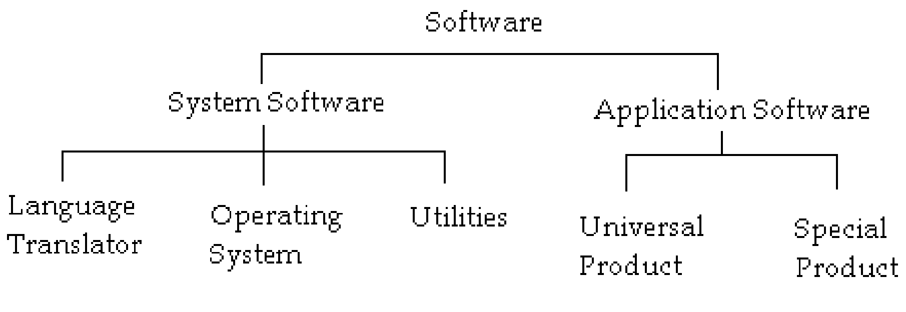
MODULE I
SYSTEM SOFTWARE

GENERAL CONCEPT

A computer software is classified in two parts- System Software and Application Softwares. System software is a type of computer program that is designed to run a computer's hardware and application programs. It is a computer software designed to provide a platform to other software. Examples of system software include operating systems. The system software is the interface between the hardware and user applications. The operating system (OS) is the best-known example of system software.

Other examples of system software and what each does:

- The **BIOS** (basic input/output system) gets the computer system started after we turn it on and manages the data flow between the operating system and attached devices such as the hard disk, video adapter, keyboard, mouse, and printer.
- The **boot** program loads the operating system into the computer's main memory or random access memory (RAM).
- An assembler takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations.
- A **device driver** controls a particular type of device that is attached to our computer, such as a keyboard or a mouse. The driver program converts the more general input/output instructions of the operating system to messages that the device type can understand.
- According to some definitions, system software also includes system utilities, such as the disk defragmenter and System Restore, and development tools such as compilers and debuggers.



A system software consist of many programs for controlling many Input/output operation. An operating system is example of system software. System software is further classified in three parts:

- **Language Translator**:- A program that convert programming source code to machine readable codes are known as language translator. There are three types of language translator.
    1. **Interpreter**:- This is a program that convert high level language to machine level language i.e. Machine level language. The basic property of Interpreter is that it first scan one line of a program or source code, if this is error free then it executes either it will stop the execution. So a interpreter will check a program line by line and execute it, if it is error free. This process takes more time in execution of any program.

    2. **Compiler**:- A compiler is a program used to covert high-level language into machine level. The basic property of compiler is that it first scan all file at a time and check for any error, if no error found then change the program to machine level either show all the errors present in the program. So it takes very less time for execution.

    3. **Assembler**:- An assembler is a program written to convert assembly level language to machine level language.

- **Operating system**:-An operating system is the system software which is used to operate the computer. An operating system manages a computer resource very efficiently, takes care of scheduling of multiple jobs for execution and manages the flow of data, instructions between input/output unit and the main memory. Windows, Unix, Linux, Macintosh etc. are few widely used operating systems. An operating system are classified in different categories with there performance.
    1. ***Single user operating system*:-** A single user operating system give the permission to run or execute one application or program at a time. That is one user can work at a time. e.g.- MS-DOS.

    2. ***Multi-user Operating system***:- A multi-user operating system give the permission to many users work at same time. A transaction process system such as railway reservation system need of hundred of terminals under a single program is example of multi-user operating system. e.g. Unix, Linux etc.

    3. ***Network operating system***:- A network operating system is a collection of software's which allow a set of computers which are interconnected by a computer network to be used together in a convenient and cost effective manner. In a network operating system the users are aware of existence of

multiple computers and can log into remote machine and copy files from one location to another. Like Unix, Windows-NT, Linux etc.

4. ***The Graphical User Interface (GUI) operating system*:-** A GUI uses graphical components like small images, pictures to represent a program, so that instead of typing it we just select it using pointing devices like Mouse etc. Ex. Windows 3.1, Windows-95/98/2000/ME/XP, Linux etc.

- **Utilities**:- Utility program are the programs which are often used by application program. These utility programs are created by the manufacturer. Ex. Text Editors, Sorting, Formatting etc.

ASSEMBLERS

Assemblers are the simplest of a class of systems programs called *translators*. A translator is simply a program which translates from one (computer) language to another (computer) language. In the case of an assembler, the translation is from assembly language to object language (which is input to a loader). Notice that an assembler, like all translators, adds nothing to the program which it translates, but merely changes the program from one form to another. The use of a translator allows a program to be written in a form which is convenient for the programmer, and then the program is translated into a form convenient for the computer.

Assembly language is almost the same as machine language. The major difference is that assembly language allows the declaration and use of symbols to stand for the numeric values to be used for opcodes, fields, and addresses. An assembler inputs a program written in assembly language, and translates all of the symbols in the input into numeric values, creating an output object module, suitable for loading. The object module is output to a storage device which allows the assembled program to be read back into the computer by the loader.

This is an external view of an assembler. Now we turn our attention to an internal view, in order to see how the assembler is structured internally to translate assembly programs into object programs.

A computer will not understand any program written in a language, other than its machine language. The programs written in other languages must be translated into the machine language. Such translation is performed with the help of software. A program which translates an assembly language program into a machine language program is called an assembler. If an assembler which runs on a computer and produces the machine codes for the same computer then it is called self assembler or resident assembler. If an assembler that runs on a computer and produces the machine codes for other computer then it is called Cross Assembler.

Assemblers are further divided into two types: One Pass Assembler and Two Pass Assembler. One pass assembler is the assembler which assigns the memory addresses to the variables and translates the source code into machine code in the first pass simultaneously. A Two Pass Assembler is the assembler which reads the source code twice. In the first pass, it reads all the variables and assigns them memory addresses. In the second pass, it reads the source code and translates the code into object code.

Assembler is a computer program which is used to translate program written in Assembly Language in to machine language. The translated program is called as object program. Assembler checks each instruction for its correctness and generates diagnostic messages, if there are mistakes in the program. Various steps of assembling are:

- Input source program in Assembly Language through an input device.
- Use Assembler to produce object program in machine language.
- Execute the program.

DATA STRUCTURES

Appropriate data structures can make a program much easier to understand, and the data structures for an assembler are crucial to its programming. An assembler must translate two different kinds of symbols: assembler-defined symbols and programmer-defined symbols. The assembler-defined symbols are mnemonics for the machine instructions and pseudo-instructions. Programmer-defined symbols are the symbols which the programmer defines in the label field of statements in his program. These two kinds of symbols are translated by two different tables: the opcode table, and the symbol table.

- **The opcode table.**

The opcode table contains one entry for each assembly language mnemonic. Each entry needs to contain several fields. One field is the character code of the symbolic opcode. In addition, for machine instructions, each entry would contain the numeric opcode and default field specification. These are the minimal pieces of information needed. With an opcode table like this, it is possible to search for a symbolic opcode and find the correct numeric opcode and field specification.

Pseudo instructions require a little more thought. There is no numeric opcode or field specification associated with a pseudo-instruction; rather, there is a function which must be performed for each pseudo-instruction. This can be encoded in several ways. One method is to include in the opcode table, a type field. The type field is used to separate pseudo-instructions from machine instructions and to separate the various pseudo-instructions. Different things need to be done for each of the different pseudo-instructions, and so each has its own type. All machine instructions can be handled the same, however, so only one type is needed for them. One possible type assignment is then,

Type	**Instruction Class**
1	Machine instruction
2	ORIG pseudo-instruction
3	CON pseudo-instruction
4	ALF pseudo-instruction
5	EQU pseudo-instruction
6	END pseudo-instruction.

Other type assignments are also possible. For example, in the above assignment, all machine instructions have one type, and are treated equally. This allows both instructions such as:

`LDA  X(0:3)` and `ENTA X(0:3)`

But the ENTA instruction is encoded as an opcode of 60 and a field of 02. Thus, it is not correct to specify a field with an ENTA. This reasoning can lead to separating memory reference instructions (which may have field specifications) from non-memory reference instructions (which should not have field specifications). Even further classification could separate the I/O instructions (which use the F field for a device number) and the MOVE instruction. The idea of attaching a type to a table entry is quite general.

Another approach to separating the pseudo-instructions in the opcode table is to consider how the type field of the above discussion would be used. It would be used in a jump table. In this case, rather than using the type to specify the code address through a jump table, we could store the address of the code directly in the opcode table. In each opcode table entry, we can store the address of the code to be executed for proper treatment of this instruction, whether it is a machine instruction or a pseudo-instruction. This is a commonly used technique for assemblers.

Other opcode table structures are possible also. Some assemblers have separate tables for machine instructions and for pseudo-instructions. Others will use a type field of one bit to distinguish between machine instructions and pseudo-instructions, and then store a numeric opcode and field specification for the machine instructions or an address of code to execute for pseudo-instructions.

For simplicity, let us assume that we store, for each entry, the symbolic mnemonic, numeric opcode, default field specification and a type, as defined above. For pseudo-instructions, the numeric opcode and default field specification of the entry will be ignored. How should we organize our opcode table entries? The opcode table should be organized to minimize both search time and table space. These two goals may not be achievable at the same time. The fastest access to table entries would require that each field of an entry be in the same relative position of a memory word, such as in Figure 8.1. But notice that, in this case, three bytes of each entry are unused, so that the table includes a large amount of wasted space. (Actually, three bytes, two signs, and the upper three bits of the type byte are unused.) To save this space would require more code and longer execution times for packing and unpacking operations. Thus, to save time it seems wise to accept this wasted space in the opcode table.

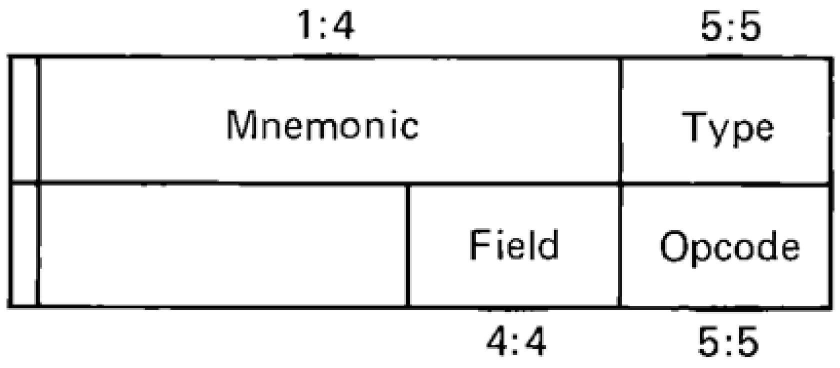


FIGURE : *Opcode table entry (each opcode table entry contains the symbolic opcode, numeric opcode and field specification, and a type field).*

To a great extent this wasted space is due to the design of the assembly language mnemonics. If the mnemonics had been defined as a maximum of three characters (instead of four), it would have been possible to store the mnemonic, field specifications, and opcode in one word. The sign bit would be "+" for machine instructions and "-" for pseudo-instructions. Pseudo instructions could have a type field or address in the lower bytes while machine instructions would store opcode and field specifications. On the other hand, once the decision is made that four character mnemonics are needed, then it is necessary to go to a multi-word opcode table entry. In this case, we could allow mnemonics of up to seven characters, by using the currently unused three bytes of the opcode table entry. On the other hand, there may be no desire to have opcode mnemonics of more than four characters; mnemonics should be short.

Another consideration is the order of entries in the table. This relates to the expected search time to find a particular entry in the table. The simplest search is a linear search starting at one end of the table and working towards the other end until a match is found (or the end of the table is reached). In this case, one should organize the table so that the more common mnemonics will be compared first. If, as expected, the LDA mnemonic is the most common instruction, then it should be put at the end of the table which is searched first.

Rather than use a linear search, it can be noted that the opcode table is a static structure; it does not change. The opcode table is like a dictionary explaining the numeric meaning of symbolic opcodes. Like a dictionary, it can be usefully organized in an alphabetical order. By ordering the entries, the table can be searched with a *binary search*. A binary search is the method commonly used to look for an entry in a dictionary, telephone book, or encyclopedia. First the middle of the book is compared with the entry being searched for (the *search key*). If the search key is smaller than the middle entry, then the key must be in the first half of the book, if it is larger it must be in the latter half of the book. After this comparison, the same idea can be repeated on the half of the book which still needs to be searched. Each comparison splits the remaining section of the book in half.

A binary search is not always the best search method to use. Each time through, the search loop cuts the size of the table yet to be searched in half. In general, a table of size 2^n will take about n comparisons to find an entry. Thus, a table of size 32 will take only 5 comparisons, while for a linear search, it is normally assumed, that on the average, half of the table must be searched, resulting in 16 comparisons. Thus, the binary search almost always requires fewer executions of its search loop than a linear search. However, notice that the binary search requires considerably more computation per comparison than a linear search. The binary search loop requires about 35 time units per comparison while a linear search can require only 5 time units. Thus, for a table of size 32, the binary search takes 5 comparisons at 35 time units each, for 175 time units, while the linear search takes 16 comparisons at 5 time units each for 80 time units. This is not to say that a binary search should never be used. For a table of size 128, a binary search will take about 245 (= 7 × 35) time units, while a linear search will take 320 (= 64 × 5). Thus, for a large table a binary search is better. Also, if a shift (SRB) could be used instead of the divide, the time per loop could be cut by 11 time units, making the binary search better. Since there are around 150 MIX opcodes, we use a binary search for the opcode table.

➢ The symbol table

The opcode table is used to translate the assembler-defined symbols into their numeric equivalents; the symbol table is used to translate programmer-defined symbols into their numeric equivalents. Thus, these two tables can be quite alike. A symbol table, like an opcode table, is a table of many entries, one entry for each programmer-defined symbol. Each entry is composed of several fields.

For the symbol table, only two fields are basically needed. It is necessary to store the symbol and the value of the symbol. A symbol in a MIXAL program can be up to 10 characters in length. This requires two MIX words. In addition, the value of the symbol can be up to five bytes plus sign, requiring another MIX word. Thus, each entry in the symbol table takes at least three words. Additional fields may be added for some assemblers. A bit may be needed to indicate if the symbol has been defined or is undefined (i.e., a forward reference). Another bit may specify whether the symbol is an absolute symbol or a relative symbol (depending on whether the output is to be used with a relocatable or absolute loader). Other fields may also be included in a symbol table entry, but for the moment let us use only the two fields, for the symbol and its value.

As with the opcode table, the organization of the symbol table is very important for proper use. But the symbol table differs from the opcode table in one important respect: it is *dynamic*. The opcode table was *static*; it is never changed, neither during nor between executions of the assembler. It is the same for each and every assembly program for the entire assembly process. The symbol table is dynamic; each program has its own set of symbols with their own values and new symbols are added to the symbol table as the assembly process proceeds. Initially the symbol table is empty; no symbols have been defined. By the time assembly is completed, however, all symbols in the program have been entered into the symbol table.

This requires the definition of two subroutines to manipulate the symbol table: a search routine and an enter routine. The search routine searches the symbol table for a symbol (its search key) and returns its value (or an index into the table to its value). The enter subroutine puts a new symbol and its value into the table.

These two subroutines need to be designed together, since both of them affect the symbol table. A binary search might be quite efficient, but it requires that the table be kept ordered. This means that the enter subroutine would have to adjust the table for each new entry, so that the table was always sorted into the correct order. Thus, although a binary search might be quick, the combination of a binary search plus an ordered enter might be very expensive.

Also consider that a linear search is more efficient than a binary search for small tables. Many assembly language programs have less than 200 different symbols, and so a linear search may be quite reasonable. A linear search allows the enter routine to simply put any new symbol and its value at the end of the table. Thus, both the search and enter routines are simple.

Many other table management techniques have been considered and are used. Some of these are quite complex and useful only for special cases. Others have wide applicability. One of the most commonly used techniques is *hashing*. The objective of hashing is quite simple. Rather than have to search for a symbol at all, we would prefer to be able to compute the location in the table of a symbol from the symbol itself. Then, to enter a symbol, we compute where it should go, and define that entry. For accessing, we compute the address of the entry and use the value stored in the table at that location.

As a simple example, assume that all of our symbols were one-letter symbols (A, B, C, …, Z). Then if we simply allocated a symbol table of 26 locations, we could index directly to a table entry for each symbol by using its character code for an index. No searching would be needed. If our symbols were any two-letter symbols, we could apply the same idea if we had a symbol table of 676 (= 26 × 26) entries, where our hash function would be to multiply the character code of the first letter by 26 and add the character code of the second (and subtract 26 to normalize). However, for three-letter symbols, we would need 17,576 spaces in our symbol table. This is clearly impossible. (Our MIX memory is only 4000 words long.) It also is not necessary, since we assume that at most only a few hundred symbols will be used in each different assembly program. What is needed is a function which produces an address for each different symbol, but maps them all into a table of several hundred words.

Many different hashing functions can be used. For example, we can add together the character codes for the different characters of the symbol, or multiply them, or shift some of them a few bits and exclusive-OR them with the other characters, or whatever we wish. Then, after we have hashed up the input characters to get some weird number, we can divide by the length of the table and use the remainder. The remainder is guaranteed to be between 0 and the length of the table and hence can be used as an index into the table. For a binary machine and a table whose length is a power of two, the division can be done by simply masking the low-order bits.

The objective of all this calculation is to arrive at an address for a symbol which hopefully is unique for each symbol. But since there are millions of 10-character symbols, and only a few hundred table entries, it must be the case that two different symbols may hash into the same table entry. For example, if we use a hash function which adds together the character codes of the letters in the symbol, then both EVIL and LIVE will hash to the same location. This is called a *collision*. The simplest solution to this problem is to then search through successive entries in the table, until an empty table entry is found. (If the end of the table is found, start over at the beginning).

The search and enter routines are now straightforward. To enter a new symbol, compute its hash function, and find an empty entry in the table. Enter the new symbol in this entry. To search for a symbol, compute its hash function and search the table starting at that entry. If an empty entry is found, the symbol is not in the table. Otherwise, it will be found before the first empty location.

The problem with using hashing is defining a good hash function. A good hash function will result in very few collisions. This means that both the search and enter routines will be very fast. In the worst case, all symbols will hash to the same location and a linear search will result. Hashing is sometimes also used for opcode tables, where, since the opcode table is static and known, a hashing function can be constructed which guarantees no collisions.

- **Other data structures**

The opcode table and the symbol table are the major data structures for an assembler, but not the only ones. The other data structures differ from assembler to assembler, depending upon the design of the assembly language and the assembler.

A buffer is probably needed to hold each input assembly language statement, and another buffer is needed to hold the line image corresponding to that statement for the listing of the program. In addition buffers may be needed to create the object module output for the loader, and to perform double buffering in order to maximize CPU-I/O overlap. Various variables are needed to count the number of cards read, the number of symbols in the symbol table, and so on.

One variable in particular is important. This is the *location counter*. The location counter is a variable which stores the address of the location into which the current instruction is to be loaded. The value of the location counter is used whenever the symbol "*" is used, and this value can be set by the ORIG pseudo-instruction. Normally the value of the location counter is increased by one after each instruction is assembled, in order to load the next instruction into the next location in memory. The value of the location counter for each instruction is used to instruct the loader where to load the instruction.

General Flow of An Assembler

With a familiarity with the basic data structures of an assembler (the opcode table, the symbol table, and the location counter), we can now describe the general flow of an assembler. Each input assembly statement, each card, is handled separately, so our most general flow would be simply, "Process each card until an ENDpseudo-instruction is found."

More specifically, consider what kind of processing is needed for each card:

1. Read in a card.
2. If the card is a comment (column 1 is an "*") then skip over processing to step 4 for printing.
3. For non-comment cards, get the opcode and search the opcode table for it. Using the type field of the opcode table entry, process this card.
4. After the card has been processed, print a line of listing for this card.
5. If the opcode was not an END pseudo-instruction, go back to step 1 to process the next card.

This much of the assembly process is common to all of the input cards. The important processing is in step 3, where each card is processed according to its type. This level of the assembler provides an organizational framework for further development.

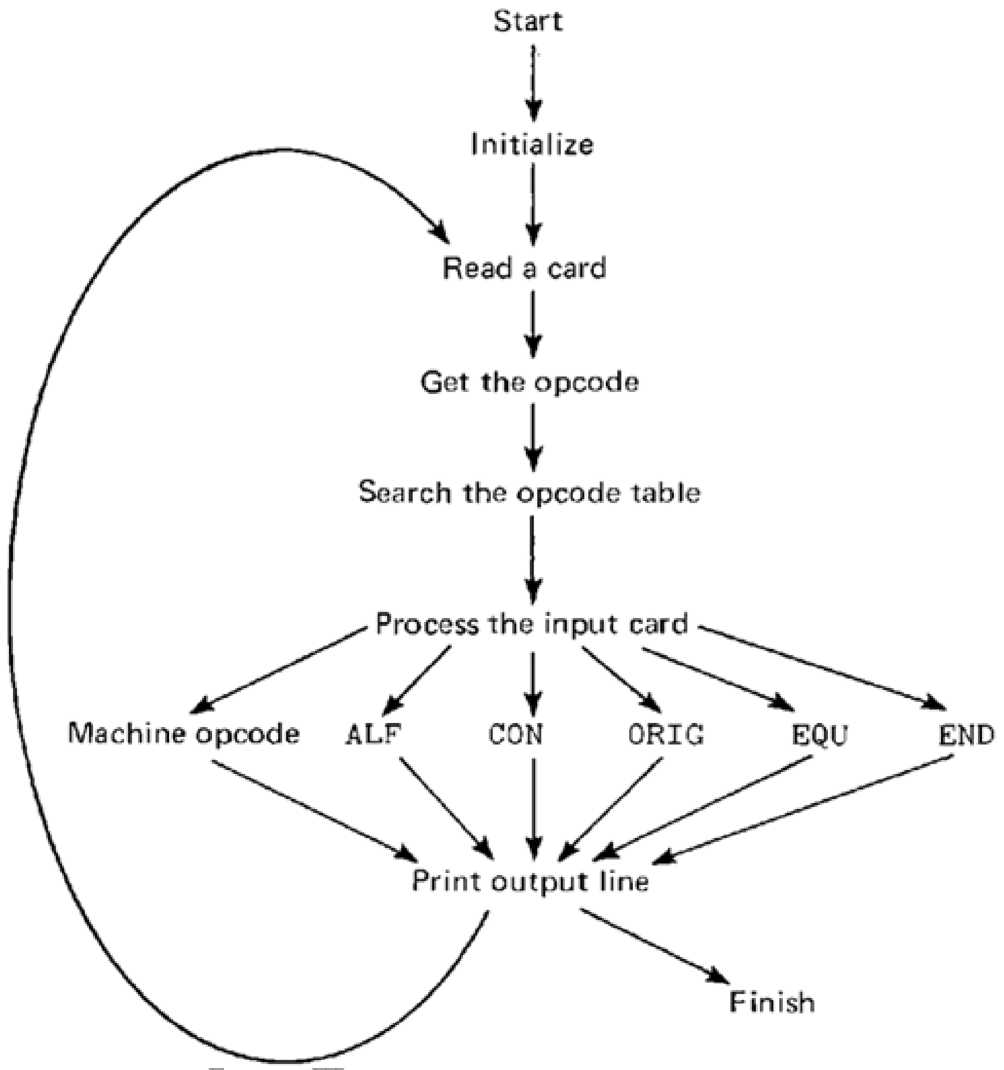


FIGURE: *The general flow of an assembler (for MIXAL). Refinement of this general design resulted in the assembler described in Section 8.3.*

Two pass assemblers

A two-pass assembler makes two passes over the input program. That is, it reads the program twice. On the first pass, the assembler constructs the symbol table. On the second pass, the complete symbol table is used to allow expressions to be evaluated without problems due to forward references. If any symbol is not in the symbol table during an expression evaluation, it is an error, not a forward reference.

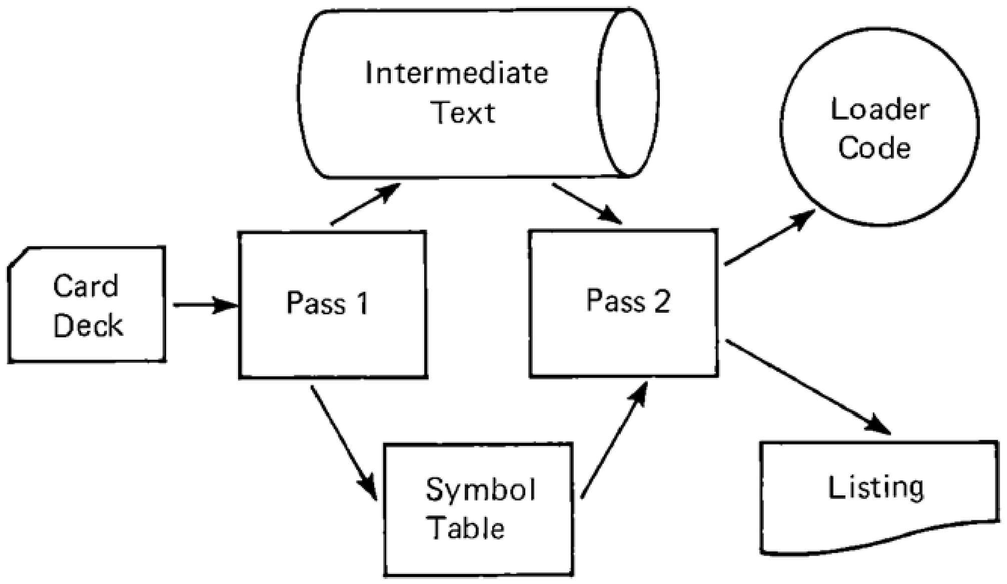


FIGURE: *A block diagram of a two-pass assembler. Pass 1 produces a symbol table and a copy of the input source program for pass 2. Pass 2 produces loader code and a listing.*

Briefly, for a two-pass assembler, the first pass constructs the symbol table; the second pass generates object code. This causes a major change in the basic flow of an assembler, and results in another important data structure: the intermediate text. Since two passes are being made over the input assembly language, it is necessary to save a copy of the program read for pass 1, to be used in pass two. Notice that since the assembly listing includes the generated machine language code, and the machine language code is not produced until pass 2, the assembly listing is not produced until pass 2. This requires storing the entire input program, including comments, and so forth, which are not needed by the assembler during pass 2.

The intermediate text can be stored in several ways. The best storage location would be in main memory. However, since MIX memory is so small, this technique is not possible on the MIX computer. On other machines, with more memory, this technique is sometimes used for very small programs. Another approach, used for very small machines, is to simply require the programmer to read his program into the computer twice, once for pass 1, and again for pass 2. A PDP-8 assembler has used this approach; even going so far as to require the program be read in a third time if a listing is desired.

A more common solution is to store the intermediate text on a secondary storage device, such as tape, drum, or disk. During pass 1, the original program is read in, and copied to secondary storage as the symbol table is constructed. Between passes, the device is repositioned (rewound for tapes, or the head moved back to the first track for moving head disk or drum). During pass 2, the program is read back from secondary storage for object code generation and printing a listing. (Notice that precautions should be taken that the same tape is not used for both storage of the intermediate text input for pass 2 and the storage of the object code produced as output from pass 2 for the loader.)

The general flow of a two-pass assembler differs from that given above, in that now each card must be processed twice, with different processing on each pass. It is also still necessary to process each type of card differently, depending upon its opcode type. This applies to both pass 1 and pass 2.

For machine instructions, pass 1 processing involves simply defining the label (if any) and incrementing the location counter by one. ALF and CON pseudo-instructions are handled in the same way. ORIG statements must be handled exactly as we described above, however, in order for the location counter to have the correct value for defining labels. Similarly, EQU statements must also be treated on pass 1. This means that no matter what approach is used, no forward references can be allowed in ORIG or EQU statements, since both are processed during pass 1. The END statement needs to have its label defined and then should jump to pass 2 for further processing.

During pass 2, EQU statements can be treated as comments. The label field can likewise be ignored. ALF, CON, and machine instructions will be processed as described above, as will the END statement.

The need to make two passes over the program can result in considerable duplication in code and computation during pass 1 and pass 2. For example, on both passes, we need to find the type of the opcode; on pass 1, to be able to treat ORIG, EQU, and END statements; on pass 2, for all types. This can result in having to search the opcode table twice for each assembly language statement. To prevent this, we need only save the index into the opcode table of the opcode (once it is found during pass 1) with each instruction. Also, consider that since the operand "*" may be used in the expressions evaluated during pass 2, it is necessary to duplicate during pass 2 the efforts of pass 1 to define the location counter, unless we can simply store with each assembly language statement the value of the location counter for that statement.

These considerations can result in extending the definition of the intermediate text to be more than simply the input to pass 1. Each input assembly language statement can be a record containing (at least) the following fields.

1. the input card image.
2. the index of the opcode into the opcode table.
3. the location counter value for this statement.

Additional fields may include error flags, a pointer to the column in which the operand starts (for free-format assemblers), a line number, and so on; whatever is conveniently computed on pass 1 and needed on pass 2.

Even more can be computed during pass 1. For example, ALF and CON statements can be completely processed during pass 1. The opcode field, field specification, and index field for a machine instruction can be easily computed during pass 1, and most of the time the address field, too, can be processed on pass 1. It is only the occasional forward reference which causes a second pass to be needed. Most two-pass assemblers store a partially digested form of the input source program as their intermediate text for pass 2.

Assemblers, like loaders, are almost always I/O-bound programs, since the time to read a card generally far exceeds the time to assemble it. Thus, requiring two passes means an assembler takes twice as long to execute as an assembler which only uses one pass.

One pass assemblers

It is these considerations which have given rise to one-pass assemblers. A one-pass assembler does everything in one pass through the program, much as described earlier. The only problem with a one-pass assembler is caused by forward references, of course. The solutions to this problem are the same as were presented earlier for one-pass relocatable linking loaders: Use-tables or Chaining.

In either case, Use-table or Chaining, it is not possible to completely generate the machine language statement which corresponds to an assembly language statement; the address field cannot always be defined. Thus, some later program must fix-up those instructions with forward reference. In a two-pass assembler, this program is pass 2. In a one-pass assembler, there is no second pass, and so the program which must fix-up the forward references is the *loader*. The loader resolves forward references for a one-pass assembler.

If Use-tables are used to solve the future reference problem, then the assembler keeps track of all forward references to each symbol. After the value of the symbol is defined, the assembler generates special loader instructions to tell the loader the addresses of all forward references to a symbol and its correct value. When the loader encounters these special instructions during loading, it will fix-up the address field of the forward reference instruction to have the correct value.

A variation of this same idea is to use chaining. With chaining, the entries in the Use-table are kept in the address fields of the instructions which forward reference symbols. Only the address of the most recent use must be kept. When a new forward reference is encountered, the address of the previous reference is used, and the address of the most recent reference is kept. When a symbol is defined which has been forward referenced (or at the end of the program), special instructions are again issued to the loader to fix-up the chains which have been produced.

Variations on this basic theme are possible also. For example, if the standard loader will not fix-up forward references, it would be possible for the assembler to generate some special instructions which would be executed first, after loading, but before the assembled program is executed to fix-up forward references. But the basic idea remains the same: a one-pass assembler generates its object code for the loader in such a way that the loader, or the loaded program itself, will fix-up forward references.

LOADERS

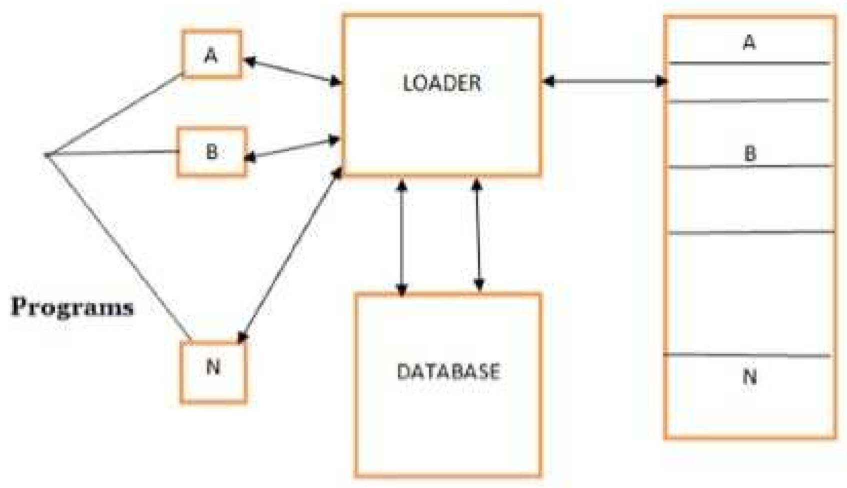
Loader is a program that loads machine codes of a program into the system memory. In Computing, a loader is the part of an Operating System that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code. All operating systems that support program loading have loaders. In many operating systems the loader is permanently resident in memory.

In a computer operating system , a loader is a component that locates a given program (which can be an application or, in some cases, part of the operating system itself) in offline storage (such as a hard disk), loads it into main storage (in a personal computer, it's called random access memory), and gives that program control of the computer (allows it to execute its instructions).

A program that is loaded may itself contain components that are not initially loaded into main storage, but can be loaded if and when their logic is needed. In a multitasking operating system, a program that is sometimes called a *dispatcher* juggles the computer processor's time among different tasks and calls the loader when a program associated with a task is not already in main storage. (By program here, we mean a binary file that is the result of a programming language compilation, linkage editing, or some other program preparation process).

Though loaders in different operating systems might have their own nuances and specialized functions native to that particular operating system, they still serve basically the same function. The following are the responsibilities of a loader:

- Validate the program for memory requirements, permissions, etc.
- Copy necessary files, such as the program image or required libraries, from the disk into the memory
- Copy required command-line arguments into the stack
- Link the starting point of the program and link any other required library
- Initialize the registers
- Jump to the program starting point in memory



The loader performs the following functions:

1) Allocation
2) Linking
3) Relocation
4) Loading

Allocation:

- Allocates the space in the memory where the object program would be loaded for Execution.
- It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.
- In absolute loader allocation is done by the programmer and hence it is the duty of the programmer to ensure that the programs do not get overlap.
- In reloadable loader allocation is done by the loader hence the assembler must supply the loader the size of the program.

Linking:

- It links two or more object codes and provides the information needed to allow references between them.
- It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.
- In absolute loader linking is done by the programmer as the programmer is aware about the runtime address of the symbols.
- In relocatable loader, linking is done by the loader and hence the assembler must supply to the loader, the locations at which the loading is to be done.

Relocation:

- It modifies the object program by changing the certain instructions so that it can be loaded at different address from location originally specified.
- There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.
- In absolute Loader relocation is done by the assembler as the assembler is aware of the starting address of the program.
- In relocatable loader, relocation is done by the loader and hence assembler must supply to the loader the location at which relocation is to be done.

Loading:

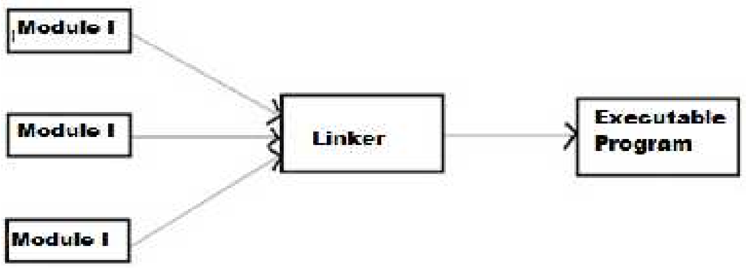
- It brings the object program into the memory for execution.
- Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.
- In both the loaders (absolute, relocatable) Loading is done by the loader and hence the assembler must supply to the loader the object program.

LINKERS:

In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Not built in libraries, it also links the user defined functions to the user defined libraries. Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.

A utility program that combines several separately compiled modules into one, resolving internal differences between them. When a program is assembled/compiled, an intermediate form is produced into which it is necessary to incorporate libraries, and any other modules supplied by the user.



In computer science, a linker is a computer program that takes one or more object files generated by a compiler and combines them into one, executable program. Computer programs are usually made up of multiple modules that span separate object files, each being a compiled computer program. The program as a whole refers to these separately-compiled object files using symbols. The linker combines these separate files into a single, unified program; resolving the symbolic references as it goes along.

Dynamic Linking

Dynamic linking is a similar process, available on many operating systems, which postpones the resolution of some symbols until the program is executed. When the program is run, these dynamic link libraries are loaded as well. Dynamic linking does not require a linker. The linker bundled with most Linux systems is called ld.

Advantages:
- Often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary.
- If a bug in a library function is corrected by replacing the library, all programs using it dynamically will benefit from the correction after restarting them. Programs that included this function by static linking would have to be re-linked first.

Disadvantages:
- Known on the Windows platform as "DLL Hell", an incompatible updated library will break executables that depended on the behavior of the previous version of the library if the newer version is not properly backward compatible.
- A program, together with the libraries it uses, might be certified (e.g. as to correctness, documentation requirements, or performance) as a package, but not if components can be replaced.

Static Linking

Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is more portable, since it does not require the presence of the library on the system where it runs.

MACROS

In Microsoft Word and other programs, a macro is a saved sequence of command s or keyboard strokes that can be stored and then recalled with a single command or keyboard stroke.

In computers, a macro (for "large"; the opposite of "micro") is any programming or user interface that, when used, expands into something larger. The original use for "macro" or "macro definition" was in computer assembler language before higher-level, easier-to-code languages became more common. In assembler language, a macro definition defines how to expand a single language statement or computer instruction into a number of instructions. The macro statement contains the name of the macro definition and usually some variable parameter information.

Macros are useful especially when a sequence of instructions is used a number of times (and possibly by different programmers working on a project). Some pre-compilers also use the macro concept. In general, however, in higher-level languages, any language statement is about as easy to write as an assembler macro statement.

Assembler macros generate instruction s inline with the rest of a program. More elaborate sequences of instructions that are used frequently by more than one program or programmer are encoded in subroutines that can be branched to from or assembled into a program.

A macro name is an abbreviation, which stands for some related lines of code. Macros are useful for the following purposes:

- To simplify and reduce the amount of repetitive coding
- To reduce errors caused by repetitive coding
- To make an assembly program more readable.

A macro consists of name, set of formal parameters and body of code. The use of macro name with set of actual parameters is replaced by some code generated by its body. This is called macro expansion.

Macros allow a programmer to define pseudo operations, typically operations that are generally desirable, are not implemented as part of the processor instruction, and can be implemented as a sequence of instructions. Each use of a macro generates new program instructions; the macro has the effect of automating writing of the program.

Macros can be defined used in many programming languages, like C, C++ etc. Example macro in C programming.Macros are commonly used in C to define small snippets of code. If the macro has parameters, they are substituted into the macro body during expansion; thus, a C macro can mimic a C function. The usual reason for doing this is to avoid the overhead of a function call in simple cases, where the code is lightweight enough that function call overhead has a significant impact on performance.

For instance,

```
    #define max (a, b) a>b? A: b
```

Defines the macro max, taking two arguments a and b. This macro may be called like any C function, using identical syntax. Therefore, after preprocessing

```
    z = max(x, y);
```

Becomes

```
    z = x>y? X:y;
```

While this use of macros is very important for C, for instance to define type-safe generic data-types or debugging tools, it is also slow, rather inefficient, and may lead to a number of pitfalls.

C macros are capable of mimicking functions, creating new syntax within some limitations, as well as expanding into arbitrary text (although the C compiler will require that text to be valid C source code, or else comments), but they have some limitations as a programming construct. Macros which mimic functions, for instance, can be called like real functions, but a macro cannot be passed to another function using a function pointer, since the macro itself has no address.

Macro Expansion.

A macro call leads to macro expansion. During macro expansion, the macro statement is replaced by sequence of assembly statements. In the above program a macro call is shown in the middle of the figure. i.e. INITZ. Which is called during program execution? Every macro begins with MACRO keyword at the beginning and ends with the ENDM (end macro).whenever a macro is called the entire is code is substituted in the program where it is called. So the resultant of the macro code is shown on the right most side of the figure. Macro calling in high level programming languages.

(C programming)

```
#define max(a,b) a>b?a:b
Main ()
{
    int x , y;
    x=4;
    y=6;
    z = max(x, y);
}
```

The above program was written using C programming statements. Defines the macro max, taking two arguments a and b. This macro may be called like any C function, using identical syntax. Therefore, after preprocessing becomes

```
z = x>y ? x: y;
```

After macro expansion, the whole code would appear like this.

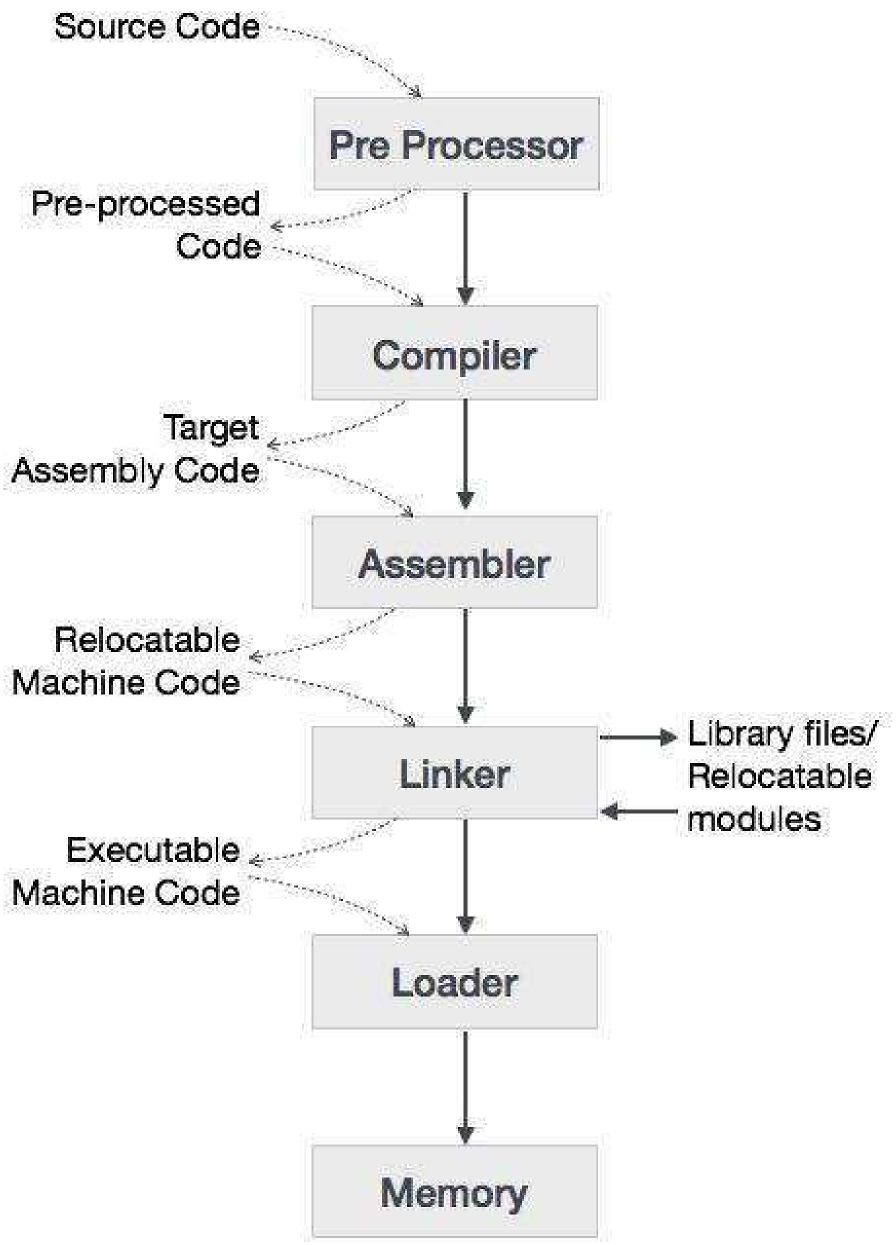
```
#define max(a,b) a>b?a:b
main()
{
    int x , y;
    x=4;
    y=6;
    z = x>y?x:y;
}
```

COMPILERS

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor. The file that is created contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called object code or sometimes an object module. The object code is machine code that the processor can execute one instruction at a time.

The Java programming language, a language used in object-oriented programming, introduced the possibility of compiling output (called bytecode) that can run on any computer system platform for which a Java virtual machine or bytecode interpreter is provided to convert the bytecode into instructions that can be executed by the actual hardware processor. Using this virtual machine, the bytecode can optionally be recompiled at the execution platform by a just-in-time compiler.



Traditionally in some operating systems, an additional step was required after compilation - that of resolving the relative location of instructions and data when more than one object module was to be run at the same time and they cross-referred to each other's instruction sequences or data. This process was sometimes called linkage editing and the output known as a load module.

A compiler works with what are sometimes called 3GL and higher-level languages. An assembler works on programs written using a processor's assembler language.
A compiler is a program that translates a programme written in HLL to executable machine language. The process of transferring HKK source program in to object code is a lengthy and complex process as compared to assembling. Compliers have diagnostic capabilities and prompt the programmer with appropriate error message while compiling a HLL program. The corrections are to be incorporated in the program, whenever needed, and the program has to be recompiled. The process is repeated until the program is mistake free and translated to an object code. Thus the job of a complier includes the following:

1. To translate HLL source program to machine codes.
2. To trace variables in the program
3. To include linkage for subroutines.
4. To allocate memory for storage of program and variables.
5. To generate error messages, if there are errors in the program.

Important phases in Compilation
The following is a typical breakdown of the overall task of a compiler in an approximate sequence -

1. ***Lexical Analysis***

Lexical analysis in a compiler can be performed in the same way as in an assembler. Generally in an HLL there are more number of tokens to be recognised - various keywords (such as, *for*, *while*, *if*, *else*, etc.), punctuation symbols (such as, comma, semi-colon, braces, etc.), operators (such as arithmetic operators, logical operators, etc.), identifiers, etc. Tools like *lex* or *flex* are used to create lexical analyzers.

2. ***Syntax Analysis***

Syntax analysis deals with recognizing the structure of input programs according to known set of *syntax rules* defined for the HLL. This is the most important aspect in which HLLs are significantly different from lower level languages such as assembly language. In case of HLLs, the syntax rules are much more complicated. In most HLLs the notion of a statement itself is very flexible, and often allows *recursion*, making nested constructs valid. These languages usually support multiple data types and often allow programmers to define abstract data types to be used in the programs.

3. ***Intermediate Code Generation***

Having recognized a given input program as valid, a compiler tries to create the equivalent program in the language of the target environment. In case of an HLL, it is futile to try to associate a single machine opcode for each statement of the input language. One of the reasons for this is, as stated above, the extent of a statement is not always fixed and may contain recursion. Moreover, data references in HLL programs can assume significant levels of abstractions in comparison to what the target execution environment may directly support.

4. ***Code Optimization***

The programs represented in the intermediate code form usually contains much scope for optimization both in terms of storage space as well as run time efficiency of the intended output program. Sometimes the input program itself contains such scope. Besides that, the process of generating the intermediate code representation usually leaves much room for such optimization. Hence, compilers usually implement explicit steps to optimize the intermediate code.

5. ***Code Generation***

Finally, the compiler converts the (optimized) program in the intermediate code representation to the required machine language. It needs to be noted that if the program being translated by the compiler actually has dependencies on some external modules, and then *linking* has to be performed to the output of the compiler. These activities are independent of whether the input program was in HLL or assembly language.

INTERPRETERS

Interpreter is a program that executes instructions written in a high-level language. There are two ways to run programs written in a high-level language. The most common is to compile the program; the other method is to pass the program through an interpreter.

An interpreter is a program which translates statements of a program into machine code. It translates only one statement of the program at a time. It reads only one statement of program, translates it and executes it. Then it reads the next statement of the program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed. On the other hand, a compiler goes through the entire program and then translates the entire program into machine codes. A compiler is 5 to 25 times faster than an interpreter.

By the compiler, the machine codes are saved permanently for future reference. On the other hand, the machine codes produced by interpreter are not saved. An interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system which has limited memory space.

An interpreter is a computer program that is used to directly execute program instructions written using one of the many high-level programming languages. The interpreter transforms the high-level program into an intermediate language that it then executes, or it could parse the high-level source code and then performs the commands directly, which is done line by line or statement by statement.

Programming languages are implemented in two ways: interpretation and compilation. As the name suggests, an interpreter transforms or interprets a high-level programming code into code that can be understood by the machine (machine code) or into an intermediate language that can be easily executed as well. The interpreter reads each statement of code and then converts or executes it directly. In contrast, an assembler or a compiler converts a high-level source code into native (compiled) code that can be executed directly by the operating system.

In most cases, a compiler is more favorable since its output runs much faster compared to a line-by-line interpretation. However, since interpretation happens per line or statement, it can be stopped in the middle of execution to allow for either code modification or debugging. Both have their advantages and disadvantages and are not mutually exclusive; this means that they can be used in conjunction as most integrated development environments employ both compilation and translation for some high-level languages.

Since an interpreter reads and then executes code in a single process, it very useful for scripting and other small programs. As such, it is commonly installed on Web servers, which run a lot of executable scripts.

An interpreter is a program that reads and executes code. This includes source code, pre-compiled code, and scripts. Common interpreters include Perl, Python, and Ruby interpreters, which execute Perl, Python, and Ruby code respectively.

Interpreters and compilers are similar, since they both recognize and process source code. However, a compiler does not execute the code like and interpreter does. Instead, a compiler simply converts the source code into machine code, which can be run directly by the operating system as an executable program. Interpreters bypass the compilation process and execute the code directly.

Since interpreters read and execute code in a single step, they are useful for running scripts and other small programs. Therefore, interpreters are commonly installed on Web servers, which allows developers to run executable scripts within their WebPages. These scripts can be easily edited and saved without the need to recompile the code.

While interpreters offer several advantages for running small programs, interpreted languages also have some limitations. The most notable is the fact that interpreted code requires an interpreter to run. Therefore, without an interpreter, the source code serves as a plain text file rather than an executable program. Additionally, programs written for an interpreter may not be able to use built-in system functions or access hardware resources like compiled programs can. Therefore, most software applications are compiled rather than interpreted.

Interpreter Versus Compiler

#	COMPILER	INTERPRETER
1	Compiler works on the complete program at once. It takes the entire program as input.	Interpreter program works line-by-line. It takes one statement at a time as input.
2	Compiler generates intermediate code, called the object code or machine code.	Interpreter does not generate intermediate object code or machine code.
3	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs faster than interpreter.	Interpreter execute conditional control statements at a much slower speed.
4	Compiled programs take more memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, interpreted programs are more memory efficient.
5	Compile once and run anytime. Compiled program does not need to be compiled every time.	Interpreted programs are interpreted line-by-line every time they are run.
6	Errors are reported after the entire program is checked for syntactical and other errors.	Error is reported as soon as the first error is encountered. Rest of the program will not be checked until the existing error is removed.
7	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and reports errors as it encounters them.
8	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.
9	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug. This makes such languages an ideal choice for new students.
10	Examples of programming languages that use compilers: C, C++, COBOL	Examples of programming languages that use interpreters: BASIC, Visual Basic, Python, Ruby, PHP, Perl, MATLAB, Lisp

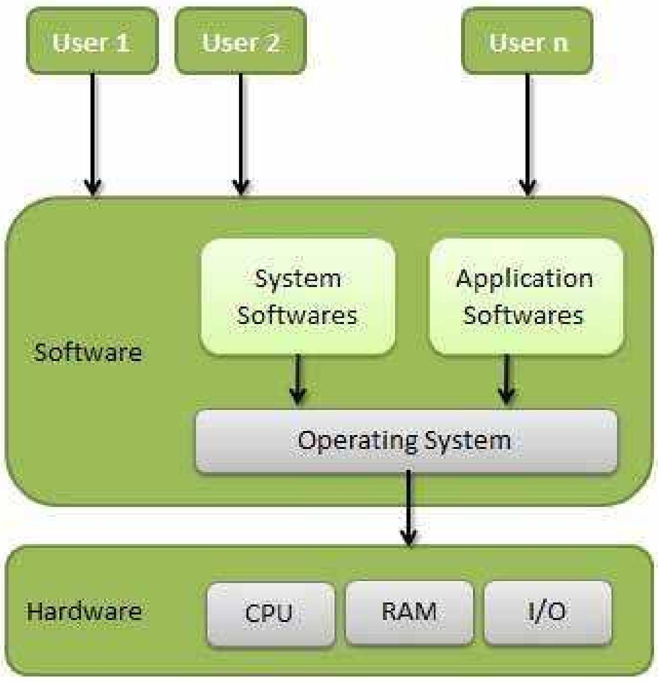
OPERATING SYSTEM

An operating system (OS) is the program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. The other programs are called applications or application programs. The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface such as a command line or a graphical user interface (GUI).

An operating system performs these services for applications:

1. Memory Management
2. Process Management
3. Processor Management
4. Device Management
5. File Management
6. Security
7. Control over system performance
8. Job accounting
9. Error detecting aids
10. Coordination between other software and users
11. Free Space management

All major computer platforms (hardware and software) require and sometimes include an operating system, and operating systems must be developed with different features to meet the specific needs of various form factors. Common desktop operating systems are Microsoft Windows, Apple's Mac OS, Linux Operating System, etc.



An Operating System (OS) is an interface between computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

➢ **Memory Management**

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address. Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

➢ **Processor Management**

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called process scheduling. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as traffic controller.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

➢ **Device Management**

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

- **File Management**

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

- **Other Important Activities**

Following are some of the important activities that an Operating System performs −

- Security − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- Control over system performance − Recording delays between request for a service and response from the system.
- Job accounting − Keeping track of time and resources used by various jobs and users.
- Error detecting aids − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- Coordination between other softwares and users − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Types of Operating Systems

Operating systems are there from the very first computer generation and they keep evolving with time. In this chapter, we will discuss some of the important types of operating systems which are most commonly used.

1. **Batch operating system**

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

The problems with Batch Systems are as follows −

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.

2. **Time-sharing operating systems**

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.
The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows −

- Provides the advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows −

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

3. **Distributed operating System**

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly. The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows −

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

4. Network operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows −

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows −

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

5. Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the response time. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc. There are two types of real-time operating systems.

Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

Soft real-time systems

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc.

6. Multi-Programming Operating System

As we know that in the Batch Processing System there are multiple jobs Execute by the System. The System first prepare a batch and after that he will Execute all the jobs those are Stored into the Batch. But the Main Problem is that if a process or job requires an Input and Output Operation, then it is not possible and second there will be the wastage of the Time when we are preparing the batch and the CPU will remain idle at that Time.

But With the help of Multi programming we can Execute Multiple Programs on the System at a Time and in the Multi-programming the CPU will never get idle, because with the help of Multi-Programming we can Execute Many Programs on the System and When we are Working with the Program then we can also Submit the Second or Another Program for Running and the CPU will then Execute the Second Program after the completion of the First Program. And in this we can also specify our Input means a user can also interact with the System.

The Multi-programming Operating Systems never use any cards because the Process is entered on the Spot by the user. But the Operating System also uses the Process of Allocation and De-allocation of the Memory Means he will provide the Memory Space to all the Running and all the Waiting Processes. There must be the Proper Management of all the Running Jobs.

7. **Multiprocessing Operating System:**

Generally a Computer has a Single Processor means a Computer have a just one CPU for Processing the instructions. But if we are Running multiple jobs, then this will decrease the Speed of CPU. For Increasing the Speed of Processing then we uses the Multiprocessing, in the Multi Processing there are two or More CPU in a Single Operating System if one CPU will fail, then other CPU is used for providing backup to the first CPU. With the help of Multi-processing, we can Execute Many Jobs at a Time. All the Operations are divided into the Number of CPU's. if first CPU Completed his Work before the Second CPU, then the Work of Second CPU will be divided into the First and Second.

DESIGN PROCEDURE OF A TWO-PASS ASSEMBLER

We need to follow a procedure that allows to design a two-pass assembler. The general design procedure of an assembler involves the following six steps:

1) Specification of the problem
2) Specification of the data structures
3) Defining the format of data structures
4) Specifying the algorithm
5) Looking for modularity that ensures the capability of a program to be subdivided into independent programming units
6) Repeating step 1 to 5 on modules for accuracy and checking errors.

The operation of a two-pass assembler can be explained with the help of the following functions that are performed by an assembler while translating a program:

- It replaces symbolic addresses with numeric addresses.
- It replaces symbolic operation codes with machine operation codes.
- It reserves storage for instructions and data.
- It translates constants into machine representation.

A two-pass assembler includes two passes, Pass1 and Pass 2. Therefore, it is named as two-pass assembler. In Pass 1 of the two-pass assembler, the assembler reads the assembly source program. Each instruction in the program is processed and is then translated to the generator. These translations are accumulated in appropriate tables, which help fetch a stored value. In addition to this, an intermediate form of each statement is generated. Once these translations have been made, Pass 2 starts its operation.

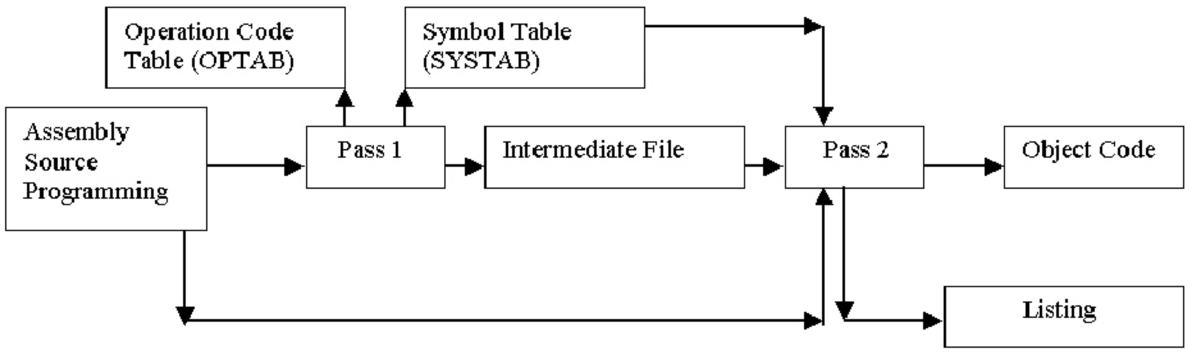


***Figure**: The Functioning of a Two-Pass Assembler*

Pass 2 examines each statement that has been saved in the file containing the intermediate program. Each statement in this file searches for the translations and then assembles these translations into the machine language program. The primary aim of the first pass of a two-pass assembler is to draw up a symbol table. Once Pass 1 has been completed, all necessary information on each user-defined identifier should have been recorded in this table. A Pass 2 over the program then allows full assembly to take place quite easily. Refer to the symbol table whenever it is necessary to determine an address for a named label, or the value of a named constant. The first pass can also perform some error checking. The above figure shows the steps followed by a two-pass assembler:

The table stores the translations that are made to the program. The table also includesoperation table and symbol table. The operation table is denoted as OPTAB. The OPTAB table contains:

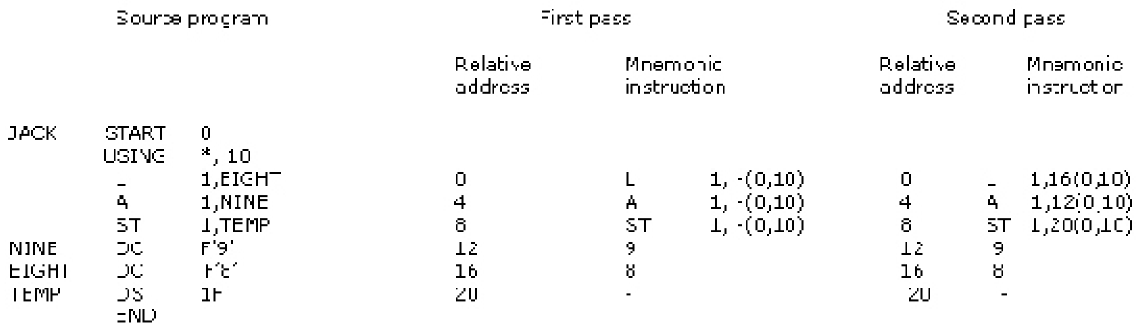
- Content: mnemonic, machine code (instruction format, length), etc.
- Characteristic: static table, and
- Implementation: array or hash table, easy for search.

An assembler designer is responsible for creating OPTAB table. Once the table is created, an assembler can use the tables contained within OPTAB, known as sub tables. These sub-tables include mnemonics and their translations. The introduction of mnemonics for each machine instruction is subsequently gets translated into the machine language for the convenience of the programmers. There are four different types of mnemonics in an assembly language:

1) Machine operations such as ADD, SUB, DIV, etc.
2) Pseudo-operations or directives: Pseudo-ops are the data definitions.
3) Macro-operation definitions.
4) Macro-operation call, which includes calls such as PUSH and LOAD.

Another type of tables used in the two-pass assembler is the symbol table. The symbol table is denoted by SYMTAB. The SYMTAB table contains:

- Content: label name, value, flag, (type, length), etc.
- Characteristic: dynamic table (insert, delete, search), and
- Implementation: hash table, non-random keys, hashing function.



Source program			First pass			Second pass		
			Relative address	Mnemonic instruction		Relative address	Mnemonic instruction	
JACK	START	0						
	USING	*, 10						
	L	1,EIGHT	0	L	1, -(0,10)	0	L	1,16(0,10)
	A	1,NINE	4	A	1, -(0,10)	4	A	1,12(0,10)
	ST	1,TEMP	8	ST	1, -(0,10)	8	ST	1,20(0,10)
NINE	DC	F'9'	12	9		12	9	
EIGHT	DC	F'8'	16	8		16	8	
TEMP	DS	1F	20	-		20	-	
	END							

***Figure**: Implementation of Two-Pass Assembler*

This table stores the symbols that are user-defined in the assembly program. These symbols may be identifiers, constants or labels. Another specification for symbol tables is that these tables are dynamic and we cannot predict the length of the symbol table. The implementation of SYMTAB includes arrays, linked lists and a hash table. The above figure shows the implementation of a two-pass assembler in which we translate a program.

We can now notice from the code shown that JACK is the name of the program. We start from the START instruction and come to know that it is a pseudo-op instruction, which is instructing the assembler. The next instruction in the code is the USING pseudo-op, which informs the assembler that register 10 is the base register and at the execution time, USING pseudo-op contains the address of the first instruction of the program. Next in the code is the load instruction: L 1,EIGHT and to execute this instruction, the assembler needs to have the address of EIGHT. Since, no index register is being used, therefore, we can place a 0 in the relative address for the index register.

The next instruction is an ADD instruction. The offset for NINE is still not known. Similar is the case with the next STORE instruction. We will notice that whenever an instruction is executed, the relative address gets incremented. A location counter is maintained by the processor, which indicates that the relative address of an instruction is being processed. Here, the counter is incremented by 4 in each of the instruction as the length of a load instruction is 4. The DC instruction is a pseudo-op that asks for the definition of data. For example, for NINE, a ‘9’ is the output and an ‘8’ for EIGHT.

In the instruction, DC F’9’, the word ‘9’ is stored at the relative location 12, as the location counter is having the value 12 currently. Similarly, the next instruction with label EIGHT that holds the relative address with value 16 and label TEMP is associated with the counter value 20. This completes the description of the column 2 of the above-mentioned code.