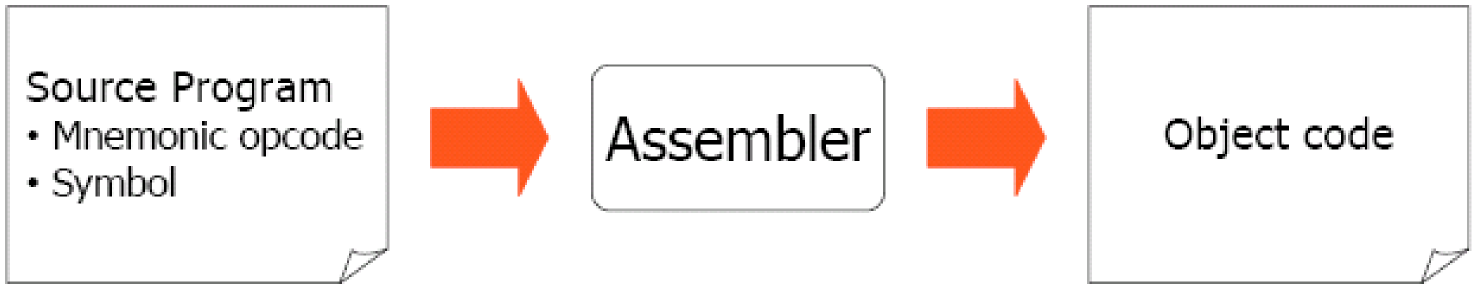
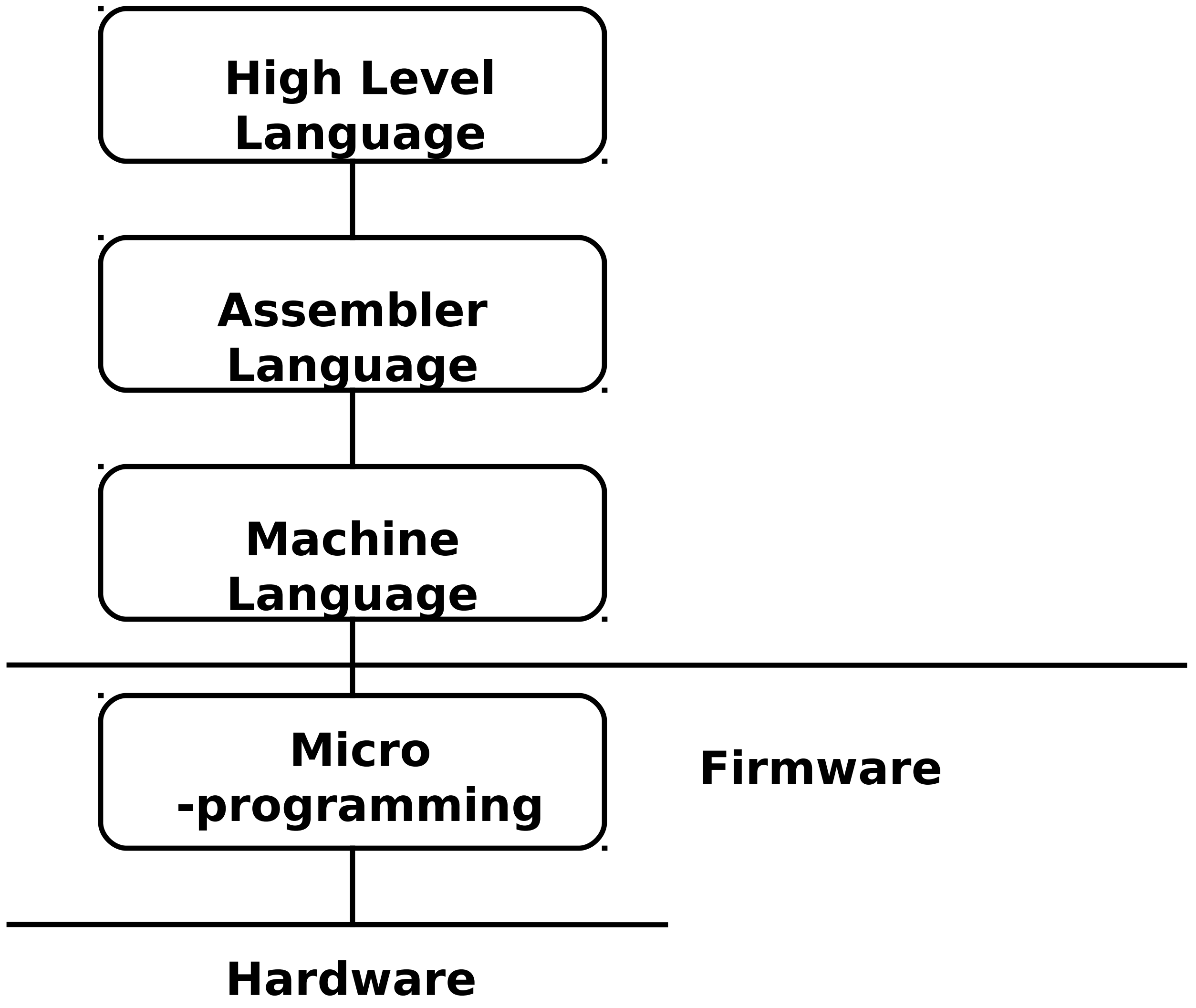
Assembler: Definition

- Translating source code written in assembly language to object code.



Language Levels



Machine code

- Machine code:
  - Set of commands directly executable via CPU
  - Commands in numeric code
  - Lowest semantic level

Machine code language

- Structure:

OpCode	**OpAddress**

  - Operation code
    - Defining executable operation
  - Operand address
    - Specification of operands
      - Constants/register addresses/storage addresses

Elements of the Assembly Language Programming

- An Assembly language is a
  - machine dependent,
  - low level Programming language specific to a certain computer system.

Three features when compared with machine language are

1. Mnemonic Operation Codes
2. Symbolic operands
3. Data declarations

Elements of the Assembly Language Programming

Mnemonic operation codes: eliminates the need to memorize numeric operation codes.

Symbolic operands: Symbolic names can be associated with data or instructions. Symbolic names can be used as operands in assembly statements (need not know details of memory bindings).

Data declarations: Data can be declared in a variety of notations, including the decimal notation (avoids conversion of constants into their internal representation).

Assembly language-structure

<Label>	**<Mnemomi**	**<Operand>**	**Comments**

- Label
  - symbolic labeling of an assembler address (command address at Machine level)
- Mnemomic
  - Symbolic description of an operation
- Operands
  - Contains of variables or addresse if necessary
- Comments

Statement format

An Assembly language statement has following format:

[Label] <*opcode*> <*operand spec*>[,<*operand spec*>..]

If a label is specified in a statement, it is associated as a symbolic name with the memory word generated for the statement.

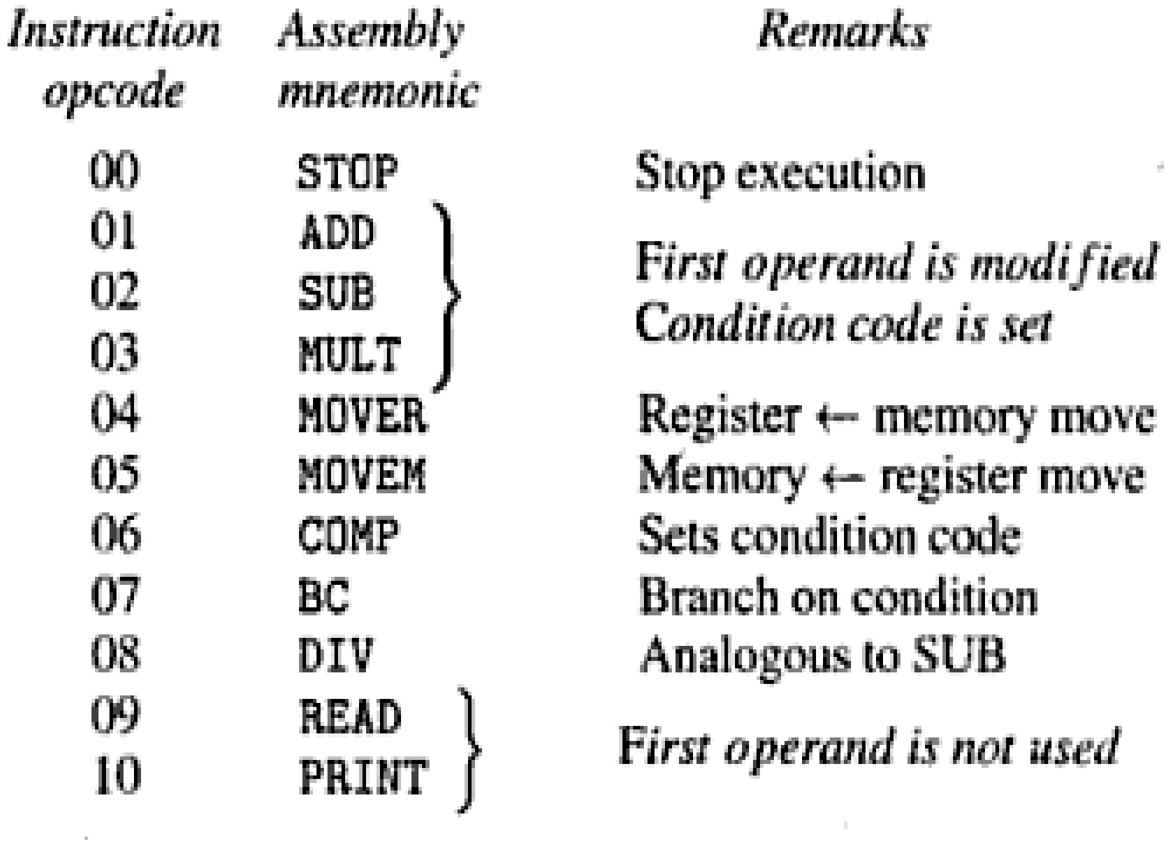
<operand spec> has the following syntax:

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

Eg. AREA, AREA+5, AREA(4), AREA+5(4)

Mnemonic Operation Codes

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



Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	*First operand is modified* *Condition code is set*
02	SUB	
03	MULT	
04	MOVER	Register ← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	*First operand is not used*
10	PRINT	

Operation Codes

- *MOVE* instructions move a value between a memory word and a register
- *MOVER* – First operand is target and second operand is source
- *MOVEM* – first operand is source, second is target

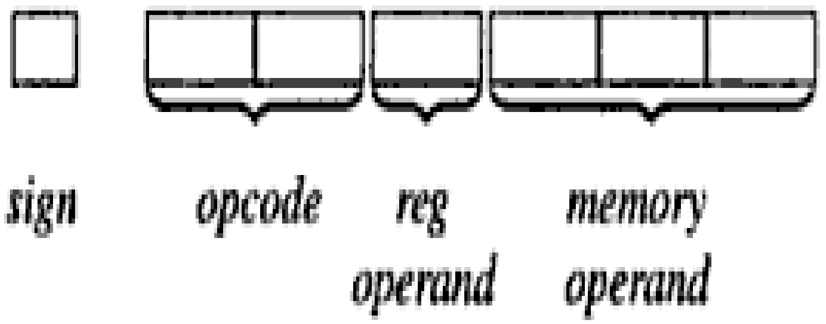
- All arithmetic is performed in a register (replaces the contents of a register) and sets *condition code.*

- A Comparision instruction sets *condition code* analogous to arithmetics, i.e. without affecting values of operands.

- *condition code* can be tested by a Branch on Condition (BC) instruction and the format is:

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

Machine Instruction Format



- sign is not a part of the instruction
- Opcode: 2 digits, Register Operand: 1 digit, Memory Operand: 3 digits
- Condition code specified in a BC statement is encoded into the first operand using the codes 1- 6 for specifications LT, LE, EQ, GT, GE and ANY respectively
- In a Machine Language Program, all addresses and constants are shown in decimal as shown in the next slide

Assembly Language Statements

- An assembly program contains three kinds of statements:

1) Imperative Statements
2) Declaration Statements
3) Assembler Directives

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

Assembly Language Statements

- **Declaration Statements:** syntax is as follows:

```
[Label]  DS    <constant>
[Label]  DC    '<value>'
```

- The DS (declare storage) statement reserves memory and associates names with them.
- Ex:

```
A  DS     1  ; reserves a memory area of 1 word, associating the name A to it
G  DS   200 ; reserves a block of 200 words and the name G is associated with the
              first word of the block (G+6 etc. to access the other words)
```

- The DC (declare constant) statement constructs memory words containing constants.
- Ex:

```
ONE    DC      '1’ ; associates name one with a memory word containing value 1
```

Assembly Language Statements

Use of Constants

- The DC statement does not really implement constants
- it just initializes memory words to given values.
- The values are not protected by the assembler and can be changed by moving a new value into the memory word.
- In the above example, the value of ONE can be changed by executing an instruction

```
MOVEM    BREG,   ONE
```

Assembly Language Statements

Use of Constants

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

- 1) Immediate operands can be used in an assembly statement only if the architecture of the target machine includes the necessary features.
  - Ex: ADD AREG,5
  - This is translated into an instruction from two operands – AREG and the value '5' as an immediate operand

Assembly Language Statements

Use of Constants

- 2) A *literal* is an operand with the syntax = '<value>'.
- It differs from a constant because its location cannot be specified in the assembly program.
- Its value does not change during the execution of the program.
- It differs from an immediate operand because no architectural provision is needed to support its use.

```
ADD   AREG, ='5'  ➔        ADD   AREG, FIVE
                           FIVE  DC     '5'
```

Use of literals vs. Use of DC

Assembly Language Statements

Assembler Directive

•Assembler directives instruct the assembler to perform certain actions during the assembly of a program.

•Some assembler directives are described in the following:

1) START *<constant>*

•This directive indicates that the first word of the target program generated by the assembler should be placed in the memory word having address *<constant>*.

2) END [<operand spec>]

•This directive indicates the end of the of the source program. The optional <operand spec> indicates the address of the instruction where the execution of the program should begin.

Advantages of Assembly Language

- The primary advantages of assembly language programming over machine language programming are due to the use of symbolic operand specifications.
  (in comparison to machine language program)

- Assembly language programming holds an edge over HLL programming in situations where it is desirable to use architectural features of a computer.
  (in comparison to high level language program)

Fundamentals of LP

- Language processing = analysis of source program + synthesis of target program
- Analysis of source program is specification of the source program
  - Lexical rules: formation of valid lexical units(tokens) in the source language
  - Syntax rules : formation of valid statements in the source language
  - Semantic rules: associate meaning with valid statements of the language

Fundamentals of LP

- Synthesis of target program is construction of target language statements
  - Memory allocation : generation of data structures in the target program
  - Code generation

A simple Assembly Scheme

- There are two phases in specifying an assembler:

1. Analysis Phase
2. Synthesis Phase(the fundamental information requirements will arise in this phase)

A simple Assembly Scheme

Design Specification of an assembler

There are four steps involved to design the specification of an assembler:

- Identify information necessary to perform a task.
- Design a suitable data structure to record info.
- Determine processing necessary to obtain and maintain the info.
- Determine processing necessary to perform the task

Synthesis Phase: Example

Consider the following statement:

MOVER BREG, ONE

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

1. Address of the memory word with which name **ONE** is associated [depends on the source program, hence made available by the Analysis phase].

2. Machine operation code corresponding to **MOVER** [does not depend on the source program but depends on the assembly language, hence synthesis phase can determine this information for itself]

Note: Based on above discussion, the two data structures required during the synthesis phase are described next

Data structures in synthesis phase

Symbol Table *--built by the analysis phase*

- The two primary fields are name and address of the symbol used to specify a value.

Mnemonics Table *--already present*

- The two primary fields are *mnemonic* and *opcode,* along with *length.*

Synthesis phase uses these tables to obtain

- The machine address with which a name is associated.
- The machine op code corresponding to a mnemonic.

•The tables have to be searched with the

- Symbol name and the mnemonic as keys

Analysis Phase

- Primary function of the Analysis phase is to build the symbol table.
  - It must determine the addresses with which the symbolic names used in a program are associated
  - It is possible to determine some addresses directly like the address of first instruction in the program (ie.,start)
  - Other addresses must be inferred
  - To determine the addresses of the symbolic names we need to fix the addresses of all program elements preceding it through *Memory Allocation*.
- To implement *memory allocation* a data structure called *location counter* is introduced.

Analysis Phase – Implementing memory allocation

- LC(location counter) :
  - is always made to contain the address of the next memory word in the target program.
  - It is initialized to the constant specified at the START statement.
- When a LABEL is encountered,
  - it enters the LABEL and the contents of LC in a new entry of the symbol table.

    LABEL – e.g. N, AGAIN, SUM etc
  - It then finds the number of memory words required by the assembly statement and updates the LC contents
- To update the contents of the LC, analysis phase needs to know lengths of the different instructions
  - This information is available in the Mnemonics table and is extended with a field called length
- We refer the processing involved in maintaining the LC as LC Processing

Example

START 100

MOVER BREG, N LC = 100 (1 byte)

MULT BREG, N LC = 101 (1 byte)

STOP LC = 102 (1 byte)

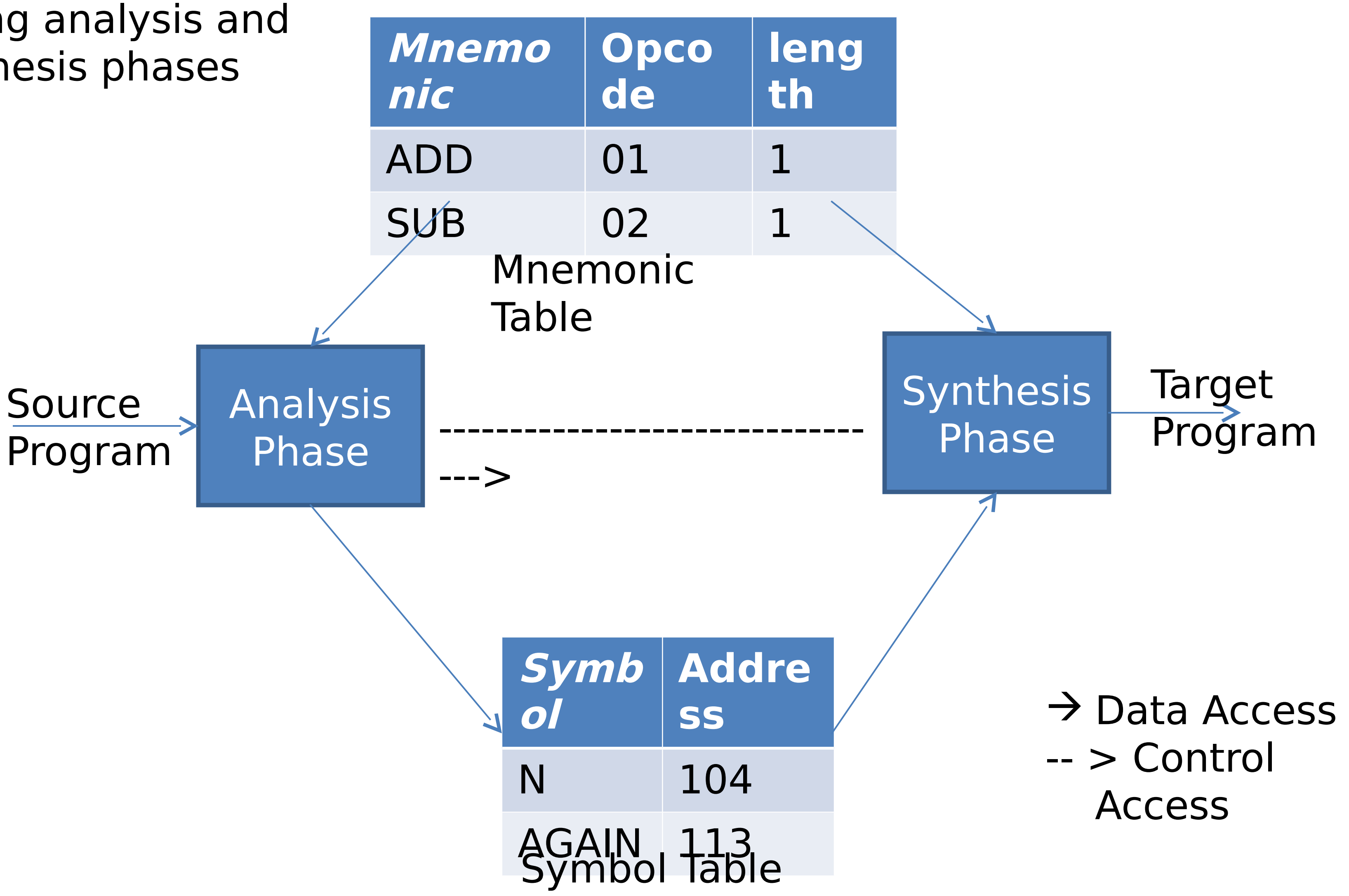
N DS 5 LC = 103

Symbol	Address
N	103

- Since there the instructions take different amount of memory, it is also stored in the mnemonic table in the “length” field

Mnemonic	Opcode	Length
MOVER	04	1
MULT	03	1

Data structures of an assembler During analysis and Synthesis phases



Mnemonic	Opcode	length
ADD	01	1
SUB	02	1

Mnemonic Table

Symbol	Address
N	104
AGAIN	113

Symbol Table

→ Data Access
-- > Control Access

Data structures

- Mnemonics table is a fixed table which is merely accessed by the analysis and synthesis phases
- Symbol table is constructed during analysis and used during synthesis

Tasks Performed : Analysis Phase

- Isolate the labels, mnemonic, opcode and operand fields of a statement.
- If a label is present, enter (symbol, <LC>) into the symbol table.
- Check validity of the mnemonic opcode using mnemonics table.
- Update value of LC.

Tasks Performed : Synthesis Phase

- Obtain machine opcode corresponding to the mnemonic from the mnemonic table.
- obtain address of the memory operand from symbol table.
- Synthesize a machine instruction or machine form of a constant, depending on the instruction.

Assembler’s functions

- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Build the machine instructions in the proper format
- Convert the data constants to internal machine representations
- Write the object program and the assembly listing

Assembler:Design

- The design of assembler can be of:
  - Scanning (tokenizing)
  - Parsing (validating the instructions)
  - Creating the symbol table
  - Resolving the forward references
  - Converting into the machine language

Assembler Design

- Pass of a language processor – one complete scan of the source program
- Assembler Design can be done in:
  - Single pass
  - Two pass
- Single Pass Assembler:
  - Does everything in single pass
  - Cannot resolve the forward referencing
- Two pass assembler:
  - Does the work in two pass
  - Resolves the forward references

Difficulties: Forward Reference

- Forward reference: reference to a label that is defined later in the program.

```
Loc  Label  Operator      Operand      _

1000        FIRST   STL     RETADR
1003        CLOOP  JSUB    RDREC
 …     …    …       …       …
1012        J       CLOOP

…      …    …       …       …
1033        RETADR          RESW   1
```

Backpatching

- The problem of forward references is handled using a process called backpatching
  - Initially, the operand field of an instruction containing a forward reference is left blank
  - Ex: MOVER BREG, ONE can be only partially synthesized since ONE is a forward reference
  - The instruction opcode and address of BREG will be assembled to reside in location 101
  - To insert the second operand's address later, an entry is added as Table of Incomplete Instructions (TII)
  - The entry TII is a pair (<instruction address>, <symbol>) which is (101, ONE) here

Backpatching

- The problem of forward references is handled using a process called backpatching
  - When END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program
  - So TII would contain information of all forward references
  - Now each entry in TII is processed to complete the instruction
  - Ex: the entry (101, ONE) would be processed by obtaining the address of ONE from symbol table and inserting it in the operand field of the instruction with assembled address 101.
  - Alternatively, when definition of some symbol L is encountered, all forward references to L can be processed

Assembler Design

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

Assembler Design

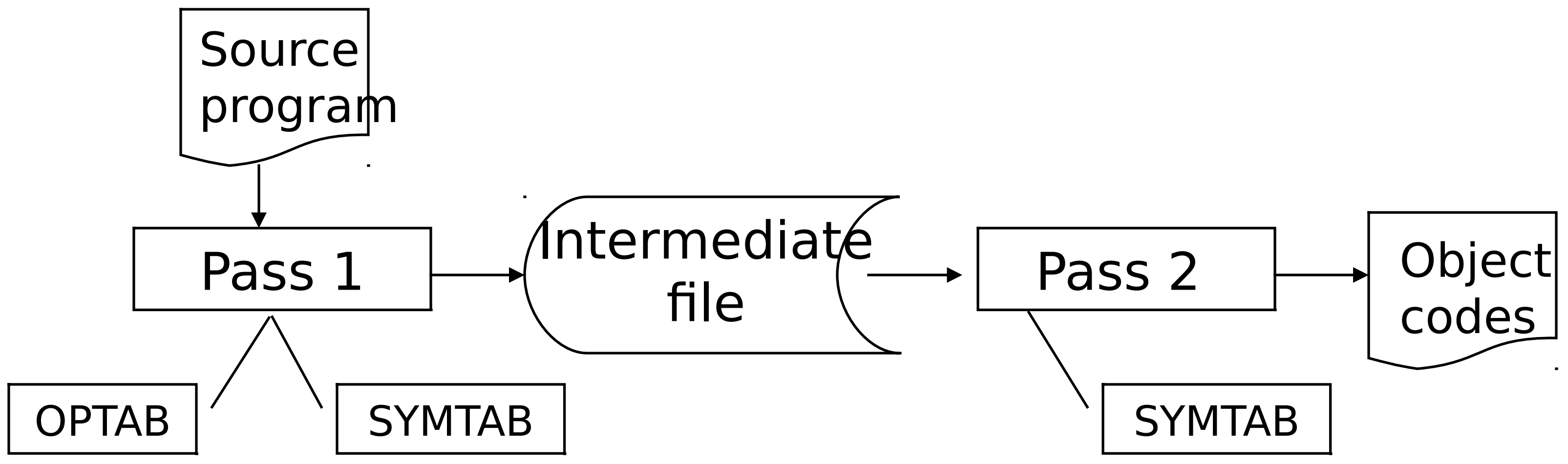
- Assembler directives are pseudo instructions.
  - They provide instructions to the assemblers itself.
  - They are not translated into machine operation codes.

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

Two Pass Assembler

- Read from input line
  - LABEL, OPCODE, OPERAND



Data Structures in Pass I

- 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

Data Structures in Pass I

- SYMTAB - Symbol Table
  - Contains address and length
- LOCCTR - Location Counter
- LITTAB – a table of literals used in the program
  - Contains literal and address
  - Literals are allocated addresses starting with the current value in LC and LC is incremented, appropriately