Systems Programming

Jagadish K Kamble (M.Tech, IIT Kharagpur)

Dept. of Information Technology, Pune Institute of Computer Technology, Pune
Course Materials

• Slides will be made available at PICT Website.

• Books

2. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Addison Wesley,
3. J. J. Donovan, Systems Programming, McGraw-Hill,

Many other books are available and may serve the same purpose
About the Course

- Venue (Theory): C-405
- Class Timings:
  M(12.15-1.15), Tu(10.00-11.00), We(11.15-12.15), TH(9.00-10:00)
- Lab (Each Two hours per week) will be conducted
- Venue (lab): C-404, C-401
- Evaluation in the theory course (100 Marks):
  In-Semester-30%
  End-semester -70%
About the Course

• Evaluation in the Lab (100 Marks): (SL-V:SP + DAA)

• Internal(I)-50 marks (SL-V:SP+DAA)

• Attendance and Continuous Assessment, 20-40% weight

• Programming exercises(Assignments and Tutorials) 80-60% weight
  - Lab exercises, rules, instruction on how / when / whom to submit will be posted on website

• External (E)(Viva-Voce) -50 marks (SL-V:SP+DAA)

• Note these weights are indicative, and may change as semester progresses
• **Attendance REALLY matters**
• Important for understanding the course
• Any student with low attendance may be deregistered from the course
Important Dates (Tentative)

• Class Test 1: 3rd Week of Jan-2018
• In-Semester: 2nd Week of Feb-2018
• Class Test 2: 3rd Week March-2018
• End-semester: May-2018

(The exact Time will be announced later)
Modern Computer System

- CPU, Memory, I/O devices, Networking Support etc.
- Machine language (1’s and 0’s)
- User Unable to instruct using M/C language
- System Software
- Bridge gap between User and Computer System
- Like Software layer
The scope of the system program

User

Application Program | Utility Program (Library)

Debugging Aids | Macro Processor | Text Editor

Compiler | Assembler | Loader and Linker

Memory Management | Processor and Process Management | Device Management | Information Management

Bare Machine (Computer H/W)
What is System Software

- What is Computer System?
- School or College student
- User of Application package like administrative stuff
- Programmer
- Abstracts view
- Computer System is combination of
  - Computer H/W
  - Computer S/W
Abstract View

Figure 1.1 Abstract views of a computer system: (a) View of a student, (b) View of the programmer
- System Program
- Environments
- Editing, Compiling and arranging for execution
- Linking, Protecting

Used to describe the collection of techniques used in design of system program.
Computer Languages

• **Machine languages**

• **Low-level languages**
  • Assembly language: symbolic representation of machine instructions.
  • Assembler: a compiler which translates from an assembly language to a machine language.

• **High-level languages**
  • Java, Python, C, ASP, HTML, ...
High level language

- Preprocessor
- Compiler
- Assembler
- Loader/Linker

Pure High level language
Assembly Code
Machine Code (relocatable)
Executable File
Course Contents

- Unit 1: INTRODUCTION TO SYSTEMS PROGRAMMING AND ASSEMBLERS
- Unit 2: MACROPROCESSORS, LOADERS AND LINKERS
- Unit 3: INTRODUCTION TO COMPILERS
- Unit 4: PARSERS
- Unit 5: SEMANTIC ANALYSIS AND STORAGE ALLOCATION
- Unit 6: CODE GENERATION AND OPTIMIZATION
Recall

- Course Structure and Contents
- Introduction
Unit 1: INTRODUCTION TO SYSTEMS PROGRAMMING AND ASSEMBLERS
Outline

■ Introduction:
  ❑ Need of System Software
  ❑ Components of System Software
  ❑ Language Processing Activities
  ❑ Fundamentals of Language Processing

■ Assemblers:
  ❑ Elements of Assembly Language Programming
  ❑ A simple Assembly Scheme
  ❑ Pass structure of Assemblers
  ❑ Design of Two Pass Assembler
  ❑ Single pass assembler
Computer software, or simply software, refers to the non-tangible components of computers, known as computer programs. The term is used to contrast with computer hardware, which denotes the physical tangible components of computers.
Software classification

- Software can be classified into
  - System software:
    - **System software** (or **systems software**) is *computer software* designed to operate and control the *computer hardware* and to provide a platform for running *application software*.

  - System software is collection of software program that perform a variety of functions like IO management, storage management, generation and execution of programs etc.
    - Operating Systems
    - Compiler / Assembler (utility software's)
    - Device Drivers

  - Application software:
    - Application software is kind of software which is designed for fulfillment specialized user requirement.
      - MS Office
      - Adobe Photoshop
      - Audio/Media Player
The system software work as middleware between application software and hardware.
Abstract View of System Components

- user 1
- user 2
- user 3
- ... user n

- compiler
- assembler
- text editor
- ... database system

system and application programs

- operating system
- computer hardware
Introduction

- Collection of system programs
- facilitate execution of programs and use of resources in computer system
- System Software consists of a variety of programs that support the operation of a computer.
- The software 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.
System Software and Machine Architecture

- One characteristic in which most system software differs from application software is **machine dependency**.

- **System programs are intended to support the operation and use of the computer itself, rather than any particular application.**

- **e.g. of system software**
  - Text editor, assembler, compiler, loader or linker, debugger, macro processors, operating system, database management systems, software engineering tools, ...
1.2 Goal of System Software

- User Convenience
  - Convenient Methods of Use

- Efficient use
  - Efficient use of Computer Resources

- Non-interference
  - Prevent Interference
System Softwares/Language Processors

- **Text editor**
  - To create and modify the program

- **Compiler and assembler**
  - You translated these programs into machine language

- **Loader or linker**
  - The resulting machine program was loaded into memory and prepared for execution

- **Debugger**
  - To help detect errors in the program
Language processors (Why?)

- Language processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behavior of software and the manner in which these ideas are implemented in computer system.

- The designer expresses the ideas in terms related to the application domain of the software.

- To implement these ideas, their description has to be interpreted in terms related to the execution domain.
The term semantics to represent the rules of meaning of a domain, and the term semantic gap to represent difference between the semantics of two domains.
Design of Airline Application System
  ◦ Query
  ◦ Book
  ◦ Cancel

The Description of reservation data and operations form specification of application.

System S/W help to implement this specification on the computer using machine language.

System S/W has to implement his specification in the execution domain of computer system.

Entities in Execution domains are cells in memory and registers in CPU and operations are the instruction of the CPU.
Semantic gap is bridged when specification in one of the domains is converted into specification in the other domain.

these issues are tackled by software engineering thru’ use of methodologies and programming languages.

Programming Language

Language Processor
s/w development team

Programming language processor
Application domain

- Reservation Data

Execution domain

- Data Structures

- Functions

- CPU Instructions

CPU Registers
Memory
IO Devices

Specification gap

Execution gap

- Query
- Book
- Cancel

Programming Language domain

- Execution domain
Language processor: A language processor is software which bridge a specification or execution gap.

System Program that bridges the gap between how a user describes a computation—call it specification of computation and how computer executes a program.

- A Language Translator
- De-translator
- Preprocessor
- Language migrator
Example

C++ Program converted to C Program by C++ Preprocessor
C++ Program converted to machine language by C++ translator.
Language Processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behavior of a software and the manner in which these ideas are implemented in a computer system.

The designer expresses the ideas in terms related to the application domain of the software. To implement these ideas, their description has to be interpreted in terms related to the execution domain.
System Programming

User

System Software

Computer
Semantic Gap

Application Domain

Semantic Gap

Execution Domain

Idea

Implement
Semantic Gap has many consequences:
- Large development time
- Large development effort
- Poor quality of software
The software engineering steps aimed at the use of a PL can be grouped into:
- Specification, design and coding steps
- PL implementation steps
Specification and Execution Gaps

- **Specification Gap**
  - It is the semantic gap between two specifications of the same task.

- **Execution Gap**
  - It is the gap between the semantics of programs (that perform the same task) written in different programming languages.
Language Processors

- “A language processor is a software which bridges a specification or execution gap”.

- The program form input to a language processor as the source program and to its output as the target program.

- The languages in which these programs are written are called source language and target language, respectively.
Types of Language Processors

- A **language translator** bridges an execution gap to the machine language (or assembly language) of a computer system. E.g. Assembler, Compiler.
- A **detranslator** bridges the same execution gap as the language translator, but in the reverse direction.
- A **preprocessor** is a language processor which bridges an execution gap but is not a language translator.
- A **language migrator** bridges the specification gap between two PLs.
Language Processors - Examples

C++ Program → C++ preprocessor → C Program

C++ Program → C++ translator → Machine Language Program

Errors
Interpreters

- An **interpreter** is a language processor which bridges an execution gap without generating a machine language program.
- An interpreter is a **language translator** according to classification.

![Diagram showing Interpreter Domain, Application Domain, PL Domain, Execution Domain]
Language Processing Activities

- Program Generation Activities
- Program Execution Activities
The program generator is a system program which accepts the specification of a program in some specification language and generates a program in target language that fulfills the specification.

- Program Generator domain
Program Generation

Specification Gap

Application Domain
Program Generator Domain
Target PL Domain
Execution Domain
**Example**

Data Entry Environment: Screen Handling Program

### Specification:

- **Employee Name**: char :start [line=2,position=25] end [line=2,position=80]
- **Married**: char :start [line=10,position=25] end [line=10,position=27]
- **Age**: numeric :start [line=12,position=25] end [line=12,position=26]

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
<th>Description</th>
<th>Data Entry Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Name</td>
<td>char</td>
<td>Name of the employee</td>
<td>Screen Handling Program</td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td>Address of the employee</td>
<td>Screen Handling Program</td>
</tr>
<tr>
<td>Married</td>
<td>char</td>
<td>Marital status</td>
<td>Screen Handling Program</td>
</tr>
<tr>
<td>Age</td>
<td>numeric</td>
<td>Age of the employee</td>
<td>Screen Handling Program</td>
</tr>
<tr>
<td>Sex</td>
<td></td>
<td>Gender</td>
<td>Screen Handling Program</td>
</tr>
</tbody>
</table>
Two popular models for program execution are translation and interpretation.

- **Program translation**
  - A program must be translated before it can be executed.
  - The translated program may be saved in a file. The saved program may be executed repeatedly.
  - A program must be retranslated following modifications.
Practical Arrangement of Language Processor

Source program → Preprocessor → Pure HLL → Compiler → Assembly code → Assembler → M/C lang. Code → Linker

Data → M/C Language Program → Results → Loader
Program Execution

- Program interpretation

Interpretation

Program execution
Fundamentals of Language Processing

Language Processing = Analysis of SP + Synthesis of TP

Collection of LP components engaged in analysis a source program as the analysis phase and components engaged in synthesizing a target program constitute the synthesis phase.
Analysis Phase

- The specification consists of three components:
  - **Lexical rules** which govern the formation of valid lexical units in the source language.
  - **Syntax rules** which govern the formation of valid statements in the source language.
  - **Semantic rules** which associate meaning with valid statements of the language.

- Consider the following example:
  
  \[
  \text{percent\_profit} = (\text{profit} \times 100) / \text{cost\_price};
  \]

  **Lexical units** identifies =, * and / operators, 100 as constant, and the remaining strings as identifiers.

  **Syntax analysis** identifies the statement as an assignment statement with percent\_profit as the left hand side and (profit * 100) / cost\_price as the expression on the right hand side.

  **Semantic analysis** determines the meaning of the statement to be the assignment of profit \times 100 / cost\_price to percent\_profit.
The synthesis phase is concerned with the construction of target language statements which have the same meaning as a source statement.

It performs two main activities:
- Creation of data structures in the target program (memory allocation)
- Generation of target code (code generation)

Example

```
MOVER AREG, PROFIT
MULT AREG, 100
DIV AREG, COST_PRICE
MOVEM AREG, PERCENT_PROFIT
...
PERCENT_PROFIT DW 1
PROFIT DW 1
COST_PRICE DW 1
```
Phases and Passes of LP

- Analysis of source statements cannot be immediately followed by synthesis of equivalent target statements due to the following reasons:
  - Forward References
  - Issues concerning memory requirements and organization of a LP
Forward Reference

- A forward Reference of a program entity is a reference to the entity in some statements of the program that occurs before the statement containing the definition or declaration of the entity.

- \( \text{Percent_profit} = (\text{profit} \times 100)/\text{cost_price} \);

- ..... 

- float profit;

- Tackle the problem using Multi-pass organization
Language Processor Pass

• A language Processor pass is the processing of every statement in a source program or in its equivalent representation to perform a language processing function.
• “Pass” is Abstract Noun denote processing done by language processor.
• Pass I:-perform Analysis of the source program and note the deduced information.
• Pass II:-Perform Synthesis of target program.
Recall

- **Language Processor**
- **Language Processing Activities**
  - Program generation Activity
  - Program Execution Activity
- **Fundamental of Language Processing**
  - Analysis Phase
  - Synthesis Phase
- **Lexical rules, Syntax rules and Semantic rules**
- **Memory allocation and Code Generation**

![Diagram showing the relationship between Application Domain, Program Generator Domain, PL Domain, and Execution Domain with gaps identified as Specification Gap, Semantic Gap, and Execution Gap.](image-url)
If Analysis Phase immediately followed by Synthesis Phase
  - Forward reference
  - High Amount Memory Requirements

Multi-pass organization (language processor Pass)
- Pass I- Performs Analysis of Source Program and note relevant information.
- Pass II-Performs synthesis of target program.

Some Language Processor processors perform certain operations more than once,

Example
  - Pass 1 : Analysis of SP to find type of each variable.
  - Pass 2 : Analysis of SP to generate target code(Synthesis phase) by using type information noted in pass I.

This Duplication and Overhead can be avoided using

Intermediate Representation
Intermediate Representation (IR)

- Representation of a source program which reflects the effects of some, but not all, analysis and synthesis functions performed during language processing.
- Ease of use
- Processing Efficiency
- Memory Efficiency
Pass I perform Analysis of the source program and reflects its result in Intermediate Representation.
Pass II reads and analyzes the IR to perform synthesis of target program.
Pass I concerned with source language issues hence called front end.
Pass II concerned with Synthesis of target program so called back end of language processor.
Reduce the memory requirements as follows.
Front end firstly loaded into memory, it analyzes the source program, generates intermediate representation and writes it to disk.
Now code front end is removed from memory and code of back end is loaded.
It reads the IR and synthesis the target program.
The diagram illustrates the process of converting a source program into a target program through two passes. The source program flows through a Front End, then an Intermediate Representation, and finally the Back End to produce a target program.
Front end

• Performs Lexical, syntax and semantic analysis of source program.

• Each Kind of Analysis Involves following functions:
  ✓ Check validity of source statement from the viewpoint of analysis.
  ✓ Determine ‘content’ of source statement.
  ✓ Construct the suitable representation of source statement.

• Use generic form for Representation of content of various stages of language processor. It consists two important Components
  • 1) Tables (The most important Symbol Table which contain information about identifiers used in source program)
  • 2) Intermediate Code
Front End of Toy Compiler

Source Program

Lexical analysis (Scanning)

Syntax analysis (parsing)

Semantic analysis

Sequence of steps

Intermediate Representation

Front End

Lexical Errors

Syntax Errors

Semantic Errors

Tokens

Trees

Symbol Table
Constant table
Other Tables

Intermediate Representation
Lexical Analysis (Scanning)

- It identifies the lexical units in a source statements. It then classifies the units into different lexical classes, e.g. operators, constants, identifiers etc. and enters them into different tables.
- It builds intermediate code that is a sequence of intermediate code units called tokens.
- It's a descriptor for each lexical unit. A token contains two fields – lexical class code and number in class.
- Class code identifies the class to which a lexical unit belongs. Number in class is the entry number of the lexical unit in the relevant table.
- We depict a token as Code # no, e.g. Id # 10
- Example (small program segment)
  ```
  int i;
  float a, b;
  a = b + i;
  ```
Lexical Analysis (Scanning) - Example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Length</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Length</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i*</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>temp</td>
<td>real</td>
<td></td>
</tr>
</tbody>
</table>

Symbol Table at different times of compilation a) after lexical analysis b) during semantic analysis

Example: int i; float a, b;
        a = b + i;

Note that int i first needed to be converted into real, that is why 4th entry is added into the table.

Addition of entry 3 and 4, gives entry 5 (temp), which is value b + (i *).

The statement a = b + i; is represented as the string of tokens

```
Id#2  Op#5  Id#3  Op#3  Id#1  Op#10
```
Syntax Analysis (Parsing)

- It processes the string of tokens built by lexical analysis to determine its grammatical structure.
- It then builds an IC (tree) which represents the structure of a statement.
- Tree can represent the hierarchical structure of statement appropriately.
- The IC is passed to semantic analysis to determine the meaning of the statement.

```
float a, b;
```

```
a = b + i
```
Semantic Analysis

- It identifies the sequence of actions necessary to implement the meaning of a source statement.
- It determines the meaning of a sub tree in the IC, it adds information to a table or adds an action to the sequence of actions. The analysis ends when the tree has been completely processed.
Intermediate Representation by Analysis Phase

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Length</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i*</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>temp</td>
<td>real</td>
<td></td>
</tr>
</tbody>
</table>

Intermediate code:

1. Convert (Id, #1) to real, giving (Id, #4)
2. Add(Id, #4) to (Id, #3), giving (Id, #5)
3. Store (Id, #5) in (Id, #2)
Recall

Analysis Phase (Front end)

Source Program

Lexical analysis (Scanning)

Syntax analysis (parsing)

Semantic analysis

Tokens

Trees

Intermediate Representation (IR)

Lexical Errors

Syntax Errors

Semantic Errors

Front End

Symbol Table
Constant table
Other Tables
Synthesis Phase (Back end)

- It performs memory allocation and code generation.

Diagram:
- Front End
  - IR
    - Memory Allocation
    - Code Generation
      - Symbol Table
      - Constants Table
      - Other tables
    - Target Program
Synthesis Phase (Back end)

- It performs memory allocation and code generation.

**Memory Allocation**
- The memory requirement of an identifier is computed from its type, length and dimensionality and memory is allocated to it.
- The address of the memory area is entered in the symbol table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Length</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>real</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Real</td>
<td>1</td>
</tr>
</tbody>
</table>
Synthesis Phase (Back end)

- **Code Generation**
  - Complex Decision involved to generate Target Code. Two Key decisions are
    - What instruction should be used for each of the actions in the IC?
    - What CPU registers should be used for evaluating expressions?
  - It uses knowledge of the target architecture, viz. knowledge of instructions and addressing modes in the target computer, to select the appropriate instructions.

  \[ a := b + i; \]

  Sequence of actions for the above statement are:
  - i) convert \( i \) to real, giving \( i^* \),
  - ii) Add \( i^* \) to \( b \), giving temp,
  - iii) Store temp in \( a \).

The synthesis phase may decide to hold the values of \( i^* \) and temp in machine registers and may generate the assembly code.

\[
\begin{align*}
\text{CONV}_R & \quad \text{AREG}, \ 1 \\
\text{ADD}_R & \quad \text{AREG}, \ B \\
\text{MOVEM} & \quad \text{AREG}, \ A
\end{align*}
\]
Symbol Tables

- Identifier used in source program call **symbol**.
- A language processor uses the symbol table to maintain information about attributes of symbols used in source program.
- LP performs four kinds operation on symbol table.
  - **Add symbol and its attributes**: Make new entry in symbol table.
  - **Locate a symbol’s entry**: Find a symbol’s entry in the symbol Table.
  - **Delete a symbol’s entry**: Remove the symbol’s information from the table.
  - **Access a symbol’s entry**: Access entry and set, modify or copy its attribute information.
- **Add** and **locate** operations take a **symbol** as a parameter.
- **The delete and access operations may either take a symbol or the pointer to a symbol’s entry returned by the locate operation as parameters.**
- Design of symbol table has two goals:
  - Table’s organization should facilitate efficient search
  - Table should be compact.
Time-Space Trade off

The time required to conduct search and memory occupied by symbol table can be traded off against one another.

Two kinds of data structures used for organizing symbol table entries:
- Linear Data Structure
- Non-Linear Data Structure

Symbol Table Entry Formats:

Attributes in programming language and in specific class
- Fixed Length Entries
  - Each Entry in the symbol table has fields for all attributes specified in programming language.
- Variable length Entries
  - The entry occupied by a symbol has fields only for the attributes specified for symbol of its class
- Hybrid Entries
  - Has a fixed length and variable length part
RECALL

Synthesis Phase
Symbol Table
Elements of Assembly Language Programming

- Machine Dependent, low level programming language
- Each statement in assembly language program corresponds to an instruction in computer or is declaration statement or directive to the assembler.
- Three basis facilities in assembly language
  - Mnemonic Operation codes
  - Symbolic operands
  - Data Declarations
Statement Formats

[Label] <Opcode> <operand specification> [, <operand specification> ..]

[...] is optional

Operand Specification has following syntax

<Symbolic name> [+/- <displacement>][(<index registers> )]

Operand AREA refers the memory word with which name AREA is associated.

AREA+5 . ‘5’ is displacement or offset from AREA

AREA(4) : operand address is obtained by adding the content of index register 4 to the address of AREA.

AREA+5(4)
A simple Assembly language

Each Statement that corresponds to an instruction has two operands, the first operand is always a CPU register which can be any one of AREG, BGER, CREG, DREG. Second operand refers to a memory word by using symbolic name and optional displacement.

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>Assembly Mnemonic</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STOP</td>
<td>STOP Execution</td>
</tr>
<tr>
<td>01</td>
<td>ADD</td>
<td>Perform Addition</td>
</tr>
<tr>
<td>02</td>
<td>SUB</td>
<td>Perform Subtraction</td>
</tr>
<tr>
<td>03</td>
<td>MULT</td>
<td>Perform Multiplication</td>
</tr>
<tr>
<td>04</td>
<td>MOVER</td>
<td>Move from memory to register</td>
</tr>
<tr>
<td>05</td>
<td>MOVEM</td>
<td>Move from register to memory</td>
</tr>
<tr>
<td>06</td>
<td>COMP</td>
<td>Compare and set condition code</td>
</tr>
<tr>
<td>07</td>
<td>BC</td>
<td>Branch on condition</td>
</tr>
<tr>
<td>08</td>
<td>DIV</td>
<td>Perform Division</td>
</tr>
<tr>
<td>09</td>
<td>READ</td>
<td>Read into register</td>
</tr>
<tr>
<td>10</td>
<td>PRINT</td>
<td>Print contents of Register</td>
</tr>
</tbody>
</table>
An Arithmetic operation is performed on the contents of CPU register and Memory word specified in instruction.

The result is put in the CPU register and condition code is set in the CPU according to result value.

The MOVE instructions move a value between memory word and CPU. (MOVER and MOVEM)

Comparison statement sets a condition code analogous to subtract instruction without affecting the values of its operands.

The condition code tested by Brach on condition (BC) instruction.

\[
\text{BC } \langle\text{condition code specification}\rangle, \langle\text{memory address}\rangle
\]

LT, GT, LE, GE, EQ, ANY
Format of Machine Instructions

- sign
- opcode
- register operand
- memory operand
Recall

Figure: Assembler
Elements of Assembly Language

1. Mnemonic Operation Code:-
   Eliminates the need to memorize numeric operation code.

2. Symbolic Operands:-
   Symbolic names can be used.

3. Data Declarations:-
   Data can be declared in any form Eg: -5, 10.5 etc.
[Label] <Opcode> <operand Spec> [<operand spec>....]

1. Label: - Is optional.[..]
2. Opcode: - Symbolic opcode
3. Operand: - Symbolic name (Register or Memory variable)
Example:

\textbf{MOV} \quad \textbf{AX, X}

--MOV is a mnemonic opcode.

--AX is a register operand in symbolic form.

--X is a memory operand in symbolic form.
<table>
<thead>
<tr>
<th>Instruction Opcode</th>
<th>Assembly Mnemonic</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STOP</td>
<td>Stop Execution</td>
</tr>
<tr>
<td>01</td>
<td>ADD</td>
<td>Op1 ← Op1+ Op2</td>
</tr>
<tr>
<td>02</td>
<td>SUB</td>
<td>Op1 ← Op1 – Op2</td>
</tr>
<tr>
<td>03</td>
<td>MULT</td>
<td>Op1 ← Op1* Op2</td>
</tr>
<tr>
<td>04</td>
<td>MOVER</td>
<td>CPU Reg ← Memory operand</td>
</tr>
<tr>
<td>05</td>
<td>MOVEM</td>
<td>Memory ← CPU Reg</td>
</tr>
<tr>
<td>06</td>
<td>COMP</td>
<td>Sets Condition Code</td>
</tr>
<tr>
<td>07</td>
<td>BC</td>
<td>Branch on Condition</td>
</tr>
<tr>
<td>08</td>
<td>DIV</td>
<td>Op1 ← Op1/ Op2</td>
</tr>
<tr>
<td>09</td>
<td>READ</td>
<td>Operand 2 ← input Value</td>
</tr>
<tr>
<td>10</td>
<td>PRINT</td>
<td>Output ← Operand2</td>
</tr>
</tbody>
</table>

**Fig:** Mnemonic Operation Codes
Instruction Format

Fig: Instruction Format
Assembly Language Statements

- Three kinds of statement
- Imperative statements
  - Indicates action to be performed during execution of program. Each imperative statement translates into one machine instruction
- Declaration Statements
  - Syntax:
    - [LABEL] DS <constant>
    - [LABEL] DC ‘<value>’
  - DS (Declare Storage) reserves and area of memory and associates symbolic name with it.
    - A DS 1
    - G DS 200
DC (declare constants) constructs memory words containing constant

ONE DC '1'

Use of Constants

- Immediate Operands
- Literals

Use of immediate operands require special addressing modes

ADDI AREG, 5

Literal is operand with syntax = '<value>'

ADD AREG, @Five
ADD AREG, ='5' @FIVE DC '5'
int z=5;
x = x + 5;

1. Literal cannot be changed during program execution
2. Literal is more safe and protected than a constant.
3. Literals appear as a part of the instruction.
Assembler Directive

- Instruct the assembler to perform certain actions while assembling program.

Two Directives

START

START <constant>

Instruct the assembler to place first word of the target program generated by it in the memory word having the address <constant>.

END

END [<operand Specification>]

Indicates end of the program. The optional <operand specification> indicates execution of program should begin with the instruction whose address is specified be <operand specification>.
Sample program to find X+Y

START 101
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
Assembly Lang to M/C lang Program

1. Find address of variables and labels.
2. Replace Symbolic addr by numeric addr.
3. Replace Symbolic opcodes by machine opcode.
4. Reserve storage for data.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register</th>
<th>Memory operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>101</td>
<td>LC</td>
</tr>
<tr>
<td>READ X</td>
<td>101</td>
<td>+ 09 0 108</td>
</tr>
<tr>
<td>READ Y</td>
<td>102</td>
<td>+ 09 0 109</td>
</tr>
<tr>
<td>MOVER AREG, X</td>
<td>103</td>
<td>+ 04 1 108</td>
</tr>
<tr>
<td>ADD AREG, Y</td>
<td>104</td>
<td>+ 01 1 109</td>
</tr>
<tr>
<td>MOVEM AREG, RESULT</td>
<td>105</td>
<td>+ 05 0 110</td>
</tr>
<tr>
<td>PRINT RESULT</td>
<td>106</td>
<td>+ 10 0 110</td>
</tr>
<tr>
<td>STOP</td>
<td>107</td>
<td>+ 00 0 000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DS</th>
<th>1</th>
<th>108</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>DS</td>
<td>109</td>
</tr>
<tr>
<td>Y</td>
<td>RESULT</td>
<td>110</td>
</tr>
<tr>
<td>RESULT</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

X DS 1 108
Y DS 1 109
RESULT DS 1 110
END
Factorial of number

START
READ N
MOVER BREG, ONE
MOVEM BREG, TERM
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'
TREM DS 1
END

+ 09 00 113
+ 04 02 116
A Simple Assembly Scheme

- Four step approach to develop a design specification for an Assembler:
  - Identify the information necessary to perform a task
  - Design a suitable data structure to record the information
  - Determine the processing necessary to obtain and maintain the information
  - Determine the processing necessary to perform the task by using recorded information
Synthesis Phase

- **MOVER BREG, ONE**
  - Address of the memory word with which name ONE is associated (depends on source program, so it must be made available by the analysis phase)
  - Machine op codes corresponding to the mnemonic MOVER (not depends on source program, it depends on the assembly language)

- **Use two data structures:**
  - Symbol Table (name, address) – build by analysis phase
  - Mnemonic Table (mnemonic, opcode, length)
Analysis Phase

- The primary function is of building of the symbol table.
- Concept of “Memory Allocation”
- To implement memory allocation a data structure called location counter (LC) is used.
- The LC is always made to contain the address of the next memory word in the target program.
- It is initialized to the constant specified in the START statement.
- To update the contents of LC, analysis phase needs to know lengths of different instructions.
Data structures of the assembler

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>SUB</td>
<td>02</td>
<td>1</td>
</tr>
</tbody>
</table>

Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGAIN</td>
<td>104</td>
</tr>
<tr>
<td>N</td>
<td>113</td>
</tr>
</tbody>
</table>
Pass Structure of Assembler

- Two Pass Translation
  - It can handle forward references easily.
  - LC processing is performed in the first pass and symbol defined in the program are entered into the symbol table.
  - The second pass synthesizes the target form using the address information found in the symbol table.
  - In effect, the first pass performs analysis of the source program while the second pass performs synthesis of the target program.
Two pass assembly

Data Structures

Intermediate Code

Pass I

Pass II

SP

TP
Pass Structure of Assembler

- Single Pass Translation
  - LC processing and construction of the symbol table proceeds as in two pass translation.
  - The problem of forward references is tacked using a process called “backpatching”
  - The operand field of an instruction containing a forward reference is left blank initially. The address of the forward referenced symbol is put into this field when its definition is encountered.
  - `MOVER  BREG, ONE` [ONE is forward reference]
  - Table of Incomplete Instruction (TII)
START 100
MOVER AREG, X
L1 ADD BREG, ONE
ADD CREG, TEN
STOP
X DC ‘5’
ONE DC ‘1’
TEN DC ‘10’
END

LC ➔ 100
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Symbol Making a forward reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>MOVER</td>
<td>AREG, X</td>
<td>100 04 1 ------</td>
</tr>
<tr>
<td>L1 ADD</td>
<td>BREG, ONE</td>
<td>101 1 2 ------</td>
</tr>
<tr>
<td>ADD</td>
<td>CREG, TEN</td>
<td>102 06 3 ------</td>
</tr>
<tr>
<td>STOP</td>
<td></td>
<td>000 00 0</td>
</tr>
<tr>
<td>X DC</td>
<td>‘5’</td>
<td>104</td>
</tr>
<tr>
<td>ONE DC</td>
<td>‘1’</td>
<td>105</td>
</tr>
<tr>
<td>TEN DC</td>
<td>‘10’</td>
<td>106</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure : TII
# Machine Instruction After Backpatching

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>1</td>
<td>104</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>105</td>
</tr>
<tr>
<td>06</td>
<td>2</td>
<td>106</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>000</td>
</tr>
</tbody>
</table>
Advanced Assembler Directives

- ORIGIN
  - ORIGIN <address spec>
  - Where <address spec> is an <operand spec> or <constant>
  - This directive indicates that LC should be set to the address given by <address spec>.
  - It is useful when the target program does not consist of consecutive memory words.
Advanced Assembler Directives

- **EQU**
  - `<symbol> EQU <address spec>`
  - Where `<address spec>` is an `<operand spec>` or `<constant>`
  - It defines the symbol to represent `<address spec>`.

- **LTORG**
  - It permits a programmer to specify where literals should be placed.
  - By default, assembler places it after the END statement.
At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of a literal pool. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Assembly</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>START</td>
<td>200</td>
<td>04 1 211</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MOVER</td>
<td>AREG, =‘5’</td>
<td>200)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MOVEM</td>
<td>AREG, A</td>
<td>05 1 217</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LOOP</td>
<td></td>
<td>04 1 217</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MOVER</td>
<td>CREG, B</td>
<td>05 3 218</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ADD</td>
<td>CREG, =‘1’</td>
<td>01 3 212</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BC</td>
<td>ANY, NEXT</td>
<td>07 6 214</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>LTORG</td>
<td></td>
<td>00 0 005</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>00 0 001</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>NEXT</td>
<td>SUB</td>
<td>02 1 219</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>LT, BACK</td>
<td>07 1 202</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>LAST</td>
<td>STOP</td>
<td>00 0 000</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>ORIGIN</td>
<td>03 3 218</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>ORIGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>A</td>
<td>DS</td>
<td>00 0 001</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>BACK</td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>B</td>
<td>DS</td>
<td>00 0 001</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>END</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design of a Two Pass Assembler

Tasks performed by the passes of a two pass assembler are as follows:

- **Pass I**
  - Separate the symbol, mnemonic opcode, operand fields
  - Build the symbol table
  - Perform LC processing
  - Construct intermediate representation

- **Pass II**
  - Synthesis the target program
Pass I of an assembler

- It uses the following data structures:
  - OPTAB – A table of mnemonic opcodes and related information
  - SYMTAB – Symbol Table
  - LITTAB – A table of literals used in the program
Pass I

- Pass I uses the following data structures
  1. Machine Opcode table (OPTAB)
  2. Symbol Table (ST)
  3. Literal Table (LT)
  4. Pool Table (PT)
1. OPTAB contains mnemonic opcode, class and mnemonic info.
   i) opcode :- Mnemonic Opcode e.g. READ, START..
   ii) Class:- Class field indicates whether opcode corresponds to imperative Statements(IS), Declarative Statements(DL) or Assembler Directive (AS)
   iii) Mnemonic field:- if an imperative statement, the mnemonic field contains pair (Machine Opcode, instruction Length) or otherwise it contains id of routine that handles declaration or directive statement. The routine is called to perform appropriate processing of the statement.

<table>
<thead>
<tr>
<th>Mnemonic Opcode</th>
<th>Class</th>
<th>Mnemonic info</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVER</td>
<td>IS</td>
<td>(04, 1)</td>
</tr>
<tr>
<td>DS</td>
<td>DL</td>
<td>R#7</td>
</tr>
<tr>
<td>START</td>
<td>AD</td>
<td>R#11</td>
</tr>
</tbody>
</table>
2. SYMTAB contains symbol, address and length.
3. LITTAB contains literal and address.
4. POOLTAB contains information concerning literal pools.
   • Each entry in LITTAB relates to Literals.
   • It contains field literal and address.
   • Entry in POOLTAB relates to Literal pool.
   • It contains single field literal no indicate which entry in the LITTAB contains first literal of pool.
```
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       ...

8       BC     ANY, NEXT
12      LTORG  =‘5’
13
13      =‘1’
14      ...
15      NEXT SUB  AREG, =‘1’
16      BC     LT, BACK
17      LAST STOP
18      ORIGIN LOOP+2
19      MULT CREG, B
20      ORIGIN LAST+1
21      A     DS   1
22      BACK EQU LOOP
23      B     DS   1
24      END
25      =‘1’
```
START 200
MOWER AREG, =‘5’ 200) +04 1 211
MOVEM AREG, A 201) +05 1 217
LOOP
MOWER AREG, A 202) +04 1 217
MOWER CREG, B 203) +05 3 218
ADD CREG, =‘1’ 204) +01 3 212
...
BC ANY, NEXT 210) +07 6 214
LTORG
=‘5’ 211) +00 0 005
=‘1’ 212) +00 0 001
...
NEXT SUB AREG, =‘1’ 214) +02 1 219
BC LT, BACK 215) +07 1 202
LAST STOP
ORIGIN LOOP+2 216) +00 0 000
MULT CREG, B 204) +03 3 218
ORIGIN LAST+1
A DS 1 217)
BACK EQU LOOP
B DS 1 218)
END
=‘1’ 219) +00 0 001
### Data Structures of Assembler

#### OPTAB

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>class</th>
<th>mnemonic info</th>
<th>symbol</th>
<th>address</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVER</td>
<td>IS</td>
<td>(04,1)</td>
<td>LOOP</td>
<td>202</td>
<td>1</td>
</tr>
<tr>
<td>DS</td>
<td>DL</td>
<td>R#7</td>
<td>NEXT</td>
<td>214</td>
<td>1</td>
</tr>
<tr>
<td>START</td>
<td>AD</td>
<td>R#11</td>
<td>LAST</td>
<td>216</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>217</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BACK</td>
<td>202</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>218</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SYMTAB

<table>
<thead>
<tr>
<th>literal</th>
<th>address</th>
<th>literal no</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>=‘5’</td>
<td>#1</td>
</tr>
<tr>
<td>2</td>
<td>=‘1’</td>
<td>#3</td>
</tr>
<tr>
<td>3</td>
<td>=‘1’</td>
<td>–</td>
</tr>
</tbody>
</table>

#### LITTAB

#### POOLTAB
Intermediate Code (IC) Forms

- Two Criteria for choice Intermediate Code
  - Processing Efficiency
  - Memory Efficiency
- Design or Consider Two variants (alternative) of intermediate code and comparing them on above criteria
- Intermediate code is sequence of Intermediate code units (IC).
- Each IC unit consists three fields
  - Address
  - Representation of Mnemonic opcode
  - Representation of Operands

<table>
<thead>
<tr>
<th>Address</th>
<th>Mnemonic Opcode</th>
<th>Operands</th>
</tr>
</thead>
</table>

Intermediate Code (IC) Forms

- Mnemonic Opcode Field contains pair of the form
  - (Statement class, code)
- Statement class can be IS, DL and AD.
- For Imperative statement, code is instruction opcode in machine language.
- For declarations and Assembler Directive, code is ordinal number within class.
- Code for DL and AD

<table>
<thead>
<tr>
<th>Declaration Statements</th>
<th>Assembler Directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>START 01</td>
</tr>
<tr>
<td>DS</td>
<td>END 20</td>
</tr>
<tr>
<td></td>
<td>ORIGIN 03</td>
</tr>
<tr>
<td></td>
<td>EQU 04</td>
</tr>
<tr>
<td></td>
<td>LTORG 05</td>
</tr>
</tbody>
</table>
The First Operand in assembly statement is represented by single digit number which is either a code in the range 1…4 that represents CPU Register. Or

The condition code itself, which is in the range 1….6.

The Second Operand is represented by pair of form

- (Operand Class, code)

Operand Class is one of C, S and L standing for constant, symbol and literal respectively.

For constant, code field contains constant itself.

For Symbol or literal, code field contains entry number of the operand (symbol or literal) in SYMTAB or LITTAB.
Intermediate Code - variant I

```
START   100
READ    A
LOOP
    MOVER   AREG, A
    SUB     AREG, ='1'
    BC      GT, LOOP
STOP
A
DS    1
LTORG
```
### RECALL

- LITTAB
- POOLTAB
- INTERMEDIATE CODE/INTERMEDIATE REPRESENTATION
- IC CODE UNITS
  - ADDRESS
  - REPRESENTATION OF MNEMONIC OPCODE
  - REPRESENTATION OF OPERANDS
- IC VARIAT 1
The First Operand in assembly statement is keep as it is in source form.

The Second Operand is represented by pair of form 

(Operand Class, code)

Operand Class is one of C and L standing for constant and literal respectively.

For constant, code field contains constant itself.

For literal, code field contains entry number of the operand (symbol or literal) in SYMTAB or LITTAB.

For Symbol, keep symbol as it is.
This variant differs from variant I of the intermediate code in that the operand fields of the source statements are selectively replaced by their processed forms (see Fig. 4.13). For declarative statements and assembler directives, processing of the operand fields is essential to support LC processing. Hence these fields contain the processed forms. For imperative statements, the operand field is processed only to identify literal references. Literals are entered in LITTAB, and are represented as \((L, m)\) in IC. Symbolic references in the source statement are not processed at all during Pass I.
Intermediate Code - variant II

START 100
READ A
LOOP MOVER AREG, A
.. ...
SUB AREG, =‘1’
BC GT, LOOP
STOP
A DS 1
LTORG

(AD, 01) (C, 200)
(IS, 09) A
(IS, 04) AREG, A
(IS, 02) AREG, (L, 01)
(IS, 07) GT, LOOP
(IS, 00)
(DL, 02) (C, 1)
(AD, 05)
Comparison of Variant I and Variant II

- **Processing:**
  - Variant I - All operands Processed.
  - Variant II - Some operands Processed and Main focus on literal Processing.

- **IC Unit**
  - Variant I - Compact.
  - Variant II - less compact

- **Pass I:**
  - Variant I - Extra Work Perform in Pass I.
  - Variant II - Reduces work of Pass I

- **Memory Utilization:**
  - Variant I - Utilization Is not Balanced.
  - Variant II - Utilization Is Balanced.

- **Memory:**
  - Variant I - Occupies more memory in Pass I
  - Variant II - Occupies less memory.
Processing of Declarations and Assembler Directive

• Alternative ways of processing declaration statements and assembler directive.

• Two key questions will arise in this context:
  • Is it necessary to represent the address of each source statement in the intermediate code?
  • Is it necessary to have representation of DS statements and Assembler Directive in the intermediate code

<table>
<thead>
<tr>
<th>START</th>
<th>100</th>
<th>--</th>
<th>(AD, 01)</th>
<th>(C, 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DS</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>DS</td>
<td>2</td>
<td>100</td>
<td>(DL, 02)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>120</td>
<td>(DL, 02)</td>
</tr>
</tbody>
</table>

IC Syntax:
- Address Representation of Mnemonic Code Representation of Operands

• If the intermediate code does not contain address of each source statement, a representation of for the DS statements and Assembler directive would have been necessary.
• DC Statement:-
  • A DC Statement must be represented in the intermediate code.

• START and ORIGIN
  • These directives set the new values into the location counter. It is not necessary to retain START and ORIGIN Statements in the intermediate code if the intermediate code contains address field.
PASS II of the Assembler

• Pass I contains Intermediate code, SYMTAB and Supporting Table.
• Pass II Uses these data structures and coverts the intermediate code into target code with the help of SYMTAB and LITTAB Entries.
Program Listing and Error Reporting

• Assembler Produce a Program Listing that shows a source statement and Target code.
• The listing also reports any error that assembler might fount in the program.
• Error Reporting is most effective when listing shows and error against erroneous statement itself.
• The most important decision is whether to report error in Pass I or postpone it until pass II.
• The reporting of errors in pass I have advantages:
  • The source code statements need not be kept till Pass II.
  • Minimization of Memory Requirements.
  • The duplicate processing is avoided.
  • The error can be reported most significant position.
Error Reporting in Pass I

START 100
MOVER AREG, A

MVER BREG, A
** Error** Invalid Opcode
ADD BREG, B

A DS 1

A DC ‘5’
** ERROR** Duplicate Definition of Symbol A

END
**error** Use of Undefined symbol B in the statement 10.
Show the content of Symbol Table, Opcode table, Literal table and Pool Table. Also Generate Intermediate code (Variant I and Variant II) AND TARGET CODE.

<table>
<thead>
<tr>
<th>Instruction Opcode</th>
<th>Assembly Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STOP</td>
</tr>
<tr>
<td>01</td>
<td>ADD</td>
</tr>
<tr>
<td>02</td>
<td>SUB</td>
</tr>
<tr>
<td>03</td>
<td>MULT</td>
</tr>
<tr>
<td>04</td>
<td>MOVER</td>
</tr>
<tr>
<td>05</td>
<td>MOVEM</td>
</tr>
<tr>
<td>06</td>
<td>COMP</td>
</tr>
<tr>
<td>07</td>
<td>BC</td>
</tr>
<tr>
<td>08</td>
<td>DIV</td>
</tr>
<tr>
<td>09</td>
<td>READ</td>
</tr>
<tr>
<td>10</td>
<td>PRINT</td>
</tr>
</tbody>
</table>

### Assembler Directives

<table>
<thead>
<tr>
<th>Assembler Directive</th>
<th>Instruction Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>01</td>
</tr>
<tr>
<td>END</td>
<td>20</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>03</td>
</tr>
<tr>
<td>EQU</td>
<td>04</td>
</tr>
<tr>
<td>LTORG</td>
<td>05</td>
</tr>
</tbody>
</table>

### Declaration Statements

<table>
<thead>
<tr>
<th>Declaration Statement</th>
<th>Instruction Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>01</td>
</tr>
<tr>
<td>DS</td>
<td>02</td>
</tr>
<tr>
<td>‘0’</td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ONE</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
Show the content of Symbol Table, Opcode table, Literal table and Pool Table. Also Generate Intermediate code (Variant I and Variant II).

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>Opcode</th>
<th>Assembly Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>START 100</td>
<td>00</td>
<td>STOP</td>
</tr>
<tr>
<td>L1 MOVER AREG,B</td>
<td>01</td>
<td>ADD</td>
</tr>
<tr>
<td>ADD AREG, C</td>
<td>02</td>
<td>SUB</td>
</tr>
<tr>
<td>MOVEM AREG,D</td>
<td>03</td>
<td>MULT</td>
</tr>
<tr>
<td>D EQU A+1</td>
<td>04</td>
<td>MOVER</td>
</tr>
<tr>
<td>L2 PRINT D</td>
<td>05</td>
<td>MOVEM</td>
</tr>
<tr>
<td>ORIGIN A-1</td>
<td>06</td>
<td>COMP</td>
</tr>
<tr>
<td>C DC ‘5’</td>
<td>07</td>
<td>BC</td>
</tr>
<tr>
<td>ORIGIN L2+4</td>
<td>08</td>
<td>DIV</td>
</tr>
<tr>
<td>STOP</td>
<td>09</td>
<td>READ</td>
</tr>
<tr>
<td>B DC ‘19’4</td>
<td>10</td>
<td>PRINT</td>
</tr>
</tbody>
</table>

### Assembler Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>01</td>
</tr>
<tr>
<td>END</td>
<td>20</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>03</td>
</tr>
<tr>
<td>EQU</td>
<td>04</td>
</tr>
<tr>
<td>LTORG</td>
<td>05</td>
</tr>
</tbody>
</table>

### Declaration Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>01</td>
</tr>
<tr>
<td>DS</td>
<td>02</td>
</tr>
</tbody>
</table>
Show the content of Symbol Table, Opcode table, Literal table and Pool Table. Also Generate Intermediate code (Variant I and Variant II) AND TARGET CODE.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>200</td>
<td>STOP</td>
</tr>
<tr>
<td>MOVER</td>
<td>AREG,=‘5’</td>
<td>00</td>
</tr>
<tr>
<td>MOVEM</td>
<td>AREG, A</td>
<td>01</td>
</tr>
<tr>
<td>MOVER</td>
<td>AREG, A</td>
<td>02</td>
</tr>
<tr>
<td>MOVER</td>
<td>AREG, B</td>
<td>03</td>
</tr>
<tr>
<td>MULT</td>
<td>AREG,=‘5’</td>
<td>04</td>
</tr>
<tr>
<td>SUB</td>
<td>AREG, A</td>
<td>05</td>
</tr>
<tr>
<td>COMP</td>
<td>AREG, ONE</td>
<td>06</td>
</tr>
<tr>
<td>LTORG</td>
<td>Z</td>
<td>07</td>
</tr>
<tr>
<td>PRINT</td>
<td>Z</td>
<td>08</td>
</tr>
<tr>
<td>STOP</td>
<td></td>
<td>09</td>
</tr>
<tr>
<td>X</td>
<td>DS</td>
<td>100</td>
</tr>
<tr>
<td>Y</td>
<td>DS</td>
<td>2</td>
</tr>
<tr>
<td>ONE</td>
<td>DC</td>
<td>‘0’</td>
</tr>
<tr>
<td>Z</td>
<td>DS</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>DS</td>
<td>1</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Directives**

<table>
<thead>
<tr>
<th>Directive</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>01</td>
</tr>
<tr>
<td>END</td>
<td>20</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>03</td>
</tr>
<tr>
<td>EQU</td>
<td>04</td>
</tr>
<tr>
<td>LTORG</td>
<td>05</td>
</tr>
</tbody>
</table>

**Declaration Statements**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>01</td>
</tr>
<tr>
<td>DS</td>
<td>02</td>
</tr>
</tbody>
</table>
Intermediate code for Imperative Statements

We consider two variants of intermediate code which differ in the information contained in their operand fields. For simplicity, the address field is assumed to contain identical information in both variants.

Variant I and Variant II
Pass II of an assembler

- Tables

For efficiency reasons SYMTAB must remain in main memory throughout Passes I and II of the assembler. LITTAB is not accessed as frequently as SYMTAB, however it may be accessed sufficiently frequently to justify its presence in the memory. If memory is at a premium, it is possible to hold only part of LITTAB in the memory because only the literals of the current pool need to be accessible at any time. For obvious reasons, no such partitioning is feasible for SYMTAB. OPTAB should be in memory during Pass I.
Pass II of an assembler

- Source Program and Intermediate Code

The source program would be read by Pass I on a statement by statement basis. After processing, a source statement can be written into a file for subsequent use in Pass II. The IC generated for it would also be written into another file. The target code and the program listings can be written out as separate files by Pass II. Since all these files are sequential in nature, it is beneficial to use appropriate blocking and buffering of records.
START

000 0 00 + 011 007 010
010 0 10 110 012 016 010
010 0 05 110 010 015 016
010 0 01 109 010 014 015
010 0 04 108 010 013 015
009 0 06 109 010 012 015
010 0 09 108 010 011 015

RESULT DS 1
RESULT DS 1
RESULT DS 1
STOP
PRINT RESULT 106
MOVEM AREG,RESULT 105
ADDS AREG,Y 104
ADDS AREG,X 103
MOVER AREG,X 102
READ X 101
READ Y 101
START 101

END

MEMORY

Opcode
Register
Memory
Operand

Pass II of an assembler