (  for =‘2’ )
```

LITTAB		
Literal no.	Literal	Address
1	=‘2’	204
2	=‘3’	205
3	=‘4’	209
4	=‘2’	210

POOLTAB[1]=1

POOLTAB[2]=3

Pass -1 of the Assembler



OPTAB
- Table of **mnemonic opcodes** and its class

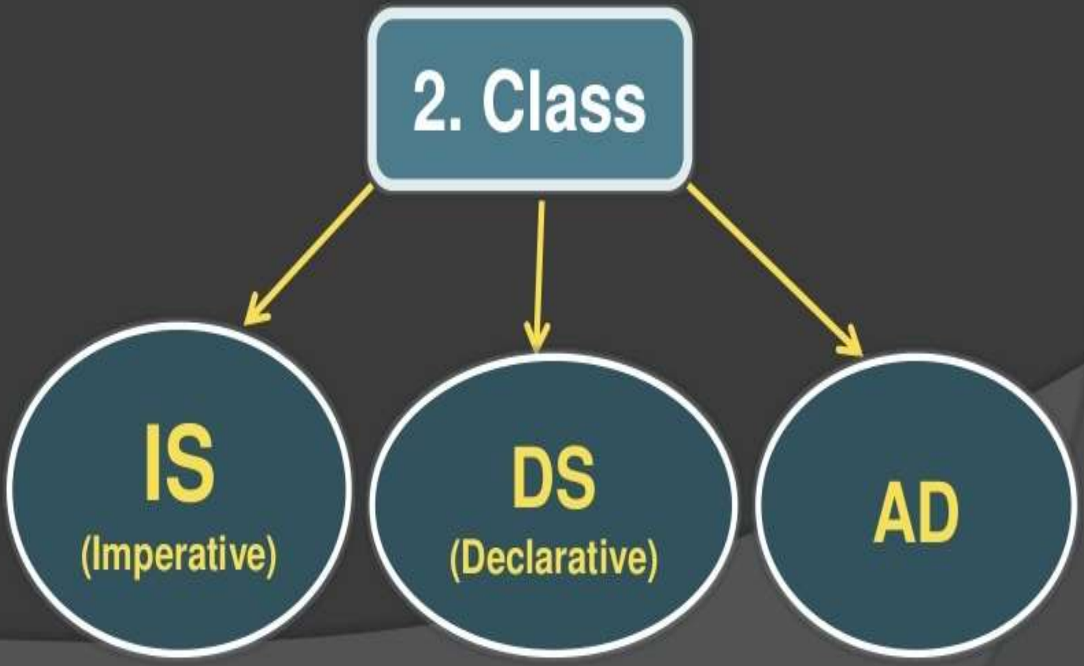
SYMTAB
- Contains **symbol name, address**

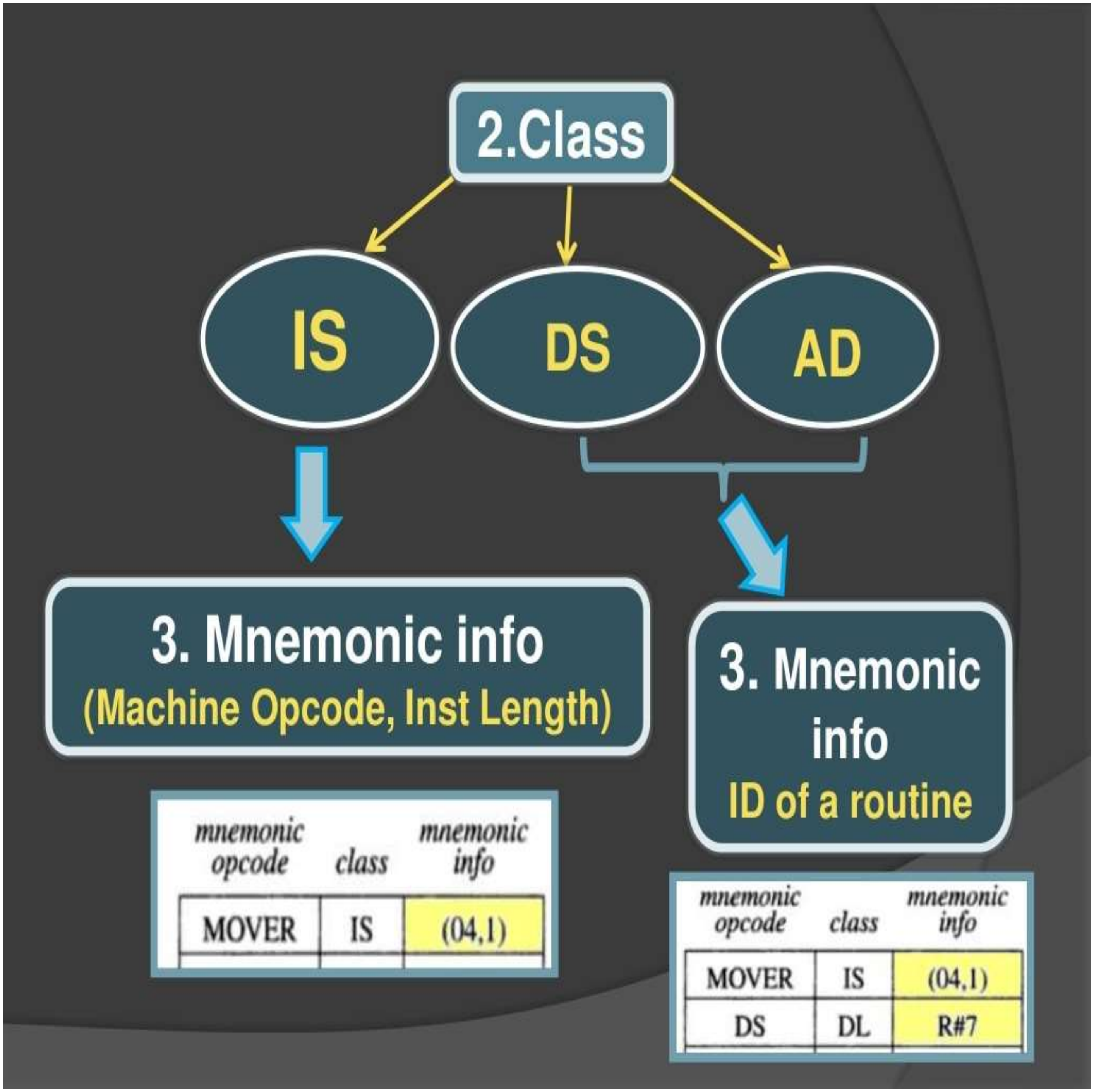
LITTAB
- Table of **literals** used in the program

OPTAB

- Contains

1. mnemonic opcode
2. class
3. mnemonic information



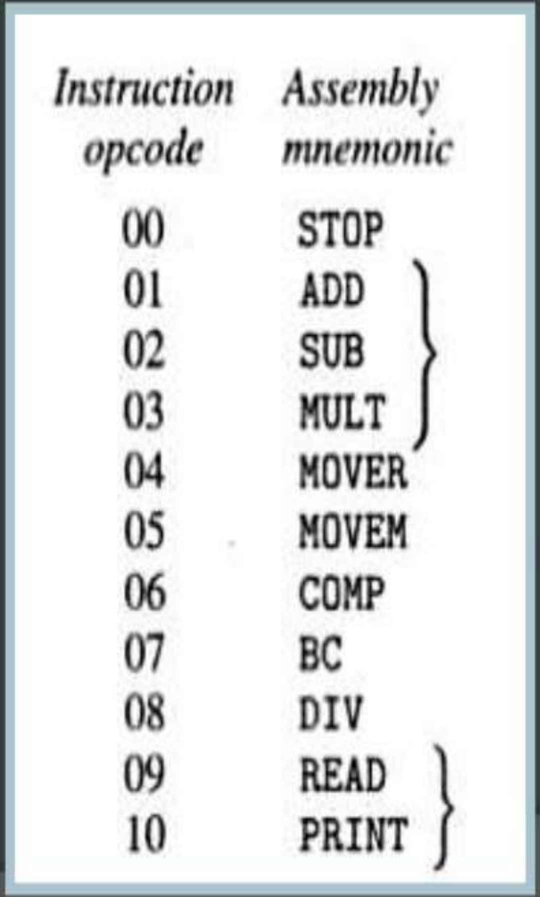


mnemonic opcode	class	mnemonic info
MOVER	IS	(04,1)

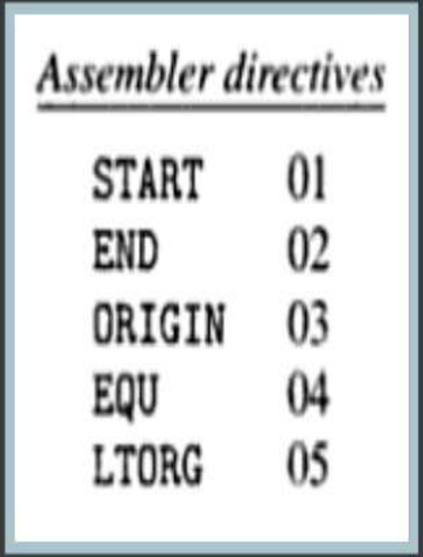
mnemonic opcode	class	mnemonic info
MOVER	IS	(04,1)
DS	DL	R#7

Opcode format

- **(Statement Class, Code)**

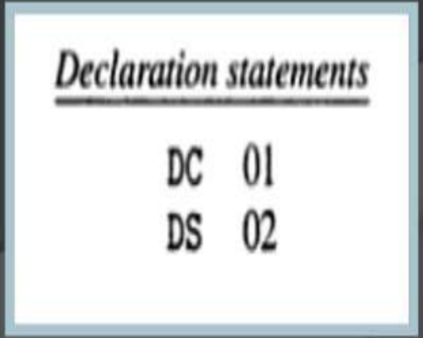


Instruction opcode	*Assembly mnemonic*
00	STOP
01	ADD
02	SUB
03	MULT
04	MOVER
05	MOVEM
06	COMP
07	BC
08	DIV
09	READ
10	PRINT



Assembler directives

START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

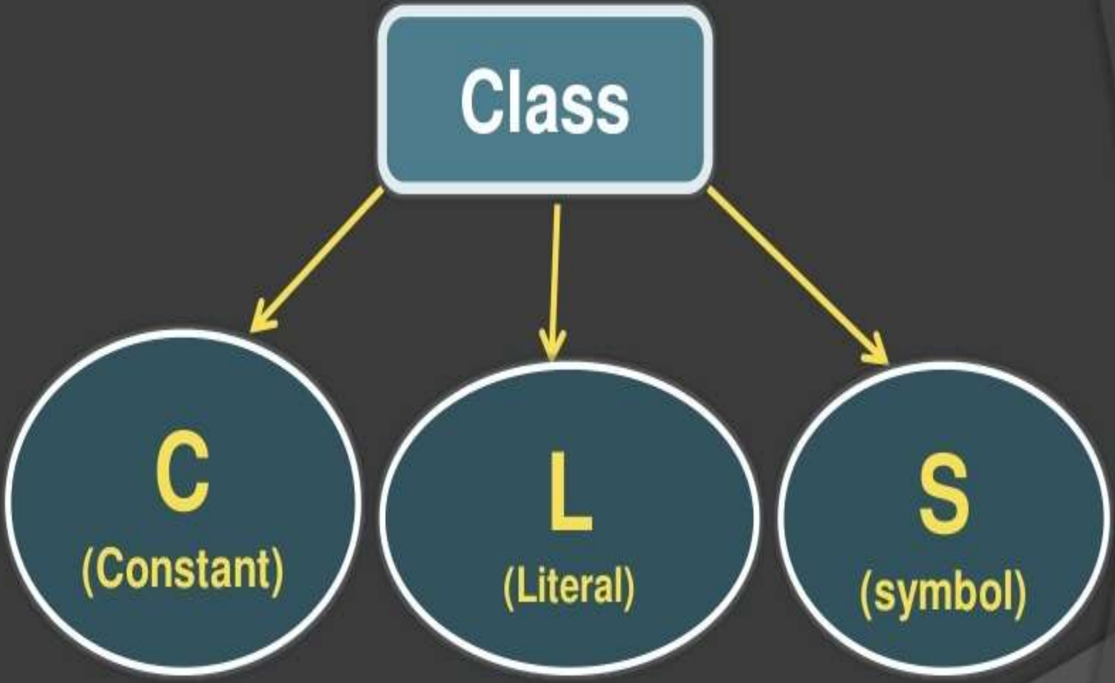


Declaration statements

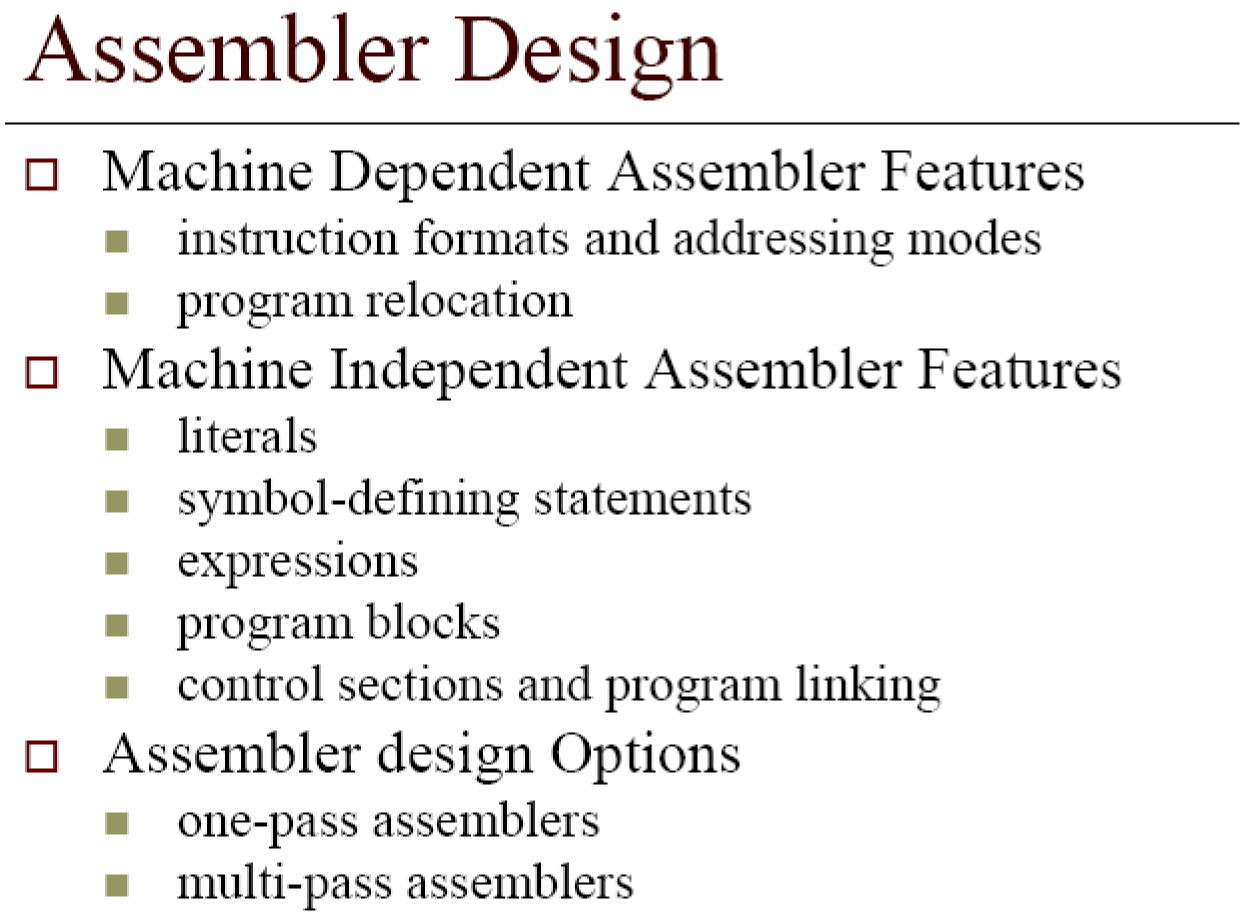
DC	01
DS	02

Operand Specification

- (Operand Class, Code)



Features of Assembler



Assembler Design

- Machine Dependent Assembler Features
  - instruction formats and addressing modes
  - program relocation
- Machine Independent Assembler Features
  - literals
  - symbol-defining statements
  - expressions
  - program blocks
  - control sections and program linking
- Assembler design Options
  - one-pass assemblers
  - multi-pass assemblers

OPTAB

Instruction (Mnemonic/ Declaration/ Assembler Directive)	Statement Class	Machine Code
STOP	IS	00
ADD	IS	01
SUB	IS	02
MULT	IS	03
MOVER	IS	04
MOVEM	IS	05
COMP	IS	06
BC	IS	07
DIV	IS	08
READ	IS	09
PRINT	IS	10
DC	DL	01
DS	DL	02
START	AD	01
END	AD	02
ORIGIN	AD	03
EQU	AD	04
LTORG	AD	05

Register Table

Reg name	M/c Code
AREG	1
BREG	2
CREG	3
DREG	4

Condition Code for BC Instr

Condition	M/c Code
LT	1
LE	2
EQ	3
GT	4
GE	5
ANY(NE)	6

Stmt No	Example ALP	
1	START 200	
2	MOVER AREG, ='5'	
3	MOVEM AREG, A	
4	LOOP	MOVER AREG, A
5	MOVER CREG, B	
6	ADD CREG, ='1'	
7	BC ANY, NEXT	
8	LTORG	
9	NEXT	SUB AREG, ='1'
10	BC LT, BACK	
11	LAST	STOP
12	ORIGIN LOOP+2	
13	MULT CREG, B	
14	ORIGIN LAST+1	
15	A	DS 1
16	BACK	EQU LOOP
17	B	DC 1
18	END	

Stmt No	ALP	Intermediate Code			
1	START 200		AD,01		C,200
2	MOVER AREG, =‘5’	200	IS,04	1	L,1
3	MOVEM AREG, A	201	IS,05	1	S,1
4	LOOP MOVER AREG, A	202	IS,04	1	S,1
5	MOVER CREG, B	203	IS,04	3	S,3
6	ADD CREG, =‘1’	204	IS,01	3	L,2
7	BC ANY, NEXT	205	IS,07	6	S,4
8	LTORG	206	AD,05	-	005
		207	AD,05	-	001
9	NEXT SUB AREG, =‘1’	208	IS,02	1	L,3
10	BC LT, BACK	209	IS,07	1	S,5
11	LAST STOP	210	IS,00	-	-
12	ORIGIN LOOP+2		AD,03	-	(S,2)+2
13	MULT CREG, B	204	IS,03	3	S,3
14	ORIGIN LAST+1		AD,03	-	(S,6)+1
15	A DS 1	211	DL,02	-	C,1
16	BACK EQU LOOP		AD,04	-	S,2
17	B DC 1	212	DL,01		C,1
18	END	213	AD,02	-	001

Stmt No	ALP	Intermediate Code				Machine Code			
1	START 200		AD,01		C,200				
2	MOVER AREG, =‘5’	200	IS,04	1	L,1	200	04	1	206
3	MOVEM AREG, A	201	IS,05	1	S,1	201	05	1	211
4	LOOP MOVER AREG, A	202	IS,04	1	S,1	202	04	1	211
5	MOVER CREG, B	203	IS,04	3	S,3	203	05	03	212
6	ADD CREG, =‘1’	204	IS,01	3	L,2	204	01	03	207
7	BC ANY, NEXT	205	IS,07	6	S,4	205	07	6	208
8	LTORG	206	AD,05	-	005	206	00	0	005
		207	AD,05	-	001	207	00	0	001
9	NEXT SUB AREG, =‘1’	208	IS,02	1	L,3	208	02	1	213
10	BC LT, BACK	209	IS,07	1	S,5	209	07	1	202
11	LAST STOP	210	IS,00	-	-	210	00	0	000
12	ORIGIN LOOP+2		AD,03	-	(S,2)+2				
13	MULT CREG, B	204	IS,03	3	S,3	204	03	3	212
14	ORIGIN LAST+1		AD,03	-	(S,6)+1				
15	A DS 1	211	DL,02	-	C,1	211			
16	BACK EQU LOOP		AD,04	-	S,2				
17	B DC 1	212	DL,01		C,1	212			
18	END	213	AD,02	-	001	213	00	0	001

Data Structures Generated after Pass I

SYMTAB			
Sym_id	Sym_name	Sym_addr	length
1	A	211	1
2	LOOP	202	1
3	B	212	1
4	NEXT	208	1
5	BACK	202	1
6	LAST	210	1

LITTAB		
Lit_no	Literal	addr
1	=‘5’	206
2	=‘1’	207
3	=‘1’	213

POOLTTAB
POOLTAB[1]=1
POOLTAB[2]=3

ALP with Data structures Generated

	START 200	
	MOVER AREG,='5'	200
	MOVEM AREG,A	201
LOOP	MOVER AREG,A	202
	MOVER CREG, B	203
	ADD CREG,='1'	204
	LTORG	205 206
NEXT1	SUB AREG,='1'	207
	ORIGIN LOOP+8	
	MUL CREG,B	210
A	DS 2	211
B	DC 3	213
NEXT2	EQU LOOP	
	END	214

	SYMBOL TABLE		
Sym_id	Sym_name	Sym_addr	length
1	A	211	2
2	LOOP	202	1
3	B	213	1
4	NEXT1	207	1
5	NEXT2	202	1

LITERAL TABLE		
Lit_no	Literal	addr
LP1 { 1	5	205
2	1	206
LP 2 { 3	1	214

POOL TABLE
POOLTAB[1] = 1
POOLTAB[2] = 3

Algorithm for Pass 1 of 2 pass Assembler

```
Step 1:  loc_cntr : = 0;    pooltab_ptr := 1;    POOLTAB[1] :=1;
         Littab_ptr :=1;   symtab_ptr:=1;

Step 2: While next stmt is not an END stmt
    (a) If label is present then ...................
    (b) If an LTORG stmt then .................
    (c ) If START or ORIGIN stmt then .............
    (d) If an EQU stmt then ……………….
    (e) If a declaration stmt then ………………
    (f) If an imperative stmt then ……………….

Step 3: Processing of END stmt
    (a) Perform step 2(b)
    (b) Generate Intermediate code.
    (C) Goto Pass 2 .
```

Label Processing

```
(a) If label is present then
        this_label = symbol in label field
        Enter(this_label, loc_cntr) in SYMTAB
```

◄

MIT-WPU

Literal Processing

(b) If an LTORG stmt then

```
        i. Process literals
    LITTAB[POOLTAB[pooltab_ptr]]……
    LITTAB[POOLTAB[pooltab_ptr+1]]-1]
        to allocate memory  and put the address field.
        Update loc_cntr.
        ii. Pooltab_ptr:=pooltab_ptr+1;
        iii. POOLTAB[pooltab_ptr] := littab_ptr;
```

◄

START ORIGIN and EQU Processing

```
(c ) If START or ORIGIN stmt then
        loc_cntr := value in operand field

(d) If an EQU stmt then
        this_addr=value of <address spec>
        update the SYMTAB entry
```

◄

Declaration(DS/DC) Statement Processing

```
(e) If a declaration stmt then
      code = code of declaration stmt,
      size = size req by DC/DS
      update the SYMTAB entry
      loc_cntr = loc_cntr + size
      generate Intermediate code
```

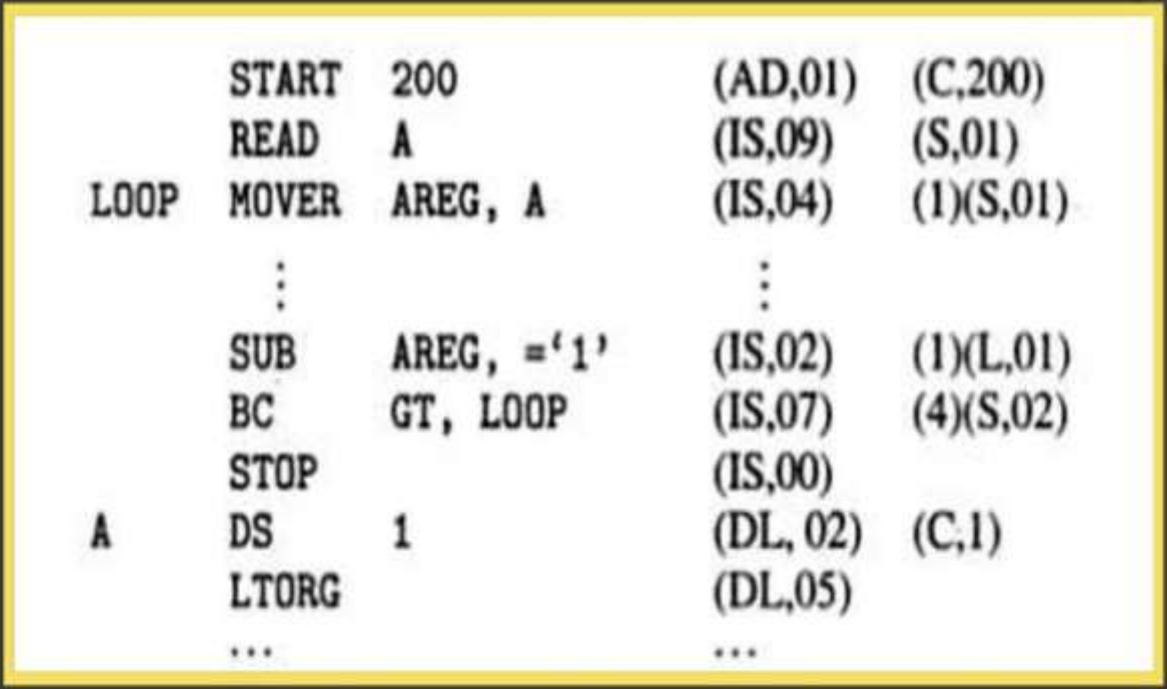
◄

Imperative Statement Processing

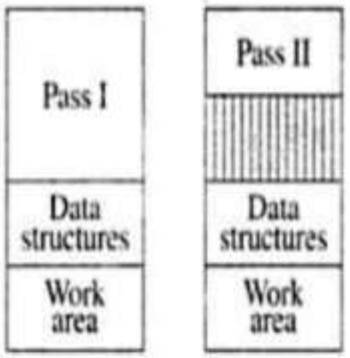
```
(f) If an imperative stmt then
        code = m/c code from MOT,
        loc_cntr = loc_cntr + length of instr
        If operand is literal then
                this_lit = literal in operand field
                LITTAB[littab_ptr] = this_lit
                littab_ptr = littab_ptr + 1
        else    this_entry = SYMTAB entry no of operand
                symtab_ptr = symtab_ptr + 1
```

◄

Variants of IC

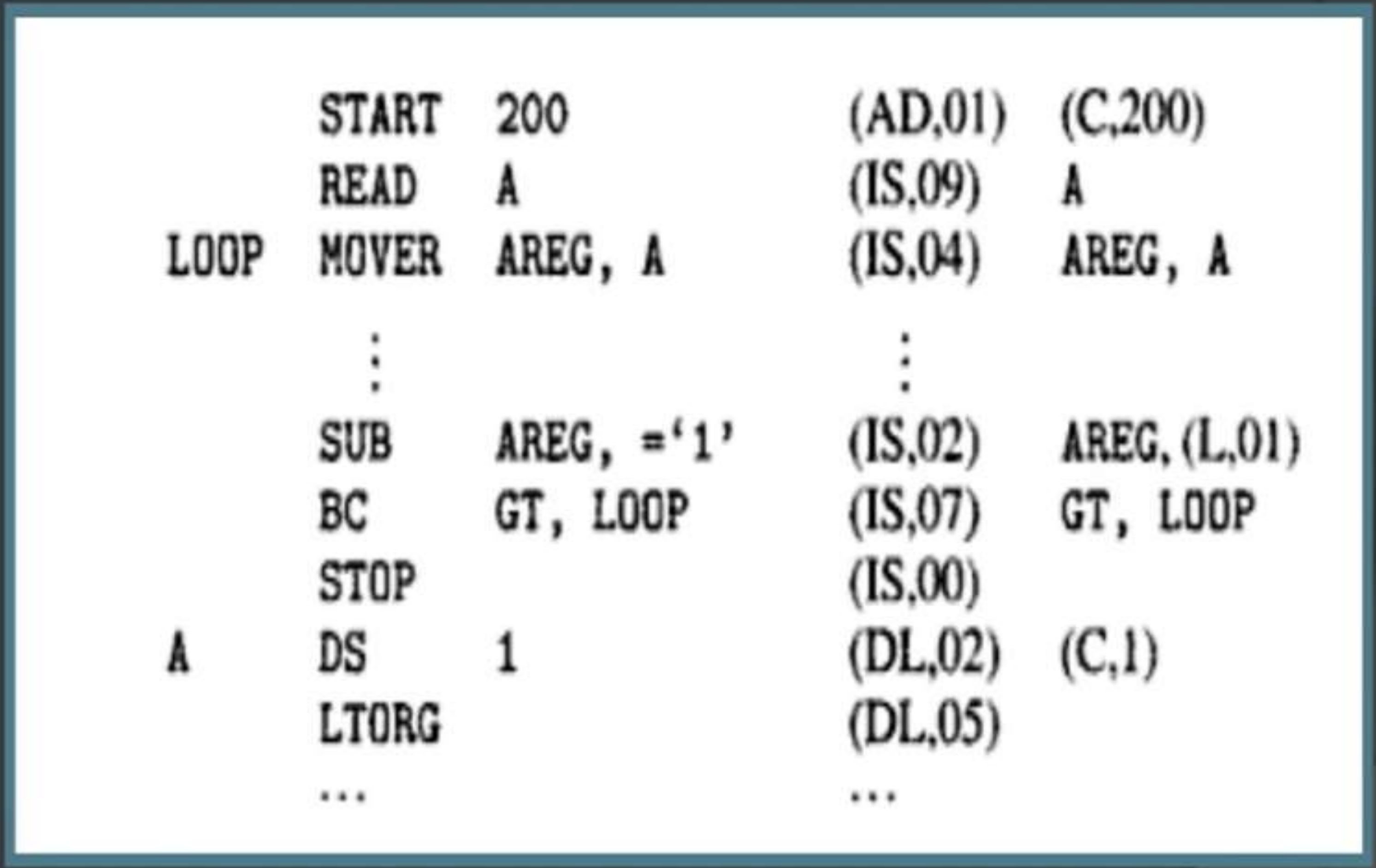


```
        START   200             (AD,01)   (C,200)
        READ    A               (IS,09)   (S,01)
LOOP    MOVER   AREG, A         (IS,04)   (1)(S,01)
          :                       :
        SUB     AREG, ='1'      (IS,02)   (1)(L,01)
        BC      GT, LOOP        (IS,07)   (4)(S,02)
        STOP                    (IS,00)
A       DS      1               (DL, 02)  (C,1)
        LTORG                   (DL,05)
        ...                     ...
```



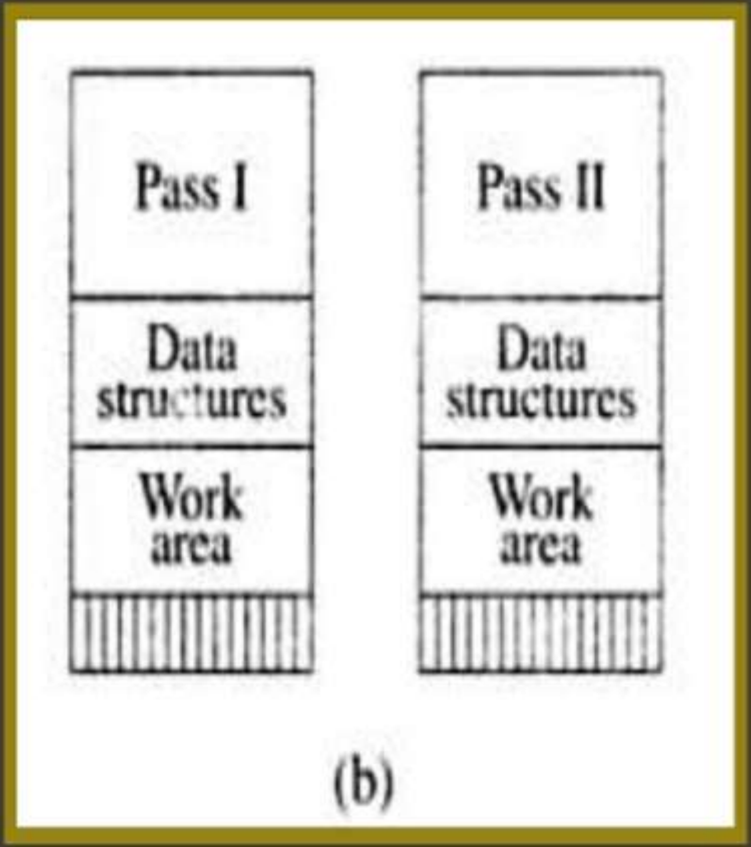
(a)

Variant - 2 of IC



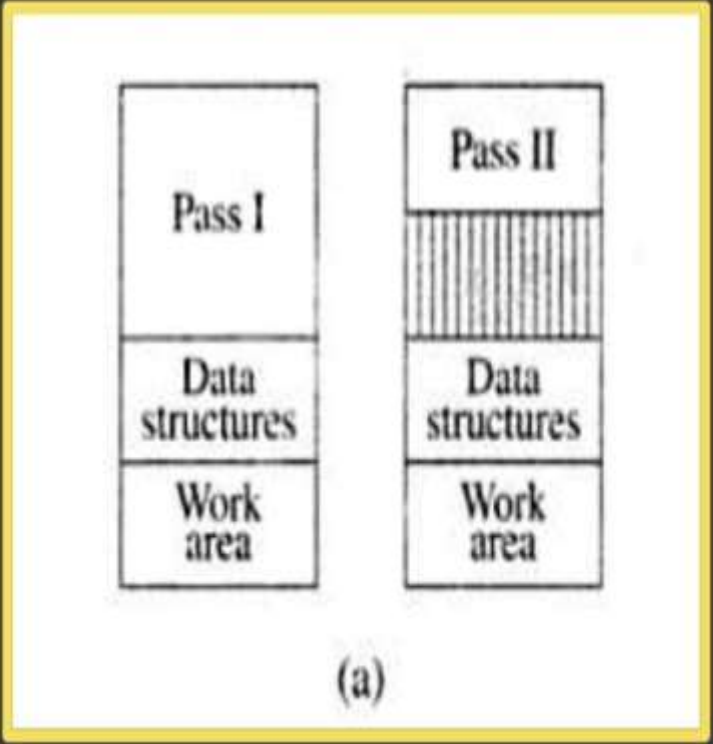
```
        START   200              (AD,01)   (C,200)
        READ    A                (IS,09)   A
LOOP    MOVER   AREG, A          (IS,04)   AREG, A
          :                        :
        SUB     AREG, ='1'       (IS,02)   AREG, (L,01)
        BC      GT, LOOP         (IS,07)   GT, LOOP
        STOP                     (IS,00)
A       DS      1                (DL,02)   (C,1)
        LTORG                    (DL,05)
        ...                      ...
```

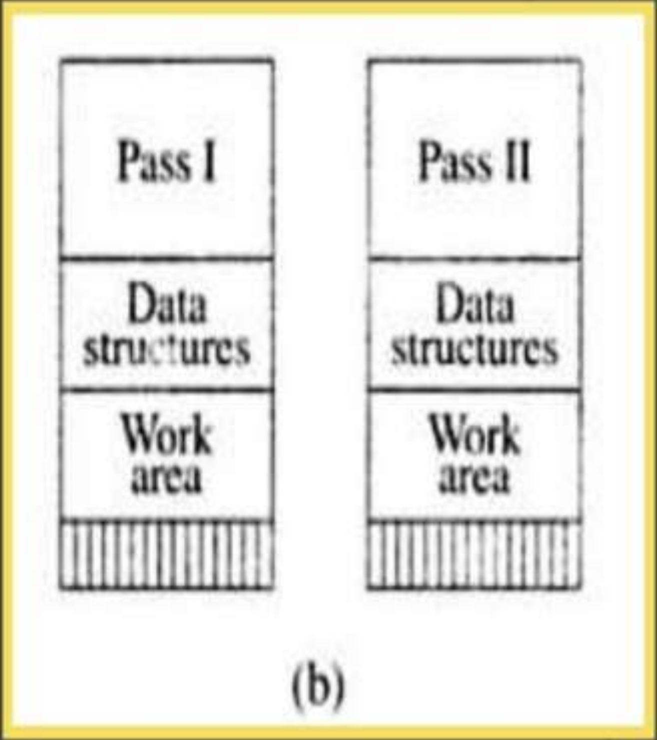
Variant-2 of IC



(b)

Variants of IC





✓Extra work in Pass I
✓Simplified Pass II
✓Pass I code occupies more memory than code of Pass II
✓Does not simplify the task of Pass II or save much memory in some situation.

✓IC is less compact
✓Memory required for two passes would be better balanced
✓So better memory utilization

Algorithm for Pass 2 of 2 pass Assembler

Machine_code_buffer :area for constructing code for one statement

Code_area: area for assembling the target program ,

code_area_address :contains address of code_area

```
1. code_area_addr=addr of code_area,
   pooltab_ptr=1,
   loc_cntr=0
```

Algorithm for Pass 2 of 2 pass Assembler (contd…)

```
2.  While next stmt is not an END stmt
(a) clear machine_code_buffer
(b) If an LTORG stmt
    (i) Process literals in
    LITTAB[POOLTAB[pooltab_ptr]]……..LITTAB[POOLTAB[pooltab_ptr+1]-1]
    assemble literals in machine_code_buffer.
    (ii) size=size of memory req for literals          (iii)pooltab_ptr=pooltab_ptr+1
(c) If a START or ORIGIN stmt then
    (i) loc_cntr=value specified in operand field      (ii) size=0
(d) If a declaration stmt
    (i) If a DC stmt then
            Assemble the const in machine_code_buffer.
    (ii) size=size of memory required by DC/DS stmt
(e) If an imperative stmt
    (i) Get operand address from SYMTAB or LITTAB
    (ii) Assemble instr in machine_code_buffer.        (iii) size=size of instr
(f) If size <> 0 then
    (i) Move contents of machine_code_buffer  to the address code_area_addr+loc_cntr
    (ii) loc_cntr=loc_cntr+size
3.Processing of END stmt
(a) Perform step 2(b) and 2(f)
(b) Write code_area into output file.
```

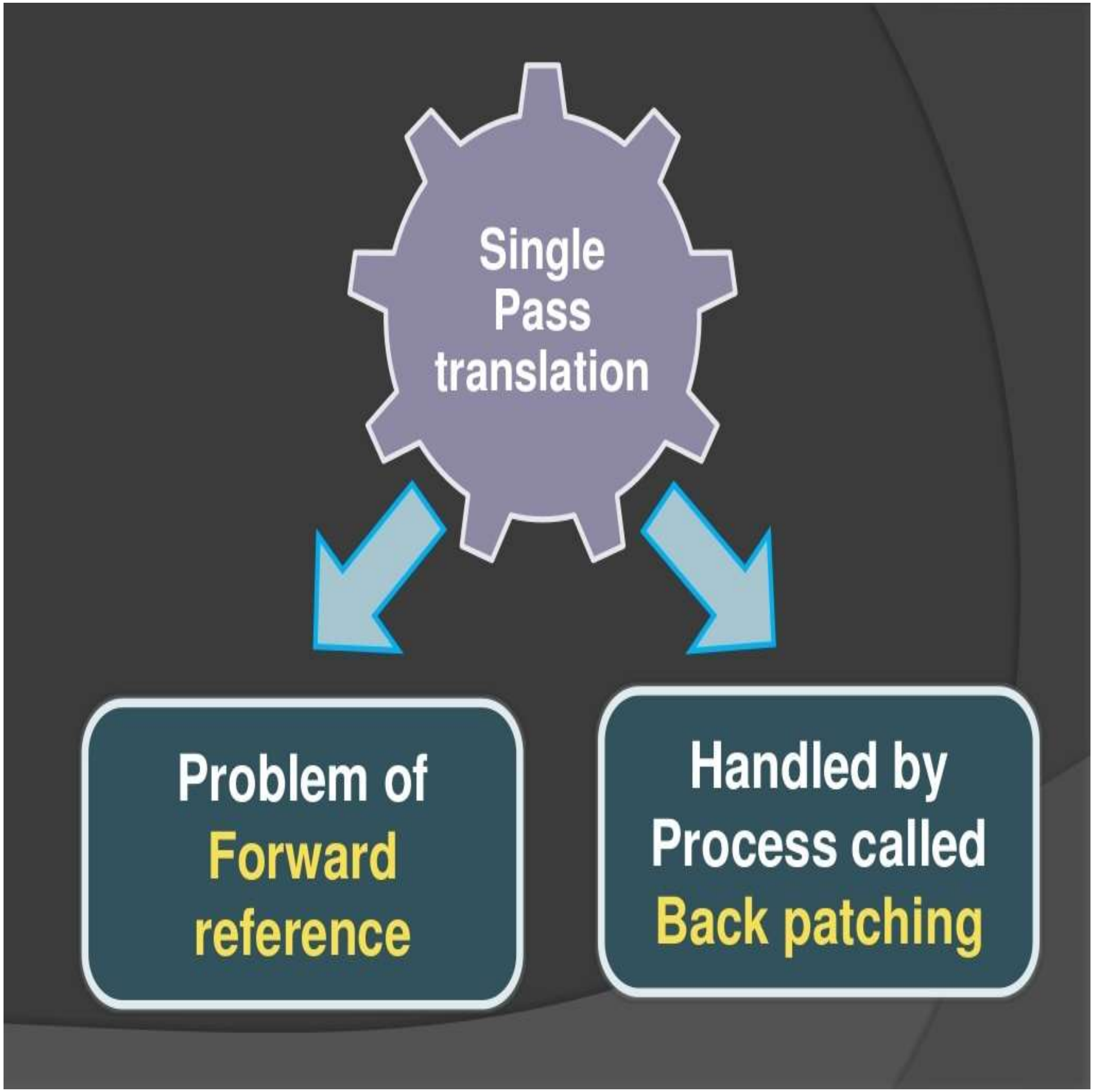
Listing and error reporting in Pass-1

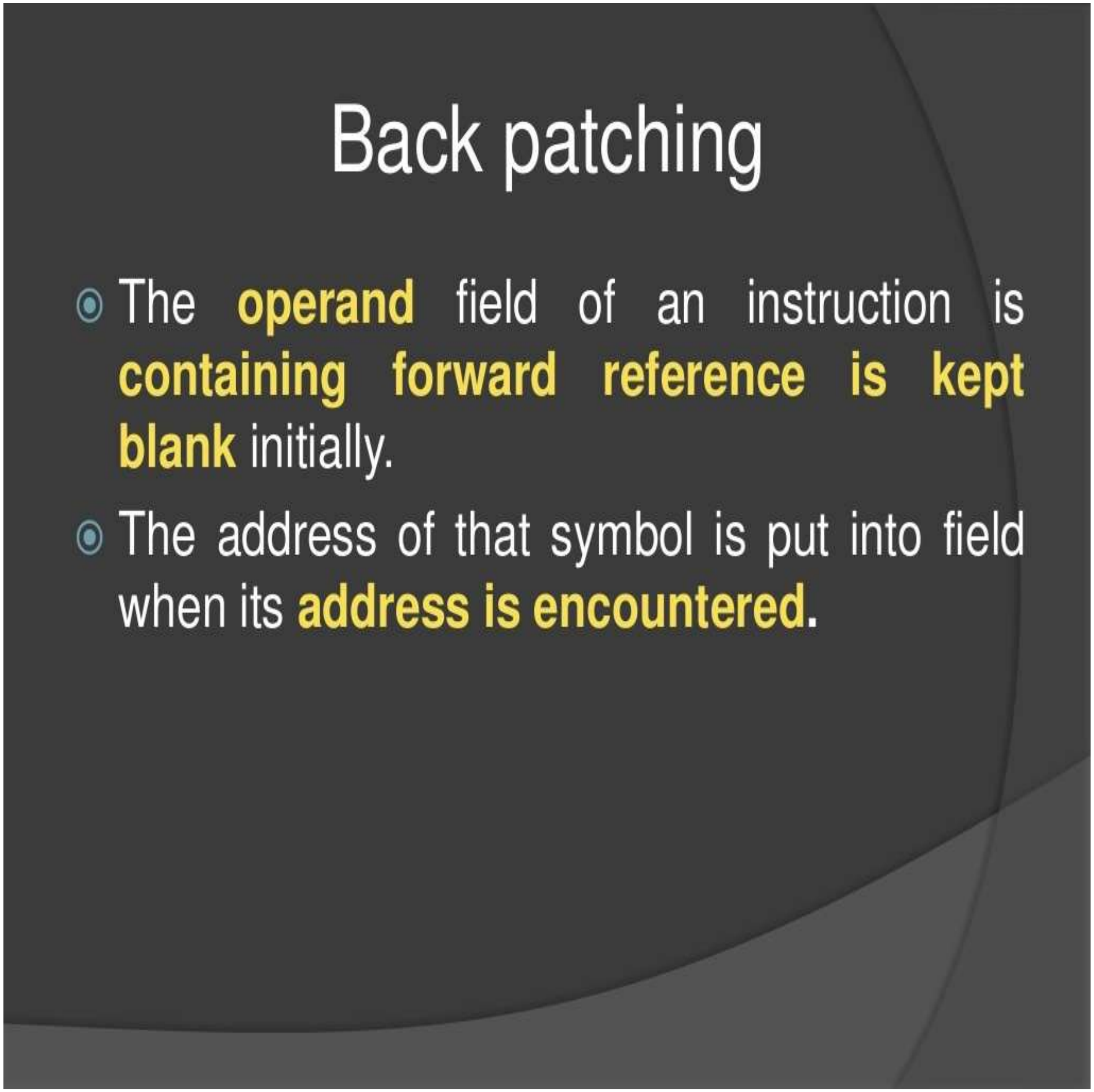
- Syntax errors(missing commas or parenthesis)
- Semantic errors (duplicate definitions of symbols)
- References to undefined variables

Sr. No	Stmt	address
001	START 200	
002	MOVER AREG,A	200
003	….	
009	MVER BREG,A	207
	**ERROR ** INVALID OPCODE	
010	ADD BREG,B	208
014	A DS 1	209
015	…….	
021	A DC ‘5’	227
	** ERROR ** DUPLICATE DEFINITION OF SYM A	
022	………	
035	END	

ERRORS IN PASS 2

Sr. No	**Stmt**	**address**
001	START 200	
002	MOVER AREG,A	200
003	….	
009	MVER BREG,A	207
	**ERROR ** INVALID OPCODE	
10	ADD BREG,B	208
	** ERROR ** UNDEFINED SYM B IN OPERAND FIELD	
014	A DS 1	209
015	…….	
021	A DC ‘5’	227
	** ERROR ** DUPLICATE DEFINITION OF SYM A	
022	………	
035	END	


Single Pass translation
Problem of Forward reference
Handled by Process called Back patching



Back patching

- The **operand** field of an instruction is **containing forward reference is kept blank** initially.
- The address of that symbol is put into field when its **address is encountered.**

Thank you