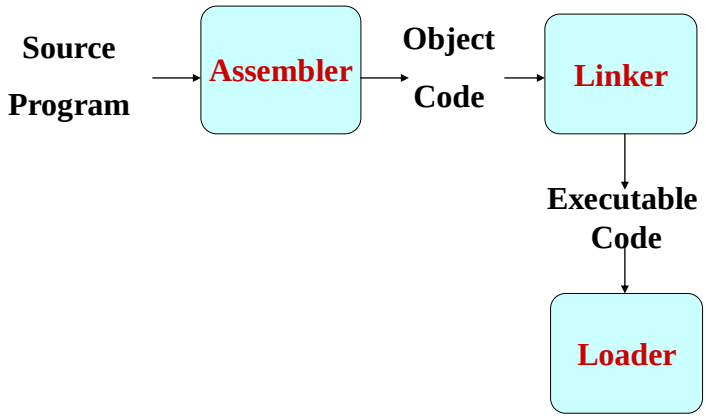
Role of Assembler



Functions of a Basic Assembler

- Translating mnemonic operation codes to their machine language equivalents
- Assigning machine addresses to symbolic labels
- Convert the data constants to internal machine representations
- Write the object program and the assembly listing

Assembly Language Statements

An assembly program contains three kinds of statements:

1. **Imperative statements**- indicates an action to be performed during the execution of the assembled program. Each imperative statement typically translates into one machine instruction.
2. **Declaration statements**- the syntax of declaration statements is :

   [Label] DS <constant>
   [Label] DC <value>
   The DS (short) for declare storage)statement reserves areas of memory and associates names with them.e.g.
   A DS 1
   The above statement reserves a memory area of 1 word and associates the name A with it.
   The DC (short for declare constant) statement declare memory words containing constants.

3. Assembler directives

- Assembler directives are Pseudo-Instructions
  - They provide instructions to the assembler itself
  - They are not translated into machine operation codes
- Basic assembler directives
  - START: specify name & starting address
  - END : end of source program, specify the first execution instruction

An example of assembly program

```
            START   101
            READ    N                   101)
            MOVER   BREG, ONE           102)
            MOVEM   BREG, TERM          103)
AGAIN       MULT    BREG, TERM                  104)
            MOVER   CREG, TERM          105)
            ADD     CREG, ONE           106)
            COMP    CREG, N             107)
            MOVEM   BREG, RESULT    108)
N           DS      1               109)
RESULT      DS      1               110)
ONE         DC      '1'             111)
TERM        DS      1                   112)
            END
```

General Design Procedure/ Specification of an Assembler

A four step approach is used to develop a design specification for an assembler:

1. Determine the processing necessary to perform the task.
2. Determine the processing necessary to obtain and maintain the information.
3. Design a suitable data structure to record the information.
4. Identify the information necessary to perform a task.

1. **Identify the information necessary to perform a task.**

The fundamental information requirements arise in the synthesis phase of an assembler. Hence it is best to begin by considering the information requirements of the synthesis tasks. The information is collected during analysis or derived during synthesis phase. For e.g. Consider the assembly statement ;

MOVER BREG , ONE

Following information is required to synthesize the machine instruction corresponding to this statement:

i) **Address of the memory word with which name ONE is associated.**
ii) **Machine operation code corresponding to the mnemonic MOVER.**

The first item of information depends on the source program. Hence it is made available by analysis phase. The second item of information does not depend on the source program , it merely depends on the assembly language. Hence the synthesis phase can determinethis information for itself.

2. **Design a suitable data structure to record the information.**

Three data structures are used :

- **SYMTAB**: symbol table
- **OPTAB**: operation code table/ mnemonics table
- **LOCCTR**: location counter

SYMTAB

- Fields : symbol name, address
- It is a dynamic table built by analysis phase
- To indicate error conditions (Ex: multiple define)
- Insert, delete and search allowed
- Usually use a hash table

OPTABLE

- Fields: Mnemonic, opcode, length
- It is a static table
- Array or hash table
- Usually use a hash table (mnemonic opcode as key)

LOCCTR

- Initialize to be the beginning address specified in the "START" statement
- LOCCTR =LOCCTR + (instruction length)
- The current value of LOCCTR gives the address to the label/symbol encountered

3. Determine the processing necessary to obtain and maintain the information.

For building symbol table during analysis phase, it is necessary to determine the addresses with which the symbol names used in a program is associated. It is possible to determine some addresses directly e.g. the address of the first instruction in the program, however others must be inferred. To determine the address of an instruction, it is required to fix the addresses of all instructions preceding it. This function is called memory allocation.

To implement memory allocation , the data structure **location counter** is used. The 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 in the START statement. Whenever the analysis phase sees a label in an assembly statement, it enters the label and the contents of LC in a new entry of the symbol table. It then finds the number of memory words required by the assembly statement and updates the LC contents. The information about the length of different instructions is included in the opcode/ mnemonics table.

The processing involved in maintaining the location counter is referred as **LC processing.**

4. Determine the processing necessary to perform the task.

The tasks performed by the analysis and synthesis phases are as follows:

Analysis phase

1. Isolate the label, mnemonic opcode and operand fields of a statement.
2. If a label is present, enter the pair 9 symbol, <LC contents>) in a new entry of Symbol table.
3. Check validity of the mnemonic opcode through a look-up in the Opcode table.
4. Perform LC processing i.e. update the value contained in LC by considering the opcode and operands of the statements.

Synthesis phase

1. Obtain the machine opcode corresponding to the mnemonic from the Opcode table.
2. Obtain address of a memory operand from the symbol table.
3. Synthesize a machine instruction or the machine form of a constant as the case may be.

Assembler Design Options

1. **One-pass assemblers**
2. **Two-pass assemblers**

One-pass Assemblers

- Main Problem
  - » forward reference
    - – data items
    - – labels on instructions

One pass translation scheme

LC processing and construction of the symbol table proceeds with analysis phase. The problem of forward references is tackled 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. The instruction corresponding to the statement

MOVER BREG , ONE

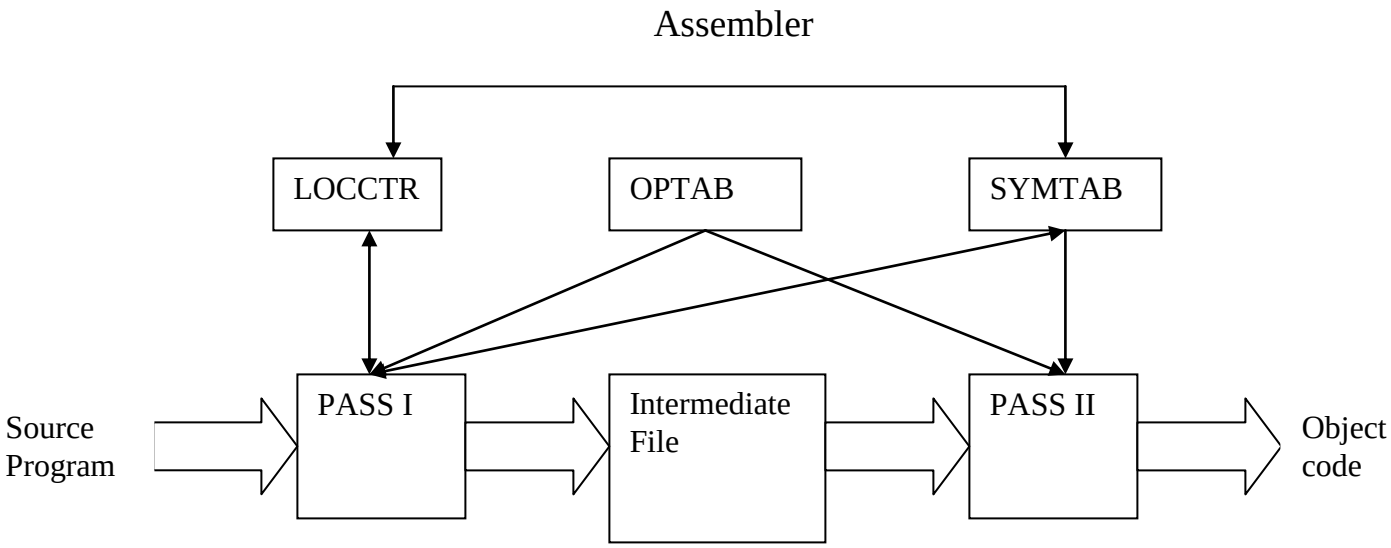
can be only partially synthesize since ONE is a forward reference. Hence the instruction opcode and address of BREG will be assembled to reside in location 101. The need for inserting the second operand's address at a later stage can be indicated by adding an entry to the **Table of Incomplete Instructions (TII)**. This entry is a pair (<instruction address>, <symbol>), e.g. (101. ONE) in this case.

By the time END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program and TII would contain information describing all forward references. The assembler can now process each entry in TII to complete the concerned instruction.

Two Pass Assemblers

Two pass translation scheme

Two pass translation of an assembly language program can handle forward references easily. LC processing is performed in the first pass and symbols 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 the analysis of the source program while the second pass performs synthesis of target program. The first pass constructs an intermediate representation (IR) of the source program for use by the second pass.



Tasks performed by different passes:

- **PASS 1: loop until the end of the program**
    1. read in a line of assembly code
    2. assign an address to this line
        - Update LC
    3. save address values assigned to labels
        - in symbol tables
    4. process assembler directives
        - constant declaration
        - space reservation
- **PASS 2: same loop**
    1. read in a line of code
    2. translate op code using op code table
    3. change labels to address using the symbol table
    4. process assembler directives
    5. produce object program

Assembler Algorithm: pass1

```
begin
    if starting address is given
        LOCCTR = starting address;
    else
        LOCCTR = 0;
    while OPCODE != END do          ;; or EOF
        begin
```

```
        read a line from the code
        if there is a label
            if this label is in SYMTAB, then error
            else insert (label, LOCCTR) into SYMTAB
        search OPTAB for the op code
        if found
            LOCCTR += N     ;; N is the length of this instruction (4 for MIPS)
        else if this is an assembly directive
            update LOCCTR as directed
        else error
        write line to intermediate file
        end
    program size =  LOCCTR - starting address;
end
```

Assembler Algorithm: pass2

```
begin
    read a line;
    if op code = START then;; .globl xxx for MIPS
        write header record;
    while op code != END do         ;; or EOF
        begin
        search OPTAB for the op code;
        if found
            if the operand is a symbol then
                replace it with an address using SYMTAB;
             assemble the object code;
        else if is a defined directive
            convert it to object code;
        add object code to the text;
        read next line;
        end
    write End record to the text;
    output text;
end
```

Two types of one-pass assembler

- load-and-go
  - produces object code directly in memory for immediate execution
- the other
  - produces usual kind of object code for later execution

Load-and-go Assembler

- Characteristics
  - Useful for program development and testing
  - Avoids the overhead of writing the object program out and reading it back
  - Both one-pass and two-pass assemblers can be designed as load-and-go.
  - However one-pass also avoids the over head of an additional pass over the source program
  - For a load-and-go assembler, the actual address must be known at assembly time, we can use an absolute program