**Varients of Assembler**

There is a list of assemblers ,[computer programs](https://en.wikipedia.org/wiki/Computer_program" \o "Computer program) that translate [assembly language](https://en.wikipedia.org/wiki/Assembly_language) [source code](https://en.wikipedia.org/wiki/Source_code) into binary programs.

* Some assemblers are components of a compiler system for a high level language and may have limited or no usable functionality outside of the compiler system.
* - Some assemblers are hosted on the target processor and operating system, while other assemblers (cross-assemblers) may run under an unrelated operating system or processor.

For example,

1. [**As part of a compiler suite**](https://en.wikipedia.org/wiki/Comparison_of_assemblers#As_part_of_a_compiler_suite)

* Assemblers for [embedded systems](https://en.wikipedia.org/wiki/Embedded_system) are not usually hosted on the target system since it would not have the storage and terminal I/O to permit entry of a program from a keyboard.
* GNU Assembler .
* SDAS (fork of ASxxxx Cross Assemblers and part of the Small Device C Compiler project): GPL: several target instruction sets including Intel 8051, Zilog Z80
* The Amsterdam Compiler Kit

**2. Single target assemblers**

An assembler may have a single target processor or may have options to support multiple processor types. Very simple assemblers may lack features, such as [macros](https://en.wikipedia.org/wiki/Macro_(computer_science)), present in more powerful versions.

Various types are:

* [6502 assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#6502_assemblers)
* [680x0 assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#680x0_assemblers)
* [ARM assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#ARM_assemblers)
* [Mainframe Assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#Mainframe_Assemblers)
* [POWER, PowerPC, and Power ISA assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#POWER,_PowerPC,_and_Power_ISA_assemblers)
* [x86 assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#x86_assemblers)
* [Z80 assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#Z80_assemblers)
* [Other single target assemblers](https://en.wikipedia.org/wiki/Comparison_of_assemblers#Other_single_target_assemblers)

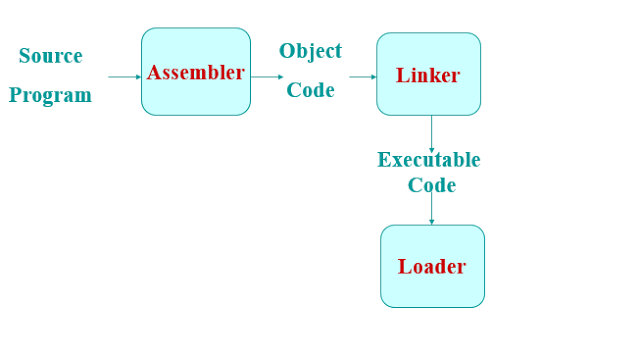
In cross-assemblers, one type of processor is used to generate machine code for another type of processor, which is an assembler. Input and output are both provided by assembly language.

**Design of Two pass assembler**

**WHY DO WE NEED A TWO-PASS ASSEMBLER?**

As explained, the one-pass assembler cannot resolve forward references of data symbols. It requires all data symbols to be defined prior to being used. A two-pass assembler solves this dilemma by devoting one pass to exclusively resolve all (data/label) forward references and then generate object code with no hassles in the next pass. If a data symbol depends on another and this another depends on yet another, the assembler resolved this recursively.

#### Pass II of the Assembler



**Introduction**

* 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

#### Two Pass Assembler

* *Read from input line*
  + LABEL, OPCODE, OPERAND

|  |
| --- |
| two pass assembler |
| Two Pass Assembler |

### Design of 2 – Pass Assembler

**PASS 1:**

1. Separate the Symbol, Mnemonic opcode, and operand fields
2. Build the symbol table
3. Perform LC Processing
4. Construct Intermediate Representation

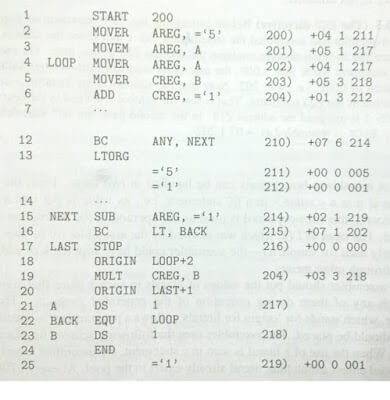
**PASS 2:**

**SYNTHESIZE THE TARGET PROGRAM**

 Advanced Assembler Directives

* ORIGIN
* EQU

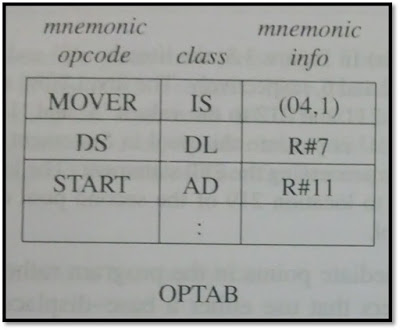
**ORIGIN** Syntax:  
**ORIGIN < Address Specification>**

[](https://2.bp.blogspot.com/-QLX7Q2DCDVE/VgLwCnoPSuI/AAAAAAAAAmA/ojG_XTveIGc/s1600/Picture1.jpg)[ Advanced Assembler Directives](https://1.bp.blogspot.com/-GPOK_QiEH9Q/VgLwCQzfz8I/AAAAAAAAAl8/I-0jKKlD7V4/s1600/Picture2.jpg)

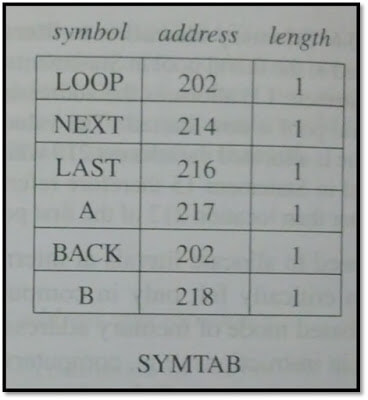
**EQU**Syntax:  
**<Symbol> EQU <Address Specification>**E.g.     MAXLEN    EQU    4096  Pass I of Assembler

* Pass I Use following Data Structures
  + OPTAB
  + SYMTAB
  + LITTAB
  + POOLTAB

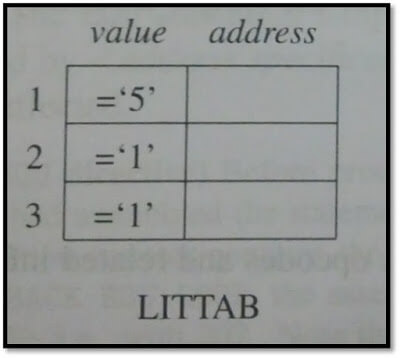
**OPTAB**



**SYMTAB: Symbol Table**

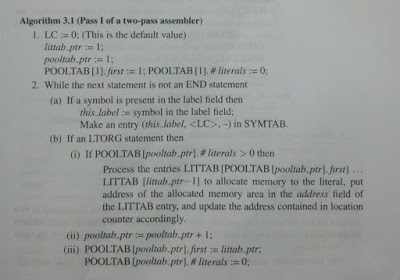


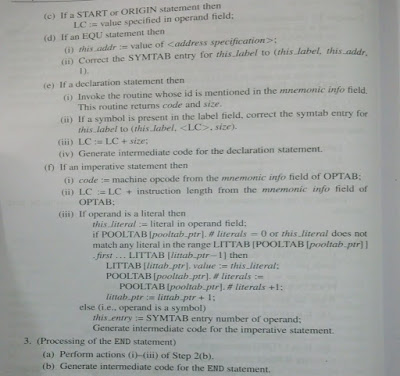
**LITTAB: Table of Literals used in program**



**POOLTAB: A table of information concerning literal pool**

 Algorithm





The main reason why most assemblers use a 2-pass system is to address the problem of forwarding references — references to variables or subroutines that have not yet been encountered when parsing the source code. A strict 1-pass scanner cannot assemble source code which contains forward references. Pass 1 of the assembler scans the source, determining the size and address of all data and instructions; then pass 2 scans the source again, outputting the binary object code. Some assemblers have been written to use a 1.5 pass scheme, whereby the source is only scanned once, but any forward references are simply assumed to be of the largest size necessary to hold any native machine data type. The unknown quantity is temporarily filled in as zero during pass 1 of the assembler, and the forward reference is added to a ‘fix-up list’. After pass 1, the ‘.5’ the pass goes through the fix-up list and patches the output machine code with the values of all resolved forward references. This can result in sub-optimal opcode construction but allows for a very fast assembly phase.

**DIFFERENCE BETWEEN ONE PASS AND TWO PASS ASSEMBLER**

One Pass Assembler The one pass assembler passes over the file once, that is it collects all the information in one loop. It Collects labels and also resolves future references There is a major problem of future referencing Assembles assembly code in one pass It creates an intermediate file which acts as an input to two pass assembler

Two Pass Assembler As the name suggests two pass assembler does two passes over the source file. In first pass, all it does is looks for label definitions and introduces them in the symbol table (a dynamic table which includes the label name and address for each label in the source program). In the second pass, after the symbol table is complete, it does the actual assembly by translating the operations into machine codes and so on.