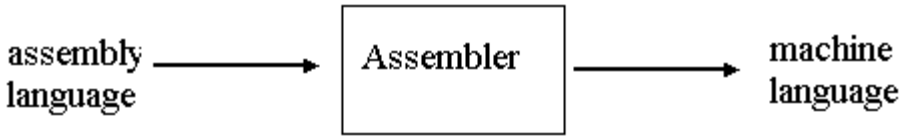
Module 1: Assemblers

Assembler:

Assembler is a language processor which converts assembly level language to machine level language.



An assembly level statement has the following format:

[label]<opcode><operand spec> [<operand spec>…]

opcode

-called mnemonic operation codes. They specify the operation.

eg: STOP stop execution

ADD, SUB, MULT — Arithmetic operation

MOVER register ← memory

MOVEM memory → register.

COMP sets condition codes

BC Branch on condition

READ, PRINT — Reading and printing.

Operand specification:-

Syntax

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

a) AREA

b) AREA+5

c) AREA (4)

d) AREA+5(4)

Assembly Statements

An assembly program consists of three kinds of statements.

1. Imperative statements:-specifies an operation to be performed.
2. Declarative:-Syntax

[label]DS<constant>
[label]DC<value>

DS is declared storage reserves areas of memory and associates name with them.

AREA DS 1

This statement reserves a memory area of 1 word and associates name AREA with it.

DC is declare constant-constructs memory word containing constants.

3. Assembler directives: - These are the instruction to the assembler and not to the machine. These are some times called pseudo operations.

a) START
b) END
c) ORIGIN
d) RESB
e) RESW

Forward reference

The reference to an entity that precedes its definition in the program is called forward reference. An example is:

:
:
:
CALL JUMP
:
:
:
JUMP: ---
:
:

Language processor pass

It is the processing of every statement in a source program or its equivalent representation to perform a language processing function. This is also used during a set of language processing functions.

Literals

A literal is an operand with the syntax ='<value>'.It differs from a constant because its location cannot be specified in the assembly language program. This helps to ensure that its value is not changed during the execution of a program.

Eg: ADD AREG, '=5'

FIVE DC '=5'

Design specification of an assembler

The assembly process is divided into two phases- ANALYSIS, SYNTHESIS.
The primary function of the analysis phase is building the symbol table. For this, it uses the addresses of symbolic names in the program (memory allocation). For this, a data structure called location counter is used, which points to the next memory word of target program. This is called LC processing. Meanwhile, synthesis phase uses both symbol table for symbolic names and mnemonic table for the accepted list of mnemonics. Thus, the machine code is obtained. So, the functions can be given as:
Analysis phase:

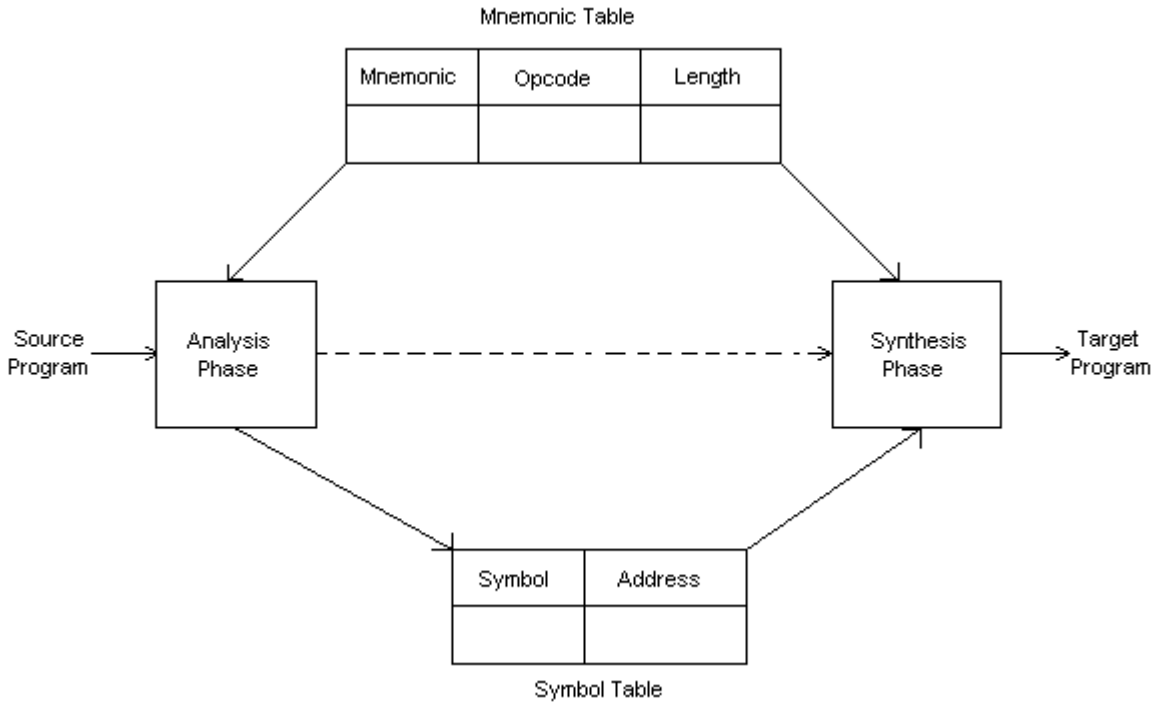
- Isolate label, mnemonic opcode and operand fields of a statement.
- If a label is present, enter the pair (symbol, <LC content>) to symbol table.
- Check validity of mnemonic opcode using mnemonic table.
- Perform LC processing.

Synthesis phase:

- Obtain machine code for the mnemonic from the mnemonic table.
- Obtain address of memory operand from symbol table.

Synthesize the machine instruction.

The phases can be represented as:



Pass structure of an assembler
A pass is defined as one complete scan of the source program, or its equivalent representation.

Single Pass Assemblers

In single pass assembler the translation of assembly language program into object program is done in only one pass. The source program is read only once. These assemblers suffer the problem of forward reference. Handling the forward reference in a single pass assembler is difficult. This type of assemblers avoids forward references.

The object code can be produced in the single pass assemblers in two different ways. In first way the object code is directly loaded into the main memory for execution. Here no loader is required. This type of loading scheme is compile and loading scheme.

In the second way the object program is stored in the secondary memory. This object program will be loaded into the main memory for execution later as necessity arises. Here a separate loader program is necessary. There are various loading schemes available. An assembler, which goes through an assembly language program only once, is known as One-pass assembler. This is faster because they scan the program only once.

Two Pass Assemblers

The two pass assemblers are widely used and the translation process is done in two passes. The two pass assemblers resolve the problem of forward references conveniently. An assembler, which goes through an assembly language program twice, is called a two pass assembler. During the first pass its collects all labels. During the second pass it produces the machine instruction and assigns address to each of them. It assigns addresses to labels by counting their position from the starting address.

Design of two pass assembler

The two pass assembler performs the following functions. It performs some function in pass 1 and some functions in pass 2.

Pass 1

1) Assign address to all statements in the assembly language program.
2) Save the address with label for use in pass 2.
3) Define symbols and literals.
4) Determine the length of machine instructions
5) Keep track of location counter.
6) Process some assembler directions or operations.

Pass 2

1) Perform processing of assembler directives which are not done during the pass 1.
2) Generate the object program.

LC processing

Location counter is the variable used to help in the assignment of addresses. After each source statement is processed the length of the assembled instruction is added with location counter. Whenever a label is reached in the source program, the current value of LOCCTR gives the address associated with that label. Location counter is always incremented to contain the address of the next memory word in the target program. LC is initialized to the constant specified in the START statement.

Data structures used in pass I

OPTAB (operation table)

Fields-

a) mnemonic opcode-shows the name of the instruction

b) class-shows whether instruction is imperative (IS) declarative (DL) and assembler directive (AD)

c) mnemonic info-shows the machine code and instruction length. For DL, AD statement this field contains the address of the routine which finds the length of the statement

Mnemonic opcode	Class	Mnemonic info
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
STOP	AD	00
MOVEM	IS	(05,1)

SYMTAB (symbol table)

-fields are

a) symbol-specifies the label

b) address-address of the label

c) length-length of the label

Symbol	Address
LOOP	202
NEXT	214
LAST	216
A	217
BACK	202
B	218

LITTAB (literal table)

-fields are

a) literals-constants

b) address-address of the literal

LITTAB collects all the literals used in the program address field will be later filled in on encountering LTORG statement

Literal	address
=’5’	211
=’1’	212
=’1’	219

POT (pseudo operation table)

A Pot is a data structure, it maintains all the pseudo operation along with physical address. During pass1 wherever a pseudo operation is identified in source program then search the POT for physical address, if pseudo operation is identified then increment the location counter along with the address of the pseudo operation.

Location counter = location counter + address of pseudo-op

During pass2 the POT tells us which instruction format to use in assembling the instruction. The structure of POT is:

Pseudo-op	physical address
START	5A1A
END	1E5A

The various tables used by the assembler are filled during the pass1 and the output is the intermediate code**.**

Algorithm for Pass1

Step 1: Read first line of the source program
If OPCODE = “START” then
Begin
(i) Save the address of the “START”, it is the starting address of the program.
(ii) Initialize location counter with “START” address
LOCCTR←addr (START)
(iii) Entered the same line (START) in to intermediate file which is used during pass 2
(iv) Read next input line.

END
ELSE
Step 2: initialize location counter to 0

LOCCTR←0
(It means the starting address is not mentioned in the source program)

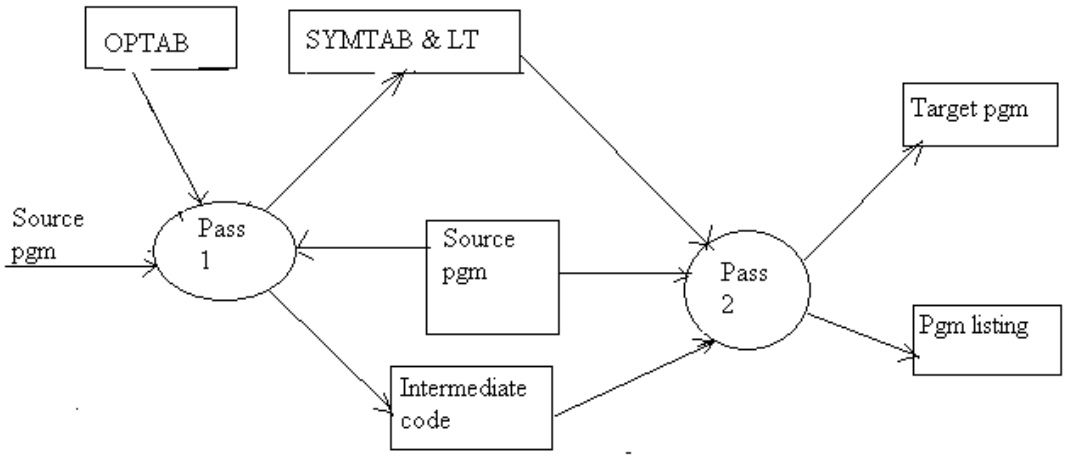
Step 3: while OPCODE! = END do
Begin
If it is a comment line ignore, then
Else
Begin
(i) if there is a symbol in the LABEL field then begin
Search "SYMTAB" for LABEL
If found then
LOCCTR←addr (symbol)
Else
Insert symbol address into "SYMTAB"
End
(ii) search OPTAB for OPCODE
If found then
LOCCTR← LOCCTR + length (OPCODE)
Else
If OPCODE = WORD then
LOCCTR← LOCCTR + 3 bytes
Else
If OPCODE = RESW then
LOCCTR← LOCCTR + 3 * # [OPERAND]
Else
If OPCODE = RESB then
LOCCTR← LOCCTR+ # [OPREAND]
Else
If OPCODE =BYTE then
LOCCTR← LOCCTR + length [constant]
End
Write line into intermediate file, read next input line
End (while)
Write lines to intermediate file
End (end of pass 1)

Algorithm for Pass II

Step 1: Read first line from intermediate file
If OPCODE = "START" then
Begin
Write listing line
Read next input line
End
Step 2: write header record to object program

```
Initialize first text record
While OPCODE! =END
Begin
If it is a comment line then read next source line
Else
Begin
Search OPTAB for OPCODE
If found then
Begin
If there is a symbol in OPERAND field then
Begin
Search SYMTAB for OPERAND
If found then
Store OPERAND address
Else
Begin
Store ‘0’ as OPERAND address
Error undefined symbol
End
End
Store ‘0’ as OPERAND address
Assemble the object code instruction
End (if OPCODE found)
Else
If OPCODE =BYTE or WORD then
Convert the constant into object code
Add object code to text record
End
Write listing line
Read next input line
End (while)
Write last text record to object program
Write END record to object program
End
```

Diagrammatic representation of pass 1 and pass 2



Macros

There can be situations where the same sets of instructions get repeatedly used. Programmer can use the macro facility. Macro instructions are single line abbreviations of the group of statements. For every occurrence of the macro call the macro processor will substitute the entire block. Macro instruction represents a commonly used group of statements in the source programming language.

Eg:

```
A       1, DATA    add contents of DATA to reg 1
A       2, DATA    add contents of DATA to reg 2
A       3, DATA    add contents of DATA to reg 3
 --------------
--------------
--------------

A       1, DATA    add contents of DATA to reg 1
A       2, DATA    add contents of DATA to reg 2
A       3, DATA    add contents of DATA to reg 3
--------------
--------------
--------------
DATA DC   F”5”
```

In the above program the sequence

A 1, DATA
A 2, DATA
A 3, DATA occurs twice

A macro facility permits to attach a name to this sequence and to use this name in its place.

The definitions of macro instructions appear in the source program following START statement. Two new assembler directives MACRO and MEND are used in macro definitions. The directive MACRO identifies the beginning of the macro definitions. MEND directive will indicate the end of the macro definition.

Macro instruction definition

MACRO →start of definition
INCR →macro name

} →Sequence to be abbreviated

MEND →end of definition

The MACRO assembler directive is the first line of operation and identifies the following macro instruction name. Following the name line is the sequence of instruction being abbreviated. The definition is terminated by a line with the MEND pseudo operation.

Source pgm	expanded source
START	-------
-------	-------
-------	---------
MACRO	
INCR	
A 1, DATA	
A 2, DATA	
A 3, DATA	
MEND	

INCR (macro call)	A 1, DATA
	A 2, DATA
-------	A 3, DATA


```
--------
---------
INCR (macro call)
------                                  A   1, DATA
------                                  A   2, DATA
-------                                 A   3, DATA
--------
DATA DC  F"5"
```

Here INCR is the name of macro. Macros are identified by the macro processor. A macro processor is a software program, it is the part of the assembler whenever a macro call is identified by the macro processor with in the assembly language program, and then it collects the macro definition and paste the definition in the place of macro call. In the above example the macro processor identify the INCR statement in the source program, then it expand the INCR code with the following lines

```
A      1, DATA
A      2, DATA
A      3, DATA
```

Syntax of macro call

<macro name><actual parameters>

Macro verses Function

A macro is a group of instruction in assembly language programs, where as functions are group of statements in high level language program. Macros are expanded by macro processor before the execution, where as functions are executed by processor at the time of compilation or execution. When the macro call is identified by the macro processor then paste the definition in the place of macro call where as when the function call is identified by the processor, the control transfer to the function definition at the time of execution. This is the reason it is called dynamic binding. Macro expansion is called static binding.

Macro calls within macros:-

It is possible to define a macro call with in another macro definition.
Consider the eg:-

```
        MACRO
        ADD1 &ARG
        L 1, &ARG
        A 1, ='1'
        ST 1, &ARG
        MEND
        MACRO
        ADDS & ARG1, &ARG2, &ARG3
```

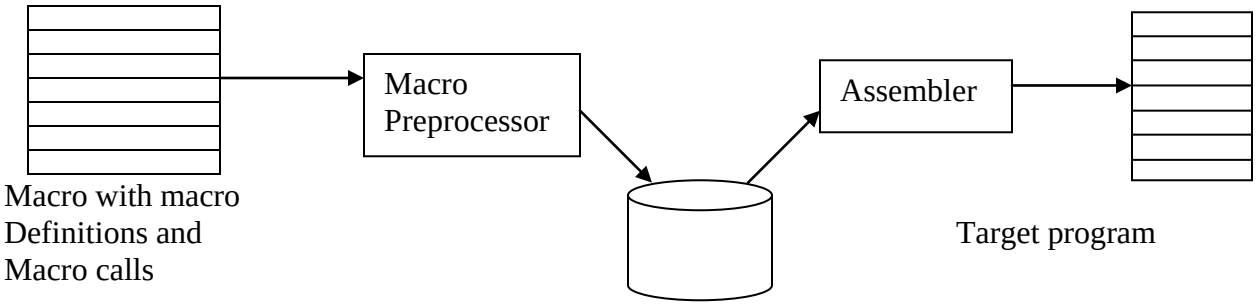
```
ADD1 &ARG1
ADD1 &ARG2
ADD1 &ARG3
MEND
.
;
;
ADDS DATA1, DATA2, DATA3
```

Here ADDS is the name of the macro2.The definition of ADDS having three macro calls to macro ADD1, where ADD1 is the of the macro1.

Macro Preprocessor

Macro preprocessor is system software. It is actually a program, which is a part of the assembler program.

A macro preprocessor accepts an assembly level program containing macro definitions and calls and translate it into an assembly program which does not contain any macro definition and macro call.



There are four basic task that any macro preprocessor must perform.Tey are

1) Identify macro definition
2) Save the definition
3) Identify macro calls
4) Expand calls and substitute arguments

1) Identify macro definition

A macro instruction processor must identify macro definition by the MACRO and MEND pseudo operations.

2) Save the definition

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

3) Identify macro calls

The processor must identify macro calls.

4) Expand calls and substitute arguments

The processor must substitute actual parameters in the place of dummy arguments and expand the macro call with macro definition.

Like assemblers, the macro processor can also be designed in two passes. The first two task performed by the macro processor during pass1 and the third and fourth task perform during the pass2 of the macro processor.

Specification of databases

The following databases are used by the two passes of the macro processor.

1) The macro definition table (MDT)

The macro definition table (MDT) used to store the body of the macro definition.

2) The macro name table (MNT)

The macro name table (MNT) used to store the name of the defined macro.

MNT Table

Index	Macro name	Pointer to MDT
1	SQRT	12

3) **The macro definition table counter (MDTC),** used to indicate the next available entry in the MDT.
4) **The macro name table counter (MNTC)**, used to indicate the next available entry in the MNT.
5) **Argument list array (ALA)**

In pass 1 it is used to substitute index mark for dummy arguments before storing the macro definition. In pass 2 it is used to substitute macro call arguments for the index mark in the stored macro definition.

6) **The macro definition table pointer (MDTP)**, used to indicate the next line of text to be used during macro expansion.

Algorithm for pass1

Algorithm for pass1, which verify each input line of the source program. If the input line is a MACRO pseudo operation then the entire MACRO definition that follows is saved in the next available location in the macro definition table. The first line of the definition is the macro name. The name is entered into the macro name table, along with a pointer to the first location of the MDT entry of the definition. When all the END pseudo operation is encountered, all of the macro definitions have been processed. so control transfers to pass2 in order to process macro calls.

Step 1: Initialization
MDTC = 1
MNTC = 1
Step 2: read next line
If pseudo-op = 'MACRO' then
Read next source line {name line}

Step 3: 1. Enter the macro name in MNT
2. Enter the current value of MDTC in MNT
Step 4: Increment the macro name table counter for next macro entry
MNTC = MNTC +1
Step 5: prepare argument list array.
Step 6: enter macro name line in to MDT and increment the MDTC
MDTC = MDTC + 1
Step 7: read the next source line substitute the arguments.
Step 8: enter the line in to MDT and increment the MDTC
MDTC =MDTC + 1
Continue this process until reach the 'MEND'.
Step 9: else (in step 2)
1. Write copy of source line.
2. If pseudo-op = END then
go to pass2 otherwise read next source line.
Step 10: repeat the step 2 to 8 until read the END pseudo-op.

Algorithm for pass2

The pass2 of the macro processor search the source file line by line for macro calls. If any macro call found then search the MNT for the corresponding entry, if it is found then get the pointer from MNT entry which is point to the corresponding macro definition in MDT.Then the macro expander prepare the argument list array, which contains actual arguments and corresponding dummy arguments. Reading of the MEND line in MDT terminates the expansion of the macro. Continue this process until reach END pseudo operation.

Step 1: read next source line, submitted by pass1.
Step 2: if (source line = macro call) then search MNT for corresponding entry.
If (macro name is found in MNT) then MDTP index from MNT entry.
Step 3: set up argument list array with dummy and actual arguments.
Step 4: increment MDTP
MDTP =MDTP +1
Step 5: get line from MDT and substitute arguments from macro calls.
dummy arguments=actual arguments
Step 6: if (pseudo-op = MEND) then
Read next source line.
Else
Write expanded source card and go to step 4
Step 7: else (for first 'if' in step 2)
Write in to expanded source line file and if (pseudo-op = END) then supply expanded source file to assembler processing.
Else
Go to step 2
Step 8: else (for second if in step 2)
Error condition
Step 9: repeat the steps until END pseudo-op encounter.

Macro Assembler

A macro processor following by assembler is an expensive way to handle macros. It requires more number of passes (2 for macro processor and 2 for assembler).If we combine the macro processor and assembler in a single unit then it is said to be macro assembler. The macro assembler performs macro expansion and program assembling. This may reduce number of passes.