**INSTRUCTION SET PRINCIPLES**

[**https://passlab.github.io/CSCE513/notes/lecture03\_ISA\_Principles.pdf**](https://passlab.github.io/CSCE513/notes/lecture03_ISA_Principles.pdf)

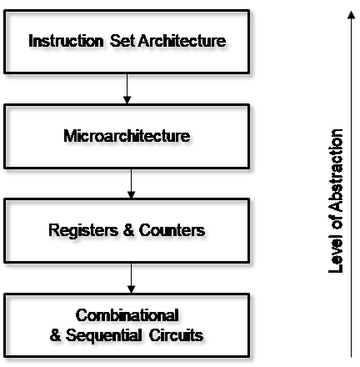
[**https://www.embedded.com/a-quick-introduction-to-instruction-set-architecture-and-extensibility/**](https://www.embedded.com/a-quick-introduction-to-instruction-set-architecture-and-extensibility/)

**Introduction**

The list of specific instructions supported by the CPU is termed as its **instruction set.**

In short, an instruction set is the link between a computer’s software and hardware.

The ISA defines the types of instructions to be supported by the processor.



An instruction in the computer should specify the following:

* The task or operation to be carried out by the processor. This termed as **opcode.**
* The address(s) in memory of the operands(s) on which the data processing is to be performed.
* The address in the memory that may store the results of the data processing operation performed by the instruction.
* The address in the memory for the next instruction, to be fetched and executed.

An instruction is divided into a number of fields and is represented as a sequence of bits. These fields include:

* The Operation code (Opcode) field which specifies the operation to be performed.
* The Address field which contains the location of the operand, i.e., register or memory location.

|  |  |
| --- | --- |
| Opcode | Operand address |

Eg: ADD R1,R0.

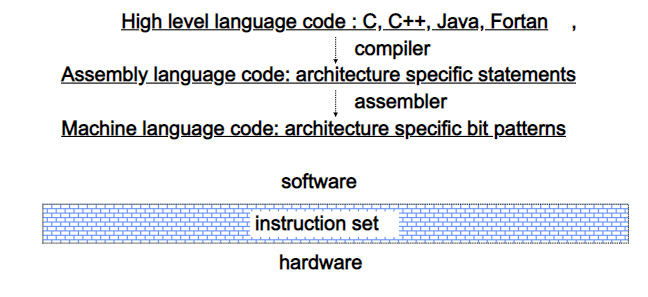
ADD is the opcode and R1,R0 are the address field

An instruction is first read into an Instruction Register(IR),then it decoded by the CPU and process it.

**The instruction set architecture (ISA)**

The instruction set architecture (ISA) specifies what the processor is capable of doing and, how it gets accomplished. So the instruction set architecture is basically the interface between your hardware and the software. The only way that you can interact with the hardware is the instruction set of the processor. To command the computer, you need to speak its language and the instructions are the words of a computer’s language and the instruction set is basically its vocabulary.

The only way that you can interact with the hardware is the instruction set of the processor.  ISA is the portion of the machine which is visible to either the assembly language programmer or a compiler writer or an application programmer. It is the only interface that you have, because the instruction set architecture is the specification of what the computer can do and the machine has to be fabricated in such a way that it will execute whatever has been specified in your ISA. The only way that you can talk to your machine is through the ISA. This gives you an idea of the interface between the hardware and software.



Let us assume you have a high-level program written in C which is independent of the architecture on which you want to work. This high-level program has to be translated into an assembly language program which is specific to a particular architecture. Let us say you find that this consists of a number of instructions like LOAD, STORE, ADD, etc., where, whatever you had written in terms of high-level language now have been translated into a set of instructions which are specific to the specific architecture. These are all English like and this is not understandable to the processor because the processor is after all made up of digital components which can understand only zeros and ones. So this assembly language will have to be finely translated into machine language, object code which consists of zeros and ones. So the translation from your high-level language to your assembly language and the binary code will have to be done with the compiler and the assembler.

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

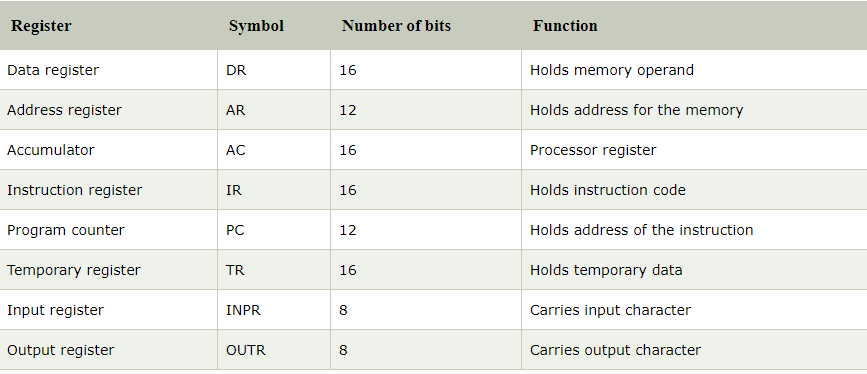
* + **Data manipulation Instructions:**  
    Data manipulation instructions are those instruction that perform Arithmetic, shift & Logical operations to manipulate data. Eg: add, mul, and, or, not etc
  + **Data Transfer Instructions:**  
    These instructions are responsible for the transfer of instructions from memory to the processor registers and vice versa. Eg: mov, load, store, push, pop etc
  + **Program control 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*. Eg: branch, jump, skip call etc

**Computer Registers**

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers. A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

Following is the list of some of the most common registers used in a basic computer:



**Classifying Instruction set Architecture**

[**https://www.youtube.com/watch?v=BYB65lV5yhk**](https://www.youtube.com/watch?v=BYB65lV5yhk)

ISAs differ based on the internal storage in a processor. The major choices are **a stack, an accumulator, or a set of registers.** Operands may be named explicitly or implicitly: The operands in a stack architecture are implicitly on the top of the stack, and in an accumulator architecture one operand is implicitly the accumulator. The general-purpose register architectures have only explicit operands—either registers or memory locations.

* **Stack organization**, where the operands are put into the stack and the operations are carried out on the top of the stack. The operands are implicitly specified here.
* S**ingle accumulator organization**, which names one of the general purpose registers as the accumulator and uses it to necessarily store one of the operands. This indicates that one of the operands is implied to be in the accumulator and it is enough if the other operand is specified along with the instruction.

Eg: ADD X , AC 🡨 AC + M[X] means add the contents at memory location X with the content of the accumulator

* **General purpose register organization,**which specifies all the operands explicitly. There are 2 major reasons for the emergence of GPR computers. First, registers are faster than memory. Second, registers are more efficient for a compiler to use than other forms of internal storage. Depending on whether the operands are available in memory or registers, it can be further classified as
* **Register** **–** **register**, where registers are used for storing operands. Such architectures are in fact also called **load** **–** **store** architectures, as only load and store instructions can have memory operands.

**[Load- move data from Memory to Register**

**Store- move data from Register to Memory**]

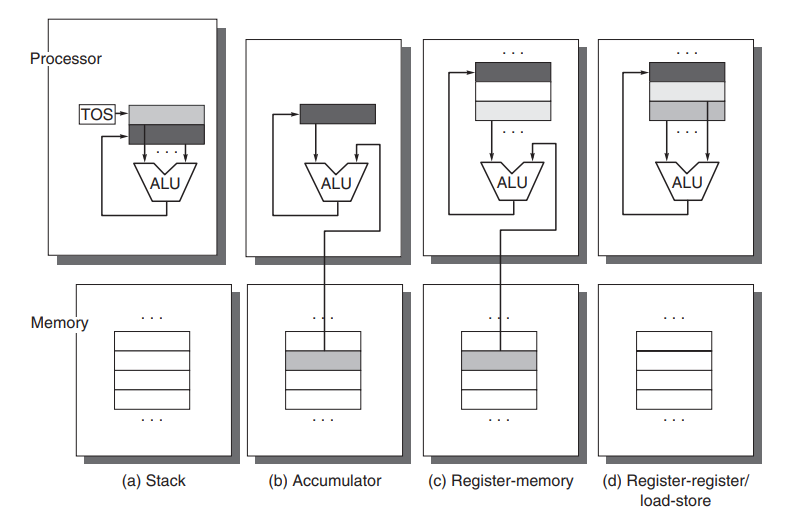
In the register – register ISA, both operands will have to move to two registers and the ADD instruction will only work on registers

* **Register** **–** **memory**, where one operand is in a register and the other one in memory.

In the register --memory ISA, one operand has to be moved into any register and the other one can be a memory operand.

* **Memory** **–** **memory**, where all the operands are specified as memory operands.

Following shows a block diagram of such architectures, and how the code sequence C = A + B would typically appear in these three classes of instruction sets.



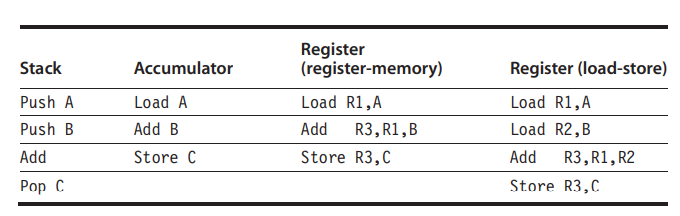
The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result.

In (a), a Top Of Stack register (TOS) points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit.

In (b), the Accumulator is both an implicit input operand and a result.

In (c), one input operand is a register, one is in memory, and the result goes to a register.

All operands are registers in (d) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).



In the case of an ***accumulator-based*** ISA, where we assume that one of the general-purpose registers is being designated as an accumulator and one of the operands will always be available in the accumulator, you have to initially load one operand into the accumulator and the ADD instruction will only specify the operand’s address.

In the ***GPR based*** ISA, you have three different classifications. In the register --memory ISA, one operand has to be moved into any register and the other one can be a memory operand. In the register – register ISA, both operands will have to move to two registers and the ADD instruction will only work on registers. The memory – memory ISA permits both memory operands. So you can directly add.

In a ***stack-based*** ISA, you’ll have to first of all push both operands onto the stack and then simply give an add instruction which will add the top two elements of the stack and then store the result in the stack.

**Memory Addressing**

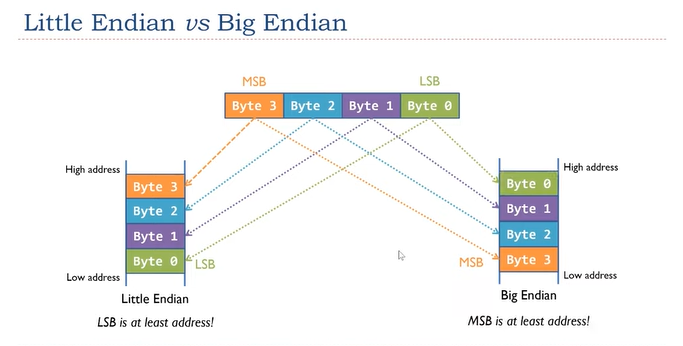
Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified.

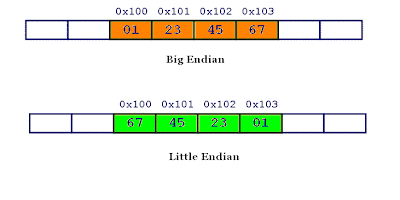
**Interpreting Memory Addresses**

All the instructions set are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the computers also provide access for double words (64 bits).

Two types of interpretation of the memory addresses – Big endian arrangement and the little-endian arrangement. Memories are normally arranged as bytes and a unique address of a memory location is capable of storing 8 bits of information. But when you look at the word length of the processor, the word length of the processor may be more than one byte. Suppose you look at a 32-bit processor, it is made up of four bytes. These four bytes span over four memory locations. When you specify the address of a word how you would specify the address of the word – are you going to specify the address of the most significant byte as the address of the word (big end) or specify the address of the least significant byte (little end) as the address of the word.

* **Big Endian Byte Order:** The **most significant** byte (the "big end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.
* **Little Endian Byte Order:** The **least significant** byte (the "little end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.





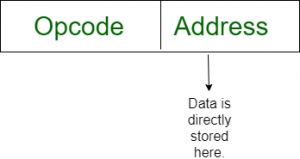
Little and big endian are two ways of storing multibyte data-types ( int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

# Addressing Modes/instruction set principles

**Addressing Modes**– The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

Types of Addressing Modes

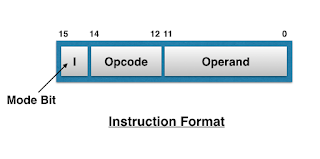
1. [Immediate Addressing Mode](https://binaryterms.com/addressing-modes-and-its-types.html#RegisterAddressingMode)
2. [Direct and indirect Addressing Mode](https://binaryterms.com/addressing-modes-and-its-types.html#DirectAddressingMode)
3. [Register Addressing Mode](https://binaryterms.com/addressing-modes-and-its-types.html#ImmediateAddressingMode)
4. [Register Indirect Addressing Mode](https://binaryterms.com/addressing-modes-and-its-types.html#RegisterIndirectAddressingMode)
5. [Index Addressing Mode](https://binaryterms.com/addressing-modes-and-its-types.html#IndexAddressingMode)
6. [Auto Increment and Auto Decrement Mode](https://binaryterms.com/addressing-modes-and-its-types.html#AutoIncrementMode)
7. [Displacement Mode](https://binaryterms.com/addressing-modes-and-its-types.html#AutoDecrementMode)

* **Immediate addressing mode (symbol #):** In this mode data is present in address field of instruction .Designed like one address instruction format.  
  **Note:** Limitation in the immediate mode is that the range of constants are restricted by size of address field.  
  

Example:  MOV AL, 35H (move the data 35H into AL register)

**2.Direct and Indirect Addressing Modes:**

   The instruction format for direct and indirect addressing mode is shown below:

[](https://1.bp.blogspot.com/-jDRgXG453Vs/V8AsvxYPFRI/AAAAAAAAAM0/OjF_Aj2WEss1XVkUshRrIZvsILWF4-AmwCLcB/s1600/Screen+Shot+2016-08-26+at+17.16.50.png)

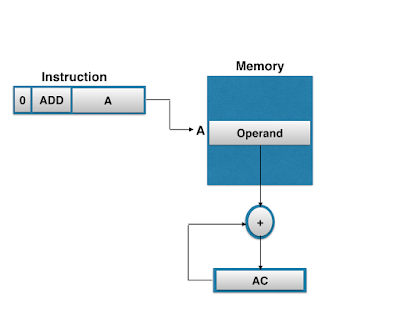
    It consists of 3-bit opcode, 12-bit address and a mode bit designated as (I). The mode bit (I) is zero for Direct Address and 1 for Indirect Address.

**Direct Addressing Mode:**

   Direct Addressing Mode is also known as “Absolute Addressing Mode”. In this mode the address of data(operand) is specified in the instruction itself. That is, in this type of mode, the operand resides in memory and its address is given directly by the address field of the instruction.

   Means, in other words, in this mode, the address field contain the Effective Address of operand  i.e., EA=A

   As an example:  Consider the instruction:

**ADD** **A**    Means  add contents of cell A to accumulator .   
  


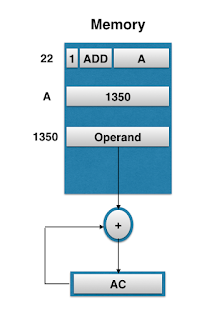
**Indirect Addressing Mode:**

   In this mode, the address field of instruction gives the memory address where on, the operand is stored in memory. That is, in this mode, the address field of the instruction gives the address where the “Effective Address” is stored in memory. i.e., EA=(A)

   Means, here, Control fetches the instruction from memory and then uses its address part to access   memory again to read Effective Address.

   As an example: Consider the instruction:

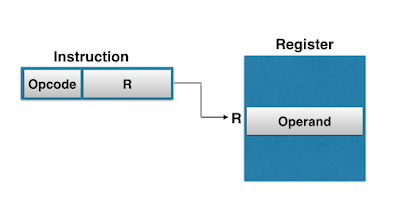
**ADD  (A)**     Means adds the content of cell pointed to contents of A to Accumulator.



**Register mode:**In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction.  
Here one register reference is required to access the data.

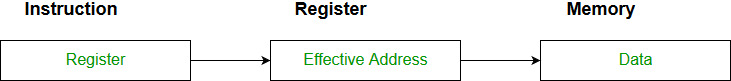
am3

Example: MOV AX, CX (move the contents of CX register to AX register)



**Register Indirect mode**: In this addressing the operand’s offset is placed in any one of the registers BX,BP,SI,DI as specified in the instruction. The effective address of the data is in the base register or an index register that is specified by the instruction.

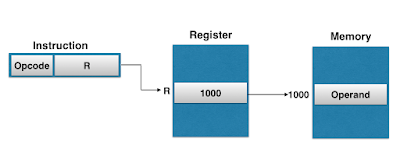
Here two register reference is required to access the data.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/Addressing_Modes_4.jpg)

The 8086 CPUs let you access memory indirectly through a register using the register indirect addressing modes.

MOV AX, [BX](move the contents of memory location s

addressed by the register BX to the register AX)



[**Index Addressing Mode**](https://binaryterms.com/addressing-modes-and-its-types.html#IndexAddressingMode)

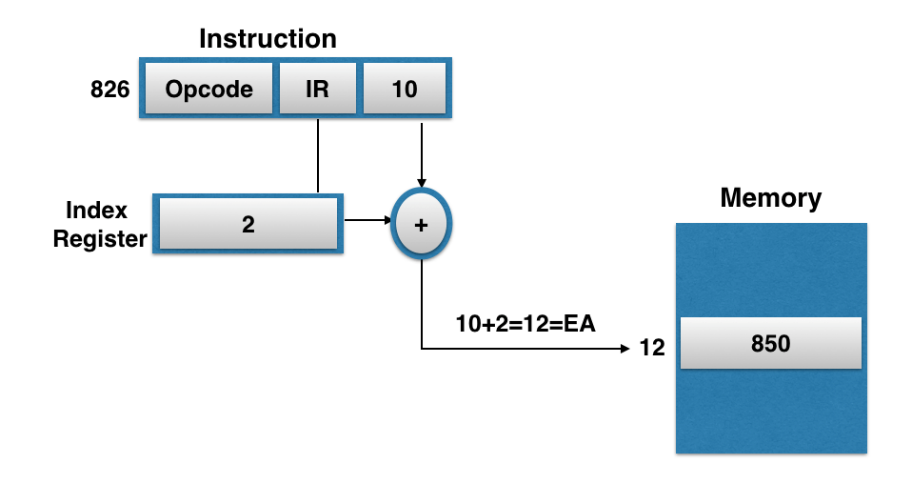
**In indexed addressing mode, the content of Index Register is added to direct address part(or field) of instruction to obtain the effective address**. Means, in it, the register indirect addressing field of instruction point to Index Register, which is a special CPU register  that contain an Indexed value, and direct addressing field contain base address.

          As, indexed type instruction make sense that data array is in memory and each operand in the array is stored in memory relative to base address. And the distance between the beginning  address and the address of operand is the indexed value stored in indexed register.

          Any operand in the array can be accessed with the same instruction, which provided that the index register contains the correct index value i.e., the index register can be incremented to facilitate access to consecutive operands.

          Thus, in index addressing mode

          EA=A+ Index



 The index addressing mode provides flexibility to specify memory locations. Disadvantage: The index addressing mode is complex to implement.

[**Auto Increment and Auto Decrement Mode**](https://binaryterms.com/addressing-modes-and-its-types.html#AutoIncrementMode)

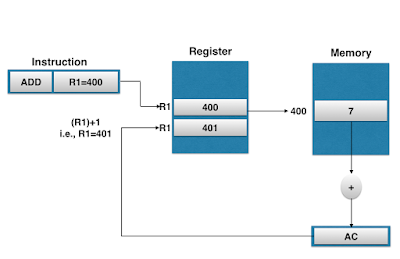
  These are similar to Register indirect Addressing Mode except that the register is incremented or decremented after (or before) its value is used to access memory.

**Auto-increment  Addressing Mode:**

   Auto-increment Addressing Mode are similar to Register Indirect Addressing Mode except that the register is incremented after its value is loaded (or accessed) at another location like accumulator (AC).

   That is, in this case also, the Effective Address is equal to  EA=(R)

   But, after accessing operand, register is incremented by 1.

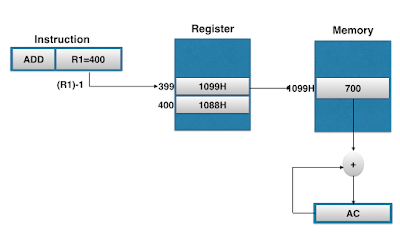


Here, we see that effective address is  (R )=400 and operand in AC is 7.

 And after loading R1 is incremented by 1.It becomes 401.Means, here we see that, in the Auto-increment mode, the R1 register is increment to 401 after  execution of instruction.

**Auto-decrement Addressing Mode:**

   Auto-decrement Addressing Mode is reverse of auto-increment, as in it the register is decrement before the execution of the instruction. That is, in this case, effective address is equal to EA=(R) -  1



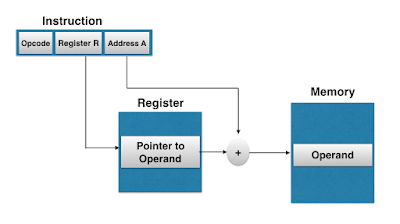
 Here, we see that, in the Auto-decrement mode, the register R1 is decremented to 399 prior to execution of the instruction, means the operand is loaded to accumulator, is of address 1099H in memory, instead of 1088H.Thus, in this case effective address is 1099H and contents loaded into accumulator is 700.

**7 ,Displacement Modes**

 Displacement Based Addressing Modes is a powerful addressing mode as it is a combination of direct    addressing or register indirect addressing mode.

   i.e., EA=A+(R)

Means, Displacement Addressing Modes requires that the instruction have two address fields, at least    one of which is explicit means, one is address field indicate direct address and other indicate indirect address. That is, value contained in one addressing field is A, which is used directly and the value in  other address field is R, which refers to a register whose contents are to be added to produce effective    address.



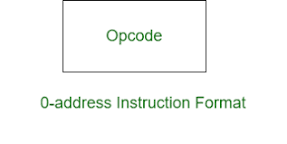
**Encoding an instruction set:**

The encoding of instructions, which is a key aspect of an Instruction Set Architecture, defines how instructions and arguments are encoded as binary values in the machine code of a system.

**Types of instruction formats are :**

* **Zero(0) Address Instruction format.** :
* The instruction format in which there is no address field is called zero address instruction. ...

Zero-address instruction is a format of machine instruction. It has **one opcode and no address fields**.



* **One(1) Address Instruction format.**:

One Address Instructions – This **use a implied ACCUMULATOR register for data manipulation**. One operand is in accumulator and other is in register or memory location. Implied means that the CPU already know that one operand is in accumulator so there is no need to specify it.

* **Two(2) Address Instruction format. :**

Two-address instruction is a format of machine instruction. It has **one opcode and two address fields**. One address field is common and can be used for either destination or source and other address field for source.

* **Three(3) Address Instruction format:**

THREE-ADDRESS INSTRUCTIONS (consisting of one opcode and three address fields.)

Two processor registers and one memory operand

Three basic variations in instruction encoding:

* **variable length,**
* **fixed length,**
* **and hybrid.**

The *variable format* can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The *fixed format* always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. The *hybrid approach* has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

