


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.

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 01234567 will be stored as following.



Addressing Modes

What is Addressing Modes:

The operation field of an instruction specifies the operation to be performed. And this operation must be performed on some data. So each instruction need to specify data on which the operation is to be performed. But the operand(data) may be in accumulator, general purpose register or at some specified memory location. So, appropriate location (address) of data is need to be specified, and in computer, there are various ways of specifying the address of data. These various ways of specifying the address of data are known as “**Addressing Modes**”

So **Addressing Modes** can be defined as-“*The technique for specifying the address of the operands*” And in computer the address of operand i.e., the address where operand is actually found is known as “Effective Address”.

Types of Addressing Modes

1. Immediate Addressing Mode
2. Direct and indirect Addressing Mode
3. Register Addressing Mode
4. Register Indirect Addressing Mode
5. Index Addressing Mode
6. Auto Increment and Auto Decrement Mode
7. Displacement Mode

1. Immediate Addressing Mode:

In Immediate Addressing Mode operand is specified in the instruction itself. In other words, an immediate mode instruction has an operand field rather than an address field, which contain actual operand to be used in conjunction with the operand specified in the instruction. That is, in this mode, the format of instruction is:



As an example: The Instruction:

MVI 06 Move 06 to the accumulator
ADD 05 ADD 05 to the content of accumulator

2.Direct and Indirect Addressing Modes:

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



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.

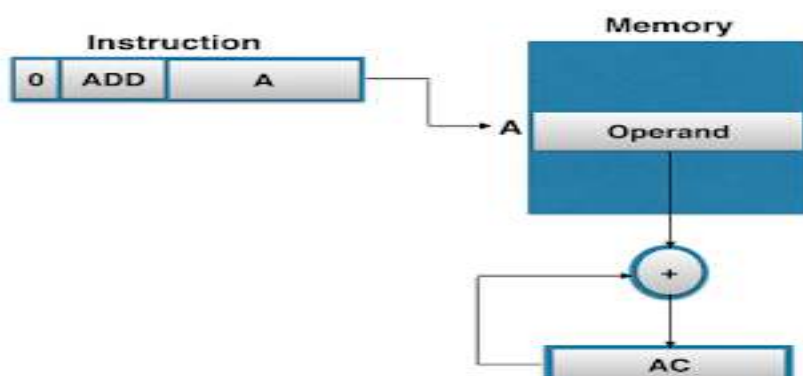
Direct Addressing Mode:

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 .



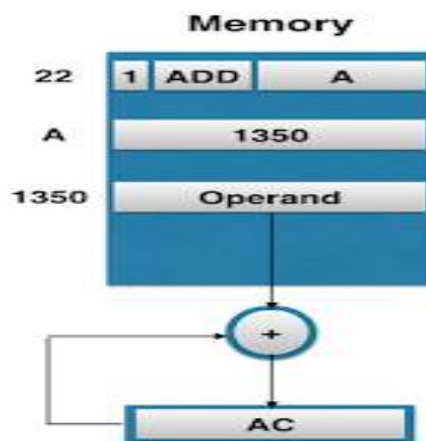
Indirect Addressing Mode:

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:

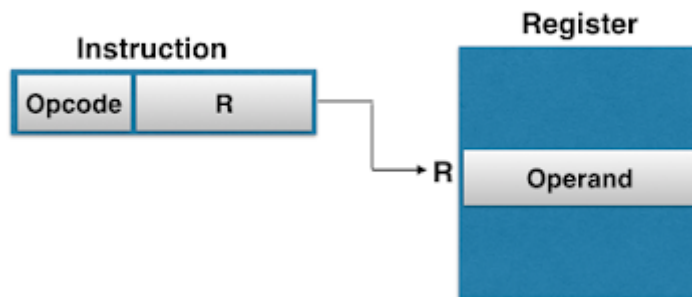
ADD (A) Means adds the content of cell pointed to contents of A to Accumulator.



3.Register Addressing Mode

In Register Addressing Mode, the operands are in registers that reside within the CPU. That is, in this mode, instruction specifies a register in CPU, which contain the operand. It is like Direct Addressing Mode, the only difference is that the address field refers to a register instead of memory location.

i.e., $EA=R$



Example of such instructions are:

MOV AX, BX Move contents of Register BX to AX
ADD AX, BX Add the contents of register BX to AX

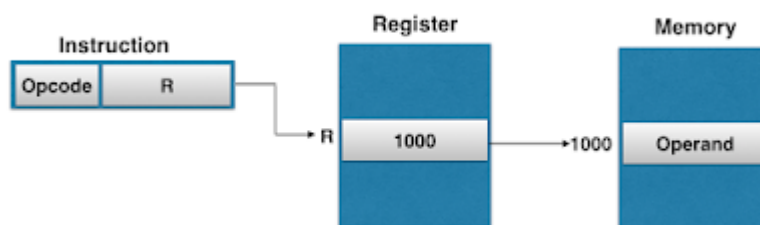
Here, AX, BX are used as register names which is of 16-bit register.

Thus, for a Register Addressing Mode, there is no need to compute the actual address as the operand is in a register and to get operand there is no memory access involved.

4.Register Indirect Addressing Mode

In Register Indirect Addressing Mode, the instruction specifies a register in CPU whose contents give the operand in memory. In other words, the selected register contains the address of operand rather than the operand itself. That is, i.e., $EA=(R)$. Here, the parentheses are to be interpreted as meaning contents of.

Means, control fetches instruction from memory and then uses its address to access Register and looks in Register(R) for effective address of operand in memory.



Example of such instructions are:

MOV AL, [BX]

Code example in Register:

MOV BX, 1000H

MOV 1000H, operand

From above example, it is clear that, the instruction(MOV AL, [BX]) specifies a register[BX], and in coding of register, we see that, when we move register [BX], the register contain the address of operand(1000H) rather than address itself.

5.Index Addressing Mode

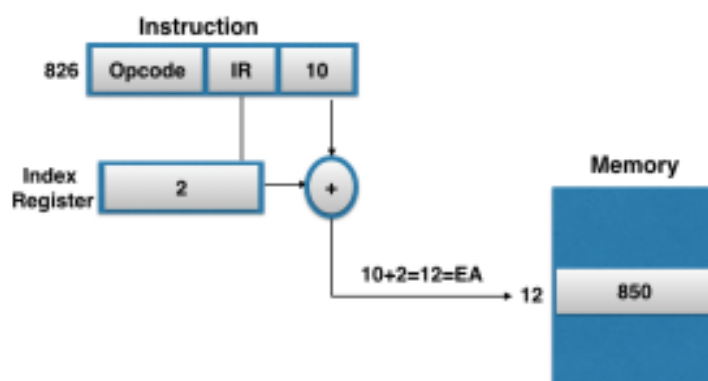
In indexed addressing mode, the content of Index Register is added to direct address part(or field) of instruction to obtain the effective address.Means, in it, the register indirect addressing field of instruction point to Index Register, which is a special CPU register that contain an Indexed value,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

$$EA = A + \text{Index}$$



6.Auto Increment and Auto Decrement Mode

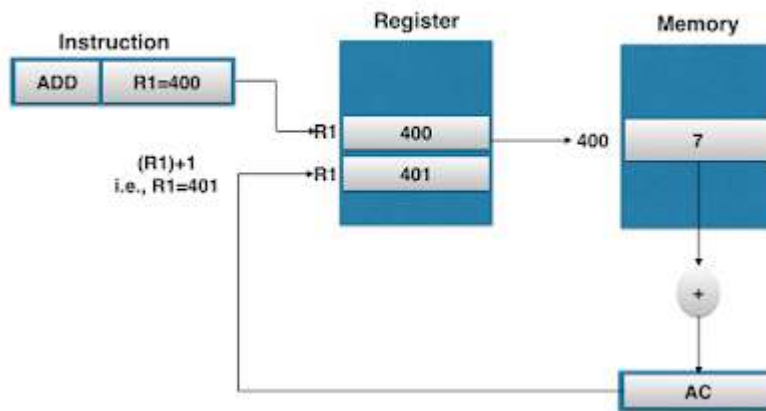
These are similar to Register indirect Addressing Mode except that the register is incremented or decremented after(or before) its value is used to access memory.

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.

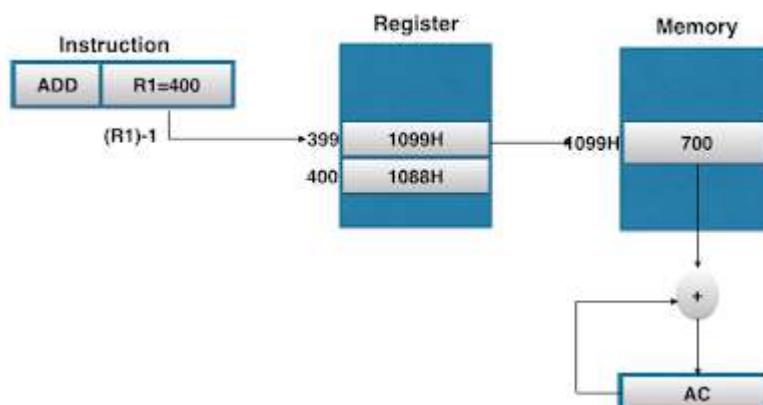


Here, we see that effective address is $(R)=400$ and operand in AC is 7.

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 incremented 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 decremented before the execution of the instruction. That is, in this case, effective address is equal to $EA=(R) - 1$



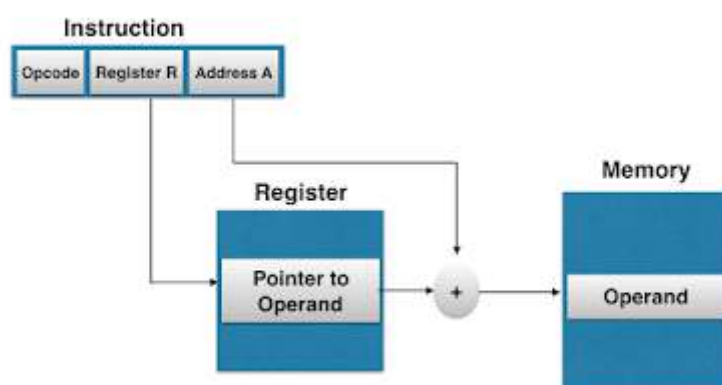
Here, we see that, in the Auto-decrement mode, the register R1 is decremented to 399 prior to execution of the instruction, means the operand is loaded to accumulator, is of address 1099H in memory, instead of 1088H. Thus, in this case effective address is 1099H and contents loaded into accumulator is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

$$\text{i.e., } EA = A + (R)$$

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

We have introduced a variety of useful instructions and addressing modes. These instructions specify the actions that must be performed by the processor circuitry to carry out the desired tasks. We have often referred to them as machine instructions. To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as machine instructions. The instructions that use symbolic names and acronyms are called assembly language instructions, which are converted into the machine instructions using the assembler program.

This representation affects not only the size of the compiled program but also the implementation of the processor, which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the opcode.

How the addressing modes of operands are encoded depends on

- the range of addressing modes
- the degree of independence between opcodes and modes

Some older computers have one to five operands with 10 addressing modes for each operand. For such a large number of combinations, typically a separate *address specifier* is needed for each operand: The address specifier tells what addressing mode is used to access the operand. At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the *opcode*.

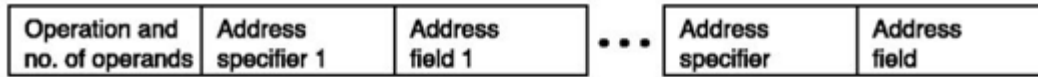
For **small** number of addressing modes, the addressing mode can be encoded in **opcode**. For a **larger** number of combinations, typically a separate **address specifier** is needed for each operand.

The architect has to balance several competing forces when encoding the instruction set:

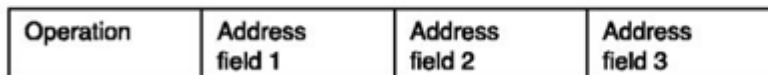
- The desire to have **as many** registers and addressing modes **as possible**.
 - The **impact of the size** of the register and addressing mode fields on the average instruction size and hence on the average program size.
 - A desire to have instructions encode into lengths that are **easy to handle** in the implementation (multiples of bytes, fixed-length) with possible sacrificing in average code size.

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

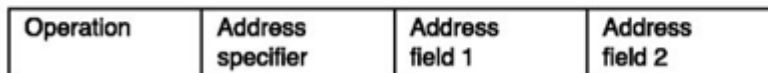
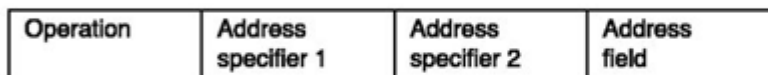
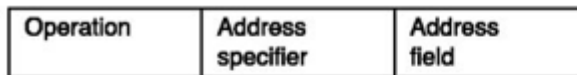
the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.



(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)



(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)