GROUP - A

EXPERIMENT NO: 01
1.1

1. Title:

Design suitable data structures and implement pass-I of a two-pass assembler for pseudo-machine in Java using object oriented feature. Implementation should consist of a few instructions from each category and few assembler directives.

2. Objectives:

- To understand data structures to be used in pass I of an assembler.
- To implement pass I of an assembler

3. Problem Statement:

Write a program to create pass-I Assembler of two-pass assembler

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the concept of Pass-I Assembler
- Understand the Programming language of Java

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

A language translator bridges an execution gap to machine language of computer system. An assembler is a language translator whose source language is assembly language.

An Assembler is a program that accepts as input an Assembly language program and converts it into machine language.

Language processing activity consists of two phases, Analysis phase and synthesis phase. Analysis of source program consists of three components, Lexical rules, syntax rules and semantic rules. Lexical rules govern the formation of valid statements in source language. Semantic rules associate the formation meaning with valid statements of language. Synthesis phase is concerned with construction of target language statements, which have the same meaning as source language statements. This consists of memory allocation and code generation.

TWO PASS TRANSLATION SCHEME:

In a 2-pass assembler, the first pass constructs an intermediate representation of the source program for use by the second pass. This representation consists of two main components - data structures like Symbol table, Literal table and processed form of the source program called as intermediate code(IC). This intermediate code is represented by the syntax of Variant –I.

Analysis of source program statements may not be immediately followed by synthesis of equivalent target statements. This is due to forward references issue concerning memory requirements and organization of Language Processor (LP).

Forward reference of a program entity is a reference to the entity, which precedes its definition in the program. While processing a statement containing a forward reference, language processor does not posses all relevant information concerning referenced entity. This creates difficulties in synthesizing the equivalent target statements. This problem can be solved by postponing the generation of target code until more information concerning the entity is available. This also reduces memory requirements of LP and simplifies its organization. This leads to multi-pass model of language processing.

Language Processor Pass: -

It is the processing of every statement in a source program or its equivalent representation to perform language-processing function.

Assembly Language statements: -

There are three types of statements Imperative, Declarative, Assembly directives. An imperative statement indicates an action to be performed during the execution of assembled program. Each imperative statement usually translates into one machine instruction. Declarative statement e.g. DS reserves areas of memory and associates names with them. DC constructs memory word containing constants. Assembler directives instruct the assembler to perform certain actions during assembly of a program,

e.g. START<constant> directive indicates that the first word of the target program generated by assembler should be placed at memory word with address <constant>

Function Of Analysis And Synthesis Phase:

Analysis Phase: -

Isolate the label operation code and operand fields of a statement.

Enter the symbol found in label field (if any) and address of next available machine word into symbol table.

Validate the mnemonic operation code by looking it up in the mnemonics table.Determine the machine storage requirements of the statement by considering the mnemonic operation code and operand fields of the statement.

Calculate the address of the address of the first machine word following the target code generated for this statement (Location Counter Processing)

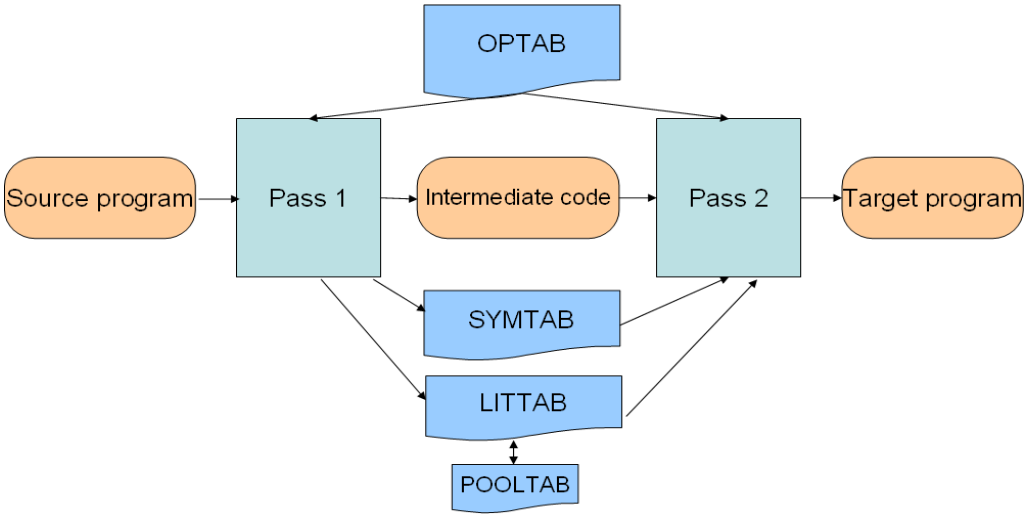
Synthesis Phase:

Obtain the machine operation code corresponding to the mnemonic operation code by searching the mnemonic table.

Obtain the address of the operand from the symbol table.

Synthesize the machine instruction or the machine form of the constant as the case may be.

DATA STRUCTURES OF A TWO PASS ASSEMBLER:



Data Structure of Assembler:

a) Operation code table (OPTAB) :This is used for storing mnemonic,

operation code and class of instruction

Structure of OPTAB is as follows

b) Data structure updated during translation: Also called as translation time data

structure. They are

I. SYMBOL TABLE (SYMTAB) : Ii contains entries such as

symbol,it's address and value.

SYMBOL TABLE have following fields :

Name of symbol	Symbol Address	Value

II. LITERAL TABLE (LITTAB) : it contains entries such as literal

and it's value.

Literal Table has following fields :

literal	**Address of Literal**

III . POOL TABLE (POOLTAB): Contains literal number of the

starting literal of each literal pool.

Pool TABLE (pooltab) have following fields.

LITERAL_NO

IV: Location Counter which contains address of next instruction by

calculating length of each instruction.

Design of a Two Pass Assembler: -

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

Pass I: -

Separate the symbol, mnemonic opcode and operand fields.

Determine the storage-required foe every assembly language statement and update the location counter.

Build the symbol table and the literal table.

Construct the intermediate code for every assembly language statement.

Pass II: -

Synthesize the target code by processing the intermediate code generated during pass1

INTERMEDIATE CODE REPRESENTATION

The intermediate code consists of a set of IC units, each IC unit consisting of the following three fields

1. Address
2. Representation of the mnemonic opcode
3. Representation of operands

Mnemonic field

The mnemonic field contains a pair of the form: *(statement class, code)*

Where *statement class* can be one of IS,DL and AD standing for imperative statement, declaration statement and assembler directive , respectively.

For imperative statement, *code* is the instruction opcode in the machine language

For declaration and assembler directives , following are the codes

Declaration Statements		Assembler directives	
DC	01	START	01
DS	02	END	02
		ORIGIN	03
		EQU	04
		LTORG	05

Mnemonic Operation Codes :

Instruction Opcode	Assembly Mnemonic	Remarks
00	**STOP**	**STOP EXECUTION**
01	**ADD**	**FIRST OPERAND IS MODIFIED CONDITION CODE IS SET**
02	**SUB**	**FIRST OPERAND IS MODIFIED CONDITION**
		CODE IS SET
03	**MULT**	**FIRST OPERAND IS MODIFIED CONDITION CODE IS SET**

04	**MOVER**	**REGISTER MEMORY MOVE**
05	**MOVEM**	**MEMORY MOVE 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**

Branch On Condition :

BC <condition code > <memory address>

Transfer Control to the Memory word With Address < memory address>

Condition code	Opcode
LT	01
LE	02
EQ	03
GT	04
GE	05
ANY	06

8. Algorithms(procedure) :

PASS 1

• Initialize location counter, entries of all tables as zero.

• Read statements from input file one by one.

• While next statement is not END statement

I. Tokenize or separate out input statement as label,numonic,operand1,operand2

II. If label is present insert label into symbol table.

III. If the statement is LTORG statement processes it by making it's entry into literal table, pool table and allocate memory.

IV. If statement is START or ORIGEN Process location counter accordingly.

V. If an EQU statement, assign value to symbol by correcting entry in symbol table.

VI. For declarative statement update code, size and location counter.

VII. Generate intermediate code.

VIII. Pass this intermediate code to pass -2.

Input:

Source code of Assembly Language

```
SAMPLE START 100
            USING *, 15
    L 1, FOUR
    A 1, =F'3'
    ST 1, RESULT
    SR 1, 2
    LTORG
      L 2, FIVE
    A 2, =F'5'
    A 2, =F'7'
      FIVE DC F'5'
      FOUR DC F'4'
    RESULT DS 1F
    END
```

Output:

```
100 SAMPLE START 100
100 USING *, 15
100 L 1, FOUR
104 A 1, =F'3'
108 ST 1, RESULT
112 SR 1, 2
114 LTORG
124 L 2, FIVE
128 A 2, =F'5'
132 A 2, =F'7'
136 FIVE DC F'5'
140 FOUR DC F'4'
144 RESULT DS 1F
152 5
156 7
160 END
```

Machine Opcode Table (MOT)

Mnemonic	Hex / Binary Code	Length (Bytes)	Format

L Conclusion:	5A	4	RX
A	1B	4	RX
ST	50	4	RX
SR	18	2	RR

Symbol Table (ST)

Sr. No	Symbol name	Address	Value	Length	Relocation
1	SAMPLE	100	--	160	R
2	FIVE	136	5	4	R
3	FOUR	140	4	4	R
4	RESULT	144	__	4	R

Literal Table (LT)

Sr. No	Literal	Address	Length
1	3	120	4
2	5	152	4
3	7	156	4

10. Conclusion:

Input assembly language program is processed by applying Pass-I algorithm of assembler and intermediate data structures, Symbol Table, Literal Table, etc. are generated.

GROUP - A

EXPERIMENT NO:1.2

1. Title:

Implement Pass-II of two pass assembler for pseudo-machine in Java using object oriented features. The output of assignment-1 (intermediate file and symbol table) should be input for this assignment.

2. Objectives:

- To understand data structures to be used in pass II of an assembler.
- To implement pass I of an assembler

3. Problem Statement:

Write a program to create pass-II Assembler

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the concept of Pass-II Assembler
- Understand the Programming language of Java

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

Design of a Two Pass Assembler: -

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

Pass I: -

Separate the symbol, mnemonic opcode and operand fields.

Determine the storage-required foe every assembly language statement and update the location counter.

Build the symbol table and the literal table.

Construct the intermediate code for every assembly language statement.

Pass II: -

Synthesize the target code by processing the intermediate code generated during pass1

Data Structure used by Pass II:

1. OPTAB: A table of mnemonic opcodes and related information.

2. SYMTAB: The symbol table

3. POOL_TAB and LITTAB: A table of literals used in the program

4. Intermediate code generated by Pass I

5. Output file containing Target code / error listing.

8. Algorithms(procedure) :
Algorithm :

1. **c**ode_area_address=address of code area;

Pooltab_ptr:=1;

loc_cntr=0;

2. While next statement is not an END statement

a) clear the machine_code_buffer

b) if an LTORG statement

I) process literals in LITTAB[POOLTAB[pooltab_ptr]]…

LITTAB[POOLTAB[pooltab_ptr+1]]-1 similar to processing of constants in a dc statement.

II) size=size of memory area required for literals

III) pooltab_ptr=pooltab_ptr+1

c) if a START or ORIGIN statement then

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 or DS:

e) if an imperative statement then

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 of END statement)

Input:
Intermediate code of pass-1.

LC LABEL INSTR. OPERANDS

---

100 SAMPLE START 100
100 USING *, 15
100 L 1, FOUR
104 A 1, =F'3'
108 ST 1, RESULT
112 SR 1, 2
114 LTORG
124 L 2, FIVE
128 A 2, =F'5'
132 A 2, =F'7'
136 FIVE DC F'5'
140 FOUR DC F'4'
144 RESULT DS 1F
152 5
156 7
160 END

Machine Opcode Table (MOT)

Mnemonic	Hex / Binary Code	Length (Bytes)	Format
L	5A	4	RX
A	1B	4	RX
ST	50	4	RX

SR	18	2	RR

Symbol Table (ST)

Sr. No	Symbol name	Address	Value	Length	Relocation
1	SAMPLE	100	--	160	R
2	FIVE	136	5	4	R
3	FOUR	140	4	4	R
4	RESULT	144	__	4	R

Literal Table (LT)

Sr. No Literal Address Length
1 3 120 4
2 5 152 4
3 7 156 4

Output:

Base Table (BT)

Register no Availability Value/ Contents
1 N --
: : :
: : :
: : :
15 Y 100

Object Code

LC OPCODE OPERAND

---

100 5A 1,40(0,15)
104 1B 1,20(0,15)
108 50 1,44(0,15)
112 18 1,2
124 5A 2,36(0,15)
128 1B 2,52(0,15)
132 1B 2,56(0,15)

Conclusion:

The intermediate data structures generated in Pass-I of assembler are given as input to the Pass-II of assembler, processed by applying Pass-II algorithm of assembler and machine code is generated

GROUP - A

EXPERIMENT NO: 02

1. Title:

Design suitable data structures and implement pass-I of a two-pass macro-processor using OOP features in Java

2. Objectives:

- To Identify and create the data structures required in the design of macro processor.
- To Learn parameter processing in macro
- To implement pass I of macroprocessor

3. Problem Statement:

Write a program to create pass-I Macro-processor

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the Programming language of Java
- Understand the concept of Pass-I Macro-processor

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

MACRO

Macro allows a sequence of source language code to be defined once & then referred to by name each time it is to be referred. Each time this name occurs is a program the sequence of codes is substituted at that point.

A macro consist of

1. Name of the macro
2. Set of parameters
3. Body of macro

Macros are typically defined at the start of program. Macro definition consists of

1. MACRO pseudo
2. MACRO name
3. Sequence of statements
4. MEND pseudo opcode terminating

A macro is called by writing the macro name with actual parameter is an assembly program. The macro call has following syntax <macro name>.

MACRO PROCESSOR

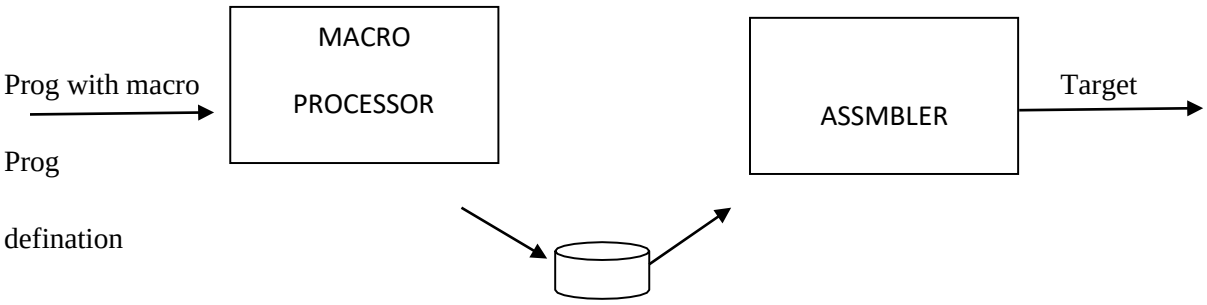


Fig 1. Assembly language program without macro

Macro processor takes a source program containing macro definition & macro calls and translates into an assembly language program without any macro definition or calls. This program can now be handled over to a conventional assembler to obtain the target language.

MACRO DEINITION

Macros are typically defined at the start of a program. A macro definition consists of

1. MACRO pseudo code
2. MACRO name
3. Sequence of statement
4. MEND pseudo opcode terminating macro definition

Structure of a macro

Example
MACRO
INCR & ARG
ADD AREG,& ARG
ADD BRA,& ARG
ADD CREG, & ARG
MEND

MACRO Expansion:

During macro expansion each statement forming the body of the macro as picked up one by one sequentially.

a. Each statement inside macro may have as it is during expansion.

b. The name of a formal parameter which is preceded by the character '&' during macro expansion an ordinary starting is retained without any modification. Formal parameters are replaced by actual parameters value.

When a call is found the call processor sets a pointer the macro definition table pointer to the corresponding macro definition started in MDT. The initial value of MDT is obtained from MDT index.

8. Design of macro processor:

- **Pass I:**
  - Generate Macro Name Table (MNT)
  - Generate Macro Definition Table (MDT)
  - Generate IC i.e. a copy of source code without macro definitions.

MNT:

Sr.No	Macro Name	MDT Index

MDT:

Sr. No	MACRO STATEMENT

ALA:

Index	Argument

Specification of Data Bases

Pass 1 data bases

1. The input macro source desk.
2. The output macro source desk copy for use by passes 2.
3. The macro definition table (MDT) used to store the names of defined macros.
4. Macro name table (MDT) used to stare the name of defined macros.
5. The Macro definition table counter used to indicate the next available entry in MNT.
6. The macro name table counter counter(MNTC) used to indicate next available entry in MNT.
7. The arguments list array (ALA) used to substitute index markers for dummy arguments before starting a macro definition.

TAKE AN EXAMPLE AND SOLVE

9. **Conclusion:**

Thus we have successfully implemented pass-I of a two-pass Macro-processor.

GROUP - B

EXPERIMENT NO:
2.2

1. Title:

Write a Java program for pass-II of a two-pass macro-processor. The output of assignment-3 (MNT, MDT and file without any macro definitions) should be input for this assignment.

2. Objectives:

- To Identify and create the data structures required in the design of macro processor.
- To Learn parameter processing in macro
- To implement pass II of macroprocessor

3. Problem Statement:

Write a program to create pass-II Macro-processor

4. Outcomes:

After completion of this assignment students will be able to:

- Understand the Programming language of Java
- Understand the concept of Pass-II Macro-processor

5. Software Requirements:

- Linux OS, JDK1.7

6. Hardware Requirement:

- 4GB RAM ,500GB HDD

7. Theory Concepts:

Pass II:

Replace every occurrence of macro call with macro definition. (Expanded Code)

There are four basic tasks that any macro instruction process must perform:

1. Recognize macro definition:

A macro instruction processor must recognize macro definition identified by the MACRO and MEND pseudo-ops. This tasks can be complicated when macro definition appears within macros. When MACROs and MENDs are nested, as the macro processor must recognize the nesting and correctly match the last or or outer MEND with first MACRO. All intervening text, including nested MACROs and MENDs defines a single macro instruction.

2. **Save the definition:**

The processor must store the macro instruction definition, which it will need for expanding macro calls.

3. **Recognize calls:**

The processor must recognize the macro calls that appear as operation mnemonics. This suggests that macro names be handled as a type of op-code.

4. **Expand calls and substitute arguments:**

The processor must substitute for dummy or macro definition arguments the corresponding arguments from a macro call; the resulting symbolic text is then substitute for macro call. This text may contain additional macro definition or call.

Implementation of a 2 pass algorithm

1. We assume that our macro processor is functionally independent of the assembler and that the output text from the macro processor will be fed into the assembler.
2. The macro processor will make two independent scans or passes over the input text , searching first for macro definitions and then for macro calls
3. The macro processor cannot expand a macro call before having found and saved the corresponding macro definitions.
4. Thus we need two passes over the input text , one to handle macro definitions and other to handle macro calls.
5. The first pass examines every operation code, will save all macro definitions in a macro Definition Table and save a copy of the input text, minus macro definitions on the secondary storage.
6. The first pass also prepares a Macro Name Table along with Macro Definition Table as seen in the previous assignment that successfully implemented pass – I of macro pre-processor.

The second pass will now examine every operation mnemonic and replace each macro name with the appropriate text from the macro definitions.

8. SPECIFICATION OF DATABASE

Pass 2 database:

1. The copy of the input source deck obtained from Pass- I
2. The output expanded source deck to be used as input to the assembler
3. The Macro Definition Table (MDT), created by pass 1
4. The Macro Name Table (MNT), created by pass 1
5. The Macro Definition Table Counter (MNTC), used to indicate the next line of text tobe used during macro expansion
6. The Argument List Array (ALA), used to substitute macro call arguments for the indexmarkers in stored macro definition

TAKE AN EXAMPLE AND SOLVE

9. Conclusion:

1.