DIGITAL DESIGN AND COMPUTER ORGANIZATION
BCS302

Module 3
Basic Structure of Computers

By,

Dr. Ashwini N
Assistant Professor
Dept. of Information Science & Engineering
BMS Institute of Technology, Bengaluru.

MODULE 3 :

- BASIC STRUCTURE OF COMPUTERS
- MACHINE INSTRUCTIONS AND PROGRAMS

The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are universal

Functional Units

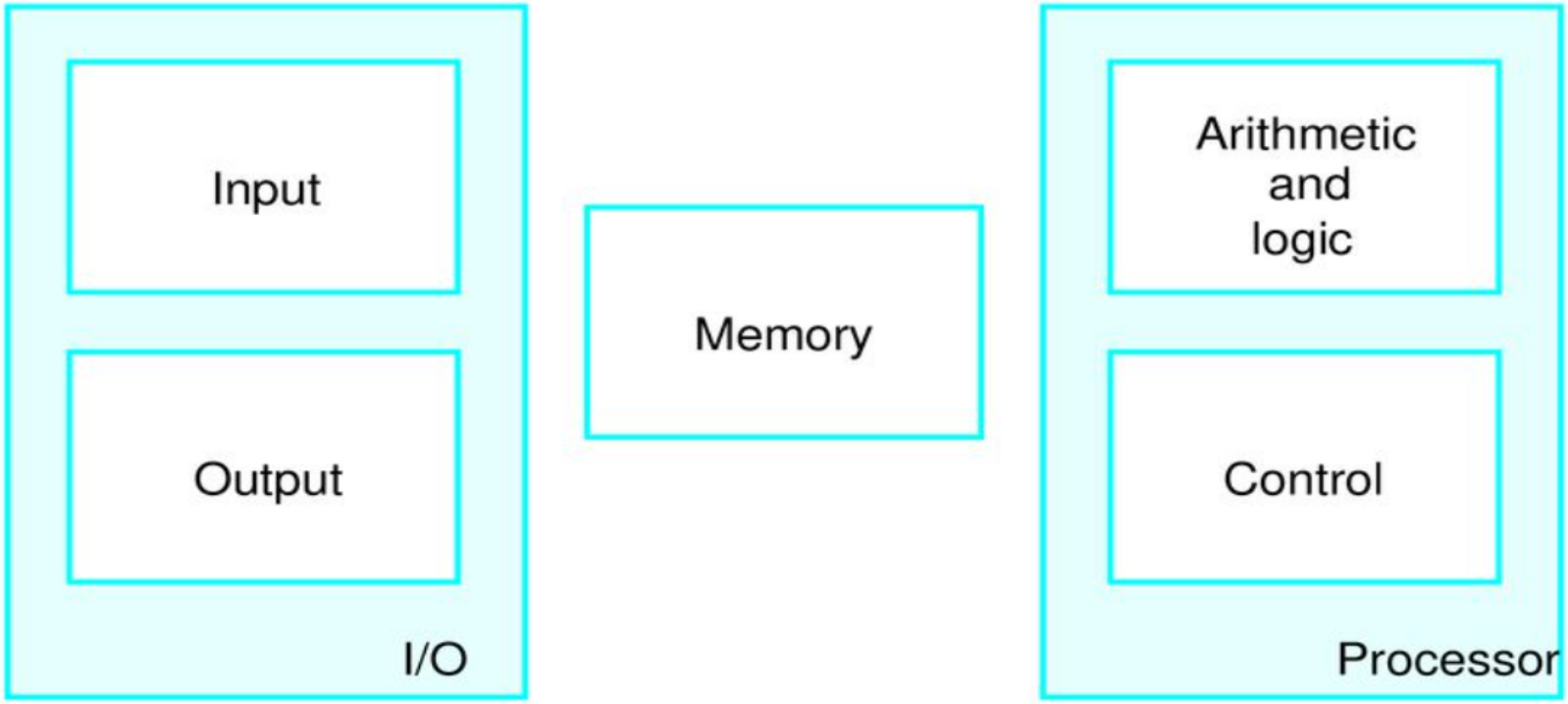


Figure 1.1. Basic functional units of a computer.

Information Handled by a Computer

- Instructions/machine instructions
  - Govern the transfer of information within a computer as well as between the computer and its I/O devices
  - Specify the arithmetic and logic operations to be performed
  - Program
- Data
  - Used as operands by the instructions
  - Source program
- Encoded in binary code – 0 and 1

Memory Unit

- Store programs and data
- Two classes of storage
  - Primary storage
    - Fast
    - Programs must be stored in memory while they are being executed
    - Large number of semiconductor storage cells
    - Processed in words
    - Address
    - RAM and memory access time
    - Memory hierarchy – cache, main memory
  - Secondary storage – larger and cheaper

Arithmetic and Logic Unit (ALU)

- Most computer operations are executed in ALU of the processor.
- Load the operands into memory – bring them to the processor – perform operation in ALU – store the result back to memory or retain in the processor.
- Registers
- Fast control of ALU

Control Unit

- All computer operations are controlled by the control unit.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Control unit is usually distributed throughout the machine instead of standing alone.
- Operations of a computer:
  - Accept information in the form of programs and data through an input unit and store it in the memory
  - Fetch the information stored in the memory, under program control, into an ALU, where the information is processed
  - Output the processed information through an output unit
  - Control all activities inside the machine through a control unit

Functional Units of Computer

Operation	Description
Take Input	➢ The process of entering data and instructions into the computer system
Store Data	➢ Saving data and instructions so that they are available for processing as and when required.
Processing Data	➢ Performing arithmetic, and logical operations on data in order to convert them into useful information.
Output Information	➢ The process of producing useful information or results for the user, such as a printed report or visual display.
Control the workflow	➢ Directs the manner and sequence in which all of the above operations are performed.

Basic Operational Concepts

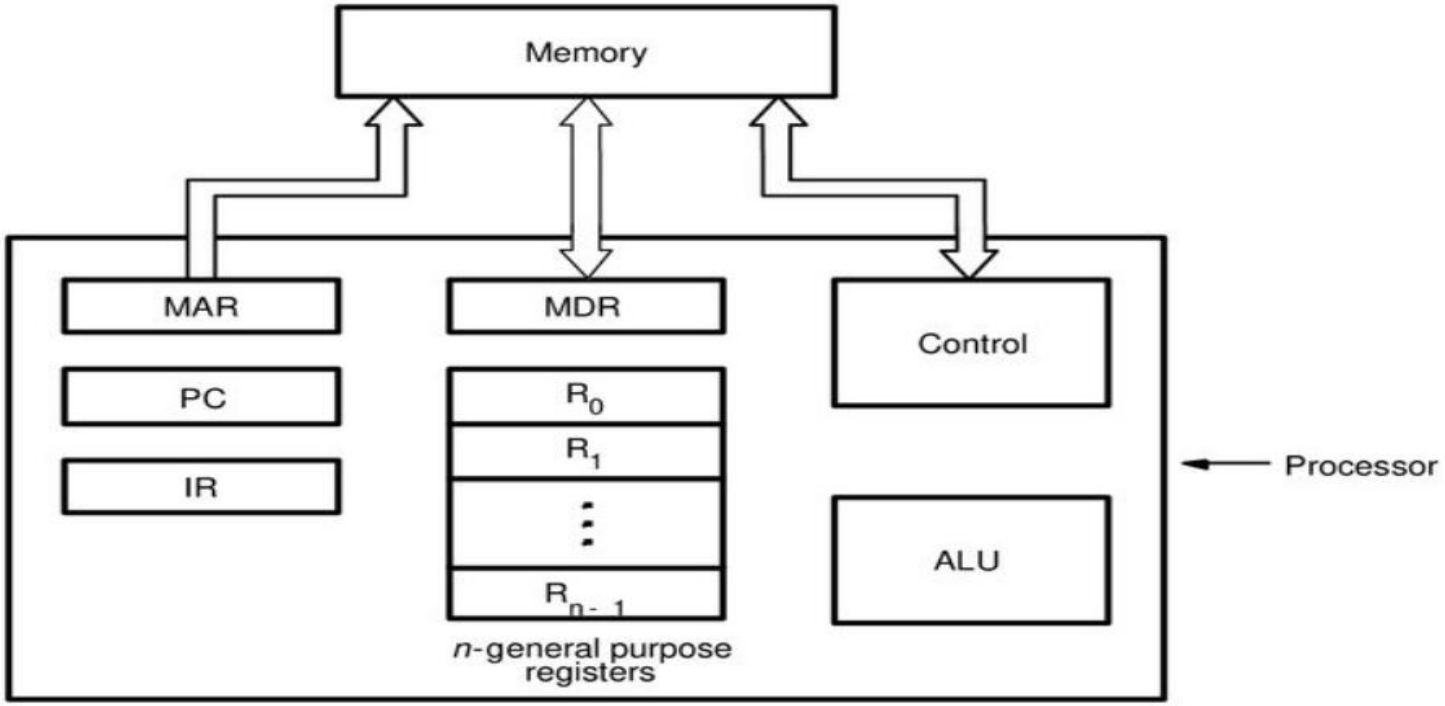
A Typical Instruction

- Add LOCA, R0
- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

Separate Memory Access and ALU Operation

- Load LOCA, R1
- Add R1, R0
- Whose contents will be overwritten?

Connection Between the Processor and the Memory



Connections between the processor and the memory.

Registers

- Instruction register (IR)
- Program counter (PC)
- General-purpose register ($R_0 - R_{n-1}$)
- Memory address register (MAR)
- Memory data register (MDR)

Typical Operating Steps

- Programs reside in the memory through input devices
- PC is set to point to the first instruction
- The contents of PC are transferred to MAR
- A Read signal is sent to the memory
- The first instruction is read out and loaded into MDR
- The contents of MDR are transferred to IR
- Decode and execute the instruction

Typical Operating Steps (Cont')

- Get operands for ALU
  - General-purpose register
  - Memory (address to MAR – Read – MDR to ALU)
- Perform operation in ALU
- Store the result back
  - To general-purpose register
  - To memory (address to MAR, result to MDR – Write)
- During the execution, PC is incremented to the next instruction

Interrupt

- Normal execution of programs may be preempted if some device requires urgent servicing.
- The normal execution of the current program must be interrupted – the device raises an *interrupt* signal.
- Interrupt-service routine
- Current system information backup and restore (PC, general-purpose registers, control information, specific information)

Bus Structures

- There are many ways to connect different parts inside a computer together.
- A group of lines that serves as a connecting path for several devices is called a *bus*.
- Address/data/control

Bus Structure

- Single-bus

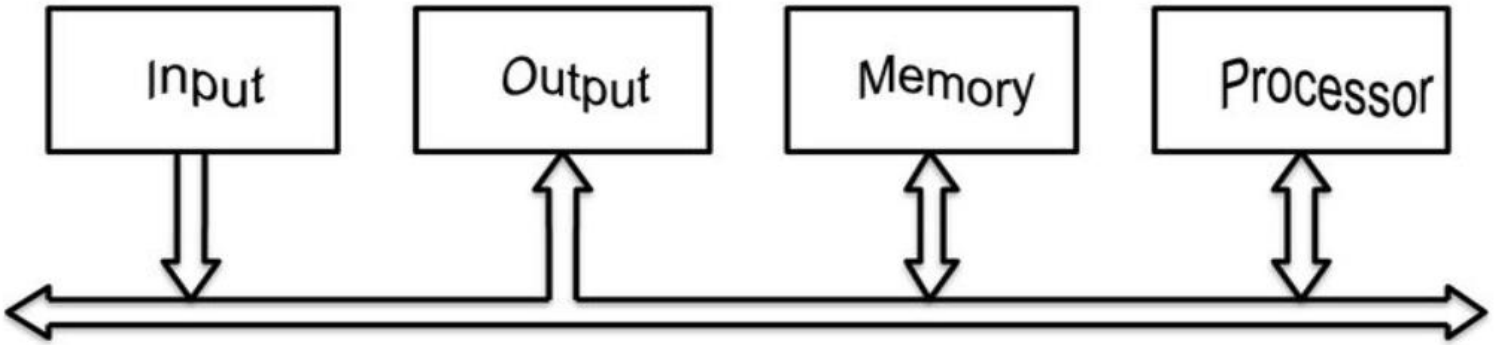


Figure 1.3. Single-bus structure.

- Multiple Buses

Speed Issue

- Different devices have different transfer/operate speed.
- If the speed of bus is bounded by the slowest device connected to it, the efficiency will be very low.
- How to solve this?
- A common approach – use buffers.

  e.g.- Printing the characters

Performance

Performance

- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
  - Hardware design
  - Instruction set
  - Compiler

Performance

- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions.

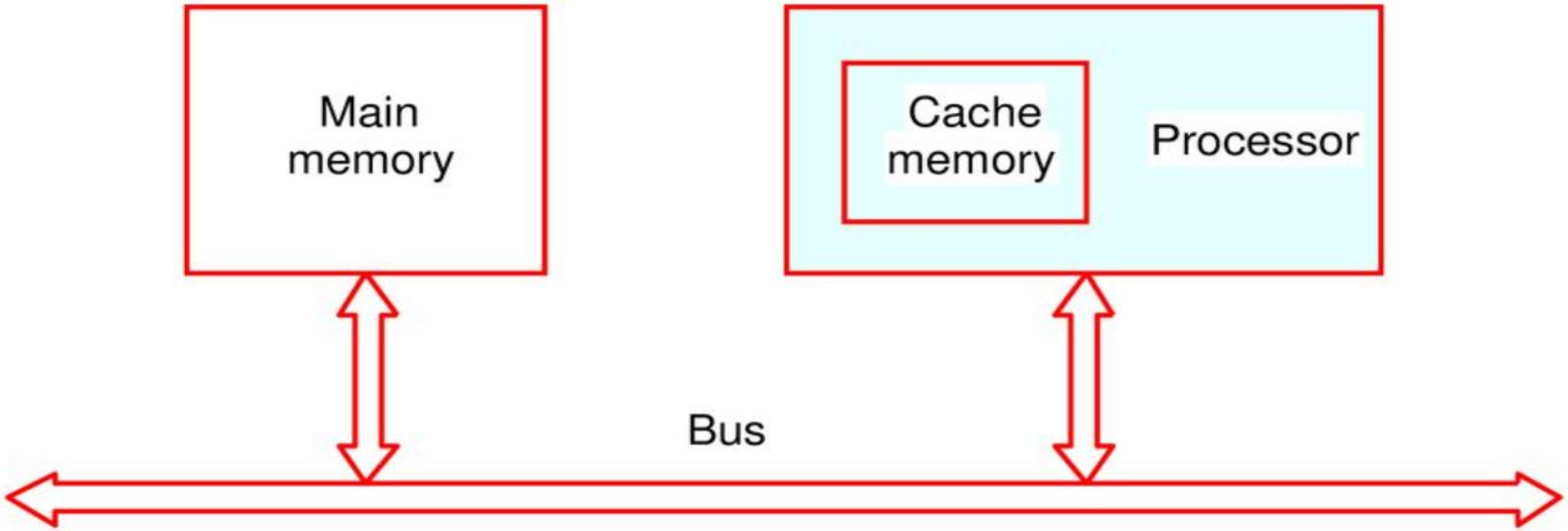


Figure 1.5. The processor cache.

Performance

- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip.
  - Speed
  - Cost
  - Memory management

Processor Clock

- Clock, clock cycle (P), and clock rate (R=1/P)
- The execution of each instruction is divided into several steps (Basic Steps), each of which completes in one clock cycle.
- Hertz – cycles per second

Basic Performance Equation

- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- Note: these are not independent to each other

$$T = \frac{N \times S}{R}$$

- How to improve T?
- Reduce N and S, Increase R, but these affect one another

Clock Rate

- Increase clock rate
  - Improve the integrated-circuit (IC) technology to make the circuits faster
  - Reduce the amount of processing done in one basic step (however, this may increase the number of basic steps needed)
- Increases in R that are entirely caused by improvements in IC technology affect all aspects of the processor's operation equally except the time to access the main memory.

Performance Measurement

- T is difficult to compute.
- Measure computer performance using benchmark programs.
- System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
- Compile and run (no simulation)
- Reference computer

$$SPEC\ rating = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

$$SPEC\ rating = \left(\prod_{i=1}^{n} SPEC_i\right)^{\frac{1}{n}}$$

- n is the number of program in the suite

Machine Instructions and Programs

Objectives

- Machine instructions and program execution, including branching and subroutine call and return operations.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

Memory Locations, Addresses, and Operations

Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.

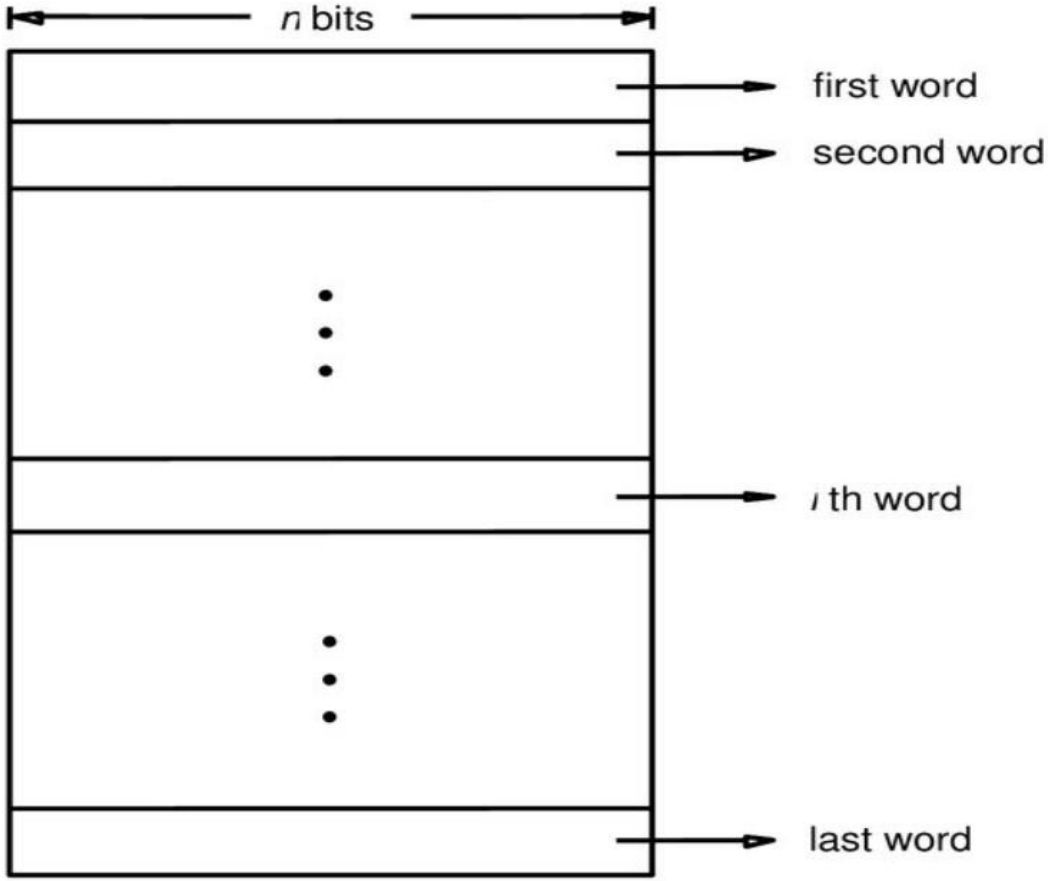
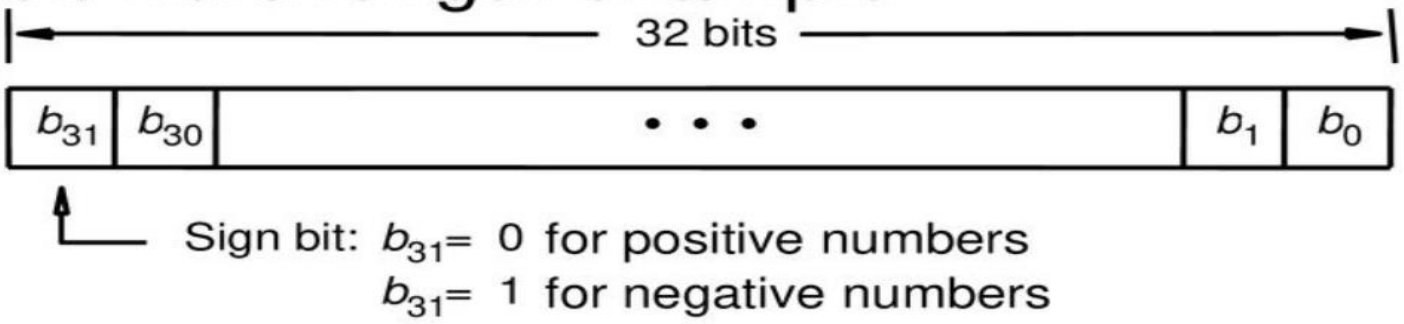


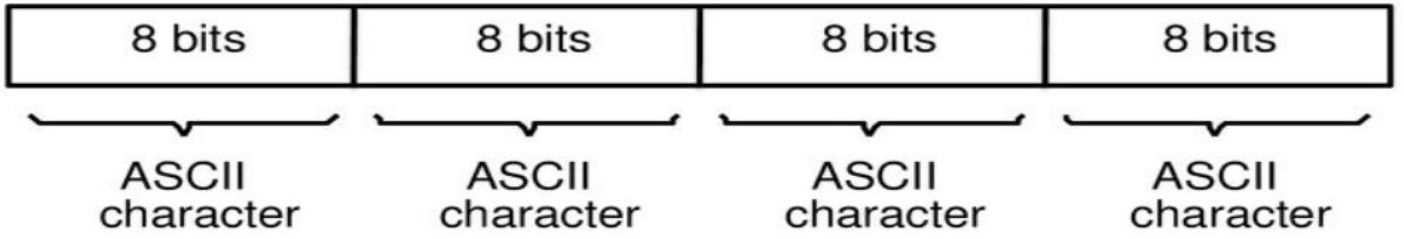
Fig: Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example



(a) A signed integer



(b) Four characters

Memory Location, Addresses, and Operation

- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A *k*-bit address memory has 2^k memory locations, namely 0 – 2^k-1, called memory space.
- 24-bit memory: 2^{24} = 16,777,216 = 16M (1M=2^{20})
- 32-bit memory: 2^{32} = 4G (1G=2^{30})
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}

Memory Location, Addresses, and Operation

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, ... If word length is 32 bits, they successive words are located at addresses 0, 4, 8,...

Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word

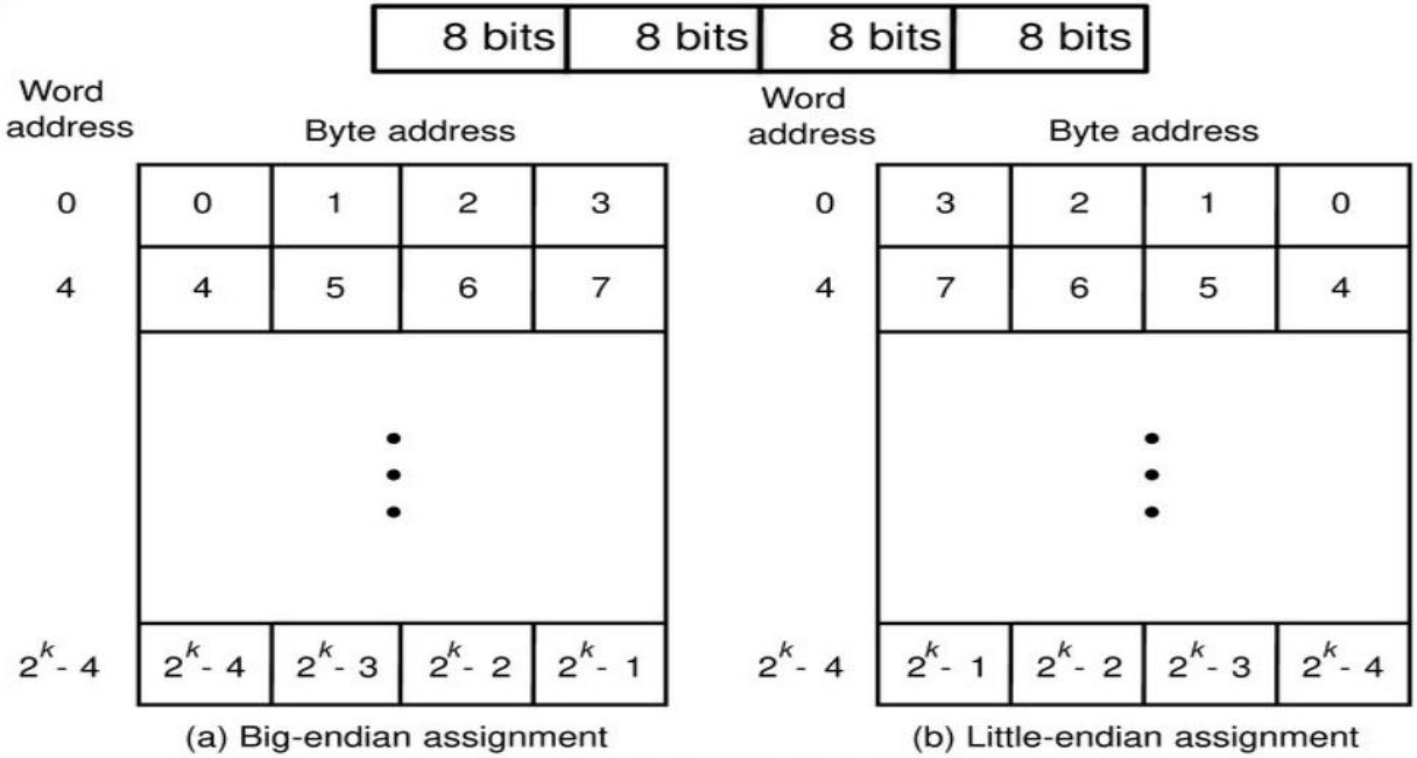


Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,....
    - 32-bit word: word addresses: 0, 4, 8,....
    - 64-bit word: word addresses: 0, 8,16,....
- Access numbers, characters, and character strings

Memory Operation

- Load (or Read or Fetch)
  - Copy the content. The memory content doesn't change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

Instruction and Instruction Sequencing

"Must-Perform" Operations

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation

- Represent machine instructions and programs.
- Move LOC, R1 => R1←[LOC]
- Add R1, R2, R3 => R3 ←[R1]+[R2]

CPU Organization

- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats

- Three-Address Instructions
  - ADD R2, R3, R1 — R1 ← [R2] + [R]3
- Two-Address Instructions
  - ADD R2, R1 — R1 ← [R1] + [R2]
- One-Address Instructions
  - ADD M — AC ← [AC] + M[AR]
- Zero-Address Instructions
  - ADD — TOS ← [TOS] + [TOS – 1]
- RISC Instructions
  - Lots of registers. Memory is restricted to Load & Store

Instruction Formats

Example: Evaluate (A+B) * (C+D)

- Three-Address

1. ADD A, B, R1 ; R1 ← M[A] + M[B]
2. ADD C, D, R2 ; R2 ← M[C] + M[D]
3. MUL R1, R2, X ; M[X] ← [R1] * [R2]

Instruction Formats

Example: Evaluate (A+B) * (C+D)

- Two-Address

1.	MOV	A, R1	; R1 ← M[A]
2.	ADD	B, R1	; R1 ← [R1] + M[B]
3.	MOV	C, R2	; R2 ← M[C]
4.	ADD	D, R2	; R2 ← [R2] + M[D]
5.	MUL	R2, R1	; R1 ← [R1] * [R2]
6.	MOV	R1, X	; M[X] ← [R1]

Instruction Formats

Example: Evaluate (A+B) * (C+D)

- One-Address

1.	LOAD A	; AC ← M[A]
2.	ADD B	; AC ← [AC] + M[B]
3.	STORE T	; M[T] ← [AC]
4.	LOAD C	; AC ← M[C]
5.	ADD D	; AC ← [AC] + M[D]
6.	MUL T	; AC ← [AC] * M[T]
7.	STORE X	; M[X] ← [AC]

Instruction Formats

Example: Evaluate (A+B) * (C+D)

- Zero-Address

1.	PUSH A	; TOS ← [A]
2.	PUSH B	; TOS ← [B]
3.	ADD	; TOS ← [A + B]
4.	PUSH C	; TOS ← [C]
5.	PUSH D	; TOS ← [D]
6.	ADD	; TOS ← [C + D]
7.	MUL	; TOS ← [C+D]*[A+B]
8.	POP X	; M[X] ← [TOS]

Instruction Formats

Example: Evaluate (A+B) * (C+D)

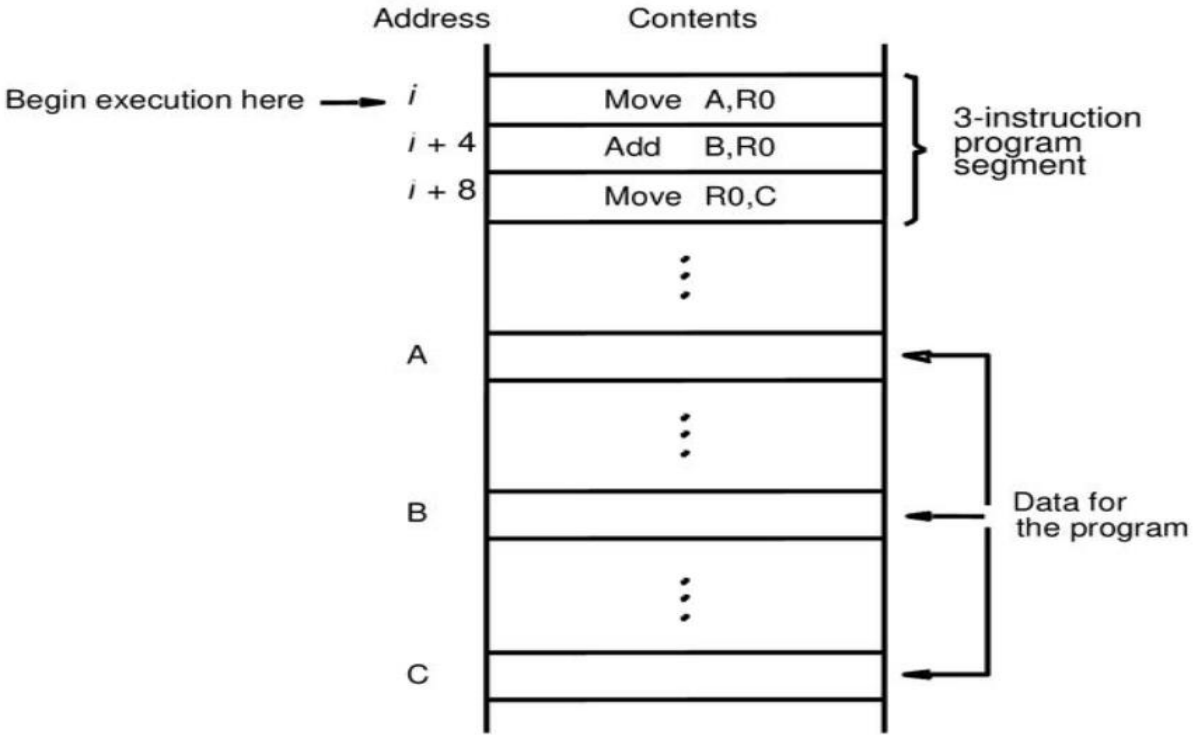
- RISC

```
1. LOAD   A, R1          ; R1 ← M[A]
2. LOAD   B, R2          ; R2 ← M[B]
3. LOAD   C, R3          ; R3 ← M[C]
4. LOAD   D, R4          ; R4 ← M[D]
5. ADD    R1, R2, R1     ; R1 ← [R1] + [R2]
6. ADD    R3, R4, R3     ; R3 ← [R3] + [R4]
7. MUL    R1, R3, R1     ; R1 ← [R1] * [R3]
8. STOREX, R1            ; M[X] ← [R1]
```

Using Registers

- Registers are faster
- Shorter instructions
  - The number of registers is smaller (e.g. 32 registers need 5 bits)
- Potential speedup
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing



Assumptions:
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure
-Instruction fetch
-Instruction execute

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

Branching

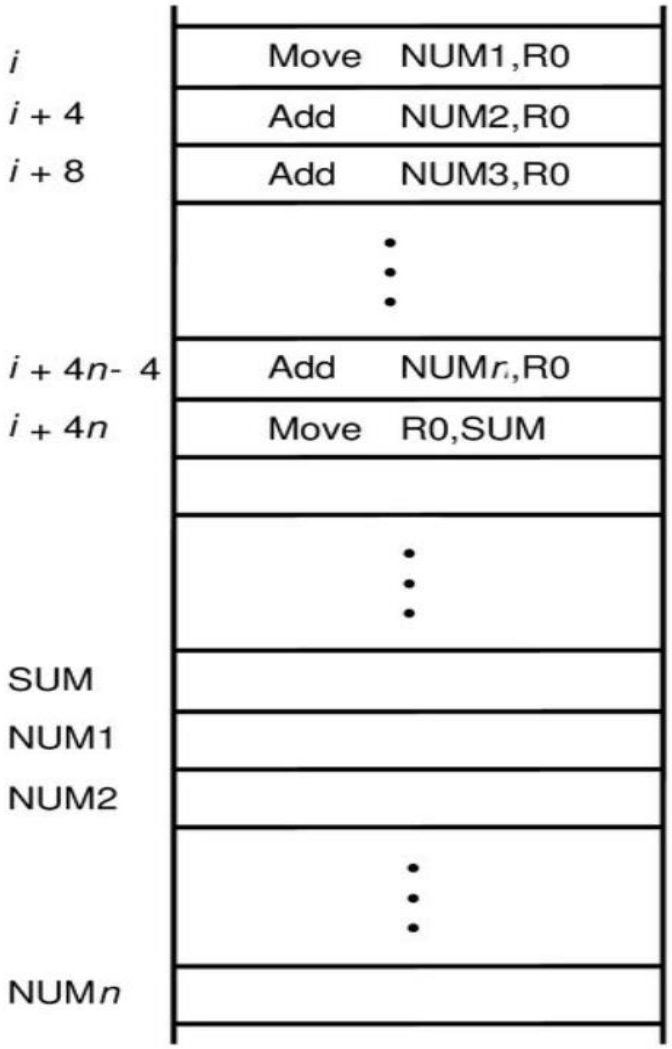


Figure 2.9. A straight-line program for adding *n* numbers.

Branching

Branch target

Conditional branch

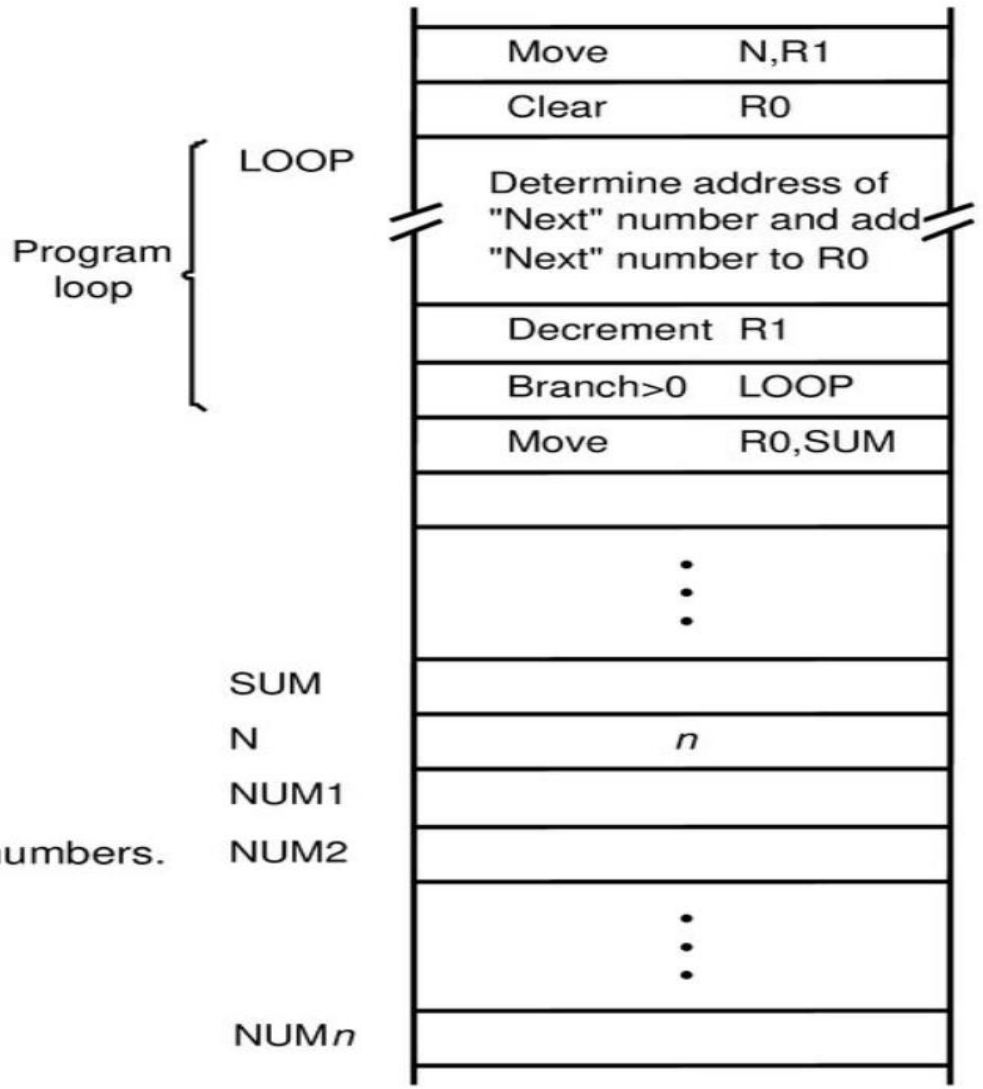


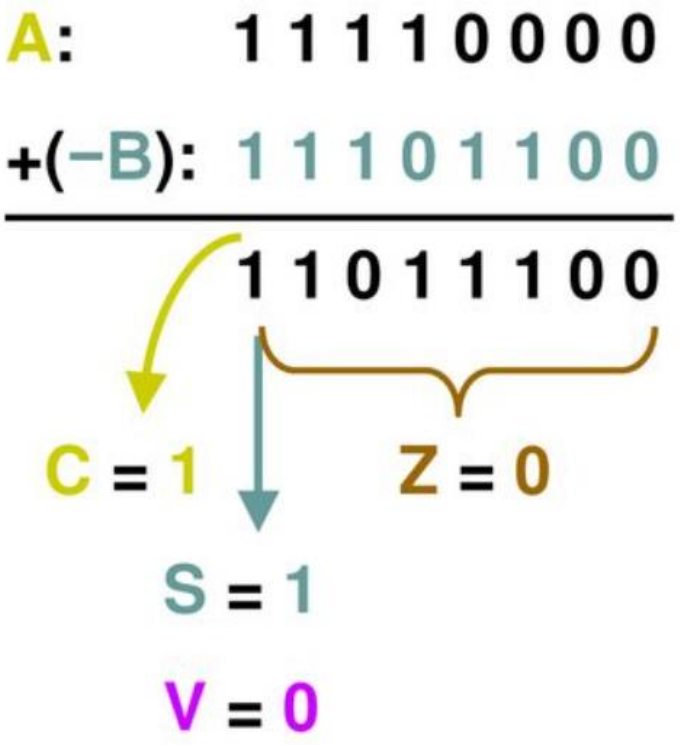
Figure 2.10. Using a loop to add *n* numbers.

Condition Codes

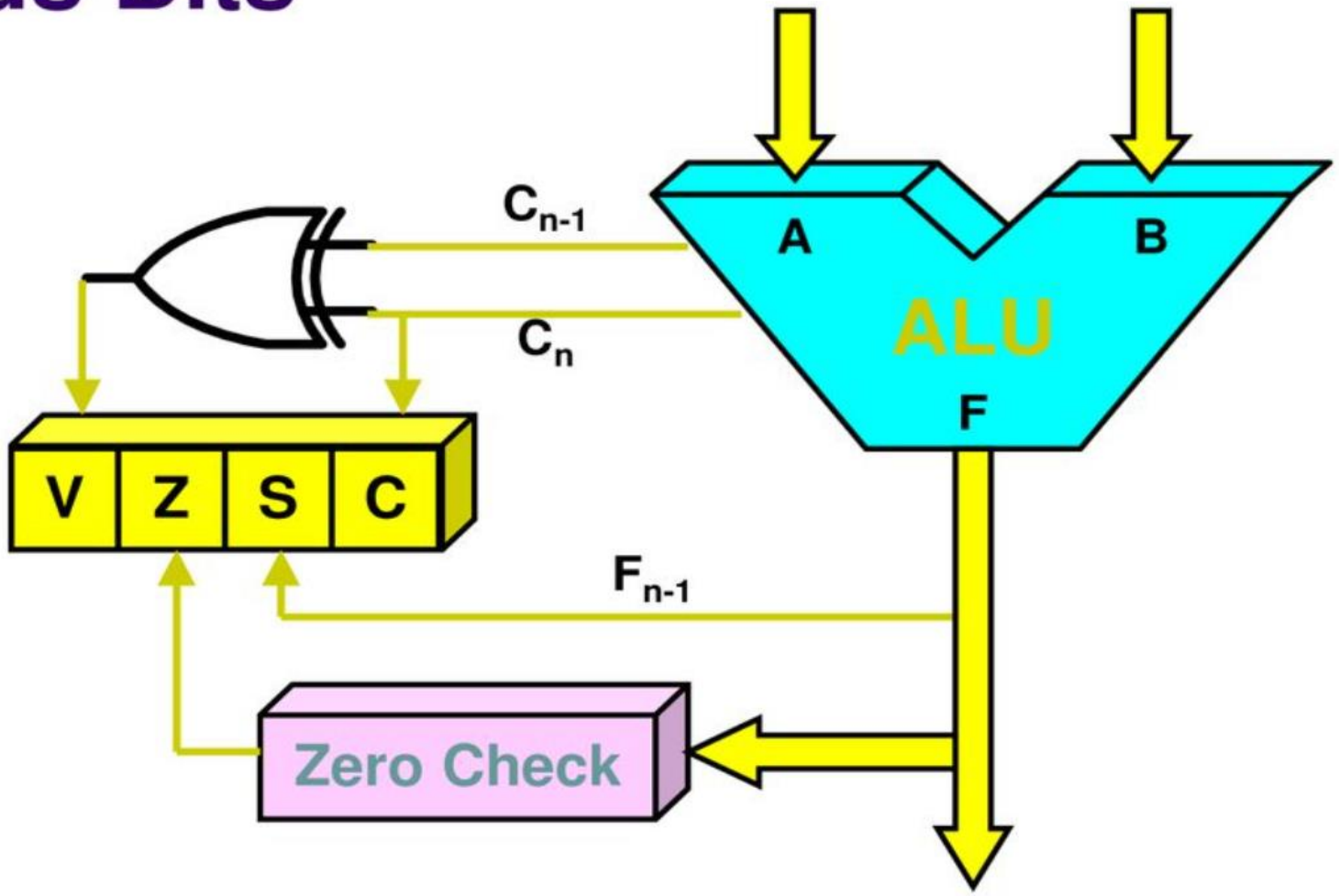
- Condition code flags (bits)
- Condition code register / status register
  - N (negative)
  - Z (zero)
  - V (overflow)
  - C (carry)
- Different instructions affect different flags

Conditional Branch Instructions

- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



Status Bits



Addressing Modes

Generating Memory Addresses

- How to specify the address of branch target?
- Can we give the memory operand address directly in a single Add instruction in the loop?
- Use a register to hold the address of NUM1; then increment by 4 on each pass through the loop.

Addressing Modes

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

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Effective Address (EA)

- In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed.
- The effective address is then used to access the operand.

Addressing Modes

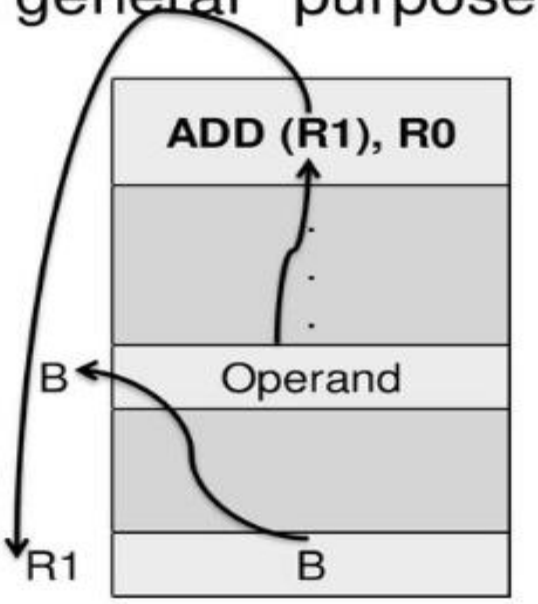
- Implied
  - AC is implied in "ADD M[AR]" in "One-Address" instr.
  - TOS is implied in "ADD" in "Zero-Address" instr.
- Immediate
  - The use of a constant in "MOV 5, R1"
    or "MOV #5, R1" i.e. R1 ← 5

  MOV #NUM1, R2 ; to copy the variable memory address
- Register
  - Indicate which register holds the operand
- Direct Address
  - Use the given address to access a memory location
  - E.g. Move NUM1, R1
  - Move R0, SUM

Addressing Modes
Indirect Addressing

- Indirect Addressing
  - Indirection and Pointer
  - Indirect addressing through a general purpose register.
  - Indicate the register (e.g. R1) that holds the address of the variable (e.g. B) that holds the operand

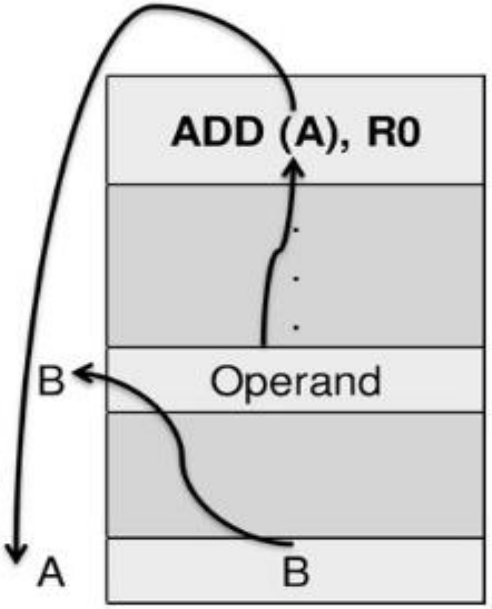
    ADD (R1), R0
  - The register or memory location that contain the address of an operand is called a pointer



Addressing Modes
Indirect Addressing

- Indirect Addressing
  - Indirect addressing through a memory addressing.
  - Indicate the memory variable (e.g. A)that holds the address of the variable (e.g. B) that holds the operand

    ADD (A), R0



Indirect Addressing Example

- Addition of N numbers

```
1.          Move N,R1           ; N = Numbers to add
2.          Move #NUM1,R2       ; R2= Address of 1st no.
3.          Clear R0            ; R0 = 00
4.  Loop :  Add (R2), R0        ; R0 = [NUM1] + [R0]
5.          Add #4, R2      ; R2= To point to the  next
                            ; number
6.          Decrement R1        ; R1 = [R1] -1
7.          Branch>0 Loop       ; Check if R1>0 or not if
                                ; yes go to Loop
8.          Move R0, SUM        ; SUM= Sum of all no.
```

Example

- Addition of N numbers

```
1.          Move N,R1           ; N = 5
2.          Move #NUM1,R2       ; R2= 10000H
3.          Clear R0            ; R0 = 00
4.  Loop :  Add (R2), R0        ; R0 = 10 + 00 = 10
5.          Add #4, R2          ; R2 = 10004H
6.          Decrement R1        ; R1 = 4
7.          Branch>0 Loop       ; Check if R1>0 if
                                ; yes go to Loop
8.          Move R0, SUM        ; SUM=
```

Example

- Addition of N numbers

```
1.          Move N,R1           ; N = 5
2.          Move #NUM1,R2       ; R2= 10000H
3.          Clear R0            ; R0 = 00
4.  Loop :  Add (R2), R0        ; R0 = 20 + 10 = 30
5.          Add #4, R2          ; R2 = 10008H
6.          Decrement R1        ; R1 = 3
7.          Branch>0 Loop       ; Check if R1>0 if
                                ; yes go to Loop
8.          Move R0, SUM        ; SUM=
```

Example

- Addition of N numbers

```
1.          Move N,R1           ; N = 5
2.          Move #NUM1,R2       ; R2= 10000H
3.          Clear R0            ; R0 = 00
4.  Loop :  Add (R2), R0        ; R0 = 30 + 30 = 60
5.          Add #4, R2          ; R2 = 1000CH
6.          Decrement R1        ; R1 = 2
7.          Branch>0 Loop       ; Check if R1>0 if
                                ; yes go to Loop
8.          Move R0, SUM        ; SUM=
```

Example

- Addition of N numbers

```
1.          Move N,R1          ; N = 5
2.          Move #NUM1,R2      ; R2= 10000H
3.          Clear R0           ; R0 = 00
4.  Loop :  Add (R2), R0       ; R0 = 40 + 60 = 100
5.          Add #4, R2         ; R2 = 10010H
6.          Decrement R1       ; R1 = 1
7.          Branch>0 Loop      ; Check if R1>0 if
                               ; yes go to Loop
8.          Move R0, SUM       ; SUM=
```

Example

- Addition of N numbers

```
1.          Move N,R1           ; N = 5
2.          Move #NUM1,R2       ; R2= 10000H
3.          Clear R0            ; R0 = 00
4.  Loop :  Add (R2), R0        ; R0 = 50 + 100 = 150
5.          Add #4, R2          ; R2 = 10014H
6.          Decrement R1        ; R1 = 0
7.          Branch>0 Loop       ; Check if R1>0 if
                                ; yes go to Loop
8.          Move R0, SUM        ; SUM =
```

Example

- Addition of N numbers

```
1.          Move N,R1          ; N = 5
2.          Move #NUM1,R2      ; R2= 10000H
3.          Clear R0           ; R0 = 00
4.  Loop :  Add (R2), R0       ; R0 = 50 + 100 = 150
5.          Add #4, R2         ; R2 = 10014H
6.          Decrement R1       ; R1 = 0
7.          Branch>0 Loop      ; Check if R1>0 if
                               ; yes go to Loop
8.          Move R0, SUM       ; SUM = 150
```

Addressing Modes
Indexing and Arrays

- Indexing and Array
- The EA of the operand is generated by adding a constant value to the contents of a register.
- X(Ri) ; EA= X + (Ri) X= Signed number
- X defined as offset or displacement

Addressing Modes
Indexing and Arrays

- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register
- $X(R_i)$: $EA = X + [R_i]$
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.

Addressing Modes
Indexing and Arrays

- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
- 2D Array
  - (Ri, Rj) so EA = [Ri] + [Rj]
  - Rj is called the base register
- 3D Array
  - X(Ri, Rj) so EA = X + [Ri] + [Rj]

Addressing Modes
Indexing and Arrays

Address	Memory
	Add 20(R1), R2
	. . . .
10000H	
Offset=20	. . . .
10020H	Operand

R1	10000H

Offset is given as a Constant

Address	Memory
	Add 10000H(R1), R2
	. . . .
10000H	
Offset=20	. . . .
10020H	Operand

R1	20H

Offset is in the index register

Addressing Modes Indexing and Arrays

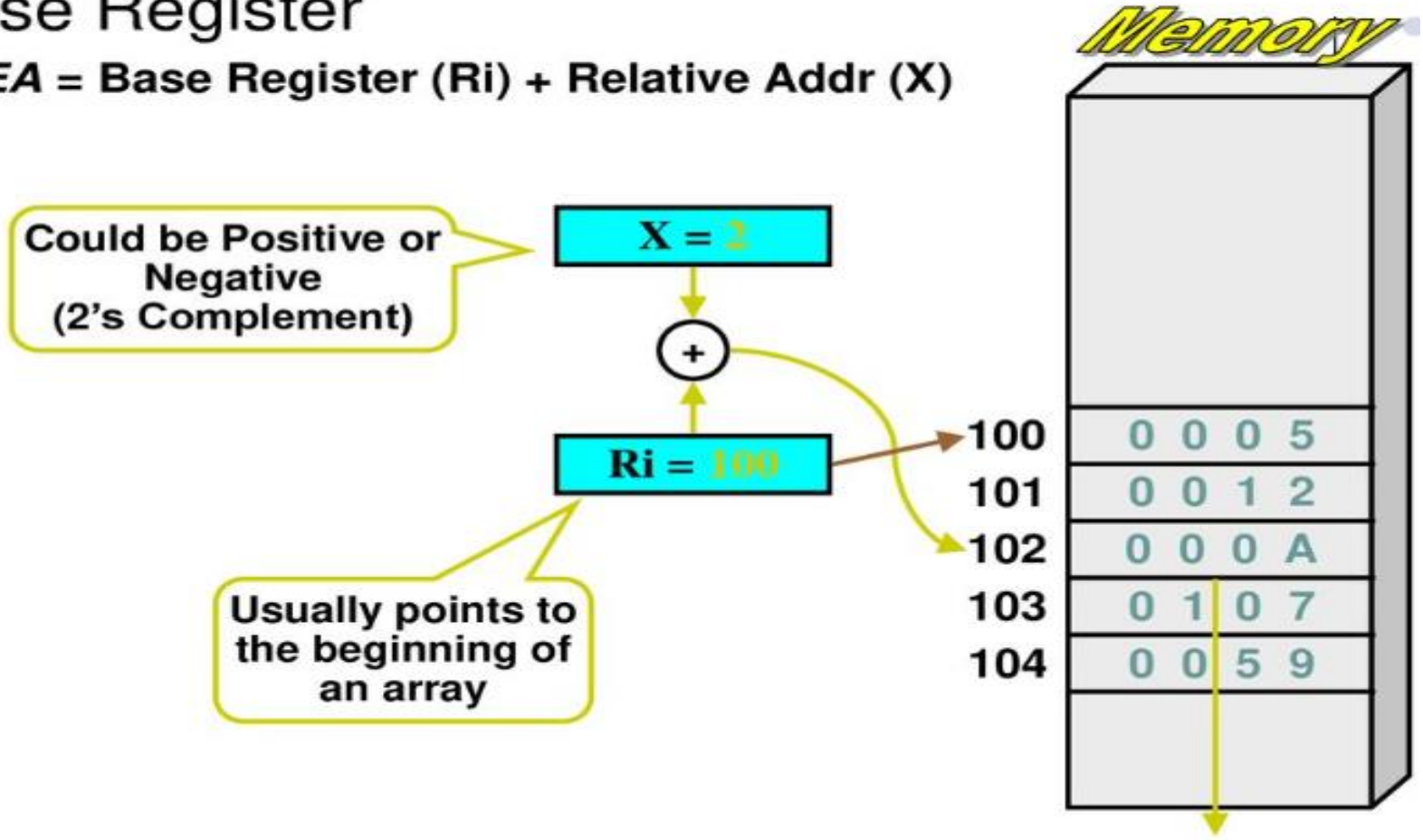
- Array
- E.g. List of students marks

Address	Memory	Comments
N	n	No. of students
LIST	Student ID1	Student 1
LIST+4	Test 1	
LIST+8	Test 2	
LIST+12	Test 3	
LIST+16	Student ID2	Student 2
LIST+20	Test 1	
LIST+24	Test 2	
LIST+28	Test 3	

- Indexed addressing used in accessing test marks from the list

Addressing Modes

- Base Register
  - *EA* = Base Register (Ri) + Relative Addr (X)



Addressing Modes
Indexing and Arrays

- Program to find the sum of marks of all subjects of reach students and store it in memory.

```
1.                  Move #LIST, R0
2.                  Clear R1
3.                  Clear R2
4.                  Move #SUM, R2
5.                  Move N, R4
6.      Loop :      Add 4(R0), R1
7.                  Add 8(R0), R1
8.                  Add 12(R0),R1
9.                  Move R1, (R2)
10.                 Clear R1
11.                 Add #16, R0
12.                 Add #4, R2
13.                 Decrement R4
14.                 Branch>0 Loop
```

Addressing Modes
Relative Addressing

- Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
- X(PC) – note that X is a signed number
- Branch>0 LOOP
- This location is computed by specifying it as an offset from the current value of PC.
- Branch target may be either before or after the branch instruction, the offset is given as a singed num.

Addressing Modes
Additional Modes

- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- $(R_i)+$. The increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.
- Autodecrement mode: $-(R_i)$ – decrement first

```
                Move        N,R1          }
                Move        #NUM1,R2      } Initialization
                Clear       R0            }
LOOP            Add         (R2)+,R0
                Decrement   R1
                Branch>0    LOOP
                Move        R0,SUM
```

Figure 2.16. The Autoincrement addressing mode used in the program of Figure 2.12.


Thank you!