System Programming & Compiler Construction

Unit 5: Compilers: Analysis Phase

Faculty Name : Tabassum Maktum

Index -

Lecture 22	Elements of Assembly Language	3
Lecture 23	Data structures used in two pass assembler design	17
Lecture 24	Two pass assembler Design :PASS1	27

Lecture No: 22
Elements of Assembly Language

Role of Assembler

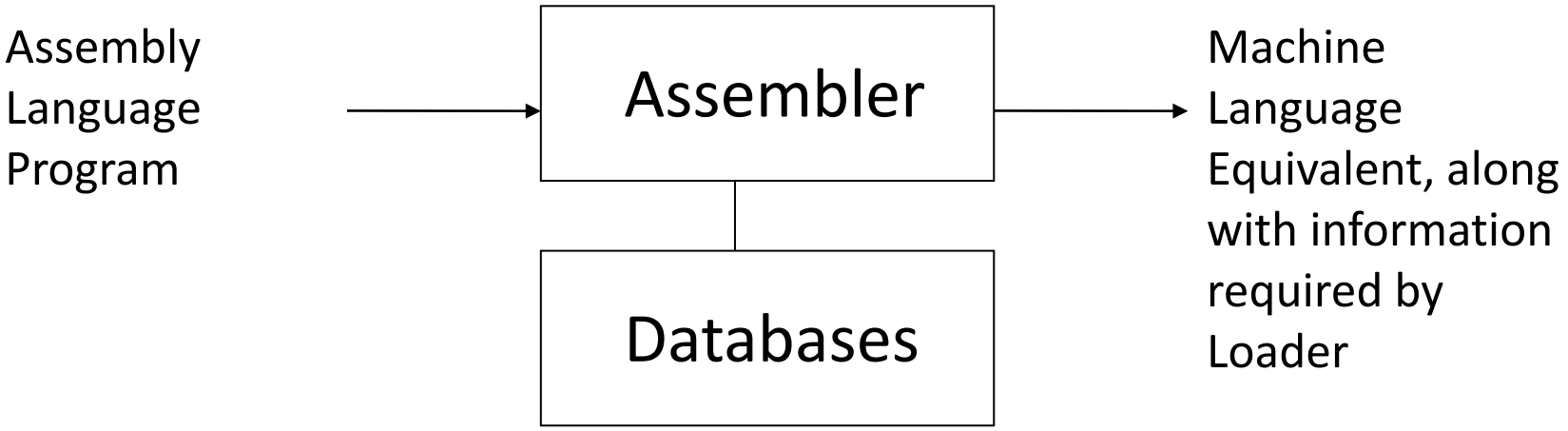


Fig. 2.1: An Assembler

Type:

Absolute / Non Relocatable Program:

- The address at which the programs need to be loaded in the memory for execution is fixed.
- Allocation is done by the programmer and loading will be done by the loader.

Relocatable Program:

- This program can be loaded anywhere in the memory for execution & locating would be done by the loader.

Assembler

Features:

Mnemonic Opcode Specification:

- Instead of writing binary opcode, mnemonics can be specified.
- Advantage of writing mnemonic opcodes are:
  - Program becomes readable..
  - Debugging becomes simple.
- It is the responsibility of assembler to replace each mnemonic opcode by its respective binary opcodes.

Symbolic Operant Specification:

- Instead of writing the address for referencing the instruction or data, symbolic operants can be specified.
- Responsibility of the assembler to replace each symbol by its address.

Features:

Storage:

- With the help of ALP some portion in the memory can be kept aside for storage purpose.

Elements of Assembly language

- Mnemonic operation codes: It is symbolic name given to each machine instruction. It eliminates the need of memorizing the numeric op-codes.
  - Pseudo-op : These are the instructions for the assembler during the assembly process of program.
  - Machine-op: These are actual machine instructions

- The general format of assembly language statement is:

  [label] <op-code> operand (s) ;

continued..

- Symbols: These are the names associated with data or instructions. These names can be used as operand in program.
- Literal: It is an operand which has syntax like =’<value>’.
  - Assembler creates data area for literals containing the constant values.
  - E.g. =F’10’

- Location Counter : Used to hold address of current instruction being executed

Statements in ALP

1. Imperative Statement (IS):

They are simply instructions to be executed by the machine. Assembler translates them into equivalent binary opcode (along with operands, if any) and puts them into an output object file.

e.g. **ADD, SUB, MUL. MOV**

2. Assembler Directive (AD):

These statements direct the assembler to take the action associated with it. They are not a part of executable instructions. So, they are not included in the final object file generated by assembler as output

e.g. **START , STOP, ORIGIN**

3. Declarative Statements (DL)

- These are special cases of assembler directives; they are used to declare symbols and associated them with values

A DB '5' (Define Byte) Reserve byte & initialize 5

B DW '4' (Define Word) Reserve a word & initialize to 4

X DD '7' (Define Double word) Initialize to 7

C DC '3' (Define Constant) assign value

Y DS 10 Define storage of 10 words

Meaning of some pseudo-op

- START: It indicates start of the source program
- END : It indicates end of the source program.
- EQU : It associates symbol with some address specification.

<symbol> EQU <address spec>

- LTORG: It tells assembler to place all literals at earlier place
- DC: Define constant
- DS: Define storage
- ORIGIN: Start storing the current program from specified address

ORIGIN<address spec>

General Design Procedure

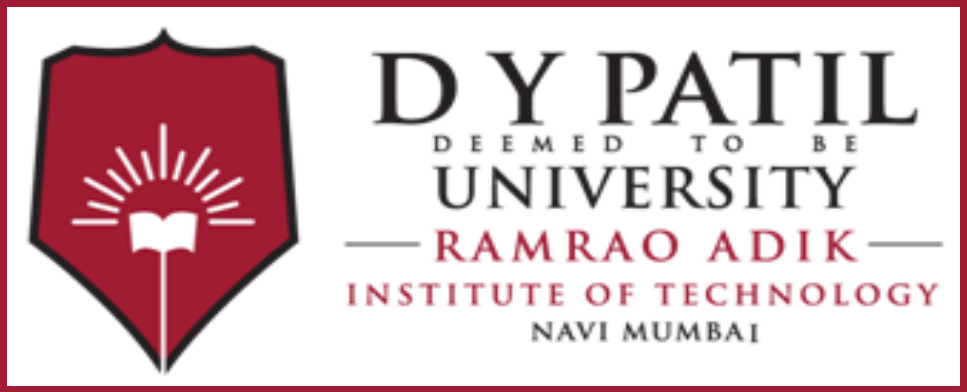
- In general following procedure is used in design of an assembler.
  - Define problem statement
  - Define data structures
  - Define format of data structures
  - Define algorithm
  - Check for modularity
  - Repeat steps from1 to 5 for all modules.

Forward Reference Problem

- In an assembly language program we can use symbols which are the names associated with data or instructions.
- It may happen that the symbols are referred before they are defined. This is called as forward reference.
- One approach to solve this problem is to have two passes over the source program. So the first pass just defines the symbols and second pass finds the addresses.

Two passes

- Pass 1 : It defines symbols and literals
  - Find length of machine instructions
  - Maintain location counter
  - Remember values of symbols till pass2
  - Remember literals
- Pass 2: Generate object program
  - Look up values of symbols
  - Generate instructions
  - Generate data

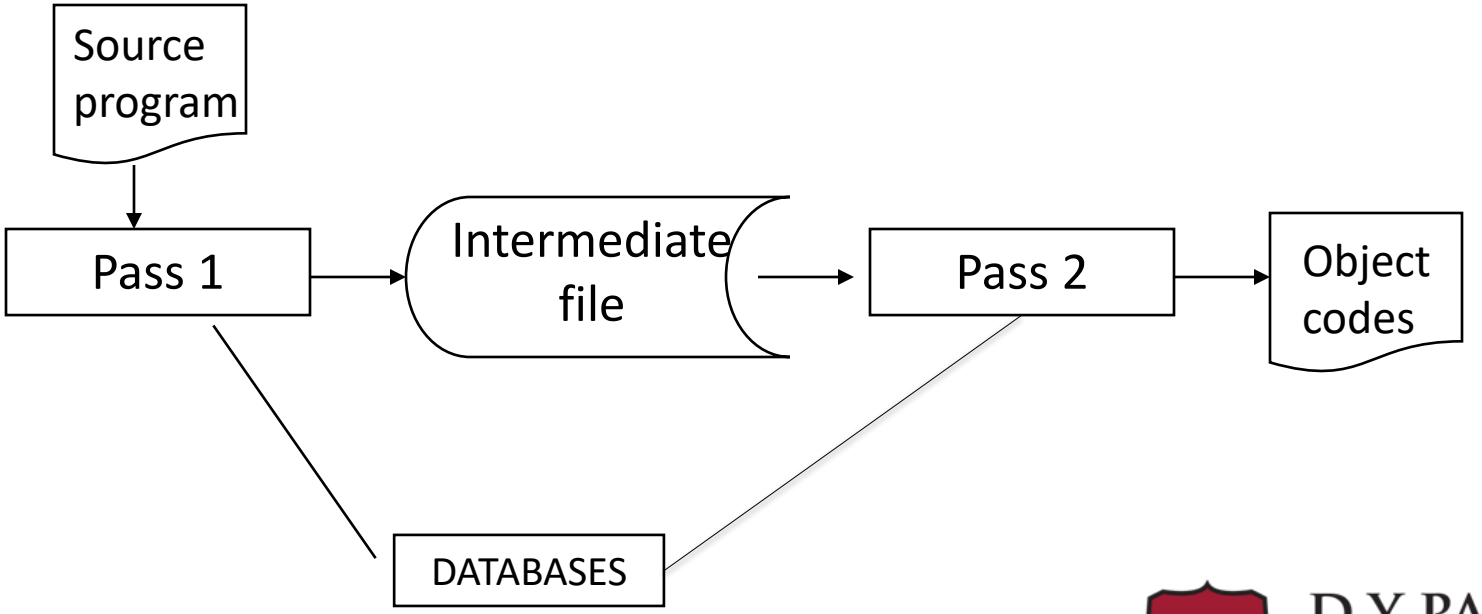

D Y PATIL
DEEMED TO BE
UNIVERSITY
RAMRAO ADIK
INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Thank You

Lecture No: 23
Data structures used in two pass assembler design

Two Pass Assembler

- Read from input line
  - LABEL, OPCODE, OPERAND



Assembler Design

- First pass:
  - Scan the code by separating the symbol, mnemonic op code and operand fields
  - Build the symbol table
  - Perform LC processing
  - Construct intermediate representation
- Second Pass:
  - Solves forward references
  - Converts the code to the machine code

Databases Used in two pass assembler

- Source program
- **Location counter(LC)** which stores location of each instruction
- **Opcode Table (OPT):**This table indicates the symbolic mnemonic for each instructions and its length.
- **Symbol Table (ST)** which stores each label along with its value.
- **Literal Table(LT)** which stores each literal and its corresponding address
- **Pool Table:** Awareness of different literal pools is maintained using auxiliary table Pool Table. This table contains literal number of starting literal of each pool. At any stage current literal pool is the last pool in literal table.

OPTAB – a table of mnemonic op codes

- 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 declaration or a directive statement
- The routine processes the operand field of the statement to determine the amount of memory required and updates LC and the SYMTAB entry of the symbol defined

Sample Opcode Table

mnemonic opcode	*class*	*mnemonic info*
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

OPTAB

Symbol Table

- Include the label name and value (address) for each label in the source program
- Include length information

symbol	*address*	*length*
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

Literal Table and Pool Table

- Each entry in literal table pertains literal
- It contains the fields as:
  - literal
  - Address
- Each Entry in pool table pertains to pool of literals. It has a field of **literal no**. to indicate which literal in literal table contains the first literal in the pool.

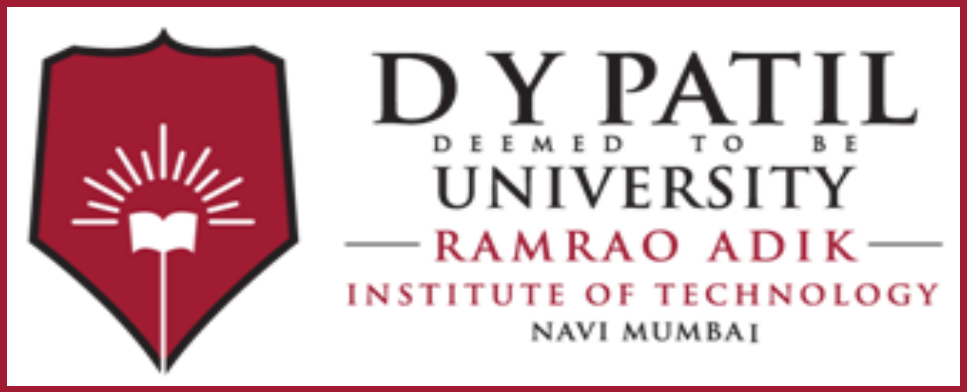
Sample Literal Table and Pool Table

	value	*address*
1	='5'	
2	='1'	
3	='1'	

LITTAB

	first	*# literals*
1	1	2
2	3	1
3	4	0

POOLTAB


D Y PATIL
DEEMED TO BE
UNIVERSITY
RAMRAO ADIK
INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Thank You

Lecture No: 24
Two pass assembler Design :PASS1

Task of Pass1 of Assembler

Pass I
1. Separate the symbol, mnemonic opcode and operand fields.
2. Build the symbol table.
3. Perform LC processing.
4. Construct intermediate representation.

Pass I 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
POOLTAB	A table of information concerning literal pools

Handling Labels

- When a label definition is encountered in the program, the assembler searches for it in Symbol Table.
- If an existing label already exists, then it is a case of duplicate label definition. So the error flag is turned on and “Duplicate Label” error is displayed.
- If the label is not found in ST, then it entry is made into Symbol Table with name , value and length information.

Handling Imperative Statements

- If imperative statement is processed then search opcode table (OP Table) to find length of corresponding opcode.
- If found, only LC is incremented by length-of-instruction (LOI).
- If not found, error flag is set ON and “Instruction not found” error can be displayed

Handling Operands

- If Operand is literal then , it is searched in literal table (LT). If found, it indicates that it has already been added.
- If not found, its entry is made in LT
- If operand is a symbol, it is searched in Symbol Table (ST). If found, it indicates that it has already been defined.
- If not found, its entry is made in ST.

Handling some Assembler Directives

- **START & ORIGIN :** The location Counter (LC) is initialized with address specified in the operand.
- **DC :** The symbol in label is searched in symbol table (ST) . If already present then its value is set using current value of location counter.

  If not present then the entry with symbol name and current value of location counter is made in ST.

  LC is updated by the length of instruction.
- **DS :** The symbol in label is searched in symbol table (ST) . If already present then its value is set using current value of location counter.

  If not present then the entry with symbol name and current value of location counter is made in ST.

  LC is updated by the length of storage occupied

Handling some Assembler Directives

- **LTORG:** Assign addresses to all literals present in literal table.
  - Current value of location counter is assigned to first literal present in literal table. Then address is incremented by length each time and addresses are assigned to all remaining literals.
  - After assigning addresses to all literal, the LC is set to the next address.
  - The entry in POOL table is also updated

- **EQU:** **T**he entry with symbol name in label field is done in symbol table. The value is set as value specified in operand field.

  LC is not updated.

- **END:** The addresses are assigned to remaining unaddressed literals.

Example 1

Line No.	Label	OPCODE	OP1	OP2
1		START	400	
2		MOVER	AREG	=‘5’
3		MOVER	BREG	A
4	X	MOVER	CREG	=‘3’
5		ORIGIN	X+3	
6		LTORG		
7	A	DS	3	
8		ADD	AREG	B
9	B	DC	‘4’	
10	C	DC	‘6’	
11	Y	EQU	‘7’	
12		ADD	CREG	Y
13		END		

Example 1: LC Processing

Line No.	Label	OPCODE	OP1	OP2	Location counter
1		START	400		400
2		MOVER	AREG	=‘5’	400
3		MOVER	BREG	A	401
4	X	MOVER	CREG	=‘3’	402
5		ORIGIN	X+3		403
6		LTORG			405
7	A	DS	3		407
8		ADD	AREG	B	410
9	B	DC	‘4’		411
10	C	DC	‘6’		412
11	Y	EQU	‘7’		413
12		ADD	CREG	Y	414
13		END			414

Example 1: SYMBOL TABLE: ST

Sr. No.	Name	Value	Length
0	A	407	1
1	X	402	1
2	B	412	1
3	C	413	1
4	Y	7	1

Example 1: LITERAL TABLE: LT & POOL Table

LITERAL TABLE : LT

SR. NO.	LITERAL	ADDRESS
0	=‘5’	405
1	=‘3’	406

POOL TABLE

FIRST	#LITERALS
0	2

Intermediate Code

Intermediate code for opcode is generated as a pair (instruction class , code)

Instruction Type	Class
Imperative Statement	IS
Declaration Statement	DL
Assembler Directive	AD

Intermediate Code

- **Intermediate code for operand is generated as a pair (operand class , code)**

Operand	Class
Constant	C
Symbol	S
Literal	L
Register	RG

- **For literal and symbol the code field consist of entry number of the operand from LT or ST**
- **For Registers the code field is set as:**

Register	Code
AREG	01
BREG	02
CREG	03

D Y PATIL DEEMED TO BE UNIVERSITY — RAMRAO ADIK — INSTITUTE OF TECHNOLOGY NAVI MUMBAI

Binary code for Instructions

00	STOP	Stop execution
01	ADD	Perform addition
02	SUB	Perform subtraction
03	MULT	Perform multiplication
04	MOVER	Move from memory to register
05	MOVEM	Move from register to memory
06	COMP	Compare and set condition code
07	BC	Branch on condition
08	DIV	Perform division
09	READ	Read into register
10	PRINT	Print contents of register

Binary code for Instructions

Declaration statements	
DC	01
DS	02

Assembler directives	
START	01
END	02
ORIGIN	03
EQU	04
LTORG	05

Example 1: Intermediate Code

Line No.	Label	OPCODE	OP1	OP2	Location counter	Intermediate Code
1		START	400		400	(AD, 01) (C,400)
2		MOVER	AREG	='5'	400	(IS,04) (RG,01) (L,0)
3		MOVER	BREG	A	401	(IS,04) (RG,02) (S,0)
4	X	MOVER	CREG	='3'	402	(S,1) (IS,04) (RG,03) (L,1)
5		ORIGIN	X+3		403	(AD,03) (C,405)
6		LTORG			405	**(AD,05) (DL,02)(C,5)** **(AD,05) (DL,02) (C,3)**
7	A	DS	3		407	(S,0) (DL,01) (C,3)
8		ADD	AREG	B	410	(IS,01) (RG,01) (S,2)
9	B	DC	'4'		411	(S,2) (DL,02) (C,4)
10	C	DC	'6'		412	(S,3) (DL,02) (C,6)
11	Y	EQU	'7'		413	(S,4) (AD,04) (C,7)
12		ADD	CREG	Y	414	(IS,01) (RG,03) (S,4)
13		END			414	(AD,02)

Algorithm: Pass 1

Initialize LC =0 and set literal table and POOL table pointers.

1. LC := 0; (This is the default value)
   littab_ptr := 1;
   pooltab_ptr := 1;
   POOLTAB [1]. *first* := 1; POOLTAB [1]. *# literals* := 0;

Algorithm: Pass 1

2. While the next statement is not an END statement
   (a) If a symbol is present in the label field then
   this_label := symbol in the label field;
   Make an entry (*this_label*, <LC>, –) in SYMTAB.
   (b) If an LTORG statement then
   (i) If POOLTAB [*pooltab_ptr*]. *# literals* > 0 then
   Process the entries LITTAB [POOLTAB [*pooltab_ptr*]. *first*] ... LITTAB [*littab_ptr*−1] to allocate memory to the literal, put address of the allocated memory area in the *address* field of the LITTAB entry, and update the address contained in location counter accordingly.
   (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
   (iii) POOLTAB [*pooltab_ptr*]. *first* := *littab_ptr*;
   POOLTAB [*pooltab_ptr*]. *# literals* := 0;

Algorithm: Pass 1

(c) If a START or ORIGIN statement then
LC := value specified in operand field;

(d) If an EQU statement then

(i) *this_addr* := value of <*address specification*>;

(ii) Correct the SYMTAB entry for *this_label* to (*this_label*, *this_addr*, 1).

(e) If a declaration statement then

(i) Invoke the routine whose id is mentioned in the *mnemonic info* field. This routine returns *code* and *size*.

(ii) If a symbol is present in the label field, correct the symtab entry for *this_label* to (*this_label*, <LC>, *size*).

(iii) LC := LC + *size*;

(iv) Generate intermediate code for the declaration statement.

Algorithm: Pass 1

(f) If an imperative statement then

(i) *code* := machine opcode from the *mnemonic info* field of OPTAB;

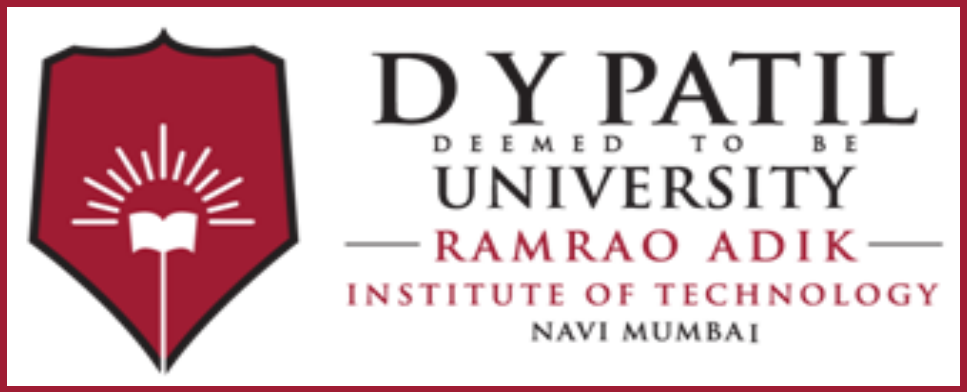
(ii) LC := LC + instruction length from the *mnemonic info* field of OPTAB;

(iii) If operand is a literal then

this_literal := literal in operand field;
if POOLTAB [*pooltab_ptr*]. *# literals* = 0 or *this_literal* does not match any literal in the range LITTAB [POOLTAB [*pooltab_ptr*]] .*first* ... LITTAB [*littab_ptr*−1] then
LITTAB [*littab_ptr*]. *value* := *this_literal*;
POOLTAB [*pooltab_ptr*]. *# literals* :=
POOLTAB [*pooltab_ptr*]. *# literals* +1;
littab_ptr := *littab_ptr* + 1;
else (i.e., operand is a symbol)
this_entry := SYMTAB entry number of operand;
Generate intermediate code for the imperative statement.

Algorithm: Pass 1

3. (Processing of the END statement)
   (a) Perform actions (i)–(iii) of Step 2(b).
   (b) Generate intermediate code for the END statement.


D Y PATIL
DEEMED TO BE
UNIVERSITY
RAMRAO ADIK
INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Thank You