System Programming Notes

There are two main types of software: systems software and application software. Systems software includes the programs that are dedicated to managing the computer itself, such as the operating system, file management utilities, and disk operating system (or DOS).

System software is a software that provides platform to other softwares. Some examples can be operating systems, antivirus softwares, disk formatting softwares, Computer language translators etc.

The most important features of system software include:
1. Closeness to the system
2. Fast speed
3. Difficult to manipulate
4. Written in low level language
5. Difficult to design

Operating System

An operating system (OS) is a type of system software that manages computer’s hardware and software resources. It provides common services for computer programs. An OS acts a link between the software and the hardware.

The most important tasks performed by the operating system
1. **Memory Management:** The OS keeps track of the primary memory and allocates the memory when a process requests it.
2. **Process Management:** Allocates the main memory (RAM) to a process and de-allocates it when it is no longer required.
3. **File Management:** Allocates and de-allocates the resources and decides who gets the resources.

4. **File Management:** Prevents unauthorized access to programs and data by means of passwords.
5. **Error-detecting Aids:** Production of dumps, traces, error messages, and other debugging and error-detecting methods.
6. **Scheduling:** The OS schedules process through its scheduling algorithms.

Compiler: A compiler is a software that translates the code written in one language to some other language without changing the meaning of the program. The compiler is also said to make the target code efficient and optimized in terms of time and space.

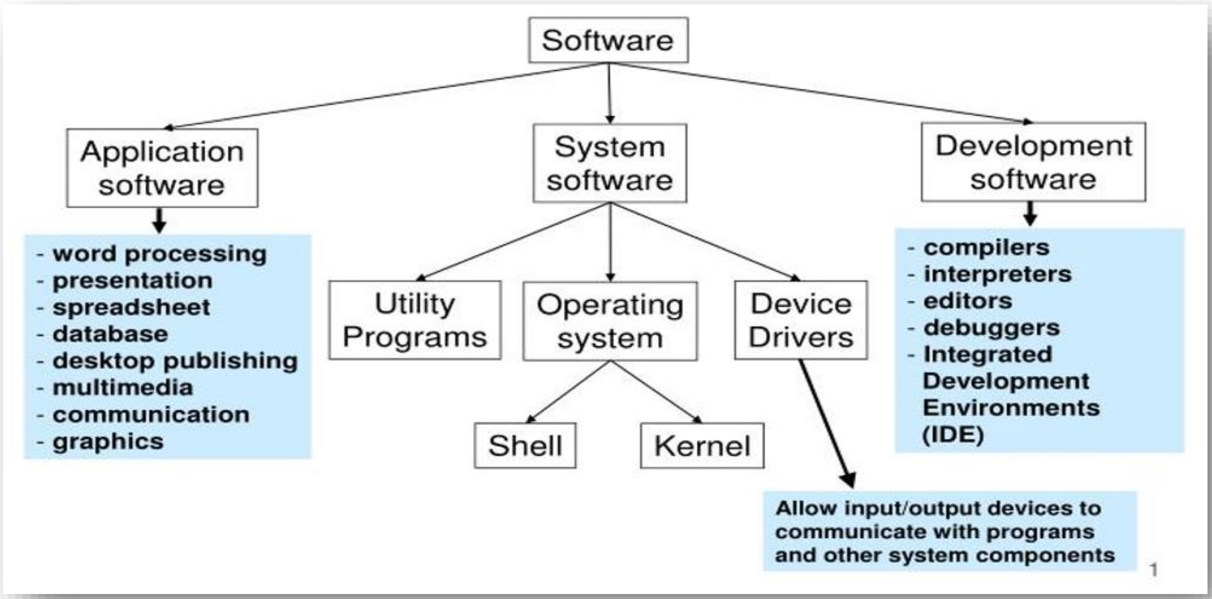
Interpreter: An interpreter is a computer program that directly executes, i.e., it performs instructions written in a programming or scripting language. Interpreter do not require the program to be previously compiled into a machine language program. An interpreter translates high-level instructions into an intermediate form.

Assembler: An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations and converts them into binary code specific to a type of processor.

What is utility software?
These Software analyse and maintain a computer. These Software are focused on how OS works on that basis it perform task to enable smooth functioning of computer. These Software may come along with OS like windows defender, windows disk clean-up tool.
Antivirus, backup software, file manager, disk compression tool all are utility software.

Device Drivers: A device driver is software that helps a peripheral device establish communication with a computer. Ex, Windows Device Manager.

Software Hierarchy



Application Software

Application software is all the computer software that causes a computer to perform useful tasks beyond the running of the computer itself. A specific instance of such software is called a software application, application program, application or app.

Applications software comprises programs designed for an end user.

Examples

accounting software enterprise

software, graphics software,

media players,

word processors, database systems, and spreadsheet programs.

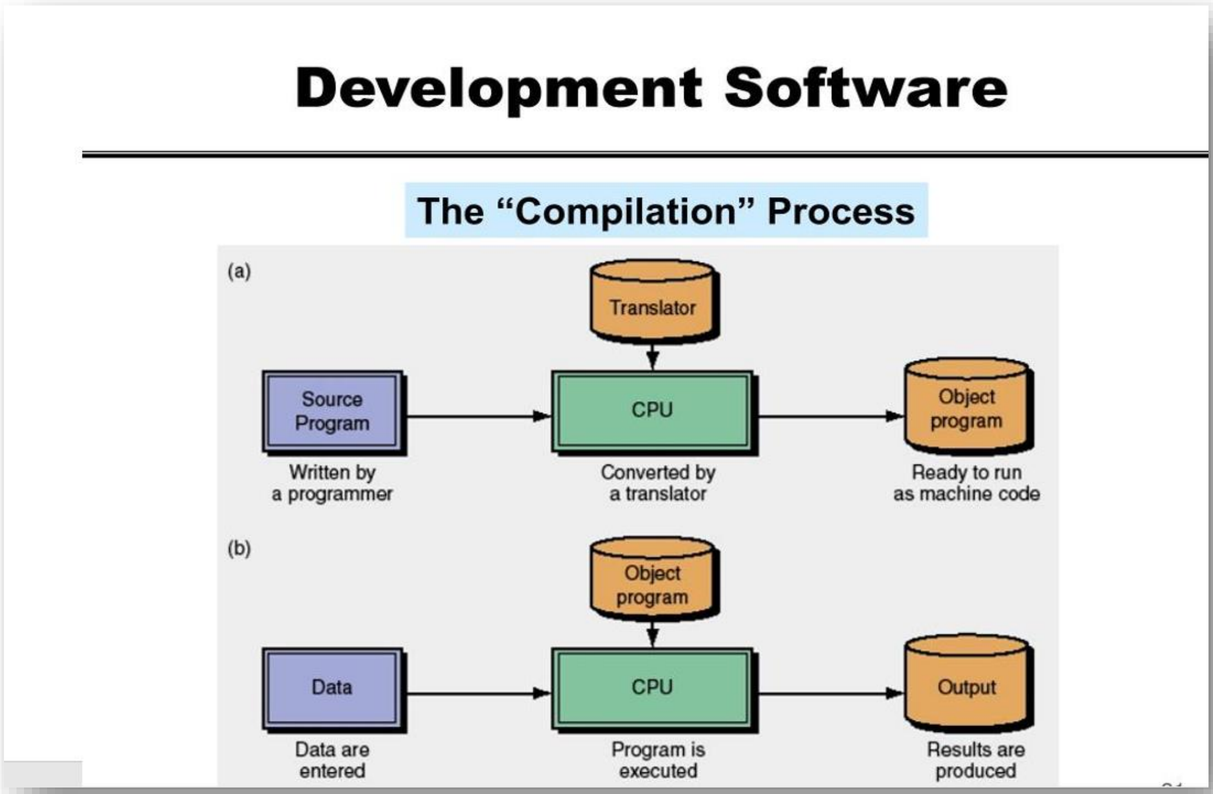
Figuratively speaking, applications software sits on top of systems software because it is unable to run without the operating system and systems utilities.

Types of Applications Software

Application software can also be seen as being either horizontal or vertical.

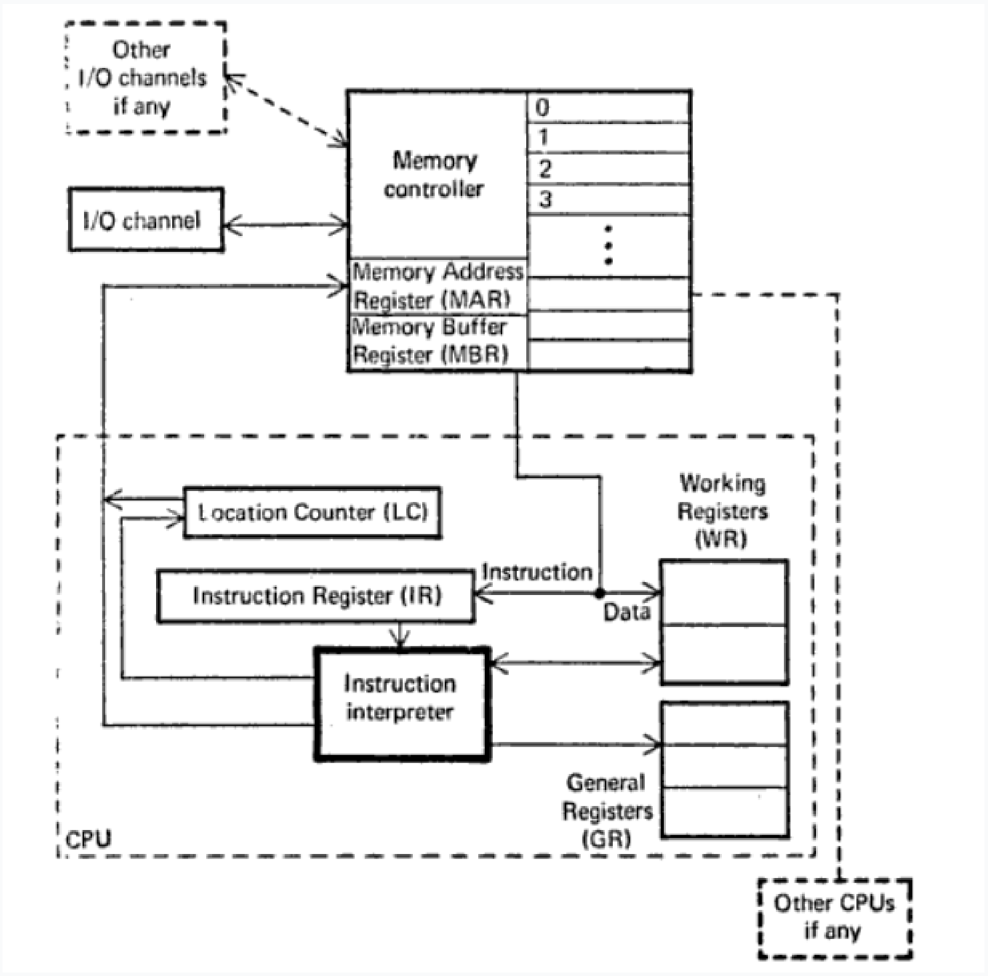
- Horizontal applications are more popular and widespread, because they are general purpose, for example word processors or databases.

Vertical applications are niche products, designed for a particular type of industry or business, or department within an organization. Integrated suites of software will try to handle every specific aspect possible of, for example, manufacturing or banking systems, or accounting, or customer service.



Machine Structure

The structure of CPU for a typical Von Neumann Machine is as follows -



The structure above consists of -

1. Instruction Interpreter
2. Location Counter
3. Instruction Register
4. Working Registers
5. General Register

The Instruction Interpreter Hardware is basically a group of circuits that perform the operation specified by the instructions fetched from the memory.

The Location Counter can also be called as Program/Instruction

Counter simply points to the current instruction being executed.

The working registers are often called as the "scratch pads" because they are used to store temporary values while calculation is in progress.

This CPU interfaces with Memory through MAR & MBR

MAR (Memory Address Register) - contains address of memory location (to be read from or stored into)
MBR (Memory Buffer Register) - contains copy of address specified by MAR

Memory controller is used to transfer data between MBR & the memory location specified by MAR

The role of I/O Channels is to input or output information from memory.

Interface:

An **interface** may refer to any of the following:

1. When referring to software, an **interface** is a program that allows a user to interact computers in person or over a network. An interface may also refer to controls used in a program that allow the user to interact with the program. One of the best examples of an interface is a GUI (Graphical User Interface). This type of interface is what you are using now to navigate your computer and how you got to this page.

2. When referring to hardware, an **interface** is a physical device, port, or connection that interacts with the computer or other hardware device. For example, IDE and SATA are disk drive interfaces for computer hard drives and ATAPI is an early interface for CD-ROM drives.

Address Space:

A computer's address space is the total amount of memory that can be addressed by the computer. The term may refer to the physical memory (RAM chips) or virtual memory (disk/SSD). For example, a 32-bit computer can address 4GB of physical memory and as much as 64TB of virtual memory.

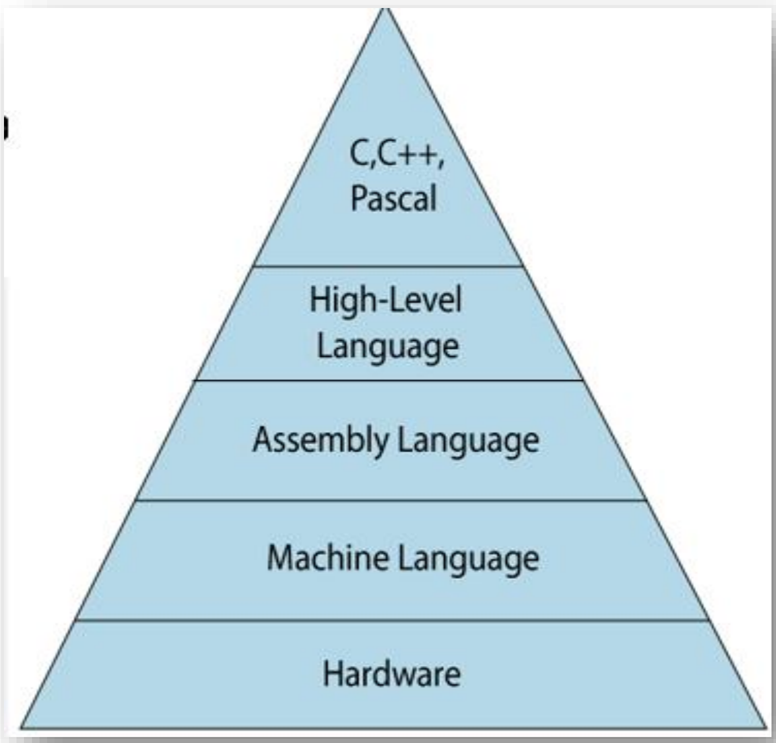
On a computer, each process and device is allocated an address space, which holds a certain portion of the processor's address space. The processor's address space is typically restricted to the width of its registers and address bus.

Address space is often classified as:

- **flat**: In this the addresses are represented as incrementally increasing integers that start at zero
- **Segmented**: In this the addresses are portrayed as independent segments augmented by offsets.
- The size of an address space can be made larger than that of physical memory by using a memory management technique called **virtual memory.**
- So, an address space consists of both physical memory and virtual memory.

Computer languages

- The computer language is defined as code or syntax which is used to write programs or any specific applications. It is used to communicate with computers.
- Computer languages can be broadly classified into 3 major categories:
- assembly language
- machine language
- High-level language.



1. Machine Language

- The machine language is sometimes referred to as machine code or object code which is a set of binary digits 0 and 1. These binary digits are understood and read by a computer system and interpreted easily. It is considered a native language as it can be directly understood by a central processing unit (CPU).

- Example of machine language for the text “Hello World”:-
- 01001000 0110101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100

2. **Assembly language**

- The assembly language is considered a low-level language for microprocessors and many other programmable devices. The assembly language is also considered a second-generation language. The assembly language is mostly famous for writing an operating system and also in writing different desktop applications.
- This language has certain drawbacks as

 - It does not contain any variables or functions in programs and also

 the program is not portable on different processors.

For Ex. ADD A,B -To add the contents of register A & B.

3. **High-level language**

- The high-level language is easy to understand and the code can be written easily as the programs written are user-friendly in a high-level language. The high-level of language uses the concept of abstraction and also focuses on programming language rather than focusing on computer hardware components.

- Examples of high-level languages are C++, C, JAVA, FORTRAN, Pascal, Perl, Ruby, and Visual Basic.

Advantages:

- The syntax used and the programming style can be easily understood by humans if it is compared to low-level language.
- The only requirement in a high-level language is the need for a compiler. As the program written in a high-level language is not directly understood by the computer system.

Computer Languages In-Demand

- The list of computer languages is never-ending. From time to time the demand for computer languages also fluctuates. Some languages were in demand years ago but as technology changes the demand also changes. Here is the list of computer languages in demand all around the world:
- Ruby
- JavaScript
- Python
- PHP
- Java
- C#
- Objective-C & Swift

Tools

There are different source code management tools available for use and a lot of them are also open-source and free to use. Depending on the team's requirements, an appropriate tool can be used.

- Tools integrating project management and build pipeline features are GitLab and Team Foundation Server.
- For on-premise setup, teams can use Git, Subversion, CVS, etc.
- For example, if Git is the chosen tool, you can create all your source code and just type git init.
- This command will initialize all the settings and files required for Git to function properly. Once you commit/staging your code, it will be available as a Git repository for others to download/contribute.

Life Cycle of Source Program

The life cycle of a source program defines the program behavior and extends through execution stage, which exhibits the behavior specified in the program.

Every source program goes through a life cycle of several stages.

Edit time: It is the phase where editing of the program code takes place and is also known as design time. At this stage, the code is in its raw form and may not be in a consistent state.

Compile time: At the compile time stage, the source code after editing is passed to a translator that translates it into machine code. One such translator is a compiler. This stage checks the program for inconsistencies and errors and produces an executable file.

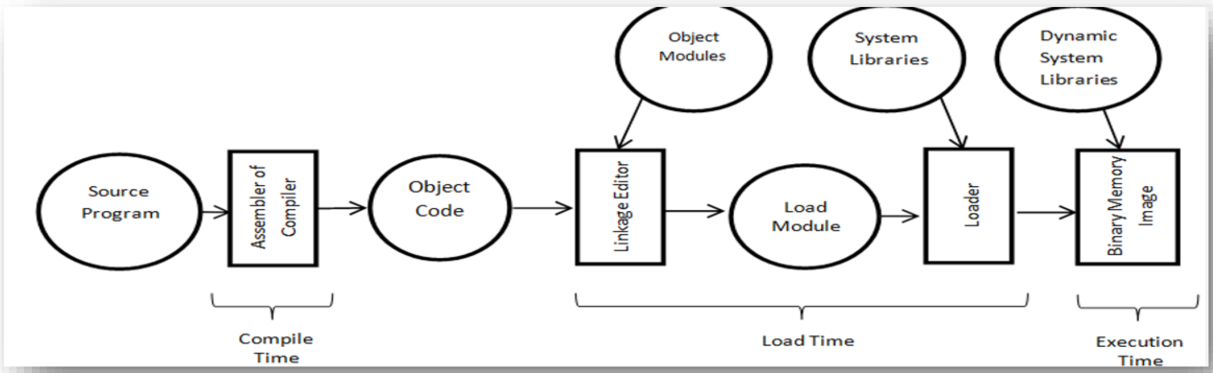
Distribution time: It is the stage that sends or distributes the program from the entity creating it to an entity invoking it. Mostly executable files are distributed.

Installation time: Typically, a program goes through the installation process, which makes it ready for execution within the system. The installation can also optionally generate calls to other stages of a program's life cycle.

Link time: At this stage, the specific implementation of the interface is linked and associated to the program invoking it. System libraries are linked by using the lookup of the name and the interface of the library needed during compile time or throughout the installation time, or invoked with the start or even during the execution process.

Load time: This stage actively takes the executable image from its stored repositories and places them into active memory to initiate the execution. Load time activities are influenced by the underlying operating system.

Run time: This is the final stage of the life cycle in which the programmed behavior of the source program is demonstrated.



Different views on the meaning of program

- In computing, a program is a specific set of ordered operations for a computer to perform. In the modern computer that John von Neumann outlined in 1945, the program contains a one-at-a-time sequence of instructions that the computer follows.

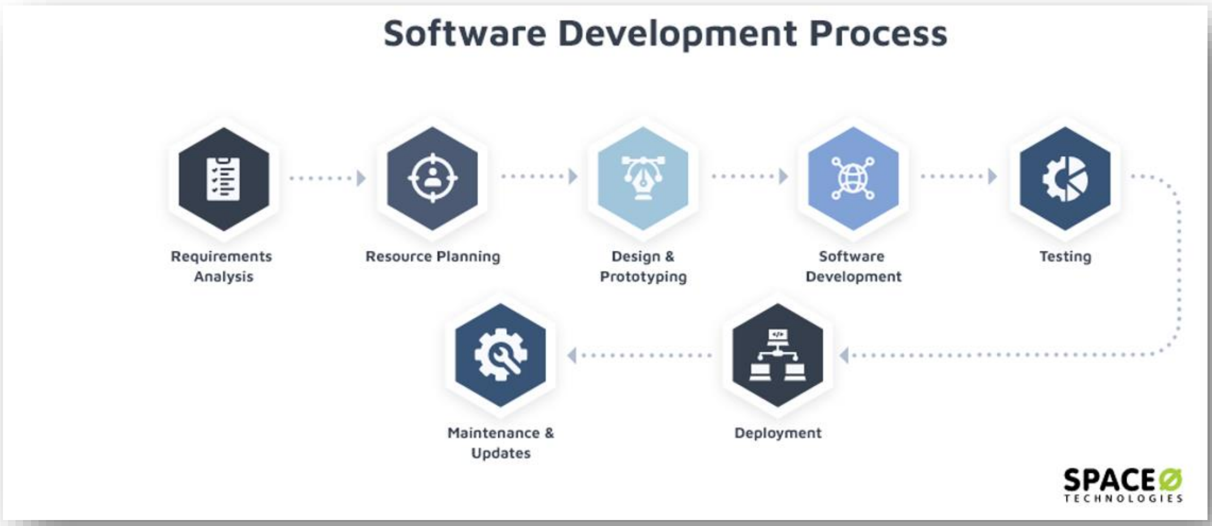
- A program is also a special kind of data that indicates how to operate on application or user data. Computer programs can be characterized as interactive or batch in terms of what drives them and how continuously they run.
- Computer program, detailed plan or procedure for solving a problem with a computer; more specifically, an unambiguous, ordered sequence of computational instructions necessary to achieve such a solution.
- Computer program, detailed plan or procedure for solving a problem with a computer; more specifically, an unambiguous, ordered sequence of computational instructions necessary to achieve such a solution.

System Software Development

- A software development process is the process of dividing software development work into distinct phases to improve design, product management, and project management. It is also known as a software development life cycle. Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

OR

- software development process is dividing the software development into tiny, sequential steps to enhance the product, project, and design altogether. the iterative logical process for software program development or application development to cater to the needs of any business or personal objectives is known as 'Software Development'.



- Requirements Analysis and Resource Planning
- Design and Prototyping
- Software Development
- Testing
- Deployment
- Maintenance and Updates

- **Requirements Analysis and Resource Planning**

The first step to any process is always planning. Being a project manager, you might have done a requirement analysis of your project, but you are going to need software engineering experts to create plan. You need to analyze if the software, you are planning to develop, aligns with your business or personal goals. This is a requirements analysis.

- **Design and Prototyping**

After the analysis and planning part is over, it is time to start creating a software architecture for the product. This architecture or design will define the complete workflow of the software.

- **Software Development**

Development in software-process only begins when you are completely sure of the requirements and onboard with the design and features. The development team starts working on the development of a program by writing the necessary code.

- **Testing**

This is actually a continuous **process of software development**, and testing is performed alongside development. Testing is done to check the functionality, usability, and stability of the product under the rapid development process.

- **Deployment**

This is a crucial stage in the software development life cycle. After coding and testing are done, the next development phase is to deploy your software on the necessary servers and devices. This is only done after you have approved of the product functionality and the stability of the product is proved.

- **Maintenance and Updates**

As described earlier, software development is a cycle. It is an iterative process of software development. After launching the product, the process is not complete. You need to keep a track of software maintenance and keep upgrading it. You need to consistently monitor software development and suggest changes whenever required

Recent Trends in Software Development

- In the recent past, IoT, data management, and digitalization of services are some aspects that software has penetrated. It has become difficult for developers and businesses to keep up with these trends. It has become crucial to understand the upcoming trends that will help you modify your business model accordingly.

The latest software development trends in 2022.

- Progressive Web Apps

A Progressive Web Application (PWA) can be accessible offline by using the previously cached data of your interactions with the app. This web application provides a seamless user experience to native mobile app and web application users.

- DevOps

DevOps has become one of the best practices in the software landscape. It is the combination of operations and development that removes the barrier between the operations and the development process. The main task of DevOps is to unify the entire software application lifecycle, including development, testing, deployment, and operations.

- **Cyber Security**
- In the current era, it has become crucial more than ever to build a safe and secure digital environment. Every business has digital assets in this modern age, so implementing the latest cyber security measures is necessary.
- Cyber security has become one of the major software development trends to help companies strengthen their online security against potential attacks.
- **Internet of Behavior (IOB)**
- IOB is a data collection process based on the behavior and interests of internet users. The top companies like Facebook and Google have incorporated IOB-based data in their services

to personalize search engine results and advertising targeting. It has helped businesses to create a personalized experience and channel their efforts in the right direction.

- **Low-code Development**

Low-code development allows companies to create apps with minimum effort as minimal coding is required. Drag-and-drop website builders have already reformed the web development landscape, which will also impact software development.

- **Python**

Recently, Python has become one of the most popular and fast-growing programming languages. Software developers widely use it to create complex web applications to fulfil the needs of modern-day businesses and customers.

- **Cloud-Native Apps**

Cloud-native apps are also expected to become the fast-growing software development trend of 2022. It allows developers to build robust and highly-functional cloud-native apps more efficiently. One of the major benefits of a cloud-native app is that you can build it using many frameworks.

Whether you are a business owner or software developer, embracing the latest software development trends is a must. You need to stay updated on the software industry's current trends so you can implement those practices accordingly.

Levels of System Software

- System software is a set of programs that handles all the basic internal working of a computer.

Levels of System Software are

1. Operating System

An operating system is system software that controls the working of computer hardware and software. Moreover, it acts as a common connection between the computer hardware and software. In other words, we can also call it an interface between the hardware and the users.

Some important tasks performed by the operating system are:

- Scheduling
- Memory Management
- File Management
- Security
- Protects data and other software from unauthorized access .

2. Language Processors

It is a special type of system software that converts the source code into machine code. The input given has to be in object code only hence, we use language processors. Also, the machine code executes faster as compared to the source code.

- Source Code
- Object Code
- Different Types of Language Processors are:

Assembler: It converts assembly language to machine language.

Interpreter: It is a type of system software that executes the program line by line.

Compiler: It is also a type of system software that executes the whole program at once.

3. Utility Software

- Application Software
- System Utilities

These types of system software are used for the proper and smooth functioning of the

computer system. They perform functions like removing outdated files, recover data which

is accidentally lost, finding information, arranging data and files in an orderly manner,

compress disk drive, install and uninstall programs, etc.

- Different types of utility software are:

1. Antivirus Software They are used to protect the system from viruses. Some examples are Quick Heal, McAfee, etc.

2. Compression Tools They help compress large files. The files can be changed to the original form when we require it. Examples are WinRAR, PeaZip, etc.

3. Disk Management Tools

They are used to manage data on the disks efficiently so that the system performance can enhance. Examples are Disk Cleanup Tool, Backup Utility, etc.

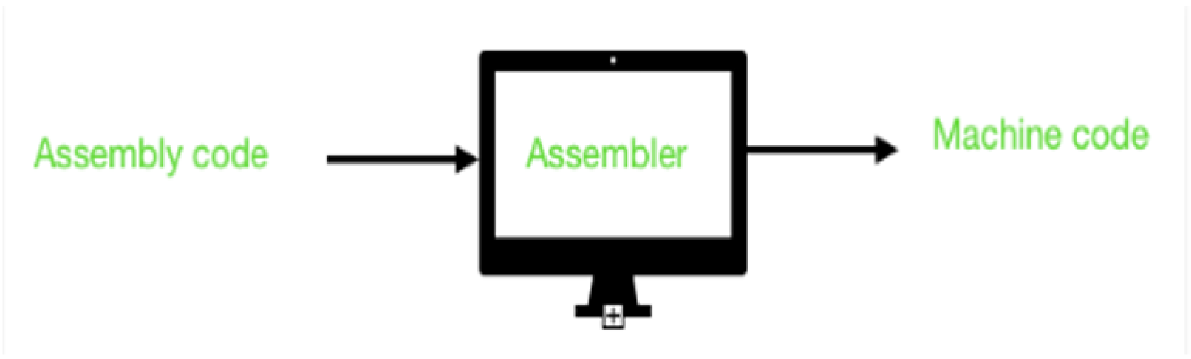
4. Device Drivers

- These types of system software are used for the operation of the peripheral devices. Each device connected to the computer has its own driver. These drivers basically contain instructions that tell the operating system how to operate the device.
- Some drivers are pre-installed on the computer while some others are installed when a new device is added. The audio

device, video device, scanner, camera, etc. all require a driver. A driver tells the operating system how to use the device.

Assembler

- An assembler is a program that converts assembly language into machine code.
- It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor.
- Assemblers are similar to compilers in that they produce executable code.
- However, assemblers are more simplistic since they only convert low-level code (assembly language) to machine code.
- Since each assembly language is designed for a specific processor, assembling a program is performed using a simple one-to-one mapping from assembly code to machine code.
- It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code.
- Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler.



Elements of Assembler

Assembly language is basically like any other language, which means that it has its words, rules and syntax. The basic elements of assembly language are:

- Labels;
- Orders;
- Directives; and
- Comments.

Syntax of Assembly language

When writing a program in assembly language it is necessary to observe specific rules in order to enable the process of compiling into executable "HEX-code" to run without errors. These compulsory rules are called syntax and there are only several of them:

- Every program line may consist of a maximum of 255 characters;
- Every program line to be compiled, must start with a symbol, label, mnemonics or directive;
- Text following the mark ";" in a program line represents a comment ignored (not compiled) by the assembler; and
- All the elements of one program line (labels, instructions etc.) must be separated by at least one space character. For the sake of better clearness, a push button TAB on a keyboard is commonly used instead of it, so that it is easy to delimit columns with labels, directives etc. in a program.

Numbers

If octal number system, otherwise considered as obsolete, is disregarded, assembly language allows numbers to be used in one out of three number systems:

Decimal Numbers

If not stated otherwise, the assembly language considers all the numbers as decimal. All ten digits are used (0,1,2,3,4,5,6,7,8,9). Since at most 2 bytes are used for saving them in the microcontroller, the largest decimal number that can be written in assembly language is 65535. If it is necessary to specify that some of the numbers is in decimal format, then it must be followed by the letter “D”. For example, 1234D.

Hexadecimal Numbers

Hexadecimal numbers are commonly used in programming. There are 16 digits in hexadecimal number system (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). The largest hexadecimal number that can be written in assembly language is FFFF. It corresponds to decimal number 65535. In order to distinguish hexadecimal numbers from decimal, they are followed by the letter “h” (either in upper- or lowercase). For example, 54h.

Binary Numbers

Binary numbers are often used when the value of each individual bit of some of the registers is important, since each binary digit represents one bit. There are only two digits in use (0 and 1). The largest binary number written in assembly language is 1111111111111111. In order to distinguish binary numbers from other numbers, they are followed by the letter “b” (either in upper- or lowercase). For example, 01100101B.

Operators

Some of the assembly-used commands use logical and mathematical expressions instead of symbols having specific values. For example:

Addition, Subtraction, Multiplication, Division, &Bitwise, logical AND, Bitwise logical OR, >>Shift right, <<Shift left, %Remainder, Bitwise logical AND, NOT, Bitwise logical XOR.

Symbols
Every register, constant, address or subroutine can be assigned a specific symbol in assembly language, which considerably facilitates the process of writing a program.

- For the purpose of writing symbols in assembly language, all letters from alphabet (A-Z, a-z), decimal numbers (0-9) and two special characters ("?" and "_") can be used. Assembly language is not case sensitive.
- In order to distinguish symbols from constants (numbers), every symbol starts with a letter or one of two special characters (? or _).
- The symbol may consist of maximum of 255 characters, but only first 32 are considered.

Some of the symbols cannot be used when writing a program in assembly language because they are already part of instructions or assembly directives.

For example, the following symbols will be considered identical:

```
Serial_Port_Buffer
SERIAL_PORT_BUFFER
```

Labels
A label is a special type of symbols used to represent a textual version of an address in ROM or RAM memory. They are always placed at the beginning of a program line. It is very complicated to call a subroutine or execute some of the jump or branch instructions without them. They are easily used:

- A symbol (label) with some easily recognizable name should be written at the beginning of a program line from which a subroutine starts or where jump should be executed.
- It is sufficient to enter the name of label instead of address in the form of 16-bit number in instructions calling a subroutine or jump.

During the process of compiling, the assembler automatically replaces such symbols with appropriate addresses.

Directives

Unlike instructions being compiled and written to chip program memory, directives are commands of assembly language itself and have no influence on the operation of the microcontroller. Some of them are obligatory part of every program while some are used only to facilitate or speed up the operation. Directives are written in the column reserved for instructions. There is a rule allowing only one directive per program line.

Ex. EQU directive and SET directive

- The EQU directive is used to replace a number by a symbol.
- The SET directive is also used to replace a number by a symbol

Design of Assembler

<u>General design procedure</u>

specify the problem

1. specify the data structure
2. define format of data structure
3. specify algorithm
4. look for modularity

5. repeat 1 ~ 5 on modules

Objectives

- Generate instructions Evaluate the mnemonic in the operation field to produce its machine code

1. Find the value of each symbol, process literals

2. Process pseudo-ops

- Assembler divide these tasks in two passes:
- **Pass-1:**
    - Define symbols and literals and remember them in symbol table and literal table respectively.
    - Keep track of location counter
    - Process pseudo-operations
- **Pass-2:**
    - Generate object code by converting symbolic op-code into respective numeric op-code
    - Generate data for literals and look for values of symbols

Pass 1 data bases

Input source program

A LC to keep track of each instruction location

A MOT (Machine Operation Table)

A POT (Pseudo operation Table)

A ST (Symbol Table)

A LT (Literal Table)

A copy of the input to be used by pass 2

Pass 2 databases

Copy of source program input to pass 1

LC

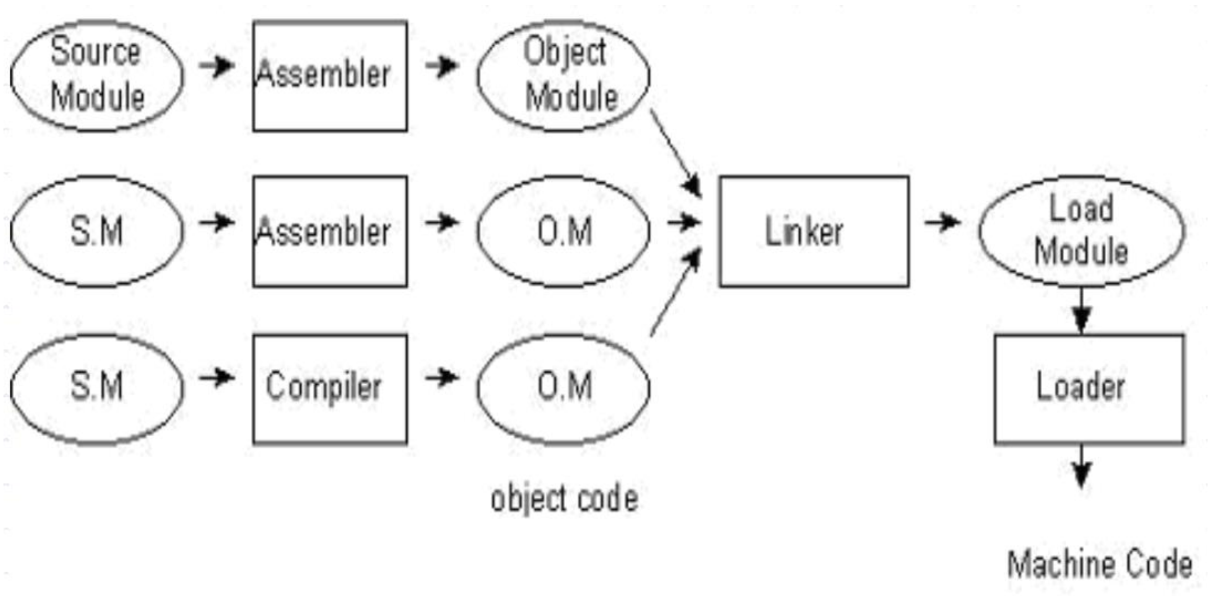
MOT

POT

ST

BT (Base table)

Output in machine code to be needed by the loader



Assembler Design Criteria

Design Specification of an assembler

Four step approach to develop a design specification

1) Identify the information necessary to perform a task

2) Design a suitable data structure to record the information

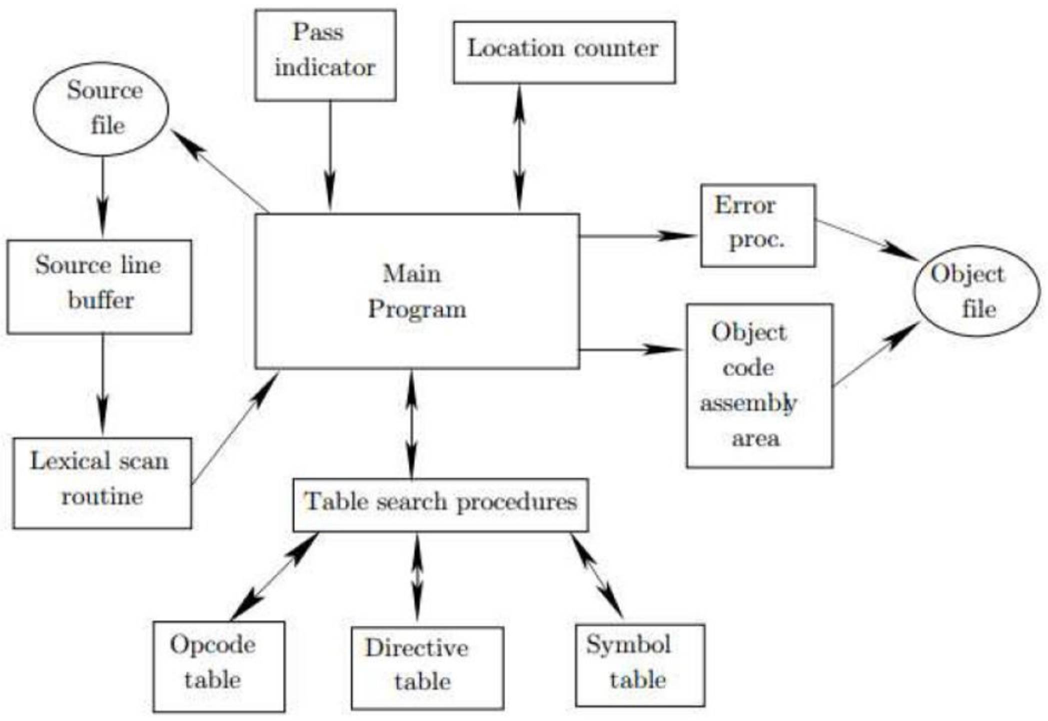
3) Determine the processing necessary to obtain and maintain the information.

4) Determine the processing necessary to perform the task.

Types of Assemblers

the two main types of assemblers namely, the one-pass and the two-pass.

The main components of assembler and main program is as follows:



One-pass assembler

- The operation of a one-pass assembler is different.
- As its name implies, this assembler reads the source file once.
- During that single pass, the assembler handles both label definitions and assembly.

- The only problem is future symbols.

Two-pass assembler

- Such an assembler performs two passes over the source file.
- In the first pass it reads the entire source file, looking only for label definitions.
- All labels are collected, assigned values, and placed in the symbol table in this pass.
- No instructions are assembled and, at the end of the pass, the symbol table should contain all the labels defined in the program.
- In the second pass, the instructions are again read and are assembled, using the symbol table.

An assembly code is given below as follows:

M ADD R1, ='3'

where, M - Label; ADD - symbolic opcode;

R1 - symbolic register operand; (='3') - Literal

Assembly Program:

```
Label Op-code   operand   LC value (Location counter)
JOHN   START     200
      MOVER     R1, ='3'    200
      MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
      LTORG                203
X      DS       1          204
```

END	205

Let's take a look on how this program is working:

1. **START:** This instruction starts the execution of program from location 200 and label with START provides name for the program. (JOHN is name for program)
2. **MOVER:** It moves the content of literal (='3') into register operand R1.
3. **MOVEM:** It moves the content of register into memory operand(X).
4. **MOVER:** It again moves the content of literal (='2') into register operand R2 and its label is specified as L1.
5. **LTORG:** It assigns address to literals (current LC value).
6. **DS (Data Space):** It assigns a data space of 1 to Symbol X.
7. **END:** It finishes the program execution.

Working of pass-1 assemblers

Define Symbol and literal table with their addresses.
Note: Literal address is specified by LTORG or END.

Step-1: START 200 (here no symbol or literal is found so both tables would be empty)
Step-2: MOVER R1, ='3' 200 (='3' is a literal so literal table is made)

Literal	Address
='3'	– – –

Step-3: MOVEM R1, X 201
X is a symbol referred prior to its declaration so it is stored in symbol table with blank address field.

Symbol	Address
X	– – –

Step-4: L1 MOVER R2, ='2' 202
L1 is a label and ='2' is a literal so store them in respective tables.

Symbol	Address
X	– – –
L1	202

Literal	Address
='3'	– – –
='2'	– – –

Step-5: LTORG 203
Assign address to first literal specified by LC value, i.e., 203

Literal	Address
='3'	203
='2'	- - -

Step-6: X DS 1 204
It is a data declaration statement i.e.; X is assigned data space of 1. But X is a symbol which was referred earlier in step 3 and defined in step 6. This condition is called Forward Reference Problem where variable is referred prior to its declaration and can be solved by back-patching. So now assembler will assign X the address specified by LC value of current step.

Symbol	Address
X	204
L1	202

Step-7: END 205
Program finishes execution and remaining literal will get address specified by LC value of END instruction. Here is the complete symbol and literal table made by pass 1 of assembler.

Symbol	Address
X	204
L1	202

Literal	Address
='3'	203
='2'	205

Now tables generated by pass 1 along with their LC value will go to pass-2 of assembler for further processing of pseudo-opcodes and machine op-codes.

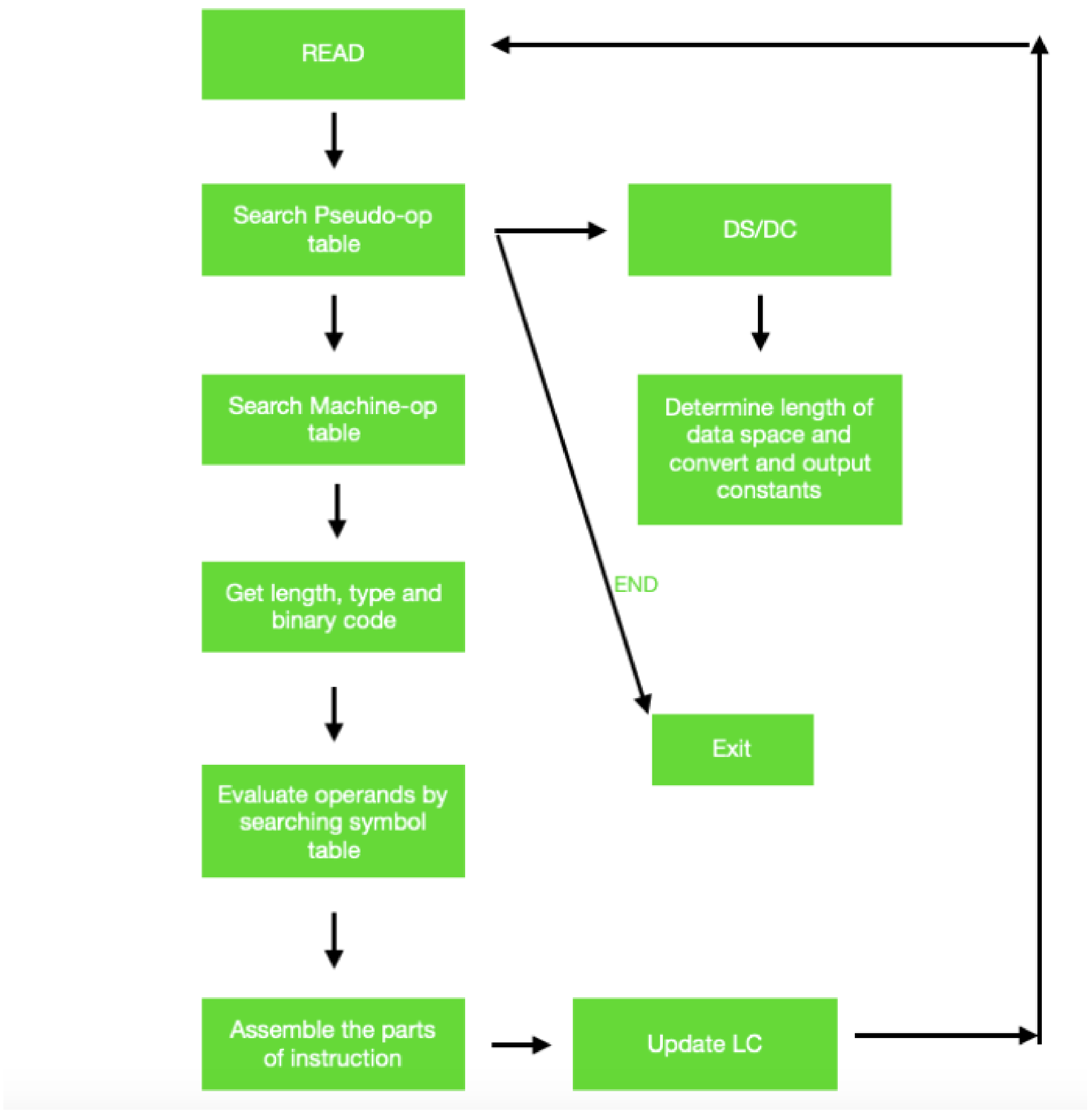
Working of pass-2 assembler

Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration (machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length, and their bit configuration. It will also process pseudo-ops and will store them in POT table (pseudo-op table).

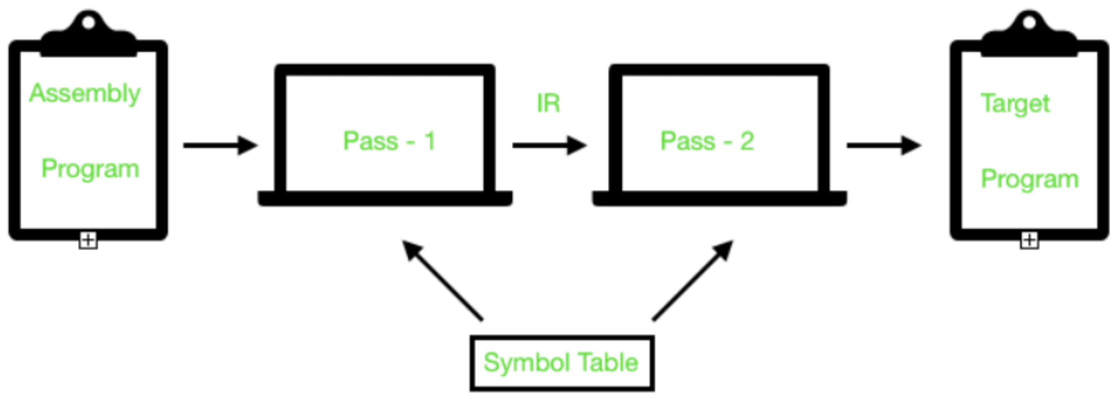
Various Data bases required by pass-2:

1. MOT table (machine opcode table)
2. POT table (pseudo-opcode table)
3. Base table (storing value of base register)
4. LC (location counter)

Look at flowchart to understand:


READ
Search Pseudo-op table
DS/DC
Search Machine-op table
Determine length of data space and convert and output constants
END
Get length, type and binary code
Exit
Evaluate operands by searching symbol table
Assemble the parts of instruction
Update LC

As a whole assembler works as:



Single pass assembler for intel x86

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Member of the Intel IA-32 instruction set

It is divided into Four parts

- Label (optional)
- Mnemonic (required)
- Operand (usually required)
- Comment (optional) Label: Mnemonic Operand(s); Comment

<u>Labels</u>

- Act as place markers
- marks the address (offset) of code and data
- Easier to memorize and more flexible

eg. mov ax, [0020]→mov ax, val

• Follow identifier rules

Data label

• must be unique

example: myArray BYTE 10

• Code label (ends with a colon)

• target of jump and loop instructions

example: L1: mov ax, bx ... jmp L1

Reserved words and identifiers

• Reserved words cannot be used as identifiers

•Instruction mnemonics, directives, type attributes, operators, predefined symbols

• Identifiers

• 1-247 characters, including digits

• case insensitive (by default)

• first character must be a letter, _, @, or $

examples: var1 Count $first _main MAX open_file @@myfile xVal _12345

- **Comments** • Comments are good! • explain the program's purpose • tricky coding techniques • application-specific explanations • Single-line comments • begin with semicolon (;) • block comments • begin with COMMENT directive and a programmer-chosen character and end with the same programmer-chosen character COMMENT ! This is a comment and this line is also a comment !

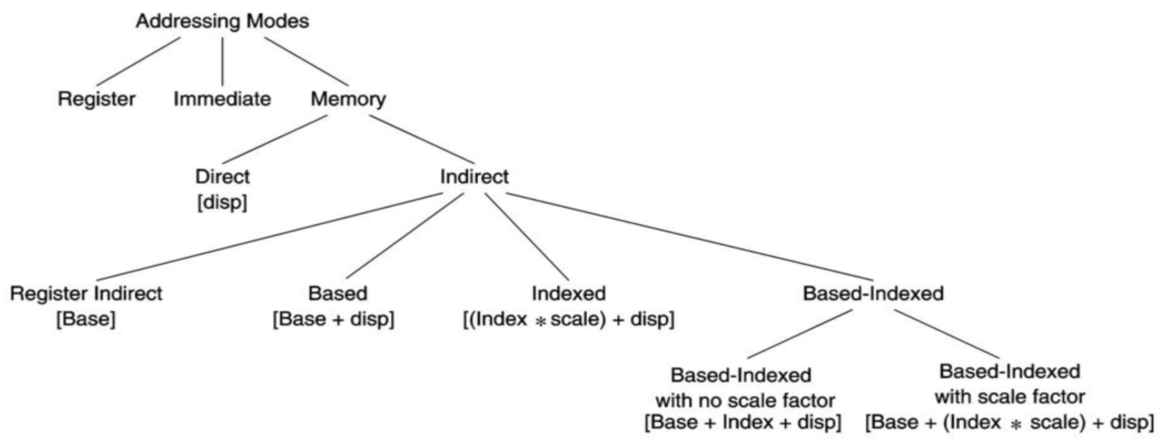
- directive marking a comment comment copy definitions from Irvine32.inc code segment. 3 segments: code, data, stack beginning of a procedure destination source defined in Irvine32.inc to end a program marks the last line and define the startup procedure Example: adding/subtracting integers TITLE Add and Subtract (AddSub.asm) ; This program adds and subtracts 32-bit integers.
- Defining data
- Intrinsic data types (1 of 2) • BYTE, SBYTE • 8-bit unsigned integer; 8-bit signed integer • WORD, SWORD • 16-bit unsigned & signed integer • DWORD, SDWORD • 32-bit unsigned & signed integer • QWORD • 64-bit integer • TBYTE • 80-bit integer
- Intrinsic data types (2 of 2) • REAL4 • 4-byte IEEE short real • REAL8 • 8-byte IEEE long real • REAL10 • 10-byte IEEE extended real
- Data definition statement • A data definition statement sets aside storage in memory for a variable. • May optionally assign a name (label) to the data. • Only size matters, other attributes such as signed are just reminders for programmers. • Syntax: [name] directive initializer [,initializer] . . . At least one initializer is required, can be ? • All initializers become binary data in memory
- Integer constants • [{+|-}] digits [radix] • Optional leading + or – sign • binary, decimal, hexadecimal, or octal digits • Common radix characters: • h– hexadecimal • d– decimal (default) • b– binary • r– encoded real • o– octal Examples: 30d, 6Ah, 42, 42o, 1101b Hexadecimal beginning with letter: 0A5h

- Integer expressions • Operators and precedence levels: • Examples:
- Real number constants (encoded reals) • Fixed point v.s. floating point • Example 3F800000r=+1.0,37.75=42170000r • double 1 8 23 S E M ±1.bbbb×2 (E-127) 1 11 52 S E M
- Real number constants (decimal reals) • [sign]integer.[integer][exponent] sign → {+|-} exponent → E[{+|-}]integer • Examples: 2. +3.0 -44.2E+05 26.E5
- Character and string constants • Enclose character in single or double quotes • 'A', "x" • ASCII character = 1 byte • Enclose strings in single or double quotes • "ABC" • 'xyz' • Each character occupies a single byte • Embedded quotes: • 'Say "Goodnight," Gracie' • "This isn't a test"
- Defining BYTE and SBYTE Data Each of the following defines a single byte of storage: value1 BYTE 'A' ; character constant value2 BYTE 0 ; smallest unsigned byte value3 BYTE 255 ; largest unsigned byte value4 SBYTE -128 ; smallest signed byte value5 SBYTE +127 ; largest signed byte value6 BYTE ? ; uninitialized byte A variable name is a data label that implies an offset (an address).
- Defining multiple bytes Examples that use multiple initializers: list1 BYTE 10,20,30,40 list2 BYTE 10,20,30,40 BYTE 50,60,70,80 BYTE 81,82,83,84 list3 BYTE ?,32,41h,00100010b list4 BYTE 0Ah,20h,'A',22h
- Defining strings (1 of 2) • A string is implemented as an array of characters • For convenience, it is usually enclosed in quotation marks • It usually has a null byte at the end • Examples: str1 BYTE "Enter your name",0 str2 BYTE 'Error: halting program',0 str3 BYTE 'A','E','I','O','U' greeting1 BYTE "Welcome to the Encryption Demo program " BYTE

"created by Kip Irvine.",0 greeting2 \ BYTE "Welcome to the Encryption Demo program " BYTE "created by Kip Irvine.",0

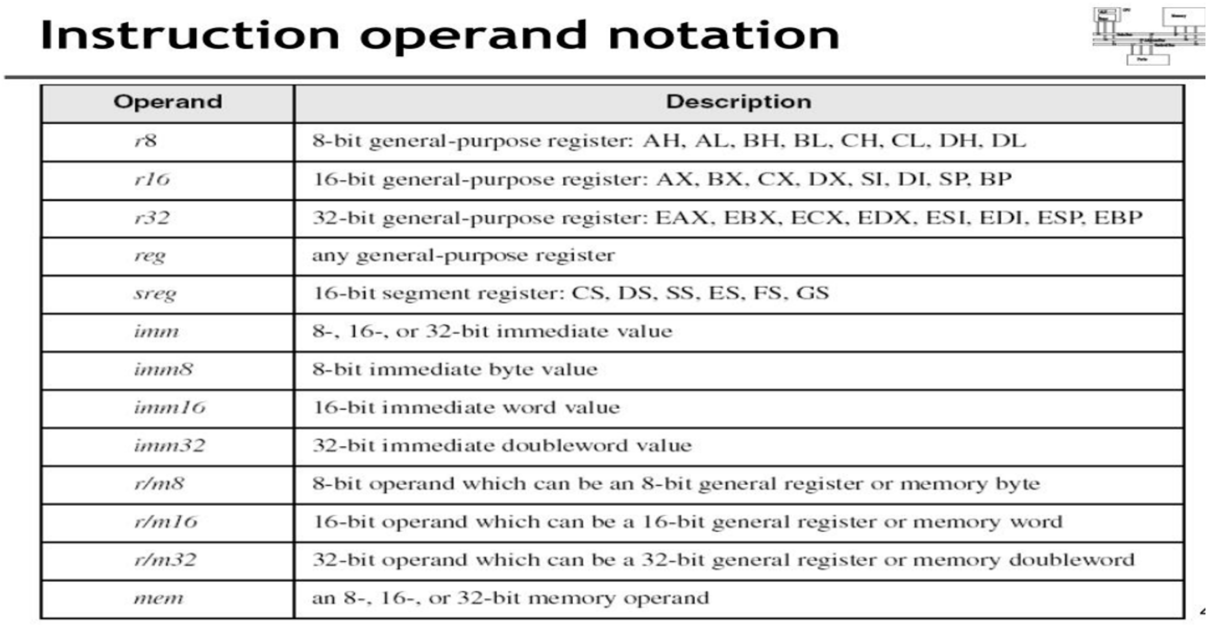
- Defining strings (2 of 2) • End-of-line character sequence: • 0Dh = carriage return • 0Ah = line feed str1 BYTE "Enter your name: ",0Dh,0Ah BYTE "Enter your address: ",0 newLine BYTE 0Dh,0Ah,0 Idea: Define all strings used by your program in the same area of the data segment.

Addressing Modes


Addressing Modes
Register
Immediate
Memory
Direct [disp]
Indirect
Register Indirect [Base]
Based [Base + disp]
Indexed [(Index ∗ scale) + disp]
Based-Indexed
Based-Indexed with no scale factor [Base + Index + disp]
Based-Indexed with scale factor [Base + (Index ∗ scale) + disp]

Operand types

- Three basic types of operands:
  - Immediate – a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - Register – the name of a register
    - register name is converted to a number and encoded within the instruction
  - Memory – reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location



Instruction operand notation

Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

Single pass assembler algorithm

```
begin
 if starting address is given
LOCCTR = starting address;
else
LOCCTR = 0;
while OPCODE != END do ;; or EOF
begin
read a line from the code
if there is a label
 if this label is in SYMTAB, then error
else insert (label, LOCCTR) into SYMTAB
search OPTAB for the op code
if found
LOCCTR += N ;; N is the length of this instruction (4 for MIPS)
else if
```

```
this is an assembly directive
update LOCCTR as directed
else error
write line to intermediate file

end

program size = LOCCTR - starting address;
end
```

Multi-pass assemblers

If we use a two-pass assembler, the following symbol definition cannot be allowed.

```
ALPHA EQU BETA

BETA EQU DELTA

DELTA RESW 1
```

This is because ALPHA and BETA cannot be defined in pass 1. If we allow multi-pass processing, DELTA is defined in pass 1, BETA is defined in pass 2, and ALPHA is defined in pass 3, and the above definitions can be allowed.

This is the motivation for using a multi-pass assembler.

- It is unnecessary for a multi-pass assembler to make more than two passes over the entire program.
- Instead, only the parts of the program involving forward references need to be processed in multiple passes.
- The method presented here can be used to process any kind of forward references.

- Use a symbol table to store symbols that are not totally defined yet.
- For a undefined symbol, in its entry, – We store the names and the number of undefined symbols which contribute to the calculation of its value. – We also keep a list of symbols whose values depend on the defined value of this symbol.
- When a symbol becomes defined, we use its value to reevaluate the values of all the symbols that are kept in this list.
- The above step is performed recursively.
- Examples

Microsoft MASM Assembler, Sun Sparc Assembler, IBM AIX Assembler

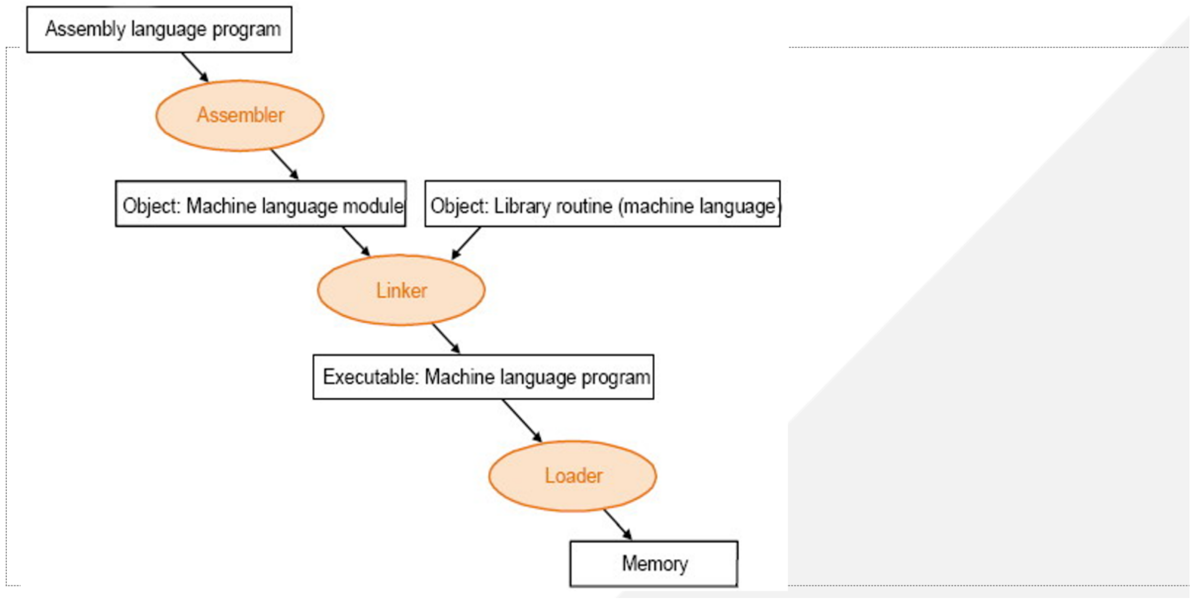
- Microsoft MASM Assembler
- SEGMENT - a collection segments, each segment is defined as belonging to a particular class, CODE, DATA, CONST, STACK
- registers: CS (code), SS (stack), DS (data), ES, FS, GS
- similar to program blocks in SIC l ASSUME

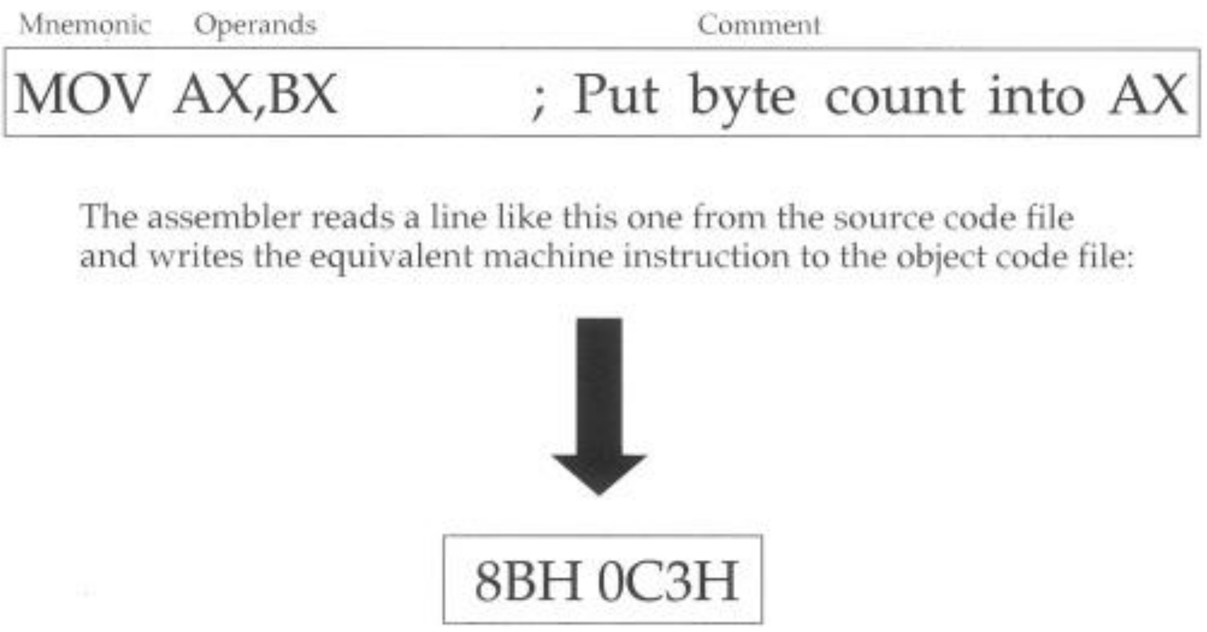
e. g. MOVE ES: DATASEG 2 AX, DATASEG 2 ES, AX » similar to BASE in SIC 11

- JUMP with forward reference
- near jump: 2 or 3 bytes
- far jump: 5 bytes
- e. g. JMP TARGET
- Warning: JMP FAR PTR TARGET
- Warning: JMP SHORT TARGET

- Pass 1: reserves 3 bytes for jump instruction phase error PUBLIC, EXTRN
- similar to EXTDEF, EXTREF in SIC 12

Advanced assembly process


Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory



MOV AX,BX ; Put byte count into AX

The assembler reads a line like this one from the source code file and writes the equivalent machine instruction to the object code file:

8BH 0C3H

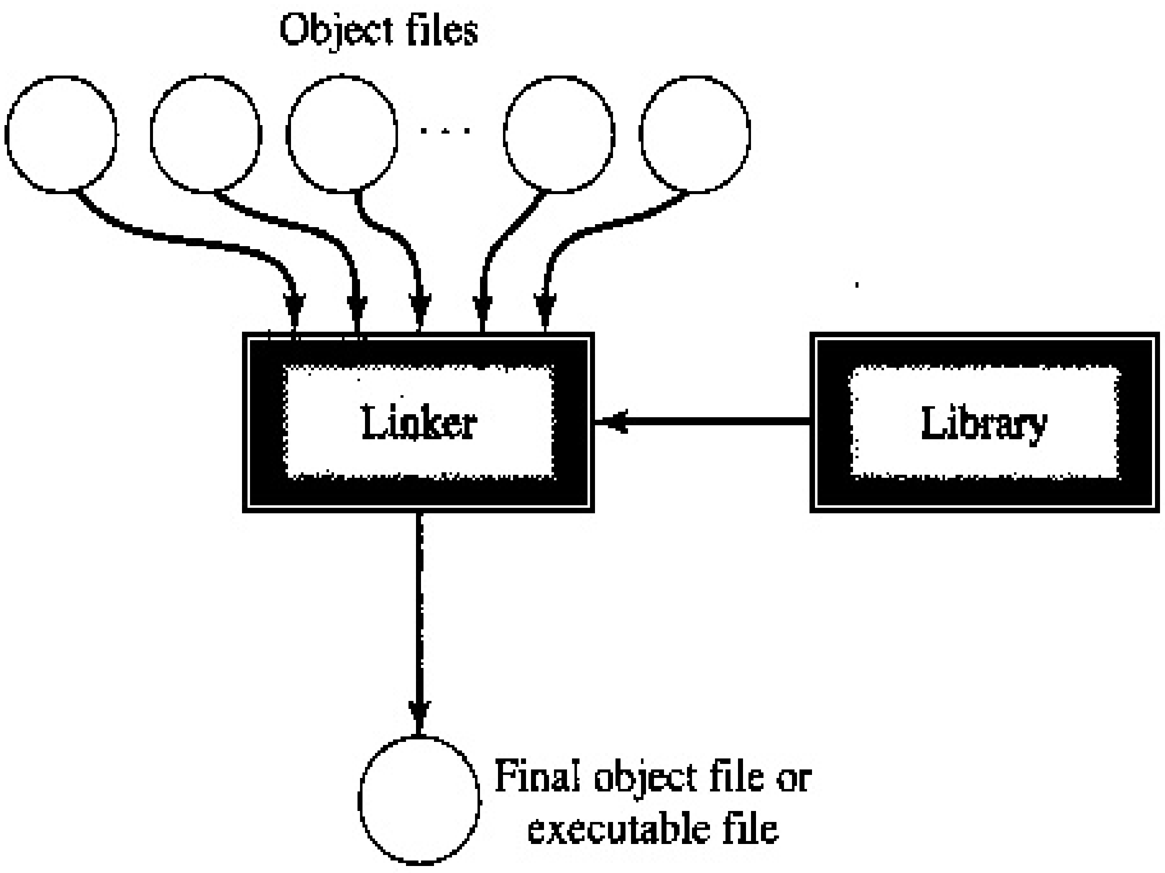
- Assembling
- At assembly time, the assembler:
    - Evaluates conditional-assembly directives, assembling if the conditions are true.
    - Expands macros and macro functions.
    - Evaluates constant expressions such as MYFLAG AND 80H, substituting the calculated value for the expression.
    - Encodes instructions and non address operands. For example, mov cx, 13; can be

encoded at assembly time because the instruction does not access memory.

- Saves memory offsets as offsets from their segments.
- Places segments and segment attributes in the object file.
- Saves placeholders for offsets and segments (relocatable addresses).
- Outputs a listing if requested.

- Passes messages (such as INCLUDELIB) directly to the linker.

- Once your source code is assembled, the resulting object file is passed to the linker. At this point, the linker may combine several object files into an executable program. The linker:
    - Combines segments according to the instructions in the object files, rearranging the positions of segments that share the same class or group.
    - Fills in placeholders for offsets (relocatable addresses).
    - Writes relocations for segments into the header of .EXE files (but not .COM files).
    - Writes the result as an executable program file.



- **Loading**

After loading the executable file into memory, the operating system:

  - Creates the program segment prefix (PSP) header in memory.
  - Allocates memory for the program, based on the values in the PSP.
  - Loads the program.
  - Calculates the correct values for absolute addresses from the relocation table.
  - Loads the segment registers SS, CS, DS, and ES with values that point to the proper areas of memory.

Useful Tools and Utilities

- DUMPBIN disassembly program
- Debuggers: OllyDbg and WinDbg
- Consol I/O: iolib.

Variants of assembler

- There is a list of assemblers, computer programs that translate assembly language source code into binary programs. Some assemblers are components of a compiler system for a high-level language and may have limited or no usable functionality outside of the compiler system.
- - Some assemblers are hosted on the target processor and operating system, while other assemblers (cross-assemblers) may run under an unrelated operating system or processor.

- **As part of a compiler suite**
- **GNU Assembler** (gas): GPL: many target instruction sets including ARM architecture, Atmel AVR, x86, x86-64, Freescale 68HC11, Freescale v4e, Motorola 680x0, MIPS, PowerPC, IBM System z, TI MSP430.
- **ASxxxx Cross Assembler** (part of the Small Device C Compiler project): GPL: several target instruction sets including Intel 8051, Freescale 68HC08, PIC microcontroller.
- **The Amsterdam Compiler Kit (ACK**) targets many architectures of the 1980s, including 6502, 6800, 680x0, ARM, x86 and Z8000.
- LLVM targets many platforms, however emits no per-target assembly language, instead more high-level typed intermediate representation assembly-like language used.

Some others self-hosted native-targeted language implementations (like Go, Free Pascal, SBCL) have their own assemblers with multiple targets. They may be used for inline assembly inside language, or even included as a library, but not always suitable for standalone application - no command-line tool exists, or only intermediate representation used as a source, or support for targets very limited.

Single target assembler

An assembler may have a single target processor or may have options to support multiple processor types. Very simple assemblers may lack features, such as macros, present in more powerful versions.

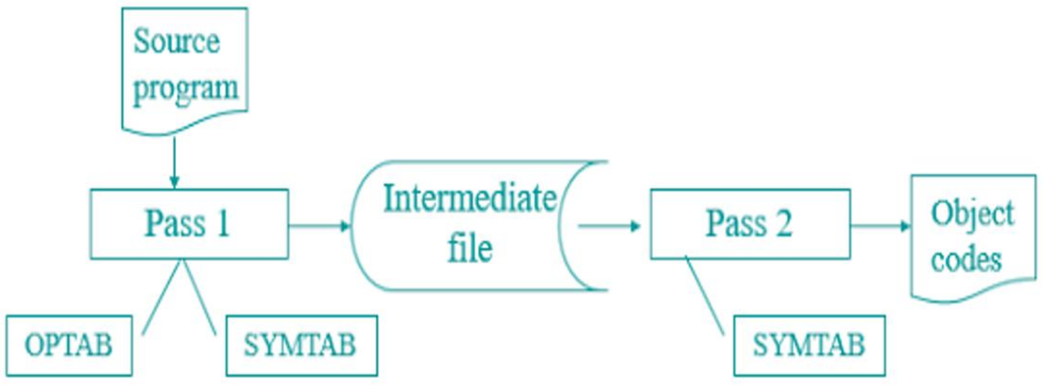
- Various types are:
- 6502 assemblers

- 680x0 assemblers
- ARM assemblers
- Mainframe Assemblers
- POWER, PowerPC, and Power ISA assemblers
- x86 assemblers
- Z80 assemblers
- Other single target assemblers

Design of 2-pass assembler

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

- Two Pass Assembler
- *Read from input line*
    - LABEL, OPCODE, OPERAND



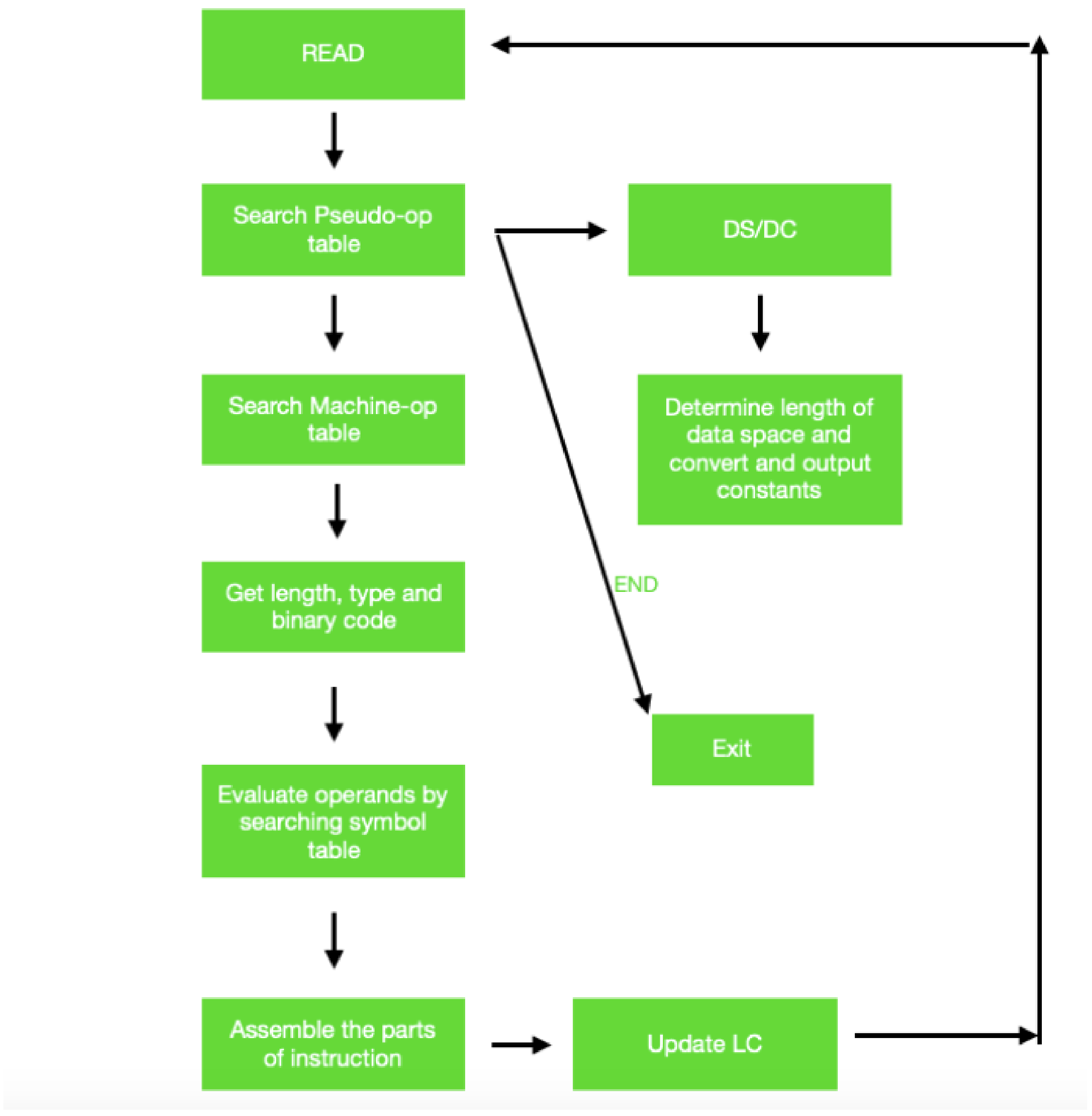
PASS 1:

- Separate the Symbol, Mnemonic opcode, and operand fields
- Build the symbol table
- Perform LC Processing
- Construct Intermediate Representation
- **PASS 2:**
- *SYNTHESIZE THE TARGET PROGRAM*
- Advanced Assembler Directives
- ORIGIN
- EQU

- EQUSyntax:
  <Symbol> EQU <Address
  Specification>E.g.   MAXLEN   EQU   4096 Pass I of
  Assembler
- Pass I Use following Data Structures

  - OPTAB
  - SYMTAB
  - LITTAB
  - POOLTAB

- 2-pass system is to address the problem of forwarding references — references to variables or subroutines that have not yet been encountered when parsing the source code. A strict 1-pass scanner cannot assemble source code which contains forward references. Pass 1 of the assembler scans the source, determining the size and address of all data and instructions; then pass 2 scans the source again, outputting the binary object code.


READ
Search Pseudo-op table
DS/DC
Search Machine-op table
Determine length of data space and convert and output constants
Get length, type and binary code
END
Exit
Evaluate operands by searching symbol table
Assemble the parts of instruction
Update LC