System Software

There are two broad categories of software:

System Software
Application Software

System Software is a set of programs that manage the resources of a compute system. System Software is a collection of system programs that perform a variety of functions.
File Editing
Resource Accounting
I/O Management
Storage, Memory Management access management.

System Software can be broadly classified into three types as:

System control programs controls the execution of programs, manage the storage & processing resources of the computer & perform other management & monitoring function. The most important of these programs is the operating system. Other examples are database management systems (DBMS) & communication monitors.
System support programs provide routine service functions to the other computer programs & computer users: E.g. Utilities, libraries, performance monitors & job accounting.
System development programs assists in the creation of application programs. E.g., language translators such as BASIC interpreter & application generators.

Application Software:

It performs specific tasks for the computer user. Application software is a program which program written for, or, by, a user to perform a particular job.
Languages already available for microcomputers include Clout, Q & A and Savvy ret rival (for use with Lotus 1-2-3).
The use of natural language touches on expert systems, computerized collections of the knowledge of many human experts in a given field, and artificial intelligence, independently smart computer systems – two topics that are receiving much attention and development and will continue to do so in the future.

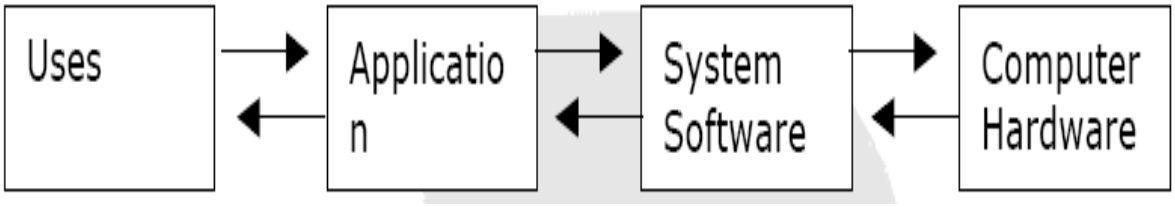
1.Operating System Software
Storage Manager
Process Manager
File – System Manager
I/O Control System
Communication Manager

2. Standard System Software
Language Processor
Loaders
Software Tools

3. Application Software
Sort/Merge Package
Payroll/Accounting Package
DBMS

General-purpose application software such as electronic spreadsheet has a wide variety of applications. Specific – purpose application s/w such as payroll & sales analysis is used for the application for which it is designed
Application programmer writes these programs. Application programmer writes these programs.

Generally computer users interact with application software. Application and system software act as interface between users & computer hardware. An application & system software become more capable, people find computer easier to use.



The Interaction between Users, Application Software, System Software & Computer Hardware:

System Software controls the execution of the application software & provides other support functions such as data storage. E.g. when you use an electronic spreadsheet on the computer, MS-DOS, the computer's Operating System, handles the storage of the worksheet files on disk.
The language translators and the operating system are themselves programs. Their function is to get the users program, which is

written, in a programming language to run-on the computer system.
All sucl. Programs, which help in the execution of user programs, are called system programs (SPs). The collection of such SPs is the “System Software” of a particular computer system.
Mast computer systems have support software, called Utility Programs, which perform routine tasks. These programs sort data, copy data from one storage medium to another, o/p data from a storage medium to printer & perform other tasks.

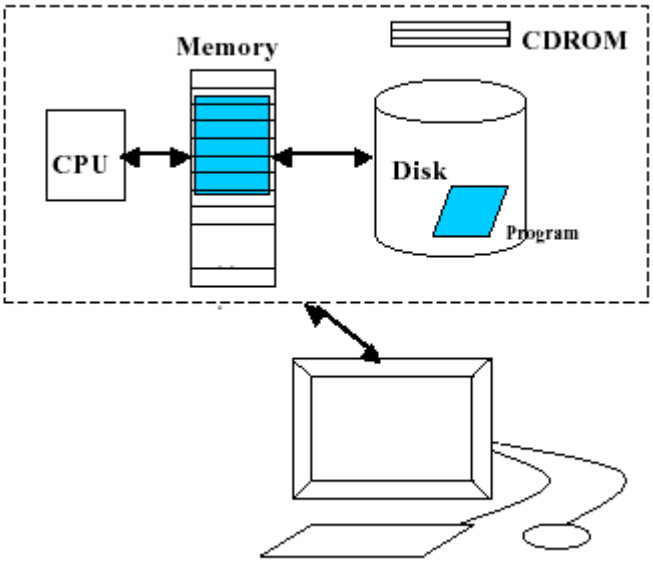
the execution at a specified starting address.

Typical Computer Structure

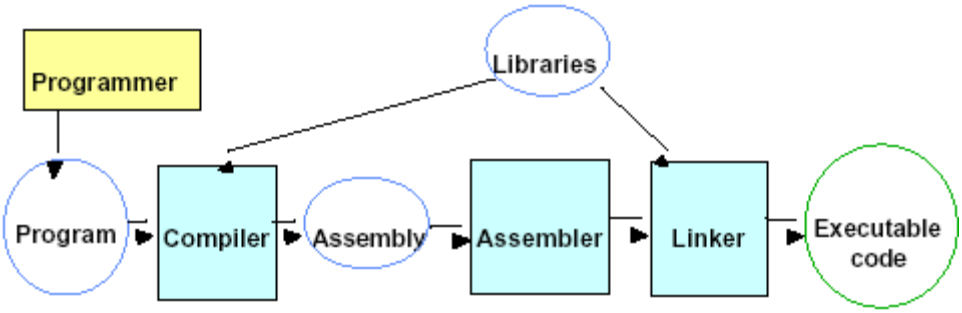
Main components:
- CPU
- Main Memory
- Secondary: disk
- IO devices:

  Input:
  - Keyboard
  - Mouse
  - CDROM

  Output:
  - Display
  - Printer



"Building" a Program



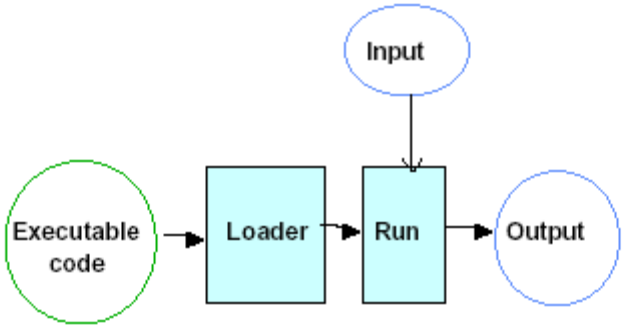

```
z = x + y;
```

```
load    r1, x
load    r2, y
add     r3, r1, r2
store   r3, z
```

```
1100 0110
1010 1111
0101 1000
1010 1111
0101 1000
0000 1001
1100 0110
0000 1001
```

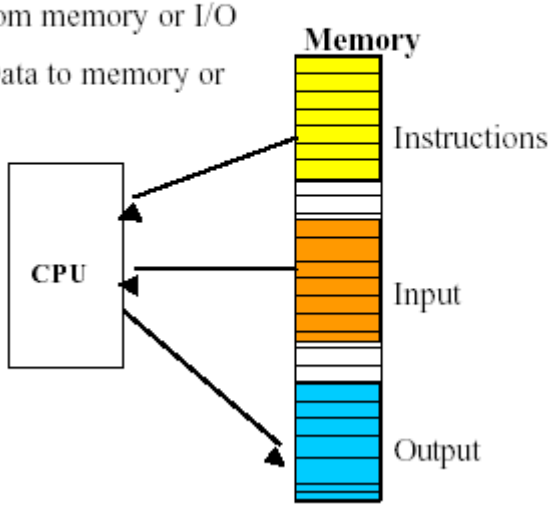
Running the Program

- *Loader* puts the program into the computer memory
- Running the program is done by an Operating System command
- Input are read from the I/O devices and from Memory
- Output is written to I/O devices and to Memory

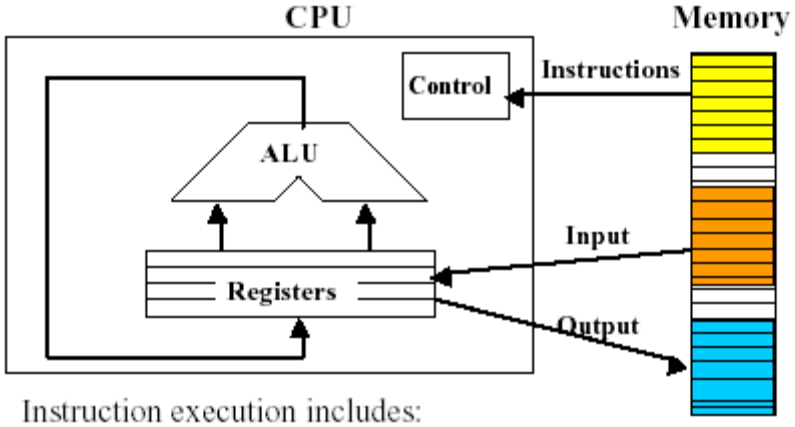


Execution of a Program

- CPU executes the Instructions
- CPU reads Input Data from memory or I/O
- CPU stores the Output Data to memory or to I/O



Execution of the Instructions



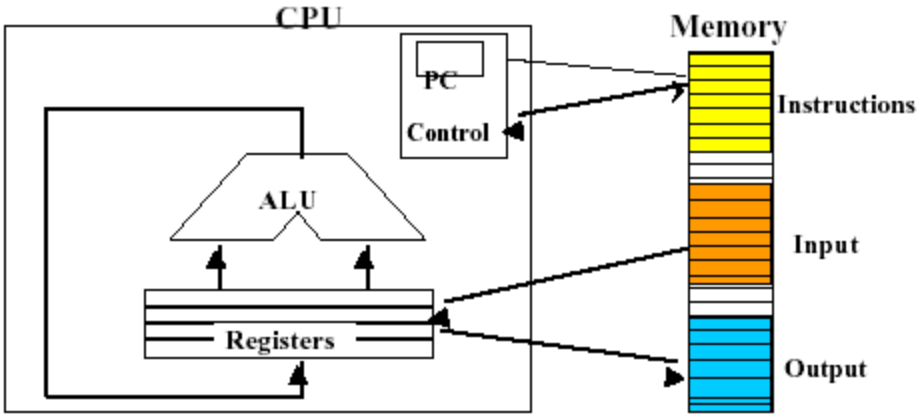
Instruction execution includes:

1. Load instruction from memory
2. Decode instruction
3. Load data from memory/registers
4. Execute the operation
5. Store result in register/memory

z = x + y;

```
load    r1, x
load    r2, y
add     r3, r1, r2
store   r3, z
```

Loading Instruction



To load an **instruction**:

- Need an "instruction address"
- Pointed by ***PC*** (Program Counter)

Next instruction is usually in ***PC + 4***, except for jumps

System Development Software:

System Development Software assists a programmer of user in developing & using an application program.
E.g. Language Translators
Linkage Editors
Application generators

Language Translators:

A language translator is a computer program that converts a program written in a procedural language such as BASIC into machine language that can be directly executed by the computer.
Computers can execute only machine language programs. Programs written in any other language must be translated into a machine language load module, which is suitable for loading directly into primary storage.
Subroutine or subprograms, which are stored on the system residence device to perform a specific standard function. E.g. if a program required the calculation of a square root,
Programmer would not write a special program. He would simply call a square root, subroutine to be used in the program.
Translators for a low-level programming language were assemblers

Language processors

Language Processing Activities

Language Processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behaviour of a software and the manner in which these ideas are implemented in a computer system.
the interpreter is a language translator. This leads to many similarities between are

Translators and interpreters. From a practical viewpoint many differences also exist

between translators and interpreters.

The absence of a target program implies the absence of an output interface the interpreter. Thus the language processing activities of an interpreter cannot be separated from its program execution activities. Hence we say that an interpreter 'executes' a program written in a PL.

Problem Oriented and Procedure Oriented Languages:

The three consequences of the semantic gap mentioned at the start of this section are in fact the consequences of a specification gap. Software systems are poor in quality and require large amounts of time and effort to develop due to difficulties in bridging the specification gap. A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain.

Such PLs can only be used for specific applications; hence they are called *problem*-oriented *languages.* They have large execution gaps, however this is acceptable because the gap is bridged by the translator or interpreter and does not concern the software designer.

A *procedure*-oriented *language* provides general purpose facilities required in most application domains. Such a language is independent of specific application domains.

The fundamental language processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap. We name these activities as
1. Program generation activities
2. Program execution activities.

A program generation activity aims at automatic generation of a program. The source languages specification language of an application domain and the target language is typically a procedure oriented PL. A Program execution activity organizes the execution of a program written in a PL on computer system. Its source language could be a procedure-oriented language or a problem oriented language.

Program Generation

The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL. In effect, the program generator

introduces a new domain between the application and PL domains we call this the *program generator domain.* The specification gap is now the gap between the application domain and the program generator domain. This gap is smaller than the gap between *the* application domain and the target PL domain.
Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.
The harder task of bridging the gap to the PL domain is performed by the generator.
This arrangement also reduces the testing effort. Proving the correctness of the pro-
gram generator amounts to proving the correctness of the transformation .
This would be performed while implementing the generator. To test an application generated by using the generator, it is necessary to only verify the correctness of the specification input to the program generator. This is a much simpler task than verifying correctness of the generated program. This task can be further simplified by providing a good diagnostic (i.e. error indication) capability in the program generator, which would detect inconsistencies in the specification.
It is more economical to develop a program generator than to develop a problem-oriented language. This is because a problem-oriented language suffers a very large execution gap between the PL domain and the execution domain whereas the program generator has a smaller semantic gap to the target PL domain, which is the domain of a standard procedure oriented language. The execution gap between the target PL domain and the execution domain is bridged by the compiler or interpreter for the PL.

Program Execution
Two popular models for program execution are translation and interpretation.

Program translation
The program translation model bridges the execution gap by translating a program written in a PL, called the *source program* (SP), into an equivalent program in the machine or assembly language of the computer system, called the *target program* (TP)
Characteristics of the program translation model are:

A program must be translated before it can be executed.
• The translated program may be saved in a file. The saved program may be executed repeatedly.
• A program must be retranslated following modifications.

Program interpretation

The interpreter reads the source program and stores it in its memory. During interpretation it takes a source statement, determines its meaning and performs actions which implement it. This includes computational and input-output actions.
The CPU uses a *program counter* (PC) to note the address of the next instruction to be executed. This instruction is subjected to the *instruction execution cycle* consisting of the following steps:

1. Fetch the instruction.
2. Decode the instruction to determine the operation to be performed, and also its operands.
3. Execute the instruction.

At the end of the cycle, the instruction address in PC is updated and the cycle is repeated for the next instruction. Program interpretation can proceed in an analogous manner. Thus, the PC can indicate which statement of the source program is to be interpreted next. This statement would be subjected to the *interpretation cycle,* which could consist of the following steps:

Fetch the statement.

Analyze the statement and determine its meaning, viz. the computation to be performed and its operands.

Execute the meaning of the statement.
From this analogy, we can identify the following characteristics of interpretation:
The source program is retained in the source form itself, i.e. no target program form exists, A statement is analyzed during its interpretation.

Comparison
A fixed cost (the translation overhead) is incurred in the use of the program translation model. If the source program is modified, the translation cost must be incurred again irrespective of the size of the modification. However, execution of the target program is efficient since the target program is in the machine language. Use of the interpretation model does not incur the translation overheads. This is advantageous if a program is modified between executions, as in program testing and debugging.

FUNDAMENTALS OF LANGUAGE PROCESSING

Definition

Language Processing = Analysis of SP + Synthesis of TP.

Definition motivates a generic model of language processing activities.

We refer to the collection of language processor components engaged in analyzing a source program as the *analysis phase* of the language processor. Components engaged in synthesizing a target program constitute the *synthesis phase.*

A specification of the source language forms the basis of source program analysis. The specification consists of three components:

1. *Lexical rules*, which govern the formation of valid lexical units in the source language.
2. *Syntax rules* which govern the formation of valid statements in the source language.
3. *Semantic rules* which associate meaning with valid statements of the language.

The analysis phase uses each component of the source language specification to determine relevant information concerning a statement in the source program. Thus, analysis of a source statement consists of lexical, syntax and semantic analysis.

The synthesis phase is concerned with the construction of target language statement(s) which have the same meaning as a source statement. Typically, this consist of two main activities:

- Creation of data structures in the target program
- Generation of target code.

We refer to these activities as *memory allocation* and *code generation,* respectively

Lexical Analysis (Scanning)

Lexical analysis identifies the lexical units in a source statement. It then classifies the units into different lexical classes e.g. id's, constants etc. and enters them into different tables. This classification may be based on the nature ofstring or on the specification of the source language. (For example, while an integer constant is a string of digits with an optional sign, a reserved id is an id whose name matches one of the reserved names mentioned in the language specification.) Lexical analysis builds a descriptor, called a *token,* for each lexical unit. A token contain two fields—*class code,* and *number in class, class code* identifies the class to which a lexical unit belongs, *number in class* is the entry number of the lexical unit in the relevant table.

Syntax Analysis (Parsing)

Syntax analysis processes the string of tokens built by lexical analysis to determine the statement class, e.g. assignment statement, if statement, etc. It then builds an IC which represents

the structure of the statement. The IC is passed to semantic analysis to determine the meaning of the statement.

Semantic analysis

Semantic analysis of declaration statements differs from the semantic analysis of imperative statements. The former results in addition of information to the symbol table, e.g. type, length and dimensionality of variables. The latter identifies the sequence of actions necessary to implement the meaning of a source statement. In both cases the structure of a source statement guides the application of the semantic rules. When semantic analysis determines the meaning of a sub tree in the IC. It adds information a table or adds an action to the sequence. It then modifies the IC to enable further semantic analysis. The analysis ends when the tree has been completely processed.

"FUNDAMENTALS OF LANGUAGE SPECIFICATION

A specification of the source language forms the basis of source program analysis. In this section, we shall discuss important lexical, syntactic and semantic features of a programming language.

Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. This section discusses key concepts and notions from formal language grammars. A language L can be considered to be a collection of valid sentences.

Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as *a. formal language.* A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal symbols, alphabet and strings

The *alphabet* of L, denoted by the Greek symbol Z, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols in Z.

A symbol in the alphabet is known as a *terminal symbol* (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g. $\Sigma \cong$ {a, b, ….z, 0,1....9}

Here the symbols {, ',' and} are part of the notation. We call them *met symbols* to differentiate them from terminal symbols. Throughout this discussion we assume that met symbols are distinct from the terminal symbols. If this is not the case, i.e. if a terminal symbol and a met symbol are identical, we enclose the terminal symbol in quotes to differentiate it from the metasymbol. For

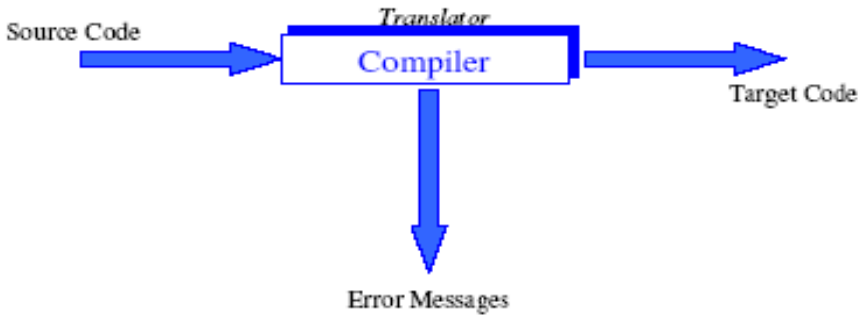
example, the set of punctuation symbols of English can be defined as {:,;','-,...} Where ',' denotes the terminal symbol 'comma'.
A *string* is a finite sequence of symbols. We will represent strings by Greek symbols-α β γ, etc. Thus α = axy is a string over Σ . The length of a string is the Number of symbols in it. Note that the absence of any symbol is also a string, the *null string* . The *concatenation* operation combines two strings into a single string.
"

To evaluate an HLL program it should be converted into the Machine language. A compiler performs another very important function. This is in terms of the diagnostics.
I.e. error – detection capability.
The important tasks of a compiler are:
Translating the HLL program input to it.
Providing diagnostic messages whenever specifications of the HLL

Compilers

Introduction

Compiler is tool: which translate notations from one system to another, usually from source code (high level code) to machine code (object code, target code, low level code).



• A compiler is a **program** that translates a **sentence**

a. from a **source** language (e.g. Java, Scheme, LATEX)
b. into a **target** language (e.g. JVM, Intel x86, PDF)
c. while preserving its **meaning** in the process

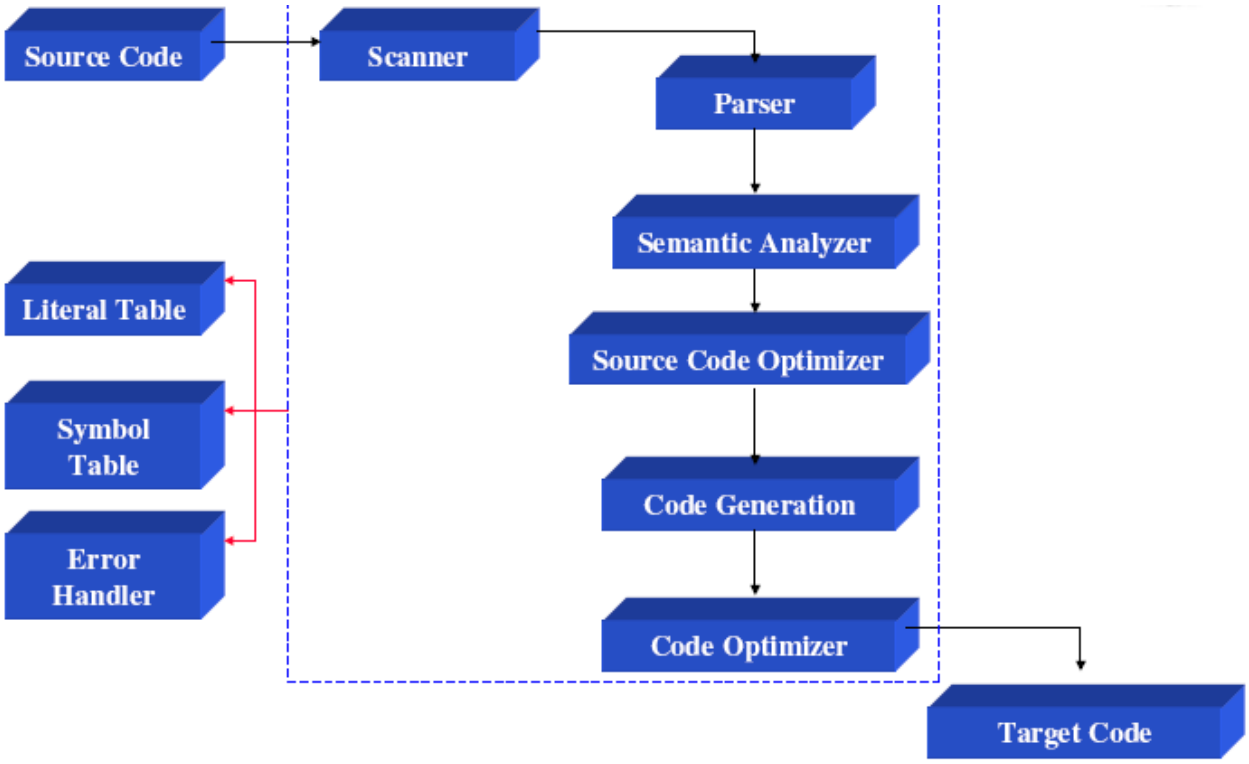
• Compiler design has a **long** history (FORTRAN 1958)

a. lots of **experience** on how to structure compilers
b. lots of **existing** designs to study (many freely available)
c. take CS 152: Compiler Design for **some** of the details. . .

- We use natural languages to communicate
- We use programming languages to speak with computers

Components of a Compiler

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation



The Scanner

a. Performs **lexical analysis**

b. Collects character sequences into **tokens**

c. Example: **a[index] = 4 + 2**

Tokens:		
	a	identifier
	[	left bracket
	index	identifier
	]	right bracket
	=	assignment
	4	number
	+	plus sign
	2	number

The scanner *may* also:

1. Enter identifiers into the symbol table
2. Enter literals (numeric constants and strings) into the literal table

Lexical Analysis (LA)

Maradona kicks the ball

Token Generation:

- Maradona
- kicks
- the
- ball

Who performs this ?

Lexical Analysis

X = Y + 30

Token Generation:

- X id_1
- = operator
- Y id_2
- + operator
- 30 literal/constant

What other functions Scanner can perform ?

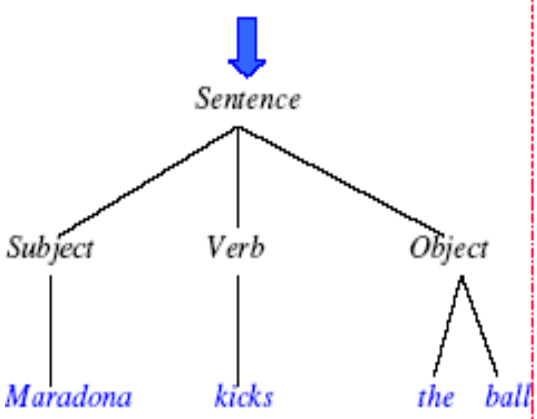
Syntax Analysis (SA)

The Parser
a. Performs **syntax analysis**
b. Builds a **parse tree** or **syntax tree**

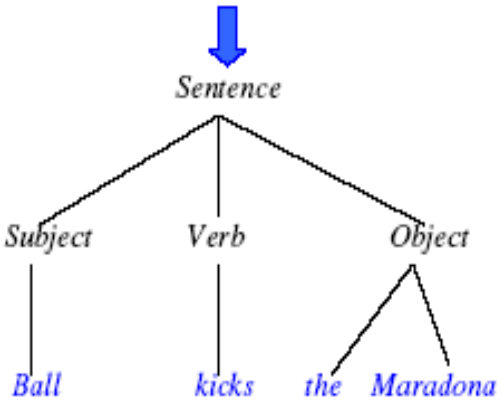
Parse tree for **a[index] = 4 + 2**

Structure of the program is determined by SA. Some thing similar to grammatical analysis.

Maradona kicks the ball

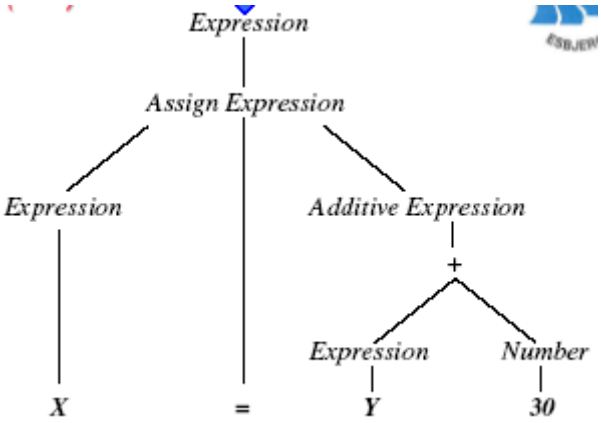


Ball kicks the Maradona



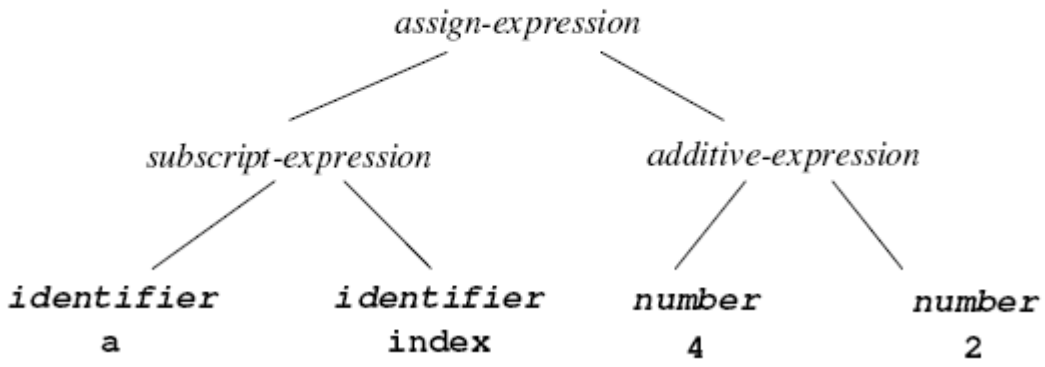
An **abstract syntax tree** is a more concise version of a parse tree.

```
X = Y + 30
```



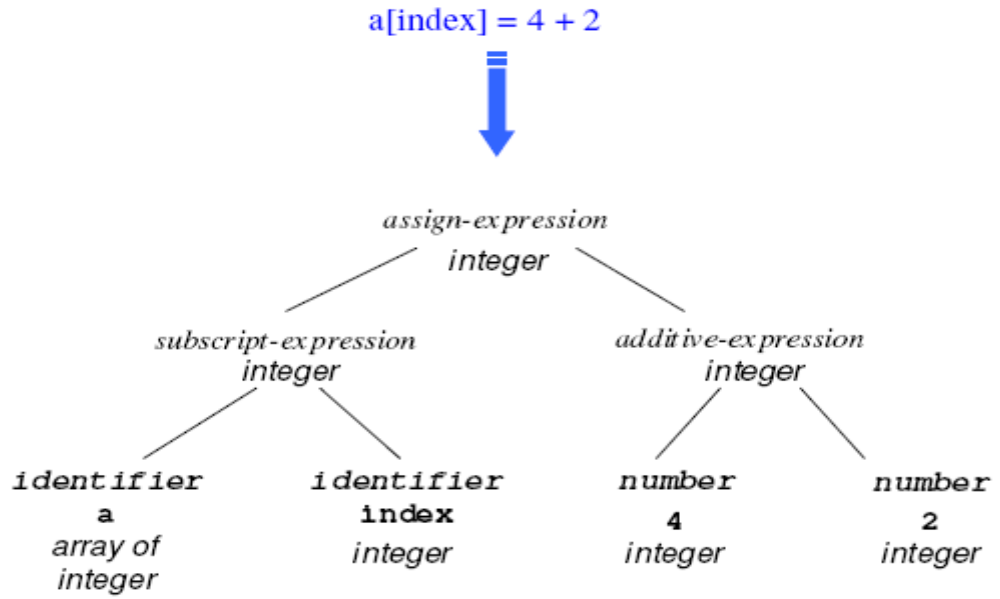
Some time syntax tree is also called as Abstract Syntax tree and could be a "trimmed" version of the parse tree with only essential information:

For Example: a[index] = 4 + 2



Semantic Analyzer

This attach meaning of tokens; For example to the same expression



Intermediate Code Generation

Same example: X = Y + 30

Temp1 = 30

Temp2 = Y

Temp3 = Temp2 + Temp1

X = Temp3

Code Optimization

Same example: X = Y + 30

Temp1 = 30

Temp2 = Y

X = Temp2 + Temp1

Code Generation

Same example: X = Y + 30

Movei Y, r1

Addi 30, r1

Movei r1, X

Algorithmic Tools

- **Token:**
  - Using Regular Expressions.
- **Scanner:**
  - Implementation of finite state machine to recognize tokens.
- **Parser:**
  - An Automaton (i.e. uses a stack), based on grammar rules in a standard format (BNF -- Backus Naur Form).
- **Semantic Analyzer and Code Generator:**
  - Recursive evaluators based on semantic rules for attributes (properties of language constructs).

Error handling

- One of the difficult part of a compiler to design.
- Must handle a wide range of errors
- Must handle multiple errors.
- Must not get stuck.
- Must not get into an infinite loop.

Kinds of errors

- **Syntax**:

```
if (x == 0) y + = z + r; }
```

- **Semantic**:

```
int x = "Hello, world!";
```

- **Runtime**:

```
int x = 2;
...
double y = 3.14159 / (x - 2);
```

Error Handling Requirements

- A compiler must handle syntax and semantic errors, but not runtime errors (whether a runtime error will occur is million dollar question).

- Sometimes a compiler is required to generate code to catch runtime errors and handle them in some graceful way (either with or without exception handling). This, too, is often difficult.

Major Compiler Data Structures

a. Tokens
   1. Represented as an enumerated type
   2. May require other information:
      a. Spelling of identifier
      b. Numeric value
   3. Scanner needs generate only one token at a time (single symbol lookahead)

b. Syntax Tree
   1. A linked structure built by the parser, with information added by the semantic analyzer
   2. Each node is a record whose fields contain information about the syntactic construct which the node represents
   3. Node may be represented using a variant record

c. Symbol Table
   1. Keeps information about identifiers
   2. Efficient insertion and lookup is required → hash table or tree structure may be used
   3. Several tables may be maintained in a list or stack

d. Literal Table

1. Stores constants and strings used in a program
2. Data in literal table applies globally to a program → deletions are not necessary

e. Intermediate code
1. Could be kept in an array, temporary file, or linked list
2. Representations include P-code and 3-address code

f. Temporary files
1. May not be used if memory constraints are not a problem
2. **Backpatching** of addresses necessary during translation

Assemblers & compilers

Assembler is a translator for the lower level assembly language of computer, while compilers are translators for HLLs.

An assembly language is mostly peculated to a certain computer, while an HLL is generally machined independent & thus portable.

Overview of the compilation process:

The process of compilation is:

Analysis of + Synthesis of = Translation of Source Text Target Text Program

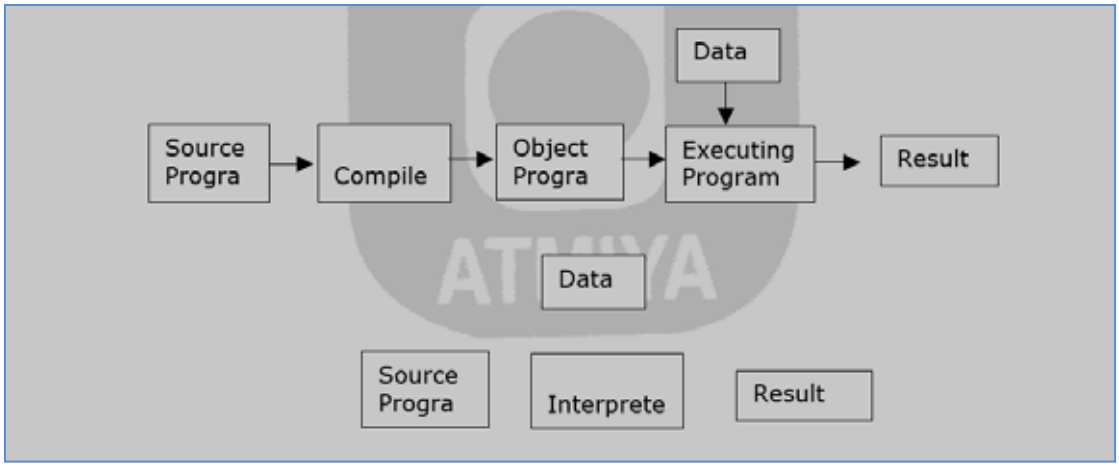
Source text analysis is based on the grimmer of the source of the source language.

The component sub – tasks of analysis phase are:
Syntax analysis, which determine the syntactic structure of the source statement.
Semantic analysis, which determines the meaning of a statement, once its grammatical structures become known.

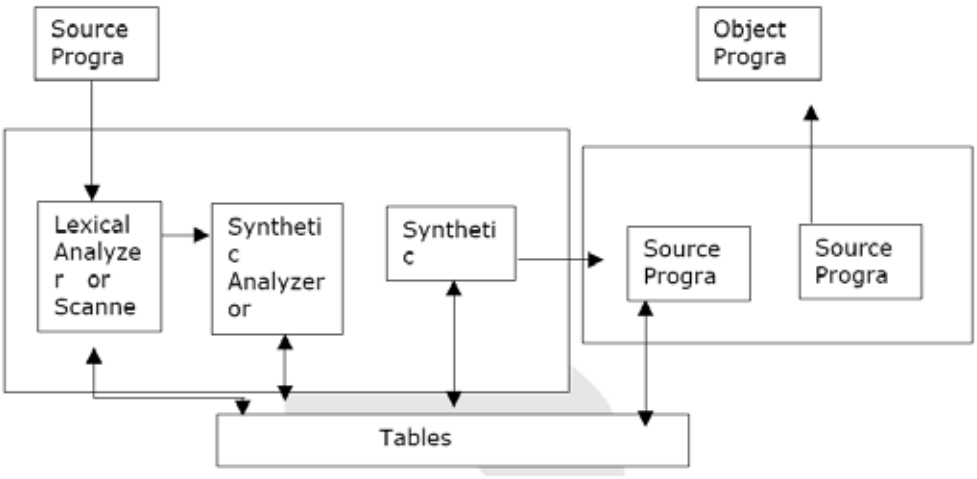
The analysis phase

The analysis phase of a compiler performs the following functions.
Lexical analysis
Syntax analysis
Semantic analysis
Syntax analysis determines the grammatical or syntactic structure or the input statement & represents it in an intermediate form from which semantic analysis can be performed.
A compiler must perform two major tasks:
The Analysis of a source program & the synthesis of its corresponding object program.

The analysis task deals with the decomposition of the source program into its basic parts using these basic parts the synthesis task builds their equivalent object program modules. A source program is a string of symbols each of which is generally a letter, a digit or a certain special constants, keywords & operators. It is therefore desirable for the compiler to identify these various types as classes.



The source program is input to a lexical analyzer or scanner whose purpose is to separate the incoming text into pieces or tokens such as constants, variable name, keywords & operators.
In essence, the lexical analyzer performs low- level syntax analysis performs low-level syntax analysis.
For efficiency reasons, each of tokens is given a unique internal representation number.
TEST: If A > B then X=Y;



The lexical analyzer supplies tokens to the syntax analyzer.
The syntax analyzer is much more complex then the lexical analyzer its function is to take the source program from the lexical analyzer & determines the manner in which it is to be decomposed into its constituent parts. That is, the syntax analyzer determines the overall structure of the source program.
The semantic analyzer uses syntax analyzer.
The function of the semantic analyzer is to determine the meaning the meaning (or semantics) of the source program.
The semantic analyzer is passed on to the code generators.
At this point the intermediate form of the source language programs usually translated to either assembly language or machine language.
The output of the code generator is passed on to a code optimizer. It's purpose to produce more program.

Introduction to Assemblers and Assembly Language

Encoding instructions as binary numbers is natural and efficient for computers. Humans, however, have a great deal of difficulty understanding and manipulating these numbers. People read and write symbols (words) much better than long sequences of digits. This lecture describes the process by which a human-readable program is translated into a form that a computer can execute, provides a few hints about writing assembly programs, and explains how to run these programs on SPIM,

What is an assembler ?

A tool called an ***assembler*** translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer's 0s and 1s that simplifies writing and reading programs. Symbolic names for operations and locations are one facet of this representation. Another facet is programming facilities that increase a program's clarity.

An assembler reads a single assembly language ***source file*** and produces an ***object file*** containing machine instructions and bookkeeping information that helps combine several object files into a program. Figure (1) illustrates how a program is built. Most programs consist of several files—also called ***modules***— that are written, compiled, and assembled independently. A program may also use prewritten routines supplied in a ***program library*** . A module typically contains *References* to subroutines and data defined in other modules and in libraries. The code in a module cannot be executed when it contains *unresolved References* to labels in other object files or libraries. Another tool, called a *linker,* combines a collection of object and library files into an *executable file* , which a computer can run.

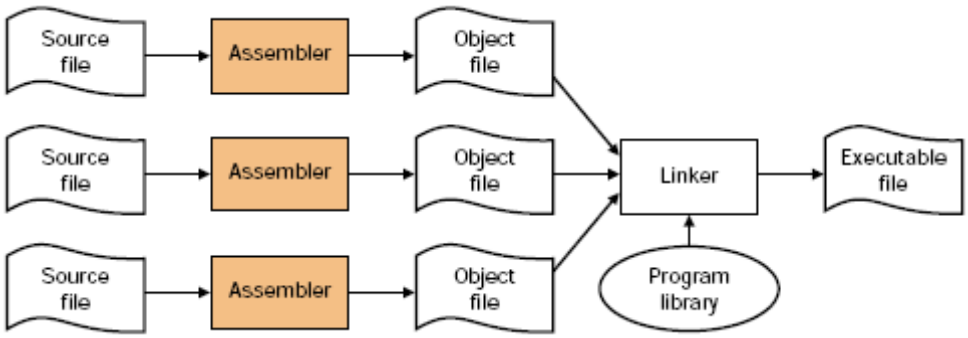


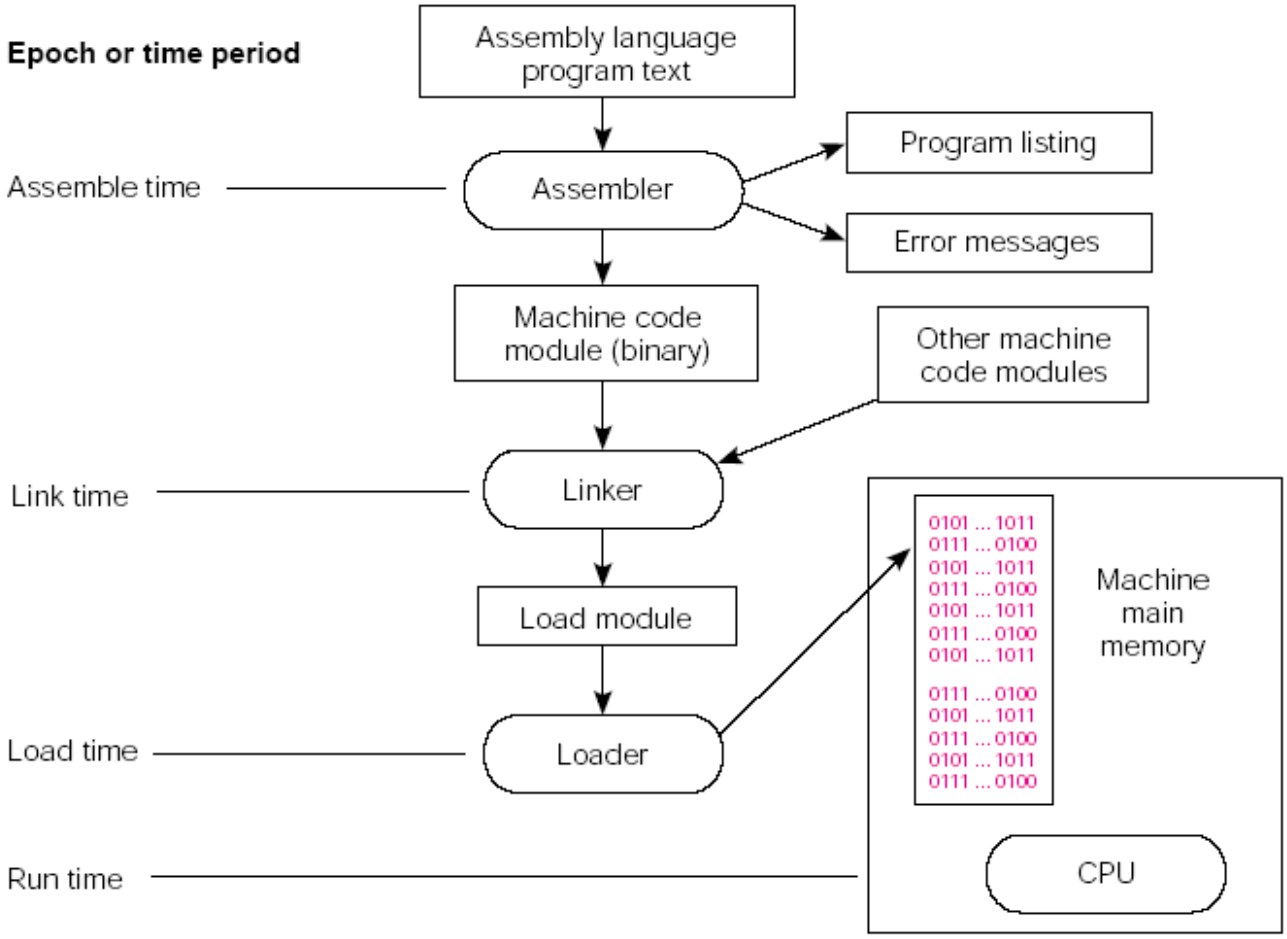
FIGURE 1: The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

1) **Assembler = a program to handle all the tedious mechanical translations**

2) **Allows you to use:**
   - symbolic opcodes
   - symbolic operand values
   - symbolic addresses

3) **The Assembler**
   - keeps track of the numerical values of all symbols
   - translates symbolic values into numerical values

4)Time Periods of the Various Processes in Program Development



5) The Assembler Provides:

a. Access to all the machine's resources by the assembled program. This includes access to the entire instruction set of the machine.
b. A means for specifying run-time locations of program and data in memory.
c. Provide symbolic labels for the representation of constants and addresses.
d. Perform assemble-time arithmetic.

e. Provide for the use of any synthetic instructions.
f. Emit machine code in a form that can be loaded and executed.
g. Report syntax errors and provide program listings
h. Provide an interface to the module linkers and program loader.
i. Expand programmer defined macro routines.

Assembler Syntax and Directives

Syntax: Label OPCODE Op1, Op2, ... ;Comment field

Pseudo-operations (sometimes called "pseudos," or directives) are "opcodes" that are actually instructions to the assembler and that do not result in code being generated.

Assembler maintains several data structures

- Table that maps text of opcodes to op number and instruction format(s)
- "Symbol table" that maps defined symbols to their value

Assembly program structure

- Each line is of the form:

Program line		
Label	Operation	Operands ; Comment

- Each field can be omitted
- The remainder of the line after (;) is ignored
- Example:

Comments

```
;; FIRST.ASM    Our first Assembly Language Program.
        .MODEL  SMALL
        .586    ; Allows Pentium Instructions. Must come after .MODEL

        .STACK  100h
        .....
```

Disadvantages of Assembly

• programmer must manage movement of data items between memory locations

and the ALU.

• programmer must take a "microscopic" view of a task, breaking it down to manipulate individual memory locations.
• assembly language is machine-specific.

• statements are not English-like (Pseudo-code)

Directives Assembler

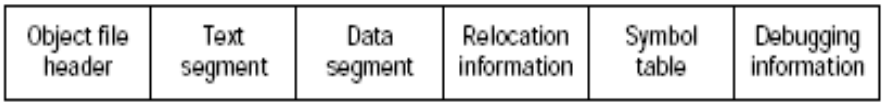
1. **Directives are commands to the Assembler**
2. **They tell the assembler what you want it to do, e.g.**
    a. Where in memory to store the code
    b. Where in memory to store data
    c. Where to store a constant and what its value is
    d. The values of user-defined symbols

Object File Format

Assemblers produce object files. An object file on Unix contains six distinct sections (see Figure 3):

- The *object file header* describes the size and position of the other pieces of the file.
- The *text segment* contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.

- The *data segment* contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The *relocation information* identifies instructions and data words that depend on absolute addresses. These references must change if portions of the program are moved in memory.
- The *symbol table* associates addresses with external labels in the source file and lists unresolved references.
- The *debugging information* contains a concise description of the way in which the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.



The assembler produces an object file that contains a binary representation of the program and data and additional information to help link pieces of a program. This relocation information is necessary because the assembler does not know which memory locations a procedure or piece of data will occupy after it is linked with the rest of the program. Procedures and data from a file are stored in a contiguous piece of memory, but the assembler does not know where this memory will be located. The assembler also passes some symbol table entries to the linker. In particular, the assembler must record which external symbols are defined in a file and what unresolved references occur in a file.

Macros

Macros are a pattern-matching and replacement facility that provide a simple mechanism to name a frequently used sequence of instructions.

Instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

The 2-Pass Assembly Process

- **Pass 1:**

1. Initialize location counter (assemble-time “PC”) to 0
2. Pass over program text: enter all symbols into symbol table
    a. May not be able to map all symbols on first pass
    b. Definition before use is usually allowed
3. Determine size of each instruction, map to a location

    a. Uses pattern matching to relate opcode to pattern
    b. Increment location counter by size
    c. Change location counter in response to ORG pseudos

- **Pass 2:**

1. Insert binary code for each opcode and value
2. “Fix up” forward references and variable-sizes instructions
    - Examples include variable-sized branch offsets and constant fields

Linker & Loader

A software processor, which performs some low level processing of the programs input to it, produces a ready to execute program form.

The basic loading function is that of locating a program in an appropriate area of the main store of a computer when it is to be executed.

A loader often performs the two other important functions.
The loader, which accepts the program form, produced by a translator & certain other program forms from a library to produce one ready – to – execute machine language program.
A unit of input to the loader is known as an object program or an object module.
The process of merging many object modules to from a single machine language program is known as linking.
The function to be performed by:
Assigning of loads the storage area to a program.
Loading of a program into the assigned area.
Relocations of a program to execute properly from its load-time storage area.
Linking of programs with one another.
Loader, linking loaders, linkage editors are used in software literature

LOADER:

The loader is program, which accepts the object program decks, prepares this program for execution by the computer and initializes the execution.
In particular the loader must perform four functions:
Allocate space in memory for the program (allocation).
Resolve symbolic references between objects decks (linking).
Adjust all address dependent locations, such as address constants, to correspond to the allocated space (relocation).
Physically place the machine instructions and data into memory (loading).

(Loaders and Linkers)

Introduction:

In this chapter we will understand the concept of linking and loading. As discussed earlier the source program is converted to object program by assembler. The loader is a program which takes this object program, prepares it for execution, and loads this executable code of the source into memory for execution.

Definition of Loader:

Loader is utility program which takes object code as input prepares it for execution and loads the executable code into the memory. Thus loader is actually responsible for initiating the execution process.

Functions of Loader:

The loader is responsible for the activities such as allocation, linking, relocation and loading

1) It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.

2) 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.

3) 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.

4) 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.

Loader Schemes:

Based on the various functionalities of loader, there are various types of loaders:

1) **“compile and go” loader:** in this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data isdirectly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77, it’s a FORTRAN compiler which uses such “load and go” scheme. This loading scheme is also called as “assemble and go”.

Advantages:

• This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.

Disadvantages:

• In this scheme some portion of memory is occupied by assembler which is simply a wastage of memory. As this scheme is combination of

assembler and loader activities, this combination program occupies large block of memory.

• There is no production of .obj file, the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled and executed each time, which then becomes a time consuming activity.

• It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.

• For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the "compile and go" loader cannot handle such programs.

• The execution time will be more in this scheme as every time program is assembled and then executed.

2) **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

Advantages:

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.

• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.

• It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.

3) **Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer orassembler should have knowledge of

memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is : specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next . modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

```
Line number
1 MAIN START 1000
. .
. .
. .
15 JMP 5000
16 STORE ;instruction at location 2000
END
1 SUM START 5000
2
20 JMP 2000
21 END
```

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN. Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of the mutually dependant dresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes

for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.

The entire process is modeled in the following figure.

Thus the absolute loader is simple to implement in this scheme-

l) Allocation is done by either programmer or assembler

2)Linking is done by the programmer or assembler

3)Resolution is done by assembler

4)Simply loading is done by the loader

As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

Advantages:

1. It is simple to implement
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient

Disadvantages:

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.

If at all any modification is done the some segments, the starting addresses of immediate next segments may get changed, the programmer has to take care of this issue and he needs to update the corresponding starting addresses on any modification in the source.

Algorithm for absolute Loader

Input: Object codes and starting address of program segments.

Output: An executable code for corresponding source program. This executable code is to be placed in the main memory

Method: Begin

```
For each program segment do Begin
Read the first line from object module to obtain
information about memory location. The starting address
say S in corresponding object module is the memory
location where executale code is to be placed.
```

```
Hence
Memory_location = S
Line counter = 1; as it is first line While (! end of
file)
For the curent object code do Begin
1. Read next line
2. Write line into location S
3. S = S + 1
4. Line counter Line counter + 1
```

Subroutine Linkage: To understand the concept of subroutine linkages, first consider the following scenario:

"In Program A a call to subroutine B is made. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler can not find the value of this symbolic reference and hence it will declare it as an error. To overcome problem, there should be some mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must -be declared as an external routine in A. To declare such subroutine asexternal, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. Similarly, if one subroutine or a variable is defined in the current segment and can be referred by other segments then those should be declared by using pseudo-ops INT. Thereby the assembler could inform loader that these are the subroutines or variables used by other segments. This overall process of establishing the relations between the subroutines can be conceptually called a_ subroutine linkage.

For example

```
MAIN START
EXT B
.
.
.
CALL B
.
.
END
B START
.
.
```

RET
END

At the beginning of the MAIN the subroutine B is declared as external. When a call to subroutine B is made, before making the unconditional jump, the current content of the program counter should be stored in the system stack maintained internally. Similarly while returning from the subroutine B (at RET) the pop is performed to restore the program counter of caller routine with the address of next instruction to be executed.

Concept of relocations:

Relocation is the process of updating the addresses used in the address sensitive instructions of a program. It is necessary that such a modification should help to execute the program from designated area of the memory.

The assembler generates the object code. This object code gets executed after loading at storage locations. The addresses of such object code will get specified only after the assembly process is over. Therefore, after loading, Address of object code = Mere address of object code + relocation constant.

There are two types of addresses being generated: Absolute address and, relative address. The absolute address can be directly used to map the object code in the main memory. Whereas the relative address is only after the addition of relocation constant to the object code address. This kind of adjustment needs to be done in case of relative address before actual execution of the code. The typical example of relative reference is : addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals, redefinable symbols. Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute.

The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.

For example

LOADA(X)+5

The expression A(X) means the address of variable X. The meaning of the above instruction is that loading of the contents of memory location which is 5 more than the address of variable X. Suppose if the address of X is 50 then by above command we try to get the memory location 50+5=55. Therefore as the address of variable X is relative A(X) + 5 is also relative. To calculate the relative addresses the simple expressions are allowed. It is expected that the expression should possess at the most

addition and multiplication operations. A simple exercise can be carried out to determine whether the given address is absolute or relative. In the expression if the address is absolute then put 0 over there and if address is relative then put lover there. The expression then gets transformed to sum of O's and l's. If the resultant value of the expression is 0 then expression is absolute. And if the resultant value of the expression is 1 then the expression is relative. If the resultant is other than 0 or 1then the expression is illegal. For example:

In the above expression the *A,* Band C are the variable names. The assembler is to c0l1sider the relocation attribute and adjust the object code by relocation constant. Assembler is then responsible to convey the information loading of object code to the loader. Let us now see how assembler generates code using relocation information.

Direct Linking Loaders

The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative. If at all the address is relative then it is the assembler who informs the loader about the relative addresses.

The assembler should give the following information to the loader

1)The length of the object code segment

2) The list of all the symbols which are not defined 111 the current segment but can be used in the current segment.

3) The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The USE table holds the information such as name of the symbol, address, address relativity.

The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table. The definition table holds the information such as symbol, address.

Overlay Structures and Dynamic Loading:

Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such

situations only those segments are resident in the memory that are actually needed at the time of execution But the question arises what will happen if the required segment is not present in the memory? Naturally the execution process will be delayed until the required segment gets loaded in the memory. The overall effect of this is efficiency of execution process gets degraded. The efficiency can then be improved by carefully selecting all the interdependent segments. Of course the assembler can not do this task. Only the user can specify such dependencies. The inter dependency of thesegments can be specified by a tree like structure called static overlay structures. The overlay structure contain multiple root/nodes and edges. Each node represents the segment. The specification of required amount of memory is also essential in this structure. The two segments can lie simultaneously in the main memory if they are on the same path. Let us take an example to understand the concept. Various segments along with their memory requirements is as shown below.

Automatic Library Search:

Previously, the library routines were available in absolute code but now the library routines are provided in relocated form that ultimately reduces their size on the disk, which in turn increases the memory utilization. At execution time certain library routines may be needed. Keeping track of which library routines are required and how much storage is required by these routines, if at all is done by an assembler itself then the activity of automatic library search becomes simpler and effective. The library routines can also make an external call to other routines. The idea is to make a list of such calls made by the routines. And if such list is made available to the linker then linker can efficiently find the set of required routines and can link the references accordingly.

For an efficient search of library routines it desirable to store all the calling routines first and then the called routines. This avoids wastage of time due to winding and rewinding. For efficient automated search of library routines even the dictionary of such routines can be maintained. A table containing the names of library routines and the addresses where they are actually located in relocatable form is prepared with the help of translator and such table is submitted to the linker. Such a table is called subroutine directory. Even if these routines have made any external calls the -information about it is also given in subroutine directory. The linker searches the subroutine directory, finds the address of desired library

routine (the address where the routine is stored in relocated form).Then linker prepares aload module appending the user program and necessary library routines by doing the necessary relocation. If the library routine contains the external calls then the linker searches the subroutine directory finds the address of such external calls, prepares the load module by resolving the external references. **Linkage Editor:** The execution of any program needs four basic functionalities and those are allocation, relocation, linking and loading. As we have also seen in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Moreover if the program contains many subroutines or functions and the program needs to be executed repeatedly then this activity becomes annoyingly complex .Each time for execution of a program, the allocation, relocation linking and -loading needs to be done. Now doing these activities each time increases the time and space complexity. Actually, there is no need to redo all these four activities each time. Instead, if the results of some of these activities are stored in a file then that file can be used by other activities. And performing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Thus dividing the essential four functions in groups reduces the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, creates linked executable text and stores this text in a file in some systematic manner. Such kind of module prepared by the binder execution is called load module. This load module can then be actually loaded in the main memory by the loader. This loader is also called as module loader. If the binder can produce the exact replica of executable code in the load module then the module loader simply loads this file into the main memory which ultimately reduces the overall time complexity. But in this process the binder should knew the current positions of the main memory. Even though the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the region of main memory available for loading the program is decided by the host operating system. The binder should also know which memory area is allocated to the loading program and it should modify the relocation information accordingly. The binder

which performs the linking function and produces adequate information about allocation and relocation and writes this information along with the program code in the file is called linkage editor. The module loader then accepts this rile as input, reads the information stored in and based on this information about allocation and relocation it performs the task of loading in the main memory. Even though the program is repeatedly executed the linking is done only once. Moreover, the flexibility of allocation and relocation helps efficient utilization of the main memory.

Direct linking: As we have seen in overlay structure certain selective subroutines can be resident in the memory. That means it is not necessary to resident all the subroutines in the memory for all the time. Only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory. This process of postponing linking and loading of external reference until execution is called dynamic linking. For example suppose the subroutine main calls A,B,C,D then it is not desirable to load A,B,C and D along with the main in the memory. Whether A, B, C or D is called by the main or not will be known only at the time of execution. Hence keeping these routines already before is really not needed. As the subroutines get executed when the program runs. Also the linking of all the subroutines has to be performed. And the code of all the subroutines remains resident in the main memory. As a result of all this is that memory gets occupied unnecessarily. Typically 'error routines' are such routines which can be invoked rarely. Then one can postpone the loading of these routines during the execution. If linking and loading of such rarely invoked external references could be postponed until the execution time when it was found to be absolutely necessary, then it increases the efficiency of overhead of the loader. In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is resumed. Thus dynamic linking both the loading and linking is done dynamically. **Advantages**

1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.
2. The system can be dynamically reconfigured.

Disadvantages The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the

process of execution needs to be suspended until the required subroutine gets loaded in the main memory

Bootstrap Loader: As we turn on the computer there is nothing meaningful in the main memory (RAM). A small program is written and stored in the ROM. This program initially loads the operating system from secondary storage to main memory. The operating system then takes the overall control. This program which is responsible for booting up the system is called bootstrap loader. This is the program which must be executed first when the system is first powered on. If the program starts from the location x then to execute this program the program counter of this machine should be loaded with the value x. Thus the task of setting the initial value of the program counter is to be done by machine hardware. The bootstrap loader is a very small program which is to be fitted in the ROM. The task of bootstrap loader is to load the necessary portion of the operating system in the main memory .The initial address at which the bootstrap loader is to be loaded is generally the lowest (may be at 0th location) or the highest location. . **Concept of Linking:** As we have discussed earlier, the execution of program can be done with the help of following steps

1. Translation of the program(done by assembler or compiler)
2. Linking of the program with all other programs which are needed for execution. This also involves preparation of a program called load module.
3. Loading of the load module prepared by linker to some specified memory location.

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary program. The "binary program also contains some necessary information about allocation and relocation. The loader then load s this program into memory for execution purpose.

Various tasks of linker are -

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.
2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.
3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.

The linking process is performed in two passes. Two passes are necessary because the linker may encounter a forward reference before knowing its address. So it is necessary to scan all the DEFINITION and USE table at least once. Linker then builds the Global symbol table with the help of USE and DEFINITION table. In Global symbol table name of each externally referenced symbol is included along with its address relative to beginning of the load module. And during pass 2, the addresses of external references are replaced by obtaining the addresses from global symbol table.

Operating System

Evolution of OS Functions

Functions of OS:

Operating System: "An operating system provides interface between the user & the hardware."
It can be basically classified into:

- Resource Allocation & Related Functions.
- User Interface Functions.

The Resource Allocation function implements resources sharing by the users of a computer system. Basically it performs binding of a set of resources with the requesting program-that is it associates resources with a program. The related functions implement protection of users sharing a set of resources against mutual interference.

Resource Allocation & Related Functions:

The resource allocation function allocates resources for use by a user's computation. Resources can be divided into two types:

1. System Provided Resources – like CPU, memory and IO devices

User created Resources – like files etc.

Resource allocation depends on whether a resource is a system resource or a user created resource.

There are two popular strategies for resource allocation:

Partitioning of resources

Allocation from a pool.

Using resource partition approach, OS decides priori what resources should be allocated to a user computation. This is known as static allocation as the allocation is made before the execution of the program starts.

Using pool allocation approach, OS maintains a common pool & allocates resources from this pool on a need basis. This is called dynamic allocation because it takes place during the execution of program. It can lead to better utilization of resources because the allocation is made when a program request a resource.

An OS can use a resource table as a central data structure for resource allocation. The table contains an entry for each resource unit in the system. The entry contains the name or address of the resource unit and its present system i.e whether it is free or allocated to some program. When a program raises a request for a resource ,the resource should be allocated to it if it is presently free.

In the partition resource allocation approach ,the OS decides on the resources to be allocated to a program based on the number of the program in the system.

For Example, an OS may decide that a program can be allocated 1 MB of memory, 200 disk blocks and a monitor. Such a collection of resources is referred to as a partition. The resource table can have an entry for each resource partition. When a new program is to be started, an available partition is allocated to it.

User Interface Functions:

Its purpose is to provide the use of OS resources for processing a user's computational requirements. OS user interfaces use command languages. For this, the user uses Command to set up an appropriate computational structure to fulfill his computational requirements.

An OS can define a variety of computational structures. A sample list of computational structures is as follows:

1. A single program

2. A sequence of single program

3. A collection of programs

The single program consist the execution of a program on a given set of data. The user initiates execution of the program through a command. Two kinds of program can exist – Sequential and concurrent. A sequential program is the simplest computational structure. In concurrent program the OS has to be aware of the identities of the different parts, which can execute concurrently.

Evolution of OS Functions:

Operating System functions have evolved in response to the following considerations and issues.

1. Efficient utilization of computing resources
2. New features in computer Architecture
3. New user requirements.

Different operating systems address these issues in different manner, however most operating system contains components, which have similar functionalities. For example, all operating systems contain components for functions of memory management, process management and protection of users from one another. The techniques used to implement these functions may vary from one OS to another, but the fundamental concept is same.

Process:

A process is execution of a program or a part of a program.

Job:

A job is computational structure, which is a sequence of program.

Types of Operating Systems:

1. Batch Processing system
2. Multiprogramming system
3. Time sharing system
4. Real time operating system
5. Distributed systems

Batch Processing Systems:

When Punch cards were used to record user jobs, processing of a job involved physical actions by the system operator e.g. loading a deck of cards into the card reader, pressing switches on the

computer's console to initiate the job. These actions wasted a lot of CPU time. BP was introduced to avoid this wastage.

A batch is a sequence of user jobs. A computer operator forms a batch by arranging user jobs in a sequence and inserting special marker cards to indicate the start and end of the batch. After forming a batch, the operator submits it to the batch processing operating system. The primary function of the BP system is to implement the processing of the jobs in a batch.

Batch processing is implemented by locating a component of the BP system called the batch monitor or supervisor, permanently in one part of the computer's memory. The remaining memory is used to process a user job the current job in the batch. The batch monitor is responsible for implementing the various function of the batch processing system. It accepts a command from the system operator for initiating the processing of a batch and sets up the processing of the first job of the batch. At the end of the job, it performs job termination processing and initiates execution of the batch; it performs batch termination processing system and awaits initiation of the next batch by the operator.

The part of memory occupied by the batch monitor is called the system area and the part occupied by the user job is called the user area.

User Service:

A user evaluates the performance of an os on the basis of the service accorded to his or her job. The notion of *turn-around time* is used to quantity user service in a batch processing system.
Note: The turn around time of a user job is the time since its submission to the time its results become available to the user

Batch processing does not guarantee improvements in the turn around time of jobs. Batch processing does not aim at improving user services-it aims at improving CPU utilization.

Batch Monitor Functions:

The basic task of the batch monitor is to exercise effective control Over the BP environment. This task can be classified into the following three functions.
Scheduling

Memory Management

Sharing and Protection

The batch monitor performs two functions before initiating the execution of a job. The third function is performed during the execution of a job.
In Batch Processing System, The CPU Of The Computer System Is The Server And The User Jobs Are The Service Requests. The Nature Of Batch Processing Dictates The Use Of The First Come First Serve(FCFS) Scheduling. The Batch Monitor Performs Scheduling By Always Selecting The Next Job In The Batch For Execution. Scheduling Does Not Influence The User Services In The BP System Because The Turn Around Time Of Each Job In A Batch Is Subject To Some Other Factors.
At any time during a BP system's operation, the memory is divided into the system area and the user area. The user area of the memory is sequentially shared by the jobs in the batch.

Multiprogramming System:
Early computer systems implemented IO operation as CPU instructions. It sent a signal to the card reader to read a card and waited for the operation to complete before initiating the next operation. However the speeds of operation of IO devices were much lower than the speed of the CPU. Programs took long to complete their execution. A new feature was introduced in the machine architecture when this weakness was realized. This feature permitted the CPU to delink itself from an IO operation so that it could execute instructions while an IO operation was in progress. Thus the CPU and the IO device could now operate concurrently.
If many user programs exist in the memory, the CPU can execute instructions of one program while the IO subsystem is busy with an IO operation for another program. The term multiprogramming is used to describe this arrangement. At any moment the program corresponding to the current job step of each job is in execution. The IO device and memory are allocated using the partitioned resource allocation approach. At any time, the CPU and IO subsystem are busy with programs belonging to different jobs. Thus they access different areas of memory. In principle the CPU and IO subsystem could operate on the same program. Each job in the memory could be current job of a batch of jobs. Thus one could have both batch processing and multiprogramming supervisor. Analogous to a BP supervisor, the MP supervisor also consists of a permanently resident part and a transient part.
The multiprogramming arrangement ensures concurrent operation of the CPU and the IO subsystem without requiring a program to use the special buffering techniques. It simply ensures that the CPU

is allocated to a program only when it is not performing an IO operation.

Functions of the Multiprogramming Supervisor:

Scheduling
Memory Management
IO management

The MP supervisor uses simple techniques to implement its functions. Function like scheduling implies sharing of the CPU between the jobs existing in the MP system. This function is performed after servicing every interrupt using a simple priority based scheme described in the next section. The allocation of memory and IO devices is performed by static partitioning of resources. Thus a part of memory and some IO devices are allocated to each job. It is necessary to protect the data and IO operations of one program from interference by another program. This is achieved by using memory protection hardware and putting CPU in non-privileged mode while executing a user program. Any effort by a user program to access memory locations situated outside its memory area now leads to an interrupt. The interrupting processing routines for these interrupts simply terminates the program causing the interrupt.

Scheduling:

The goal of multiprogramming is to exploit the concurrency of operation between the CPU and IO subsystem to achieve high levels of system utilization. A useful characterization of system utilization is offered by *throughput* of a system .

Throughput: The throughput of a system is the number of programs processed by it per unit time.

Throughput = Number of programs completed
Total time taken

To optimize the throughput, a MP system uses the following concepts:

A proper mix of programs:

For good throughput it is important to keep both the CPU and IO subsystems busy.

A CPU bound program is a program involving a lot of computation and very little IO. It uses the CPU for a long time.

An IO bound program is a program involving very little computation and a lot of IO.

2.Preemptive and priority based scheduling:

Scheduling is priority based that is the CPU is always allocated to the highest priority programs. The Scheduling is preemptive the is a

low priority program executing on the CPU is preempted if a higher priority program wishes to use the CPU.

3.Degree of multiprogramming:
Degree of multiprogramming is the number of programs existing simultaneously in the system's memory.

Deadlocks

Deadlocks

Processes compete for physical and logical resources in the system (e.g. disks or files). Deadlocks affect the progress of processes by causing indefinite delays in resource allocation. Such delays have serious consequences for the response times of processes, idling and wastage of resources allocated to processes, and the performance of the system. Hence an OS must use resource allocation policies, which ensure an absence of deadlocks. This chapter characterizes the deadlock problem and describes the policies an OS can employ to ensure an absence of deadlocks.

DEFINITIONS

We define three events concerning resource allocation:

1. *Resource request:* A user process requests a resource prior to its use. This is done through an OS call. The OS analyses the request and determines whether the requested resource can be allocated to the process immediately. If not. The process remains blocked on the request till the resource is allocated.

2. *Resource allocation:* The OS allocates are source to a requesting process. The resource status information is updated and the state of the process is changed to *ready.* The process now becomes the *holder* of the resource.

3. *Resource release:* After completing resource usage, a user process releases the resource through an OS call. If another process is blocked on the resource, OS allocates the resource to it. If several processes are blocked on the resource, the

OS uses some tie-breaking rule, e.g. FCFS allocation or allocation according to process priority, to perform the allocation.

Deadlock: A deadlock involving a set of processes D is a situation in which

1.Every process pi in D is blocked on some event ei

Event ei can only be caused by some process (es) in D.

If the event awaited by each process in D is the granting of some resource, it results in a resource deadlock. A communication deadlock occurs when the awaited events pertain to the receipt of interprocess messages, and synchronization deadlock when the awaited events concern the exchange of signals between processes. An

OS is primarily concerned with resource deadlocks because allocation of resources is an OS responsibility. The other two forms of deadlock are seldom tackled by an OS.

HANDLING DEADLOCKS

Two fundamental approaches used for handling deadlocks are:

1. Detection and resolution of deadlocks
2. Avoidance of deadlocks.

In the former approach, the OS detects deadlock situations as and when they arise. It then performs some actions aimed at ensuring progress for some of the deadlocked processes. These actions constitute deadlock resolution. The latter approach focuses on avoiding the occurrence of deadlocks. This approach involves checking each resource request to ensure that it does not lead to a deadlock. The detection and resolution approach does not perform any such checks. The choice of the deadlock handling approach would depend on the relative costs of the approach, and its consequences for user processes.

DEADLOCK DETECTION AND RESOLUTION

The deadlock characterization developed in the previous section is not very useful in practice for two reasons. First, it involves the overheads of building and maintaining an RRAG. Second, it restricts each resource request to a single resource unit of one or more resource classes. Due to these limitations, deadlock detection cannot be implemented merely as the determination of a graph property. For a practical implementation, the definition can be interpreted as follows: A set of *blocked* processes *D is* deadlocked if there does not exist any sequence of resource allocations and resource releases in the system whereby each process in *D* can complete. The OS must determine this fact through exhaustive analysis.

Deadlock analysis is performed by simulating the completion of a *running* process. In the simulation it is assumed that a *running* process completes without making additional resource requests. On completion, the process releases all resources allocated to it. These resources are allocated to a blocked process only if the process can enter the *running* state. The simulation terminates in one of two situations—either all *blocked* processes become *running* and complete, or some set *B of blocked* processes cannot be allocated their requested resources. In the former case no deadlock exists in the system at the time when deadlock analysis is performed, while in the latter case processes in *B* are deadlocked.

Deadlock Resolution

Given a set of deadlocked processes *D, deadlock resolution* implies breaking the deadlock to ensure progress for some processes *{pi} £ D.* This can be achieved by satisfying the resource request of a process *pi* in one of two ways:

1. Terminate some processes *{pj}* e *D* to free the resources required by *pi.* (We call each *pj* a *victim* of deadlock resolution.)
2. Add a new unit of the resource requested by *pi.*

Note that deadlock resolution only ensures some progress for *pi.* It does not guarantee that a *pi* would run to completion. That would depend on the behaviour of processes after resolution.

CP/M

Control Program/Microcomputer. An operating system created by Gary Kildall, the founder of Digital Research. Created for the old 8-bit microcomputers that used the 8080, 8085, and Z-80 microprocessors. Was the dominant operating system in the late

1970s and early 1980s for small computers used in a business environment.

DOS

Disk Operating System. A collection of programs stored on the DOS disk that contain routines enabling the system and user to manage information and the hardware resources of the computer. DOS must be loaded into the computer before other programs can be started.

operating system (OS)

A collection of programs for operating the computer. Operating systems perform housekeeping tasks such as input and output between the computer and peripherals as well as accepting and interpreting information from the keyboard. DOS and OS/2 are examples of popular 0S's.

0S/2

A universal operating system developed through a joint effort by IBM and Microsoft Corporation. The latest operating system from IBM for microcomputers using the Intel 386 or better microprocessors. OS/2 uses the protected mode operation of the processor to expand memory from 1M to 4G and to support fast, efficient multitasking. The 0512 Workplace Shell, an integral part of the system, is a graphical interface similar to Microsoft Windows and the Apple Macintosh system. The latest version runs DOS, Windows, and OS/2-specific software.

1

(Introduction to Operating System)

Definition: An operating system is a program that control the execution of application programs and acts as an interface between the user of a computer and the computer hardware.

Introduction:

• Operating system performs three functions:

1. Convenience: An as makes a computer more convenient to use.
2. Efficiency: An as allows the computer system resources to be used in an efficient manner.
3. Ability to evolve : An as should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without at the same time interfaring with service .

Operating System as a User Interface:

• Every general purpose computer consists of the hardware, operating system, system programs, application programs. The hardware consists of memory, CPU, ALU, I/O devices, peripheral device and storage device. System program consists of compilers, loaders, editors, as etc. The application program consists of business program, database program.
• The Figure below shows the conceptual view of a computer system. Every computer must have an operating system to run other programs. The operating system controls and co-ordinates the use of the hardware among the various system programs and application program for a various users. It simply provides an environment within which other programs can do useful work.

• The operating system is a set of special programs that run on a computer system that allow it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling a peripheral devices.
• OS is designed to serve two basic purposes :
1. It controls the allocation and use of the computing system's resources among the various users and tasks.
2. It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.
• The operating system must support the following tasks. The tasks are:
1. Provides the facilities to create, modification of program and data files using an editor.
2. Access to the compiler for translating the user program from high level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

Editor
Loade
Compiler
Application and utilities
Operating system
Computer hardware

Operating System Services:

• An operating system provides services to programs and to the users of those programs. It provides an environment for the execution of programs. The services provided by one operating system is different than other operating system.
• Operating system makes the programming task easier. The common services

Provided by the operating system is listed below.
1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection.
1. **Program execution**: Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.
2. ***I/O* operation:** I/O means any file or any specific I/O device. Program may require any I/O device while running. So operating system must provide the required I/O.
3. **File system manipulation**: Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.
4. **Communication:** Data transfer between two processes is required for some time. The both processes are on the one computer or on different computer but connected through computer network. Communication may be implemented by two methods: shared memory and message passing.
5. **Error detection:** Error may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.
Operating system with multiple users provides following services. 1. Resource allocation 2. Accounting 3. Protection •

• An operating system is a lower level of software that user programs run on.
OS is built directly on the hardware interface and provides an interface between the hardware and the user program. It shares characteristics 'with both software and hardware.
• We can view an operating system as a resource allocator. OS keeps track of the status of each resource and decides who gets a resource, for how long, and when. as makes sure that different programs and users running at the same time but do not interfere with each other. It is also responsible for security, ensuring that unauthorized users do not access the system.
• The primary objective of operating systems is to increase productivity of a processing resource, such as computer hardware or users.
• The operating system is the first program nm on a computer when the computer boots up. The services of the as are invoked with a system call instruction that is used just like any other hardware instruction.

• Name of the operating systems are: DOS, Windows 95, Windows NT/2000, Unix, Linux etc.

Operating System as Resource Manager

• A computer is a set of resources for the movement, storage and processing of data and for the control of these functions. The as is responsible for managing these resources.

• Main resources that are managed by the operating system. A portion of the operating system is in main memory. This includes the Kernel, which contains the most frequently used functions in the operating system and at a given time, other portions of the OS currently in use.

• The remainder of main memory contains other user programs and data. The allocation of main memory is controlled jointly by the OS and memory management hardware in the processor.

• The operating system decides when an I/O device can be used by a program in execution and controls access to and use of files. The processor itself is a resource, and the operating system must determine how much processor time is to be devoted to the execution of a particular user program.

History of Operating System

• Operating systems have been evolving through the years. Following table shows the history of OS.

Mainframe System: An operating system may process its workload serially or concurrently. That is resources of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution.

• Several variations of both serial and multiprogrammed operating systems exist.

Characteristics of mainframe systems

1. The first computers used to tackle various applications and still found today in corporate data centers.
2. Room-sized, high I/O capacity, reliability, security, technical support.
3. Mainframes focus on I/O bound business data applications.

Mainframes provide three main functions:

a. Batch processing: insurance claims, store sales reporting, etc.
b. Transaction processing: credit card, bank account, etc. c. Time-sharing: multiple users querying a database. **Batch Systems**

• Some computer systems only did one thing at a time. They had a list of instructions to carry out and these would be carried out one after the

other. This is called a **serial system.** The mechanics of development and preparation of programs in such environments are quite slow and numerous manual operations involved in the process.

• Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined into a single unit called **job.**

• Memory management in batch system is very simple. Memory is usually divided into two areas: Operating system and user program area.

Resident portion

• Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e. first come first served fashion.

• When a job completes execution, its memory is released and the output for the job gets copied into an output **spool** for later printing.

• Spooling an acronym for **simultaneous peripheral operation on line.** Spooling uses the disk as a large buffer for outputting data to printers and other devices. It can also be used for input, but is generally used for output. Its main use is to prevent two users from alternating printing lines to the line printer on the same page, getting their output completely mixed together. It also helps in reducing idle time and overlapped I/O and CPU.

• Batch system often provides simple forms of file management. Access ·to file is serial. Batch systems do not require any time critical device management.

• Batch systems are inconvenient for users because users can not interact with their jobs to fix problems. There may also be long turnaround times. Example of this system is generating monthly bank statement.

Spooling:

• Acronym for simultaneous peripheral operations on line. Spooling refers to putting jobs in a buffer, a special area in memory or on a disk where a device can access them when it is ready.

• Spooling is useful because device access data at different rates. The buffer provides a waiting station where data can rest while the slower device catches up.

• Computer can perform I/O in parallel with computation, it becomes possible to have the computer read a deck of cards to a tape, drum or disk and to write out to a tape printer while it was computing. This process is called **spooling.**

• The most common spooling application is print spooling. In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate.

• Spooling is also used for processing data at remote sites. The CPU sends the data via communications path to a remote printer. Spooling overlaps the I/O of one job with the computation of other jobs.
• One difficulty with simple batch systems is that the computer still needs to read the deck of cards before it can begin to execute the job. This means that the CPU is idle during these relatively slow operations.
• Spooling batch systems were the first and are the simplest of the multiprogramming systems.

Advantages of Spooling:

1. The spooling operation uses a disk as a very large buffer.
2. Spooling is however capable of overlapping I/O operation for one job with processor operations for another job.

Advantages of Batch System:

1. Move much of the work of the operator to the computer.
2. Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Bach System:

1. Turn around time can be large from user standpoint.
2. Difficult to debug program.
3. A job could enter an infinite loop.
4. A job could corrupt the monitor, thus affecting pending jobs.
5. Due to lack of protection scheme, one batch job can affect pending jobs.

Multiprogramming Operating System: When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It increases CPU utilization by organizing jobs so that the CPU always has one to execute.
• The operating system keeps several jobs 111 memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the job in the memory.
• Multiprogrammed systems provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.
• Jobs entering into the system are kept into the memory. Operating system picks the job and begins to execute one of the jobs in the memory. Having several programs in memory at the same time requires some form of memory management.
• Multiprogramming operating system monitors the state of all active programs and system resources. This ensures that the CPU is never idle unless there are no jobs.

Advantages

1. High CPU utilization.
2. It appears that many programs are allotted CPU almost simultaneously.

Disadvantages

1. CPU scheduling is required.
2. To accommodate many jobs in memory, memory management is required.

Time Sharing Systems:

• Time sharing system supports interactive users. Time sharing is also called **multitasking.** It is logical extension of multiprogramming. Time sharing system uses CPU scheduling and multiprogramming to provide an economical interactive system of two or more users.
• In time sharing, each user is given a time-slice for executing his job in round-robin fashion. Job continues until the time-slice ends.
• Time sharing systems are more complex than multiprogramming operating system. Memory management in time sharing system provides for isolation and protection of **co-resident** programs.
• Time sharing uses medium-term scheduling such as round-robin for the foreground. Background can use a different scheduling technique.
• Time sharing system can run several programs at the same time, so it is also a multiprogramming system. But multiprogramming operating system is not a time sharing system.
• Difference between both the systems is that, time sharing system allows more frequent context switches. This gives each user the impression that the entire computer is dedicated to his use. In multiprogramming system a context switch occurs only when the currently executing process stalls for some reason.

Desktop System: During the late 1970, computers had faster CPU, thus creating an even greater disparity between their rapid processing speed and slower I/O access time. Multiprogramming schemes to increase CPU use were limited by the physical capacity of the main memory, which was a limited resource and very expensive. These system includes PC running MS window and the Apple Macintosh. The Apple Macintosh OS support new advance hardware i.e. virtual memory and multitasking with virtual memory, the entire program did not need to reside in memory before execution could begin.
• Linux, a unix like OS available for PC, has also become popular recently. The microcomputer was developed for single users in the late 1970. Physical size was smaller than the minicomputers of that time, though larger than the microcomputers of today.
• Microcomputer grew to accommodate software with large capacity and greater speeds. The distinguishing characteristics of a microcomputer is

its single user status. MS-DOS is an example of a microcomputer operating system.

• The most powerful microcomputers used by commercial; educational, government enterprises. Hardware cost for microcomputers are sufficiently low that a single user (individuals) have sole use of a computer. Networking capability has been integrated into almost every system.

Multiprocessor System:

• Multiprocessor system have more than one processor in close communication. They share the computer bus, system clock and input-output devices and sometimes memory. In multiprocessing system, it is possible for two processes to run in parallel.

• Multiprocessor systems are of two types: symmetric multiprocessing and asymmetric multiprocessing.

• In symmetric multiprocessing, each processor runs an identical copy of the operating system and they communicate with one another as needed. All the CPU shared the common memory. Figure below shows the symmetric multiprocessing system.

Symmetric multiprocessing system (shared memory)

• In asymmetric multiprocessing, each processor is assigned a specific task. It uses master-slave relationship. A master processor controls the system. The master processor schedules and allocates work to the slave processors. Figure below shows the asymmetric multiprocessor.

Asymmetric multiprocessors (NO shared memory)

Features of multiprocessor systems

1. If one processor fails, then another processors should retrive the interrupted process state so that executation of the process can continue.
2. The processors should support efficient context switching operation.
3. Multiprocessor system supports large physical address space & large virtual address sapce.
4. The IPC mechanism should be provided & implemented in hardware as it becomes efficient & easy.

Distributed System: Distributed operating systems depend on networking for their operation. Distributed as runs on and controls the resources of multiple machines. It provides resource sharing across the boundaries of a single computer system. It looks to users like a single machine as. Distributing as owns the whole network and makes it look like a virtual uniprocessor or may be a virtual multiprocessor.

• Definition: A distributed operating system is one that looks to its users like an ordinary operating system but runs on multiple, independent CPU.

Advantages of distributed OS:

1. Resource sharing: Sharing of software resources such as software libraries, database and hardware resources such as hard disks, printers and

CDROM can also be done in a very effective way among all the computers and the users.
2. Higher reliability: Reliability refers to the degree of tolerance against errors and component failures. Availability is one of the important aspect of reliability. Availability refers to the fraction of time for which a system is available for use. Availability of a hard disk can be increased by having multiple hard disks located at different sites. If one hard disk fails or is unavailable, the program can use some other hard disk.
3. Better price performance ratio. Reduction in the price of microprocessor and increasing computing power gives good price-performance ratio.
4. Shorter responses times and higher throughput.
5. Incremental growth: To extend power and functionality of a system by simply adding additional resources to the system.

Difficulties in distributed OS are:

1. There are no current commercially successful examples.
2. Protocol overhead can dominate computation costs.
3. Hard to build well.
4. Probably impossible to build at the scale of the Internet.

Cluster System:

• It is a group of computer system connected with a high speed communication link. Each computer system has its own memory and peripheral devices. Clustering is usually performed to provide high availability. Clustered systems are integrated with hardware cluster and software cluster. Hardware cluster means sharing of high performance disks. Software cluster is in the form of unified control of the computer system in a cluster.
• A layer of software cluster runs on the cluster nodes. Each node can monitor one or more of the others. If the monitoring machine fails, the monitoring machine can take ownership of its storage and restart the application that were running on the failed machine.
• Clustered system can be categorized into two groups: asymmetric clustering and symmetric clustering.
• In asymmetric clustering, one machine is in hot standy mode while the other is running the applications. Hot standy mode monitors the active server and sometimes becomes the active server when the original server fails.
• In symmetric clustering mode, two or more than two hosts are running applications and they are monitoring each other.
• Parallel clusters and clustering over a WAN is also available in clustering.

Parallel clusters allow multiple hosts to access the same data on the shared storage. A cluster provides all the key advantages of distributed

systems. A cluster provides better reliability than the symmetrical multiprocessor system.
• Cluster technology is rapidly changing. Clustered system use and features should expand greatly as storage area networks. Storage area network allows easy attachment of multiple hosts to multiple storage units.

Real Time System:

• Real time systems which were originally used to control autonomous systems such as satellites, robots and hydroelectric dams. A real time operating system is one that must react to inputs and responds to them quickly. A real time system can not afford to be late with a response to an event.
• A real time system has well defined, fixed time constraints. Deterministic scheduling algorithms are used in real time systems. Real time systems are divided into two groups : **Hard real time system** and **soft real time system.**
• A hard real time system guarantees that the critical tasks be completed on time. This goal requires that all delay in the system be bounded. Soft real time system is a less restrictive type. In this, a critical r.eal time task gets priority over other tasks, and retains that priority until it completes.
• Real time operating system uses priority scheduling algorithm to meet the response requirement of a real time application.
• Memory management in real time system is comparatively less demanding than in other types of multiprogramming systems. Time-critical device management is one of the main characteristics of real time systems. The primary objective of file management in real time system is usually speed of access, rather than efficient utilization of secondary storage.

Comparison between Hard and Soft Real Time System

• Hard real time system guarantees that critical tasks complete on time. To achieve this, all delays in the system must be bounded i.e. the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Soft real time system are less restrictive than the hard real time system. In soft real time, a critical real time task gets priority over other tasks and retains that priority until it complete.
• Time constraints are the main properties for the hard real time systems. Since none of the operating system support hard real time system, Kernal delays need to be bounded in soft real time system. Soft real time systems are useful in the area of multimedia, virtual reality and advance scientific projects. Soft real time systems can not be used in -robotics and industrial control because of their lack of deadline support. Soft real time system

requires two conditions to implement. CPU scheduling must be priority based and dispatch latency must be small.

Handheld System:

• Personal Digital Assistants (PDA) is one type of handheld systems. Developing such device is the complex job and many challenges will face by developers. Size of these system is small i.e. height is 5 inches and width is 3 inches.

• Due to the limited size, most handheld devices have a small amount of memory, include slow processors and small display screen. Memory of handheld system is in the range of 512 kB to 8 MB. Operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer needed. Developers are working only on confines of limited physical memory because any handheld devices not using virtual memory.

• Speed of the handheld system is major factor. Faster processors require for handheld systems. Processors for most handheld devices often run at a fraction of the speed of a processor in a Pc. Faster processors require more power. Larger battery requires for faster processors.

• For mimimum size of handheld devices, smaller, slower processors which consumes less power are used. Typically small display screen is available in these devices. Display size of handheld device is not more than 3 inches square.

• At the same time, display size of monitor is up to 21 inches. But these handheld device provides the facility for reading email, browsing web pages on smaller display. Web clipping is used for displaying web page on the handheld devices.

• Wireless technology is also used in handheld devices. Bluetooth protocol is used for remote access to email and web browsing. Cellular telephones with connectivity to the Internet fall into this category.

Computing Environments:

• Different types of computing environments are:

a.Traditional computing

b.Web based computing

c. Embedded computing

• Typical office environment uses traditional computing. Normal PC is used in traditional computing.

• Web technology also uses traditional computing environment. Network computers are essentially terminals that understand web based computing. In domastic application, most of user had a single computer with Internet connection. Cost of the accessing Internet is high.

• Web based computing has increased the emphasis on networking. Web based computing uses PC, handheld PDA and cell phones. One of the

features of this type is load balancing. In load balancing, network connection is distributed among a pool of similar servers.
• Embedded computing uses realtime operating systems. Application of embedded computing is car engines, manufacturing robots to VCR and microwave ovens. This type of system provides limited features.

Essential Properties of the Operating System

1. **Batch:** Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off line operation, spooling and multiprogramming. A Batch system is good for executing large jobs that need little interaction, it can be submitted and picked up latter.
2. **Time sharing:** Uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another i.e. the CPU is shared between a number of interactive users. Instead of having a job defined by spooled card images, each program reads its next control instructions from the terminal and output is normally printed immediately on the screen.
3. **Interactive:** User is on line with computer system and interacts with it via an interface. It is typically composed of many short transactions where the result of the next transaction may be unpredictable. Response time needs to be short since the user submits and waits for the result.
4.**Real time system**: Real time systems are usually dedicated, embedded systems. They typically read from and react to sensor
data. The system must guarantee response to events within fixed periods of time to ensure correct performance.
5. **Distributed:** Distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines.

System Components: Modern operating systems share the goal of supporting the system components. The system components are:

1. Process management
2. Main memory management
3. File management
4. Secondary storage management
5. I/O system management
6. Networking
7. Protection system
8. Command interpreter system.

Process Management

• Process refers to a program in execution. The process abstraction is a fundamental operating system mechanism for management of concurrent program execution. The operating system responds by creating a process.
• A process needs certain resources, such as CPU time, memory, files and I/O devices. These resources are either given to the process when it is created or allocated to it while it is running.
• When the process terminates, the operating system will reclaim any reusable resources.
• The term process refers to an executing set of machine instructions. Program by itself is not a process. A program is a passive entity.
• The operating system is responsible for the following activities of the process
management.
1. Creating and destroying the user and system processes.
2. Allocating hardware resources among the processes.
3. Controlling the progress of processes.
4. Providing mechanisms for process communications.
5. Also provides mechanisms for deadlock handling.

Main Memory Management

• The memory management modules of an operating system are concerned with the management of the primary (main memory) memory. Memory management is concerned with following functions:
1. Keeping track of the status of each location of main memory. i.e. each memory location is either free or allocated.
2. Determining allocation policy for memory.
3. Allocation technique i.e. the specific. location must be selected and allocation information updated.
4. Deallocation technique and policy. After deallocation, status information must be updated.
• Memory management is primarily concerned with allocation of physical memory of finite capacity to requesting processes. The overall resource utilization and other performance criteria of a computer system are affected by performance of the memory management module. Many memory management schemes are available and the effectiveness of the different algorithms depends on the particular situation.

File Management

• Logically related data items on the secondary storage are usually organized into named collections called files. In short, file is a logical collection of information. Computer uses physical media for storing the different information.
• A file may contain a report, an executable program or a set of commands to the operating system. A file consists of a sequence of bits,

bytes, lines or records whose meanings are defined by their creators. For storing the files, physical media (secondary storage device) is used.
• Physical media are of different types. These are magnetic disk, magnetic tape and optical disk. All the media has its own characteristics and physical organization. Each medium is controlled by a device.
• The operating system is responsible for the following in connection with file management.
1. Creating and deleting of files.
2. Mapping files onto secondary storage.
3. Creating and deleting directories.
4. Backing up files on stable storage media.
5. Supporting primitives for manipulating files and directories.
6. Transmission of file elements between main and secondary storage.
• The file management subsystem can be implemented as one or more layers of the operating system.

Secondary Storage Management

• A storage device is a mechanism by which the computer may store information in such a way that this information may be retrieved at a later time. Secondary storage device is used for storing all the data and programs. These programs and data access by computer system must be kept in main memory. Size of main memory is small to accommodate all data and programs. It also lost the data when power is lost. For this reason secondary storage device is used. Therefore the proper management of disk storage is of central importance to a computer system.
• The operating system is responsible for the following activities in connection with the· disk management.
1. Free space management 2. Storage allocation 3. Disk scheduling
• The entire speed and performance of a computer may hinge on the speed of the disk subsystem.

I/O System Management : II The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.
• The I/O subsystem consists of
1. A memory management component that includes buffering, caching and spooling.
2. A general device driver interface.
3. Drivers for specific hardware devices.

Networking: Networking enables computer users to share resources and speed up computations. the processors communicate with one another through various communication lines. For example, a distributed system. A distributed system is a collection of processors. Each processor has its own local memory and clock. The processors in the system are connected

through a communication network, which can be configured in a number of different ways.

• Following parameter are considered while designing the networks.

1. Topology of network
2. Type of network
3. Physical media
4. Communication protocols
5. Routing algorithm.

Protection System:

• Modern computer systems support many users and allow the concurrent execution of multiple processes. Organizations rely on computers to store information. It is necessary that the information and devices must be protected from unauthorised users or processors. The protoction is any mechanism for controlling the access of programs, processes or users to the resources defined by a computer system.

• Protection mechanisms are implemented in operating systems to support various security policies. The goal of the security system is to authenticate subjects and to authorise their access to any object.

• Protection can improve reliability by detecting latent errors at the interfaces between compoent subsystems. Protection domains are extensions of the hardware supervisor mode ability

Command Interpreter System:

• Command interpreter is the interface between user and the operating system.

It is system programs for an operating system. Command interpreter is a special program in Unix and MS-DOS operating system.

• When users login first time or when a job is initiated, the command interpreter is initially some operating system is included in the Kernel. A control statement is processed by the command interpreter. Command interpreter reads the control statement, analyses it and carries out the required action.

Operating System Services:

• An operating system provides services to programs and to the users of those programs. It provides an environment for the execution of programs. The services provided by one operating system is different than other operating system. Operating system makes the programming task easier.

• The common services provided by the operating system is listed below.

1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection.

1. **Program execution**: Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.
2. **I/O Operation**: I/O means any file or any specific I/O device. Program may require any I/O device while running. So operating system must provide the required I/O.
3. **File system manipulation**: Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.
4. **Communication**: Data transfer between two processes is required for some time. The both processes are on the one computer or on different computer but connected through computer network. Communication may be implemented by two methods : shared memory and message passing.
5. **Error detection:** Error may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.

• Operating system with multiple users provides following services.
1. Resource allocation
2. Accounting
3. Protection

A) Resource allocation:

If there are more than one user or jobs running at the same time, then resources must be allocated to each of them. Operating system manages different types of resources. Some resources require special allocation code, i.e., main memory, CPU cycles and file storage.

• There are some resources which require only general request and release code. For allocating CPU, CPU scheduling algorithms are used for better utilization of CPU. CPU scheduling routines consider the speed of the CPU, number of available registers and other required factors.

B) Accounting:

• Logs of each user must be kept. It is also necessary to keep record of which user uses how much and what kinds of computer resources. This log is used for accounting purposes.

• The accounting data may be used for statistics or for the billing. It also used to improve system efficiency.

C) System Calls:

• Protection involves ensuring that all access to system resources is controlled.

Security starts with each user having to authenticate to the system, usually by means of a password. External I/O devices must be also protected from invalid access attempts.

• In protection, all the access to the resources is controlled. In multiprocess environment, it is possible that, one process to interface with the other, or with the operating system, so protection is required.

System Calls:

• Modern processors provide instructions that can be used as system calls. System calls provide the interface between a process and the operating system. A system call instruction is an instruction that generates an interrupt that cause the operating system to gain control of the processor.
• System call works in following ways :
1. First the program executes the system call instructions.
2. The hardware saves the current (instruction) and PSW register in the ii and iPSW register.
3. 0 value is loaded into PSW register by hardware. It keeps the machine in system mode with interrupt disabled.
4. The hardware loads the i register from the system call interrupt vector location. This completes the execution of the system call instruction by the hardware.
5. Instruction execution continues at the beginning of the system call interrupt handler.
6.The system call handler completes and executes a return from interrupt (rti) instructions. This restores the i and PSW from the ii and iPSW.
7. The process that executed the system call instruction continues at the instruction after the system call.

Types of System Call: A system call is made using the system call machine language instruction. System calls can be grouped into five major categories.
1. File management
2. Interprocess communication
3. Process management
4. I/O device management
5. Information maintenance.

Hardware Protection:

• For single-user programmer operating systems, programmer has the complete control over the system. They operate the system from the console. When new operating systems developed with some additional features, the system control transfers from programmer to the operating system.
•Early operating systems were called resident monitors, and starting with the resident monitor, the operating system began to perform many of the functions, like input-output operation.
•Before the operating system, programmer is responsible for the controls of input-output device operations. As the requirements of programmers

from computer systems go on increasing and development in the field of communication helps to the operating system.

•Sharing of resource among different programmers is possible without increasing cost. It improves the system utilization but problems increase. If single system was used without share, an error occurs, that could cause problems for only the one program which was running on that machine.

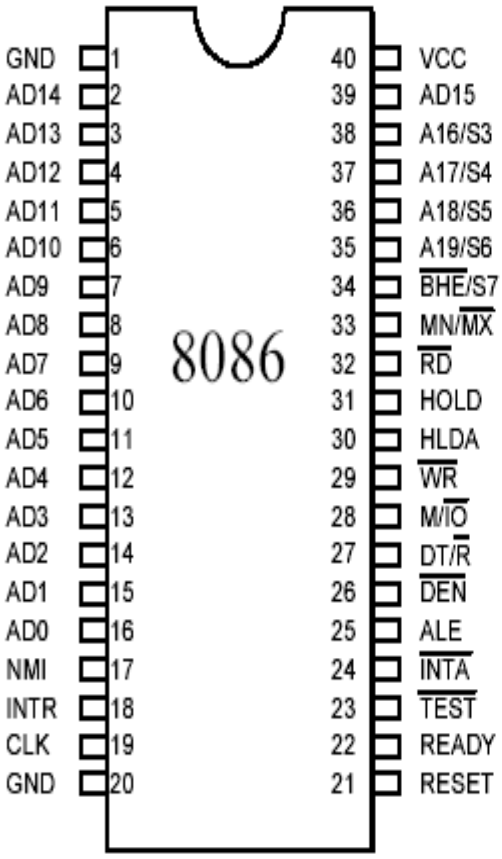
•In sharing, other programs also affected by single program. For example, batch operating system faces the problem of infinite loop. This loop could prevent the correct operation of many jobs. In multiprogramming system, one erroreous program affects the other program or data of that program.

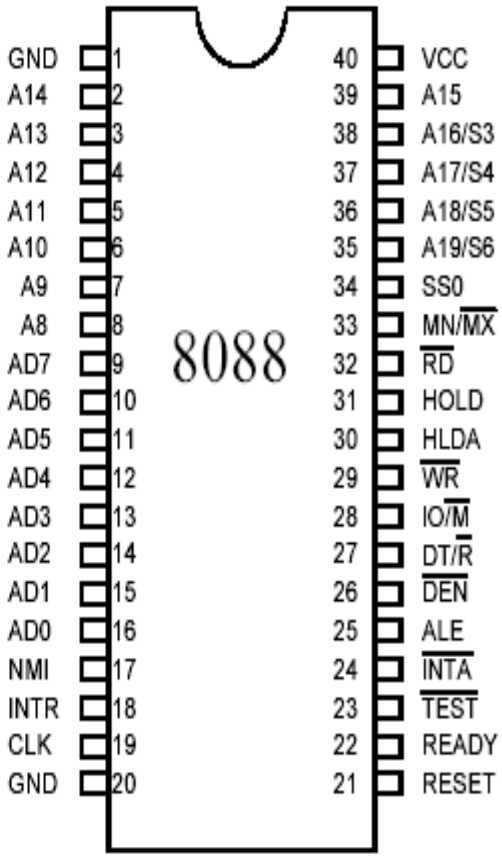
• For proper operation and error free result, protection of error is required. Without protection, only single process will execute one at a time otherwise the output of each program is separated. While designing the operating system, this type of care must be taken into consideration.

• Many programming errors are detected by the computer hardware. Operating system handled this type of errors. Execution of illegal instruction or access of memory that is not in the user's address space, this type of operation found by the hardware and will trap to the operating system.

• The trap transfers control through the interrupt vector to the operating system. Operating system must abnormally terminate the program when program error occurs. To handle this type of situation, different types of hardware protection is used

8086/8088 Pinout Diagrams



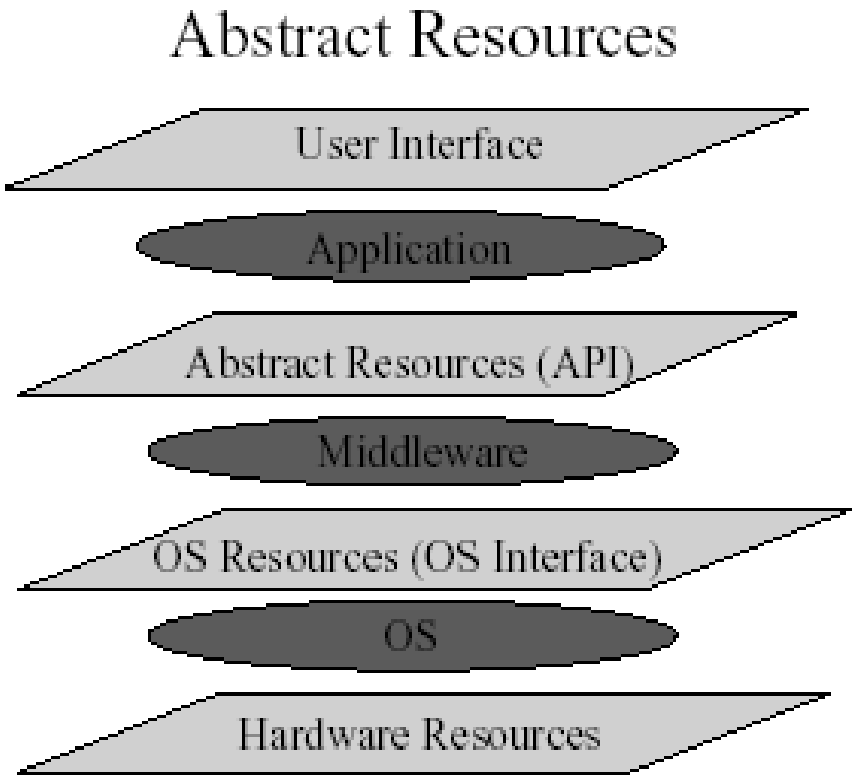


8086			8088		
Pin	Mode		Pin	Mode	
	Minimum	Maximum		Minimum	Maximum
31	HOLD	RQ/GT0	31	HOLD	RQ/GT0
30	HLDA	RQ/GT1	30	HLDA	RQ/GT1
29	WR	LOCK	29	WR	LOCK
28	M/IO	S2	28	IO/M	S2
27	DT/R	S1	27	DT/R	S1
26	DEN	S0	26	DEN	S0
25	ALE	QS0	25	ALE	QS0
24	INTA	QS1	24	INTA	QS1
			34	SS0	High State

**Minimum/Maximum Mode Refers to the Bus Handshaking*

Resources

- **Process**: An executing program
- **Resource**: Anything that is needed for a process to run
  - Memory
  - Space on a disk
  - The CPU
- An **OS** creates resource abstractions
- An **OS** manages resource sharing



• **Resident monitors** were the first, rudimentary, operating systems

– monitor is similar to OS kernel that must be resident in memory

– control-card interpreters eventually become command processors or shells

• There were still problems with computer utilization. Most of these problems revolved around I/O operations

Operating System Classification

OS	single-user	multi-user
single-tasking	MS-DOS, CP/M	Intellec S-IV (MS-DOS station in Novell)
multi-tasking	Windows, MacOS	Unix, Windows-NT server

Single-tasking system: only one process can be run simultaneously

Multi-tasking system: can run arbitrary number of processes simultaneously (yes, limited by the size of memory, etc.)

More precise classification:

• **multiprogrammed systems** - several tasks can be started and left un-finished; the CPU is assigned to the individual tasks by rotation, task waiting to the completion of the I/O operation (or other event) are blocked to save CPU time

• **time-sharing systems** - the CPU switching is so frequent that several users can interact with the computer simultaneously - interactive processing

Classification

From the hardware point of view :

• **software** = set of instructions

- o either always in memory (resident)
- o either loaded on request (non-resident or transient)

From the user point of view : Classification from the functionality

• **System software** :

- o Operating systems (including monitor, supervisor, …)
- o Loaders
- o Libraries and utility programs

• **Support software** (developpers) :

- o Assemblers
- o Compilers and interpreters
- o Editors
- o Debuggers

• **Application software**

The supervisor (monitor or kernel)

Memory Resident

• The utility programs are stored on the secondary storage device

- **Loaded** into memory at power-on (bootstrapping)
- On **request** (user or automatically) : tasks execution
- User interface : Job Control Language task(JCL)

1

(Introduction to Operating System)

Definition: An operating system is a program that control the execution of application programs and acts as an interface between the user of a computer and the computer hardware.

Introduction:

• Operating system performs three functions:

1. Convenience: An as makes a computer more convenient to use.
2. Efficiency: An as allows the computer system resources to be used in an efficient manner.
3. Ability to evolve : An as should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without at the same time interfaring with service .

Operating System as a User Interface:

• Every general purpose computer consists of the hardware, operating system, system programs, application programs. The hardware consists of memory, CPU, ALU, I/O devices, peripheral device and storage device. System program consists of compilers, loaders, editors, as etc. The application program consists of business program, database program.

• The Figure below shows the conceptual view of a computer system. Every computer must have an operating system to run other programs. The operating system controls and co-ordinates the use of the hardware among the various system programs and application program for a various users. It simply provides an environment within which other programs can do useful work.

• The operating system is a set of special programs that run on a computer system that allow it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling a peripheral devices.

• OS is designed to serve two basic purposes :

1. It controls the allocation and use of the computing system's resources among the various users and tasks.
2. It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.

• The operating system must support the following tasks. The tasks are:
1. Provides the facilities to create, modification of program and data files using an editor.
2. Access to the compiler for translating the user program from high level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

Editor
Loade
Compiler
Application and utilities
Operating system
Computer hardware

Operating System Services:

• An operating system provides services to programs and to the users of those programs. It provides an environment for the execution of programs. The services provided by one operating system is different than other operating system.
• Operating system makes the programming task easier. The common services
Provided by the operating system is listed below.
1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection.

1. **Program execution**: Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.
2. ***I/O* operation:** I/O means any file or any specific I/O device. Program may require any I/O device while running. So operating system must provide the required I/O.
3. **File system manipulation**: Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.
4. **Communication:** Data transfer between two processes is required for some time. The both processes are on the one computer or on different computer but connected through computer network. Communication may be implemented by two methods: shared memory and message passing.
5. **Error detection:** Error may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.

Operating system with multiple users provides following services. 1. Resource allocation 2. Accounting 3. Protection •

• An operating system is a lower level of software that user programs run on.
OS is built directly on the hardware interface and provides an interface between the hardware and the user program. It shares characteristics 'with both software and hardware.
• We can view an operating system as a resource allocator. OS keeps track of the status of each resource and decides who gets a resource, for how long, and when. as makes sure that different programs and users running at the same time but do not interfere with each other. It is also responsible for security, ensuring that unauthorized users do not access the system.
• The primary objective of operating systems is to increase productivity of a processing resource, such as computer hardware or users.
• The operating system is the first program nm on a computer when the computer boots up. The services of the as are invoked with a system call instruction that is used just like any other hardware instruction.
• Name of the operating systems are: DOS, Windows 95, Windows NT/2000, Unix, Linux etc.

Operating System as Resource Manager

• A computer is a set of resources for the movement, storage and processing of data and for the control of these functions. The as is responsible for managing these resources.
• Main resources that are managed by the operating system. A portion of the operating system is in main memory. This includes the Kernel, which contains the most frequently used functions in the operating system and at a given time, other portions of the OS currently in use.
• The remainder of main memory contains other user programs and data. The allocation of main memory is controlled jointly by the OS and memory management hardware in the processor.
• The operating system decides when an I/O device can be used by a program in execution and controls access to and use of files. The processor itself is a resource, and the operating system must determine how much processor time is to be devoted to the execution of a particular user program.

History of Operating System

• Operating systems have been evolving through the years. Following table shows the history of OS.

Mainframe System: An operating system may process its workload serially or concurrently. That is resources of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution.

• Several variations of both serial and multiprogrammed operating systems exist.

Characteristics of mainframe systems

1. The first computers used to tackle various applications and still found today in corporate data centers.
2. Room-sized, high I/O capacity, reliability, security, technical support.
3. Mainframes focus on I/O bound business data applications.

Mainframes provide three main functions:

a. Batch processing: insurance claims, store sales reporting, etc.
b. Transaction processing: credit card, bank account, etc. c. Time-sharing: multiple users querying a database. **Batch Systems**

• Some computer systems only did one thing at a time. They had a list of instructions to carry out and these would be carried out one after the other. This is called a **serial system.** The mechanics of development and preparation of programs in such environments are quite slow and numerous manual operations involved in the process.

• Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined into a single unit called **job.**

• Memory management in batch system is very simple. Memory is usually divided into two areas: Operating system and user program area.

Resident portion

• Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e. first come first served fashion.

• When a job completes execution, its memory is released and the output for the job gets copied into an output **spool** for later printing.

• Spooling an acronym for **simultaneous peripheral operation on line.** Spooling uses the disk as a large buffer for outputting data to printers and other devices. It can also be used for input, but is generally used for output. Its main use is to prevent two users from alternating printing lines to the line printer on the same page, getting their output completely mixed together. It also helps in reducing idle time and overlapped I/O and CPU.

• Batch system often provides simple forms of file management. Access ·to file is serial. Batch systems do not require any time critical device management.

• Batch systems are inconvenient for users because users can not interact with their jobs to fix problems. There may also be long turnaround times. Example of this system is generating monthly bank statement.

Spooling:

• Acronym for simultaneous peripheral operations on line. Spooling refers to putting jobs in a buffer, a special area in memory or on a disk where a device can access them when it is ready.
• Spooling is useful because device access data at different rates. The buffer provides a waiting station where data can rest while the slower device catches up.
• Computer can perform I/O in parallel with computation, it becomes possible to have the computer read a deck of cards to a tape, drum or disk and to write out to a tape printer while it was computing. This process is called **spooling.**
• The most common spooling application is print spooling. In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate.
• Spooling is also used for processing data at remote sites. The CPU sends the data via communications path to a remote printer. Spooling overlaps the I/O of one job with the computation of other jobs.
• One difficulty with simple batch systems is that the computer still needs to read the deck of cards before it can begin to execute the job. This means that the CPU is idle during these relatively slow operations.
• Spooling batch systems were the first and are the simplest of the multiprogramming systems.

Advantages of Spooling:

1. The spooling operation uses a disk as a very large buffer.
2. Spooling is however capable of overlapping I/O operation for one job with processor operations for another job.

Advantages of Batch System:

1. Move much of the work of the operator to the computer.
2. Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Bach System:

1. Turn around time can be large from user standpoint.
2. Difficult to debug program.
3. A job could enter an infinite loop.
4. A job could corrupt the monitor, thus affecting pending jobs.
5. Due to lack of protection scheme, one batch job can affect pending jobs.

Multiprogramming Operating System: When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It increases CPU utilization by organizing jobs so that the CPU always has one to execute.

• The operating system keeps several jobs 111 memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the job in the memory.

• Multiprogrammed systems provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.

• Jobs entering into the system are kept into the memory. Operating system picks the job and begins to execute one of the jobs in the memory. Having several programs in memory at the same time requires some form of memory management.

• Multiprogramming operating system monitors the state of all active programs and system resources. This ensures that the CPU is never idle unless there are no jobs.

Advantages

1. High CPU utilization.
2. It appears that many programs are allotted CPU almost simultaneously.

Disadvantages

1. CPU scheduling is required.
2. To accommodate many jobs in memory, memory management is required.

Time Sharing Systems:

• Time sharing system supports interactive users. Time sharing is also called **multitasking.** It is logical extension of multiprogramming. Time sharing system uses CPU scheduling and multiprogramming to provide an economical interactive system of two or more users.

• In time sharing, each user is given a time-slice for executing his job in round-robin fashion. Job continues until the time-slice ends.

• Time sharing systems are more complex than multiprogramming operating system. Memory management in time sharing system provides for isolation and protection of **co-resident** programs.

• Time sharing uses medium-term scheduling such as round-robin for the foreground. Background can use a different scheduling technique.

• Time sharing system can run several programs at the same time, so it is also a multiprogramming system. But multiprogramming operating system is not a time sharing system.

• Difference between both the systems is that, time sharing system allows more frequent context switches. This gives each user the impression that

the entire computer is dedicated to his use. In multiprogramming system a context switch occurs only when the currently executing process stalls for some reason.

Desktop System: During the late 1970, computers had faster CPU, thus creating an even greater disparity between their rapid processing speed and slower I/O access time. Multiprogramming schemes to increase CPU use were limited by the physical capacity of the main memory, which was a limited resource and very expensive. These system includes PC running MS window and the Apple Macintosh. The Apple Macintosh OS support new advance hardware i.e. virtual memory and multitasking with virtual memory, the entire program did not need to reside in memory before execution could begin.

• Linux, a unix like OS available for PC, has also become popular recently. The microcomputer was developed for single users in the late 1970. Physical size was smaller than the minicomputers of that time, though larger than the microcomputers of today.

• Microcomputer grew to accommodate software with large capacity and greater speeds. The distinguishing characteristics of a microcomputer is its single user status. MS-DOS is an example of a microcomputer operating system.

• The most powerful microcomputers used by commercial; educational, government enterprises. Hardware cost for microcomputers are sufficiently low that a single user (individuals) have sole use of a computer. Networking capability has been integrated into almost every system.

Multiprocessor System:

• Multiprocessor system have more than one processor in close communication. They share the computer bus, system clock and input-output devices and sometimes memory. In multiprocessing system, it is possible for two processes to run in parallel.

• Multiprocessor systems are of two types: symmetric multiprocessing and asymmetric multiprocessing.

• In symmetric multiprocessing, each processor runs an identical copy of the operating system and they communicate with one another as needed. All the CPU shared the common memory. Figure below shows the symmetric multiprocessing system.

Symmetric multiprocessing system (shared memory)

• In asymmetric multiprocessing, each processor is assigned a specific task. It uses master-slave relationship. A master processor controls the system. The master processor schedules and allocates work to the slave processors. Figure below shows the asymmetric multiprocessor.

Asymmetric multiprocessors (NO shared memory)

Features of multiprocessor systems

1. If one processor fails, then another processors should retrive the interrupted process state so that executation of the process can continue.
2. The processors should support efficient context switching operation.
3. Multiprocessor system supports large physical address space & large virtual address sapce.
4. The IPC mechanism should be provided & implemented in hardware as it becomes efficient & easy.

Distributed System: Distributed operating systems depend on networking for their operation. Distributed as runs on and controls the resources of multiple machines. It provides resource sharing across the boundaries of a single computer system. It looks to users like a single machine as. Distributing as owns the whole network and makes it look like a virtual uniprocessor or may be a virtual multiprocessor.

• Definition: A distributed operating system is one that looks to its users like an ordinary operating system but runs on multiple, independent CPU.

Advantages of distributed OS:

1. Resource sharing: Sharing of software resources such as software libraries, database and hardware resources such as hard disks, printers and CDROM can also be done in a very effective way among all the computers and the users.
2. Higher reliability: Reliability refers to the degree of tolerance against errors and component failures. Availability is one of the important aspect of reliability. Availability refers to the fraction of time for which a system is available for use. Availability of a hard disk can be increased by having multiple hard disks located at different sites. If one hard disk fails or is unavailable, the program can use some other hard disk.
3. Better price performance ratio. Reduction in the price of microprocessor and increasing computing power gives good price-performance ratio.
4. Shorter responses times and higher throughput.
5. Incremental growth: To extend power and functionality of a system by simply adding additional resources to the system.

Difficulties in distributed OS are:

1. There are no current commercially successful examples.
2. Protocol overhead can dominate computation costs.
3. Hard to build well.
4. Probably impossible to build at the scale of the Internet.

Cluster System:

• It is a group of computer system connected with a high speed communication link. Each computer system has its own memory and peripheral devices. Clustering is usually performed to provide high availability. Clustered systems are integrated with hardware cluster and software cluster. Hardware cluster means sharing of high performance

disks. Software cluster is in the form of unified control of the computer system in a cluster.
• A layer of software cluster runs on the cluster nodes. Each node can monitor one or more of the others. If the monitoring machine fails, the monitoring machine can take ownership of its storage and restart the application that were running on the failed machine.
• Clustered system can be categorized into two groups: asymmetric clustering and symmetric clustering.
• In asymmetric clustering, one machine is in hot standy mode while the other is running the applications. Hot standy mode monitors the active server and sometimes becomes the active server when the original server fails.
• In symmetric clustering mode, two or more than two hosts are running applications and they are monitoring each other.
• Parallel clusters and clustering over a WAN is also available in clustering.
Parallel clusters allow multiple hosts to access the same data on the shared storage. A cluster provides all the key advantages of distributed systems. A cluster provides better reliability than the symmetrical multiprocessor system.
• Cluster technology is rapidly changing. Clustered system use and features should expand greatly as storage area networks. Storage area network allows easy attachment of multiple hosts to multiple storage units.

Real Time System:

• Real time systems which were originally used to control autonomous systems such as satellites, robots and hydroelectric dams. A real time operating system is one that must react to inputs and responds to them quickly. A real time system can not afford to be late with a response to an event.
• A real time system has well defined, fixed time constraints. Deterministic scheduling algorithms are used in real time systems. Real time systems are divided into two groups : **Hard real time system** and **soft real time system.**
• A hard real time system guarantees that the critical tasks be completed on time. This goal requires that all delay in the system be bounded. Soft real time system is a less restrictive type. In this, a critical r.eal time task gets priority over other tasks, and retains that priority until it completes.
• Real time operating system uses priority scheduling algorithm to meet the response requirement of a real time application.
• Memory management in real time system is comparatively less demanding than in other types of multiprogramming systems. Time-

critical device management is one of the main characteristics of real time systems. The primary objective of file management in real time system is usually speed of access, rather than efficient utilization of secondary storage.

Comparison between Hard and Soft Real Time System

• Hard real time system guarantees that critical tasks complete on time. To achieve this, all delays in the system must be bounded i.e. the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Soft real time system are less restrictive than the hard real time system. In soft real time, a critical real time task gets priority over other tasks and retains that priority until it complete.

• Time constraints are the main properties for the hard real time systems. Since none of the operating system support hard real time system, Kernal delays need to be bounded in soft real time system. Soft real time systems are useful in the area of multimedia, virtual reality and advance scientific projects. Soft real time systems can not be used in -robotics and industrial control because of their lack of deadline support. Soft real time system requires two conditions to implement. CPU scheduling must be priority based and dispatch latency must be small. **Handheld System:**

• Personal Digital Assistants (PDA) is one type of handheld systems. Developing such device is the complex job and many challenges will face by developers. Size of these system is small i.e. height is 5 inches and width is 3 inches.

• Due to the limited size, most handheld devices have a small amount of memory, include slow processors and small display screen. Memory of handheld system is in the range of 512 kB to 8 MB. Operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer needed. Developers are working only on confines of limited physical memory because any handheld devices not using virtual memory.

• Speed of the handheld system is major factor. Faster processors require for handheld systems. Processors for most handheld devices often run at a fraction of the speed of a processor in a Pc. Faster processors require more power. Larger battery requires for faster processors.

• For mimimum size of handheld devices, smaller, slower processors which consumes less power are used. Typically small display screen is available in these devices. Display size of handheld device is not more than 3 inches square.

• At the same time, display size of monitor is up to 21 inches. But these handheld device provides the facility for reading email, browsing web pages on smaller display. Web clipping is used for displaying web page on the handheld devices.

• Wireless technology is also used in handheld devices. Bluetooth protocol is used for remote access to email and web browsing. Cellular telephones with connectivity to the Internet fall into this category.

Computing Environments:

• Different types of computing environments are:

a.Traditional computing

b.Web based computing

c. Embedded computing

• Typical office environment uses traditional computing. Normal PC is used in traditional computing.

• Web technology also uses traditional computing environment. Network computers are essentially terminals that understand web based computing. In domastic application, most of user had a single computer with Internet connection. Cost of the accessing Internet is high.

• Web based computing has increased the emphasis on networking. Web based computing uses PC, handheld PDA and cell phones. One of the features of this type is load balancing. In load balancing, network connection is distributed among a pool of similar servers.

• Embedded computing uses realtime operating systems. Application of embedded computing is car engines, manufacturing robots to VCR and microwave ovens. This type of system provides limited features.

Essential Properties of the Operating System

1. **Batch:** Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off line operation, spooling and multiprogramming. A Batch system is good for executing large jobs that need little interaction, it can be submitted and picked up latter.

2. **Time sharing:** Uses CPU scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another i.e. the CPU is shared between a number of interactive users. Instead of having a job defined by spooled card images, each program reads its next control instructions from the terminal and output is normally printed immediately on the screen.

3. **Interactive:** User is on line with computer system and interacts with it via an interface. It is typically composed of many short transactions where the result of the next transaction may be unpredictable. Response time needs to be short since the user submits and waits for the result.

4.**Real time system**: Real time systems are usually dedicated, embedded systems. They typically read from and react to sensor data. The system must guarantee response to events within fixed periods of time to ensure correct performance.

5. **Distributed:** Distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines.

System Components: Modern operating systems share the goal of supporting the system components. The system components are:

1. Process management
2. Main memory management
3. File management
4. Secondary storage management
5. I/O system management
6. Networking
7. Protection system
8. Command interpreter system.

Process Management

• Process refers to a program in execution. The process abstraction is a fundamental operating system mechanism for management of concurrent program execution. The operating system responds by creating a process.

• A process needs certain resources, such as CPU time, memory, files and I/O devices. These resources are either given to the process when it is created or allocated to it while it is running.

• When the process terminates, the operating system will reclaim any reusable resources.

• The term process refers to an executing set of machine instructions. Program by itself is not a process. A program is a passive entity.

• The operating system is responsible for the following activities of the process
management.

1. Creating and destroying the user and system processes.
2. Allocating hardware resources among the processes.
3. Controlling the progress of processes.
4. Providing mechanisms for process communications.
5. Also provides mechanisms for deadlock handling.

Main Memory Management

• The memory management modules of an operating system are concerned with the management of the primary (main memory) memory. Memory management is concerned with following functions:

1. Keeping track of the status of each location of main memory. i.e. each memory location is either free or allocated.
2. Determining allocation policy for memory.
3. Allocation technique i.e. the specific. location must be selected and allocation information updated.

4. Deallocation technique and policy. After deallocation, status information must be updated.

• Memory management is primarily concerned with allocation of physical memory of finite capacity to requesting processes. The overall resource utilization and other performance criteria of a computer system are affected by performance of the memory management module. Many memory management schemes are available and the effectiveness of the different algorithms depends on the particular situation.

File Management

• Logically related data items on the secondary storage are usually organized into named collections called files. In short, file is a logical collection of information. Computer uses physical media for storing the different information.

• A file may contain a report, an executable program or a set of commands to the operating system. A file consists of a sequence of bits, bytes, lines or records whose meanings are defined by their creators. For storing the files, physical media (secondary storage device) is used.

• Physical media are of different types. These are magnetic disk, magnetic tape and optical disk. All the media has its own characteristics and physical organization. Each medium is controlled by a device.

• The operating system is responsible for the following in connection with file management.

1. Creating and deleting of files.
2. Mapping files onto secondary storage.
3. Creating and deleting directories.
4. Backing up files on stable storage media.
5. Supporting primitives for manipulating files and directories.
6. Transmission of file elements between main and secondary storage.

• The file management subsystem can be implemented as one or more layers of the operating system.

Secondary Storage Management

• A storage device is a mechanism by which the computer may store information in such a way that this information may be retrieved at a later time. Secondary storage device is used for storing all the data and programs. These programs and data access by computer system must be kept in main memory. Size of main memory is small to accommodate all data and programs. It also lost the data when power is lost. For this reason secondary storage device is used. Therefore the proper management of disk storage is of central importance to a computer system.

• The operating system is responsible for the following activities in connection with the· disk management.

1. Free space management 2. Storage allocation 3. Disk scheduling

• The entire speed and performance of a computer may hinge on the speed of the disk subsystem.

I/O System Management : II The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.

• The I/O subsystem consists of

1. A memory management component that includes buffering, caching and spooling.
2. A general device driver interface.
3. Drivers for specific hardware devices.

Networking: Networking enables computer users to share resources and speed up computations. the processors communicate with one another through various communication lines. For example, a distributed system. A distributed system is a collection of processors. Each processor has its own local memory and clock. The processors in the system are connected through a communication network, which can be configured in a number of different ways.

• Following parameter are considered while designing the networks.

1. Topology of network
2. Type of network
3. Physical media
4. Communication protocols
5. Routing algorithm.

Protection System:

• Modern computer systems support many users and allow the concurrent execution of multiple processes. Organizations rely on computers to store information. It is necessary that the information and devices must be protected from unauthorised users or processors. The protoction is any mechanism for controlling the access of programs, processes or users to the resources defined by a computer system.

• Protection mechanisms are implemented in operating systems to support various security policies. The goal of the security system is to authenticate subjects and to authorise their access to any object.

• Protection can improve reliability by detecting latent errors at the interfaces between compoent subsystems. Protection domains are extensions of the hardware supervisor mode ability

Command Interpreter System:

• Command interpreter is the interface between user and the operating system.

It is system programs for an operating system. Command interpreter is a special program in Unix and MS-DOS operating system.

• When users login first time or when a job is initiated, the command interpreter is initially some operating system is included in the Kernel. A

control statement is processed by the command interpreter. Command interpreter reads the control statement, analyses it and carries out the required action.

Operating System Services:

• An operating system provides services to programs and to the users of those programs. It provides an environment for the execution of programs. The services provided by one operating system is different than other operating system. Operating system makes the programming task easier.

• The common services provided by the operating system is listed below.

1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection.

1. **Program execution**: Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.

2. **I/O Operation**: I/O means any file or any specific I/O device. Program may require any I/O device while running. So operating system must provide the required I/O.

3. **File system manipulation**: Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.

4. **Communication**: Data transfer between two processes is required for some time. The both processes are on the one computer or on different computer but connected through computer network. Communication may be implemented by two methods : shared memory and message passing.

5. **Error detection:** Error may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.

• Operating system with multiple users provides following services.

1. Resource allocation
2. Accounting
3. Protection

A) Resource allocation:

If there are more than one user or jobs running at the same time, then resources must be allocated to each of them. Operating system manages different types of resources. Some resources require special allocation code, i.e., main memory, CPU cycles and file storage.

• There are some resources which require only general request and release code. For allocating CPU, CPU scheduling algorithms are used

for better utilization of CPU. CPU scheduling routines consider the speed of the CPU, number of available registers and other required factors.

B) Accounting:

• Logs of each user must be kept. It is also necessary to keep record of which user uses how much and what kinds of computer resources. This log is used for accounting purposes.

• The accounting data may be used for statistics or for the billing. It also used to improve system efficiency.

C) System Calls:

• Protection involves ensuring that all access to system resources is controlled.

Security starts with each user having to authenticate to the system, usually by means of a password. External I/O devices must be also protected from invalid access attempts.

• In protection, all the access to the resources is controlled. In multiprocess environment, it is possible that, one process to interface with the other, or with the operating system, so protection is required.

System Calls:

• Modern processors provide instructions that can be used as system calls. System calls provide the interface between a process and the operating system. A system call instruction is an instruction that generates an interrupt that cause the operating system to gain control of the processor.

• System call works in following ways :

1. First the program executes the system call instructions.
2. The hardware saves the current (instruction) and PSW register in the ii and iPSW register.
3. 0 value is loaded into PSW register by hardware. It keeps the machine in system mode with interrupt disabled.
4. The hardware loads the i register from the system call interrupt vector location. This completes the execution of the system call instruction by the hardware.
5. Instruction execution continues at the beginning of the system call interrupt handler.
6. The system call handler completes and executes a return from interrupt (rti) instructions. This restores the i and PSW from the ii and iPSW.
7. The process that executed the system call instruction continues at the instruction after the system call.

Types of System Call: A system call is made using the system call machine language instruction. System calls can be grouped into five major categories.

1. File management
2. Interprocess communication
3. Process management

4. I/O device management
5. Information maintenance.

Hardware Protection:

• For single-user programmer operating systems, programmer has the complete control over the system. They operate the system from the console. When new operating systems developed with some additional features, the system control transfers from programmer to the operating system.

•Early operating systems were called resident monitors, and starting with the resident monitor, the operating system began to perform many of the functions, like input-output operation.

•Before the operating system, programmer is responsible for the controls of input-output device operations. As the requirements of programmers from computer systems go on increasing and development in the field of communication helps to the operating system.

•Sharing of resource among different programmers is possible without increasing cost. It improves the system utilization but problems increase. If single system was used without share, an error occurs, that could cause problems for only the one program which was running on that machine.

•In sharing, other programs also affected by single program. For example, batch operating system faces the problem of infinite loop. This loop could prevent the correct operation of many jobs. In multiprogramming system, one erroreous program affects the other program or data of that program.

• For proper operation and error free result, protection of error is required. Without protection, only single process will execute one at a time otherwise the output of each program is separated. While designing the operating system, this type of care must be taken into consideration.

• Many programming errors are detected by the computer hardware. Operating system handled this type of errors. Execution of illegal instruction or access of memory that is not in the user's address space, this type of operation found by the hardware and will trap to the operating system.

• The trap transfers control through the interrupt vector to the operating system. Operating system must abnormally terminate the program when program error occurs. To handle this type of situation, different types of hardware protection is used.

A specification of the source language forms the basis of source program analysis. In this section, we shall discuss important lexical, syntactic and semantic features of a programming language.

Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. This section discusses key concepts and notions from formal language grammars. A language L can be considered to be a collection of valid sentences.

Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as *a. formal language.* A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal symbols, alphabet and strings

The *alphabet* of L, denoted by the Greek symbol Z, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols in Z.

A symbol in the alphabet is known as a *terminal symbol* (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g.

Σ ≅ {a, b, ….z, 0,1....9}

Here the symbols {, ',' and} are part of the notation. We call them *met symbols* to differentiate them from terminal symbols. Throughout this discussion we assume that met symbols are distinct from the terminal symbols. If this is not the case, i.e. if a terminal symbol and a met symbol are identical, we enclose the terminal symbol in quotes to differentiate it from the meta symbol. For example, the set of punctuation symbols of English can be defined as

{:,;','-,...}

Where ',' denotes the terminal symbol 'comma'.

A *string* is a finite sequence of symbols. We will represent strings by Greek

symbols-α β γ, etc. Thus α = axy is a string over Σ . The length of a string is the

Number of symbols in it. Note that the absence of any symbol is also a string, the *null string* . The *concatenation* operation combines two strings into a single string.

To evaluate an HLL program it should be converted into the Machine language. A compiler performs another very important function. This is in terms of the diagnostics.

I.e. error – detection capability.

The important tasks of a compiler are:

Translating the HLL program input to it.

Providing diagnostic messages whenever specifications of the HLL

Assemblers & compilers

Assembler is a translator for the lower level assembly language of computer, while compilers are translators for HLLs.

An assembly language is mostly peculated to a certain computer, while an HLL is generally machined independent & thus portable.

Overview of the compilation process:

The process of compilation is:

Analysis of + Synthesis of = Translation of

Source Text Target Text Program

Source text analysis is based on the grimmer of the source of the source language.

The component sub – tasks of analysis phase are:

Syntax analysis, which determine the syntactic structure of the source statement.

Semantic analysis, which determines the meaning of a statement, once its grammatical structures become known.

The analysis phase

The analysis phase of a compiler performs the following functions.

Lexical analysis

Syntax analysis

Semantic analysis

Syntax analysis determines the grammatical or syntactic structure or the input statement & represents it in an intermediate form from which semantic analysis can be performed.

A compiler must perform two major tasks:

The Analysis of a source program & the synthesis of its corresponding object program.

The analysis task deals with the decomposition of the source program into its basic parts using these basic parts the synthesis task builds their equivalent object program modules. A source program is a string of symbols each of which is generally a letter, a digit or a certain special constants, keywords & operators. It is therefore desirable for the compiler to identify these various types as classes.

The analysis task deals with the decomposition of the source program into its basic parts using these basic parts the synthesis task builds their equivalent object program modules. A source program is a string of symbols each of which is generally a letter, a digit or a certain special constants, keywords & operators. It is therefore desirable for the compiler to identify these various types as classes.

The source program is input to a lexical analyzer or scanner whose purpose is to separate the incoming text into pieces or tokens such as constants, variable name, keywords & operators.

In essence, the lexical analyzer performs low- level syntax analysis performs low-level syntax analysis.

For efficiency reasons, each of tokens is given a unique internal representation number.

CP/M

Control Program/Microcomputer. An operating system created by Gary Kildall, the founder of Digital Research. Created for the old 8-bit

microcomputers that used the 8080, 8085, and Z-80 microprocessors. Was the dominant operating system in the late

1970s and early 1980s for small computers used in a business environment.

DOS

Disk Operating System. A collection of programs stored on the DOS disk that contain routines enabling the system and user to manage information and the hardware resources of the computer. DOS must be loaded into the computer before other programs can be started.

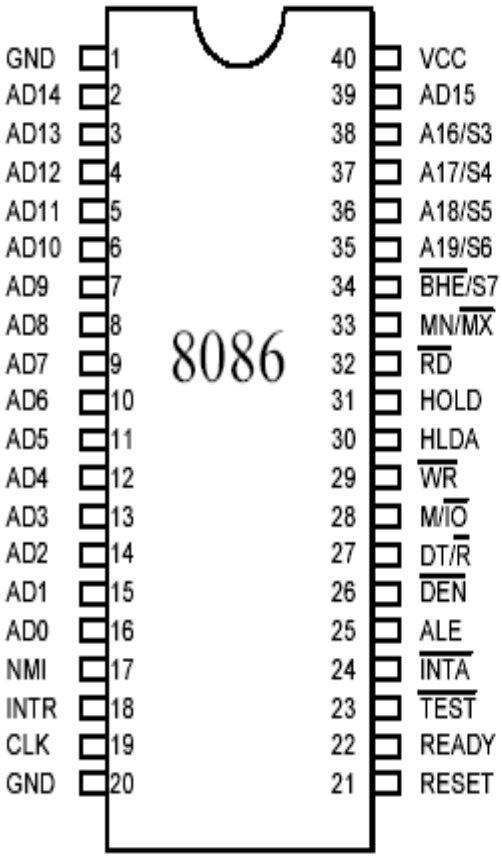
operating system (OS)

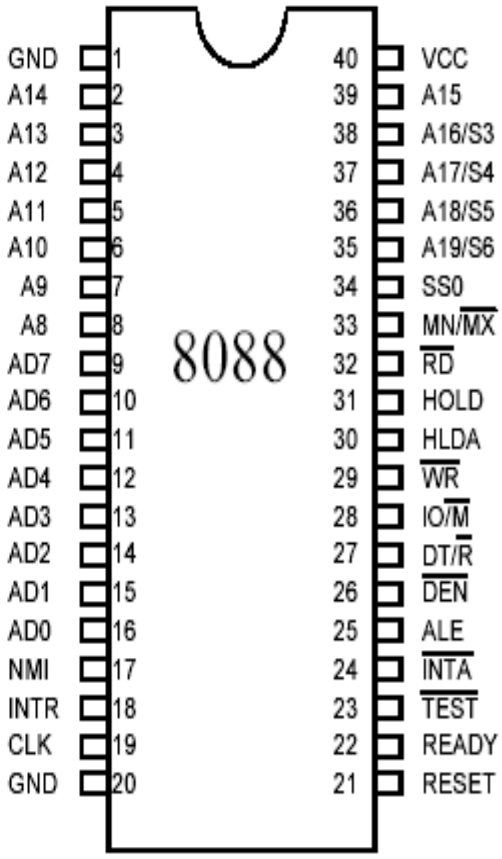
A collection of programs for operating the computer. Operating systems perform housekeeping tasks such as input and output between the computer and peripherals as well as accepting and interpreting information from the keyboard. DOS and OS/2 are examples of popular 0S's.

0S/2

A universal operating system developed through a joint effort by IBM and Microsoft Corporation. The latest operating system from IBM for microcomputers using the Intel 386 or better microprocessors. OS/2 uses the protected mode operation of the processor to expand memory from 1M to 4G and to support fast, efficient multitasking. The 0512 Workplace Shell, an integral part of the system, is a graphical interface similar to Microsoft Windows and the Apple Macintosh system. The latest version runs DOS, Windows, and OS/2-specific software.

8086/8088 Pinout Diagrams



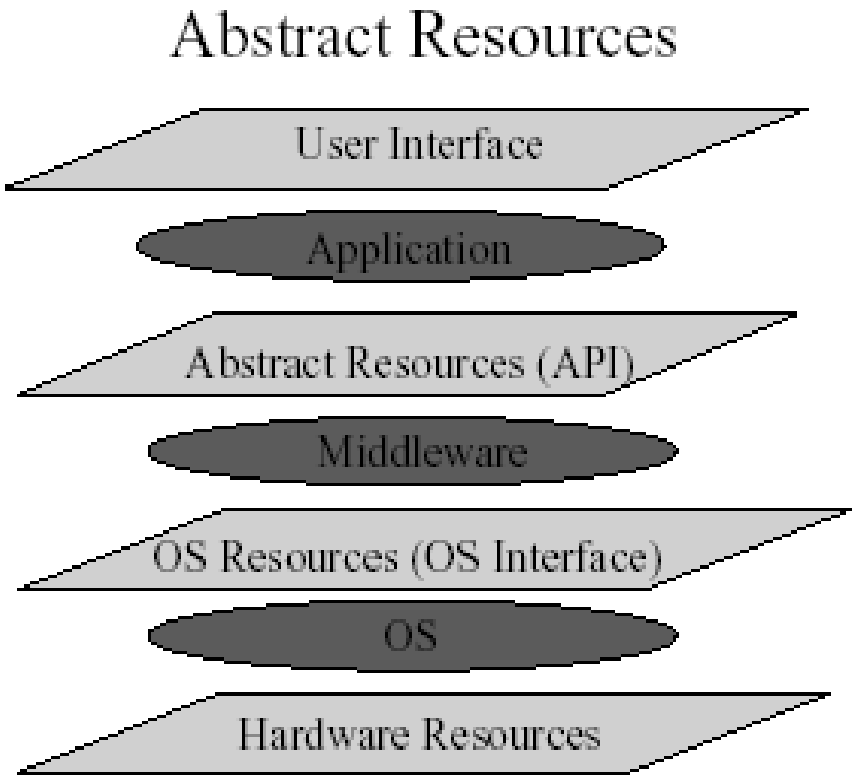


8086			8088		
Pin	Mode		Pin	Mode	
	Minimum	Maximum		Minimum	Maximum
31	HOLD	RQ/GT0	31	HOLD	RQ/GT0
30	HLDA	RQ/GT1	30	HLDA	RQ/GT1
29	WR	LOCK	29	WR	LOCK
28	M/IO	S2	28	IO/M	S2
27	DT/R	S1	27	DT/R	S1
26	DEN	S0	26	DEN	S0
25	ALE	QS0	25	ALE	QS0
24	INTA	QS1	24	INTA	QS1
			34	SS0	High State

**Minimum/Maximum Mode Refers to the Bus Handshaking*

Resources

- **Process**: An executing program
- **Resource**: Anything that is needed for a process to run
  - Memory
  - Space on a disk
  - The CPU
- An **OS** creates resource abstractions
- An **OS** manages resource sharing



• **Resident monitors** were the first, rudimentary, operating systems

– monitor is similar to OS kernel that must be resident in memory

– control-card interpreters eventually become command processors or shells

• There were still problems with computer utilization. Most of these problems revolved around I/O operations

Operating System Classification

OS	single-user	multi-user
single-tasking	MS-DOS, CP/M	Intellec S-IV (MS-DOS station in Novell)
multi-tasking	Windows, MacOS	Unix, Windows-NT server

Single-tasking system: only one process can be run simultaneously

Multi-tasking system: can run arbitrary number of processes simultaneously (yes, limited by the size of memory, etc.)

More precise classification:

• **multiprogrammed systems** - several tasks can be started and left un-finished; the CPU is assigned to the individual tasks by rotation, task waiting to the completion of the I/O operation (or other event) are blocked to save CPU time

• **time-sharing systems** - the CPU switching is so frequent that several users can interact with the computer simultaneously - interactive processing

Classification

From the hardware point of view :

• **software** = set of instructions

- either always in memory (resident)
- either loaded on request (non-resident or transient)

From the user point of view : Classification from the functionality

• **System software** :

- Operating systems (including monitor, supervisor, …)
- Loaders
- Libraries and utility programs

• **Support software** (developpers) :

- Assemblers
- Compilers and interpreters
- Editors
- Debuggers

• **Application software**

The supervisor (monitor or kernel)

Memory Resident

• The utility programs are stored on the secondary storage device

- **Loaded** into memory at power-on (bootstrapping)
- On **request** (user or automatically) : tasks execution
- User interface : Job Control Language task(JCL)

(Loaders and Linkers)

Introduction:

In this chapter we will understand the concept of linking and loading. As discussed earlier the source program is converted to object program by assembler. The loader is a program which takes this object program, prepares it for execution, and loads this executable code of the source into memory for execution.

Definition of Loader:

Loader is utility program which takes object code as input prepares it for execution and loads the executable code into the memory. Thus loader is actually responsible for initiating the execution process.

Functions of Loader:

The loader is responsible for the activities such as allocation, linking, relocation and loading

1) It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.

2) 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.

3) 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.

4) 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.

Loader Schemes:

Based on the various functionalities of loader, there are various types of loaders:

1) **“compile and go” loader:** in this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data isdirectly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77, it’s a FORTRAN compiler

which uses such “load and go” scheme. This loading scheme is also called as “assemble and go”.

Advantages:

• This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.

Disadvantages:

• In this scheme some portion of memory is occupied by assembler which is simply a wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.

• There is no production of .obj file, the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled and executed each time, which then becomes a time consuming activity.

• It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.

• For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the “compile and go” loader cannot handle such programs.

• The execution time will be more in this scheme as every time program is assembled and then executed.

2) **General Loader Scheme:** in this loader scheme, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.

Advantages:

• The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, then loader can make use of this object program to convert it to executable form.

• There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.

• It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.

3) **Absolute Loader:** Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places

them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler. The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer orassembler should have knowledge of memory management. The resolution of external references or linking of different subroutines are the issues which need to be handled by the programmer. The programmer should take care of two things: first thing is : specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next . modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is ,while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction. For example

```
Line number
1 MAIN START 1000
. .
. .
. .
15 JMP 5000
16 STORE ;instruction at location 2000
END
1 SUM START 5000
2
20 JMP 2000
21 END
```

In this example there are two segments, which are interdependent. At line number 1 the assembler directive START specifies the physical starting address that can be used during the execution of the first segment MAIN. Then at line number 15 the JMP instruction is given which specifies the physical starting address that can be used by the second segment. The assembler creates the object codes for these two segments by considering the stating addresses of these two segments. During the execution, the first segment will be loaded at address 1000 and second segment will be loaded at address 5000 as specified by the programmer. Thus the problem of linking is manually solved by the programmer itself by taking care of

the mutually dependant dresses. As you can notice that the control is correctly transferred to the address 5000 for invoking the other segment, and after that at line number 20 the JMP instruction transfers the control to the location 2000, necessarily at location 2000 the instruction STORE of line number 16 is present. Thus resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes
for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.
The entire process is modeled in the following figure.
Thus the absolute loader is simple to implement in this scheme-
l) Allocation is done by either programmer or assembler
2)Linking is done by the programmer or assembler
3)Resolution is done by assembler
4)Simply loading is done by the loader
As the name suggests, no relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

Advantages:

1. It is simple to implement
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient

Disadvantages:

1. In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.
If at all any modification is done the some segments, the starting addresses of immediate next segments may get changed, the programmer has to take care of this issue and he needs to update the corresponding starting addresses on any modification in the source.
Algorithm for absolute Loader
Input: Object codes and starting address of program segments.

Output: An executable code for corresponding source program. This executable code is to be placed in the main memory

Method: Begin

```
For each program segment do Begin
Read the first line from object module to obtain
information about memory location. The starting address
say S in corresponding object module is the memory
location where executale code is to be placed.
Hence
Memory_location = S
Line counter = 1; as it is first line While (! end of
file)
For the curent object code do Begin
1. Read next line
2. Write line into location S
3. S = S + 1
4. Line counter Line counter + 1
```

Subroutine Linkage: To understand the concept of subroutine linkages, first consider the following scenario:

"In Program A a call to subroutine B is made. The subroutine B is not written in the program segment of A, rather B is defined in some another program segment C"

Nothing is wrong in it. But from assembler's point of view while generating the code for B, as B is not defined in the segment A, the assembler can not find the value of this symbolic reference and hence it will declare it as an error. To overcome problem, there should be some mechanism by which the assembler should be explicitly informed that segment B is really defined in some other segment C. Therefore whenever segment B is used in segment A and if at all B is defined in C, then B must -be declared as an external routine in A. To declare such subroutine asexternal, we can use the assembler directive EXT. Thus the statement such as EXT B should be added at the beginning of the segment A. This actually helps to inform assembler that B is defined somewhere else. Similarly, if one subroutine or a variable is defined in the current segment and can be referred by other segments then those should be declared by using pseudo-ops INT. Thereby the assembler could inform loader that these are the subroutines or variables used by other segments. This overall process of establishing the relations between the subroutines can be conceptually called a_ subroutine linkage.

For example

MAIN START

EXT B

.

.

```
.
CALL B
.
.
END
B START
.
.
RET
END
```

At the beginning of the MAIN the subroutine B is declared as external. When a call to subroutine B is made, before making the unconditional jump, the current content of the program counter should be stored in the system stack maintained internally. Similarly while returning from the subroutine B (at RET) the pop is performed to restore the program counter of caller routine with the address of next instruction to be executed.

Concept of relocations:

Relocation is the process of updating the addresses used in the address sensitive instructions of a program. It is necessary that such a modification should help to execute the program from designated area of the memory.

The assembler generates the object code. This object code gets executed after loading at storage locations. The addresses of such object code will get specified only after the assembly process is over. Therefore, after loading, Address of object code = Mere address of object code + relocation constant.

There are two types of addresses being generated: Absolute address and, relative address. The absolute address can be directly used to map the object code in the main memory. Whereas the relative address is only after the addition of relocation constant to the object code address. This kind of adjustment needs to be done in case of relative address before actual execution of the code. The typical example of relative reference is : addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals, redefinable symbols. Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute.

The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.

For example

```
LOADA(X)+5
```

The expression A(X) means the address of variable X. The meaning of the above instruction is that loading of the contents of memory location which is 5 more than the address of variable X. Suppose if the address of X is 50 then by above command we try to get the memory location 50+5=55. Therefore as the address of variable X is relative A(X) + 5 is also relative. To calculate the relative addresses the simple expressions are allowed. It is expected that the expression should possess at the most addition and multiplication operations. A simple exercise can be carried out to determine whether the given address is absolute or relative. In the expression if the address is absolute then put 0 over there and if address is relative then put lover there. The expression then gets transformed to sum of O's and l's. If the resultant value of the expression is 0 then expression is absolute. And if the resultant value of the expression is 1 then the expression is relative. If the resultant is other than 0 or 1then the expression is illegal. For example:

In the above expression the *A,* Band C are the variable names. The assembler is to c0l1sider the relocation attribute and adjust the object code by relocation constant. Assembler is then responsible to convey the information loading of object code to the loader. Let us now see how assembler generates code using relocation information.

Direct Linking Loaders

The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader can not have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative. If at all the address is relative then it is the assembler who informs the loader about the relative addresses.

The assembler should give the following information to the loader

1)The length of the object code segment

2) The list of all the symbols which are not defined 111 the current segment but can be used in the current segment.

3) The list of all the symbols which are defined in the current segment but can be referred by the other segments.

The list of symbols which are not defined in the current segment but can be used in the current segment are stored in a data structure called USE table. The USE table holds the information such as name of the symbol, address, address relativity.

The list of symbols which are defined in the current segment and can be referred by the other segments are stored in a data structure called DEFINITION table. The definition table holds the information such as symbol, address.

Overlay Structures and Dynamic Loading:

Sometimes a program may require more storage space than the available one Execution of such program can be possible if all the segments are not required simultaneously to be present in the main memory. In such situations only those segments are resident in the memory that are actually needed at the time of execution But the question arises what will happen if the required segment is not present in the memory? Naturally the execution process will be delayed until the required segment gets loaded in the memory. The overall effect of this is efficiency of execution process gets degraded. The efficiency can then be improved by carefully selecting all the interdependent segments. Of course the assembler can not do this task. Only the user can specify such dependencies. The inter dependency of thesegments can be specified by a tree like structure called static overlay structures. The overlay structure contain multiple root/nodes and edges. Each node represents the segment. The specification of required amount of memory is also essential in this structure. The two segments can lie simultaneously in the main memory if they are on the same path. Let us take an example to understand the concept. Various segments along with their memory requirements is as shown below.

Automatic Library Search:

Previously, the library routines were available in absolute code but now the library routines are provided in relocated form that ultimately reduces their size on the disk, which in turn increases the memory utilization. At execution time certain library routines may be needed. Keeping track of which library routines are required and how much storage is required by these routines, if at all is done by an assembler itself then the activity of automatic library search becomes simpler and effective. The library routines can also make an external call to other routines. The idea is to make a list of such calls made by the routines. And if such list is made available to the linker then linker can efficiently find the set of required routines and can link the references accordingly.

For an efficient search of library routines it desirable to store all the calling routines first and then the called routines. This avoids wastage of time due to winding and rewinding. For efficient automated search of

library routines even the dictionary of such routines can be maintained. A table containing the names of library routines and the addresses where they are actually located in relocatable form is prepared with the help of translator and such table is submitted to the linker. Such a table is called subroutine directory. Even if these routines have made any external calls the -information about it is also given in subroutine directory. The linker searches the subroutine directory, finds the address of desired library routine (the address where the routine is stored in relocated form).Then linker prepares aload module appending the user program and necessary library routines by doing the necessary relocation. If the library routine contains the external calls then the linker searches the subroutine directory finds the address of such external calls, prepares the load module by resolving the external references. **Linkage Editor:** The execution of any program needs four basic functionalities and those are allocation, relocation, linking and loading. As we have also seen in direct linking loader for execution of any program each time these four functionalities need to be performed. But performing all these functionalities each time is time and space consuming task. Moreover if the program contains many subroutines or functions and the program needs to be executed repeatedly then this activity becomes annoyingly complex .Each time for execution of a program, the allocation, relocation linking and -loading needs to be done. Now doing these activities each time increases the time and space complexity. Actually, there is no need to redo all these four activities each time. Instead, if the results of some of these activities are stored in a file then that file can be used by other activities. And performing allocation, relocation, linking and loading can be avoided each time. The idea is to separate out these activities in separate groups. Thus dividing the essential four functions in groups reduces the overall time complexity of loading process. The program which performs allocation, relocation and linking is called binder. The binder performs relocation, creates linked executable text and stores this text in a file in some systematic manner. Such kind of module prepared by the binder execution is called load module. This load module can then be actually loaded in the main memory by the loader. This loader is also called as module loader. If the binder can produce the exact replica of executable code in the load module then the module loader simply loads this file into the main memory which ultimately reduces the overall time

complexity. But in this process the binder should knew the current positions of the main memory. Even though the binder knew the main memory locations this is not the only thing which is sufficient. In multiprogramming environment, the region of main memory available for loading the program is decided by the host operating system. The binder should also know which memory area is allocated to the loading program and it should modify the relocation information accordingly. The binder which performs the linking function and produces adequate information about allocation and relocation and writes this information along with the program code in the file is called linkage editor. The module loader then accepts this rile as input, reads the information stored in and based on this information about allocation and relocation it performs the task of loading in the main memory. Even though the program is repeatedly executed the linking is done only once. Moreover, the flexibility of allocation and relocation helps efficient utilization of the main memory.

Direct linking: As we have seen in overlay structure certain selective subroutines can be resident in the memory. That means it is not necessary to resident all the subroutines in the memory for all the time. Only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory. This process of postponing linking and loading of external reference until execution is called dynamic linking. For example suppose the subroutine main calls A,B,C,D then it is not desirable to load A,B,C and D along with the main in the memory. Whether A, B, C or D is called by the main or not will be known only at the time of execution. Hence keeping these routines already before is really not needed. As the subroutines get executed when the program runs. Also the linking of all the subroutines has to be performed. And the code of all the subroutines remains resident in the main memory. As a result of all this is that memory gets occupied unnecessarily. Typically 'error routines' are such routines which can be invoked rarely. Then one can postpone the loading of these routines during the execution. If linking and loading of such rarely invoked external references could be postponed until the execution time when it was found to be absolutely necessary, then it increases the efficiency of overhead of the loader. In dynamic linking, the binder first prepares a load module in which along with program code the allocation and relocation information is stored. The loader simply loads the main module in the main memory. If any external ·reference to a subroutine comes, then the execution is suspended for a while, the loader brings the required subroutine in the main memory and then the execution process is

resumed. Thus dynamic linking both the loading and linking is done dynamically. **Advantages**
1. The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.
2. The system can be dynamically reconfigured.
Disadvantages The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory

Bootstrap Loader: As we turn on the computer there is nothing meaningful in the main memory (RAM). A small program is written and stored in the ROM. This program initially loads the operating system from secondary storage to main memory. The operating system then takes the overall control. This program which is responsible for booting up the system is called bootstrap loader. This is the program which must be executed first when the system is first powered on. If the program starts from the location x then to execute this program the program counter of this machine should be loaded with the value x. Thus the task of setting the initial value of the program counter is to be done by machine hardware. The bootstrap loader is a very small program which is to be fitted in the ROM. The task of bootstrap loader is to load the necessary portion of the operating system in the main memory .The initial address at which the bootstrap loader is to be loaded is generally the lowest (may be at 0th location) or the highest location. . **Concept of Linking:** As we have discussed earlier, the execution of program can be done with the help of following steps
1. Translation of the program(done by assembler or compiler)
2. Linking of the program with all other programs which are needed for execution. This also involves preparation of a program called load module.
3. Loading of the load module prepared by linker to some specified memory location.
The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary program. The "binary program also contains some necessary information about allocation and relocation. The loader then load s this program into memory for execution purpose.

Various tasks of linker are -

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.
2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.
3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.

The linking process is performed in two passes. Two passes are necessary because the linker may encounter a forward reference before knowing its address. So it is necessary to scan all the DEFINITION and USE table at least once. Linker then builds the Global symbol table with the help of USE and DEFINITION table. In Global symbol table name of each externally referenced symbol is included along with its address relative to beginning of the load module. And during pass 2, the addresses of external references are replaced by obtaining the addresses from global symbol table.

FUNDAMENTALS OF LANGUAGE PROCESSING

Definition

Language Processing = Analysis of SP + Synthesis of TP.

Definition motivates a generic model of language processing activities.

We refer to the collection of language processor components engaged in analyzing a source program as the *analysis phase* of the language processor. Components engaged in synthesizing a target program constitute the *synthesis phase.*

A specification of the source language forms the basis of source program analysis. The specification consists of three components:

1. *Lexical rules*, which govern the formation of valid lexical units in the source language.
2. *Syntax rules* which govern the formation of valid statements in the source language.
3. *Semantic rules* which associate meaning with valid statements of the language.

The analysis phase uses each component of the source language specification to determine relevant information concerning a statement in the source program. Thus, analysis of a source statement consists of lexical, syntax and semantic analysis.

The synthesis phase is concerned with the construction of target language statement(s) which have the same meaning as a source statement. Typically, this consist of two main activities:

- Creation of data structures in the target program
- Generation of target code.

We refer to these activities as *memory allocation* and *code generation,* respectively

Lexical Analysis (Scanning)

Lexical analysis identifies the lexical units in a source statement. It then classifies the units into different lexical classes e.g. id's, constants etc. and enters them into different tables. This classification may be based on the nature of string or on the specification of the source language. (For example, while an integer constant is a string of digits with an optional sign, a reserved id is an id whose name matches one of the reserved names mentioned in the language specification.) Lexical analysis builds a descriptor, called a *token,* for each lexical unit. A token contain two fields—*class code,* and *number in class, class code* identifies the class to which a lexical unit belongs, *number in class* is the entry number of the lexical unit in the relevant table.

Syntax Analysis (Parsing)

Syntax analysis processes the string of tokens built by lexical analysis to determine the statement class, e.g. assignment statement, if statement, etc. It then builds an IC which represents

the structure of the statement. The IC is passed to semantic analysis to determine the meaning of the statement.

Semantic analysis

Semantic analysis of declaration statements differs from the semantic analysis of imperative statements. The former results in addition of information to the symbol table, e.g. type, length and dimensionality of variables. The latter identifies the sequence of actions necessary to implement the meaning of a source statement. In both cases the structure of a source statement guides the application of the semantic rules. When semantic analysis determines the meaning of a sub tree in the IC. It adds information a table or adds an action to the sequence. It then modifies the IC to enable further semantic analysis. The analysis ends when the tree has been completely processed.

"FUNDAMENTALS OF LANGUAGE SPECIFICATION

A specification of the source language forms the basis of source program analysis. In this section, we shall discuss important lexical, syntactic and semantic features of a programming language.

Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. This section discusses key concepts and notions from formal language grammars. A language L can be considered to be a collection of valid sentences.

Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as *a. formal language.* A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal symbols, alphabet and strings

The *alphabet* of L, denoted by the Greek symbol Z, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols in Z.

A symbol in the alphabet is known as a *terminal symbol* (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g. $\Sigma \cong$ {a, b, ….z, 0,1….9}

Here the symbols {, ',' and} are part of the notation. We call them *met symbols* to differentiate them from terminal symbols. Throughout this discussion we assume that met symbols are distinct from the terminal symbols. If this is not the case, i.e. if a terminal symbol and a met symbol are identical, we enclose the terminal symbol in quotes to differentiate it from the metasymbol. For example, the set of punctuation symbols of English can be defined as {:,;',’-,...} Where ',' denotes the terminal symbol 'comma'.

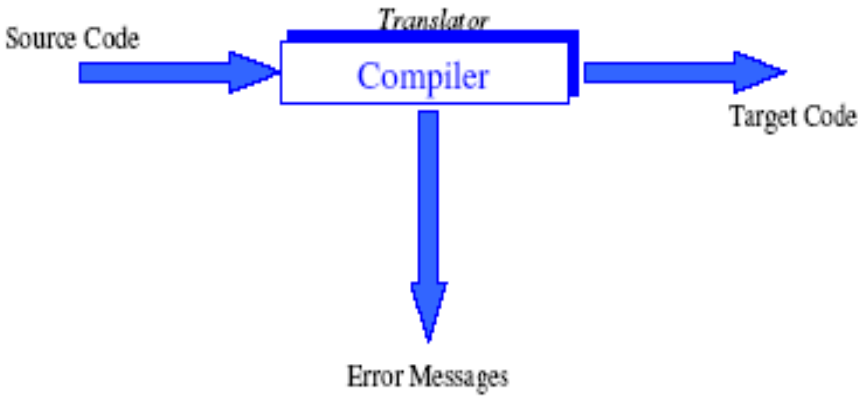
A *string* is a finite sequence of symbols. We will represent strings by Greek symbols-α β γ, etc. Thus α = axy is a string over Σ . The length of a string is the Number of symbols in it. Note that the absence of any symbol is also a string, the *null string* . The *concatenation* operation combines two strings into a single string.
To evaluate an HLL program it should be converted into the Machine language. A compiler performs another very important function. This is in terms of the diagnostics.

I.e. error – detection capability.
The important tasks of a compiler are:
Translating the HLL program input to it.
Providing diagnostic messages whenever specifications of the HLL

Compilers

Introduction

Compiler is tool: which translate notations from one system to another, usually from source code (high level code) to machine code (object code, target code, low level code).



• A compiler is a **program** that translates a **sentence**

a. from a **source** language (e.g. Java, Scheme, LATEX)
b. into a **target** language (e.g. JVM, Intel x86, PDF)
c. while preserving its **meaning** in the process

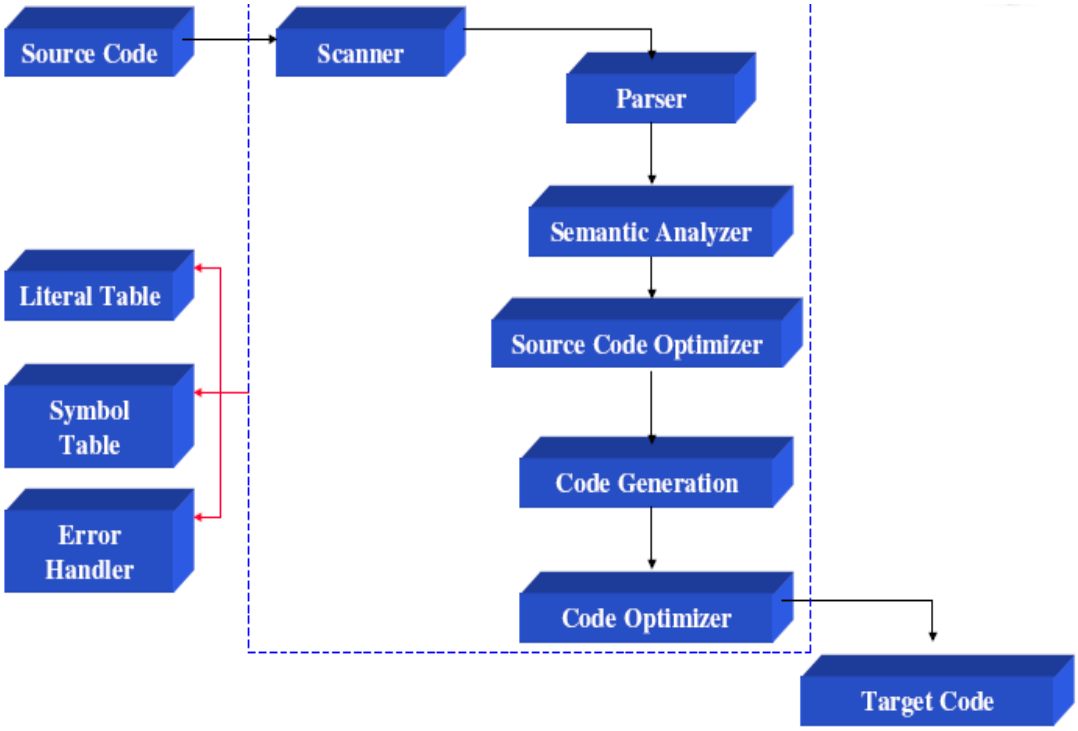
• Compiler design has a **long** history (FORTRAN 1958)

a. lots of **experience** on how to structure compilers
b. lots of **existing** designs to study (many freely available)

- We use natural languages to communicate
- We use programming languages to speak with computers

Components of a Compiler

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation



The Scanner

a. Performs **lexical analysis**

b. Collects character sequences into **tokens**

c. Example: **a[index] = 4 + 2**

Tokens:		
	a	identifier
	[	left bracket
	index	identifier
	]	right bracket
	=	assignment
	4	number
	+	plus sign
	2	number

The scanner *may* also:

1. Enter identifiers into the symbol table
2. Enter literals (numeric constants and strings) into the literal table

Lexical Analysis (LA)

Maradona kicks the ball

Token Generation:

- **Maradona**
- **kicks**
- **the**
- **ball**

Who performs this ?

Lexical Analysis

X = Y + 30

Token Generation:

- **X** **id_1**
- **=** **operator**
- **Y** **id_2**
- **+** **operator**
- **30** **literal/constant**

What other functions Scanner can perform ?

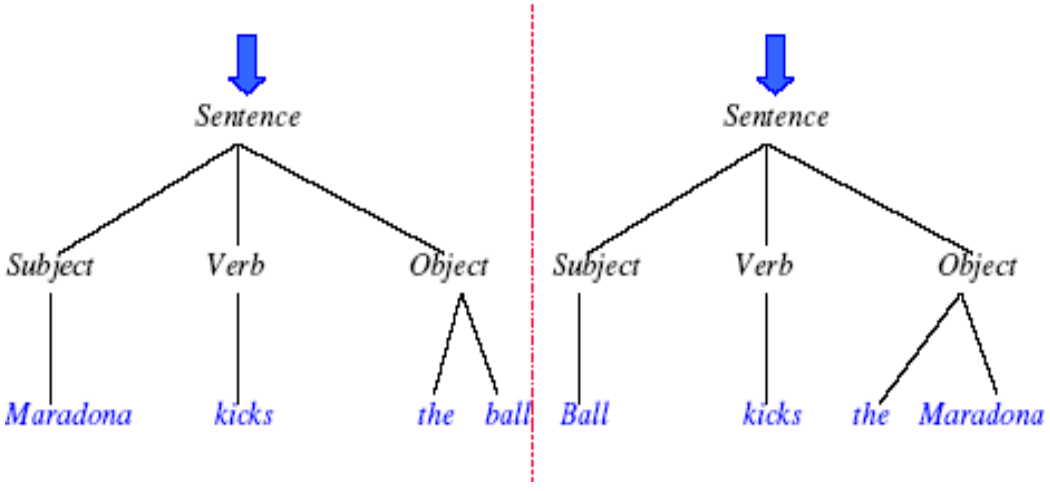
Syntax Analysis (SA)

The Parser
a. Performs **syntax analysis**
b. Builds a **parse tree** or **syntax tree**

Parse tree for **a[index] = 4 + 2**

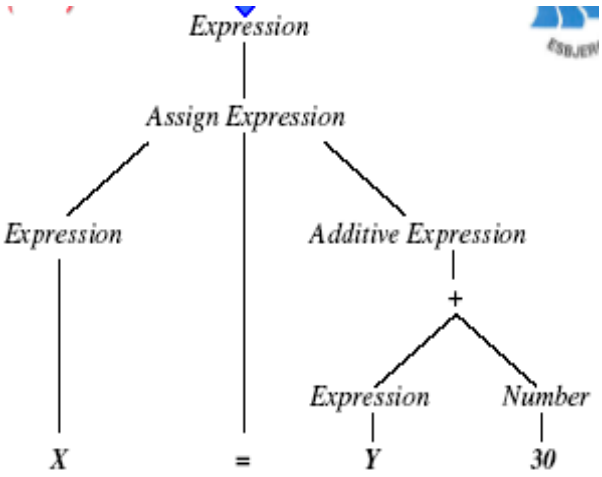
Structure of the program is determined by SA. Some thing similar to grammatical analysis.

Maradona kicks the ball **Ball kicks the Maradona**

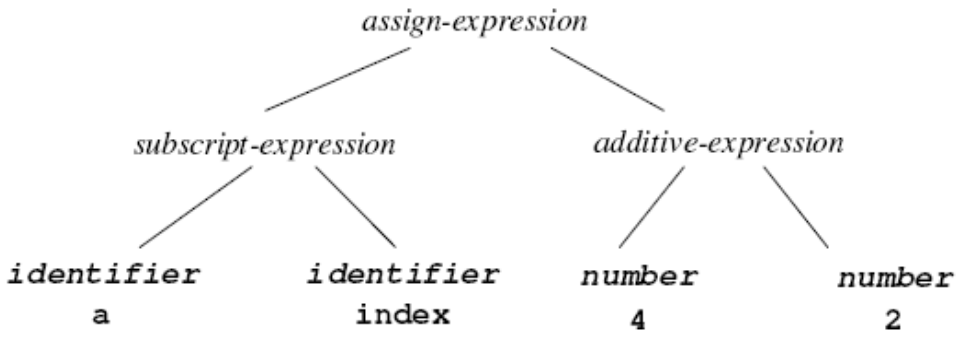


An **abstract syntax tree** is a more concise version of a parse tree.

X = Y + 30



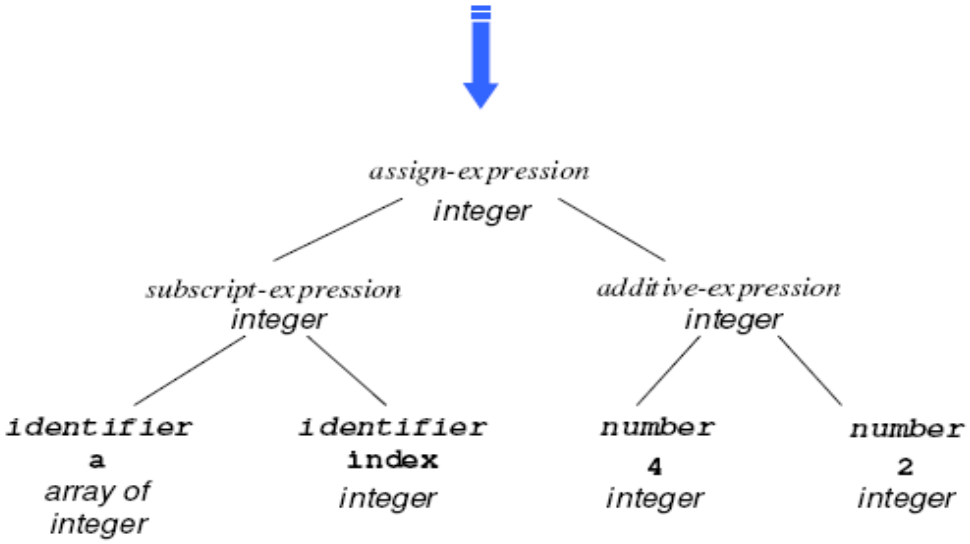
Some time syntax tree is also called as Abstract Syntax tree and could be a "trimmed" version of the parse tree with only essential information:
For Example: a[index] = 4 + 2



Semantic Analyzer

This attach meaning of tokens; For example to the same expression

a[index] = 4 + 2



Intermediate Code Generation

Same example: X = Y + 30

Temp1 = 30

Temp2 = Y

Temp3 = Temp2 + Temp1

X = Temp3

Code Optimization

Same example: X = Y + 30

Temp1 = 30

Temp2 = Y

X = Temp2 + Temp1

Code Generation

Same example: X = Y + 30

Movei Y, r1

Addi 30, r1

Movei r1, X

Algorithmic Tools

- **Token:**
  - Using Regular Expressions.
- **Scanner:**
  - Implementation of finite state machine to recognize tokens.
- **Parser:**
  - An Automaton (i.e. uses a stack), based on grammar rules in a standard format (BNF -- Backus Naur Form).
- **Semantic Analyzer and Code Generator:**
  - Recursive evaluators based on semantic rules for attributes (properties of language constructs).

Error handling

- One of the difficult part of a compiler to design.
- Must handle a wide range of errors
- Must handle multiple errors.
- Must not get stuck.
- Must not get into an infinite loop.

Kinds of errors

- **Syntax**:

```
if (x == 0) y + = z + r; }
```

- **Semantic**:

```
int x = "Hello, world!";
```

- **Runtime**:

```
int x = 2;
...
double y = 3.14159 / (x – 2);
```

Error Handling Requirements

- A compiler must handle syntax and semantic errors, but not runtime errors (whether a runtime error will occur is million dollar question).

- Sometimes a compiler is required to generate code to catch runtime errors and handle them in some graceful way (either with or without exception handling). This, too, is often difficult.

Major Compiler Data Structures

a. Tokens
   1. Represented as an enumerated type
   2. May require other information:
      a. Spelling of identifier
      b. Numeric value
   3. Scanner needs generate only one token at a time (single symbol lookahead)

b. Syntax Tree
   1. A linked structure built by the parser, with information added by the semantic analyzer
   2. Each node is a record whose fields contain information about the syntactic construct which the node represents
   3. Node may be represented using a variant record

c. Symbol Table
   1. Keeps information about identifiers
   2. Efficient insertion and lookup is required → hash table or tree structure may be used
   3. Several tables may be maintained in a list or stack

d. Literal Table

1. Stores constants and strings used in a program
2. Data in literal table applies globally to a program → deletions are not necessary

e. Intermediate code
   1. Could be kept in an array, temporary file, or linked list
   2. Representations include P-code and 3-address code

f. Temporary files
   1. May not be used if memory constraints are not a problem
   2. **Backpatching** of addresses necessary during translation

FUNDAMENTALS OF LANGUAGE SPECIFICATION

A specification of the source language forms the basis of source program analysis. In this section, we shall discuss important lexical, syntactic and semantic features of a programming language.

Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. This section discusses key concepts and notions from formal language grammars. A language L can be considered to be a collection of valid sentences.

Each sentence can be looked upon as a sequence of words, and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as *a. formal language.* A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal symbols, alphabet and strings

The *alphabet* of L, denoted by the Greek symbol Z, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols in Z.

A symbol in the alphabet is known as a *terminal symbol* (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g.

$\Sigma \cong$ {a, b, ….z, 0,1....9}

Here the symbols {, ',' and} are part of the notation. We call them *met symbols* to differentiate them from terminal symbols. Throughout this discussion we assume that met symbols are distinct from the terminal symbols. If this is not the case, i.e. if a terminal symbol and a met symbol are identical, we enclose the terminal symbol in quotes to differentiate it from the meta symbol. For example, the set of punctuation symbols of English can be defined as

{:,;’,’-,...}

Where ',' denotes the terminal symbol 'comma'.

A *string* is a finite sequence of symbols. We will represent strings by Greek symbols-α β γ, etc. Thus α = axy is a string over Σ . The length of a string is the Number of symbols in it. Note that the absence of any symbol is also a string, the *null string* . The *concatenation* operation combines two strings into a single string.

To evaluate an HLL program it should be converted into the Machine language. A compiler performs another very important function. This is in terms of the diagnostics.

I.e. error – detection capability.

The important tasks of a compiler are:

Translating the HLL program input to it.

Providing diagnostic messages whenever specifications of the HLL

Assemblers & compilers

Assembler is a translator for the lower level assembly language of computer, while compilers are translators for HLLs.

An assembly language is mostly peculated to a certain computer, while an HLL is generally machined independent & thus portable.

Overview of the compilation process:

The process of compilation is:

Analysis of + Synthesis of = Translation of

Source Text Target Text Program
Source text analysis is based on the grimmer of the source of the source language.
The component sub – tasks of analysis phase are:
Syntax analysis, which determine the syntactic structure of the source statement.
Semantic analysis, which determines the meaning of a statement, once its grammatical structures become known.

The analysis phase

The analysis phase of a compiler performs the following functions.

Lexical analysis

Syntax analysis

Semantic analysis

Syntax analysis determines the grammatical or syntactic structure or the input statement & represents it in an intermediate form from which semantic analysis can be performed.

A compiler must perform two major tasks:
The Analysis of a source program & the synthesis of its corresponding object program.

The analysis task deals with the decomposition of the source program into its basic parts using these basic parts the synthesis task builds their equivalent object program modules. A source program is a string of symbols each of which is generally a letter, a digit or a certain special constants, keywords & operators. It is therefore desirable for the compiler to identify these various types as classes.

The analysis task deals with the decomposition of the source program into its basic parts using these basic parts the synthesis task builds their equivalent object program modules. A source program is a string of symbols each of which is generally a letter, a digit or a certain special constants, keywords & operators. It is therefore desirable for the compiler to identify these various types as classes.

The source program is input to a lexical analyzer or scanner whose purpose is to separate the incoming text into pieces or tokens such as constants, variable name, keywords & operators.
In essence, the lexical analyzer performs low- level syntax analysis performs low-level syntax analysis.
For efficiency reasons, each of tokens is given a unique internal representation number.

Introduction to Assemblers and Assembly Language

Encoding instructions as binary numbers is natural and efficient for computers. Humans, however, have a great deal of difficulty understanding and manipulating these numbers. People read and write symbols (words) much better than long sequences of digits. This lecture describes the process by which a human-readable program is translated into a form that a computer can execute, provides a few hints about writing assembly programs, and explains how to run these programs on SPIM,

What is an assembler ?

A tool called an ***assembler*** translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer's 0s and 1s that simplifies writing and reading programs. Symbolic names for operations and locations are one facet of this representation. Another facet is programming facilities that increase a program's clarity.

An assembler reads a single assembly language ***source file*** and produces an ***object file*** containing machine instructions and bookkeeping information that helps combine several object files into a program. Figure (1) illustrates how a program is built. Most programs consist of several files—also called ***modules***— that are written, compiled, and assembled independently. A program may also use prewritten routines supplied in a ***program library*** . A module typically contains *References* to subroutines and data defined in other modules and in libraries. The code in a module cannot be executed when it contains *unresolved References* to labels in other object files or libraries. Another tool, called a *linker,* combines a collection of object and library files into an *executable file* , which a computer can run.

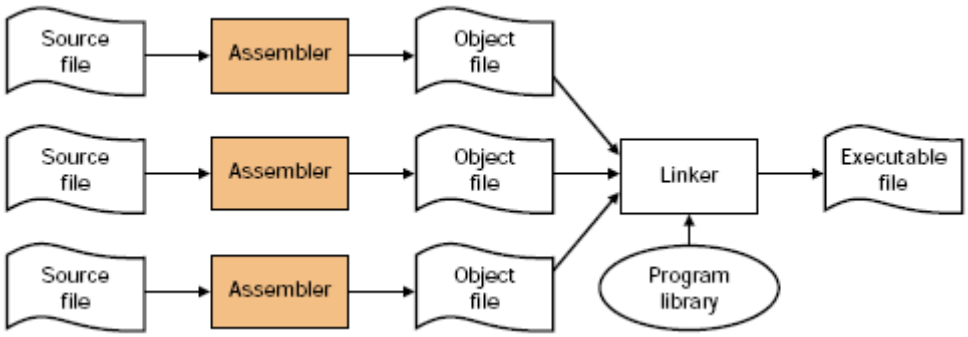


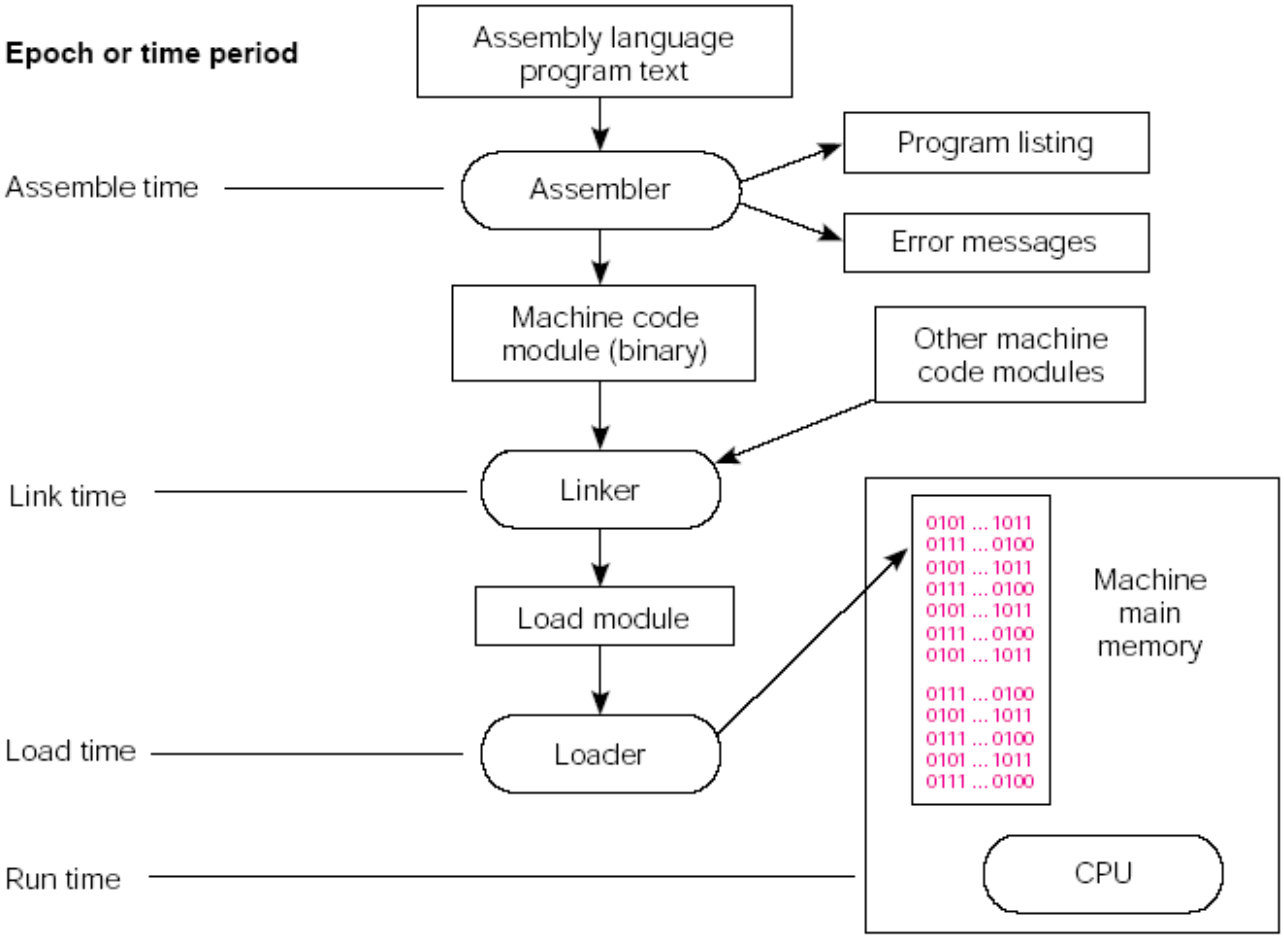
FIGURE 1: The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

1) **Assembler = a program to handle all the tedious mechanical translations**

2) **Allows you to use:**
   - symbolic opcodes
   - symbolic operand values
   - symbolic addresses

3) **The Assembler**
   - keeps track of the numerical values of all symbols
   - translates symbolic values into numerical values

4)Time Periods of the Various Processes in Program Development



5) The Assembler Provides:

a. Access to all the machine's resources by the assembled program. This includes access to the entire instruction set of the machine.
b. A means for specifying run-time locations of program and data in memory.
c. Provide symbolic labels for the representation of constants and addresses.
d. Perform assemble-time arithmetic.
e. Provide for the use of any synthetic instructions.
f. Emit machine code in a form that can be loaded and executed.
g. Report syntax errors and provide program listings
h. Provide an interface to the module linkers and program loader.
i. Expand programmer defined macro routines.

Syntax: Label OPCODE Op1, Op2, ... ;Comment field

Pseudo-operations (sometimes called "pseudos," or directives) are "opcodes" that are actually instructions to the assembler and that do not result in code being generated.

Assembler maintains several data structures

- Table that maps text of opcodes to op number and instruction format(s)
- "Symbol table" that maps defined symbols to their value

Assembly program structure

- Each line is of the form:

Program line		
Label	Operation	Operands ; Comment

- Each field can be omitted
- The remainder of the line after (;) is ignored
- Example:

Comments

```
;; FIRST.ASM    Our first Assembly Language Program.
        .MODEL  SMALL
        .586    ; Allows Pentium Instructions. Must come after .MODEL

        .STACK  100h
        .....
```

Disadvantages of Assembly

• programmer must manage movement of data items between memory locations and the ALU.

• programmer must take a "microscopic" view of a task, breaking it down to manipulate individual memory locations.
• assembly language is machine-specific.

• statements are not English-like (Pseudo-code)

Directives Assembler

1. **Directives are commands to the Assembler**
2. **They tell the assembler what you want it to do, e.g.**
    a. Where in memory to store the code
    b. Where in memory to store data
    c. Where to store a constant and what its value is
    d. The values of user-defined symbols

Object File Format

Assemblers produce object files. An object file on Unix contains six distinct sections (see Figure 3):

- The *object file header* describes the size and position of the other pieces of the file.
- The *text segment* contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.
- The *data segment* contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The *relocation information* identifies instructions and data words that depend on absolute addresses. These references must change if portions of the program are moved in memory.
- The *symbol table* associates addresses with external labels in the source file and lists unresolved references.
- The *debugging information* contains a concise description of the way in which the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

The assembler produces an object file that contains a binary representation of the program and data and additional information to help link pieces of a program. This relocation information is necessary because the assembler does not know which memory locations a procedure or piece of data will occupy after it is linked with the rest of the program. Procedures and data from a file are stored in a contiguous piece of memory, but the assembler does not know where this memory will be located. The assembler also passes some symbol table entries to the linker. In particular, the assembler must record which external symbols are defined in a file and what unresolved references occur in a file.

Macros

Macros are a pattern-matching and replacement facility that provide a simple mechanism to name a frequently used sequence of instructions. Instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

The 2-Pass Assembly Process

- **Pass 1:**

1. Initialize location counter (assemble-time "PC") to 0
2. Pass over program text: enter all symbols into symbol table
    a. May not be able to map all symbols on first pass
    b. Definition before use is usually allowed
3. Determine size of each instruction, map to a location

    a. Uses pattern matching to relate opcode to pattern
    b. Increment location counter by size
    c. Change location counter in response to ORG pseudo

- **Pass 2:**

1. Insert binary code for each opcode and value
2. “Fix up” forward references and variable-sizes instructions
    - Examples include variable-sized branch offsets and constant fields

Architecture and Organization

• **Architecture** is the design of the system visible to the assembly level programmer.

– What instructions

– How many registers

– Memory addressing scheme

• **Organization** is how the architecture is implemented.

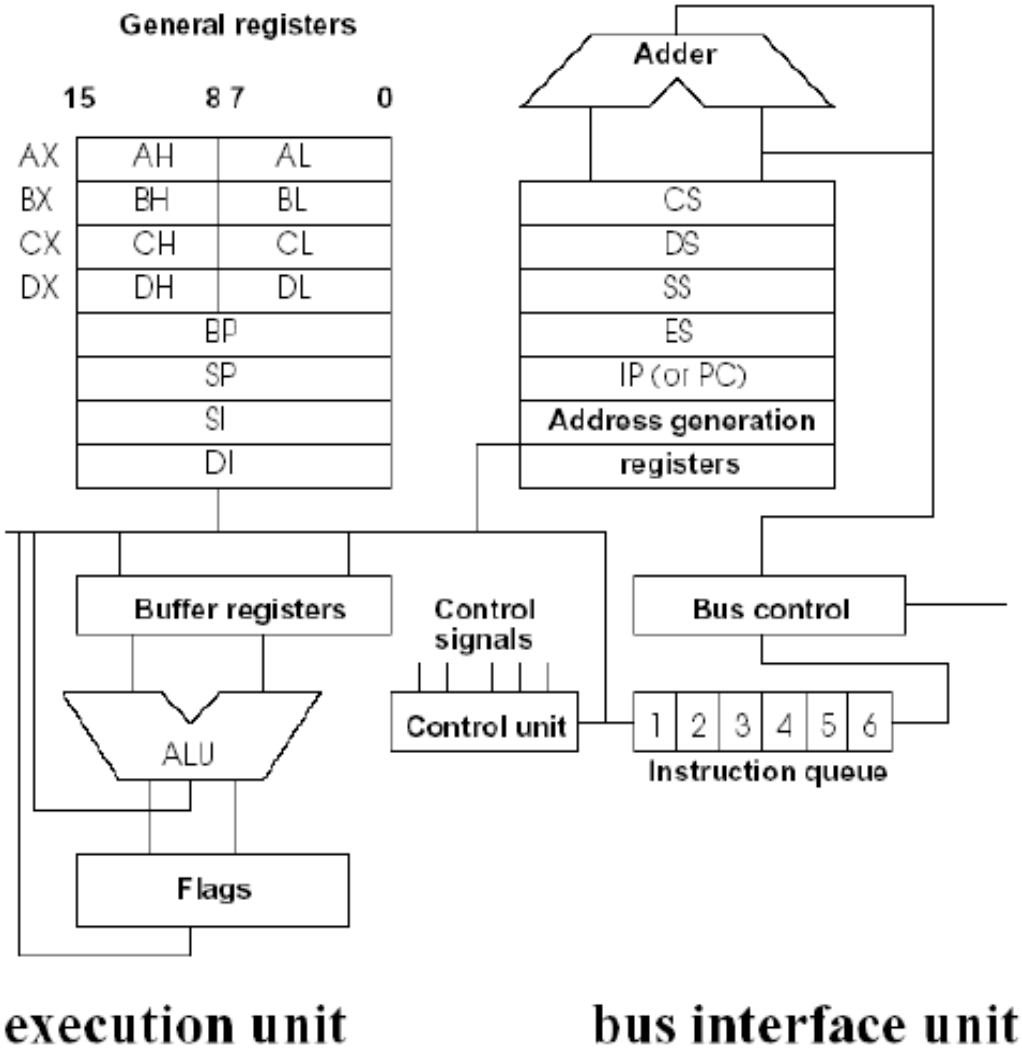
– How much cache memory

– Microcode or direct hardware

– Implementation technology

(8086 Architecture)

1. Hardware Organization



On the structural scheme of the i8086 processor we can see two separate asynchronous processing units.
The execution unit (EU) executes instructions;
the bus interface unit (BIU) fetches instructions, reads operands, and writes results.
The two units can operate almost independently of one another and are able, under most circumstances, to extensively overlap instruction fetch with execution. The result is that, in most cases, the time normally required to fetch instructions "disappears" because the EU executes instructions that have already been fetched by BIU. Of course nothing special, but remember the time when i8086 was designed.

Execution Unit
The execution unit consists of general registers, buffer registers, control unit, arithmetic/logic unit, and flag register. The ALU maintains the CPU status and control flags and manipulates the general registers and instruction operands. The EU is not connected to the system bus. It obtains instructions from a queue maintained by the BIU. Likewise, when an instruction requires access to memory or to a peripheral device, the EU requests the BIU to obtain or store the data. The EU manipulates only with 16-bit addresses (effective addresses). An address relocation that enables the EU access to the full megabyte is performed by BIU.

Bus Interface Unit
The bus interface unit performs all bus operations for the EU. Data is transferred between the CPU and memory or I/O devices upon demand from the EU. During periods when the EU is busy executing instructions, the BIU fetches more instructions from memory. The instructions are stored in an internal RAM array called the instruction stream queue. The 8086 queue can store up to six instruction bytes. This allows the BIU to keep the EU supplied with prefetched instructions under most conditions. The BIU of 8086 does not initiate a fetch until there are two empty bytes in its queue. The BIU normally obtains two instruction bytes per fetch, but if a program transfer forces fetching from an odd address, the 8086 BIU automatically reads one byte from the odd address and then resumes fetching two-byte words from the subsequent even addresses. Under most circumstances the queue contains at least one byte of the instruction stream and the EU does not have to wait for instructions to be fetched. The instructions in the queue are the next logical instructions so long as execution proceeds serially. If the EU executes an instruction that transfers control to another location, the BIU resets the queue, fetches the

instruction from the new address, passes it immediately to the EU, and then begins refilling the queue from the new
cation. In addition, the BIU suspends instruction fetching whenever the EU requests a memory or I/O read or write (except that a fetch already in progress is completed before executing the EU's bus request).

The Details of the Architecture

Registers

The general registers of the 8086 are divided into two sets of four 16-bit registers each. The data registers and the pointer and index registers. The data registers' upper and lower halves are separately addressable. In other words, each data register can be used interchangeably as a 16-bit register or as two 8-bit registers. The data registers can be used without constraint in most arithmetic and logic operations. Some instructions use certain registers implicitly thus allowing compact yet powerful encoding. The pointer and index registers can be used only as 16-bit registers. They can also participate in most arithmetic and logic operations. In fact all eight general registers fit the definition of "accumulator" as used in first and second generation microprocessors. The pointer and index registers (except BP) are also used implicitly in some instructions. The segment registers contain the base addresses of logical segments in the 8086 memory space. The CPU has direct access to four segments at a time. The CS register points to the current code segment; instructions are fetched from this segment. The SS points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data segment; it generally contains program variables. The ES points to the current extra segment; which also is typically used for data storage. The IP (instruction pointer) is updated by the BIU so that it contains the offset of the next instruction from the beginning of the current code segment. During normal execution IP contains the offset of the next instruction to be *fetched* by the BIU; whenever IP is saved on the stack, however, it is first automatically adjusted to point to the next instruction to be *executed.* Programs do not have direct access to the IP.

There are eight 16-bit general registers.

The data registers:
AX (AH and AL)
BX (BH and BL)
CX (CH and CL)
DX (DH and DL)

The pointer and index registers: BP, SP, SI, DI
The upper and lower halves of the data registers are separately addressable. Memory space is divided into logical segments up to 64k bytes each. The CPU has direct access to four segments at a time; their base addresses are contained in the segment registers
CS, DS, SS, ES.
CS = code segment;
DS = data segment;
SS = stack segment;
ES = extra segment;
Flags are maintained in the flag register depending on the result of the arithmetic or logic operation. A group of instructions is available that allows a program to alter its execution depending on the state of the flags, that is, on the result of a prior operation. There are:
• **AF** (the auxiliary carry flag) used by decimal arithmetic instructions. Indicates carry out from the low nibble of the 8-bit quantity to the high nibble, or borrow from the high nibble into the low nibble.
• **CF** (the carry flag) indicates that there has been carry out of , or a borrow into, the high-order bit of the result.
• **OF** (the overflow flag) indicates that an arithmetic overflow has occurred.
• **SF** (the sign flag) indicates the sign of the result (high-order bit is set, the result is negative).
• **PF** (the parity flag) indicates that the result has an even parity, an even number of 1-bits.
• **ZF** (the zero flag) indicates that the result of the operation is 0.
Three additional control flags can be set and cleared by programs to alter processor operations:
• **DF** (the direction flag) causes string instructions to auto-decrement if it is set and to auto-increment if it is cleared.
• **IF** (the interrupt enable flag) allows the CPU to recognize external interrupts.
• **TF** (the trap flag) puts the processor into single-step mode for debugging.

Memory Organization

The 8086 can accommodate up to 1,048,576 bytes of memory. From the storage point of view, the memory space is organized as array of 8-bit bytes. Instructions, byte data and word data may be freely stored at any byte address without regard for alignment. The Intel convention is that the most-significant byte of word data is stored in the higher memory location. A special class of data (pointers) is stored as double words. The lower addressed word of a pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored following the above convention. The i8086 programs "view" the megabyte of memory space as a group of segments that are defined by the application. A segment is a logical unit of memory that may be up to 64k bytes long. Each segment is made up of contiguous memory locations and is an independent separately addressable unit. The software must assign to every segment a base address, which is its starting location in the memory space. All segments begin on 16-byte memory boundaries called paragraphs. The segment registers point to the four currently addressable segments. To obtain code and data from other segments, program must change the content of segment registers to point to the desired segments. Every memory location has its physical address and its logical address. A physical address is the 20-bit value that uniquely identifies each byte location in the memory space. Physical addresses may range from 0H to FFFFFH. All exchanges between the CPU and memory components use physical addresses. However, programs deal with logical rather than physical addresses. A logical address consists of a segment base and offset value. The logical to physical address translation is done by BIU whenever it accesses memory. The BIU shifts segment base by 4 to the left and adds the offset to this value. Thus we obtain 20-bit physical address and get the explanation for 16-byte memory boundaries for the segment base beginning. The offset of the memory variable is calculated by the EU depending on the addressing modes and is called the operand's effective address (EA). Stack is implemented in memory and is located by the stack segment register and the
stack pointer register. An item is pushed onto the stack by *decrementing* SP by 2 and writing the item at the new top of stack (TOS). An item is popped off the stack by copying it from TOS and then *incrementing* SP by 2.

The memory locations 0H through 7FH are dedicated for interrupt vector table, and locations FFFF0H through FFFFFH are dedicated for system reset.

Input/Output

The 8086 I/O space can accommodate up to 64k 8-bit ports or up to 32k 16-bit ports. The IN and OUT instructions transfer data between the accumulator and ports located in I/O space. The I/O space is not segmented; to access a port, the BIU simply places the port address on the lower 16 lines of the address bus. I/O devices may also be placed in the 8086 memory space. As long as the devices respond like the memory components, the CPU does not know the difference. This adds programming flexibility, and is paid by longer execution of memory oriented instructions.

Processor Control and Monitoring

The interrupt system of the 8086 is based on the interrupt vector table which is located from 0H through 7FH (dedicated) and from 80H through 3FFH (user available). Every interrupt is assigned a type code that identifies it to the CPU. By multiplying (type * 4), the CPU calculates the location of the correct entry for a given interrupt. Every table entry is 4 bytes long and contains the offset and the segment base (pointer) of the corresponding interrupt procedure that should be executed. After system reset all segments are initialized to 0H except CS which is initialized to FFFFH. Since, the processor executes the first instruction from absolute memory location FFFF0H. This location normally contains an intersegment direct JMP instruction whose target is the actual beginning of the system program.

Software Organization

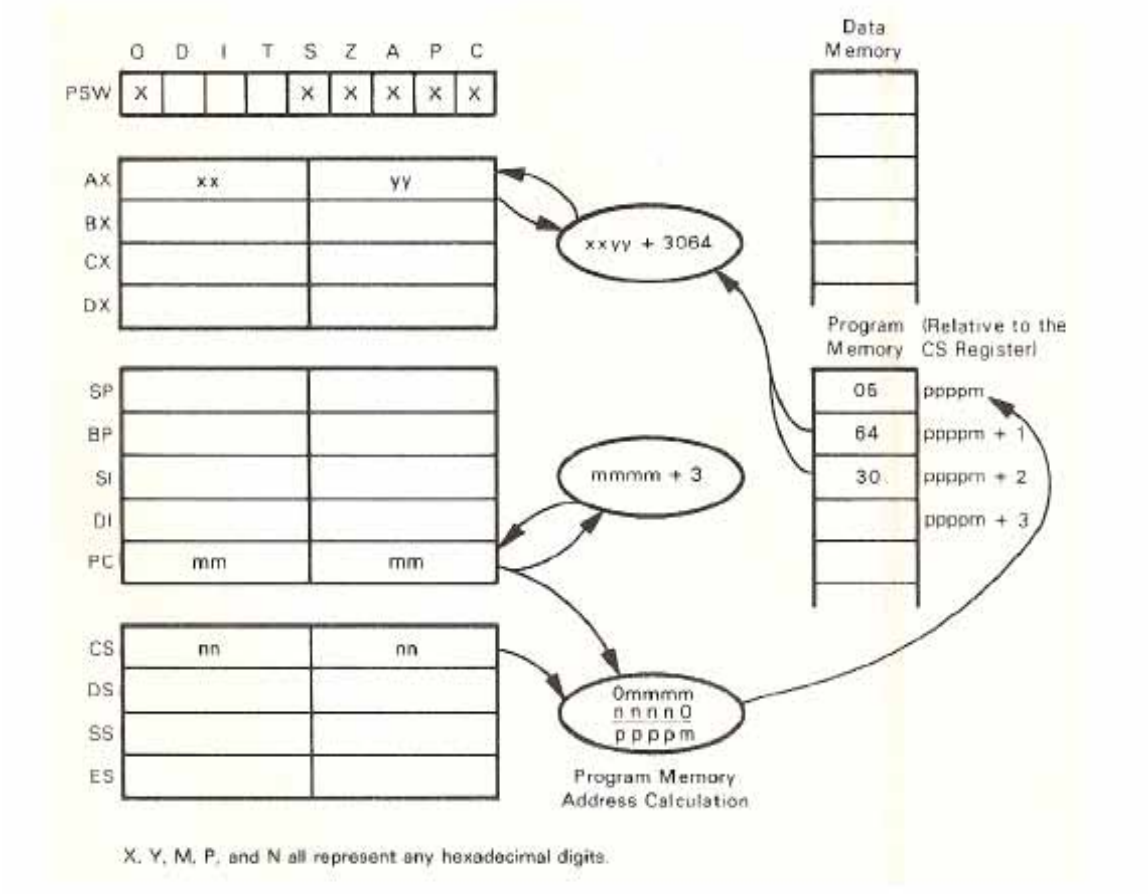
Instruction Set

The 8086 instruction set from programmer's point of view contains about 100 instructions. However the number of machine instructions is more then 3800. For example MOV instruction has 28 different machine forms. On the functional level we can divide the instruction set on :

1. Data transfer instructions (MOV, XCHG, LEA, ...),
2. Arithmetic instructions (ADD, SUB, INC, DEC, ...),
3. Bit manipulation instructions (AND, SHR, ROR, ...),
4. String instructions (MOVS, LODS, REP, ...),
5. Program transfer instructions (CALL, JMP, JZ, RET, ...),
6. Interrupt instructions (INT, INTO, IRET),
7. Processor control instructions (CLC, STD, HLT, ...).

Data Memory Addressing Modes:
The 8086 offers a wide variety of addressing; we will condense it into six basic operation. These options are:
1- Immediate
2- Direct
3- Direct, Indexed
4- Implied
5- Base Relative
6- Stack

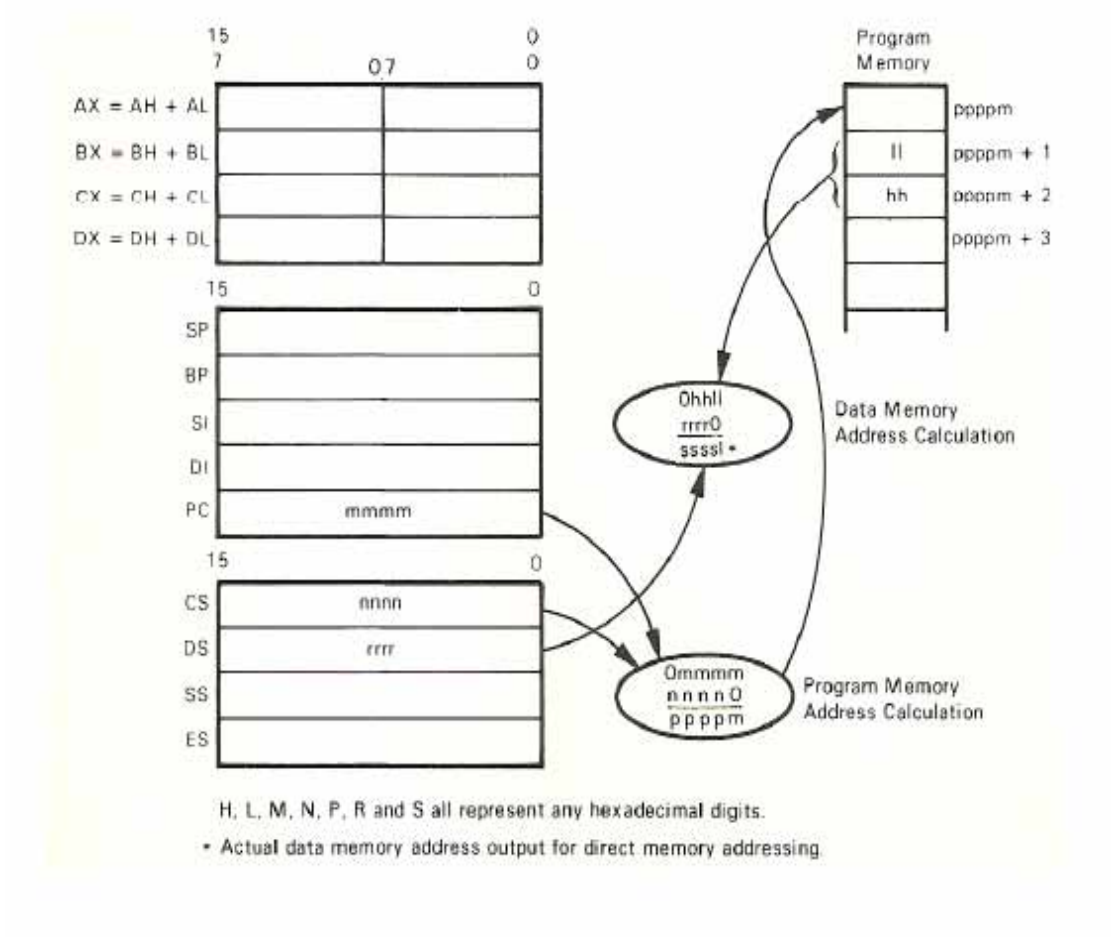
Immediate Memory Addressing:
In this form of addressing, one of the operands is present in the byte(s) immediately following the instruction object code (op-code). If addressing bytes follow the op-code, then the immediate data will follow the addressing bytes. For example:
ADD AX, 3064H
Requests the assembler to generate an ADD instruction which will add 3064 to the AX register. This may be illustrated as follows:



Note that the 16-bit immediate operand, when stored in program memory, has the low-order byte preceding the high-order byte. This is consistent

with the way the 8086A stores immediate operands in program memory. In addition, this is consistent with the way the 8086 stores 16-bit operands in data memory. When a 16-bit store is performed, the low-order 8 bits of data are stored into the low-order memory byte, and the high-order 8 bits of data are stored into the succeeding memory byte.
In this example, the two bytes immediately following the op-code for the ADD to AX instruction are added to the AX register.

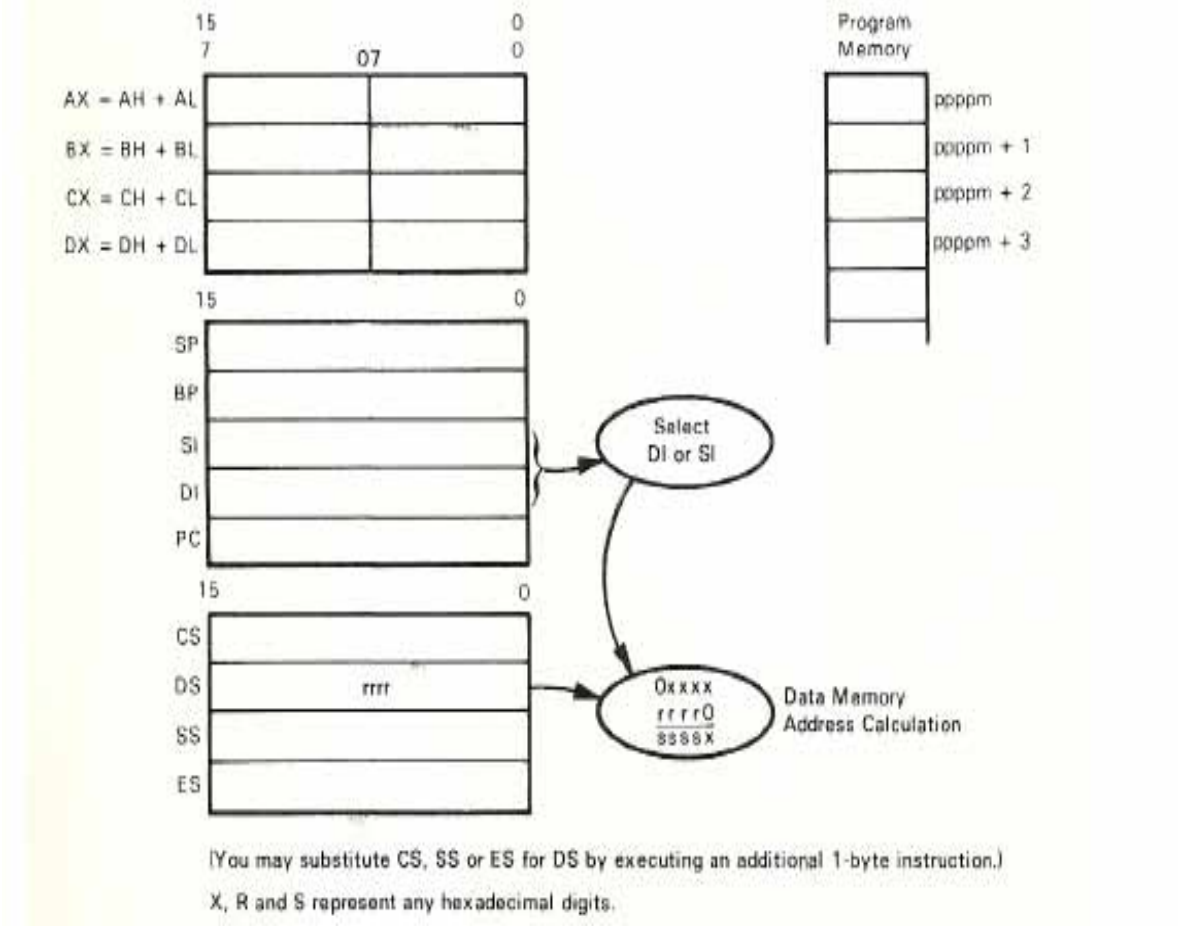
Direct Memory Addressing:
The 8086 implements straight forward direct memory addressing by adding a 16-bit displacement, provided by two object code bytes, to the data segment register. The sum becomes the actual memory address. This may be illustrated as follows:



Note that a 16-bit address displacement, when stored in program memory, has the low-order byte preceding the high-order byte. This is consistent with the way the 8080A stores addresses in program memory.
DS must provide the segment base address when addressing data memory directly, as illustrated above.

Direct, Indexed Memory Addressing:

Direct, indexed addressing is allowed by specifying the SI or DI register as an index register. You have the option of adding an 8-bit or 16-bit displacement to the contents of the specified index register in order to generate the effective address. A16-bit displacement is stored in two object code bytes; the low-order byte of the displacement precedes the high-order byte of the displacement, as illustrated for direct memory addressing. If an 8-bit displacement is specified, then the high-order bit of the low-order byte is propagated into the high-order byte to create a 16-bit displacement this may be illustrated a follows:



Implied Memory Addressing:

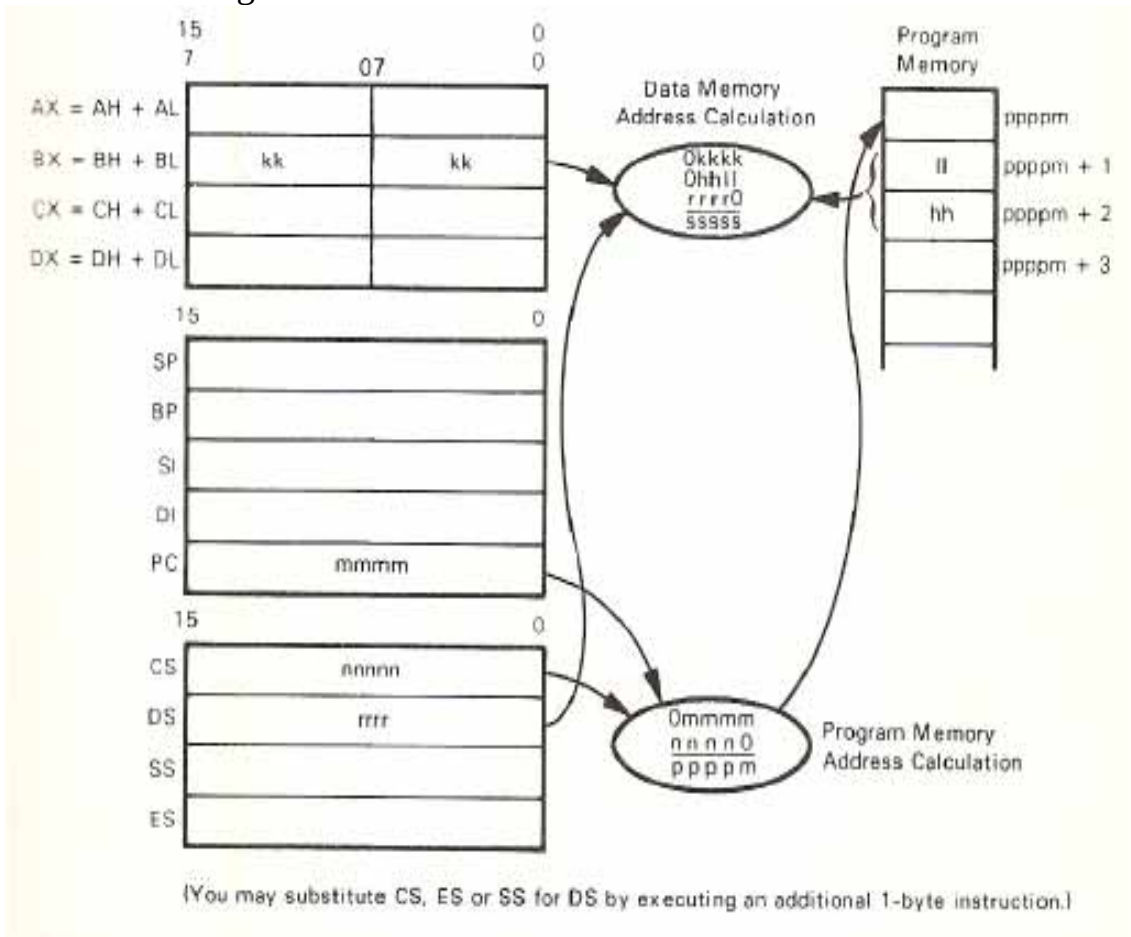
Implied memory addressing is implemented on the 8086 as a degenerate version of a direct, indexed memory addressing. If you do not specify a displacement when using the direct, index addressing mode, then you have, in effect, implied memory addressing via the SI or DI register. The may be illustrated as follows:

Base Relative Addressing:

The 8086 implements base relative addressing in two ways:

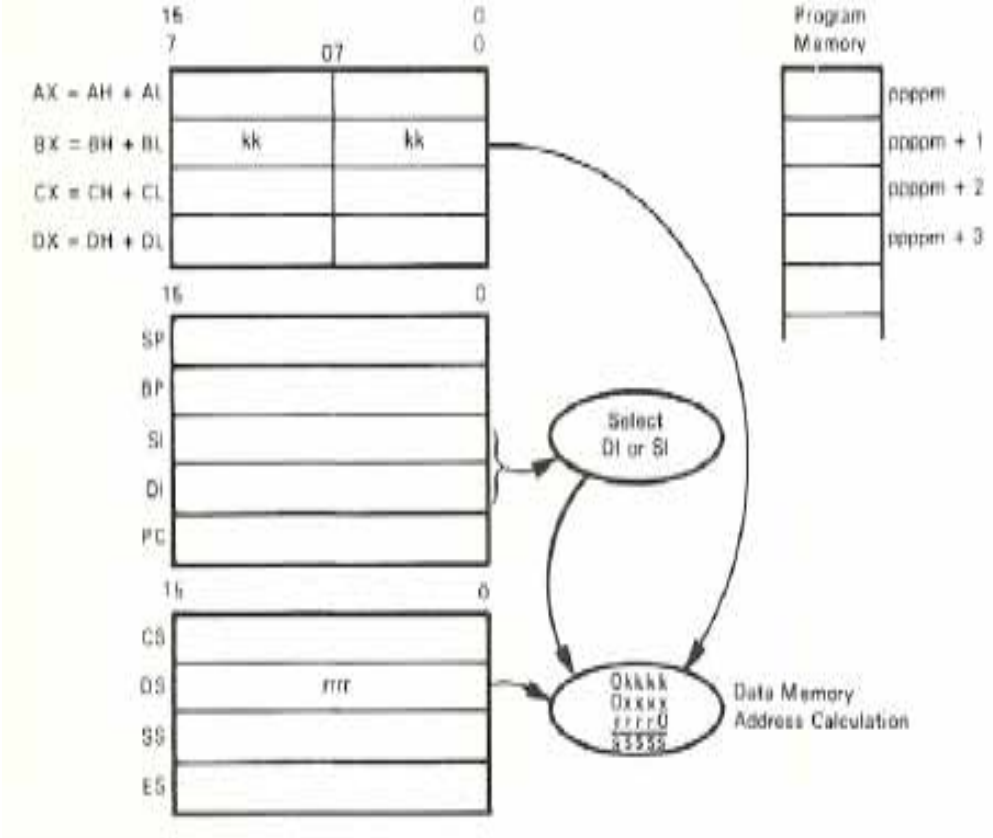
- Data memory base relative addressing, which is within the DS segment (data memory)
- Stack base relative addressing, which is in the SS segment (stack memory)

Data memory base relative addressing uses the BX register contents to provide the base for the effective address. All of the data memory addressing options thus far described, with the exception of immediate addressing ,are available with base relative data memory addressing. In effect, base relative data memory addressing merely adds the contents of the BX register to the effective memory address which would otherwise have been generated. Here, for example, is an illustration of base relative direct addressing:

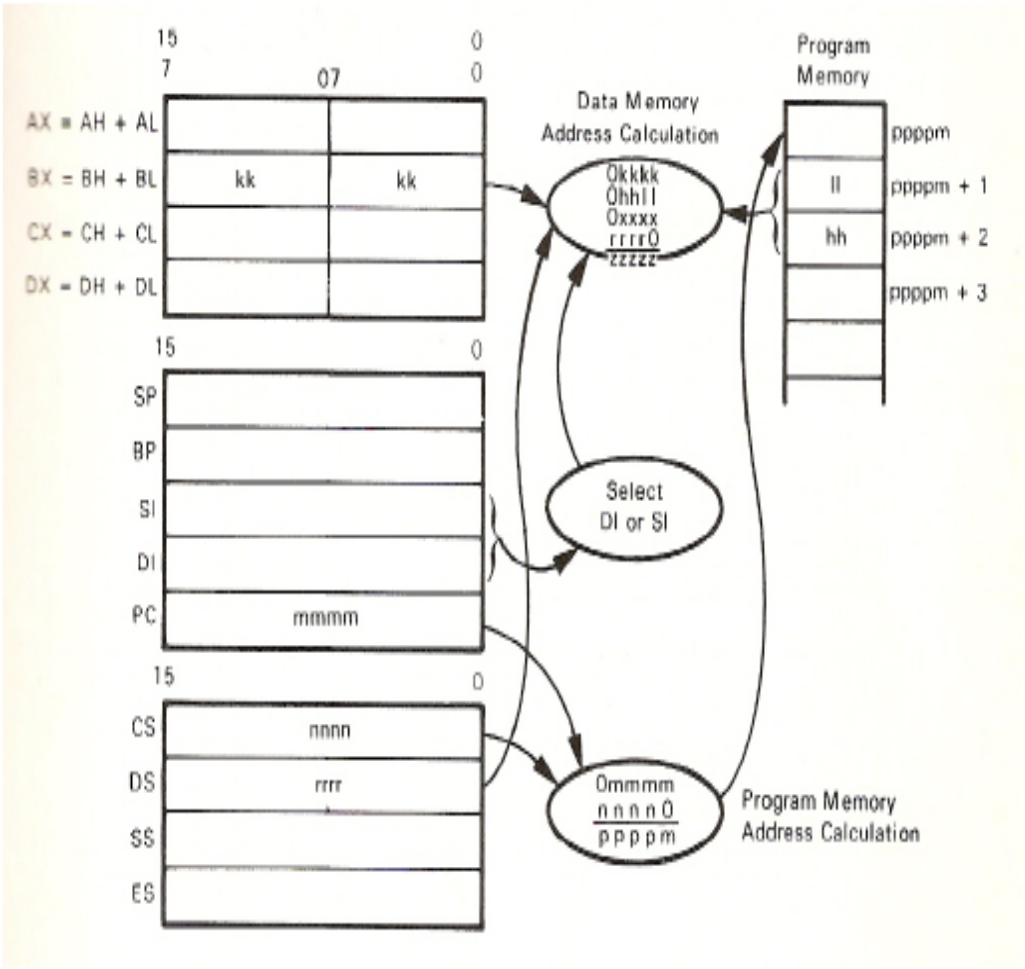


Simple, direct addressing, which we described earlier, always generated a 16-bit displacement. Base relative, direct addressing allows the displacement, illustrated above as HHLL, to be a 16-bit displacement, an 8-bit displacement with sign extended, or no displacement at all.

Base relative implied memory addressing simply adds the contents of the BX register to the selected index register in order to compute the effective memory address. This may be illustrated as follows:



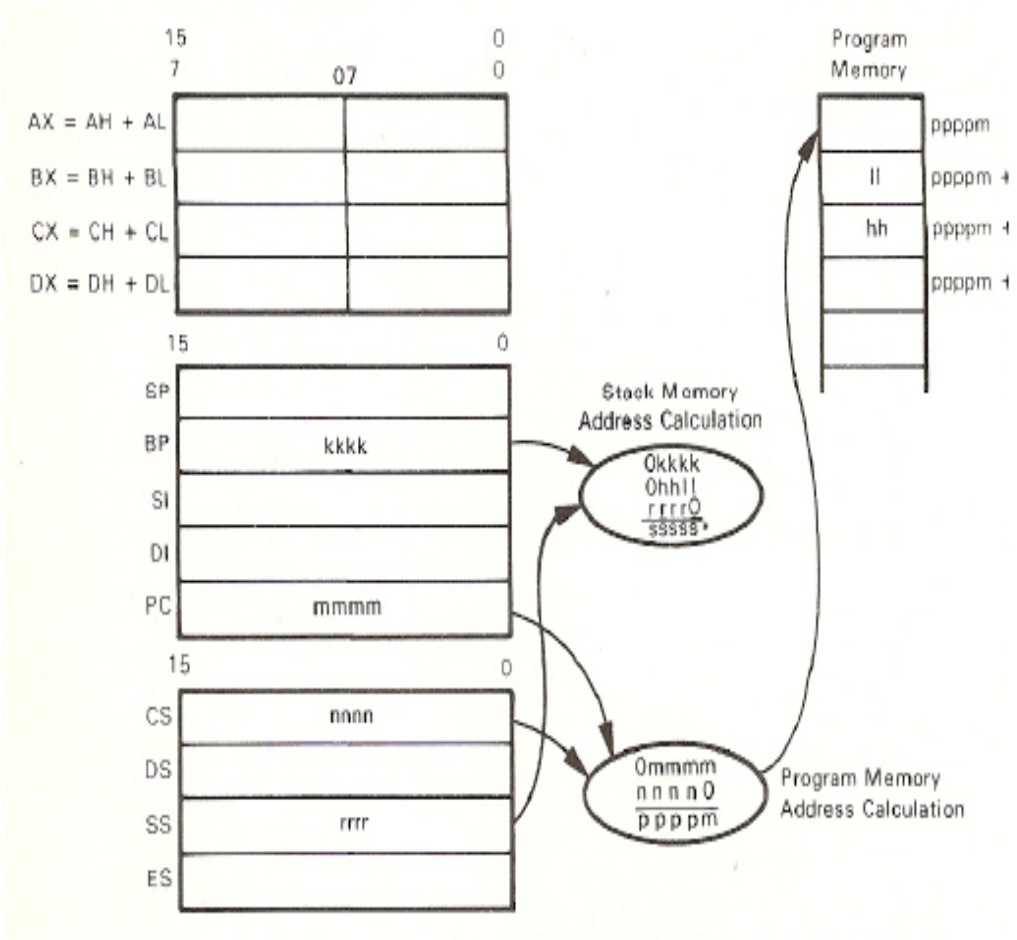
Base relative, direct, indexed data memory addressing may appear to be complicated, but in fact it is not. We simply add the contents of the BX register to the effective memory address, as computed for normal direct, indexed addressing. Thus, base relative, direct, indexed data memory addressing may be illustrated as follows:



The index xxxx in the illustration above is optional. Base relative, direct memory addressing is also available. In this instance neither SI or DI will contribute to the address computation, and 0xxxx must be remove from the illustration.

Stack Memory Addressing:

The 8086 also has stack memory addressing variations of the base relative, data memory addressing options just described. In this case, however, the BP register is used as the base register. Here, for example, is base relative, direct stack addressing:



In the illustration above, the displacement HHLL is present, either as a 16-bit displacement or as 8-bit displacement with sign extended. Base relative stack memory addressing requires a displacement be specified, even if zero.

The more commonly used instructions:

1. Arithmetic instructions:

These instructions are used for arithmetic operation on the source and destination operands.

***ADD ac , data** (add immediate data to AX register)

This instruction is used to add the immediate data present in the succeeding program memory byte (s) to the AL (8-bit operation) or AX (16-bit operation) register.

***ADD mem/reg , data** (add immediate data to register or memory location).

***ADC mem/reg1,mem/reg2**

Add data with carry from .register to register

.register to memory

.memory to register

Add the contents of the register or memory location specified by mem/reg2 and the carry status to the contents of the register or memory location specified by mem/reg1. An 8- or 16-bit operation may be specified. Either mem/reg1 or mem/reg2 may be a memory operand, but one of the operand must be a register operand.

***DIV mem/reg**

Divide AH:AL or DX:AX registers by register or memory location

Divided the AH:AL (8-bit operation) or DX:AX (16-bit operation) register by the contents of the specified 8- or 16-bit register or memory location, considering both operands as unsigned binary numbers.

*** IDIV mem/reg**

Divided AH:AL or DX:AX by register or memory location

*** IMUL mem/reg**

Multiply AL or AX register by register or memory location

Multiply the specified register or memory location contents by contents by the AL (8-bit operation) or AX (16-bit operation).

*** MUL mem/reg**

Multiply AL or AX register by register or memory location

Multiply the specified register or memory location contents by the AL (8-bit operation) or AX (16-bit operation) register, considering both operands as unsigned number, i.e., a simple binary multiplication. If an 8-bit operation is performed, the low- order eight bits of the result are stored in the AL register, the high-order eight bits of the result are stored in the AH register. If a 16-bit operation is performed, the low-order 16 bits of the result is stored in the AX register, the high-order 16 bits of the result are stored in the DX register.

*** SBB ac,data**

Subtract immediate from AX or AL register with borrow.

Subtract the immediate data in the succeeding program memory byte (s) from the AL (8-bit operation) or AX (16-bit operation) register with borrow.

***SUB ac,data**

Subtract immediate data from the AL or AX register

This instruction is used to subtract immediate data from the AL (8-bit operation) register.

2- Logical Instructions:

These instructions are used for logical operations on the operands.

***AND ac, data**

AND immediate data with the AL or AX register

This instruction is used to AND immediate data present in the succeeding program memory byte(s) with the (8-bit operation) or AX (16-bit operation) register contents.

***AND mem/reg , data**

AND immediate data with register or memory location.

***NEG mem/reg**

Negate the contents of register or memory location

This instruction performs a twos complement subtraction of the specified operand from zero. The result is stored in the specified operand. An 8- or 16-bit operand may be specified.

***NOT mem/reg**

Ones complement of register or memory location

Complement the contents of the specified register or memory location.

***OR ac,data**

OR immediate data with the AX or AL register

OR the immediate data in the succeeding program memory byte(s) with AL (8-bit operation) or AX (16-bit operation) register.

***TEST ac,data**

Test immediate data with AX or AL register

AND the immediate data in the succeeding program memory byte(s) with the contents of the AL (8-bit operation) or AX (16-bit operation) register, but do not return the result to the register.

***XOR**

XOR immediate data with AX or AL register

This instruction exclusive-ORs 8- or 16-bit data elements with AL(8-bit) or AX(16-bit) register via immediate addressing.

3- Movement Instructions:

***MOV mem/reg1,mem/reg2**

Move data from register to register or memory to register or register to memory.

This instruction is used to move 8- or 16-bit data elements between a register and a register or memory location.

***MOVS (MOVSB) (MOVSW)**

Move byte or word from memory to memory

Move 8 or 16 bits from the memory location pointed to by the SI register to the memory location pointed to by the DI register. The SI and DI register are incremented / decremented depending on the value of the DF flag.

4- Loading Instructions:

***LODS (LODSB)(LODSW)**

Load from memory into AL or AX register

Move from the memory location addressed by the SI register to the AL (8-bit operation) or the AX (16-bit operation) register. The SI register is incremented / decremented depending on the value of the DF flag.

***LDS reg,mem**

Load register and DS from memory

Load the contents of the specified memory word into the specified register. Load the contents of the memory word following the specified memory word into the DS register.

5- Jumping Instructions:

***JCXZ disp** jump if CX=0
***JE disp**jump if equal
***JZ disp** jump if zero
***JG disp** jump if greater
***JNLE disp** jump if not less nor equal
***JGE disp** jump if greater than or equal
***JNL disp** jump if not less
***JL disp** jump if less
***JNGE disp** jump if not greater than or equal
***JLE disp** jump if less than or equal
***JNG disp** jump if not greater
***JMP addr** jump to the instruction identified in the operand
***JNE disp** jump if not equal
***JNZ disp** jump if not zero
***JNO disp** jump if not overflow
***JNP disp** jump if not parity
***JPO disp** jump if parity odd
***JNS disp** jump if not sign
***JO disp** jump if overflow

***JP disp** jump if parity even
***JPE disp** jump if parity even
***JS disp** jump if sign status is one

6- Looping Instructions:

***LOOP disp**

Decrement CX register and jump if not zero

This instruction decrements the CX register (not affecting the flsgs) and then functions in the same manner as the JMP disp instruction, except that if the CX register has not been decremented to 0, then the jump is executed; otherwise the next instruction is executed.

***LOOPZ disp**

LOOPE disp

Decrement CX register and jump if CX=0 and ZF=1

This instruction decrements the CX register (not affecting the flags) and then functions in the same manner as the JMP disp instruction, except that if the CX register has not been decremented to 0 and the zero flag is 1 then the jump is executed; otherwise the next instruction is executed.

***LOOPNZ disp**

LOOPNE disp

Decrement CX register and jump if CX!=0 and ZF=0

This instruction decrements the CX register (not affecting the flag) and then functions in the same manner as the JMP disp instruction, except that if the CX register has not been decremented to 0 and the zero flag is 0, then the jump is executed; otherwise the next instruction is executed.

7- Stack Instructions:

***POP reg**

Read from the top of the stack

Pop the two top stack bytes into the designated 16-bit register.

***POPF**

Read from the top of the stack into flags register.

***PUSH reg**

Write to the top of the stack

This instruction pushes the contents of the specified 16-bit register into the top of stack.

***PUSHF**

Write the flags register to the top of stack.

8- Count Instructions:

***DEC mem/reg**

Decrement register or memory location

Subtract 1 from the contents of the specified register or memory location. An 8- or 16-bit operation may be specified.

***INC mem/reg**

Increment register or memory location

Add 1 to the contents of the specified register or memory location. An 8- or 16-bit operation may be specified.

9-Compare Instructions:

***CMP ac,data**

Compare immediate data with accumulator

This instruction is used to compare immediate data present in the succeeding program memory byte(s) with the AL register (8-bit operation) or the AX register (16-bit operation). The comparison is performed by subtracting the data in the immediate byte(s) from the specified register, thus no registers are affected, only the statuses.

***CMP mem/reg,data**

Compare immediate data with register or memory

***CMP mem/reg1,mem/reg2**

***CMPS (CMPSB)(CMPSW)**

Compare memory with memory

Compare the contents of the memory location addressed by the SI register with the contents of memory location addressed by the DI register. The comparison is performed by subtracting the contents of memory location addressed by the DI register from the contents the memory location addressed by SI register and using the result to set the flags.

10-Flag Instructions:

***CLC** Clear the carry status

***CLD** Clear the direction flag

***CLI** Clear the interrupt flag

***CMC** Complement the carry status

***STC** Set the carry flag

***STD** Set the direction flag

***STI** Set the interrupt flag