Chapter 1

Instruction Set Architecture

Er. Shiva Ram Dam

Assistant Professor, Pokhara University

Contents:

1. Computer Organization and Architecture
2. Levels of programming language
3. The compilation process
4. Instruction set architecture
5. Instruction and its elements
6. Operand datatypes
7. Instruction types
8. Instruction format
9. Addressing modes
10. Design issues of ISA

1.1 Computer Organization

- **Computer organization** refers to the operational units and their interconnection to form the computer system that can perform as intended.
- Organizational attributes include the hardware detail such as:
  - Control signal computer
  - Interfaces between the computer and peripherals
  - Memory technology used, etc

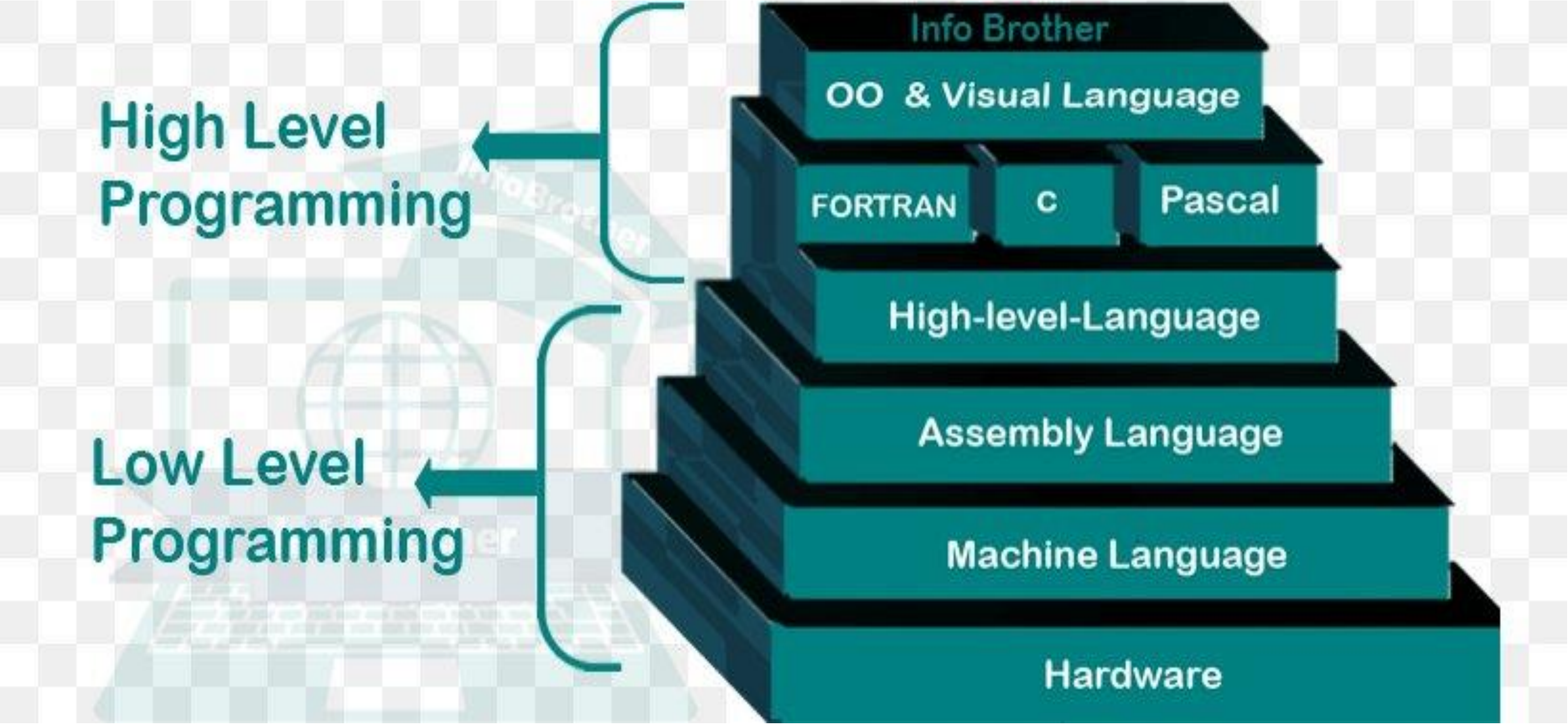
1.2 Computer Architecture

- **Computer architecture** refers to those attributes of a system that have a direct impact on the logical execution of a program.
- It is concerned with the structure and behavior of the computer as seen by the user,
- i.e. those attributes of a system visible to a programmer.
- It deals with the structure of behavior of various functional modules and how they interact to provide the processing needs of the computer.
- It includes:
  - The instruction set
  - The no. of bits used to represent various data types.
  - I/O mechanism
  - Memory addressing techniques

CO vs CA

- The difference between architecture and organization is best described by a non-computer example.
  - Is the gear level in a motorcycle part of it is architecture or organization?
  - The architecture of a motorcycle is simple; it transports you from A to B.
  - The gear level belongs to the motorcycle's organization because it implements the function of a motorcycle but is not part of that function.
- We can take another example of multiplying two numbers.
  - Computer architecture defines whether direct multiplication instruction is used or multiply by repeated addition.
  - Whereas, computer organization defines whether multiplication unit is used or repeated addition by adder.

1.3 Levels of Programming language



1. Low-level language

- Low-level language is a programming language in which each statement or instruction is directly translated into a single machine code.
- LLL are much closer to the computer or hardware.
- Before creating a program for hardware, it is required to have through or sound knowledge of that hardware.
- A program cannot be run on different hardware, which means LLL are specific to the hardware.
- LLL is also divided into two types:
  - Machine language
  - Assembly language

a) Machine language

- Machine languages are the lowest level programming languages.
- A computer understands the programs written only in the machine language.
- While easily understood for humans to use, they consist entirely of numbers.
- Ultimately, machine code consists entirely of 0’s and 1’s of the binary system which are also called bits.
- The machine code may be different from machine to machine, i.e. they are hardware dependent.
- Programs written in machine language are faster and efficient.
- Writing programs in machine language is very tedious, time consuming, difficult to find bugs in longer programs.

Merits:

- It is directly understood by the processor so has faster execution time since the programs written in this language need not to be translated.
- It doesn’t need larger memory.

Demerits:

- It is very difficult to program using 1GL since all the instructions are to be represented by 0s and 1s.
- Use of this language makes programming time consuming.
- It is difficult to find error and to debug.
- It can be used by experts only.

b) Assembly language

- To overcome the difficulties of programming in machine language, assembly languages were developed.
- An assembly language contains the same instruction as a machine language.
- But each instruction and variable have a symbol instead of being just numbers.
- And these symbols are called ‘mnemonics’.
- For e.g., if 78 is the number to command “Add two numbers”, ADD could be used to replace it.
- After developing assembly language, it was easier to program using symbols instead of numbers.
- But the programs written in assembly language, however, must be converted to machine language, which could be done by assembler.
- They are also machine-dependent.

Merits:

- It makes programming easier than 1GL since it uses mnemonics code for programming. Eg: ADD for addition, SUB for subtraction, DIV for division, etc.
- It makes programming process faster.
- Error can be identified much easily compared to 1GL.
- It is easier to debug than machine language.

- **Demerits:**
- Programs written in this language is not directly understandable by computer so translators should be used.
- It is hardware dependent language so programmers are forced to think in terms of computer's architecture rather than to the problem being solved.
- Being machine dependent language, programs written in this language are very less or not portable.
- Programmers must know its mnemonics codes to perform any task.

2. High Level Language

- The programming languages are called HLL if their syntax is closer to the human languages.
- HLL were developed to make programs easier.
- Most of the HLL are English-like language.
- They use familiar English words, special symbols and mathematical symbols in their syntax.
- Therefore, HLL are easier to read, write, understand and program.
- Each HLL has their own set of grammar and rules to represent a set of instructions.
- Programming languages such as C, C++, Java, etc. are some examples of HLL.
- Like an assembly language programs, programs written in HLL also need to be translated into machine language.
- This can be done either by a compiler or an interpreter.

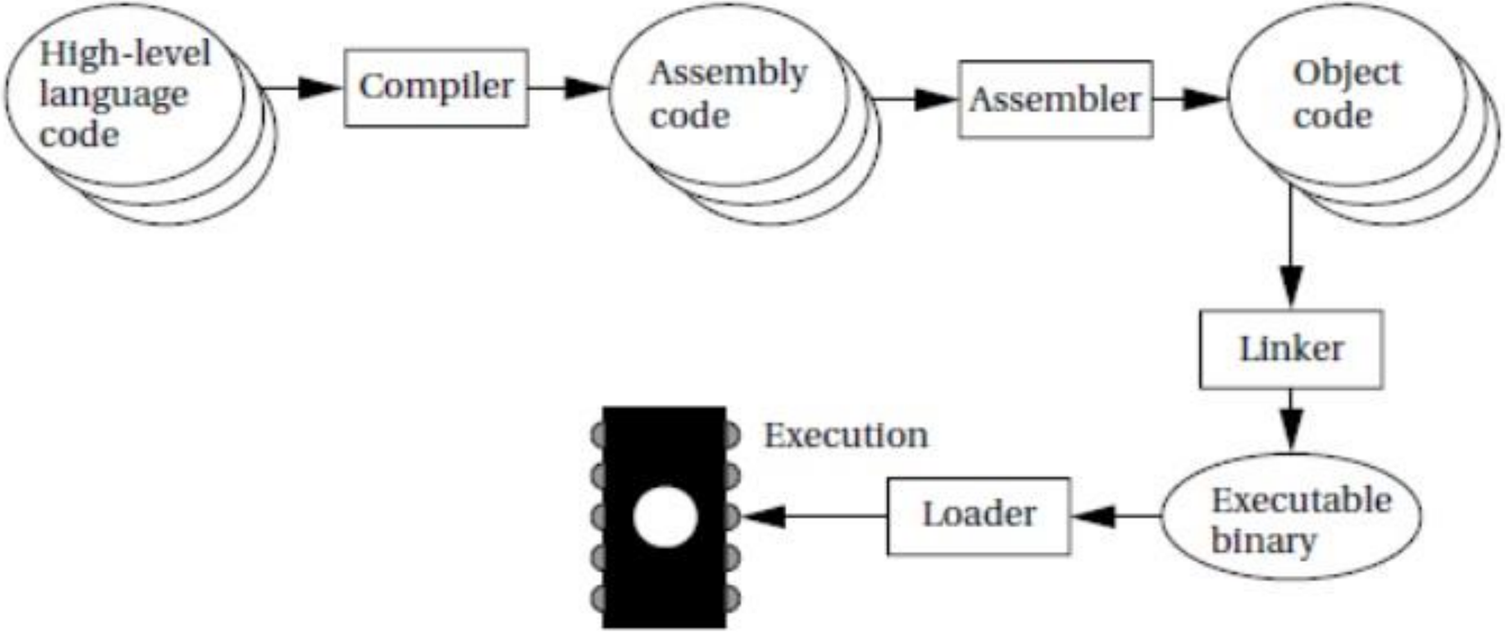
Merits:

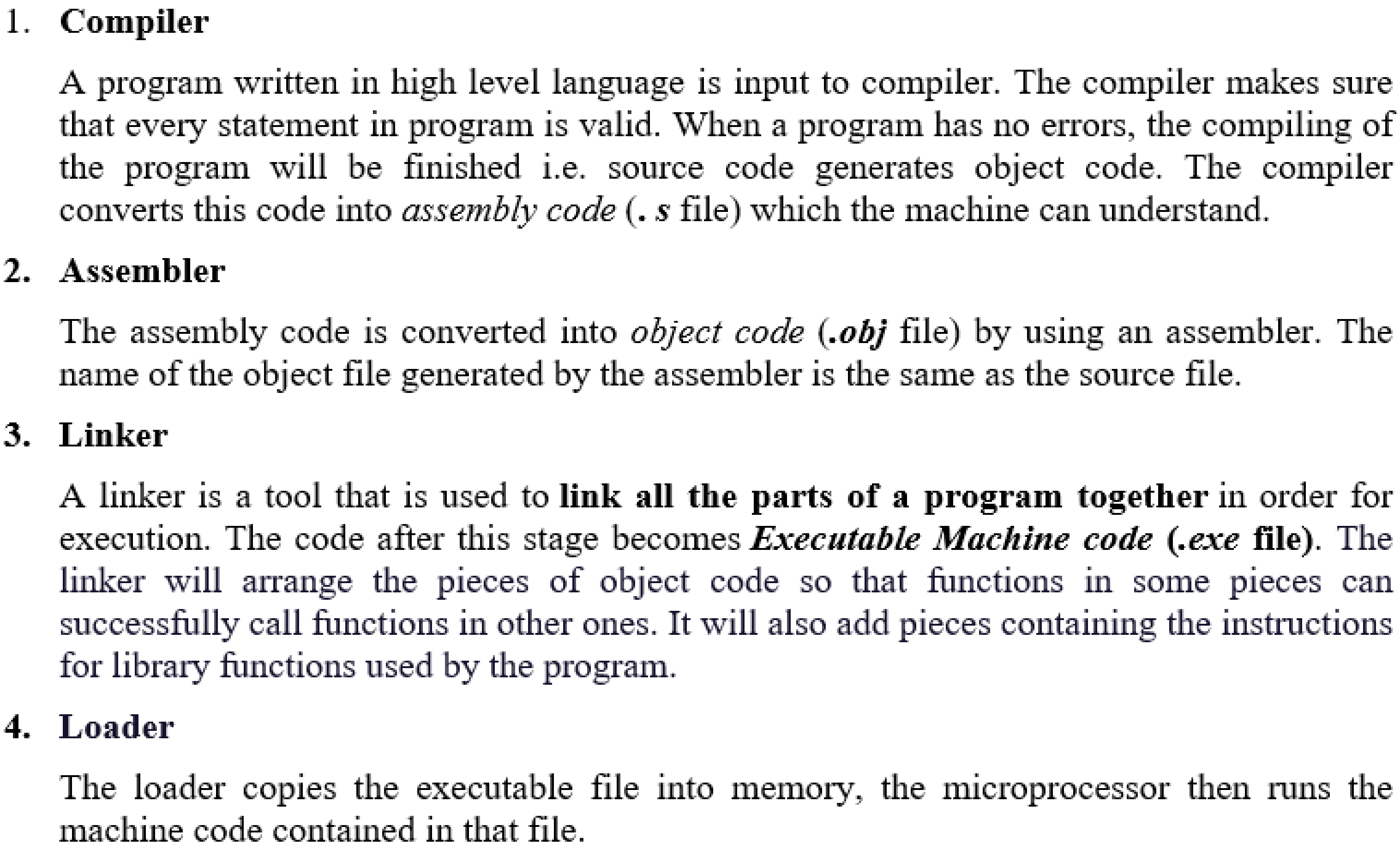
- Because of their flexibility, procedural languages are able to solve a variety of problems.
- Programmer does not need to think in term of computer architecture which makes them focused on the problem.
- Programs written in this language are portable.

Demerits:

- It is easier but needs higher processor and larger memory.
- It needs to be translated therefore its execution time is more.

1.4 The Compilation process





1. **Compiler**

   A program written in high level language is input to compiler. The compiler makes sure that every statement in program is valid. When a program has no errors, the compiling of the program will be finished i.e. source code generates object code. The compiler converts this code into *assembly code* (**.** ***s*** file) which the machine can understand.

2. **Assembler**

   The assembly code is converted into *object code* (***.obj*** file) by using an assembler. The name of the object file generated by the assembler is the same as the source file.

3. **Linker**

   A linker is a tool that is used to **link all the parts of a program together** in order for execution. The code after this stage becomes ***Executable Machine code*** **(*.exe* file)**. The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other ones. It will also add pieces containing the instructions for library functions used by the program.

4. **Loader**

   The loader copies the executable file into memory, the microprocessor then runs the machine code contained in that file.

1.5 Instruction set Architecture (ISA)

- The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler designer.
- They are the parts of a processor design that need to be understood in order to write assembly language, such as the machine language instructions and registers.
- The ISA serves as the boundary between software and hardware.
- The ISA defines
  - the supported data types,
  - the registers,
  - how the hardware manages main memory,
  - key features (such as virtual memory), which instructions a microprocessor can execute, and
  - the input/output model of multiple ISA implementations.

Instruction Set

- The collection of different instructions that the processor can execute is referred to as the processor’s instruction set.
- The instruction format involves:
  - The instruction length,
  - the type;
  - length and position of operation codes in an instruction;
  - and the number and length of operand addresses, etc

1.6. Opcode and Operands

- **Opcode: (What operation to perform?)**
  - An operation code field termed as **opcode** that specifies the operation to be performed.
- **Operands: (Where are the operands?)**
  - An address field of operand on which data processing is to be performed.
  - An operand can reside in the memory or a processor register or can be incorporated within the operand field of instruction as an immediate constant.
  - Therefore, a mode field is needed that specifies the way the operand or its address is to be determined.
- Eg: MOV A, B

  Here MOV is opcode and A, B are operands

Elements of an instruction

- **Operation code:** Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or opcode.
- **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.
- **Result operand reference:** The operation may produce a result.
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

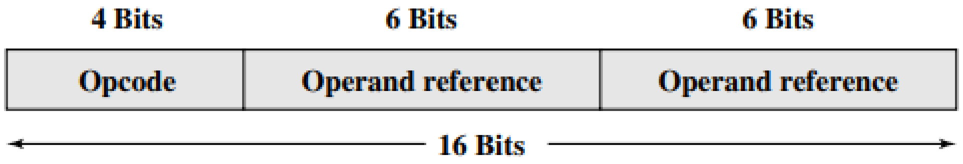


Figure: A Simple instruction format

1.7. Operands data types

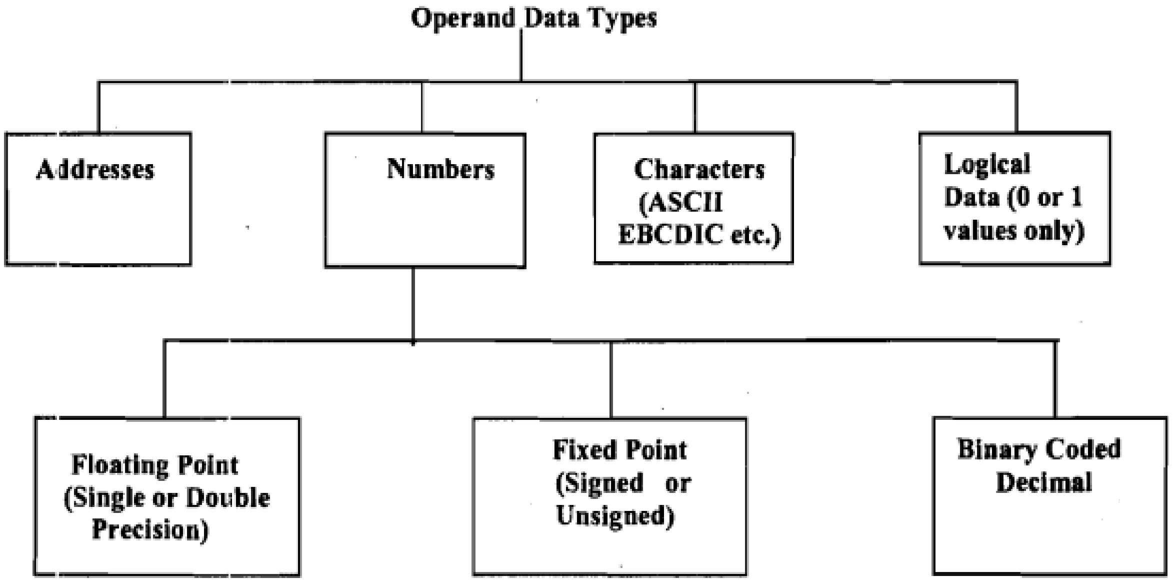
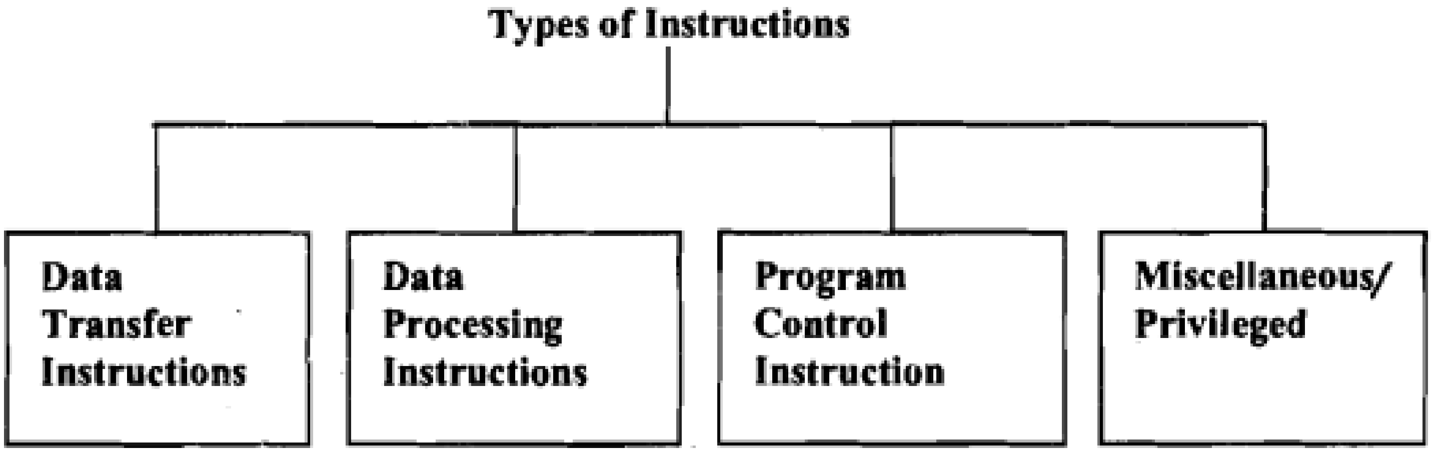

Operand Data Types
Addresses
Numbers
Characters (ASCII EBCDIC etc.)
Logical Data (0 or 1 values only)
Floating Point (Single or Double Precision)
Fixed Point (Signed or Unsigned)
Binary Coded Decimal

Figure 2: Operand Data Types

1.7. Operands data types (Contd.)

- **Address:** Addresses provided in the instruction are operand references.
- **Numbers:** All machine languages include numeric data types like floating-point numbers, fixed-point numbers, binary coded decimal numbers
- **Characters:** A common form of data is text or character strings. Represented in ASCII, EBCDIC
- **Logical data:** Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data.

1.8 Instruction type



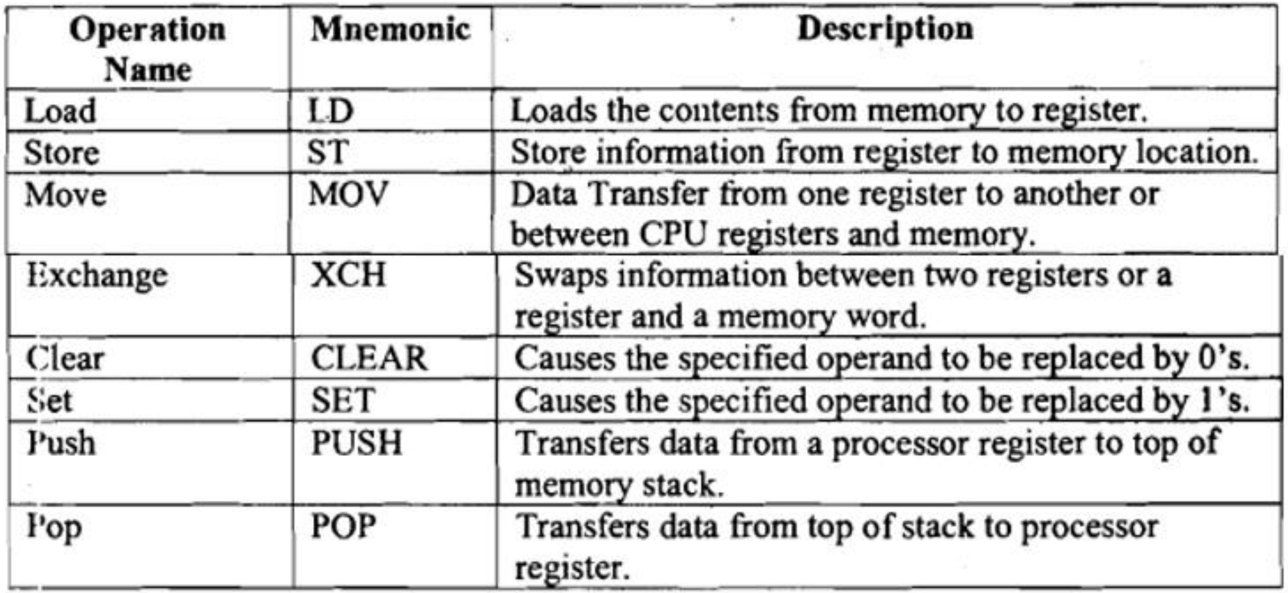
Types of Instructions

- Data Transfer Instructions
- Data Processing Instructions
- Program Control Instruction
- Miscellaneous/ Privileged

Figure 3: Types of Instructions

1. Data Transfer Instructions

- These instructions transfer data from one location in the computer to another location without changing the data content.

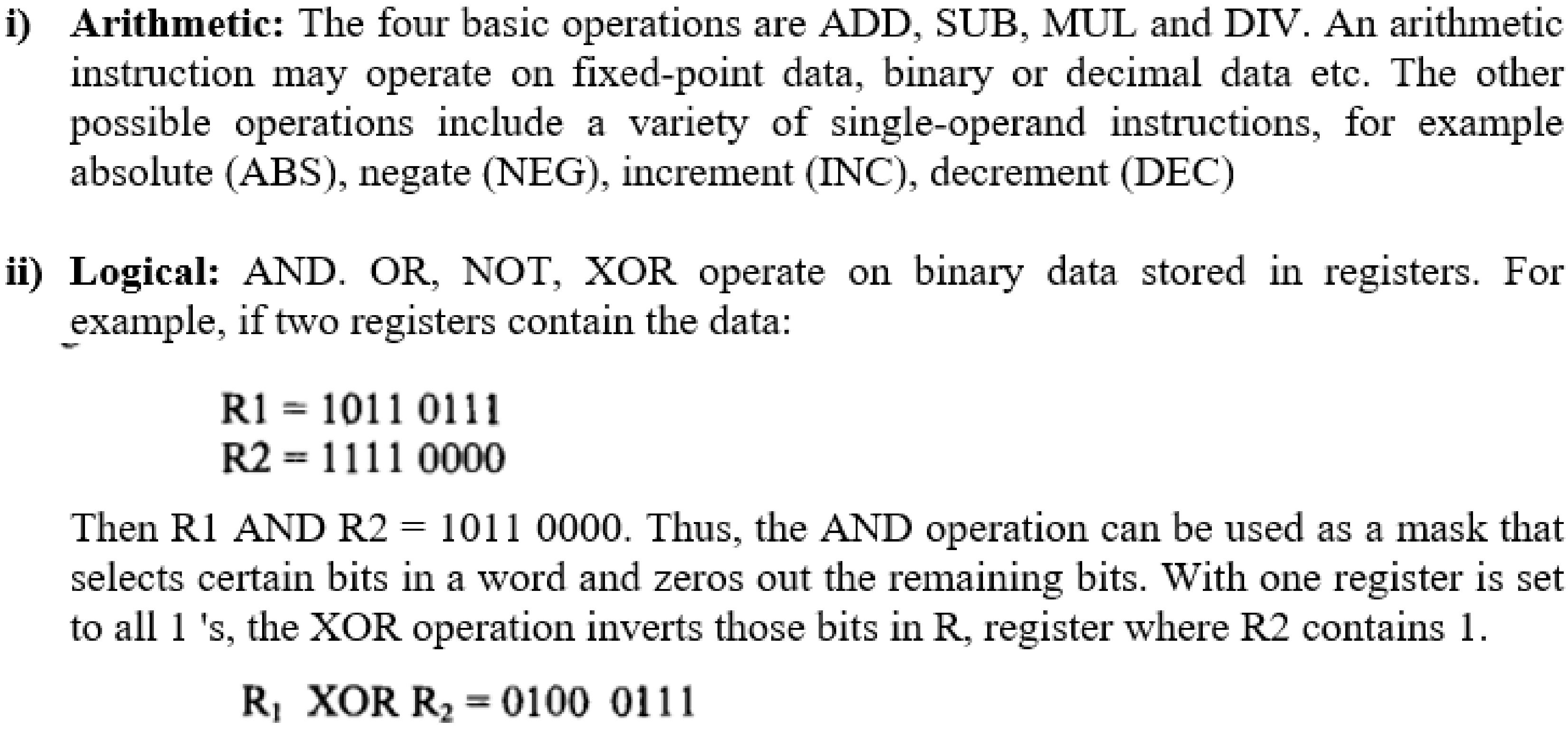


Operation Name	Mnemonic	Description
Load	LD	Loads the contents from memory to register.
Store	ST	Store information from register to memory location.
Move	MOV	Data Transfer from one register to another or between CPU registers and memory.
Exchange	XCH	Swaps information between two registers or a register and a memory word.
Clear	CLEAR	Causes the specified operand to be replaced by 0's.
Set	SET	Causes the specified operand to be replaced by 1's.
Push	PUSH	Transfers data from a processor register to top of memory stack.
Pop	POP	Transfers data from top of stack to processor register.

2. Data Processing (or manipulation) instructions

- These instructions perform arithmetic and logical operations on data.
- Data manipulation Instructions can be divided into three basic types:
  - Arithmetic
  - Logical
  - Shift

2. Data Processing instructions (Contd.)



i) **Arithmetic:** The four basic operations are ADD, SUB, MUL and DIV. An arithmetic instruction may operate on fixed-point data, binary or decimal data etc. The other possible operations include a variety of single-operand instructions, for example absolute (ABS), negate (NEG), increment (INC), decrement (DEC)

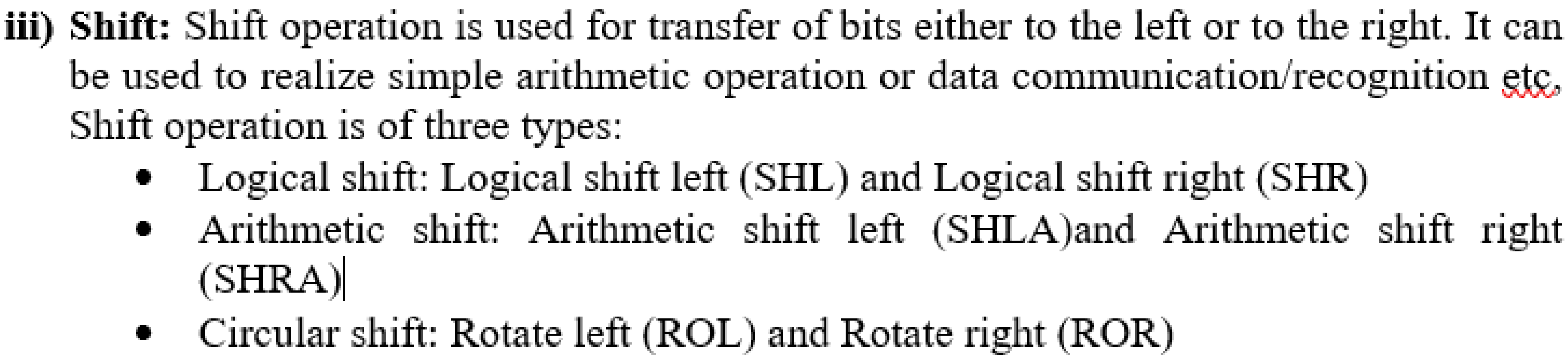
ii) **Logical:** AND. OR, NOT, XOR operate on binary data stored in registers. For example, if two registers contain the data:

R1 = 1011 0111
R2 = 1111 0000

Then R1 AND R2 = 1011 0000. Thus, the AND operation can be used as a mask that selects certain bits in a word and zeros out the remaining bits. With one register is set to all 1 's, the XOR operation inverts those bits in R, register where R2 contains 1.

R_1 XOR R_2 = 0100 0111

2. Data Processing instructions (Contd.)



iii) **Shift:** Shift operation is used for transfer of bits either to the left or to the right. It can be used to realize simple arithmetic operation or data communication/recognition etc, Shift operation is of three types:

- Logical shift: Logical shift left (SHL) and Logical shift right (SHR)
- Arithmetic shift: Arithmetic shift left (SHLA)and Arithmetic shift right (SHRA)
- Circular shift: Rotate left (ROL) and Rotate right (ROR)

Logical shift operations

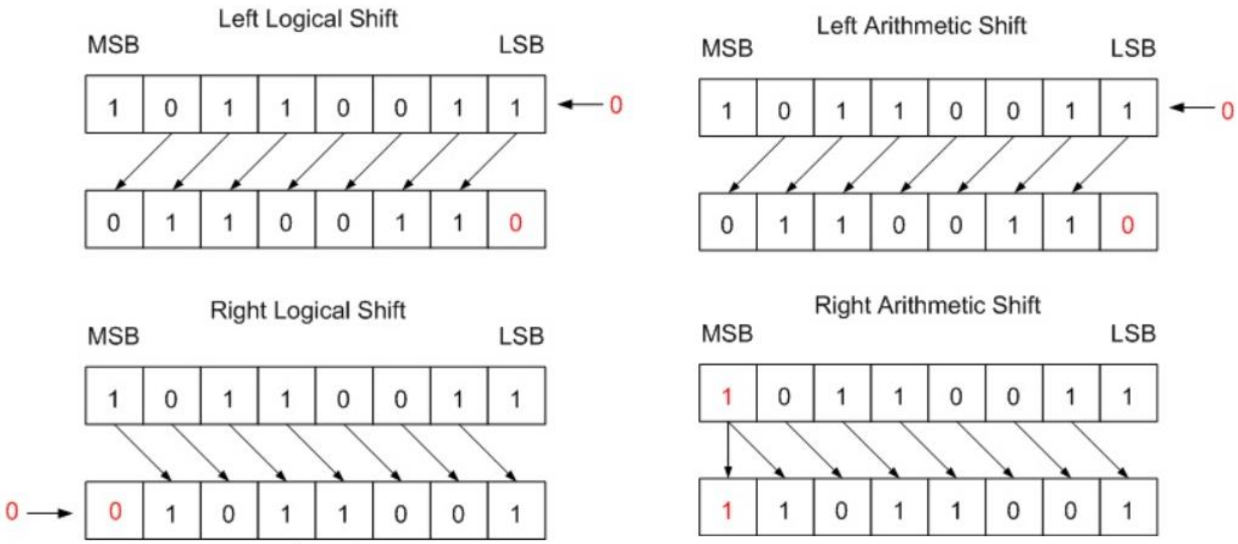
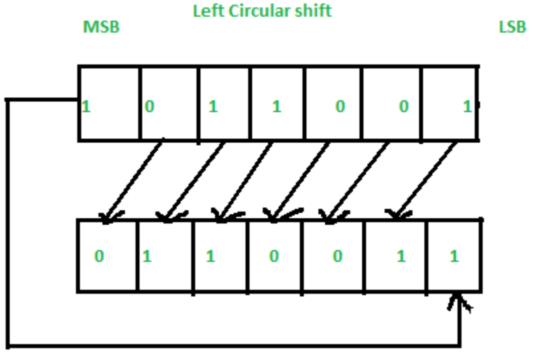
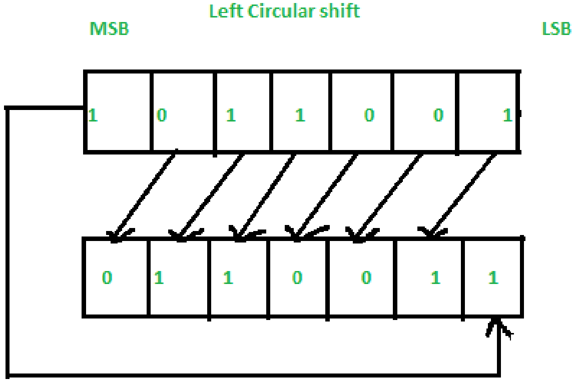


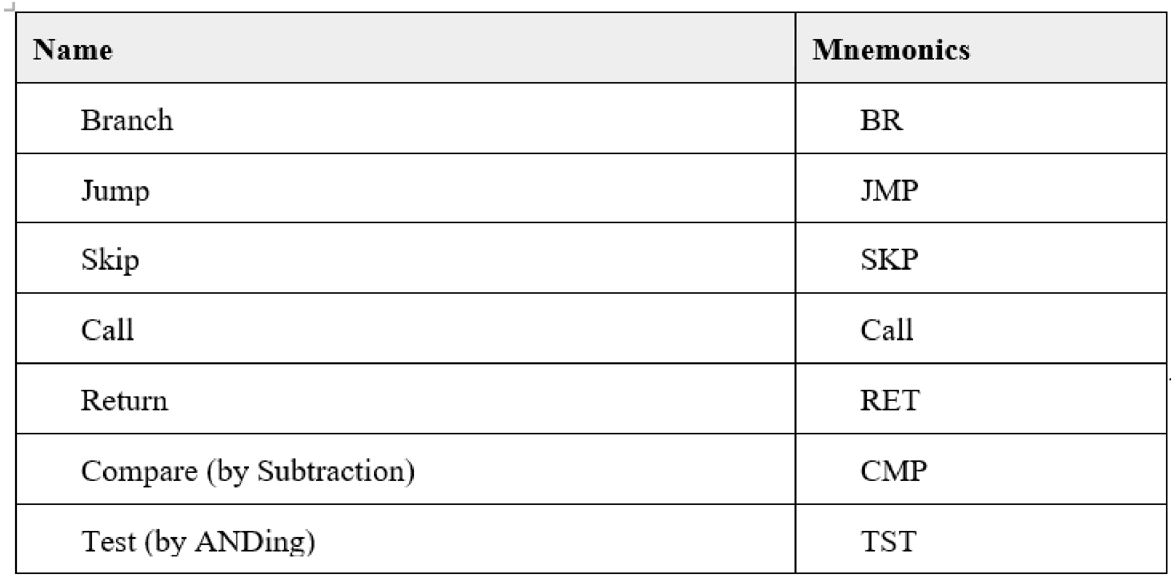
Fig. 1 Logical Shift by one bit





3. Program sequencing and control instructions

The program control instructions control the flow of program execution and are capable of branching to different program segments.



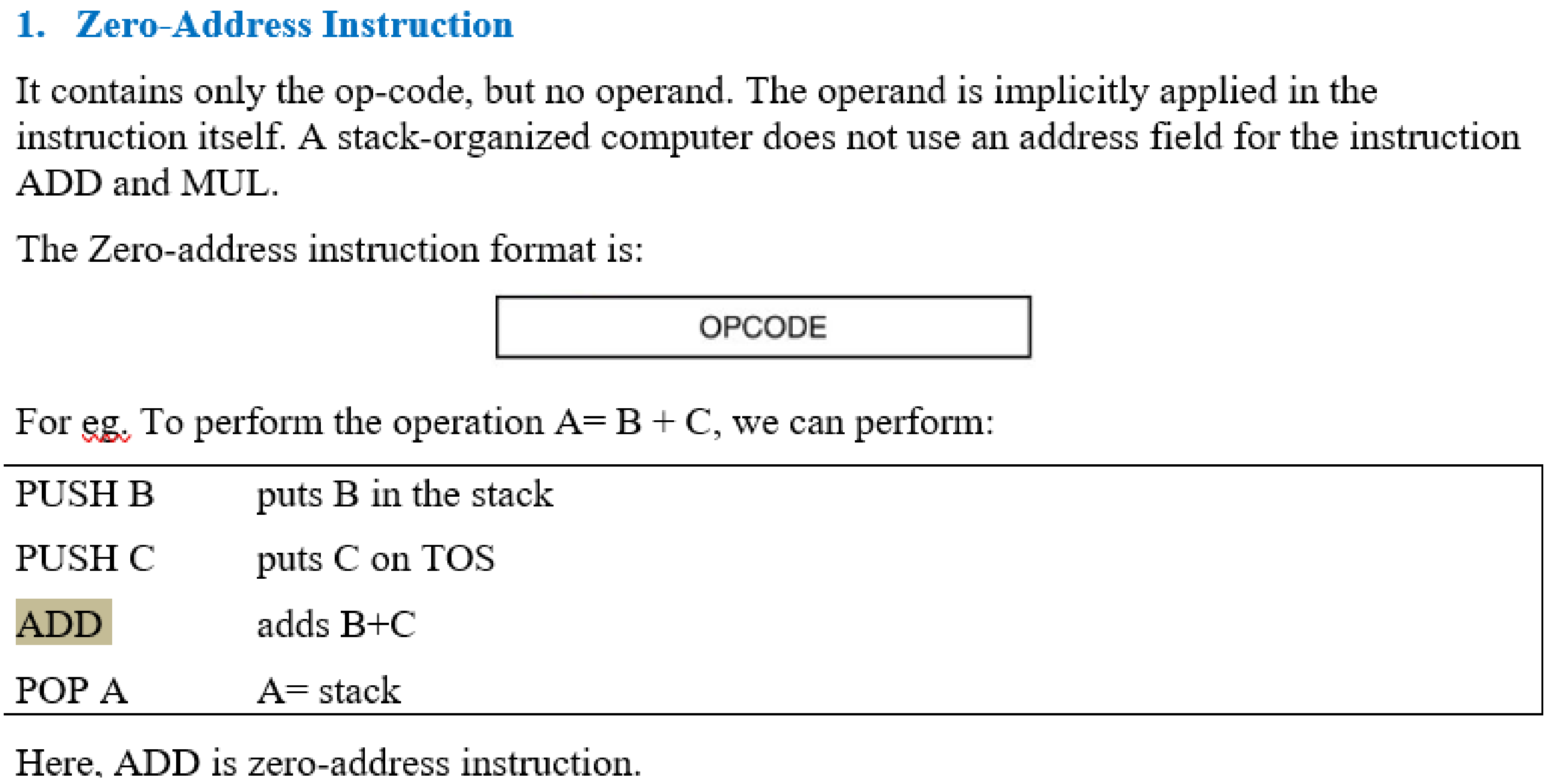
Name	Mnemonics
Branch	BR
Jump	JMP
Skip	SKP
Call	Call
Return	RET
Compare (by Subtraction)	CMP
Test (by ANDing)	TST

4. Miscellenous Instructions

- These instructions do not fit in any of the above categories.
- Those instructions for I/O, interrupt, halt, timer setting, memory clearing, etc fall in this category.

1.9 Address Instruction Formats

- An Instruction Format defines the layout of the bits of an instruction in terms of its constituent fields.
- An Instruction must include an opcode and zero or more operands either implicitly or explicitly.
- Four types:
  - Zero-Address instruction
  - One-Address instruction
  - Two-Address instruction
  - Three-Address instruction



1. Zero-Address Instruction

It contains only the op-code, but no operand. The operand is implicitly applied in the instruction itself. A stack-organized computer does not use an address field for the instruction ADD and MUL.

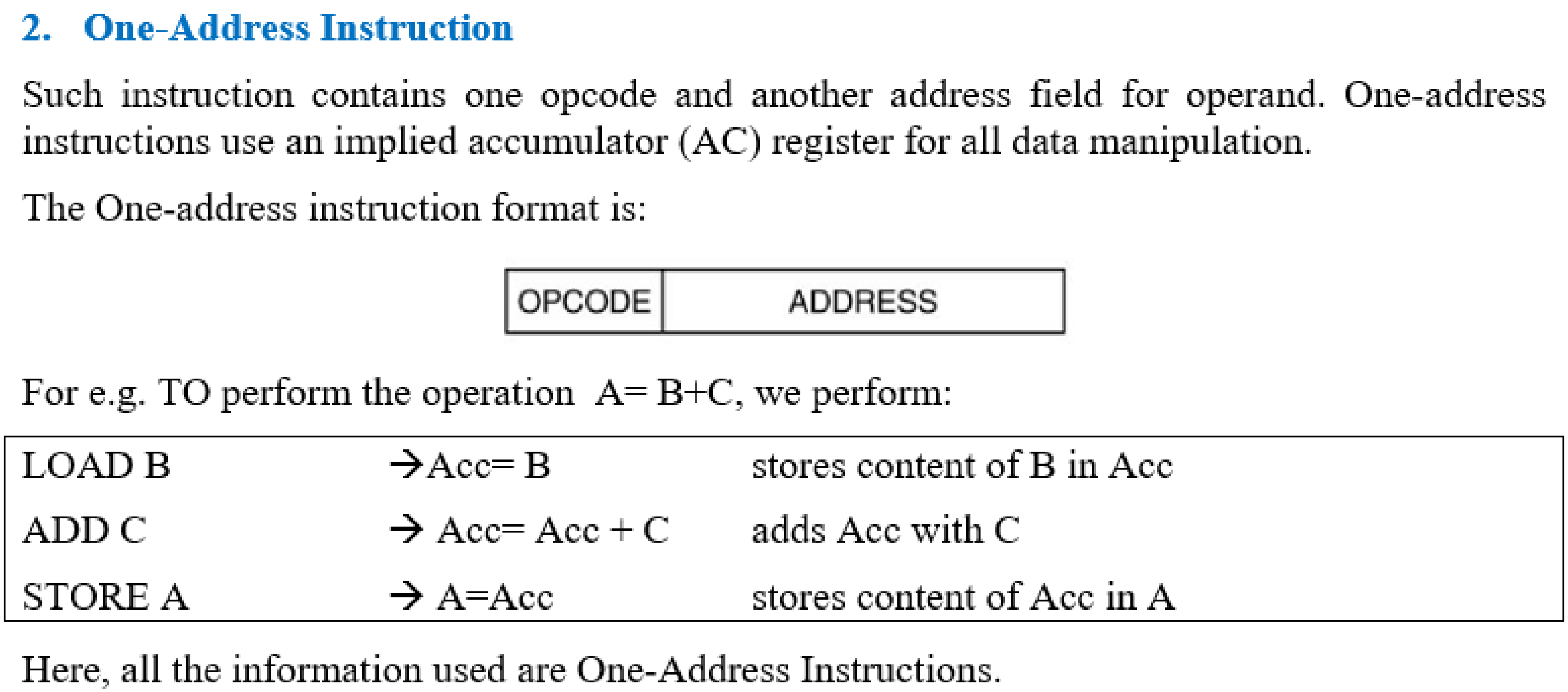
The Zero-address instruction format is:

OPCODE

For eg. To perform the operation A= B + C, we can perform:

PUSH B	puts B in the stack
PUSH C	puts C on TOS
ADD	adds B+C
POP A	A= stack

Here, ADD is zero-address instruction.



2. One-Address Instruction

Such instruction contains one opcode and another address field for operand. One-address instructions use an implied accumulator (AC) register for all data manipulation.

The One-address instruction format is:

OPCODE	ADDRESS

For e.g. TO perform the operation A= B+C, we perform:

LOAD B	→Acc= B	stores content of B in Acc
ADD C	→ Acc= Acc + C	adds Acc with C
STORE A	→ A=Acc	stores content of Acc in A

Here, all the information used are One-Address Instructions.

3. Two-Address Instruction

Such instruction sets have one opcode and two operand fields.

- One is the destination register, and
- The other is the source register or data.

Such instruction formats are the most common in commercial computer.

OPCODE	ADDRESS1	ADDRESS2

E.g.: To add A=B+C

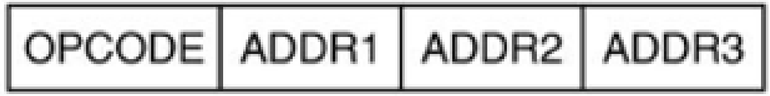
MOVE A, B	→ A=B
ADD A, C	→ add A and C, and store in A

Here, both are Two-Address instructions.

4. Three-Address Instruction

Such instructions have one opcode and three operand fields. The first operand is the destination field and the other two are source fields or data.

The format is:



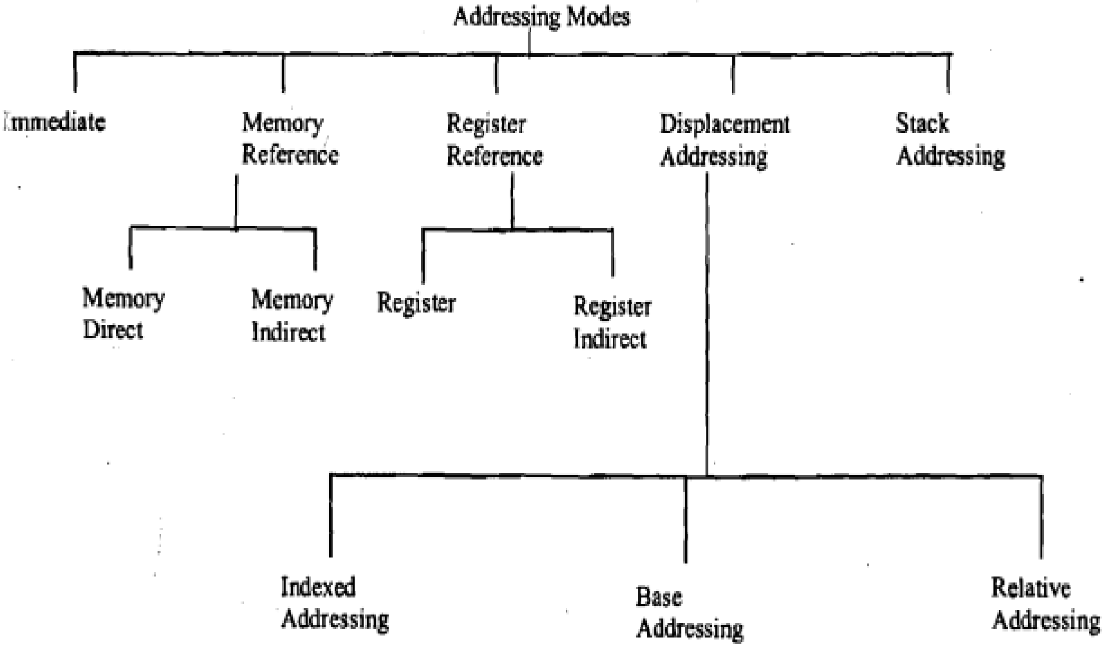
OPCODE	ADDR1	ADDR2	ADDR3

E.g.: To add: A=B+C, we can perform as:

ADD A, B, C → adds B and C, and stores in register A

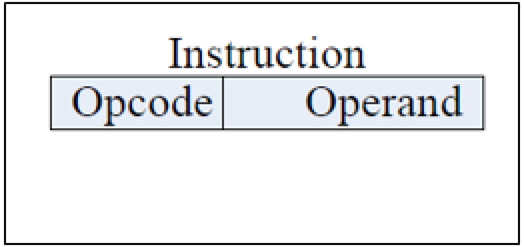
1.10 Addressing Modes

- Addressing mode refers to the way of assigning the address of the operand in memory.



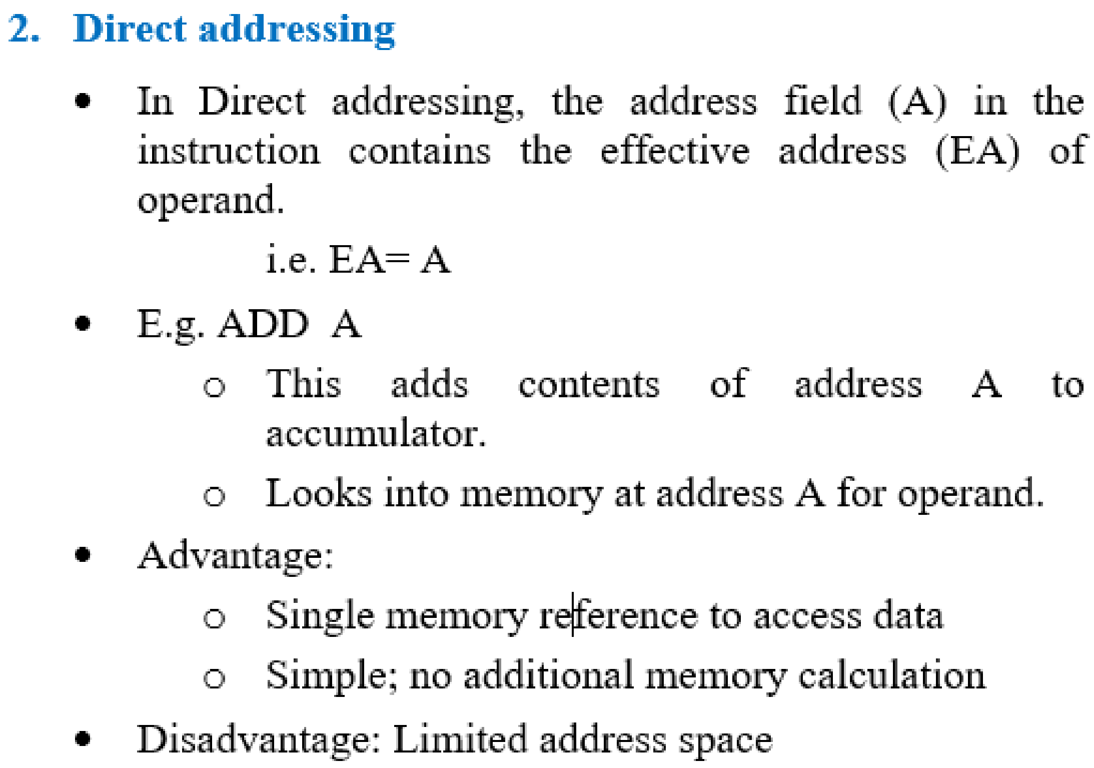
1. Immediate

- This is the simplest addressing mode.
- The operand is the part of instruction.
- Such instruction is mainly used to initialize variable and define constant.
- E.g. LOAD 5 ; This instruction loads 5 to the accumulator.
- Advantage: No memory reference to fetch data. Faster execution.
- Disadvantage: Limited operand magnitude.



Instruction

Opcode	Operand

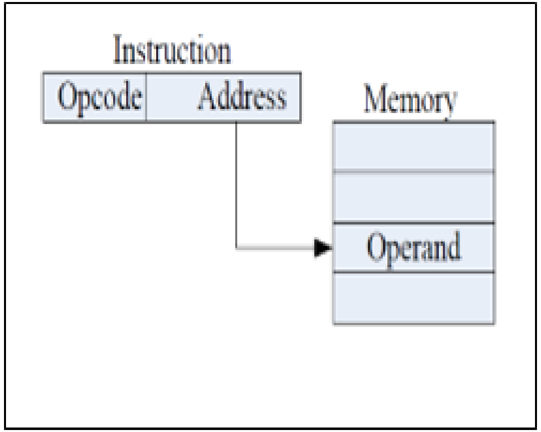


2. Direct addressing

- In Direct addressing, the address field (A) in the instruction contains the effective address (EA) of operand.

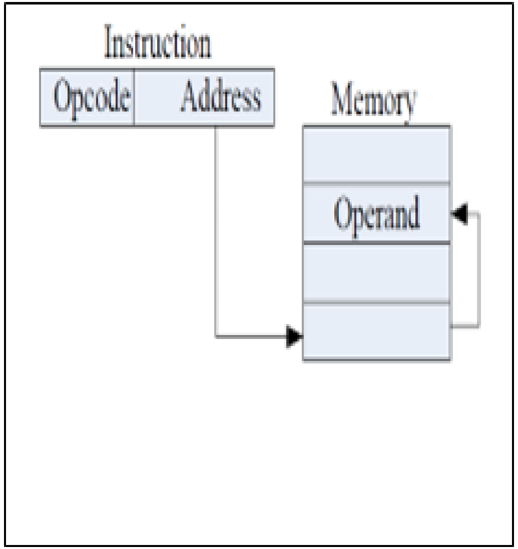
  i.e. EA= A

- E.g. ADD A
  - This adds contents of address A to accumulator.
  - Looks into memory at address A for operand.
- Advantage:
  - Single memory reference to access data
  - Simple; no additional memory calculation
- Disadvantage: Limited address space



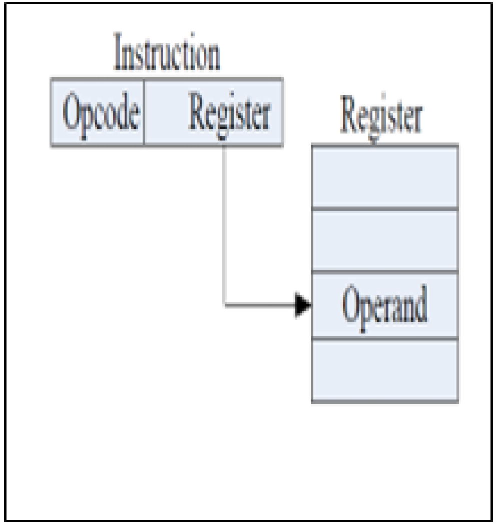
3. Indirect Addressing Mode

- The address field contains the address of word in a memory and that word contains the address of the operand.
- Here, EA =(A)
- E.g.: ADD (A)
  - This adds the content of memory word, pointed by A, to the accumulator.
- Advantage: support large memory space, Eg: EA=(((A)))
- Disadvantage: multiple memory access to find operand, hence slower.



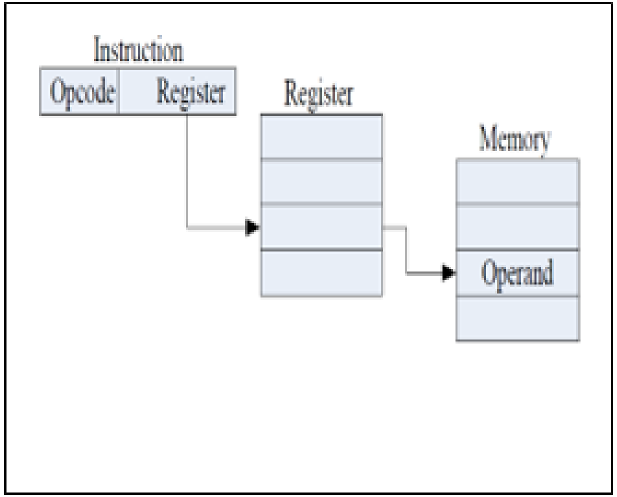
4. Register Addressing Mode

- The address field contains the register name which holds the actual operand.
- Here, EA=R
- E.g.: MOVE R0, R1
  - Contents of R1 moved to R1 register
- Advantage:
  - Only a small field is required in the instruction.
  - No memory reference required, hence faster.
- Disadvantage: Very limited address space.



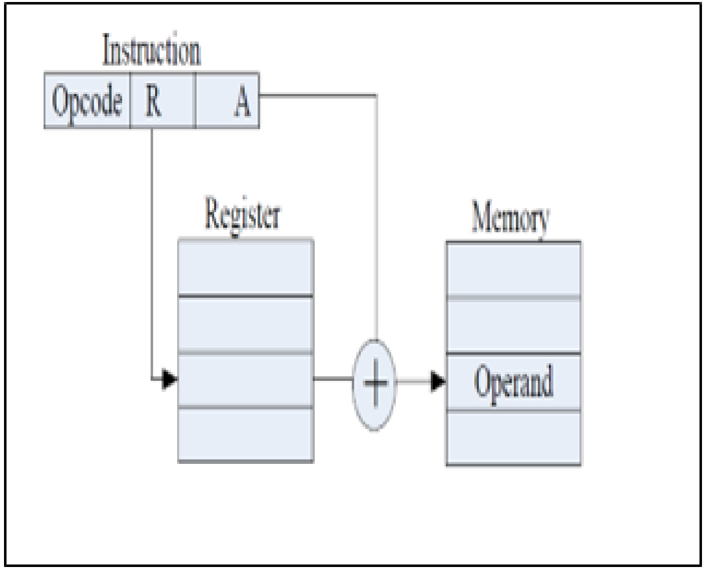
5. Register Indirect Addressing Mode

- The address field contains registers, and that register contains the address of actual operand in the memory.
- Similar to indirect addressing mode.
- Here, EA=(R)
- Advantage: Large address space (2^n)
- Disadvantage: Extra memory reference required.



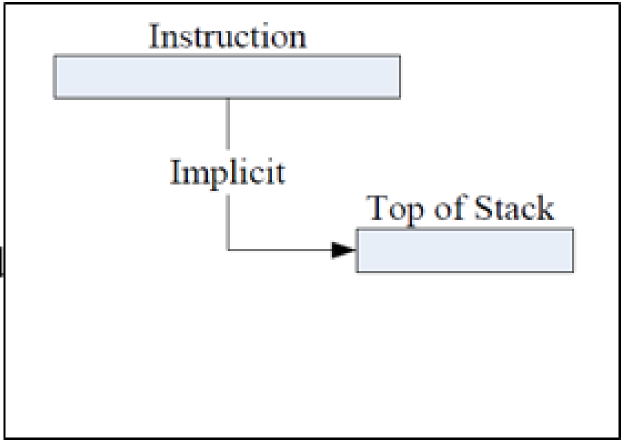
6. Displacement Addressing Mode

- Contains the capabilities of both direct addressing and register indirect addressing.
- EA= A + (R)
- Address field holds two values:
  - A→ base value
  - R→ register that hold displacement
- Advantage: flexible
- Disadvantage: more complex
- Types:
  - Relative addressing
  - Base-register addressing
  - Indexed register addressing (indexing)

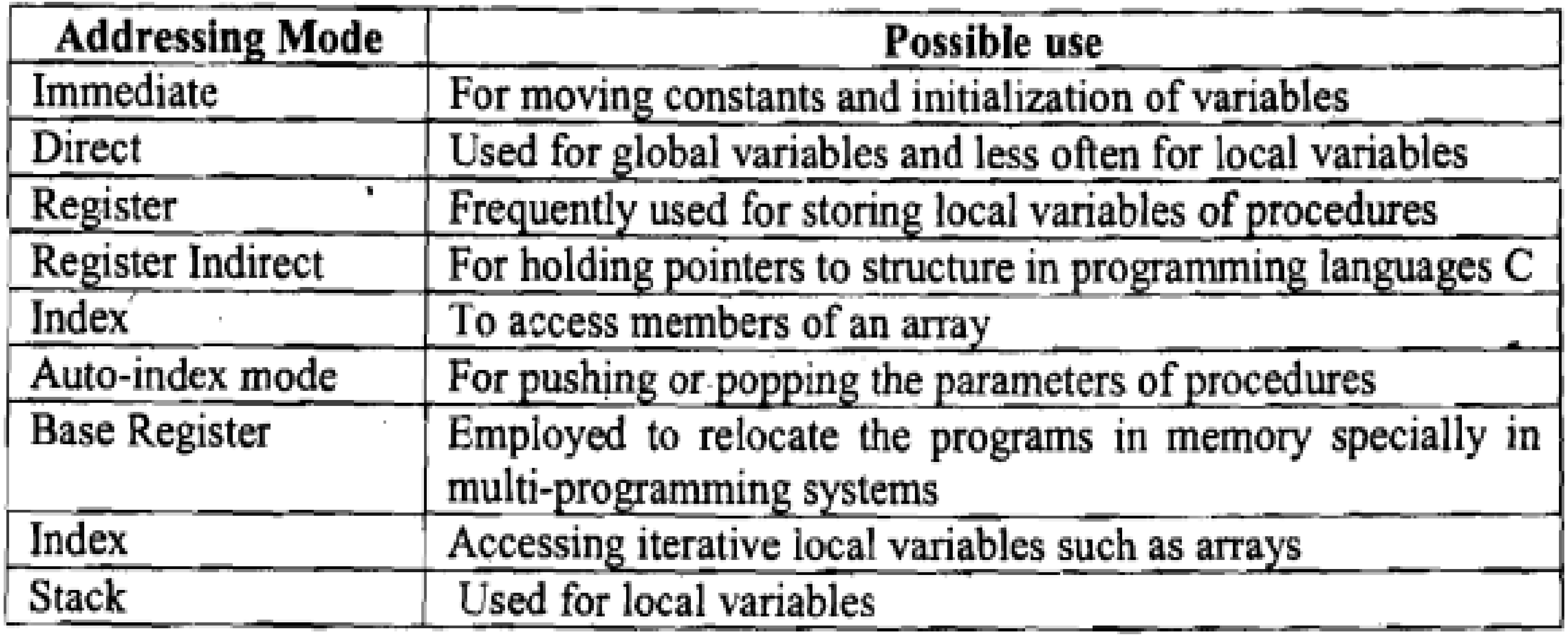


7. Stack Addressing

- Operand is implicitly on top of stack.
- EA= TOS (top of stack)
- E.g.: ADD this adds two items from stack and add
- Advantage: No memory reference
- Disadvantage: Limited applicability



Summary



Addressing Mode	Possible use
Immediate	For moving constants and initialization of variables
Direct	Used for global variables and less often for local variables
Register	Frequently used for storing local variables of procedures
Register Indirect	For holding pointers to structure in programming languages C
Index	To access members of an array
Auto-index mode	For pushing or popping the parameters of procedures
Base Register	Employed to relocate the programs in memory specially in multi-programming systems
Index	Accessing iterative local variables such as arrays
Stack	Used for local variables

1.11 Design Issues for ISA

- **Operation repertoire:** How many and which operations to provide, and how complex operations should be
- **Data types:** The various types of data upon which operations are performed
- **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on
- **Registers:** Number of processor registers that can be referenced by instructions, and their use
- **Addressing:** The mode or modes by which the address of an operand is specified

End of chapter 1