Computer Organization and Architecture

-Er. Hari K.C.

MSc. (Electronics and Computer Engineering)

Syllabus

Chapter 01: Instruction set architecture (ISA)

- Introduction to architecture
- levels of programming language
- Compiling and assembling process
- Assembly language instructions
- Addressing modes
- Design consideration of ISA

Chapter 02: Computer Organization

- Oganization and structure
- Basic computer organization
- Instruction cycle
- Memory subsystem
- I/O subsystem

Chapter 03: RTL and HDL

- Microoperation
- RTL
- Using RTL to specify digital system
- Modulo 6 counter
- Introduction to VHDL

Chapter 04: CPU Design

- Specifying CPU
- Design and implementation of CPU
- Designing hardwired control unit

Chapter 05: Microprogrammed control unit

- Micro-sequencer design
- Microinstruction format
- Generating correct sequence and designing mapping logic
- Microprogrammed vs hardwired control unit

Chapter 06: Arithmetic Unit

- Addition algorithm
- Multiplication algorithms
- Division algorithms

Chapter 07: Memory Organization

- Hierarchical memory system
- Cache memory
- Associative cache
- Mapping techniques
- Virtual memory

Chapter 08: Input-output organization

- Asynchronous data transfer
- Programmed input –output
- DMA

Chapter 09: RISC

- Introduction to RISC
- RISC vs CISC
- RISC pipelining

Chapter 10: Parallel processing

- Multiprocessing
- System topology
- Communication in multiprocessor
- Memory organization in multiprocessor

Chapter one: Instruction set architecture

Instruction set Architecture (ISA):

Architecture: It refer to those attributes of a system visible to a programmer.

Attributes includes instruction set, no. of bits used to represent various data types, I/0 mechanism and addressing modes.

ISA is defined as an architecture that includes microprocessor instruction set, set of all assembly language instructions that microprocessor can execute.

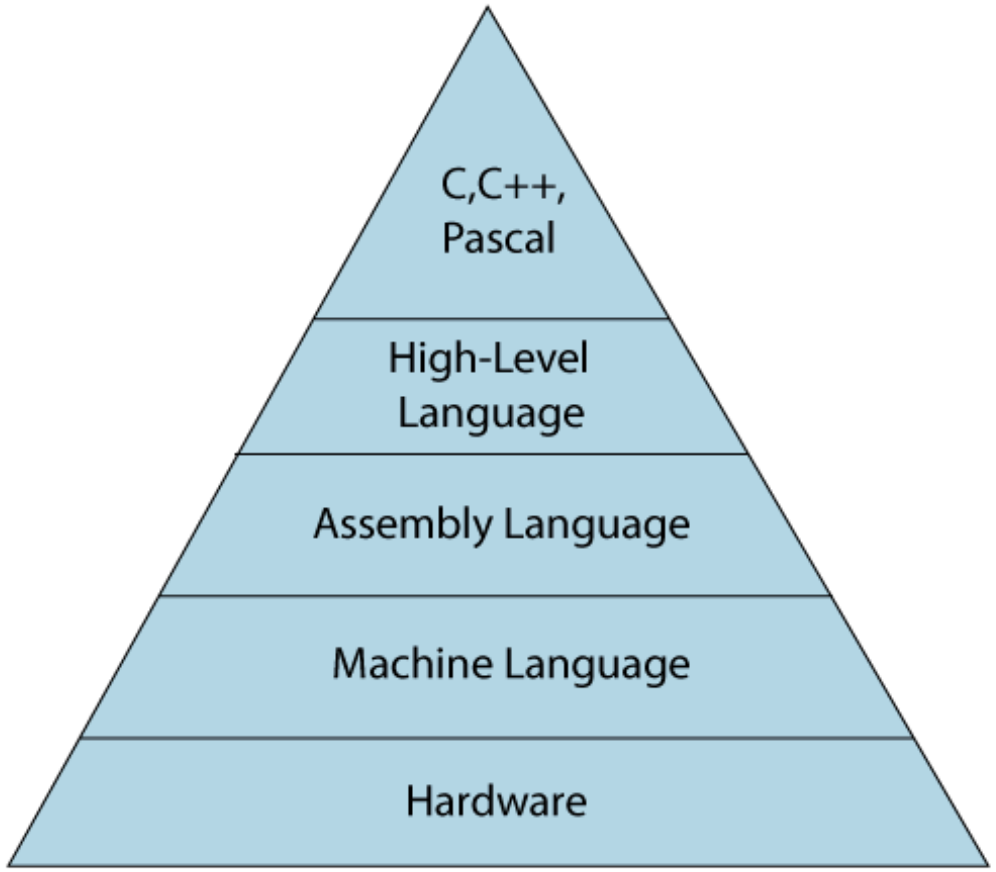
ISA has to specify the processor registers, their sizes and instructions in instruction set that can use each register.

ISA also include information necessary to interact with memory.

Levels of programming language

- A programming language defines a set of instructions that are compiled together to perform a specific task by the CPU (Central Processing Unit).
- The programming language mainly refers to high-level languages such as C, C++, Pascal, Ada, COBOL, etc.
- Each programming language contains a unique set of keywords and syntax, which are used to create a set of instructions.
- Thousands of programming languages have been developed till now, but each language has its specific purpose.
- These languages vary in the level of abstraction they provide from the hardware. Some programming languages provide less or no abstraction while some provide higher abstraction

- Based on the levels of abstraction, they can be classified into two categories:
- Low-level language
- High-level language



Low-level language

- The low-level language is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions.
- The languages that come under this category are the Machine level language and Assembly language.

Machine-level language

- The machine-level language is a language that consists of a set of instructions that are in the binary form 0 or 1.
- As we know that computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1, so the instructions given to the computer can be only in binary codes.
- Creating a program in a machine-level language is a very difficult task as it is not easy for the programmers to write the program in machine instructions.
- It is error-prone as it is not easy to understand, and its maintenance is also very high.
- A machine-level language is not portable as each computer has its machine instructions, so if we write a program in one computer will no longer be valid in another computer.
- The different processor architectures use different machine codes, for example, a PowerPC processor contains RISC architecture, which requires different code than intel x86 processor, which has a CISC architecture.

Assembly Language

- The assembly language contains some human-readable commands such as mov, add, sub, etc.
- The problems which we were facing in machine-level language are reduced to some extent by using an extended form of machine-level language known as assembly language.
- Since assembly language instructions are written in English words like mov, add, sub, so it is easier to write and understand.
- As we know that computers can only understand the machine-level instructions, so we require a translator that converts the assembly code into machine code. The translator used for translating the code is known as an assembler.
- The assembly language code is not portable because the data is stored in computer registers, and the computer has to know the different sets of registers.
- The assembly code is not faster than machine code because the assembly language comes above the machine language in the hierarchy, so it means that assembly language has some abstraction from the hardware while machine language has zero abstraction.

Machine-level language	Assembly language
The machine-level language comes at the lowest level in the hierarchy, so it has zero abstraction level from the hardware.	The assembly language comes above the machine language means that it has less abstraction level from the hardware.
It cannot be easily understood by humans.	It is easy to read, write, and maintain.
The machine-level language is written in binary digits, i.e., 0 and 1.	The assembly language is written in simple English language, so it is easily understandable by the users.
It does not require any translator as the machine code is directly executed by the computer.	In assembly language, the assembler is used to convert the assembly code into machine code.
It is a first-generation programming language.	It is a second-generation programming language

High-Level Language

- The high-level language is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine-level languages.
- When writing a program in a high-level language, then the whole attention needs to be paid to the logic of the problem.
- A compiler is required to translate a high-level language into a low-level language.

Advantages of a high-level language

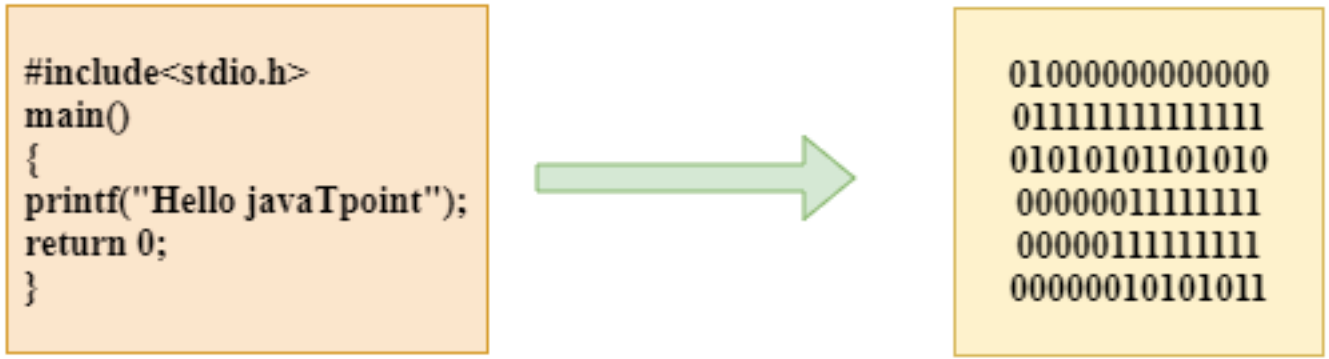
- The high-level language is easy to read, write, and maintain as it is written in English like words.
- The high-level languages are designed to overcome the limitation of low-level language, i.e., portability. The high-level language is portable; i.e., these languages are machine-independent.

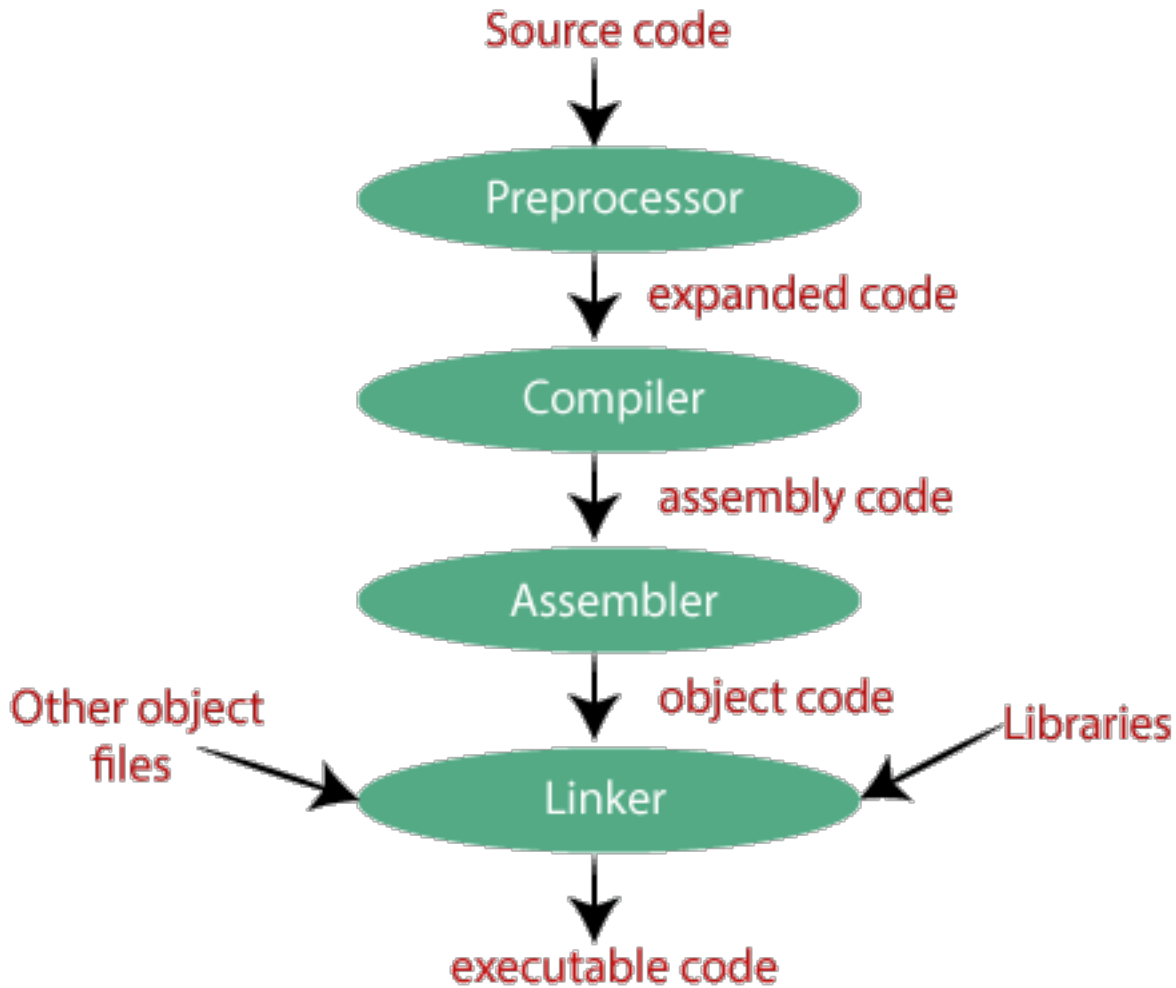
Low-level language	High-level language
It is a machine-friendly language, i.e., the computer understands the machine language, which is represented in 0 or 1.	It is a user-friendly language as this language is written in simple English words, which can be easily understood by humans.
The low-level language takes more time to execute.	It executes at a faster pace.
It requires the assembler to convert the assembly code into machine code.	It requires the compiler to convert the high-level language instructions into machine code.
The machine code cannot run on all machines, so it is not a portable language.	The high-level code can run all the platforms, so it is a portable language.
It is memory efficient.	It is less memory efficient.
Debugging and maintenance are not easier in a low-level language.	Debugging and maintenance are easier in a high-level language.

Compiling and assembling process

The compilation is a process of converting the source code into object code. It is done with the help of the compiler.

The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.




Source code
Preprocessor
expanded code
Compiler
assembly code
Assembler
object code
Other object files
Libraries
Linker
executable code

Preprocessor

- The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

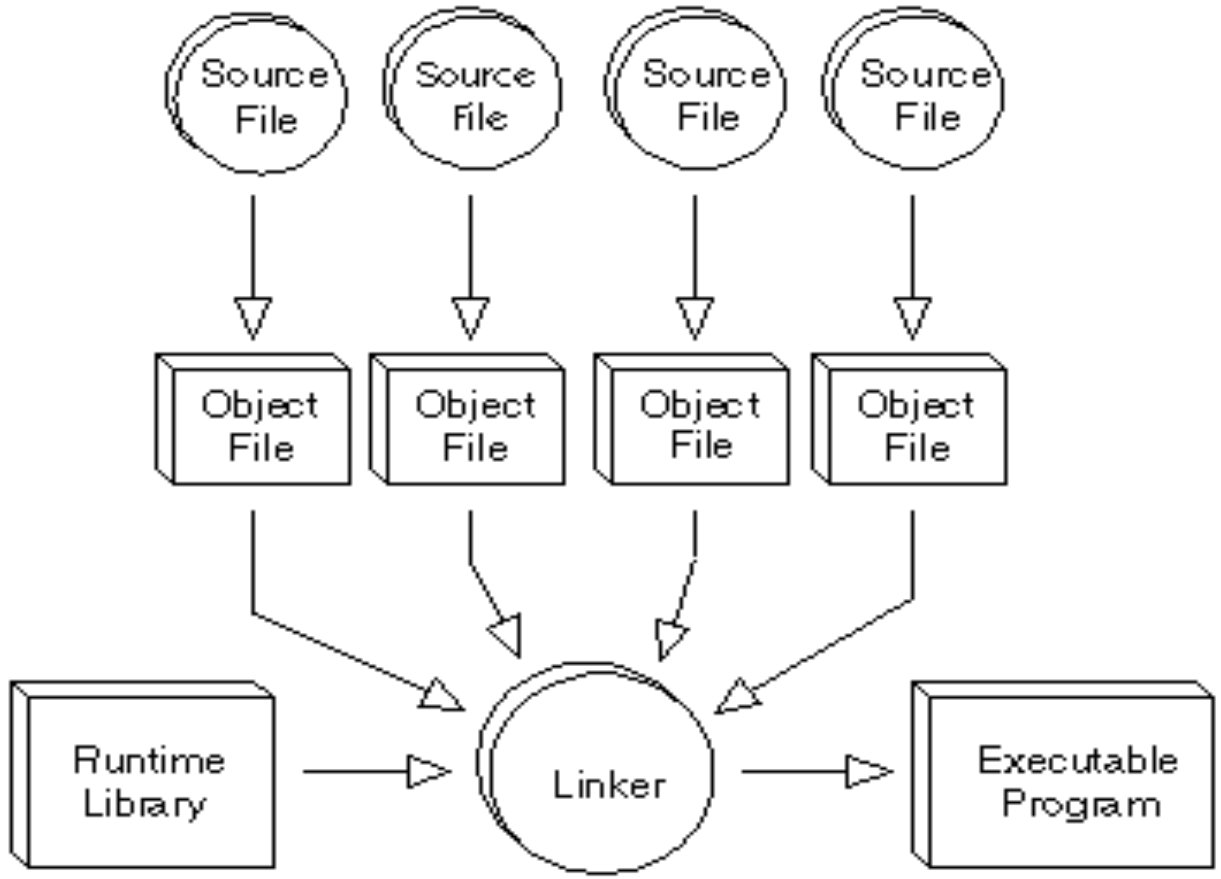
Compiler

- The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

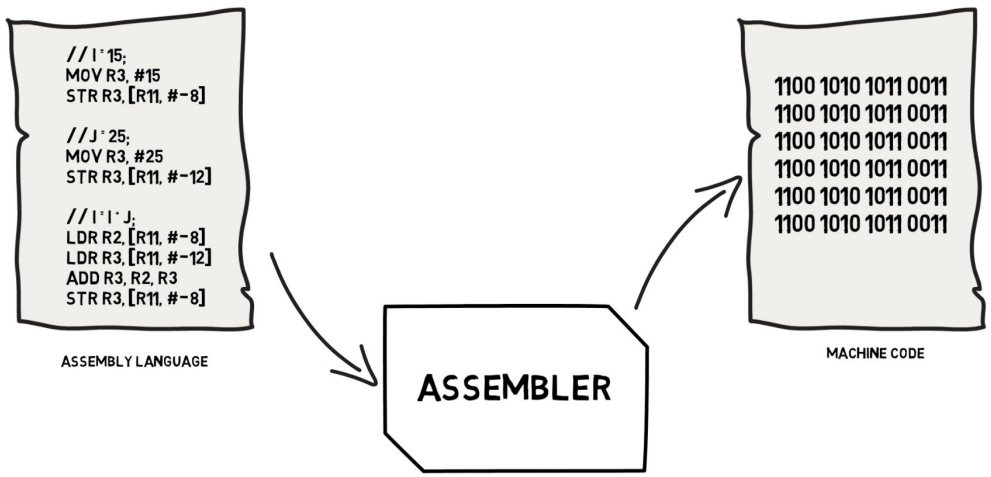
- The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

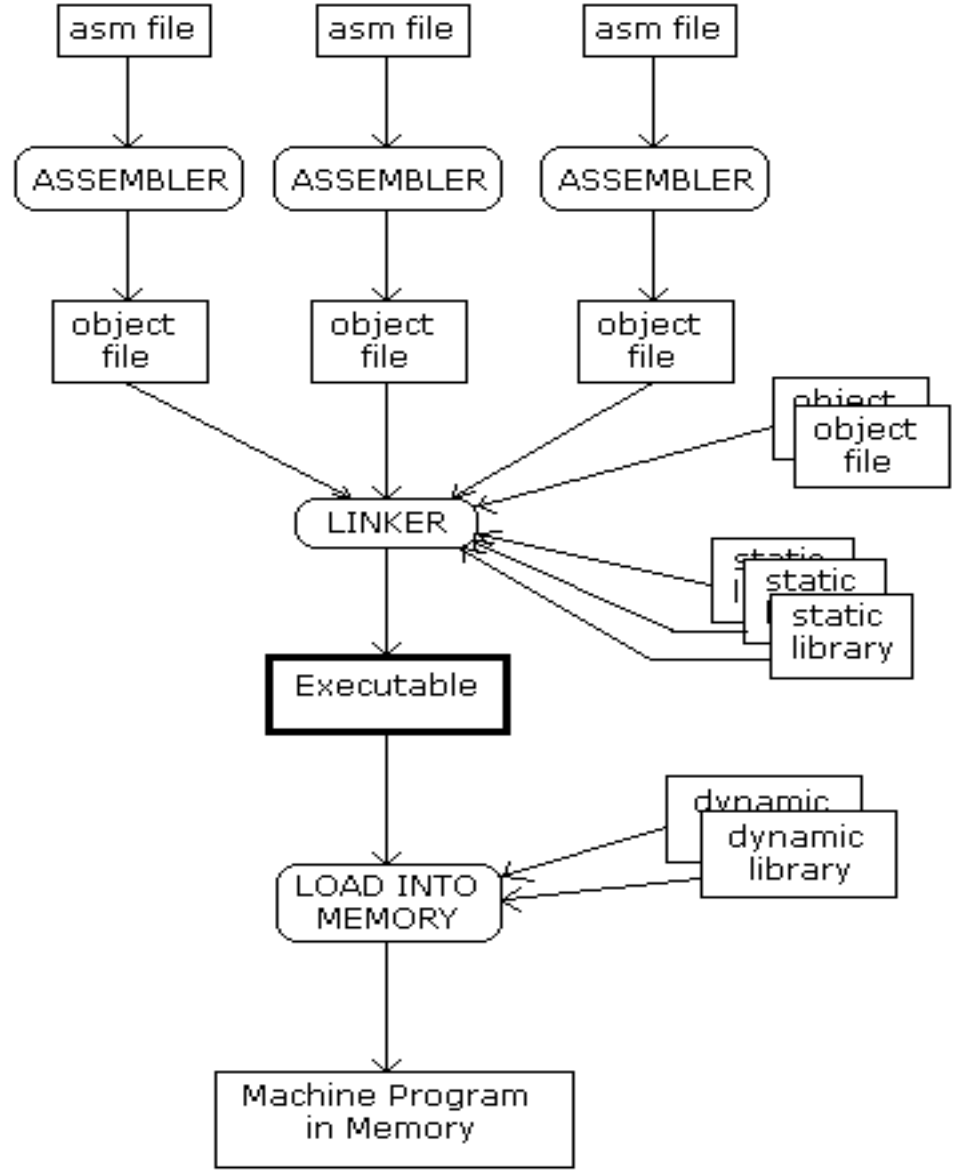
Compilation process



- To transform a program written in a high-level programming language from *source code* into *object code*.
- Programmers write programs in a form called source code. Source code must go through several steps before it becomes an executable program.
- The first step is to pass the source code through a *compiler*, which translates the high-level language instructions. into object code.
- The final step in producing an executable program -- after the compiler has produced object code -- is to pass the object code through a *linker*.
- The linker combines modules and gives real values to all symbolic addresses, thereby producing machine code.

Assembly process





Addressing modes

- The various ways of specifying the operands for an instruction.

1. Register Addressing

- Instruction gets its source data from a register.
- Data resulting from the operation is stored in another register. • Data length depends on register being used.

– 8-bit registers: AH, AL, BH, BL, CH, CL, DH, DL.

– 16-bit registers: AX, BX, CX, DX, SP, BP, SI, DI.

– 32-bit registers: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI.

– 64-bit registers: RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, and R8 through R15.

Examples: – MOV AX, BX ;Copy the 16-bit content of BX to AX

– MOV AL, BL ;Copy the 8-bit content of BL to AL

– MOV SI, DI ;Copy DI into SI

– MOV DS, AX ;Copy AX into DS

- Note that the instruction must use registers of the same size. – Cannot mix between 8-bit and 16-bit registers.

– Will result in an error when assembled.

2. Immediate Addressing

- The source data is coded directly into the instruction.
- The term immediate means that the data immediately follows the hexadecimal opcode in the memory.
- Immediate data are constant data such as a number, a character, or an arithmetic expression.
- Examples: – MOV AX, 100 – MOV BX, 189CH – MOV AH, 10110110B – MOV AL, (2 + 3)/5

3. Direct Addressing

- The operand is stored in a memory location, usually in data segment.
- The instruction takes the offset address.
- – This offset address must be put in a bracket [ ].
- Example: – MOV [1234H], AL
- – The actual memory location is obtained by combining the offset address with the segment address in the segment register DS (unless specified otherwise)
- If we want to use another segment register such as ES, you can use the syntax ES:[1234H] – Assuming DS = 1000H, then this instruction will move the content of AL into the memory location 1234H.

4. Register Indirect Addressing

- Similar to direct data addressing, except that the offset address is specified using an index or base register.
- Base registers = BP, BX. Index registers = DI, SI.
- In 80386 and above, any register (EAX, EBX, ECX, EDX, EBP, EDI, ESI) can store the offset address.
- The registers must be specified using a bracket [ ]
- DS is used as the default segment register for BX, DI and SI.
- Example: – MOV AX, [BX] – Assuming DS = 1000H and BX = 1234H, this instruction will move the content memory location 11234H and 11235H into AX.

5. Base-plus-index Addressing

- Similar to register indirect addressing, except that the offset address is obtained by adding a base register (BP, BX) and an index register (DI, SI).
- Example: – MOV [BX+SI], BP
- Assuming DS = 1000H, BX = 0300H and SI = 0200H, this instruction will move the content of register BP to memory location 10500H.

6. Register Relative Addressing

- Similar to register indirect addressing, except that the offset address is obtained by adding an index or base register with a displacement.
- Example 1: – MOV AX, [DI+100H]

– Assuming DS = 1000H and DI = 0300H, this instruction will move the content from memory location 10400H into AX.

Example 2: – MOV ARRAY[SI], BL

– Assuming DS = 1000H, ARRAY = 5000H and SI = 500H,

this instruction will move the content in register BL to memory location 15500H.

7. Base Relative-plus-index Addressing

- Combines the base-plus-index addressing and relative addressing
- Examples: –
- MOV AH, [BX+DI+20H]
- MOV FILE[BX+DI], AX
- MOV LIST[BP+SI+4], AL

Instruction and its types

- Data transfer instruction : mov a, b, sta 2050 h
- Logical instructions: ani 20h , rcl, xra b
- Arithmetic instructions : add b , adi 30h , inr c
- Branching instructions: jnz 2020h, jc 3030h, jmp 2020
- Input-output instructions: In 01 h, Out 20h
- Stack instructions: Push, pop

Data types

- Numeric data: integers, hexadecimal
- Floating data: decimal , fractional value
- Boolean data: true, false
- Character data: a, b or strings

Instruction formats

- 3 address instruction formats: add a,b,c
- 2 address instruction formats: add b,c
- 1 address instruction formats: add b
- 0 address instruction formats: push, pop, add, sub

Design consideration of ISA

- Orthogonality of instructions: minimum instructions to perform tasks
- Registers set: large no. of registers
- Backward compatibility: run program of previous processors
- Data size and types: varieties of data types
- Instruction format and addressing modes: varieties of instruction format and addressing modes for flexibility.

End of chapter one