ICT 409

Md. Sharif Hossen
Assistant Prof., ICT, CoU

Dec 21, 2020

Addressing Modes

Operand and Operator

In computer programming, an **operand** is a term used to describe any object that is capable of being manipulated. For example, in "1 + 2" the "1" and "2" are the **operands** and the plus symbol is the operator.

In mathematics an **operand** is the object of a mathematical operation, i.e., it is the object or quantity that is operated on. Example, 3 + 6 = 9 where '+' is the symbol for the operation called addition. The **operand** '3' is one of the inputs (quantities) followed by the addition operator, and the operand '6' is the other input necessary for the operation. The result of the operation is 9.

Instruction Sets

- Instruction Formats
- Addressing Modes
- Relative Addressing
- Instruction Types

Instruction Formats

- The **purpose of an instruction** is to specify both an **operation** to be carried out by a CPU or other processor and the set of **operands** or data to be used in the operation. The operands include the input data or arguments of the operation and the results that are produced.

Most instructions specify a register-transfer operation of the form

$$X_i = op(x_1, x_2, \ldots x_n)$$

Which applies the operation op to n operands $X_1, X_2, \ldots X_n$

In assembly language format

$$op\ X_1, X_2, \ldots X_n$$

Instruction Formats contd.

The operation op is specified by a field called the opcode (operation code). The n $X_1, X_2, \ldots X_n$ fields are referred to as **addresses**. An address X_i typically names a ***register*** that stores an operand value. In some instances, X_i **itself is the desired value**, in which case it is called an immediate address.

Inside the computer, instructions are **stored as binary words**. There can be several different sizes and **formats**. **RISCs** tend to have few instruction formats, while ***CISCs*** tend to have many to accommodate more opcode types and operand addressing methods.

The Motorola 680X0 is a CISC micorprocessor series. Instruction length in the 680X0 varies from **2 to 10 bytes**.

Instruction Formats

The 2-byte opcode field of the 680X0 is often used to hold one or two **3-bit register addresses**, blurring the distinction between opcode and operand.

In the 680X0 family, simple instructions are assigned short formats. For example, the add-register instruction

$$ADD.L\ D_1, D_2$$

denotes register-to-register addition of 32-bit (long word) operands, that is,

$$D2: = D2 + D1$$

Where D_1 and D_2 are two of the 680X0 's data registers.

Instruction Formats

Similarly, for Motorola 680X0 family

- ADD.L D1, D2 (Register to register addition)

  → D2:=D2+D1

- ADD.L ADR1, D2 (Memory to register addition) [4 byte]

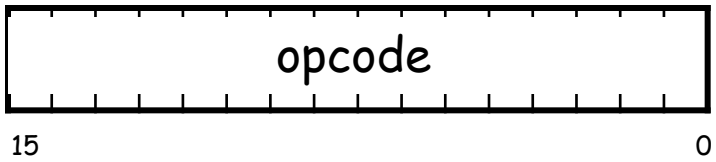
  →D2=D2+M(ADR1)

- MOVE.B ADR1,ADR2 (Memory to Memory value transfer)

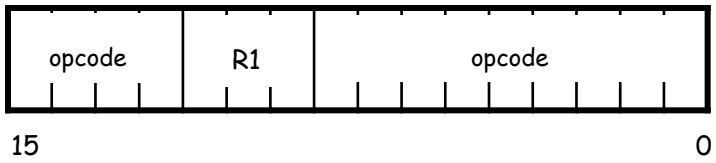
  →M(ADR2)=M(ADR1)

Instruction Formats

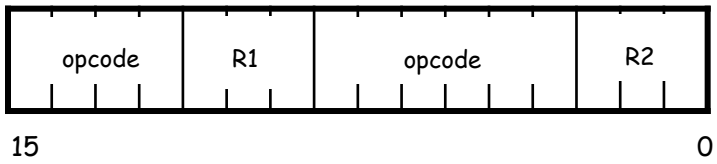
- Format 1 (F1)



- Format 2 (F2)



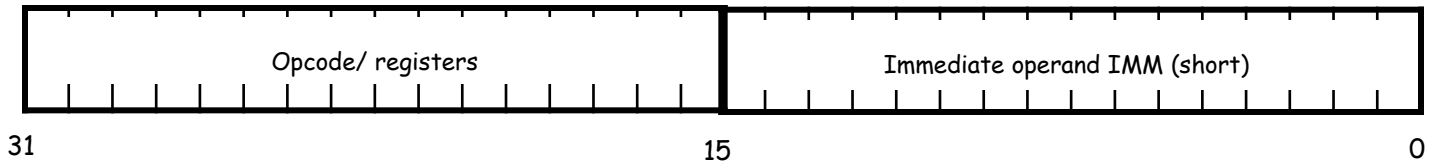
- Format 3 (F3)



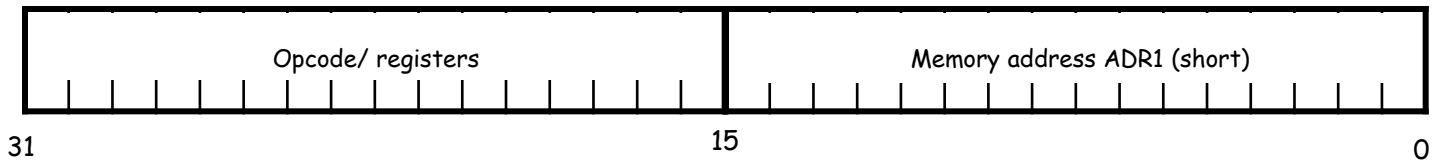
- ADD.L D1,D2

Instruction Formats

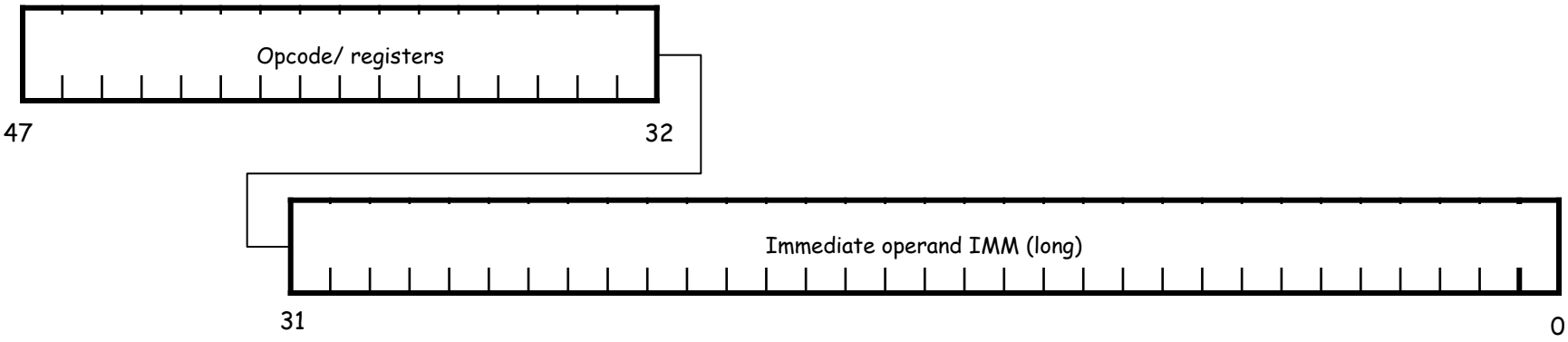
- Format 4 (F4)



- Format 5 (F5)

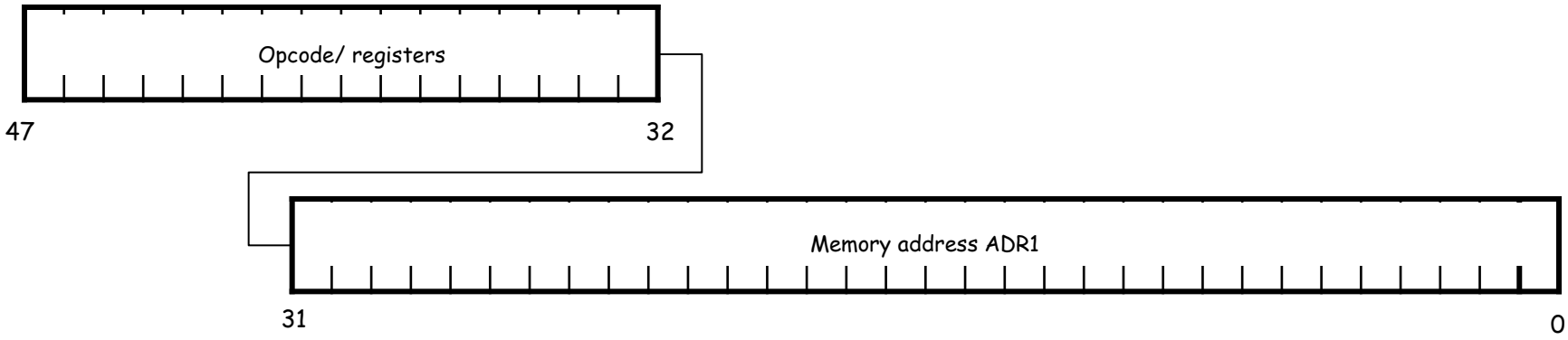


- Format 6 (F6)

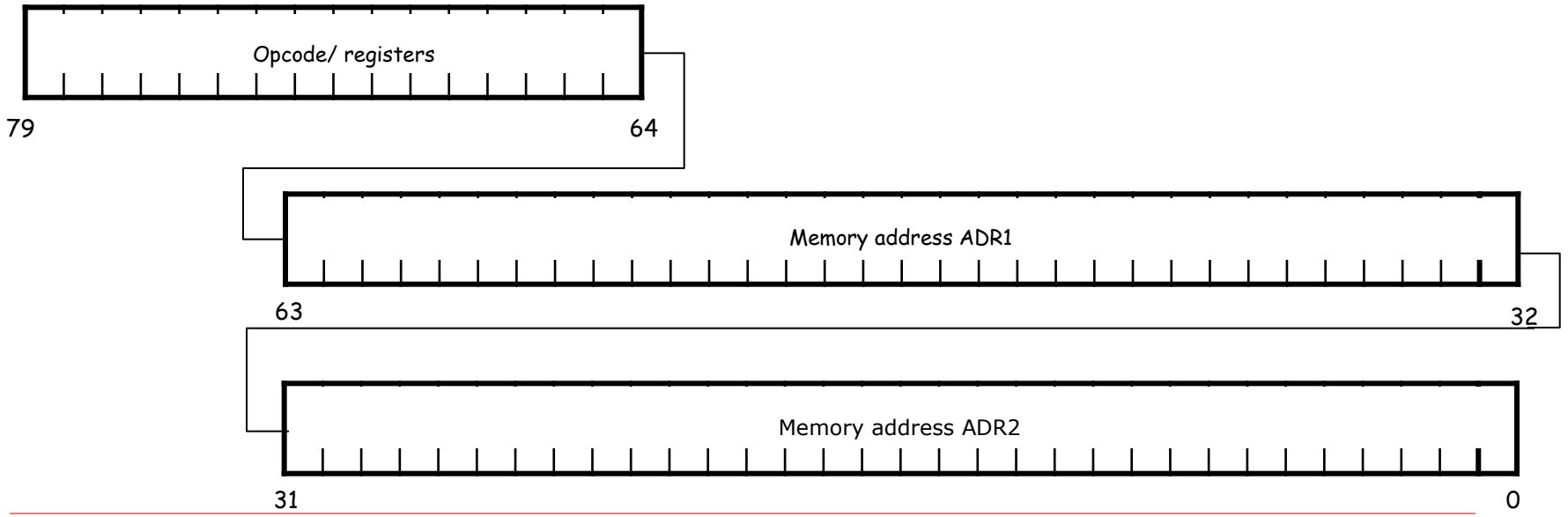


Instruction Formats

- Format 7 (F7)



- Format 8 (F8)



Addressing Modes

Addressing Modes– The term addressing modes refer 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.

Addressing Modes

The **purpose** of an address field is to point to the current value V(X) of some operand X used by an instruction. This **value can be specified in various ways**, which are termed addressing modes. The addressing mode of X **affects the following issues**:

- The **speed** with which V(X) can be accessed by the CPU.
- The **ease** with which V(X) can be specified and altered.

Access speed is influenced by the physical location of V(X) - normally the CPU or the external memory M. Operand values located in CPU registers, such as the general-register file and the program counter PC, can be accessed faster than operands in M

Addressing Modes contd.

There are three basic addressing modes:

(i) Immediate addressing

(ii) Direct addressing

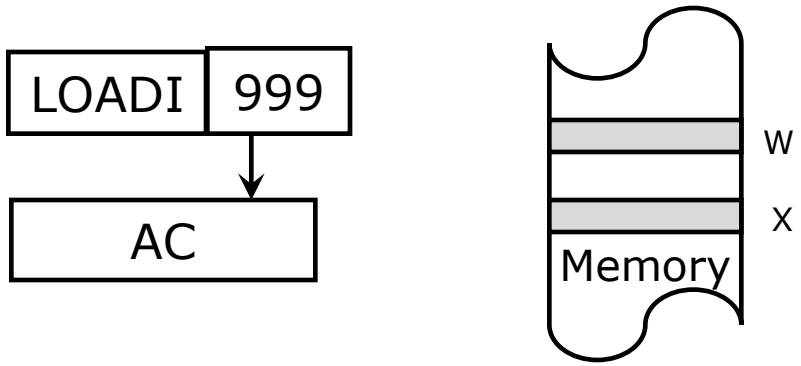
(iii) Indirect addressing

Addressing Modes contd.

(i) Immediate addressing: If the value V(X) of the target **operand** is contained in the **address field itself**, then X is called an immediate operand, and the corresponding addressing mode is immediate addressing. By implication **X is a constant**, since it is very undesirable to modify instruction fields during execution.

For example, the operation A: = 999 is an intermediate operand, and hence
the 8085 instruction
MVI A, 999

with opcode MVI must be used.



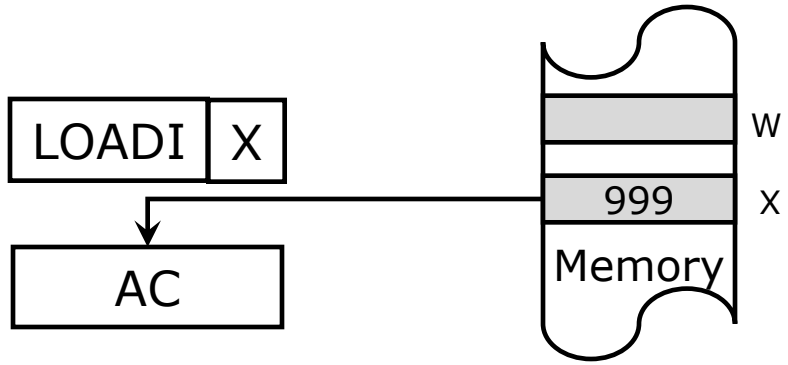
Addressing Modes contd.

In this mode, the **operand is specified in the instruction itself**. An immediate mode instruction has an **operand field rather than the address field**.

For example: ADD 7, which says Add 7 to contents of accumulator. 7 is the operand here

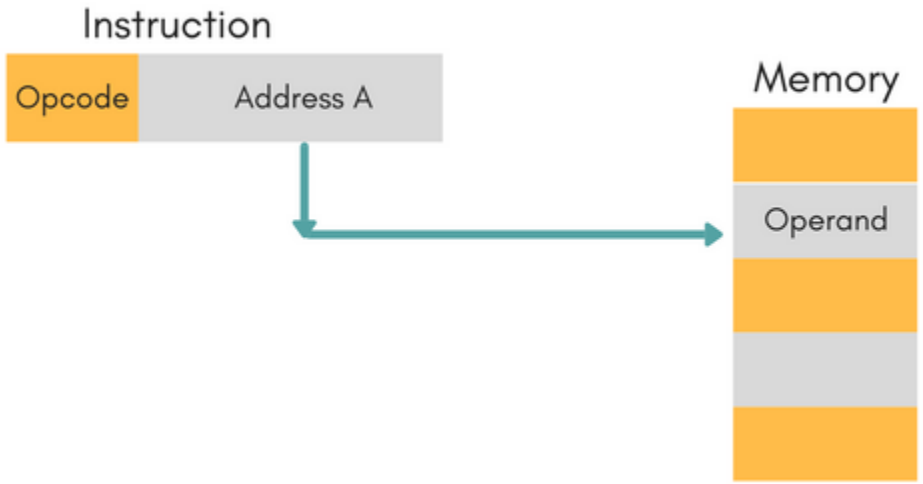
Addressing Modes contd.

(ii) Direct addressing: In this case, X is a **variable** and the corresponding **address field identifies the storage location** that contains the required value V(X). Thus X corresponds to a variable, and its value V(X) can be varied without modifying the instruction address field. Operand specification of this type is called direct addressing.



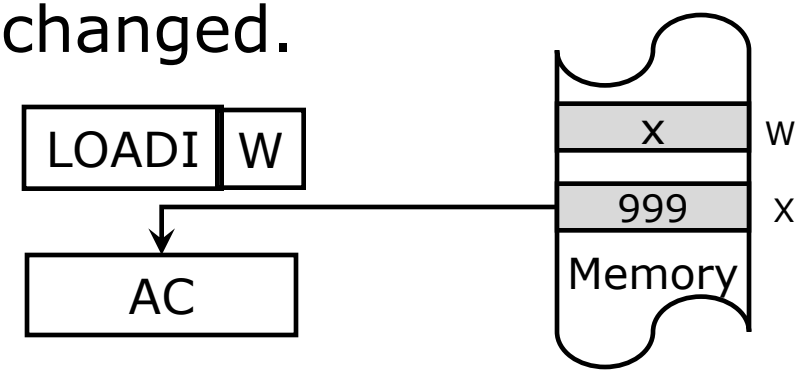
Addressing Modes contd.

In this mode, effective **address of operand is present** in instruction itself.



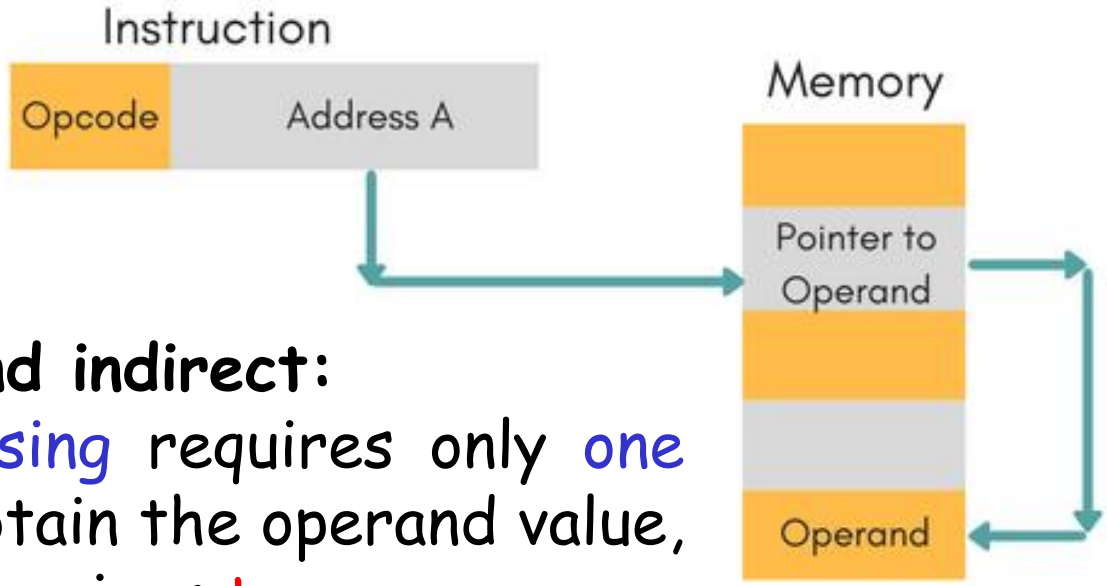
Addressing Modes contd.

(iii) Indirect addressing: It is sometimes useful to **change the location of X** without changing the address fields of any instructions that refer to X. This may be accomplished by indirect addressing, whereby the instruction contains the address W of a storage location, which in turn contains the address X of the desired operand value V(X). By changing the contents of W, the address of the operand value required by the instruction is effectively changed.



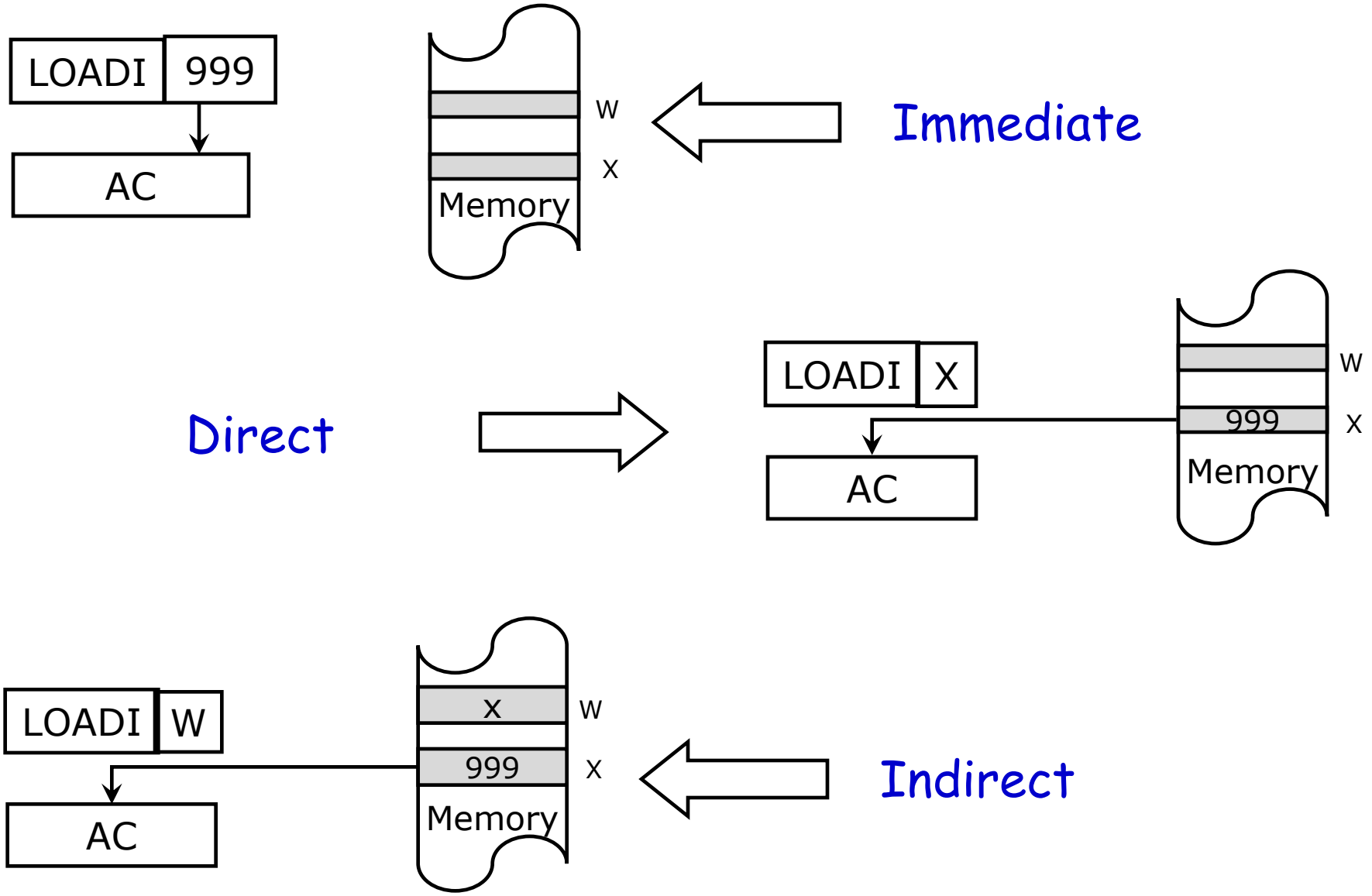
Addressing Modes contd.

In this, the ***address field*** **of instruction** gives the address where the **effective address is stored in memory**. This slows down the execution, as this includes multiple memory lookups to find the operand.



Difference direct and indirect:
While direct addressing requires only one fetch operation to obtain the operand value, indirect addressing requires two.

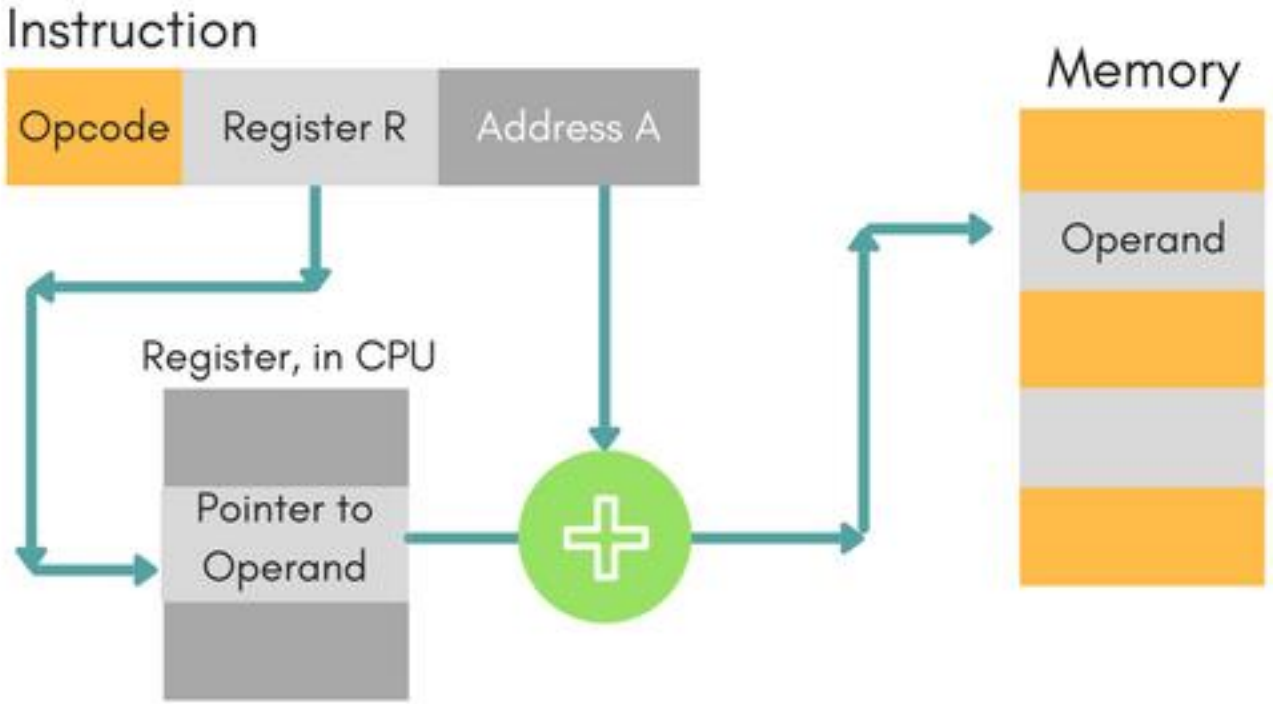
Addressing Modes contd.



Displacement Addressing Mode

In this, the contents of the indexed register are added to the Address part of the instruction, to obtain the effective address of operand.

EA = A + (R), In this the address field holds two values, A (which is the base value) and R(that holds the displacement), or vice versa.



Relative Addressing

Often a program needs to **jump to the next instruction**, i.e., just a few memory locations away from the current instruction.

One efficient way is to just add a small offset to the current address in the program counter. (N.B. **PC always points to the next instruction** to be executed). This is called relative addressing.

Relative addressing means that the next instruction to be carried out is an offset number of locations away, relative to the address of the current instruction.

Relative Addressing

Relative addressing is an **address specified by indicating its distance from another address**, called base address. For example, a relative address might be B+15, where B being the base address and 15 the distance, called offset.

It is a version of Displacement addressing mode.

In this the **contents of PC (Program Counter) is added to address part of instruction to obtain the effective address**.

EA = A + (PC), where EA is effective address and PC is program counter.

The operand is A cells away from the current cell(the one pointed to by PC)

Instruction types

A computer must have the following types of instructions:

- Data transfer instructions
- Data manipulation instructions
- Program sequencing and control instructions
- Input and output instructions

Data transfer instructions perform data transfer between the various storage places in the computer system, viz. registers, memory and I/O.

Instruction types

Data manipulation instructions perform operations on data and indicate the computational capabilities for the processor. These operations can be arithmetic operations, logical operations or shift operations. Arithmetic operations include addition (with and without carry), subtraction (with and without borrow), multiplication, division, increment, decrement and finding the complement of a number. The logical and bit manipulation instructions include AND, OR, XOR, Clear carry, set carry, etc. Similarly, you can perform different types of shift and rotate operations.

Input and Output instructions are used for transferring information between the registers, memory and the input / output devices. It is possible to use special instructions that exclusively perform I/O transfers, or use memory – related instructions itself to do I/O transfers.

Instruction types

We generally assume a sequential flow of instructions. That is, instructions that are stored in consequent locations are executed one after the other. However, you have program sequencing and control instructions that help you change the flow of the program. This is best explained with an example. Consider the task of adding a list of *n* numbers. A possible sequence is given below.

```
Move DATA1, R0
Add DATA2, R0
Add DATA3, R0
Add DATAn, R0
Move R0, SUM
```

Instruction types

Requirements to be satisfied by an instruction set are as follows:

1) It should be **complete** in the sense that we should be able to construct a machine-language program to evaluate any function that is computable using a reasonable amount of memory space.
2) It should be **efficient** in that frequently required functions can be performed rapidly using relatively few instructions.
3) It should be **regular** in that the instruction set should contain expected opcodes and addressing modes; for example, if there is a left shift, there should be a right shift.
4) It should be **compatible** with existing machines to reduce both hardware and software design costs.