System Programming : An Introduction

Prof. Reshma Pise

Comp Engg. Dept

Vishwakarma University

Course Outcomes :

On completion of the course, the students will be able to –

1. Discriminate among different System software and their functionalities.
2. Design language translators like Macro processor and Assembler.
3. Develop approaches and methods for implementing compiler, linker and loader.
4. Use LEX tool for lexical analysis.
5. Interpret the techniques of implementing utility software.

Outline of Course

Unit 1: **Introduction to System Software**

Unit 2 : **Macro Processor**

Unit 3: Linkers and Loaders

Unit 4: Compilers

Unit 5: **Device drivers and TSR Programming**

Text Book :

"Systems Programming and Operating Systems", M. Dhamdhere, Tata McGraw-Hill, ISBN 13:978-0-07-463579-7, Second Revised Edition.

Reference Book:

"System Software An introduction to Systems Programming", Leland L. Beck, Pearson Education, ISBN13: 9788177585551

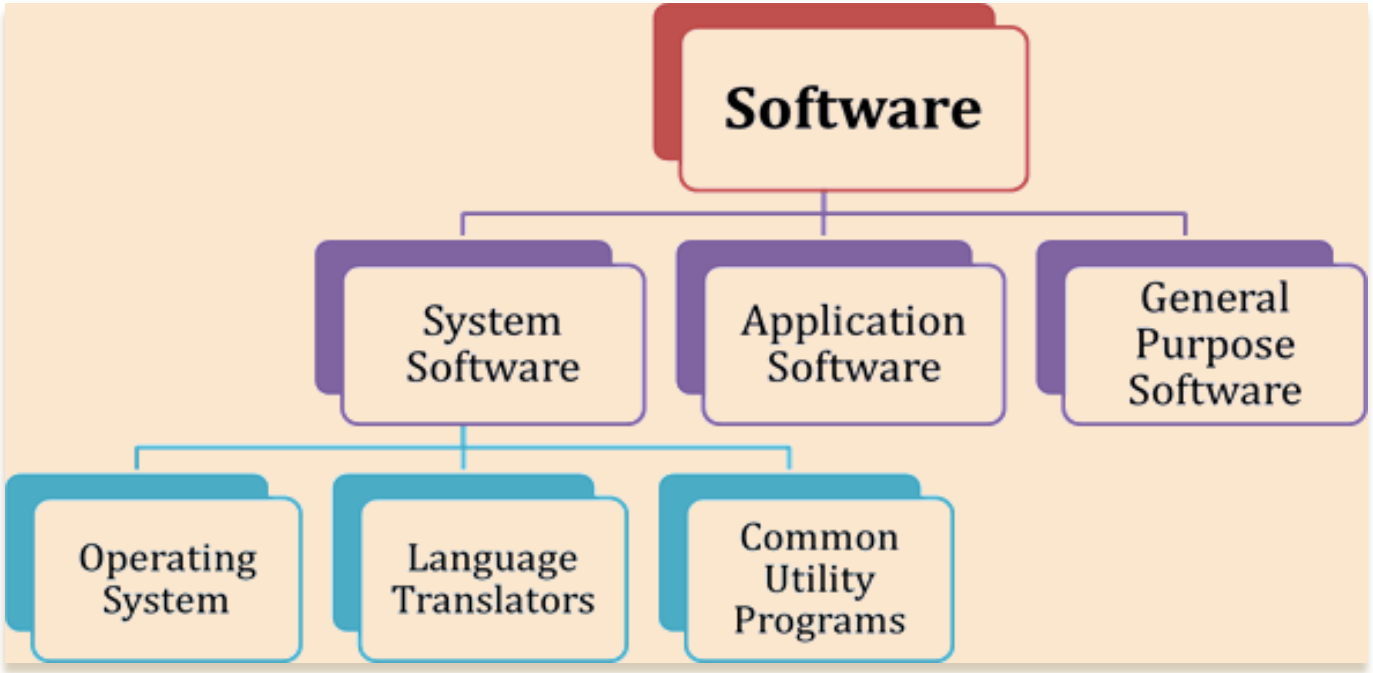
Software

It is a general term used for computer program. A set of computer Instructions or data, anything that can be stored electronically is s/w.

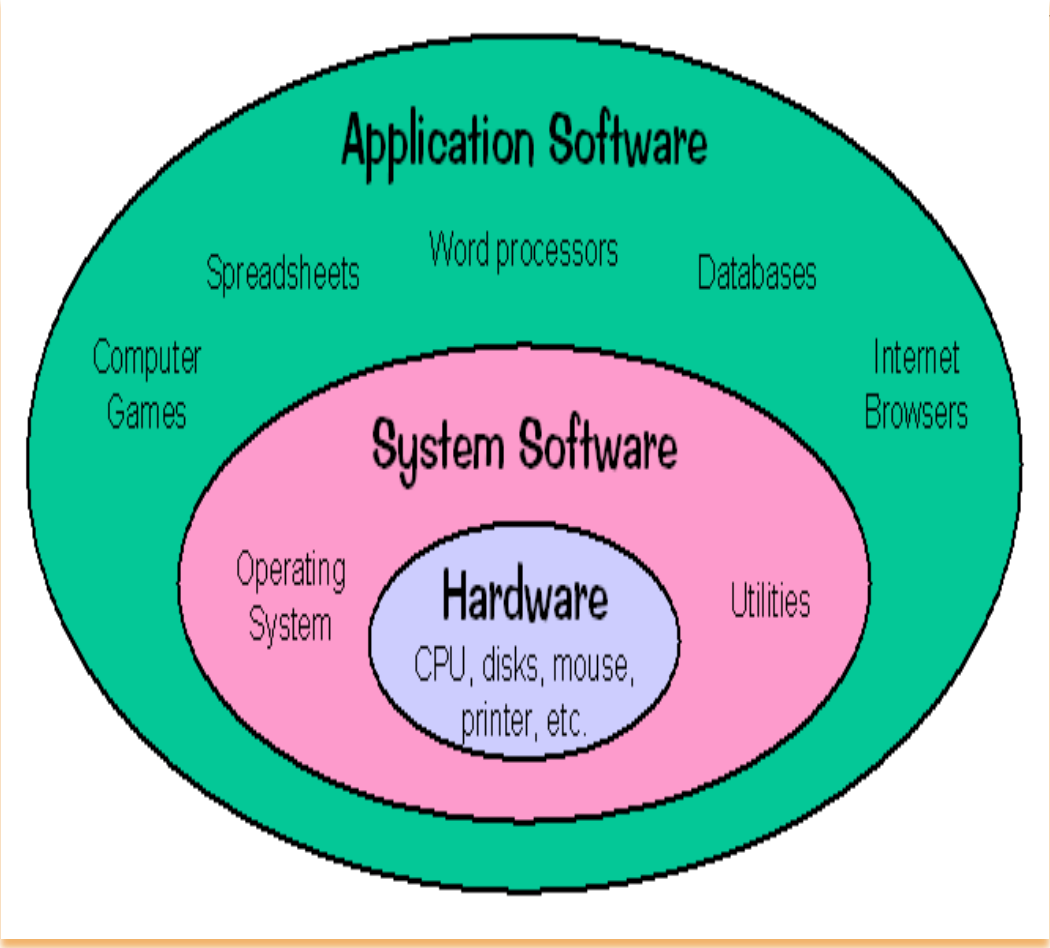
Software Examples:-

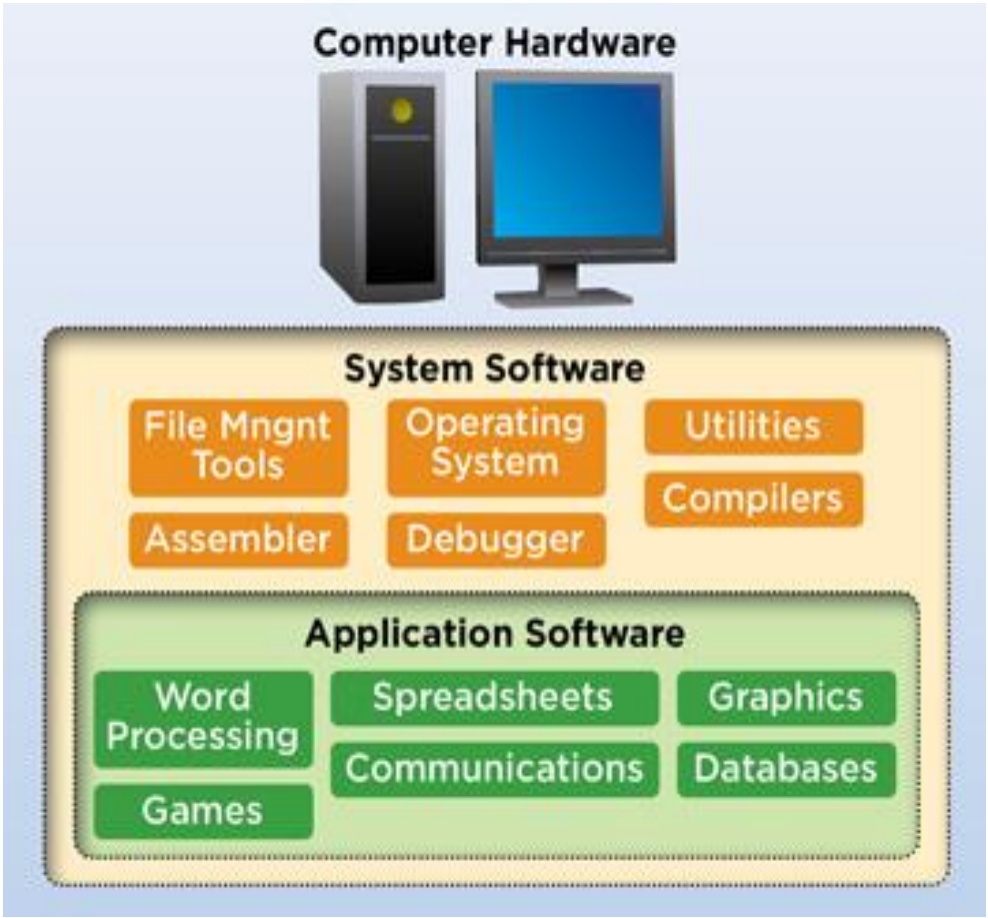
OS, Browser, Ms word, DBMS, Power point, Library management system, Compiler etc.

Types of Software



Types of Software (Layered diagram)

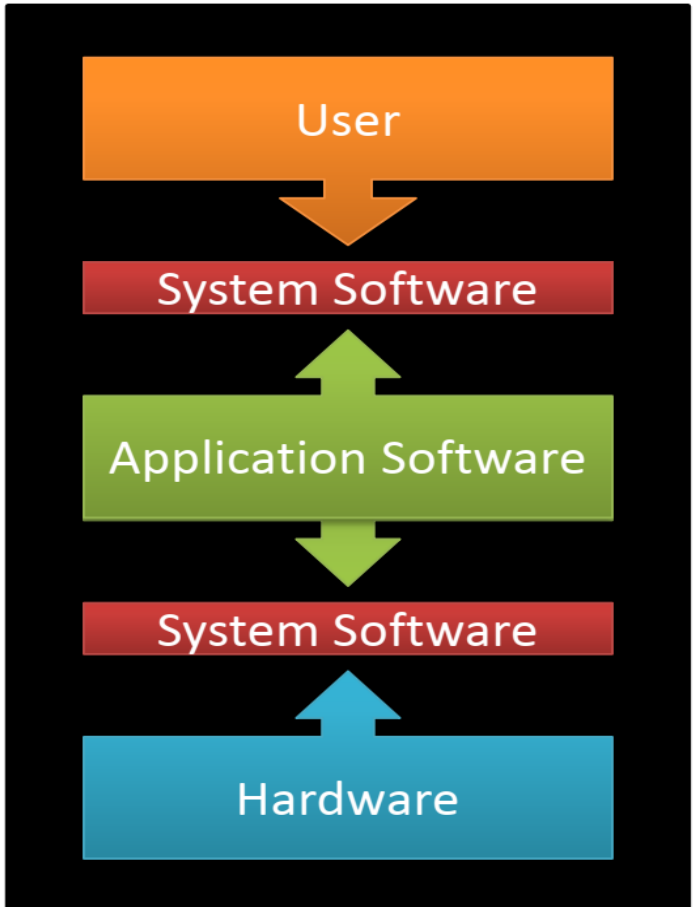




System Software

System software is designed to operate and control the computer the hardware to provide basic functionality, and provide platform for the development and execution of the application software.

- It serves as the interface between the user, the application software, and the computer's hardware.
- Makes it possible for the users to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.



Example:
Operating Systems, Compiler, Loader, Linker, Interpreter and device drivers , editors etc.

- System control programs
- System support programs
- System Development Programs

Application Software

Application Software includes programs that do real job for end user.
Designed to make users more productive and / assist them with personal tasks.

Designed to satisfy a particular need / to solve a specific problem.

Examples of application software are-

enterprise software, railway reservation software, income tax software, word processors , Spreadsheet and Database Management System, graphics software and media players etc.

Types of System Software

- Operating System
- Language Translators
- Utility Software
- Device Drivers

Types of System Software :Operating System

The operating system is the component of the system software in a computer system.

An operating system is software which manages & controls computer hardware and software resources like memory, CPU, storage and I/O devices. It also provides common services to computer programs.

Application programs are dependent on operating system to function.

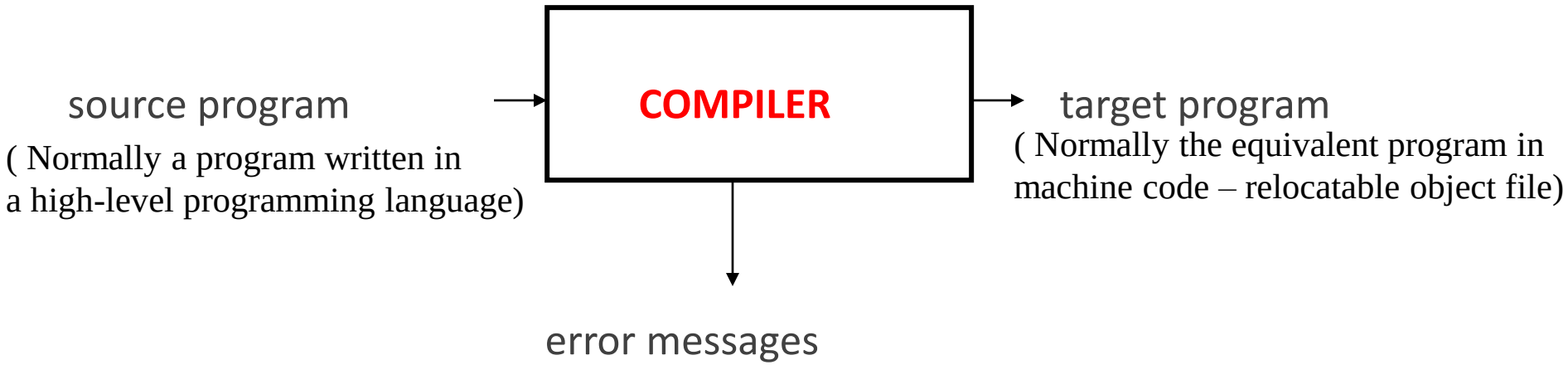
Example : Windows, Linux , Mac

Types of System Software : Language Translators

- Compiler
- Interpreter
- Assembler

Language Translators : Compilers

A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Examples : C, C++, Java

Language Translators : Interpreter

A computer program that executes instructions written in a programming language and does not produce the executable file.

Reads one instruction at a time and converts it into machine language.

Examples of interpreter based language:

PHP, JavaScript, Python

Language Translators : Assembler

A program that translates source code written in assembly language to object code.

Translate mnemonic operation codes to their machine language equivalents.

Assigns machine addresses to symbolic labels

Types of System Software :Utility software

Utility program is a type of system software that allows a user to perform maintenance-type tasks, usually related to managing / configuring a computer, its devices or its programs.

Examples:
Virus-detection and recovery utilities
File-compression utilities
Disk Scanner
Disk defragmente
Screen saver
Spam and pop-up blocker utilities
Backup Utility
Uninstall

Types of System Software : Device Drivers

Device drivers are programs that allow hardware such as sound cards, video cards, USB ports and other components , to communicate and interface with the Operating System, and other hardware attached to the computer.

Device driver is actually a communication software between device and computer.

When a new device is added the driver should be installed in order to run the program.

Quiz

1) Name the three types of System software :
2) Compiler is an example of _____ software.
3) _____ software executes instructions line by line.
4) Device drivers are example of _____ software.
5) _____ is main and important system software.
6) _____ software helps in running application software.
7) MS Power point is _____ software.
8) Utility program is a type of system software that allows a user to perform tasks such as configuration of a computer.
9) Virus-detection and recovery is an example of _____ software.
10) Python Language is an example of interpreted language
11) Assembler translate source code written in assembly language to _____.
12) _____ is a software that converts program written in one language to other language.

Quiz : Solution

1) Name the three types of System software : **Operating system , Translators and Utility/Device Drivers**
2) Compiler is an example of _____ software. **Translator**
3) _____ software executes instructions line by line. **Interpreter**
4) Device drivers are example of _____ software. **System software**
5) _____ is main and important system software. **Operating System**
6) _____ software helps in running application software. **System software**
7) MS Power point is _____ software. **Application**
8) _____ is a type of system software to perform computer configuration tasks. **Utility program**
9) Virus-detection and recovery is an example of _____ software. **Utility**
10) _____ Language is an example of interpreted language. **Python / PHP**
11) Assembler translate source code written in assembly language to _____. **Machine / Object**
12) _____ is a software that converts program written in one language to other language. **Translator**

Assignments

1) Differentiate between System software and Application Software with example.
2) Write short note on a) Device Drivers b) Utility Programs
3) List the functions of Operating system.
4) Compare Compiler and Interpreter.
5) Describe the types of translators.
6) Justify the significance of system software.
7) What do u understand by application software. Give some examples of it.
8) List any 3 : a) Device drivers b) Utility software c) Application software
9) List any 3 : a) Operating systems b) Compilers c) Interpreters

Unit 1: Assemblers Introduction

Prof. Reshma Pise

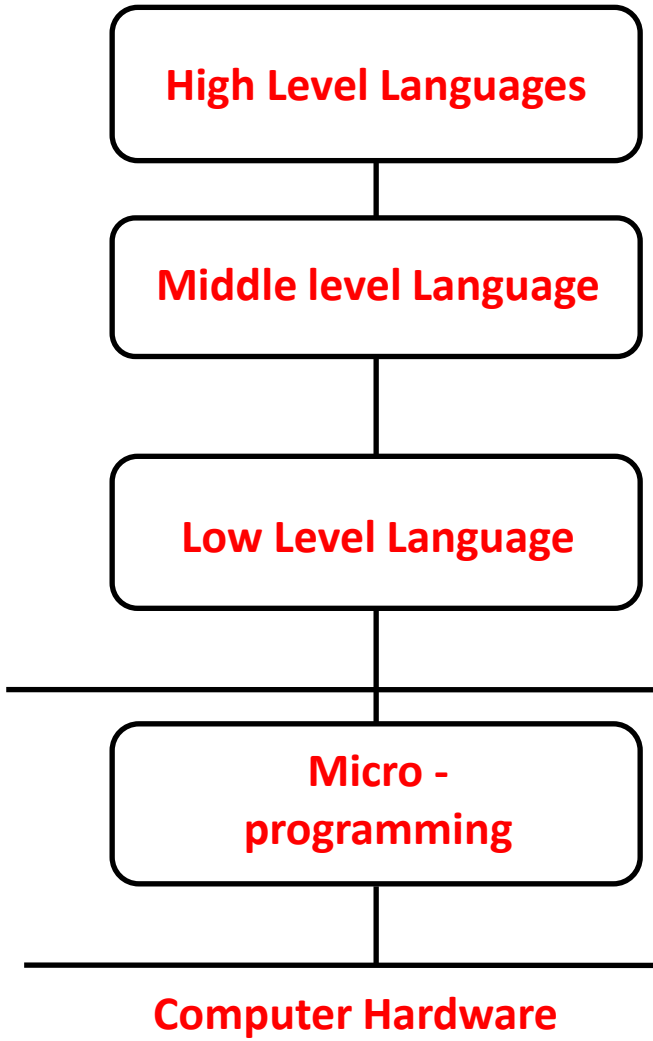
Comp Engg. Dept

Vishwakarma University

Assembler

Assembler is a translator which takes assembly language code as input and produces machine code as output.

Programming Languages



Similar to human Lang.
if , print, while()
Ex: C, C++, Java

Uses Mnemonics
MOV , ADD
Ex: Assembly Language

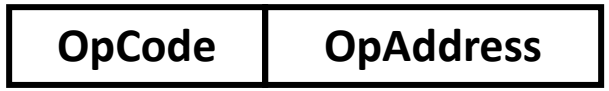
Uses 0s and 1s
Machine Language

Firmware

Discuss advantages / disadvantages

Machine code

Instruction Format:



- Operation code
  Defines executable operation (Add, Mul etc.)

- Operand address
  Specification of operands
  Operands can be constants/register addresses/ memory variable‘s addresses

Assembly Language Programming (ALP)

Assembly language is an example of middle-level language.

We use predefined words called **mnemonics**.

Binary code instructions in machine language are replaced with mnemonics and operands.

As the computer cannot understand mnemonics, we use a translator called **Assembler** to translate mnemonics into machine language.

Three features of Assembly Language:

- Mnemonic Operation Codes
- Symbolic operands
- Data declarations

Assembly Language Advantages

***Mnemonic operation codes*:**

Writing instructions in a middle-level language is easier than writing instructions in a machine language.

Assembly language is more readable compared to low-level language.

Easy to find errors in program and modify.

Not necessary to memorize numeric operation codes.

Symbolic operands:

Symbolic names can be associated with data or instructions.

Symbolic names can be used as operands in assembly instructions as data operands or address operands (need not know memory address).

Data declarations: Data can be declared in a variety of notations, including the decimal notation.

Avoids conversion of constants into their machine representation.

ALP Statement Format

<Label> :	Mnemomic	<operand spec>[,<operand spec>…]

Ex : Next : MOV AREG, Number

Label :

Symbolic labeling of an assembler address.

If a label is specified in a statement, it is associated as a symbol name with the memory address generated for the instruction.

Mnemomic : Symbolic description of an operation (e.g: mov, add)

Operands : Reg, Variable, Adress or Constant

<operand spec> has the following syntax:

<symbolic name> [+*<displacement>*] [(*<index register>*)]

e.g. AREA, AREA+5, AREA(4), AREA+5(4)

Comments : You need the comments for readability of the program and debugging.

ALP: Mnemonic Operation Codes

Each statement has two operands, first operand is a register and second operand refers to a memory word using a symbolic name and optional displacement.

Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	*First operand is modified*
02	SUB	*Condition code is set*
03	MULT	*Condition code is set*
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	*First operand is not used*
10	PRINT	*First operand is not used*

Source: ***M. Dhamdhere, Systems Programming and Operating Systems , Tata McGraw-Hill,***

ALP:Mnemonics

- *MOVE* instructions move a value between a memory word and a register
- *MOVER* – First operand is target and second operand is source
- *MOVEM* – first operand is source, second is target
- All arithmetic is performed in a register and sets *condition code.*
- A Comparision instruction sets *condition code* analogous to arithmetics, i.e. without affecting values of operands.
- *condition code* can be tested by a Branch on Condition (BC) instruction and the format is:

BC *<condition code spec> , <memory address>*

Condition code can be : LT, LE, EQ, GT, GE and ANY

Example: ALP and its equivalent Machine Language Program

```
          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)  + 00 0 001
TERM      DS      1                 116)
          END
```

Source: ***M. Dhamdhere, Systems Programming and Operating Systems , Tata McGraw-Hill,***

Types of Assembly Language Statements

There are 3 types :

- Imperative Statements
- Declarative Statements
- Assembler Directives

Imperative Statements: They specify an action to be performed during the execution of an assembled program. Each imperative statement is translated into one machine instruction.

Ex: MOV , ADD , MUL , BC

Types of Assembly Language Statements

Declarative Statements: Two types and syntax is

[Label] DC '<*value*>'

[Label] DS <*constant*>

The DC (Declare constant) statement defines memory words containing constants.

Ex: NUM DC '5'

Defines symbol NUM which is a memory word containing numeric value 5

The DS (declare storage) statement reserves memory word and associates names with them.

Ex:

N DS 1 : Reserves a memory area of 1 word, associating the name N to it

ARRAY DS 100 : Reserves a block of 100 words and the name ARRAY is associated with first word of the block. (ARRAY + <displacement> to access the other words)

Types of Assembly Language Statements

Use of Constants

The DC statement does not really implement constants

It just initializes memory words to given values.

The values are not protected by the assembler and can be changed by moving a new value into the memory word.

In the above example, the value of NUM can be changed by executing an instruction

```
MOVEM   BREG,   NUM
```

Types of Assembly Language Statements

Use of Constants

An Assembly Program can have constants like HLL, in two ways – as immediate operands, and as literals.

1.) Immediate operands can be used in an assembly statement only if the architecture of the target machine supports.

Ex: SUB BREG, 6

This is translated into an instruction with two operands – BREG and the value 6 as an immediate operand.

Types of Assembly Language Statements

Use of Constants

2) A *Literal* is an operand with the syntax = '<value>'.

- It differs from a constant because its location cannot be defined in the assembly program.
- It's value does not change during the execution of the program.
- Literals are allocated memory addresses at end of program or it can be specified by assembler directive LTORG assembler directive.

Ex: ADD AREG, ='5' ➔ ADD AREG, NUM

NUM DC '5'

Use of literals vs. Use of DC

Types of Assembly Language Statements

Assembler Directives (AD):

An assembler directive is a statement to give direction to the **assembler** to perform task of the assembly process.

It controls the organization of the program and provide necessary information to the **assembler** to generate necessary machine codes.

They ADs are not translated into machine code. Do not occupy memory.

Few examples of assembler directives are:

1) START <*constant*> : Specifies that the first word of the target program generated should be placed in the memory word having address <*constant*>.

2) END [<operand spec>]: This directive indicates the end of the of the source program. The optional <operand spec> indicates the address of the instruction where the execution of the program should begin.

Other examples : ORG, LTORG, EQU

Quiz

1) State the 3 levels of languages :
2) _____ Language doesn't require translator.
3) HLL requires _____ to execute program.
4) _____ Language makes use of mnemonics.
5) Opcode indicates _____ in instruction.
6) State the three features of ALP ?
7) The 4 fields in ALP statement are _____
8) _____ is a Symbolic description of an operation.
9) State the 3 types of ALP statements.
10) _____ is a constant whose value does not change during the execution of the program.
11) Literals are allocated memory addresses at _____ of program by default.
12) Literal addresses can be specified by _____ assembler directive.
13)The ADs are not translated into machine code. (True / False)

Quiz : Solution

1) State the 3 levels of languages : Low/ Machine level , Middle level , High Level
2) _____ Language doesn't require translator. Machine
3) HLL requires _____ to execute program. Translator
4) _____ Language makes use of mnemonics. Assembly
5) Opcode indicates _____ in instruction. Operation
6) State the three features of ALP ? Mnemonic Operation Codes, Symbolic operands , Data declarations

7) The 4 fields in ALP statement : Label, Mnemomic, Operand 1 , Operand 2
8) _____ is a Symbolic description of an operation. Mnemomic
9) State the 3 types of ALP statements. Imperative, Declarative and Assembler Directives
10) _____ is a constant whose value does not change during the execution of the program. Literal

11) Literals are allocated memory addresses at _____ of program by default. End

12) Literal addresses can be specified by _____ assembler directive. LTORG

13) The ADs are not translated into machine code. (True / False) True

Assignments

1. Discuss the advantages of assembly language.
2. What are three features of ALP
3. Explain the three types of ALP statements.
4. Differentiate between DC statement and Literal.
5. Write a short note on : Assembler Directives.
6. Study any 3 ADs not discussed in lecture.
7. Explain the concept of literals.
8. Consider : N DS 10 what does this statement do?
9. Consider 1) N DC 10 , 2) MOV AREG, N and 3) MOV AREG, 10
   Are statements 2 & 3 same ? If not why Justify.
10. What is the advantage of symbols / Labels ?

Design of Assembler : Phases

Prof. Reshma Pise

Comp Engg. Dept

Vishwakarma University

Language Processing

There are two phases in Designing of an assembler:

Language processing = Analysis of source program + Synthesis of target program

Analysis of source program is specification of the source program

Basically it involves :

- Checking if program is syntactically and semantically correct.

- Collect information required for synthesis. (e.g. Symbol Table)

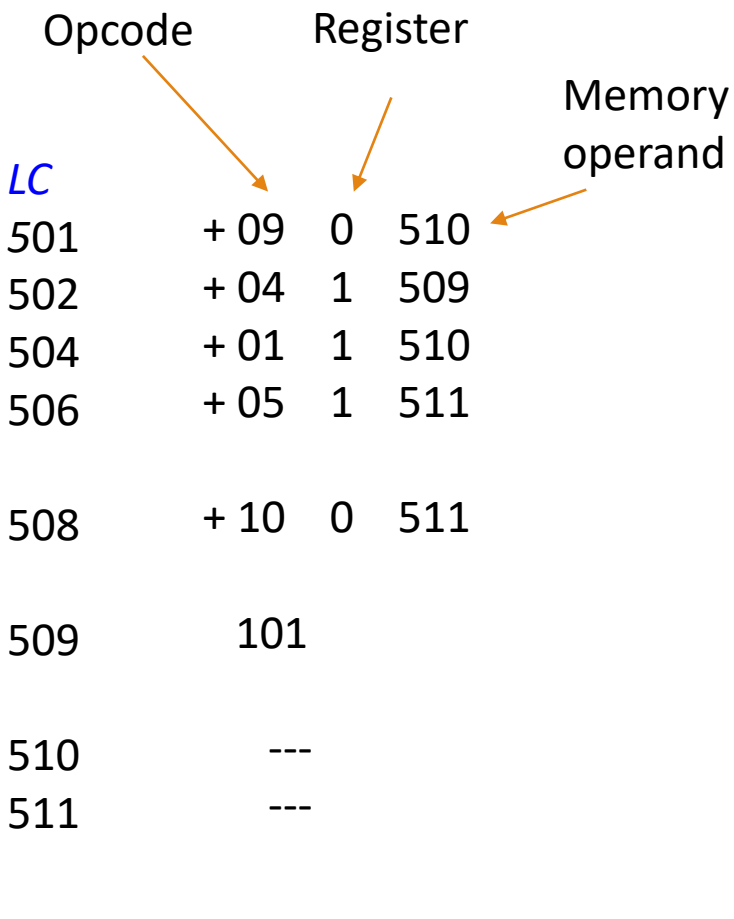
Synthesis of target program is generation of target language statements.

In assembler, it involves generation of machine code

Assembly Lang. to M/C Language

1. Find address of variables and labels.
2. Replace Symbolic address by numeric address.
3. Replace Symbolic opcodes by machine opcode.
4. Reserve storage for data.

START	501
READ	M
MOVER	AREG, N
ADD	AREG, M
MOVEM	AREG, SUM
PRINT	SUM
N	DC 5
M	DS 1
SUM	DS 1
END	



Synthesis Phase: Data Structures

Ex: MOVER BREG, NUM

The following information is needed to synthesize machine instruction for this stmt:

1. Machine operation code of instruction **MOVER** [This does not depend on the source program but depends on the assembly language, hence synthesis phase can determine this information for itself]
2. Address of the memory word with which symbol **NUM** is associated [depends on the source program and it has to be computed by the Analysis phase].
3. If there is literal operand , the address of literal is required. (e.g. MOVER BREG, ='5')

The data structures required during the synthesis phase :

- Mnemonics Table
- Symbol Table
- Literal Table

Synthesis Phase: Data Structures

1. Mnemonics Table

- The primary fields are mnemonic , opcode, and size. (ADD , 01 , 2)
- It is a fixed table for specific assembler which is merely accessed by the analysis and synthesis phases

2. Symbol Table

- Each entry has two primary fields : Symbol name and Address. (NUM , 510)
- Symbol table is constructed during analysis and used during synthesis

3. Literal Table

- Each entry has two fields : Literal and Address. e.g. (5 , 510)
- Generated by the analysis phase and used during synthesis.

Analysis Phase : Tasks Performed

The Main function of the Analysis phase is to build the Symbol table and Literal table.

- Determine the addresses with which the symbolic names used in a program are associated.

  To perform the this, we need to fix the addresses of all program instructions. This function is called *Memory Allocation*.

- To implement *memory allocation* a data structure called *Location counter (LC)* is maintained. it is initialized to the constant specified in the START statement or if nothing is mentioned in START it is initialized to 0 by default.

  To update the contents of LC, analysis phase needs to know lengths of different instructions.

Analysis phase : Scan each statement of ALP and

- Isolate the label, mnemonics opcode, and operand fields of a statement.
- If a label is present, enter the pair (symbol, <LC value>) in a new entry of symbol table.
- Check if the mnemonics opcode is valid (Refer Mnemonic table)
- Update the value of LC (LC processing)

Synthesis Phase : Tasks Performed

- Obtain Machine opcode corresponding to the mnemonic from the mnemonic table.
- Obtain address of the Symbol (can be memory operand or label) from symbol table.
- Convert the data constants to internal machine representations.
- Build / Synthesize a machine instruction in the proper format.

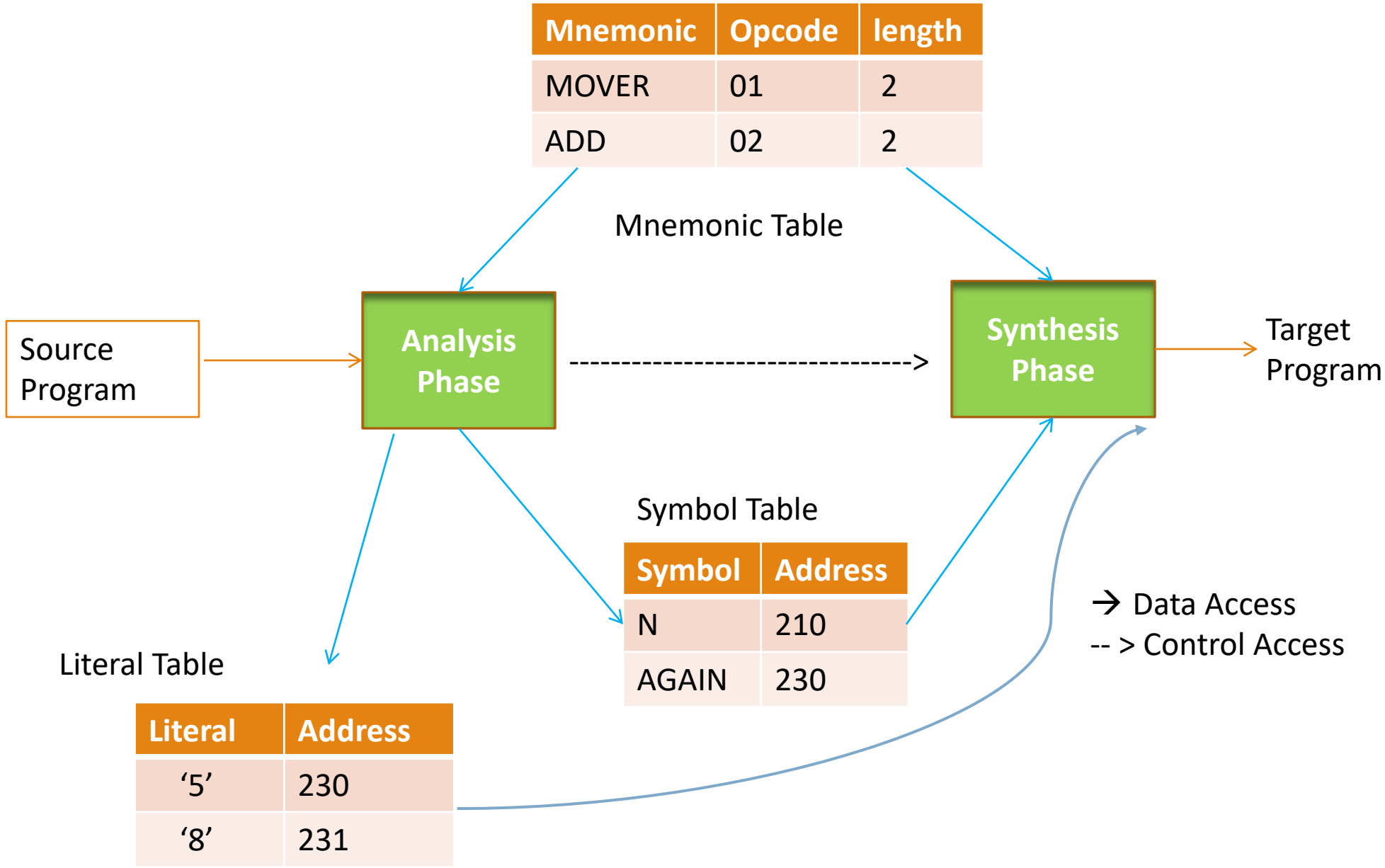
Example

Assembly Language Program	LC
START 200	200
NEXT: MOVER AREG, NUM	200
ADD BREG, AREG	202
BC ANY, NEXT	204
NUM DC 5	206
END	206

Mnemonic	Opcode	Size/length
MOVER	01	2
ADD	02	2
BC	03	2
…	…	…

Symbol	Address
NEXT	200
NUM	206

Analysis and Synthesis phase of assembler



Quiz

1) _____ phase involves checking if program is syntactically and semantically correct.
2) In _____ phase machine code is generated.
3) The data structures required during the synthesis phase :___ , ____ and _____.
4) The primary fields of MOT are _____, _____, and _____.
5) The process of updating LC is : _____
6) The two tables built in the analysis phase are :_____ and _____
7) State True / false : Literal table is updated in synthesis phase

Quiz : Solution

1) _____ phase involves checking if program is syntactically and semantically correct. Analysis
2) In _____ phase machine code is generated. Synthesis
3) The The data structures required during the synthesis phase :
   - Mnemonics Table
   - Symbol Table
   - Literal Table
4) The primary fields of MOT are : mnemonic , opcode, and size.
5) The process of updating LC is called : LC processing
6) The two tables built in the analysis phase are :_____ and _____

Symbol table & Literal table

1) True / False : Literal table is updated in synthesis phase. False

Assignments

1. Discuss functions of analysis and synthesis phase in language processing.
2. What important info. Are required for the synthesis phase.
3. Draw a neat diagram of phase structure of assembler.
4. List the operations in Analysis phase.
5. State the operations in Synthesis phase.
6. List the main tasks in converting assembly lang. program to m/c Language.

Design of Assembler : Pass structure

Prof. Reshma Pise

Comp Engg. Dept

Vishwakarma University

Assembler Design

- Pass of a language processor – One complete scan of the source program.
- Assembler Design can be done in:
  - Single pass
  - Two pass
  - Multi pass

Single Pass Assembler

Single Pass Assembler:

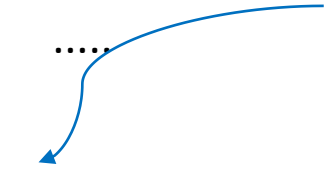
- Does analysis and synthesis tasks in a single pass.
- In single pass assembler, the problem of forward reference should be handled.

Forward reference :

When the variables are used before their definition. It is reference to a symbol that is defined later in the program.

Ex:

```
START 100
MOVER AREG, N      // To translate this instruction, address of N is required.
                   and the assembler has not yet processed the symbol N
.....
N   DC   10
```

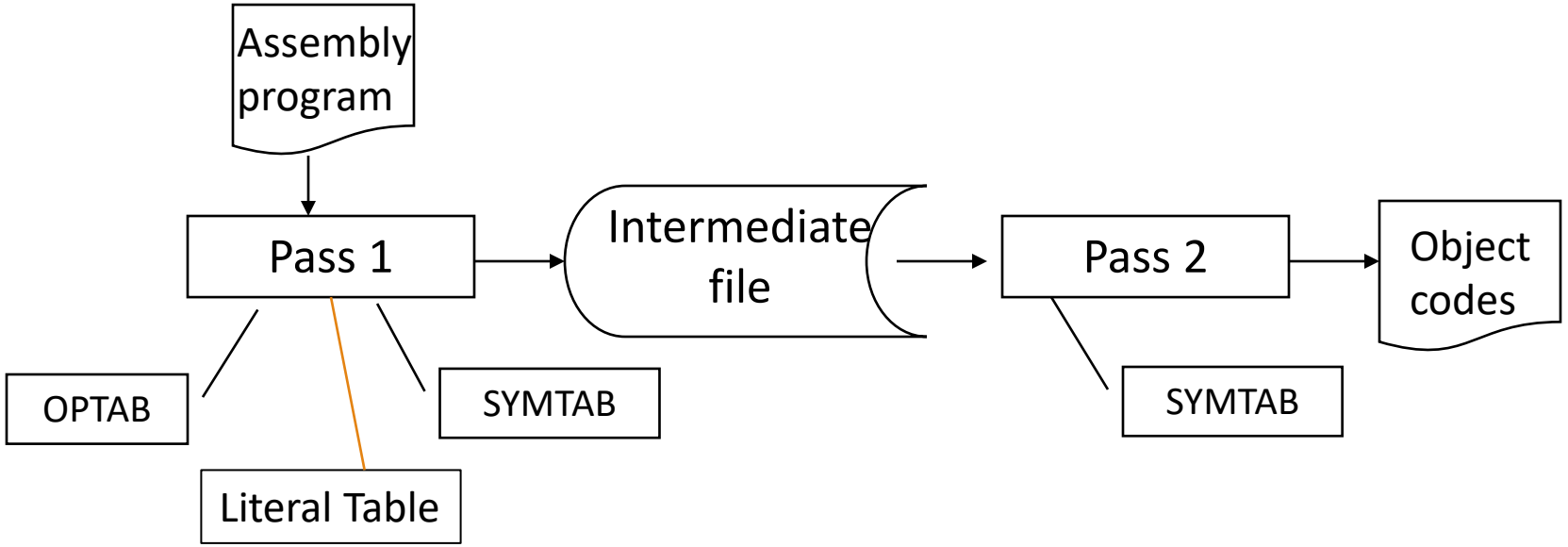


Two Pass Assembler

- Does two scans of source program.
- Translation work is divided in two passes.
- Resolves the forward references easily.
- Analysis -> Pass 1, Synthesis –> Pass

Two Pass Assembler

- The first pass separates the symbol / label , mnemonic opcode and operand fields. LC processing is performed and the symbol defined are inserted into the symbol table.
- Literals are entered in Literal table. Intermediate code generated.
- The second pass synthesizes the machine code using intermediate code & the address information found in the symbol and literal tables. It determines forward references.



Advanced Assembler Directives

- **ORG/ ORIGIN**

  Syntax : ORG <address spec>

  This directive indicates that LC should be set to the address given by <address spec>.

  Ex : ORG 400

- **EQU**

  Syntax : <symbol> EQU <address spec>

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

  Ex : MAX EQU 100 or A EQU B

- **LTORG**

  A programmer can specify where literals should be placed.

  By default, assembler places all the literals after the END statement.

Literals

The assembler allocates memory to the literals of literal pool at every LTORG statement and at the END statement, .
The pool contains all literals used in the program since the start of the program or since the last LTORG statement

```
LC        START 200
200       ADD AREG,  =  '5'
202       SUB  BREG , =  '8'
            ……

230       LTORG
232       MOVER AREG, BREG
234       MOVER  CREG, = '3'
          …..

250       END
```

The LTORG statement allocates the addresses 230 and 231 to the values '5' and '8'.
First literal pool

The END statement allocates the addresses 250 to the literal '3'.
Second literal pool.

Data Structures in Pass I

1. Location Counter : LC
2. OPTAB – a table of mnemonic op codes, static table

   Implemented as array or hash table, easy for search
   - Contains mnemonic op code, class and mnemonic info
   - Class field indicates whether the op code corresponds to
     - an imperative statement (IS) / a declaration statement (DL) or an assembler Directive (AD)
   - For IS, mnemonic info field contains the pair (machine opcode, instruction length)
   - Else, it contains the ID of the routine to handle the AD or DL statement.

Ex :	MOVER	IS	(04, 1)
	BC	IS	(07, 1)
	LTORG	AD	R#5
	DS	DL	R#1
	DC	DL	R#2

Data Structures in Pass I

3. SYMTAB - Symbol Table

- A SYMTAB entry contains primarily symbol name and address . Implemented as hash table.

Index no.	Symbol	Address
1	LOOP	502
2	AGAIN	514
3	NUM	550

4. LITTAB – a table of literals

Index	Literal	Address
1	=‘5’	511
2	=‘1’	512
3	=‘1’	520

5. POOLTAB – POOL Table

Literal no.
#1
#3

POOLTAB contains the LITTAB index of the first literal of each literal pool. When LTORG / END statement is processed, literals in the current pool are allocated addresses starting with current LC value.

Algorithm - First Pass of 2- Pass Assembler

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB [1] := 1;
   littab_ptr := 1;
2. While next statement is not an END statement
   (a) If label is present then
   this_label := symbol in label field;
   Enter (*this_label*, *loc_cntr*) in SYMTAB.
   (b) If an LTORG statement then
   (i) Process literals LITTAB [POOLTAB [*pooltab_ptr*]] . . . LITTAB [*littab_ptr*−1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.
   (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (iii) POOLTAB [*pooltab_ptr*] := *littab_ptr*;
   (c) If a START or ORIGIN statement then
   loc_cntr := value specified in operand field;
   (d) If an EQU statement then
   (i) *this_addr* := value of <*address spec*>;
   (ii) Correct the symtab entry for *this_label* to (*this_label*, *this_addr*).

Source: *M. Dhamdhere, Systems Programming and Operating Systems , Tata McGraw-Hill,*

Algorithm - First Pass of 2- Pass Assembler

(e) If a declaration statement then
  (i) *code* := code of the declaration statement;
  (ii) *size* := size of memory area required by DC/DS.
  (iii) *loc_cntr* := *loc_cntr* + *size*;
  (iv) Generate IC '(DL, *code*) ···'.
(f) If an imperative statement then
  (i) *code* := machine opcode from OPTAB;
  (ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;
  (iii) If operand is a literal then
    this_literal := literal in operand field;
    LITTAB [*littab_ptr*] := *this_literal*;
    littab_ptr := *littab_ptr* + 1;
  else (i.e. operand is a symbol)
    this_entry := SYMTAB entry number of operand;
    Generate IC '(IS, *code*)(S, *this_entry*)';

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

Source: *M. Dhamdhere, Systems Programming and Operating Systems , Tata McGraw-Hill,*

Quiz

1. The main challenge in single pass assembly is _____
2. Output of Pass 1 of 2-Pass assembler are _____, _____ and _____.
3. _____ Assembler Directive changes the value of LC.

4. For AD, DL statements mnemonic info field contains _____

5. _____ table contains index to first literal of each literal pool.

Quiz : Solution

1. The main challenge in single pass assembly is _____
2. Output of Pass 1 of 2-Pass assembler are _____, ____ and ____.
3. _____ Assembler Directive changes value of LC.
4. For AD, DL statements mnemonic info field contains _____
5. _____ table contains index to first literal of each literal pool.

Answers:

1. Forward References
2. Symbol Table, Literal Table and IC
3. ORG
4. ID of routine to handle AD and DL
5. Pool Table

Assignments

1. List the operations performed in Pass 1 and Pass 2 of a 2-pass assembler.

2. What is advantage of generating IC in 2-pass assembler ?

3. Explain the forward reference by an example. Why is it a challenge in single pass assembly?

4. Elaborate significance of LTORG Assembler directive.

5. Devise an algorithm for Pass 1 of 2-pass assembler.

Assignments

Consider the following Assembly language code.

```
                START
                MOV     A , = ' 6 '
LOOP :          READ    N
                MOV     B , N
                SUB     A , B
                JZ      LOOP
                ORG 500
                LTORG
                ADD     B , = ' 30 '
                LTORG
                INC     B
N               DC      10
STR             DS      ( 20 )
NUM             EQU     N
                END
```

Assume each imperative instruction is of length 2.

Generate Symbol table.

Generate Literal table and Pool Table

Perform LC processing for the program given.

Intermediate Code Forms: 2 Pass Assembler

Prof. Reshma Pise

Comp Engg. Dept

Vishwakarma University

Intermediate Code

Output of Pass 1 of two – pass assembler is Intermediate code
Intermediate code consist of a set of IC units, each unit consists of 3 fields :
1. Address (LC)
2. Representation of mnemonics opcode.
3. Representation of operands

There are 2 possible forms of IC :
1) Variant I
2) Variant II

They differ in representation of operands fields.
In Variant I operand fields are completely processed by Pass 1.

Intermediate Code

The mnemonics field contains a pair of the form (statement class, code).
Statement class can be one of IS, DL, and AD for imperative statement, declaration statement and assembler directive respectively.

For imperative statement, code is the instruction opcode in the machine language from MOT.

e.g. MOV = > (IS , 04) // 4 is opcode of MOV

For declarations and assembler directives, code is the index number within the class. Thus, (AD, 05) stands for assembler directive number 5 which LTORG.

Intermediate Code: Variant I

Representation of First operand :

A single digit number which is a code for a register or the condition code.

(Ex: 1 – AREG or 1 – LT etc.)

Representation of Second operand :

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

Ex: The operand descriptor for the statement START 200 is

(C , 200).

For a symbol or literal, the code field contains the index of symbol / literal in SYMTAB or LITTAB

(S , 1) or (L , 1)

```
        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)
        ...                     ...
```

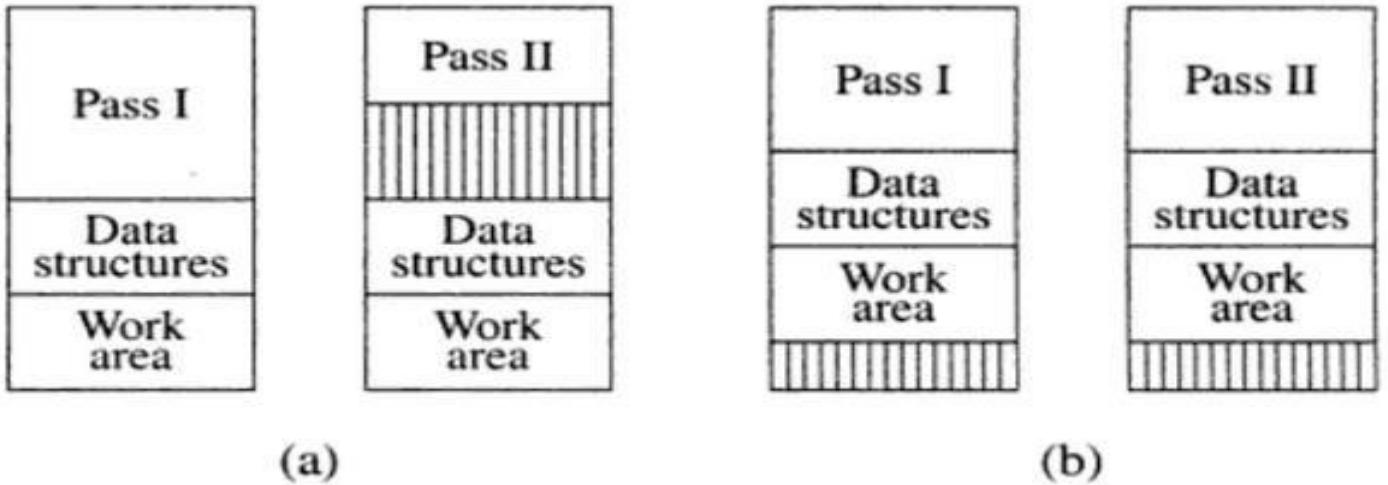
Source: *M. Dhamdhere, Systems Programming and Operating Systems , Tata McGraw-Hill,*

Intermediate Code: Variant II

- In variant II , symbols, condition codes and CPU register operands are not processed.
- Declarative statement and assembler directives : operand field is processed to support LC processing.
- Imperative statements the operand field is processed only to identify literal references.

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

Intermediate Code Forms



Source: *M. Dhamdhere, Systems Programming and Operating Systems , Tata McGraw-Hill,*

Memory requirement with (a) Variant I and (b) Variant II

In Variant I operand fields are completely processed by Pass 1. Pass 1 is overloaded and requires more space.

Intermediate Code Forms

Comparison of the variants

Variant I	Variant II
✓ Extra work in pass I	✓ Extra work in pass II
✓ Simplifies tasks in pass II	✓ Simplifies tasks in pass I
✓ Occupies more memory then pass II	✓ Memory utilization of two passes get better balanced.

Quiz

1. Each IC unit consists of _____, _____ and _____
2. The two forms of IC are _____ and _____.
3. In _____ form of IC operand fields are processed completely.

4. In _____ form of IC , the memory utilization is balanced in both pass 1 and Pass 2.
5. Pass 1 is overloaded in _____ form of IC.
6. State True / False

   In Variant 2 , operand field of declarative statement and assembler directives is processed

Quiz : Solution

1. Each IC unit consists of LC, Representation of mnemonics opcode and Representation of operands.
2. The two forms of IC are Variant I and Variant II.
3. In Variant I form of IC , operand fields are processed completely.
4. In Variant II form of IC , the memory utilization is balanced in both pass 1 and Pass 2.
5. Pass 1 is overloaded in Variant I form of IC.

6. State True / False

   In Variant 2 , operand field of declarative statement and assembler directives is processed : True

Assignments

1. Explain with an example Variant 1 and 2 forms of IC.

2. Compare Variant I and Variant II forms of IC.

3. Why is it necessary to process operand fields of AD and DL statements in Variant 2 ?

Design of Assembler : Pass 2 of 2–Pass Assembler

Prof. Reshma Pise

Comp Engg. Dept

Vishwakarma University

Pass 2 of 2–Pass Assembler

Functions :

- Process IC to synthesize the machine code.
- LC Processing
- Error Reporting

Input : Intermediate Code generated in Pass 1.

Output : Machine code (Object file)

Data Structures Used : LC, Symbol Table, Literal Table and Pool Table.

Pass 2 : Algorithm

The target code is to be assembled in the area named code_area.

1. code_area_address = address of code_area;
   Pooltab_ptr = 1;
   Loc_cntr = 0;
2. While next statement is not an END statement
   a) Clear machine_code_buffer
   b) If an LTORG statement
      i) Process literals in LITTAB from current pool and assemble the literals in machine_code_buffer.
      ii) size= size of memory area required for literals
      iii) Pooltab_ptr = pooltab_ptr + 1;
   c) If a START or ORIGIN statement
      i) Loc_cntr = value specified in operand field;
      ii) size = 0;
   d) If a declaration statement
      i) If a DC statement then assemble the constant in machine_code_buffer;
      ii) size = size of memory area required by DC / DS;

Pass 2 : Algorithm

e) If an Imperative statement

i) Get operand address from SYMTAB or LITTAB

ii) Assemble instruction in machine_code_buffer;

iii) size = size of instruction;

f) If size≠ 0 then

i) Move contents of machine_code_buffer to the address code_area_address + Loc_cntr;

ii) Loc_cntr = Loc_cntr + size;

3. Processing end statement

i) Perform steps 2(b) and 2(f)

ii) Write code_area into output file.

Error Reporting

The symbol B referenced in statement 10 is undefined. It is not possible to such error during pass I.
This kind of error can be detected only after completing pass I.
During pass II , SYMTAB is available. Symbol table is searched for the entry B. If a match is not found, error is reported.

Sr. No.	Statement	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 A	
022	.....	
035	END	
	Error ** Undefined Symbol B at stmt 10	

Quiz

1. Output of Pass 2 is _____

2. The appropriate pass for error reporting is _____

3. The instructions are assembled in _____.

4. State True / False
   - LC processing is not required in Pass 2
   - AD are present in Machine code.

Quiz : Solution

1. Output of Pass 2 is Machine code (Object file)

2. The appropriate pass for error reporting is : Pass 2

3. The instructions are assembled in Machine code buffer

4. State True / False
   - LC processing is not required in Pass 2 : False
   - AD are present in Machine code. : False

Assignments

1. Explain the processing of Literals in Pass 2.

2. Explain the processing of DL and AD statements in Pass 2.

3. State the data structures used in Pass 2.

4. Write an algorithm for Pass 2 of 2-Pass Assembler.

5. Write a short note on Error reporting in assembler.

Design of Assembler : One Pass Assembler

Prof. Reshma Pise

Comp Engg. Dept

Vishwakarma University

Single Pass Assembler

One-pass assemblers

- A one pass assembler requires 1 scan of the source program to generate machine code
- No intermediate code generated
- LC processing and construction of the symbol table is done as in two pass assembler.

Main problem - Forward references

Data items

Labels on instructions

Forward reference : When the variables are used before their definition. It is reference to a symbol that is defined later in the program.

Ex:

```
START 100
MOVER AREG, N      // To translate this instruction, address of N is required.
                   // and the assembler has not yet processed the symbol N
.....
N   DC   10
```

Single Pass Assembler:Backpatching

The problem of forward references is handled using a process called Backpatching

The operand field of an instruction containing a forward reference is left blank

Ex: MOVER AREG, N

Is partially synthesized since N is a forward reference. Second operand field in the machine code is left blank. It will be inserted later.

- An entry is added in Table of Incomplete Instructions (TII) , A table of instructions containing forward references.
- Each entry in the TII is a pair of the form (<instruction address>, <symbol referenced>)
- When the END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program.

Single Pass Assembler:Backpatching

- When the END statement is processed , TII would contain information of all forward references.

- Now each entry in TII is processed to complete the machine code.

  Ex: the entry (501, N) would be processed by finding the address of N from the symbol table and inserting it in the operand field of the instruction with assembled address 501.

- At the end of the program, report the error if symbol is not found in symbol table. (It means the symbol is undefined)

Quiz

1. _____ is a reference to a symbol that is defined later in the program.

2. The problem of forward references is handled using a process called _____.

3. _____ is a table of instructions containing forward references.
4. Each entry in TII is processed when ____ statement is processed.

Quiz : Solution

1. Forward reference is a reference to a symbol that is defined later in the program.
2. The problem of forward references is handled using a process called Backpatching

3. Table of Incomplete Instructions (TII) , a table of instructions containing forward references.

4. Each entry in TII is processed when the END statement is processed.

Assignments

1. Explain the design of single pass assembler.
2. What is Back Patching?
3. What is the role of TII in single pass assembler?
4. How the entries in TII are processed to generate M/C code, explain with an example?

Mind Map of Assembler