İstanbul
Bilgi Üniversitesi

CMPE321 - Computer Architecture

Lecture 10
Addressing Modes

Hakan Ayral, PhD.

Register and Register-Indirect Modes

- Earlier, we mentioned that the address field of the instruction may specify either a memory location or a processor register.
- When the address field specifies a processor register, the instruction is said to be in the **register mode**.
- In this mode, the operands are in registers that reside within the processor of the computer.
- The particular register is selected by a register address field in the instruction format.

Register-indirect Mode

- In the **register-indirect mode**, the instruction specifies a register in the processor whose content gives the address of the operand in memory.
- In other words, the selected register contains the memory address of the operand, rather than the operand itself.
- Before using a register-indirect mode instruction, the programmer must ensure that the memory address is available in the processor register.
- A reference to the register is then equivalent to specifying a memory address.
- The advantage of register-indirect mode is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement/Autodecrement Mode

- An **autoincrement** or **autodecrement mode** is similar to the register-indirect mode, except that the register is incremented or decremented after (or before) its address value is used to access memory.
- When the address stored in the register refers to an array of data in memory, it is convenient to increment the register after each access to the array.
- This can be achieved by using a separate register-increment instruction.
- However, because it is such a common requirement, some computers incorporate an autoincrement mode that increments the content of the register containing the address after the memory data are accessed.

Autoincrement/Autodecrement Mode

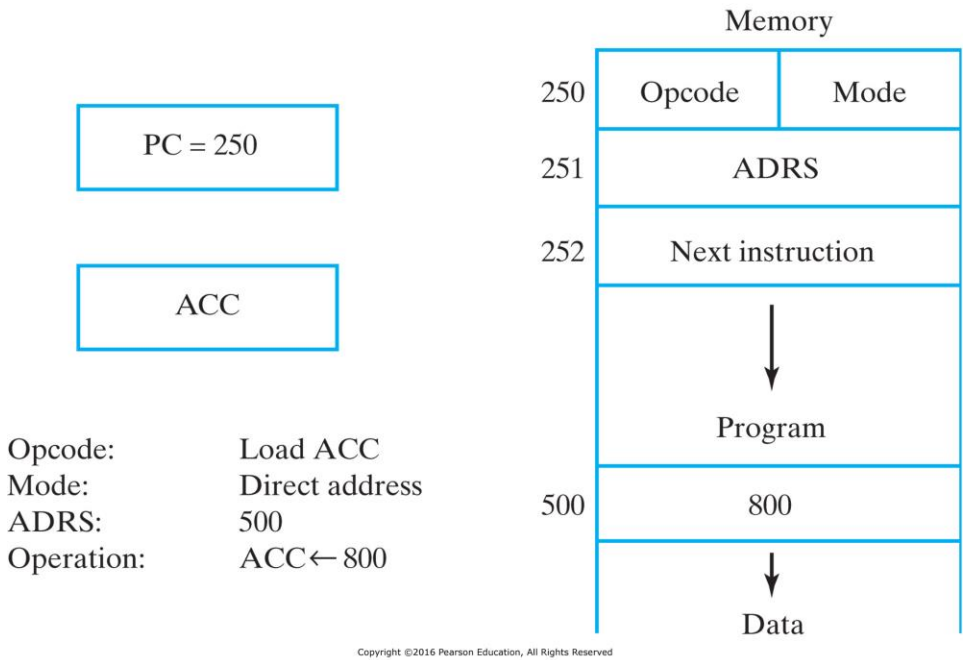
- In the following instruction, an autoincrement mode is used to add the constant value 3 to the elements of an array addressed by register R1:

$$\text{ADD} \qquad (\text{R1})+, 3 \qquad\qquad M[R1] \leftarrow M[R1] + 3, R1 \leftarrow R1 + 1$$

- R1 is initialized to the address of the first element in the array.
- Then the ADD instruction is repeatedly executed until the addition of 3 to all elements of the array has occurred.
- The register transfer statement accompanying the instruction shows the addition of 3 to the memory location addressed by R1 and the incrementing of R1 in preparation for the next execution of the ADD on the next element in the array.

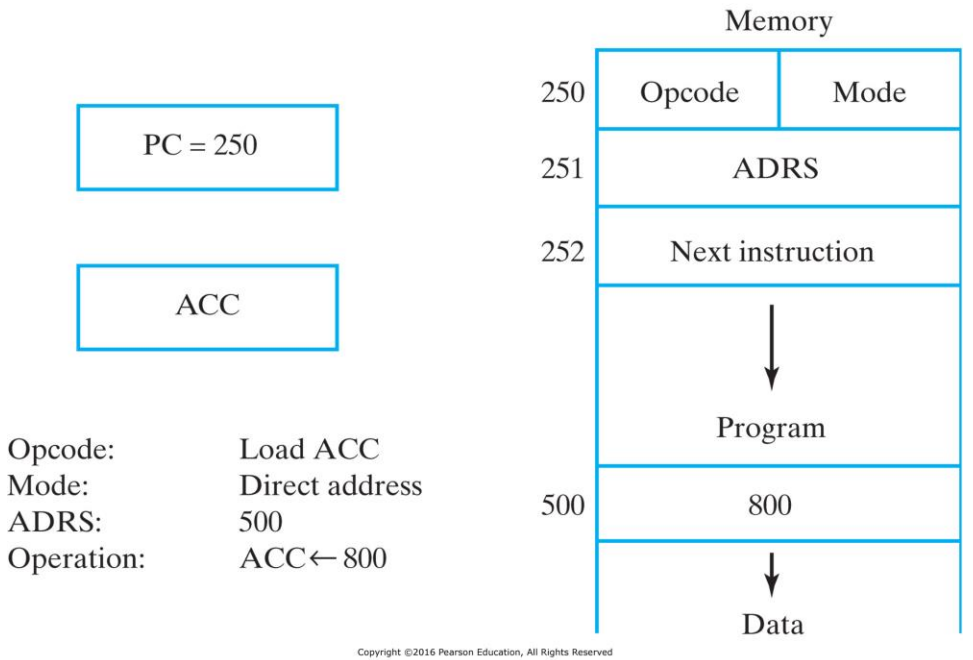
Direct Addressing Mode

- In the **direct addressing mode**, the address field of the instruction gives the address of the operand in memory in a data-transfer or data-manipulation instruction.
- An example of a data-transfer instruction is shown in Figure.
- The instruction in memory consists of two words.
- The first, at address 250, has the opcode for “load to ACC” and a mode field specifying a direct address.
- The second word of the instruction, at address 251, contains the address field, symbolized by ADRS, and is equal to 500.
- The ***PC*** holds the address of the instruction, which is brought from memory using two memory accesses.



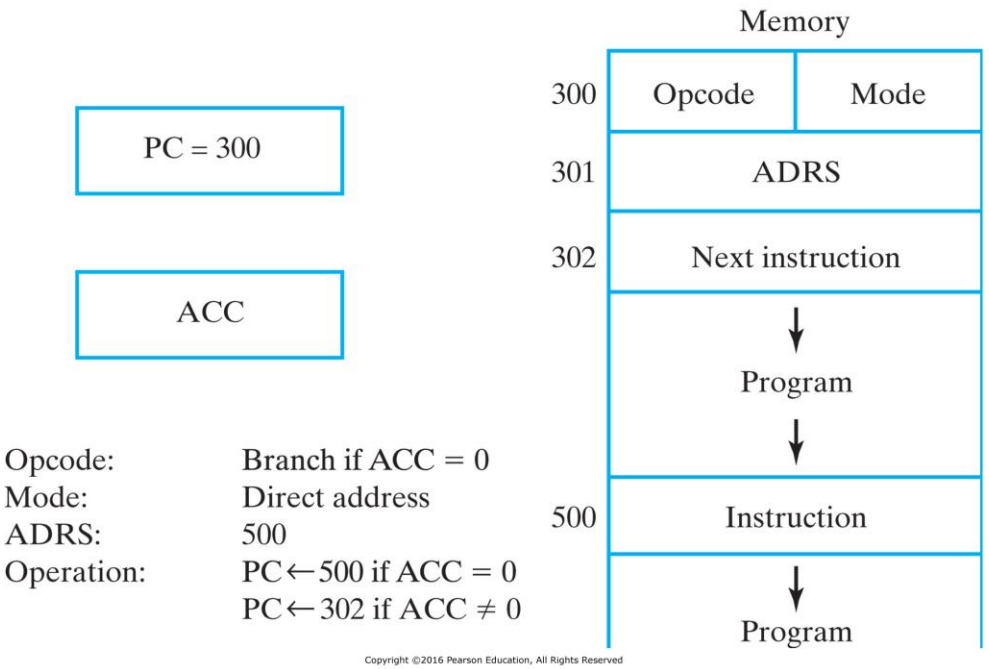
Direct Addressing Mode

- Simultaneously with or after the completion of the first access, the *PC* is incremented to 251.
- Then the second access for ADRS occurs and the *PC* is again incremented.
- The execution of the instruction results in the operation $ACC \leftarrow M[\text{ADRS}]$
- Since ADRS 500 and M[500] 800, the *ACC* receives the number 800.
- After the instruction is executed, the *PC* holds the number 252, which is the address of the next instruction in the program.



Direct Addressing Mode

- Now consider a branch-type instruction, as shown in Figure.
- If the contents of *ACC* equal 0, control branches to ADRS; otherwise, the program continues with the next instruction in sequence.
- When *ACC*=0, the branch to address 500 is accomplished by loading the value of the address field ADRS into the *PC*.
- Control then continues with the instruction at address 500.
- When *ACC*≠0, no branch occurs, and the *PC*, which was incremented twice during the fetch of the instruction, holds the address 302, the address of the next instruction in sequence.

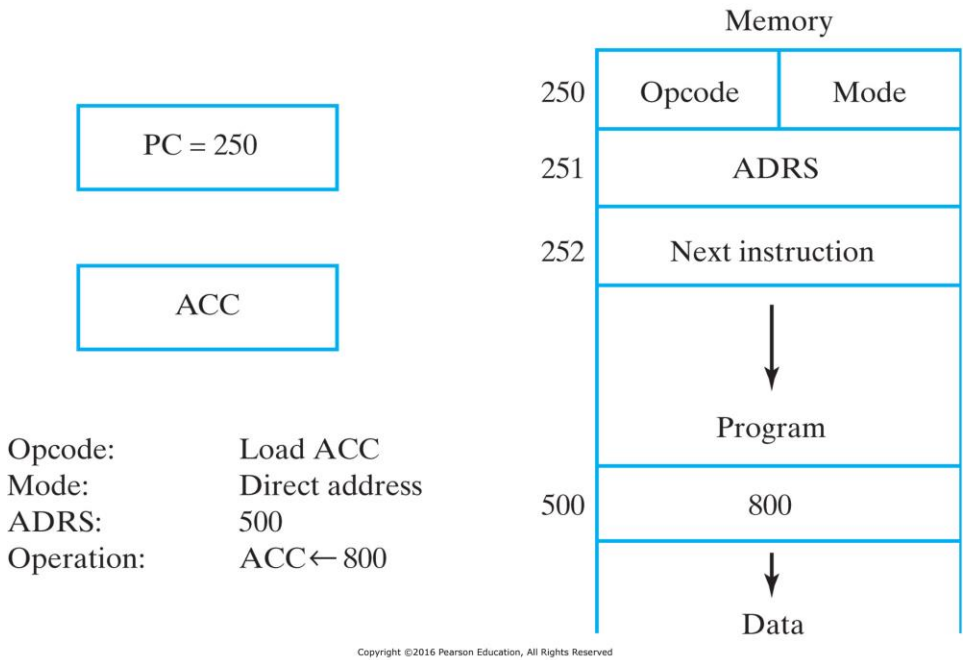


Direct Addressing Mode

- Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- To differentiate among the various addressing modes, it is useful to distinguish between the address part of the instruction, as given in the address field, and the address used by the control when executing the instruction.
- Recall that we refer to the latter as the **effective address**.

Indirect Addressing Mode

- In the indirect addressing mode, the address field of the instruction gives the address at which the effective address is stored in memory.
- The control unit fetches the instruction from memory and uses the address part to access memory again in order to read the effective address.
- Consider the instruction "load to *ACC*" given in Figure.
- If the mode specifies an indirect address, the effective address is stored in *M*[ADRS].
- Since ADRS = 500 and *M*[ADRS] = 800, **the effective address is 800**.
- This means that the operand loaded into the *ACC* is the one found in memory at address 800 (not shown in the figure).



Relative Addressing Mode

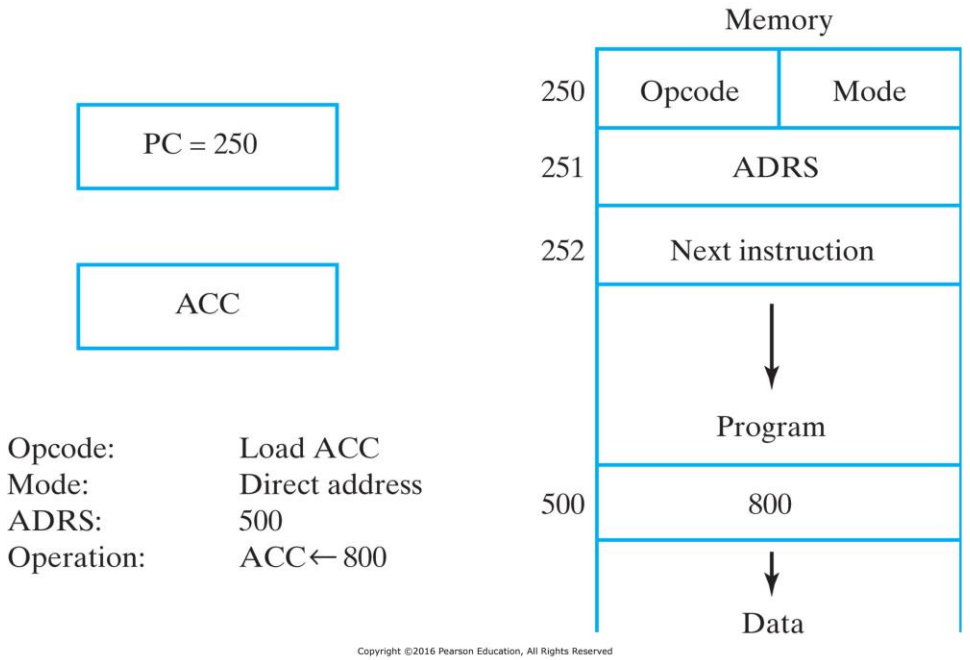
- Some addressing modes require that the address field of the instruction be added to the content of a specified register in the CPU in order to evaluate the effective address.
- Often, the register used is the *PC*. In the relative addressing mode, the effective address is calculated as follows:

$$\text{Effective address} = \text{Address part of the instruction} + \text{Contents of } PC$$

- The address part of the instruction is considered to be a signed number that can be either positive or negative.
- When this number is added to the contents of the *PC*, the result produces an effective address whose position in memory is relative to the address of the next instruction in the program.

Relative Addressing Mode

- To clarify this with an example, let us assume that the *PC* contains the number 250 and the address part of the instruction contains the number 500, as in Figure, with the mode field specifying a relative address.
- The instruction at location 250 is read from memory during the fetch phase of the operation cycle, and the *PC* is incremented by 1 to 251.
- Since the instruction has a second word, the control unit reads the address field into a control register, and the *PC* is incremented to 252.
- The computation of the effective address for the relative addressing mode is 252 + 500 = 752.
- The result is that the operand associated with the instruction is 500 locations away, relative to the location of the next instruction.



Relative Addressing Mode

- Relative addressing is often used in branch-type instructions when the branch address is in a location close to the instruction word.
- Relative addressing produces more compact instructions, since the relative address can be specified with fewer bits than are required to designate the entire memory address.
- This permits the relative address field to be included in the same instruction word as the opcode.

Indexed Addressing Mode

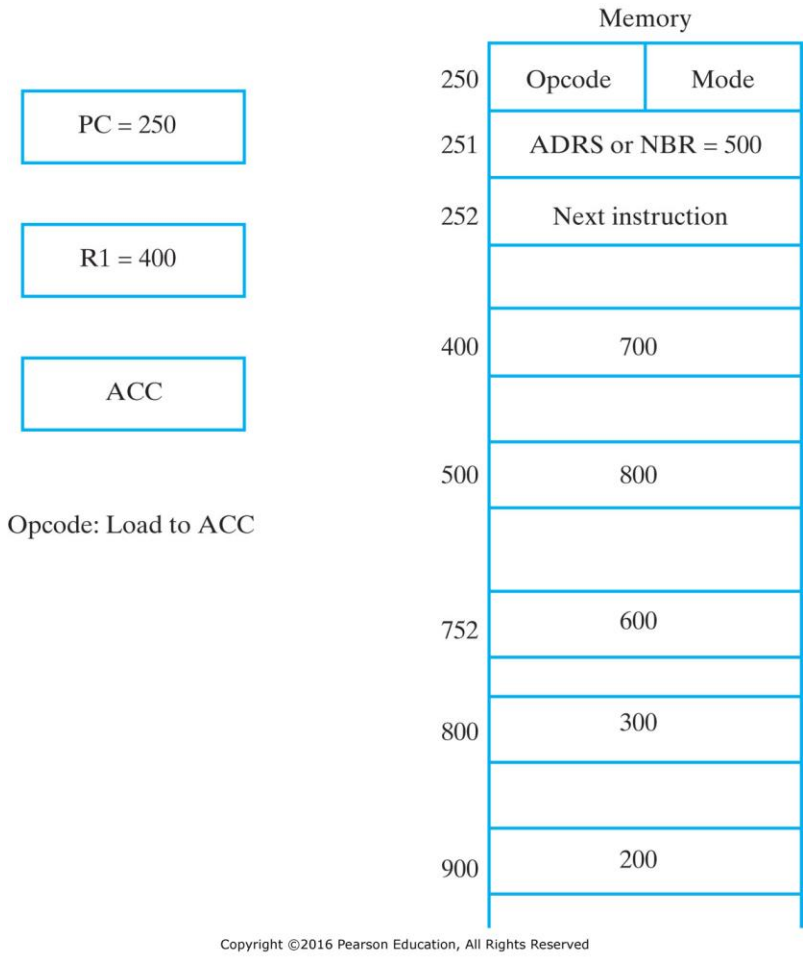
- In the indexed addressing mode, the content of an index register is added to the address part of the instruction to obtain the effective address.
- The index register may be a special CPU register or simply a register in a register file.
- We illustrate the use of indexed addressing by considering an array of data in memory.
- The address field of the instruction defines the beginning address of the array.
- Each operand in the array is stored in memory relative to the beginning address.
- The distance between the beginning address and the address of the operand is the index value stored in the register.
- Any operand in the array can be accessed with the same instruction, provided that the index register contains the correct index value.
- The index register can be incremented to facilitate access to consecutive operands.
- Some computers dedicate one CPU register to function solely as an index register.
- This register is addressed implicitly when an index-mode instruction is used.
- In computers with many processor registers, any CPU register can be used as an index register.
- In such a case, the index register to be used must be specified with a register field within the instruction format.

Indexed Addressing Mode

- A specialized variation of the index mode is the base-register mode.
- In this mode, the contents of a base register are added to the address part of the instruction to obtain the effective address.
- This is similar to indexed addressing, except that the register is called a base register instead of an index register.
- The difference between the two modes is in the way they are used rather than in the way addresses are computed:
  - an **index register** is assumed to hold an index number that is relative to the address field of the instruction;
  - a **base register** is assumed to hold a base address, and the address field of the instruction gives a displacement relative to the base address.

Summary of Addressing Modes

- In order to show the differences among the various modes, we investigate the effect of the addressing mode on the instruction shown in Figure.
- The instruction in addresses 250 and 251 is “load to ACC,” with the address field ADRS (or an operand NBR) equal to 500.
- The PC has the number 250 for fetching this instruction.
- The content of a processor register *R*1 is 400, and the *ACC* receives the result after the instruction is executed.
  - In the **direct mode**, the effective address is 500, and the operand to be loaded into the *ACC* is 800.
  - In the **immediate mode**, the operand 500 is loaded into the *ACC*.
  - In the **indirect mode**, the effective address is 800, and the operand is 300.
  - In the **relative mode**, the effective address is 500 252 752, and the operand is 600.
  - In the **index mode**, the effective address is 500 400 900, assuming that *R*1 is the index register.
  - In the **register mode**, the operand is in *R*1, and 400 is loaded into the *ACC*.
  - In the **register-indirect mode**, the effective address is the contents of *R*1, and the operand loaded into the *ACC* is 700.



Summary of Addressing Modes

- Table 9-1 lists the value of the effective address and the operand loaded into the *ACC* for the seven addressing modes.
- The table also shows the operation with a register transfer statement and a symbolic convention for each addressing mode.
- LDA is the symbol for the load-to-accumulator opcode.
- In the direct mode, we use the symbol ADRS for the address part of the instruction.
- The # symbol precedes the operand NBR in the immediate mode.
- The symbol ADRS enclosed in square brackets symbolizes an indirect address, which some compilers or assemblers designate with the symbol @.
- The symbol $ before the address makes the effective address relative to the *PC*.
- An index-mode instruction is recognized by the symbol of a register placed in parentheses after the address symbol.
- The register mode is indicated by giving the name of the processor register following LDA.
- In the register-indirect mode, the name of the register that holds the effective address is enclosed in parentheses.

TABLE 9-1
Symbolic Convention for Addressing Modes

			Refers to Figure 9-6	
Addressing Mode	**Symbolic Convention**	**Register Transfer**	**Effective Address**	**Contents of *ACC***
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register-indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

Instruction Set Architectures

- Computers provide a set of instructions to permit computational tasks to be carried out.
- The instruction sets of different computers differ in several ways from each other.
- For example, the binary code assigned to the opcode field varies widely for different computers.
- Likewise, although a standard exists, the symbolic name given to instructions varies for different computers.
- In comparison to these minor differences, however, there are two major types of instruction set architectures that differ markedly in the relationship of hardware to software:
  - ***Complex instruction set computers*** (**CISC**s) provide hardware support for highlevel language operations and have compact programs;
  - ***Reduced instruction set computers*** (**RISC**s) emphasize simple instructions and flexibility that, when combined, provide higher throughput and faster execution.
- These two architectures can be distinguished by considering the properties that characterize their instruction sets.

RISC Architecture

- A ***RISC architecture*** has the following properties:
  1. Memory accesses are restricted to load and store instructions, and datamanipulation instructions are register-to-register.
  2. Addressing modes are limited in number.
  3. Instruction formats are all of the same length.
  4. Instructions perform elementary operations.

- The goal of a RISC architecture is high throughput and fast execution.
- To achieve these goals, accesses to memory, which typically take longer than other elementary operations, are to be avoided, except for fetching instructions.
- A result of this view is the need for a relatively large register file.
- Because of the fixed instruction length, limited addressing modes, and elementary operations, the control unit of a RISC is comparatively simple and is typically hardwired.
- In addition, the underlying organization is universally a pipelined design.

CISC Architecture

- A purely ***CISC architecture*** has the following properties:
  1. Memory access is directly available to most types of instructions.
  2. Addressing modes are substantial in number.
  3. Instruction formats are of different lengths.
  4. Instructions perform both elementary and complex operations.

- The goal of the CISC architecture is to match more closely the operations used in programming languages and to provide instructions that facilitate compact programs and conserve memory.
- In addition, efficiencies in performance may result through a reduction in the number of instruction fetches from memory, compared with the number of elementary operations performed.
- Because of the high memory accessibility, the register files in a CISC may be smaller than in a RISC.
- Also, because of the complexity of the instructions and the variability of the instruction formats, microprogrammed control is more likely to be used.
- In the quest for speed, the microprogrammed control in newer designs is likely to be controlling a pipelined datapath. CISC instructions are converted to a sequence of RISC-like operations that are processed by the RISC-like pipeline.

Instruction Set Architectures

- Actual instruction set architectures range between those which are purely RISC and those which are purely CISC.
- Nevertheless, there is a basic set of elementary operations that most computers include among their instructions.
- Most elementary computer instructions can be classified into three major categories:
  - data-transfer instructions,
  - data-manipulation instructions, and
  - program-control instructions.
- Data-transfer instructions cause transfer of data from one location to another without changing the binary information content.
- Data-manipulation instructions perform arithmetic, logic, and shift operations.
- Program-control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.
- In addition to the basic instruction set, a computer may have other instructions that provide special operations for particular applications.