UNIT III

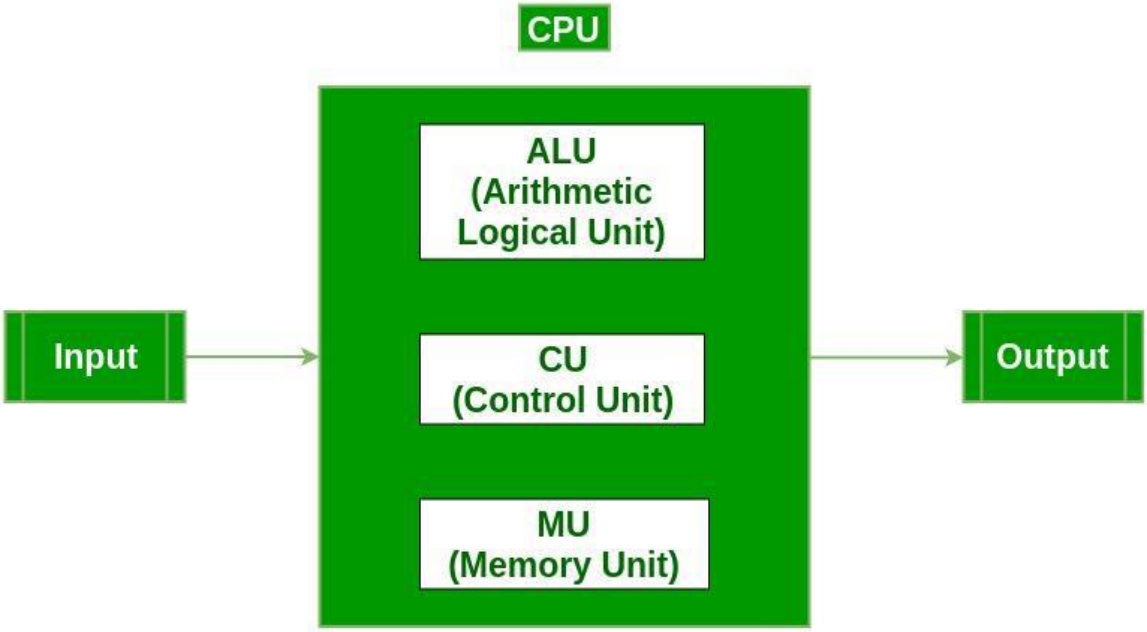
COMPUTER FUNDAMENTALS

FUNCTIONAL UNITS OF A DIGITAL COMPUTER

Computer: A computer is a combination of **hardware and software** resources which integrate together and provides various functionalities to the user. Hardware are the physical components of a computer like the processor, memory devices, monitor, keyboard etc. while software is the set of programs or instructions that are required by the hardware resources to function properly. There are a few basic components that aids the working-cycle of a computer i.e. the Input- Process- Output Cycle and these are called as the functional components of a computer. It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.

Digital Computer: A digital computer can be defined as a programmable machine which reads the binary data passed as instructions, processes this binary data, and displays a calculated digital output. Therefore, Digital computers are those that work on the digital data.

Details of Functional Components of a Digital Computer



 Input Unit : The input unit consists of input devices that are attached to the computer. These devices take input and convert it into binary language that the computer understands. Some of the common input devices are keyboard, mouse, joystick, scanner etc.

 Central Processing Unit (CPU) : Once the information is entered into the computer by the input device, the processor processes it. The CPU is called the brain of the computer because it is the control center of the computer. It first fetches instructions from memory and then interprets them so as to know what is to be done. If required, data is fetched from memory or input device. Thereafter CPU executes or performs the required computation and then either stores the output or displays on the output device. The CPU has three main components which are responsible for different functions – Arithmetic Logic Unit (ALU), Control Unit (CU) and Memory registers

 Arithmetic and Logic Unit (ALU) : The ALU, as its name suggests performs mathematical calculations and takes logical decisions. Arithmetic calculations include addition, subtraction, multiplication and division. Logical decisions involve comparison of two data items to see which one is larger or smaller or equal.

 Control Unit : The Control unit coordinates and controls the data flow in and out of CPU and also controls all the operations of ALU, memory registers and also input/output units. It is also responsible for carrying out all the instructions stored in the program. It decodes the fetched instruction, interprets it and sends control signals to input/output devices until the required operation is done properly by ALU and memory.

 Memory Registers : A register is a temporary unit of memory in the CPU. These are used to store the data which is directly used by the processor. Registers can be of different sizes(16 bit, 32 bit, 64 bit and so on) and each register inside the CPU has

a specific function like storing data, storing an instruction, storing address of a location in memory etc. The user registers can be used by an assembly language programmer for storing operands, intermediate results etc. Accumulator (ACC) is the main register in the ALU and contains one of the operands of an operation to be performed in the ALU.

 Memory **:** Memory attached to the CPU is used for storage of data and instructions and is called internal memory The internal memory is divided into many storage locations, each of which can store data or instructions. Each memory location is of the same size and has an address. With the help of the

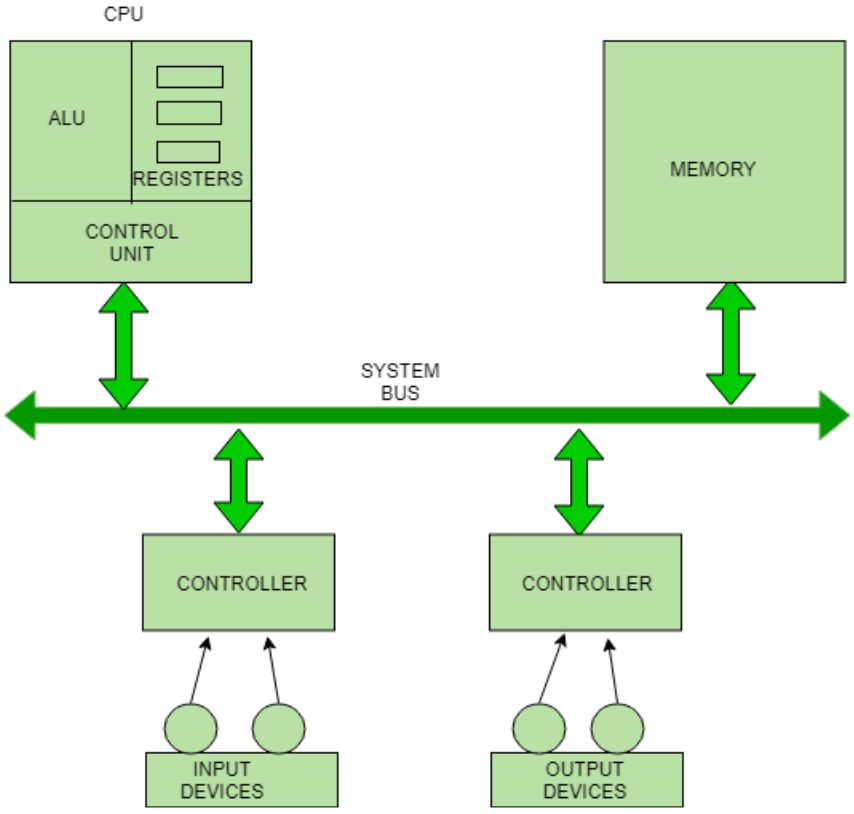
address, the computer can read any memory location easily without having to search the entire memory. when a program is executed, it's data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory. The time of access of data is independent of its location in memory, therefore this memory is also called Random Access memory (RAM). Read this for different types of RAMs

 Output Unit : The output unit consists of output devices that are attached with the computer. It converts the binary data coming from CPU to human understandable form. The common output devices are monitor, printer, plotter etc.

Interconnection between Functional Components

A computer consists of input unit that takes input, a CPU that processes the input and an output unit that produces output. All these devices communicate with each other through a common bus. A bus is a transmission path, made of a set of conducting wires over which data or information in the form of electric signals, is passed from one component to another in a computer. The bus can be of three types – Address bus, Data bus and Control Bus.

Following figure shows the connection of various functional components:



The address bus carries the address location of the data or instruction. The data bus carries data from one component to another and the control bus carries the control signals. The system bus is the common communication path that carries signals to/from CPU, main memory and input/output devices. The input/output devices communicate with the system bus through the controller circuit which helps in managing various input/output devices attached to the computer.

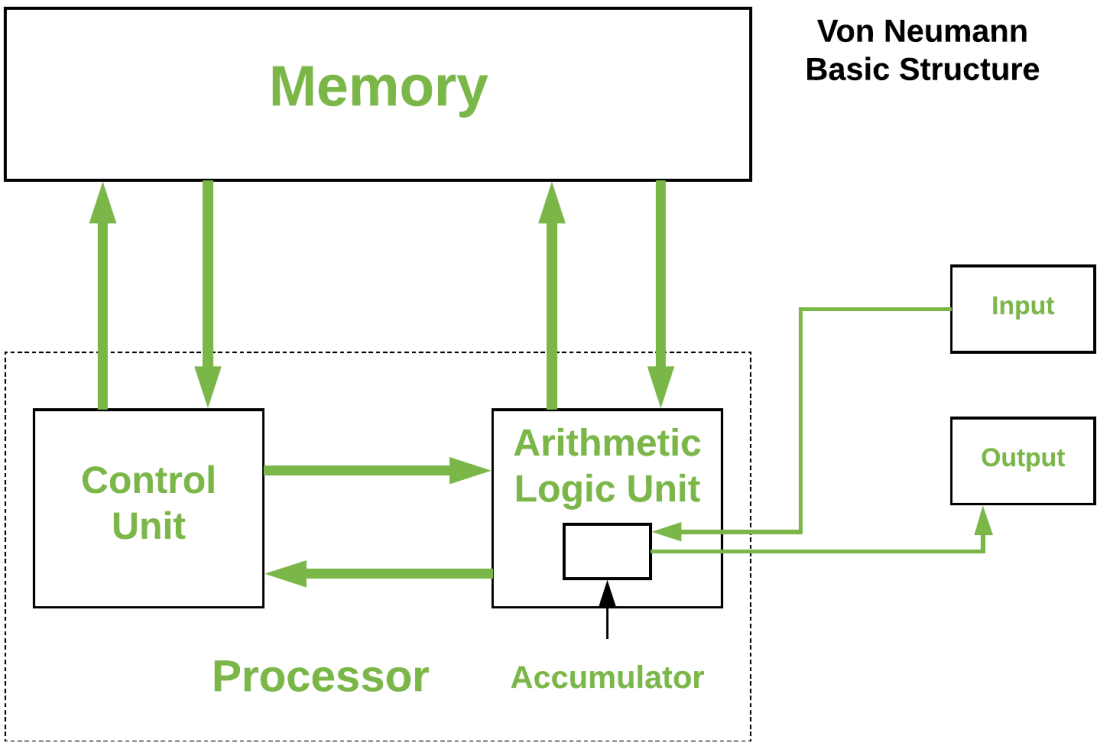
VON-NEUMANN ARCHITECTURE

Historically there have been 2 types of Computers:

1. **Fixed Program Computers –** Their function is very specific and they couldn't be re-programmed, e.g. Calculators.
2. **Stored Program Computers –** These can be programmed to carry out many different tasks, applications are stored on them, hence the name.

The modern computers are based on a stored-program concept introduced by John Von Neumann. In this stored-program concept, programs and data are stored in a separate storage unit called memories and are treated the same. This novel idea meant that a computer built with this architecture would be much easier to reprogram.

The basic structure is like this,



It is also known as **ISA** (Instruction set architecture) computer and is having three basic units:

1. The Central Processing Unit (CPU)
2. The Main Memory Unit
3. The Input/Output Device

Let's consider them in details.

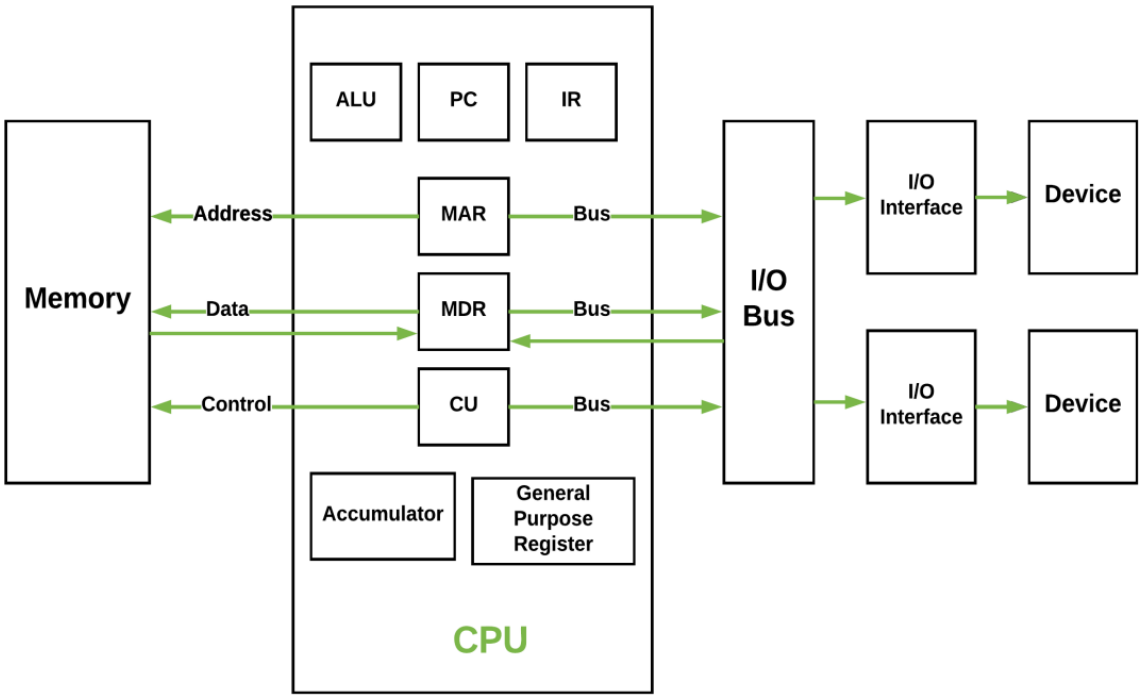
 Control Unit – A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions, and controls how data moves around the system.

 Arithmetic and Logic Unit (ALU) – The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic operations.

Figure – Basic CPU structure, illustrating ALU

 Main Memory Unit (Registers) –

1. **Accumulator:** Stores the results of calculations made by ALU.



Basic CPU structure, illustrating ALU

 Main Memory Unit (Registers) –

1. **Accumulator:** Stores the results of calculations made by ALU.

2. **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).
3. **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from memory or stored into memory.
4. **Memory Data Register (MDR):** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.
5. **Current Instruction Register (CIR):** It stores the most recently fetched instructions while it is waiting to be coded and executed.
6. **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

 Input/Output Devices – Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer. If some results are evaluated by computer and it is stored in the computer, then with the help of output devices, we can present them to the user.

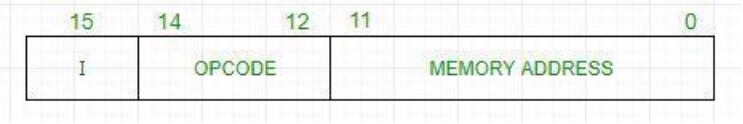
 Buses – Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by the means of Buses. Types:

1. **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
2. **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
3. **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

Von Neumann bottleneck – Whatever we do to enhance performance, we cannot get away from the fact that instructions can only be done one at a time and can only be carried out sequentially. Both of these factors hold back the competence of the CPU. This is commonly referred to as the 'Von Neumann bottleneck'. We can provide a Von Neumann processor with more cache, more RAM, or faster components but if original gains are to be made in CPU performance then an influential inspection needs to take place of CPU configuration.

This architecture is very important and is used in our PCs and even in Super Computers.

Memory Reference – These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12-bit address, 3-bit opcode (other than 111) and 1-bit addressing mode for direct and indirect addressing.



Example – IR register contains = 0001XXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

Hence, DR ← M[AR]

AC ← AC + DR, SC ← 0

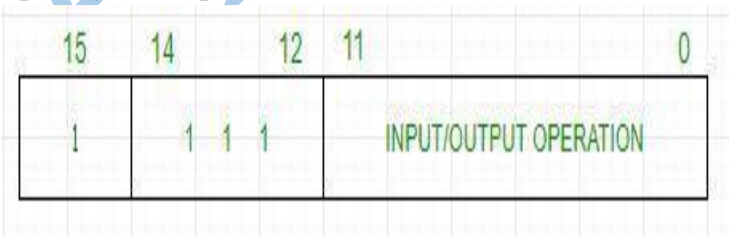
2. **Register Reference –** These instructions perform operations on registers rather than memory addresses. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.



Example – IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.

Hence, AC ← ~AC

3. **Input/Output –** These instructions are for communication between computer and outside environment. The IR(14 – 12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.



Example – IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputing character. Hence, INPUT character from peripheral device.

The set of instructions incorporated in16 bit IR register are:

1. Arithmetic, logical and shift instructions (and, add, complement, circulate left, right, etc)

2. To move information to and from memory (store the accumulator, load the accumulator)
3. Program control instructions with status conditions (branch, skip)
4. Input output instructions (input character, output character)

Symbol	Hexadecimal Code		Description
AND	0xxx	8xxx	And memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store AC content in memory
BUN	4xxx	Cxxx	Branch Unconditionally
BSA	5xxx	Dxxx	Branch and Save Return Address
ISZ	6xxx	Exxx	Increment and skip if 0
CLA	7800		Clear AC
CLE	7400		Clear E(overflow bit)
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC > 0
SNA	7008		Skip next instruction if AC < 0
SZA	7004		Skip next instruction if AC = 0
SZE	7002		Skip next instruction if E = 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt On
IOF	F040		Interrupt Off

INSTRUCTION FORMATS (ZERO, ONE, TWO AND THREE ADDRESS INSTRUCTION)

A computer performs a task based on the instruction provided. Instruction in computers comprises groups called fields. These fields contain different information as for computers everything is in 0 and 1 so each field has different significance based on which a CPU decides what to perform. The most common fields are:

 Operation field specifies the operation to be performed like addition.

 Address field which contains the location of the operand, i.e., register or memory location.

 Mode field which specifies how operand is to be founded.

Instruction is of variable length depending upon the number of addresses it contains. Generally, CPU organization is of three types based on the number of address fields:

1. Single Accumulator organization
2. General register organization
3. Stack organization

In the first organization, the operation is done involving a special register called the accumulator. In second on multiple registers are used for the computation purpose. In the third organization the work on stack basis operation due to which it does not contain any address field. Only a single organization doesn't need to be applied, a blend of various organizations is mostly what we see generally.

Based on the number of address, instructions are classified as:

Note that we will use X = (A+B)*(C+D) expression to showcase the procedure.

1. **Zero Address Instructions –**



A stack-based computer does not use the address field in the instruction. To evaluate an expression first it is converted to reverse Polish Notation i.e. Postfix Notation.

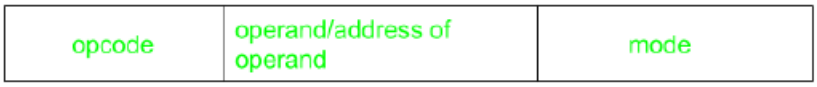
Expression: X = (A+B)*(C+D)

Postfixed : X = AB+CD+*

TOP means top of stack

M[X] is any memory location PUSH	A	TOP = A
PUSH	B	TOP = B

ADD		TOP = A+B
PUSH	C	TOP = C
PUSH	D	TOP = D
ADD		TOP = C+D
MUL		TOP = (C+D)*(A+B)
POP	X	M[X] = TOP



Expression: X = (A+B)*(C+D)
AC is accumulator
M[] is any memory location
M[T] is temporary location

LOAD	A	AC = M[A]
ADD	B	AC = AC + M[B]
STORE	T	M[T] = AC
LOAD	C	AC = M[C]

3.Two Address Instructions – This is common in commercial computers. Here two addresses can be specified in the instruction. Unlike earlier in one address instruction, the result was stored in the accumulator, here the result can be stored at different locations rather than just accumulators, but require more number of bit to represent address



Here destination address can also contain operand.
Expression: X = (A+B)*(C+D)
R1, R2 are registers
M[] is any memory location

MOV	R1, A	R1 = M[A]
ADD	R1, B	R1 = R1 + M[B]
MOV	R2, C	R2 = C
ADD	R2, D	R2 = R2 + D
MUL	R1, R2	R1 = R1 * R2

MOV X, R1 M[X] = R1



Expression: X = (A+B)*(C+D)
R1, R2 are registers

M[] is any memory location	R1, A, B	R1 = M[A] + M[B]
ADD		
ADD	R2, C, D	R2 = M[C] + M[D]
MUL	X, R1, R2	M[X] = R1 * R2

INSTRUCTION SET ARCHITECTURE

In this article, we look at what an *Instruction Set Architecture (ISA)* is and what is the difference between an **'ISA'** and *Microarchitecture*. An **ISA** is defined as the design of a computer from the *Programmer's Perspective*.

This basically means that an **ISA** describes the **design of a Computer** in terms of the **basic operations** it must support. The ISA is not concerned with the implementation-specific details of a computer. It is only concerned with the set or collection of basic operations the computer must support. For example, the AMD Athlon and the Core 2 Duo processors have entirely different implementations but they support more or less the same set of basic operations as defined in the x86 Instruction Set.

Let us try to understand the Objectives of an ISA by taking the example of the **MIPS ISA**. MIPS is one of the most widely used ISAs in education due to its simplicity.

1. The ISA defines the **types of instructions** to be supported by the processor. Based on the type of operations they perform MIPS Instructions are classified into 3 types:

- **Arithmetic/Logic Instructions:** These Instructions perform various Arithmetic & Logical operations on one or more operands.
- **Data Transfer Instructions:** These instructions are responsible for the transfer of instructions from memory to the processor registers and vice versa.
- **Branch and Jump Instructions:** These instructions are responsible for breaking the sequential flow of instructions and jumping to instructions at various other locations, this is necessary for the implementation of *functions* and *conditional statements*.

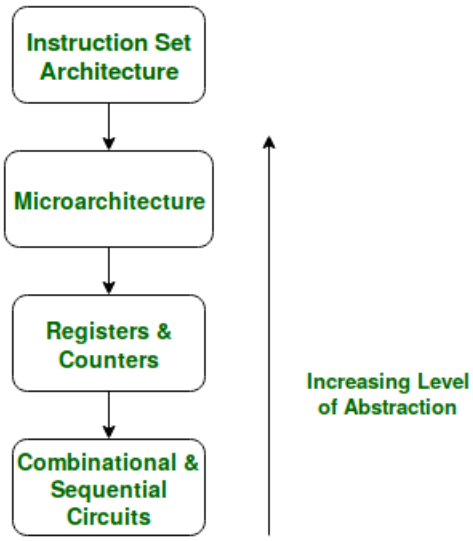
2. The ISA defines the **maximum length** of each type of instruction.

Since the MIPS is a 32 bit ISA, each instruction must be accommodated within 32 bits.

3. The ISA defines the **Instruction Format** of each type of instruction. The **Instruction Format** determines how the entire instruction is encoded within 32 bits There are 3 types of Instruction Formats in the MIPS ISA:

- R-Instruction Format
- I-Instruction Format
- J-Instruction Format

If we look at the Abstraction Hierarchy



We note that the **Microarchitectural** level lies just below the **ISA** level and hence is concerned with the implementation of the basic operations to be supported by the Computer as defined by the **ISA**. Therefore we can say that the AMD Athlon and Core 2 Duo processors are based on the same ISA but have different microarchitectures with different performance and efficiencies.

Now one may ask the need to distinguish between **Microarchitecture** and **ISA**? The answer to this lies in the need to standardize and maintain the compatibility of programs across different hardware implementations based on the same **ISA**. Making different machines compatible with the same set of basic instructions (The ISA) allows the same program to run smoothly on many different machines thereby making it easier for the programmers to document and maintain code for many different machines simultaneously and efficiently.

This Flexibility is the reason we first define an ISA and then design different microarchitectures complying with this ISA for implementing the machine. The design of a lower-level ISA is one of the major tasks in the study of Computer Architecture.

Instruction Set Architecture

The ISA is responsible for defining the set of instructions to be supported by the processor. For example, some of the instructions defined by the ARMv7 ISA are given below. The Branch of **Computer Architecture** is more inclined towards the Analysis and Design of Instruction Set Architecture. For Example, Intel developed the *x86* architecture, ARM developed the *ARM* architecture, & AMD developed the *amd64* architecture. The RISC-V ISA developed by UC Berkeley is an example of an Open Source ISA.

Microarchitecture

On the other hand, the Branch of **Computer Organization** is concerned with the implementation of a particular ISA deals with various hardware implementation techniques, i.e. is the Microarchitecture level. For Example, ARM licenses other companies like Qualcomm, Apple for using ARM ISA, but each of these companies have their own implementations of this ISA thereby making them different in performance and power efficiency. The *Krait* cores developed by Qualcomm have a different microarchitecture and the Apple A-series processors have a different microarchitecture. The Microarchitecture is more concerned with the lower level implementation of how the instructions are going to be executed and deals with concepts like Instruction Pipelining, Branch Prediction, Out of Order Execution.

INSTRUCTIONS & INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, suchas adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen.

A computer must have instructions capable of performing 4 types of operations: 1) Data transfers between the memory and the registers (MOV, PUSH, POP, XCHG). 2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT). 3) Program sequencing and control(CALL.RET, LOOP, INT). 4) I/0 transfers (IN, OUT).

REGISTER TRANSFER NOTATION (RTN) Here we describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or

registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names. • The possible locations in which transfer of information occurs are: 1) Memory-location 2) Processor register & 3) Registers in I/O device.

Location	Hardware Binary Address	Example	Description
Memory	LOC, PLACE, NUM	R1 ← [LOC]	Contents of memory-location LOC are transferred into register R1.
Processor	R0, R1 ,R2	[R3] ← [R1]+[R2]	Add the contents of register R1 &R2 and places their sum into R3.
I/O Registers	DATAIN, DATAOUT	R1 ← DATAIN	Contents of I/O register DATAIN are transferred into register R1.

ASSEMBLY LANGUAGE NOTATION • To represent machine instructions and programs, assembly language format is used.

Assembly Language Format	Description
Move LOC, R1	Transfer data from memory-location LOC to register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.
Add R1, R2, R3	Add the contents of registers R1 and R2, and places their sum into register R3.

BASIC INSTRUCTION TYPES

Instruction Type	Syntax	Example	Description	Instructions for Operation C<-[A]+[B]
Three Address	Opcode Source1,Source2,Destination	Add A,B,C	Add the contents of memory-locations A & B. Then, place the result into location C.	
Two Address	Opcode Source, Destination	Add A,B	Add the contents of memory-locations A & B. Then, place the result into location B, replacing the original contents of this location. Operand B is both a source and a destination.	Move B, C Add A, C
One Address	Opcode Source/Destination	Load A	Copy contents of memory-location A into accumulator.	Load A Add B Store C
		Add B	Add contents of memory-location B to contents of accumulator register & place sum back into accumulator.	
		Store C	Copy the contents of the accumulator into location C.	
Zero Address	Opcode [no Source/Destination]	Push	Locations of all operands are defined implicitly. The operands are stored in a pushdown stack.	Not possible

INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING • The program is executed as follows: 1) Initially, the address of the first instruction is loaded into PC (Figure 2.8). 2) Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called Straight-Line sequencing. 3) During the

execution of each instruction, PC is incremented by 4 to point to next instruction. • There are 2 phases for Instruction Execution: 1) Fetch Phase: The instruction is fetched from the memory-location and placed in the IR. 2) Execute Phase: The contents of IR is examined to determine which operation is to be performed. The specified-operation is then performed by the processor.

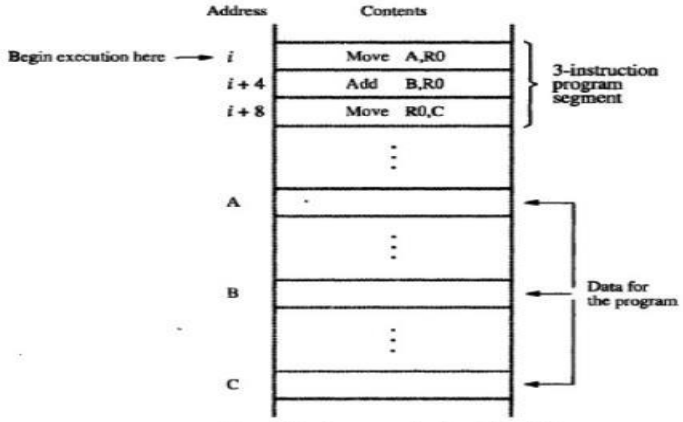


Figure 2.8 A program for C ← [A] + [B].

Program Explanation • Consider the program for adding a list of n numbers (Figure 2.9). • The Address of the memory-locations containing the n numbers are symbolically given as NUM1, NUM2…..NUMn. • Separate Add instruction is used to add each number to the contents of register R0. • After all the numbers have been added, the result is placed in memory-location SUM.

APEC

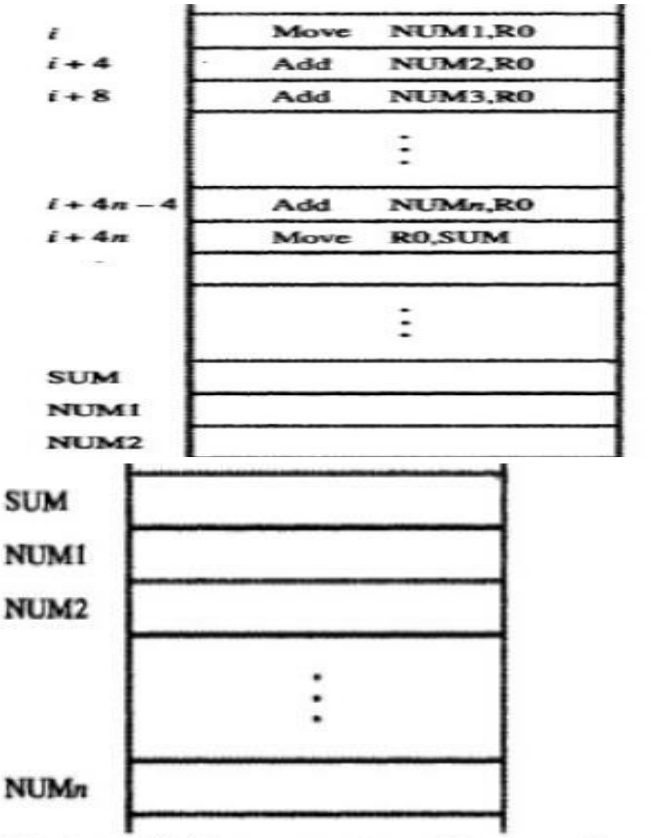


Figure 2.9 A straight-line program for adding *n* numbers.

ADDRESSING MODES

The different ways in which the location of an operand is specified in an instruction are referred to as

1)Immediate Mode • The operand is given explicitly in the instruction. • For example, the instruction Move #200, R0 ;Place the value 200 in register R0. • Clearly, the immediate mode is only used to specify the value of a source-operand.

2)Register Mode • The operand is the contents of a register. • The name (or address) of the register is given in the instruction. • Registers are used as temporary storage locations where the data in a register are accessed. • For example, the instruction Move R1, R2 ;Copy content of register R1 into register R2.

3)Absolute (Direct) Mode • The operand is in a memory-location. • The address of memory-location is given explicitly in the instruction. • The absolute mode can represent global variables in the program. • For example, the instruction Move LOC, R2 ;Copy content of memory-location LOC into register R2.

4)INDIRECTION AND POINTERS • Instruction does not give the operand or its address explicitly. • Instead, the instruction provides information from which

the new address of the operand can be determined. • This address is called Effective Address (EA) of the operand.

Indirect Mode • The EA of the operand is the contents of a register(or memory-location). • The register (or memory-location) that contains the address of an operand is called a Pointer. • We denote the indirection by → name of the register or → new address given in the instruction. E.g: Add (R1), R0;The operand is in memory. Register R1 gives the effective-address (B) of the operand. The data is read from location B and added to contents of register R0.

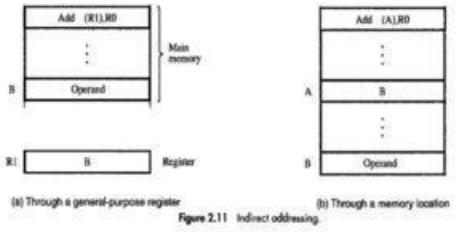


Figure 2.11 Indirect addressing.

• To execute the Add instruction in fig 2.11 (a), the processor uses the value which is in register R1, as the EA of the operand. • It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0.

• Indirect addressing through a memory-location is also possible as shown in fig 2.11(b). In this case, the processor first reads the contents of memory-location A, then requests a second read operation using the value B as an address to obtain the operand

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

Program Explanation • In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.

• The initialization-section of the program loads the counter-value n from memory-location N into R1 and uses the immediate addressing-mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.

• The first two instructions in the loop implement the unspecified instruction block starting at LOOP.

• The first time through the loop, the instruction Add (R2), R0 fetches the operand at location NUM1 and adds it to R0.
• The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

5)INDEXING AND ARRAYS

• A different kind of flexibility for accessing operands is useful in dealing with lists and arrays. Index mode
• The operation is indicated as X(Ri) where X=the constant value which defines an offset(also called a displacement). Ri=the name of the index register which contains address of a new location.
• The effective-address of the operand is given by EA=X+[Ri]
• The contents of the index-register are not changed in the process of generating the effectiveaddress.
• The constant X may be given either → as an explicit number or → as a symbolic-name representing a numerical value.

6)Base with Index Mode

• Another version of the Index mode uses 2 registers which can be denoted as(Ri, Rj)
• Here, a second register may be used to contain the offset X.
• The second register is usually called the base register.
• The effective-address of the operand is given by EA=[Ri]+[Rj]
• This form of indexed addressing provides more flexibility in accessing operands because both components of the effective-address can be changed.

7)Base with Index & Offset Mode

• Another version of the Index mode uses 2 registers plus a constant, which can be denoted as X(Ri, Rj)
• The effective-address of the operand is given by EA=X+[Ri]+[Rj]
•This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

8)RELATIVE MODE

• This is similar to index-mode with one difference: The effective-address is determined using the PC in place of the general-purpose register Ri.
• The operation is indicated as X(PC).

• X(PC) denotes an effective-address of the operand which is X locations above or below the current contents of PC.
• Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
• This mode is used commonly in conditional branch instructions.
• An instruction such as Branch > 0 LOOP; Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

9) Auto Increment Mode

Effective-address of operand is contents of a register specified in the instruction (Fig: 2.16). After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list. Implicitly, the increment amount is 1. This mode is denoted as (Ri)+ ;where RI=pointer-register.

10) Auto Decrement Mode The contents of a register specified in the instruction are first automatically decremented and are then used as the effective-address of the operand. This mode is denoted as -(Ri); where Ri=pointer-register. These 2 modes can be used together to implement an important data structure called a stack.

ENCODING OF MACHINE INSTRUCTIONS

• To be executed in a processor, an instruction must be encoded in a binary-pattern. Such encoded instructions are referred to as Machine Instructions. • The instructions that use symbolic-names and acronyms are called assembly language instructions. • We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers. • Let us examine some typical cases.

The instruction Add R1, R2 ; Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing-mode is used for each operand.

The instruction Move 24(R0), R5; Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24.

• In all these examples, the instructions can be encoded in a 32-bit word (Fig 2.39). • The OP code for given instruction refers to type of operation that is to be performed. • Source and destination field refers to source and destination operand

respectively. • The "Other info" field allows us to specify the additional information that may be needed such as an index value or an immediate operand. • Using multiple words, we can implement complex instructions, closely resembling operations in high level programming languages. The term complex instruction set computers (CISC) refers to processors that use • CISC approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used. • In RISC (reduced instruction set computers), any instruction occupies only one word.

• The RISC approach introduced other restrictions such as that all manipulation of data must be done on operands that are already in registers. Ex: Add R1,R2,R3

• In RISC type machine, the memory references are limited to only Load/Store operations.Ro

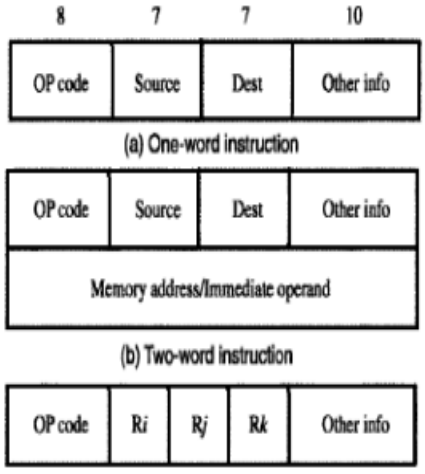


Figure 2.39 Encoding instructions into 32-bit words.

Rotate operations:

• In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C. • To preserve all bits, a set of rotate instructions can be used.

• They move the bits that are shifted out of one end of the operand back into the other end.

• Two versions of both the left and right rotate instructions are usually provided. In one version, the bits of the operand is simply rotated. In the other version, the rotation includes the C flag.

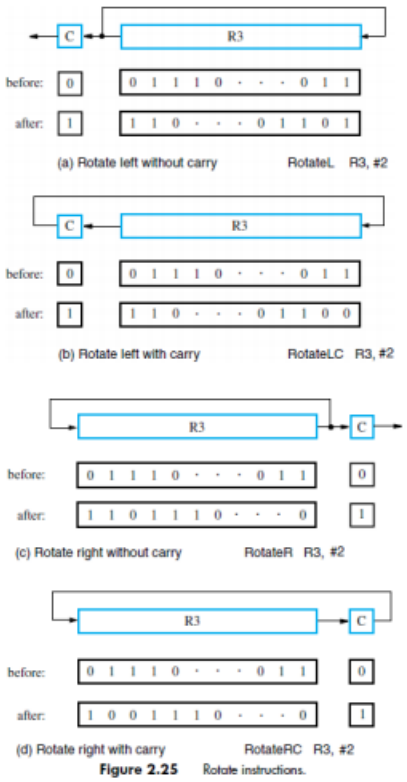


Figure 2.25 Rotate instructions.

SHIFT AND ROTATE INSTRUCTIONS

• There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions.

• The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.

• For general operands, we use a logical shift. For a number, we use an arithmetic shift, which preserves the sign of the number.

LOGICAL SHIFTS

• Two logical shift instructions are

1) Shifting left (LShiftL) &

2) Shifting right (LShiftR).

These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction

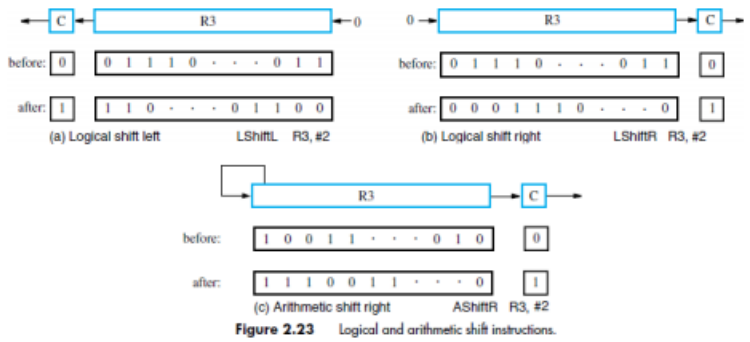


Figure 2.23 Logical and arithmetic shift instructions.

Move	#LOC,R0	R0 points to data.
MoveByte	(R0)+,R1	Load first byte into R1.
LShiftL	#4,R1	Shift left by 4 bit positions.
MoveByte	(R0),R2	Load second byte into R2.
And	#$F,R2	Eliminate high-order bits.
Or	R1,R2	Concatenate the BCD digits.
MoveByte	R2,PACKED	Store the result.

Figure 2.31 A routine that packs two BCD digits.

INTERACTION BETWEEN ASSEMBLY LANGUAGE AND HIGH LEVEL LANGUAGE

High level languages such as Python, C, C++, Java, C# allows the programmer to write the alpha numeric codes.

The compiler converts the alpha numeric codes into assembly code. Assembly language codes example:

ADD R1, R2, R3 ADDi R1, R2, 20 LOAD R1, 20(R2) STORE R1, 20(R2)

The assembler intern converts the assembly language code into binary code (machine code).

Parameters	**Assembly Language**	**High-Level Language**
Conversion	The assembly language requires an assembler for the process of conversion.	A high-level language requires an interpreter/ compiler for the process of conversion.
Process of Conversion	We perform the conversion of an assembly language into a machine language.	We perform the conversion of a high-level language into an assembly language and then into a machine-level language for the computer.
Machine Dependency	The assembly language is a machine- dependent type of language.	A high-level language is a machine- independent type of language.
Codes	It makes use of the mnemonic codes for operation.	It makes use of the English statements for operation.

Operation of Lower Level	It provides support for various low-level operations.	It does not provide any support for low- level languages.
Access to Hardware Component	Accessing the hardware component is very easy in this case.	Accessing the hardware component is very difficult in this case.
Compactness in Code	The code is more compact in this case.	No code compactness is present in this case.
Type of Processor	The program that we write for one processor in an assembly language will not run on any other processor type. It means that it is processor-dependent.	This language is processor-independent. It means that the programs that we write using high-level languages can easily run on any processor independent of its type.
Accuracy	It has better accuracy.	Accuracy is much lesser in this case.
Performance	An assembly language performs better than any high-level language, in general.	The performance is comparatively not so good.
Length of Executable Code	It is shorter in assembly language.	It is larger in a high-level language.
Time Taken in Code Execution	Execution of code takes less time in this case because the code is not very large.	It takes up more time for execution because it needs to execute a large code.
Efficiency	It is way more efficient because of the shorter executable codes.	It is comparatively less efficient because the executable codes are comparatively longer in length.
Reading of Pointers	We can do that directly at a physical address in the case of an assembly language.	It is not possible to do so in the case of a high-level language.
Extra Instructions	We don‟t need that in the case of an assembly language.	This language must give some extra instructions for running any code on the computer.

Ease of Understanding	It is very difficult to debug and understand the code of an assembly language.	It is very easy to debug and understand the code of an assembly language

APEC