COMPUTER ORGANIZATION AND ARCHITECTURE

Text Book – Carl Hamacher

Unit –I ; Chapter -1

BASIC STRUCTURE OF COMPUTERS

COMPUTER TYPES

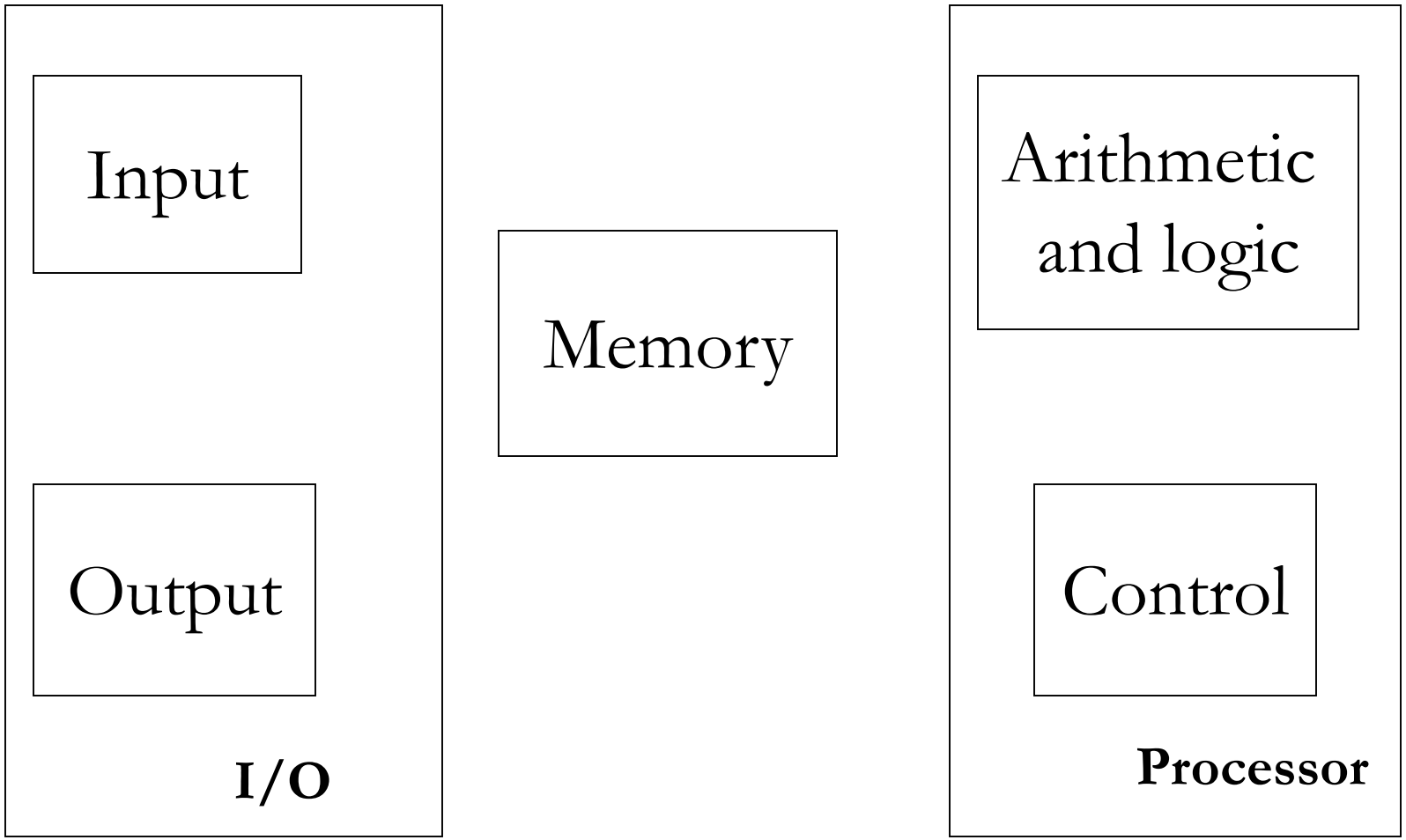
- Computer is a fast electronic calculating machine that accepts digitized **input** information, **processes** it according to the list of **stored** instructions, and produces the resulting **output** information.

- Computers are classified based on the parameters like
  - Speed of operation
  - Cost
  - Computational power
  - Type of application

Computer types..

- PERSONAL or DESK TOP COMPUTERS
  - Processing & storage units, visual display & audio units, keyboards
  - Storage media-Hard disks, CD-ROMs
  - Eg: Personal computers used in homes and offices
  - Advantage: Cost effective, easy to operate, suitable for general purpose educational or business application
- NOTEBOOK COMPUTERS
  - Compact form of personal computer (laptop)
  - Advantage is portability
- WORK STATIONS
  - More computational power than PC
  - Costlier
  - Used to solve complex problems which arises in engineering application (graphics, CAD/CAM etc)
- SERVERS and SUPERCOMPUTERS
  - Servers – low end systems Supercomputers –High end Systems
  - Servers- have sizeable database storage and capable of handling large no. of requests to access data.
  - Supercomputers –capable of large-scale numerical calculations required in applications such as weather forecasting, and aircraft design and simulation.

Functional Units



Input Unit

- Input Unit reads the data
- The most common Input devices are Keyboard, joystick, trackballs, microphone and mouse

Output Unit

- Counterpart of Input unit
- Its function is to send processed results to outside world
- The familiar example of output device is printer (various types)

Memory Unit

- The function of memory unit is to store programs and data
- There are 2 classes of storage:
  - **Primary Storage**:
    - Fast memory that operates at electronic speeds
    - The memory contains a large number of semiconductor storage cells, each capable of storing 1 bit of information
    - These cells are processed in groups of fixed size called **word**
    - The number of bits in each word is known as **word length**
      - Range from 16 to 64 bits
    - To provide easy access to any word in the memory, a distinct ***address*** is associated with each word location
      - Addresses are numbers that identify successive locations

Memory Unit

- Memory in which any location can be reached in a short and fixed amount of time after specifying its address is **RAM**
- Time required to access one word is called the memory **access time**
- Memory of a computer is normally implemented as a memory hierarchy of 3 or 4 levels of semiconductor RAM units with different speeds & sizes
- The small, fast, units are called **caches**
- The largest & slowest unit is referred to as the **main memory**
- Primary storage is expensive

Memory Unit

- **Secondary Storage**:
  - Is used when large amount of data and many programs have to be stored
  - It contains infrequently accessed information
  - Additional & cheaper memory
  - Ex: Magnetic disks and tapes & optical disks (CD-ROMs)

Arithmetic And Logic Unit

- ALU performs all the arithmetic and logic operations
- For ex: addition, multiplication, division, comparison etc
- Any operation is initiated by bringing the required operands into the processor, where the operation is performed by the ALU
- When operands are brought into the processor, they are stored in high- speed storage elements called **registers**

Arithmetic and Logic Unit (ALU)

- Each register can store one word of data
- Access time to registers are faster than cache unit
- CU (control unit) & ALU are many times faster than other devices connected to a computer system
  - This enables a single processor to control a number of external devices such as keyboards, displays, magnetic & optical disks, sensors & mechanical controllers

Control Unit (CU)

- It controls the entire operations of the computer
- The control unit is the nerve centre that sends control signals to other units and senses their states
- The **timing signals** that govern the I/O transfers are generated by **control circuits**
  - Timing signals are signals that determine when a given action is to take place
- Data transfers b/w processor & memory are also controlled by CU through timing signals
- A large set of **control lines** (wires) carries the signals used for timing & synchronization of events in all units

Operation of a Computer - Summarized

- The computer accepts information in the form of pgms & data through an I/P unit & stores it in the memory
- Information stored in the memory is fetched, under pgm control, into an ALU, where it is processed
- Processed information leaves the computer through an O/P unit
- All activities inside the machine are directed by the CU

Information Vs Instructions

- Instructions/ Machine instructions :
  - Are explicit **commands** that
    - Governs transfer of information within the computer as well as between computer and its I/O devices
    - Specify arithmetic and logic operations to be performed
- Data:
  - **Numbers** and **encoded characters** that are used as **operands** by the instructions
  - Any digital information
  - Programs can also be considered as data if it is to be processed by another pgm
    - Ex: Compiling a HLL *source pgm* into machine language pgm, called the *object pgm*

      (*Source pgm* is I/P data to compiler & *object pgm* is O/P data)
  - The **processed data** is called information

Information Vs Instructions

- Information must be **encoded** in a suitable format
- Each number, char, or instruction is encoded as a string of binary digits called bits, each having either 0 or 1
- Encoding Schemes:
  - **BCD** (Binary - Coded Decimal)
    - Each decimal digit is encoded by 4 bits
  - **ASCII** (American standard Code for Information Interchange)
    - Each char is represented as a 7-bit code
  - **EBCDIC** (Extended Binary- Coded Decimal Interchange Code)
    - 8 bits are used to denote a char

Basic Operational Concepts

- For processing, individual instructions are brought from memory into the processor, which executes the specified operations
- Data to be used as operands are also stored in memory
- A typical instruction may be:

  Add LOCA, R0

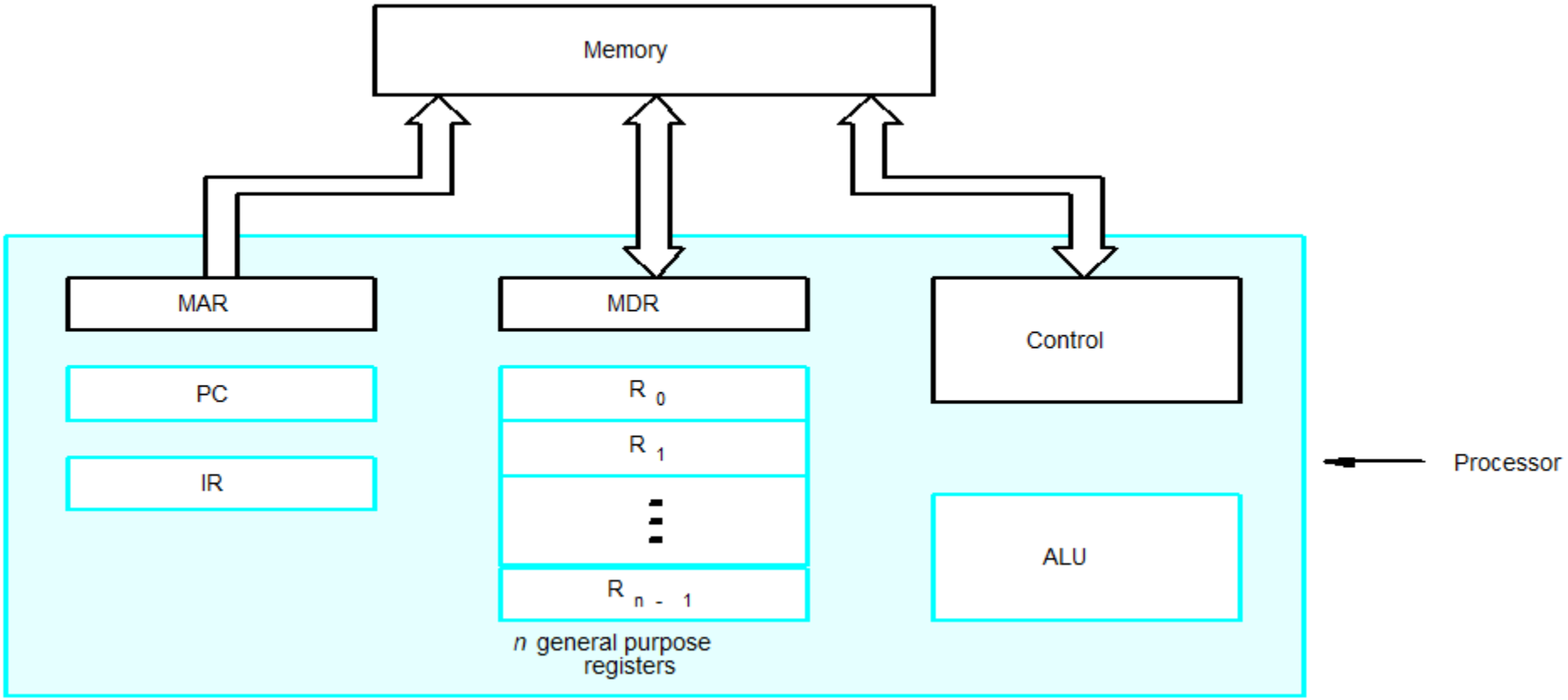
  - Adds the operand at memory location LOCA to the operand in register R0 and store the result in R0 (i.e R0=R0+LOCA)
  - The original content of A is preserved, whereas R0 is overwritten
  - In some computers, the above operation requires two instructions

    Load LOCA, R1

    Add R1,R0

    - The first instruction transfers the contents of LOCA into the processor register R1
    - The second instruction adds the contents of R1 and R0 and places the content in R0

Connection between the processor and memory



MAR - Memory Address Register

PC - Program Counter

MDR - Memory Data Register

IR - Instruction Register

Different parts of a processor

- CPU = ALU + CU + registers
- Diff: types of registers are:
- IR (instruction register):
  - Holds the instruction that is currently being executed
  - Its o/p is available to control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction
- PC (program counter):
  - Keeps track of the execution of a pgm
  - Contains the memory address of next instruction to be fetched & executed
  - During the execution of an instruction, the contents of PC are updated to correspond to the address of next instruction that is to be fetched from the memory
    - PC points to the next instruction that is to be fetched from memory

Different parts of a processor

- n general- purpose registers (R0 thru' Rn-1):
  - are used for holding data, intermediate results of operations.
  - They are also known as **scratch-pad** registers.

- MAR (memory address register):
  - Facilitates communication with memory
  - Holds the address of the location to be accessed
- MDR (memory data register):
  - Facilitates communication with memory
  - Contains data to be written into or read out of the addressed location

Execution of an instruction

- Execution of an instruction by CPU during pgm execution involves the following steps:
  - The CU takes the address of the next instruction to be executed from the PC register & reads instruction from corresponding memory address into the instruction register of CU
  - The CU sends the operation part & address part of instruction to the decoder & MAR respectively
  - The decoder interprets the instruction and accordingly the CU sends signals to the appropriate unit which needs to be involved in carrying out the task specified in the instruction
    - Ex: if it is arithmetic/logical operation, the signal is send to ALU
  - As each instruction is executed, the address of the next instruction to be executed will be automatically loaded into the PC register & steps 1 to 4 are repeated

Steps involving Instruction Fetch & Execution

- Pgms reside in the memory and usually get there through the i/p unit

INSTRUCTION FETCH :

- Execution of a program starts by setting the **PC to point to the first instruction** of the program

- The contents of **PC are transferred to the MAR** and a **Read control signal** is sent to the memory

Steps involving Instruction Fetch & Execution

- The addressed word (here it is the first instruction of the program) is read out of memory and **loaded into the MDR**

- The contents of MDR are **transferred to the IR**
  - Now the instruction is ready to be **decoded & executed**

Steps involving Instruction Fetch & Execution

INSTRUCTION EXECUTION :

- The **operation field** of the instruction in IR is examined to determine the type of operation to be performed by the ALU

- The specified operation is performed by **fetching the operand**(s) from the memory locations or from GP registers in the processor

Steps involving Instruction Fetch & Execution

- Fetching the operands from the memory requires sending the memory location address to the MAR and initiating a Read cycle

- The operand is read from the memory into the MDR and then from MDR to the ALU

- The ALU performs the desired operation on one or more operands fetched in this manner and sends the result either to **memory location** or to a **GP register**

Steps involving Instruction Fetch & Execution

- If the result of this operation is to be stored in the memory, then the result is sent to MDR
- The address of the location where the result is to be stored is sent to MAR and a **Write cycle** is initiated
- Thus, the execute cycle ends for the current instruction and the PC is **incremented** to point to the next instruction for a new fetch cycle.

Execution of an instruction

- Normal execution of pgms may be preempted if some device requires urgent servicing
- To deal with the situation immediately, normal execution of current pgm must be interrupted
- To do this, the device raises an ***interrupt signal***
- Interrupt
  - Is a **request from an I/O device** for service by the processor
- Processor provides the requested service by executing an appropriate ***interrupt-service routine***

Steps to Execute an Instruction-summary

1) The **address** of first instruction (to be executed) gets loaded into **PC**.
2) The contents of PC (i.e. address) are transferred to the **MAR** & control-unit issues **Read signal** to memory.
3) After certain amount of elapsed time, the first instruction is read out of memory and placed into **MDR**.
4) Next, the contents of MDR are transferred to **IR**. At this point, the instruction can be **decoded** & **executed**.
5) To **fetch an operand**, it's address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from **MDR to ALU**.
6) Likewise required number of operands is fetched into processor.
7) Finally, **ALU** performs the desired **operation**.
8) If the **result** of this operation is to be stored in the **memory**, then the result is sent to the MDR.
9) The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
10) At some point during execution, contents of PC are **incremented** to point to next instruction in the program.

Computer Instructions

- Assembly Language
  - MOVE NUM1,R1
  - MOVE #1,R2
  - ADD #1,R1
  - ADD R1,R2

- Register Transfer Notation
  - R1 ← [NUM1]
  - R2 ← 1
  - R1 ← 1 + [R1]
  - R2 ← [R1] + [R2]

The "fetch-execute cycle" – Example Instruction

MOVE NUM1,R1

- Fetch
  - MAR ← [PC]
  - PC ← [PC] + 1
  - MDR ← [MEM([MAR])]
  - IR ← [MDR]

- Execute
  - MAR ← NUM1
  - MDR ← [MEM([MAR])]
  - R1 ← [MDR]

Another Example

ADD #1,R1

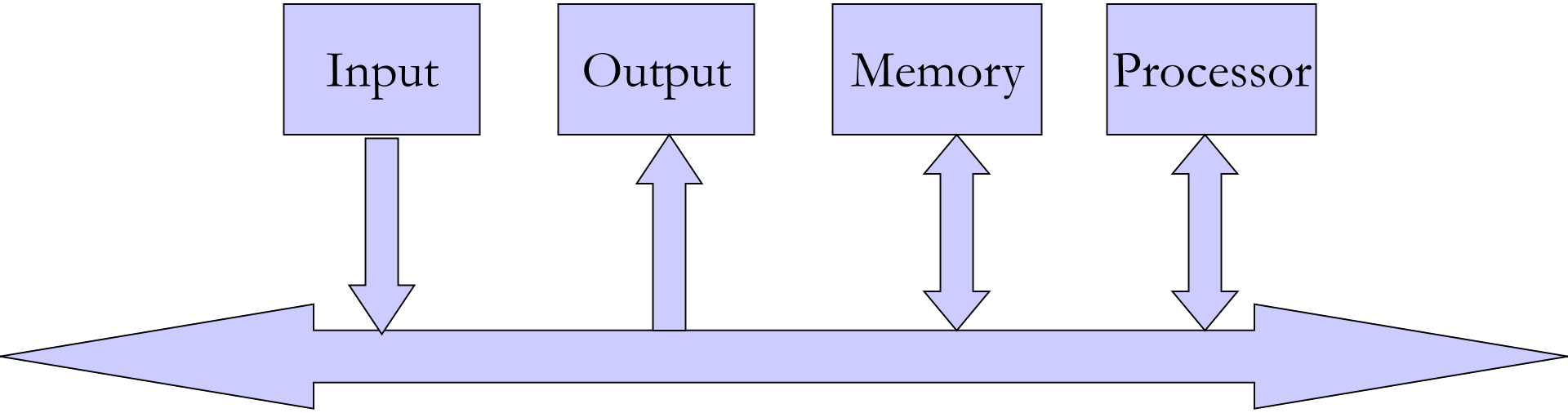
- Fetch
  - MAR ← [PC]
  - PC ← [PC] + 1
  - MDR ← [MEM([MAR])]
  - IR ← [MDR]

- Execute
  - R1 ← 1 + [R1]

Bus Structures

- A group of lines that serves as a connecting path for several devices is called a bus
- Bus must have lines for
  - Data
  - Address
  - Control

Single-bus Structure



- The simplest way to interconnect functional units is to use a single bus
- Since the bus can be used for **only one transfer** at a time, only two units can actively use the bus at any given time

Single-bus Structure

- It's basic feature is it's **low cost** and flexibility for attaching peripheral devices
- Systems containing multiple buses increase it's performance capability (by concurrency in operations) but at an increased cost
- Buffer registers within the devices
  - Smooths out the differences in speed among processor, memory and i/o devices
  - holds the information during transfers
  - Allows processor to switch rapidly from one device to another
  - Ex: use of **printer buffer** during printing

Single-bus Structure

- Transfer of a character from a processor to a character printer
  - Processor sends the character to the printer buffer
  - Once buffer is loaded, the printer can start printing without intervention by the processor
  - Prevents high-speed processor from being blocked to a slow i/o device during a sequence of transfers.

Single-bus Structure

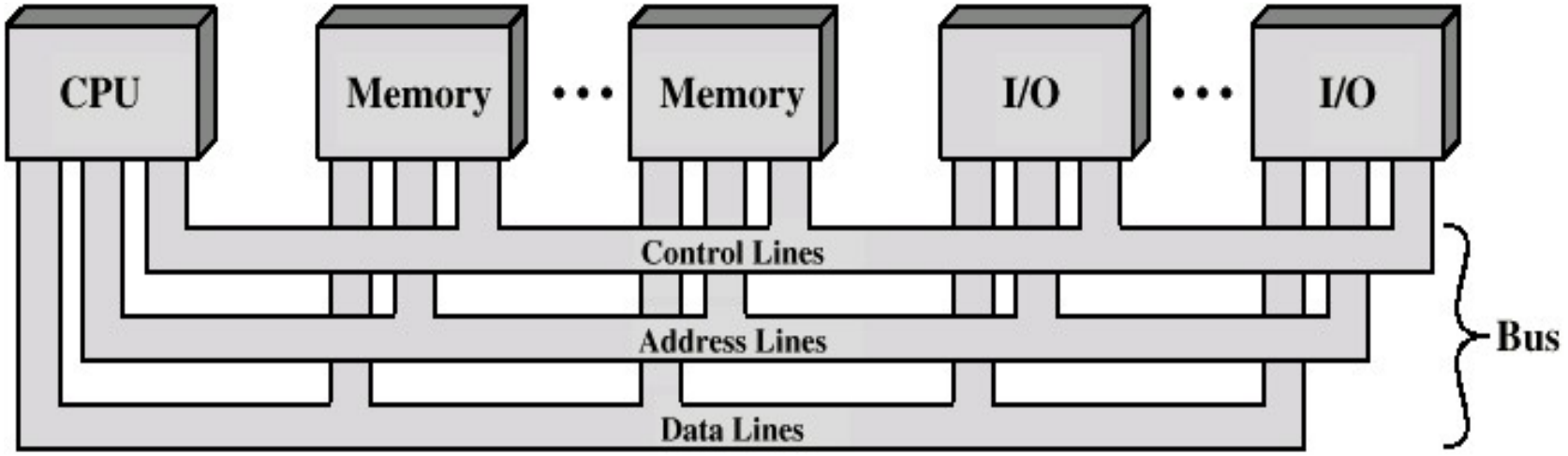
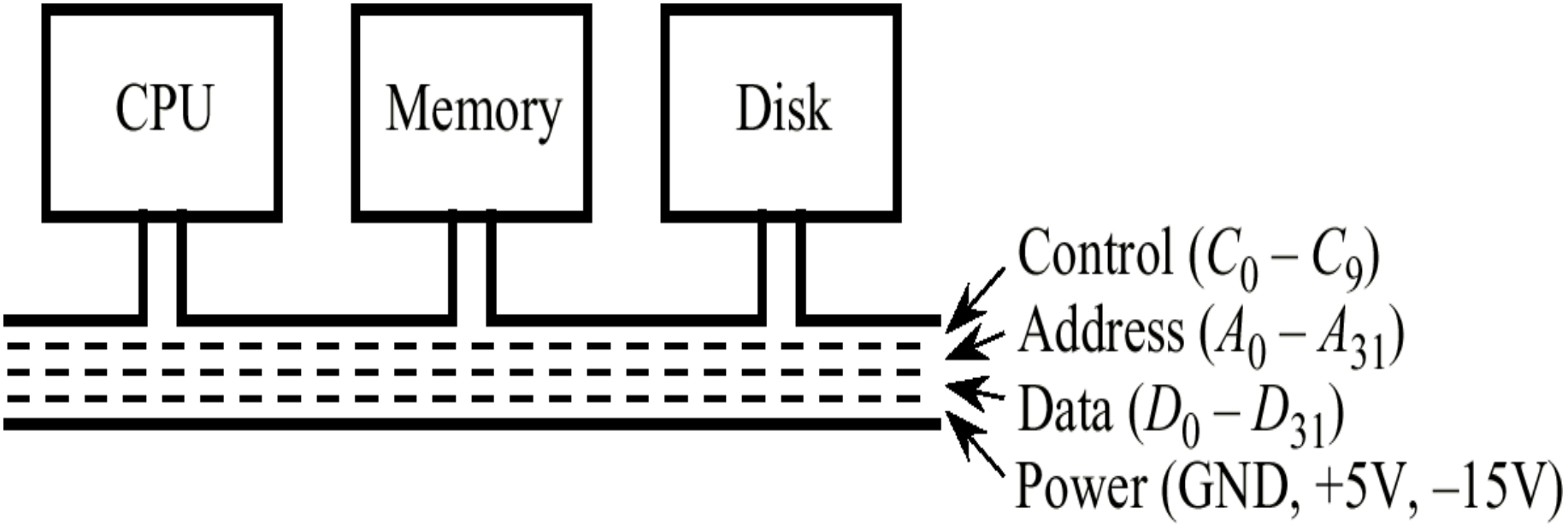
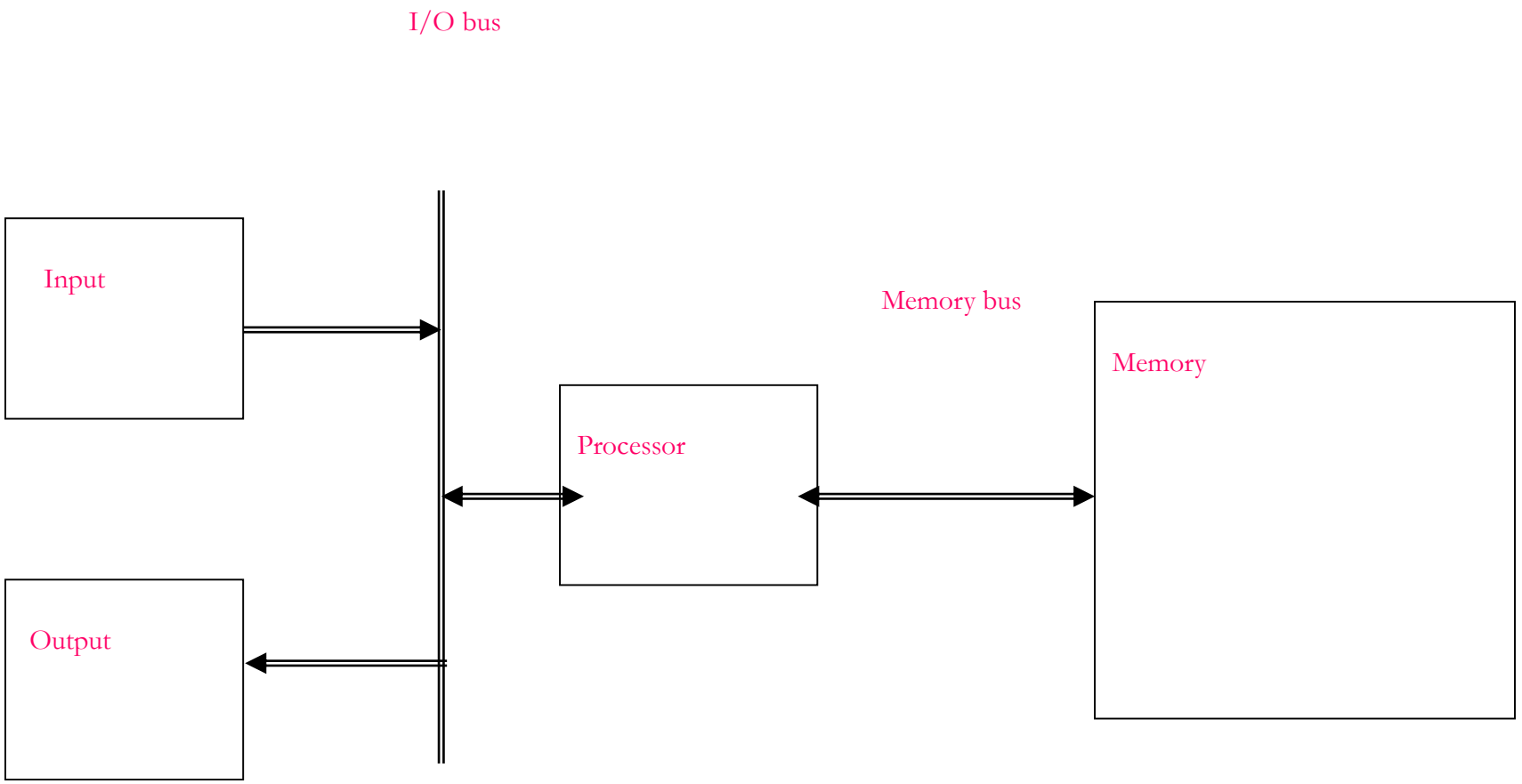


Figure 3.16 Bus Interconnection Scheme

Simplified Illustration of a Bus



Two-Bus Structure



Two-Bus Structure

- The bus is said to perform two distinct functions by connecting the I/O units with memory and processor unit with memory. The processor interacts with the memory through a **memory bus** and handles input/output functions over **I/O bus**.

- The main advantage of this structure is **good operating speed** but on account of **more cost**.

- The memory bus is also called the *front side bus, local bus, system bus,, processor bus* or *host bus*.
  - Connects CPU with memory

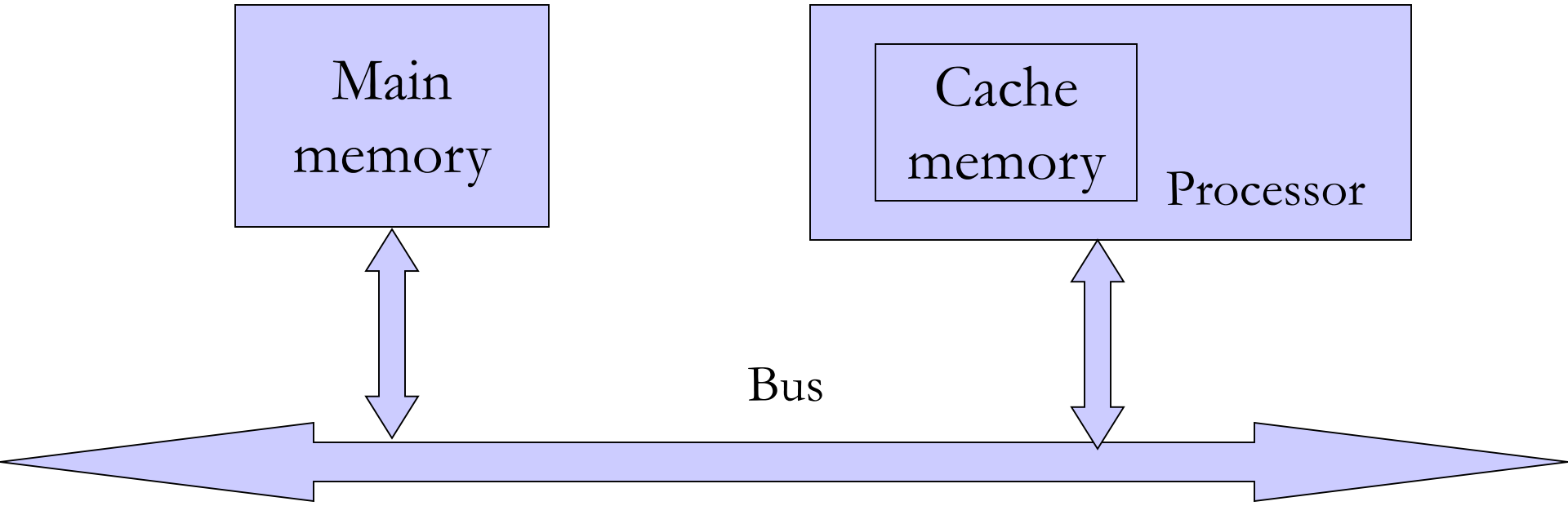
Performance

- The most important measure of the performance in a computer is how quickly it can execute programs
- The speed of execution depends on the design of its hardware and its machine language instructions and also the compiler
- For better performance all components should be designed in a coordinated way

Performance

- Performance of computer - program execution speed
- Elapsed time
  - Total time required to execute the pgm
  - Measure of computer's performance
  - Depends on speed of processor, disk & I/O
- Processor time
  - Period during which the processor is active to execute the pgm

The Processor Cache



The Processor Cache

- At the start of execution, all pgm instructions & the required data are stored in main memory
- As execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache
- When the execution of an instruction calls for data located in the main memory, the data are fetched & a copy is placed in the cache
- Later, if the same instruction/ data item are needed a second time, it is read directly from the cache

The Processor Cache

- A program will be executed faster if the movement of instructions and data between the processor and main memory is minimized, which is achieved by using cache

Processor Clock

- Processor circuits are controlled by timing signal called a *clock*
- The regular time intervals are known as *clock cycles*
- To execute machine instructions the processor divides the action into a sequence of steps such that each step can be completed in one clock cycle
- The length T of one clock cycle is an important performance parameter of processor
- Clock rate or frequency, R=1/T, measured in cycles/sec (Hertz/Hz)

Processor Clock

- Terminology:
  - Million - **Mega** (M) = 10^6
  - Billion - **Giga** (G) = 10^9

- f=500 million cycles/sec, T=?
  - f= 500 X 10^6 = 500 MHz
  - Clock period T=1/f
    =1/ (500 X 10^6)= 2 X 10^{-9} = 2ns

- T= 0.8 ns, f=?
  - Clock Rate f=1/T = 1/ (0.8 X 10^{-9})= (1.25 X 10^9)
    = 1.25 GHz =1250 million cycles/sec

Basic Performance Equation

- **$T=(N \times S) / R$**
  - T is the processor time required to execute a program
  - N is the actual number of instructions executed
  - S is the average number of basic steps needed to execute one machine instruction
  - R is the clock rate (or frequency f)

- To achieve high performance, reduce the value of T, which means reducing N & S and increasing R

Units of Performance Measurement

- MIPS: Million instructions per second
- Megaflops: Million floating point operations per second
- Megahertz: Million clock cycles per second

Techniques to improve performance

- Pipelining and Superscalar Operation
  - Decreases S
- Increasing the clock rate
  - Increases R
- Choice of Instruction Set : CISC or RISC
  - Affects N and S
- Use of optimizing compiler
  - Decreases N

Pipelining and Superscalar Operation

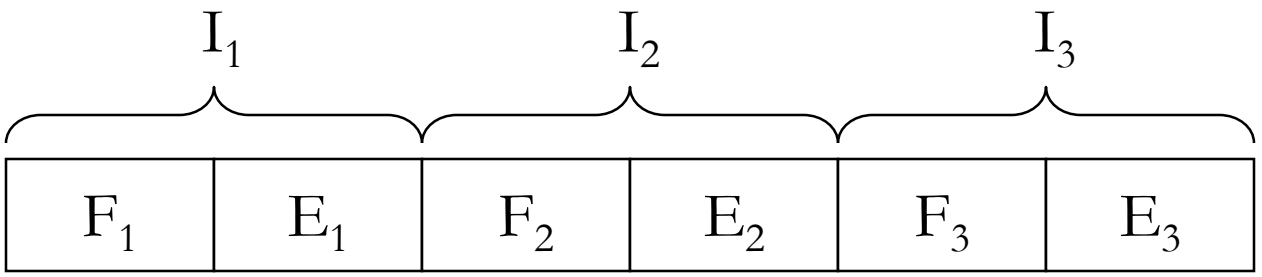
- Assumed that instructions are executed one after another

- Improvement in performance can be achieved by overlapping the successive instructions using a technique called *pipelining*

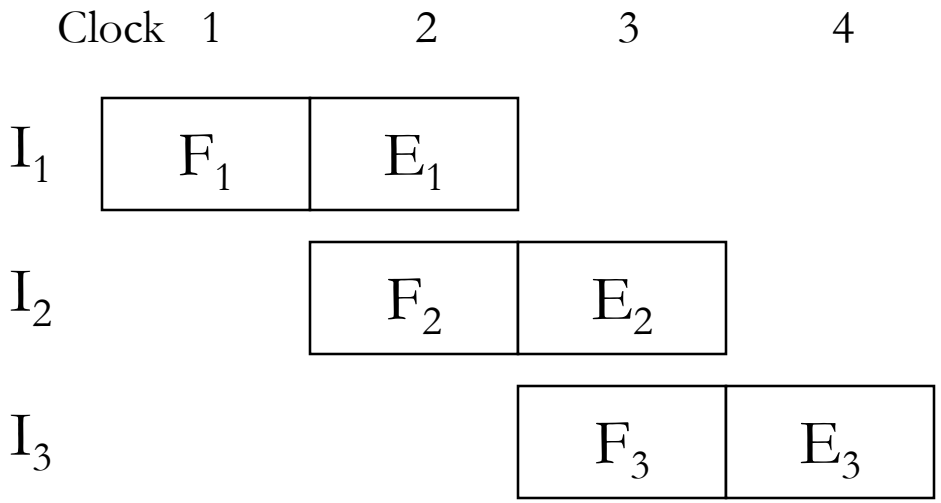
Pipelining and Superscalar Operation

- Consider the instruction Add R1,R2,R3
  - Which adds the contents of registers R1 & R2 and places the sum into R3
  - Contents of R1 & R2 are first transferred to the inputs of ALU
  - After add operation is performed, sum is transferred to R3
  - The processor can read the next instruction while the addition operation is being performed
  - Then, if that instruction also uses ALU, it's operands can be transferred to the ALU inputs at the same time that the result of Add instruction is being transferred to R3

Pipelining



Sequential Execution



Legends:
I – instruction
F – Fetch
E- Execute

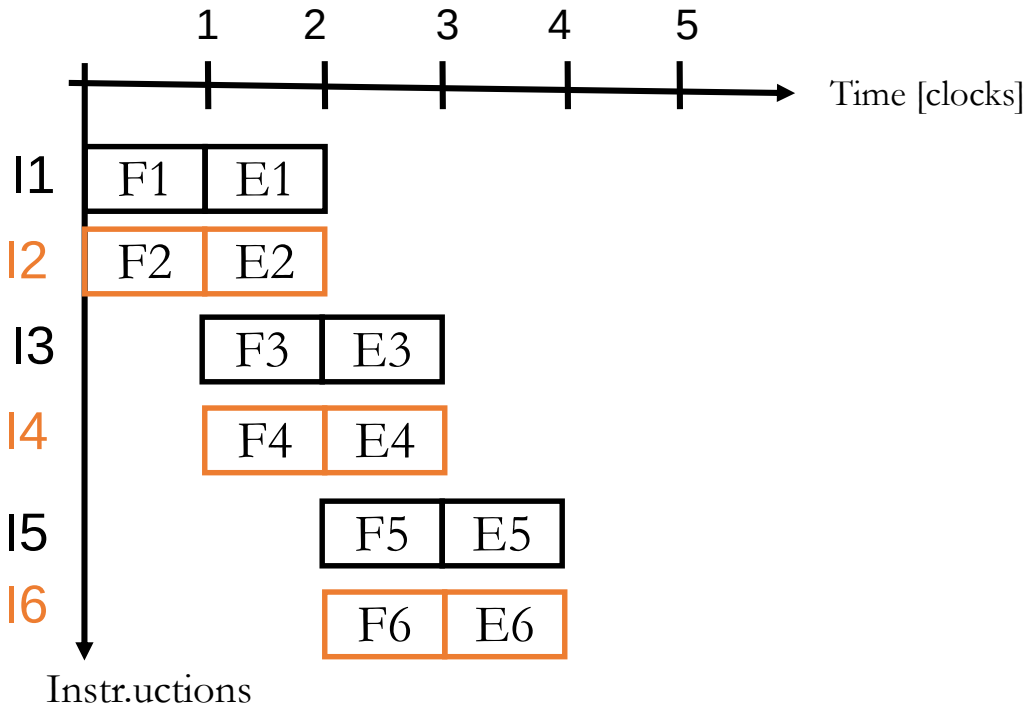
Pipelined Execution

Pipelining

- For purpose of computing, effective value of S is 1 (can't attain in practice)
- Pipelining increases rate of executing instructions significantly & causes the value of S to approach 1
- A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor
  - Means that multiple functional units are used, creating parallel paths through which different instructions can be executed in parallel
  - With this, it becomes possible to start the execution of several instructions in every clock cycle
  - This mode of operation is called *Superscalar execution*

A 2-issue Superscalar execution

- Fetch 2 instructions every clock



Clock Rate

- There are 2 possibilities for increasing the clock rate, R
  - First, improving the Integrated Circuit (IC) technology makes logic circuits faster, which reduces the time needed to complete a basic step
    - This allows the clock period, P to be reduced and the clock rate, R to be increased
  - Second, reducing the amount of processing done in one basic step makes it possible to reduce clock period, P

Instruction Set: CISC & RISC

- Simple instructions require small number of basic steps to execute (Reduced Instruction Set Computers)
- Complex instructions involve a larger number of basic steps (Complex Instruction Set Computers)
- For RISC a large number of instruction is needed, lead to large value for N, and small value for S
- For CISC individual instructions perform more complex operations with fewer instructions, leading to lower value of N and larger value of S
- It is not obvious if one choice is better than the other

CISC vs RISC

- Complex Instruction Set Computers (CISC)
  - Smaller N
  - Larger S
- Reduced Instruction Set Computers (RISC)
  - Larger N
  - Smaller S
  - Easier to Pipeline

Compiler

- High-level Language → Machine Language
- To reduce N, suitable machine instruction set + compiler that makes good use of it
- An *optimizing compiler* must reduce the number of clock cycles needed to execute a program
- The number of clock cycles is dependent not only on the choice of instructions but also on the order in which they appear in the program
- The compiler may rearrange the instructions to achieve better performance
  - Such changes must not affect the result of the computation
- Ultimate objective is to reduce the total no: of clock cycles needed to perform a required pgmg task

Performance Measurement

- The only parameter that describes the performance of a computer is the execution time T
- A nonprofit organization called System Performance Evaluation Corporation (SPEC) publishes
- Provides benchmark programs for different application domains, together with test results for many commercially available computers
- Benchmark programs - standardized set of programs used for measuring performance

Performance Measurement

- A Benchmark Program is compiled and run on computer under test- note the running time
- The same pgm is compiled and run on a reference computer
- Compute the SPEC ratio as

$$SPEC\ Ratio = \frac{running\ time\ on\ the\ refence\ computer}{running\ time\ on\ the\ computer\ under\ test}$$

SPEC ratio of 50 means comp. under test is 50 times faster than ref. comp.

Performance Measurement

- The above process is repeated for all pgms in the SPEC suite
- The overall SPEC ratio for the computer

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

Where,

n – number of pgms in the suite

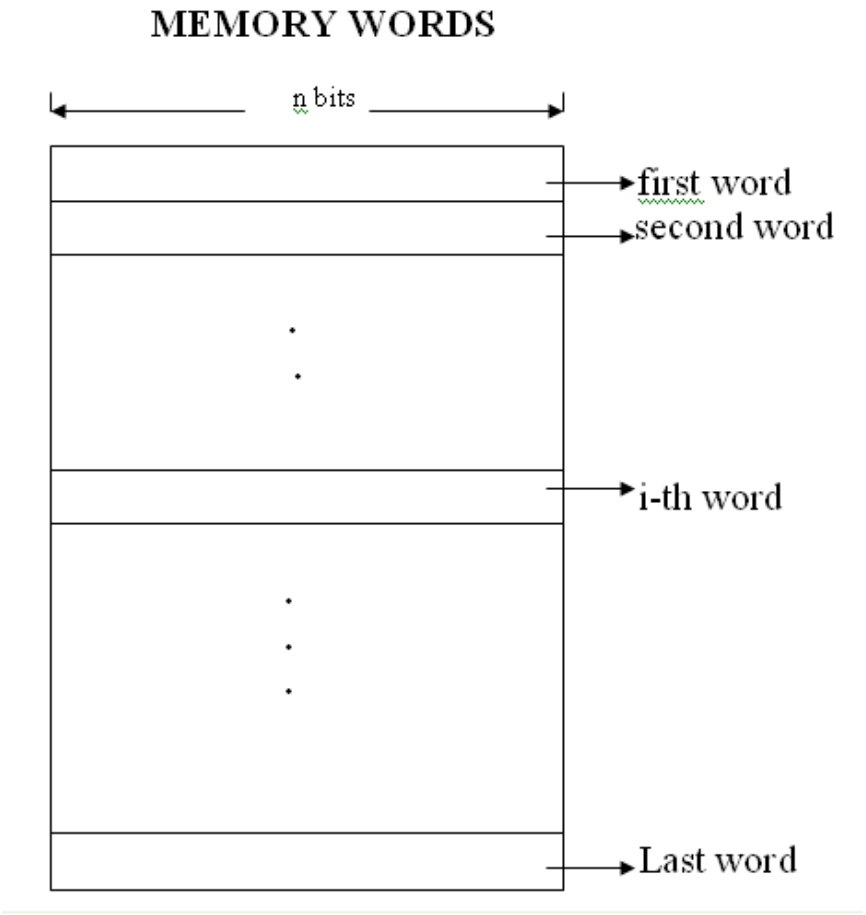
SPECi – ratio for the pgm i in the suite

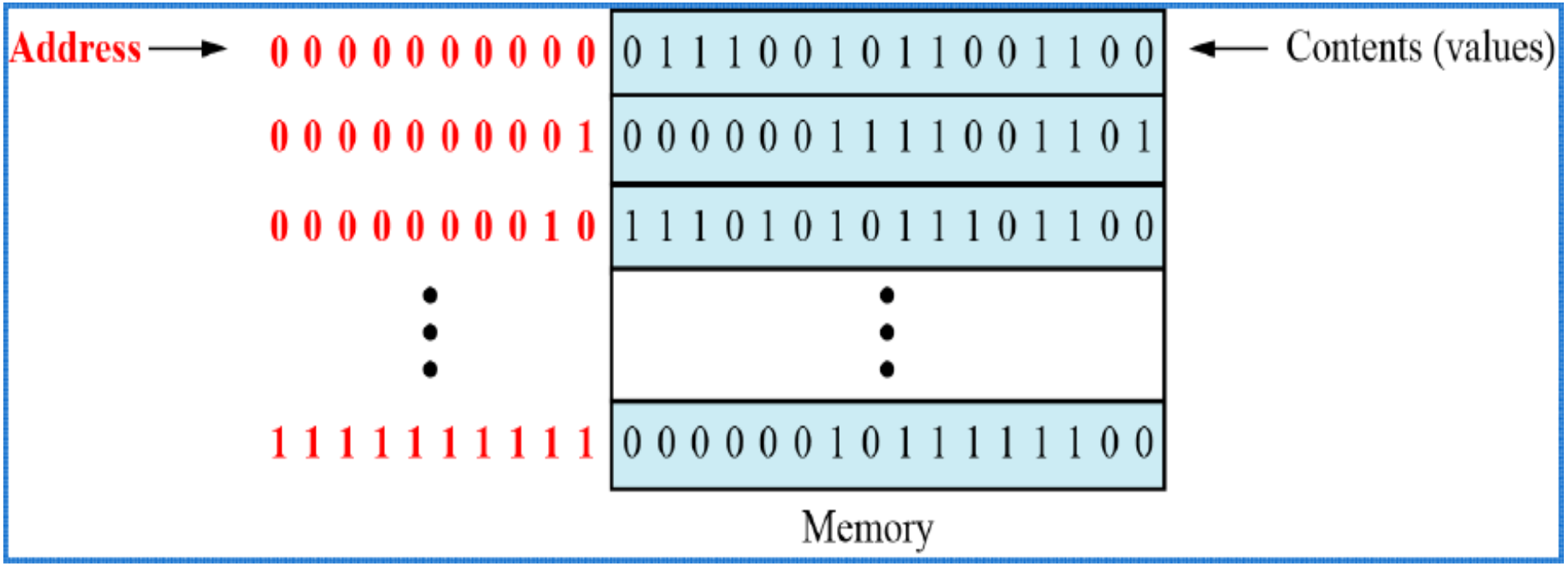
CHAPTER 2

MACHINE INSTRUCTIONS AND PROGARMS

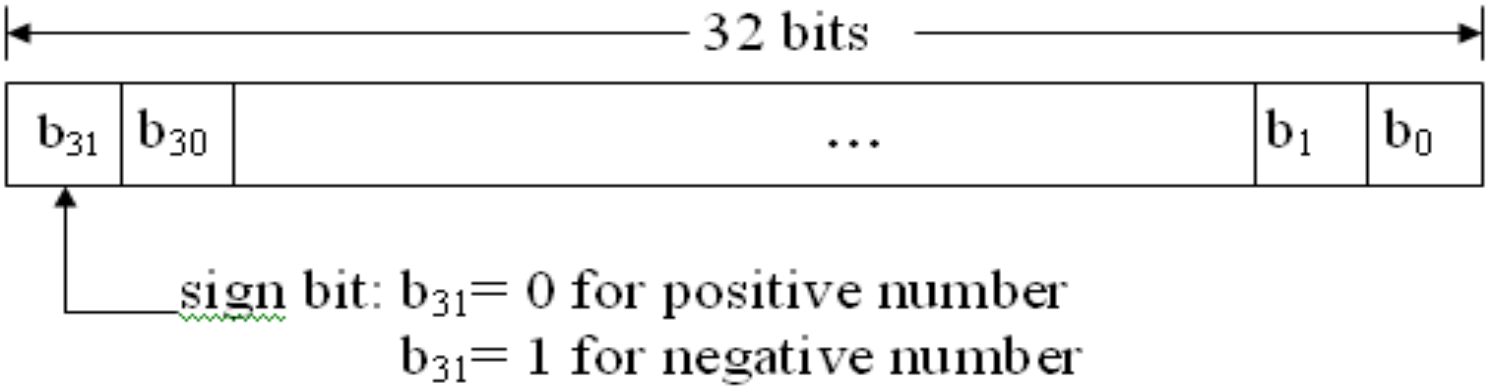
Memory Locations & Addresses

- **Word** - group of n bits; n is **word length**
- **Word length ranges from 16 to 64 bits**

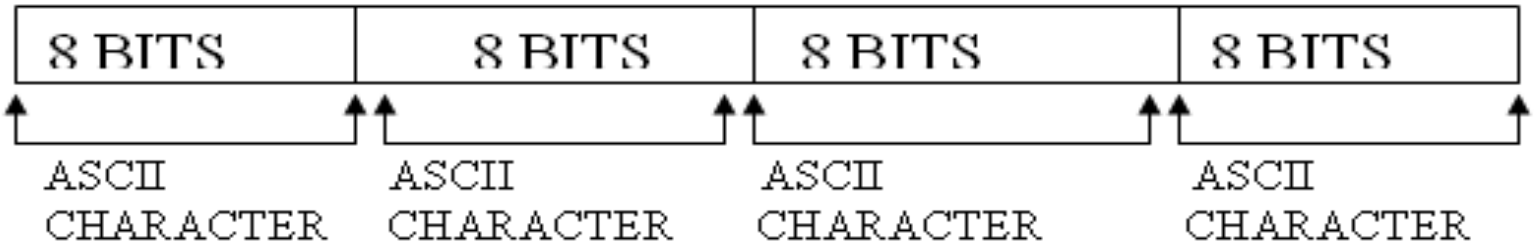



Address
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1 0
1 1 1 1 1 1 1 1 1 1 1
0 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0
0 0 0 0 0 0 1 1 1 1 0 0 1 1 0 1
1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 0
0 0 0 0 0 0 1 0 1 1 1 1 1 1 0 0
Contents (values)
Memory

Encoded information in a 32 bit word



a) A signed integer



b) Four characters

Contd..

- If word length = 32 bits, it can store
  - One 32 bit 2's complement no.
  - or four ASCII characters
- **Address** – a number used to access a memory location
- k bit address – 2^k addresses
- Used to address 0 to 2^k-1 successive locations

Address Space

- Address Space of the computer :- The maximum no. of addressable locations
- With
  - k bit address, Address Space= 2^k addresses
  - 24 bit, A.S=2^{24}=16M
- Note : 2^{10}=1K

  2^{20}=1M

  2^{30}=1G

  2^{40}=1T

Units of Memory

Unit	*Exact Number of Bytes*	*Approximation*
kilobyte	2^{10} (1024) bytes	10^3 bytes
megabyte	2^{20} (1,048,576) bytes	10^6 bytes
gigabyte	2^{30} (1,073,741,824) bytes	10^9 bytes
terabyte	2^{40} bytes	10^{12} bytes

Example 1

A computer has 32 MB (megabytes) of memory. How many bits are needed to address any single byte in memory?

Solution

The memory address space is 32 MB, or 2^{25} ($2^5 \times 2^{20}$). This means that we need $\log_2 2^{25}$, or **25 bits**, to address each byte.

Example 2

A computer has 128 MB of memory. Each word in this computer is eight bytes. How many bits are needed to address any single word in memory?

Solution

The memory address space is 128 MB, which means 2^{27}. However, each word is eight (2^3) bytes, which means that we have 2^{24} words. This means that we need $\log_2 2^{24}$, or **24 bits**, to address each word.

Byte Addressability

- Byte Addressable Memory : Successive addresses to successive locations
- If

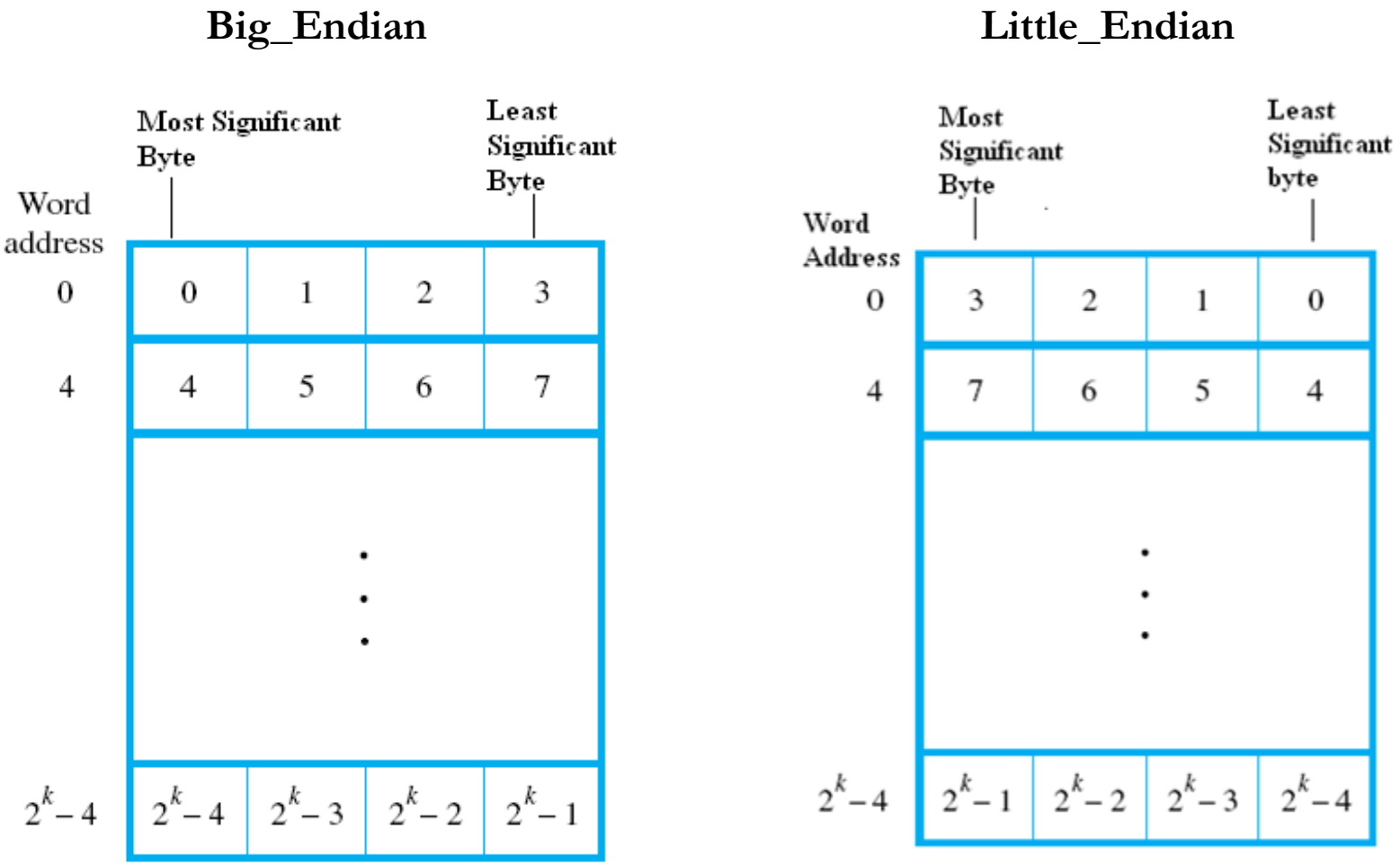
  Byte location addresses are 0,1,2,….

  and Word length=32 bits

Then, Successive word addresses are 0, 4, 8,….

Big-Endian & Little-Endian

Two types of assigning bytes to a memory word



Little-Endian

- the "little" end (LSB) of a multi-byte integer gets stored first, and the next bytes get stored in higher (increasing) memory locations.

- Little-Endian byte order is "littlest end goes first (to the littlest address)".

Big-Endian

- the "big" end (MSB) of a multi-byte integer gets stored first, and the next bytes get stored in higher (increasing) memory locations.

- Big-Endian byte order is "biggest end goes first (to the lowest address)".

Example - big endian

- Suppose we have a 32 bit data, written as $90AB12CD_{16}$
- In big endian, you store the most significant byte in the smallest address.

Address	Value
1000	90
1001	AB
1002	12
1003	CD

Example- little endian

Given data is $90AB12CD_{16}$

In little endian, you store the *least* significant byte in the smallest address.

Address	Value
1000	CD
1001	12
1002	AB
1003	90

Contd...

- Example for little-endian : Intel, DEC
- Example for big-endian :IBM, Motorola, Sun, HP

- In both cases, addresses 0,4,8,… are the addresses of successive words in memory.

- And these addresses are used for reading and writing the words from the memory.

Contd...

- Big-Endian : - Lower Byte addresses for more significant bytes (leftmost bytes)

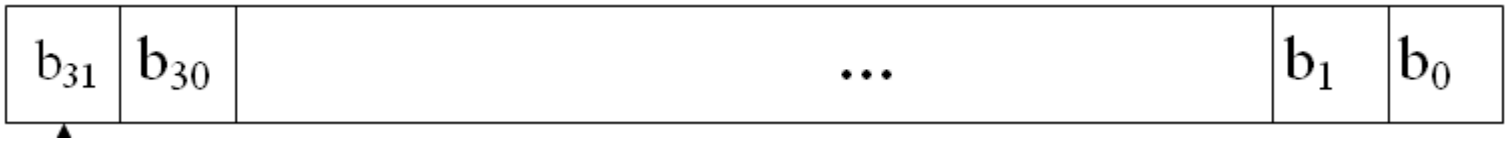
  - 0^{th} byte (MSB of the 1^{st} word) – address 1000
  - 3^{rd} byte (LSB of the 1^{st} word) – address 1003

  - 4^{th} byte (MSB of the 2^{nd} word) – address 1004
  - 7^{th} byte (LSB of the 2^{nd} word) – address 1007

Contd…

- Little-Endian : - Lower Byte addresses for less significant bytes (rightmost bytes)
  - 0th byte (MSB of the 1st word) – address 1003
  - 3rd byte (LSB of the 1st word) – address 1000

  - 4th byte (MSB of the 2nd word) – address 1007
  - 7th byte (LSB of the 2nd word) – address 1004
- Bit ordering :- labeling bits within a byte/word



Endianness and Character Data

- Single-byte character data such as ASCII is not affected by Endianness…
- since each character is one byte long and the start character of the string is always stored at the lowest memory location

- For the string "abcd", the "a" is stored "first"

Memory address	Byte value	ASCII Character
101	61	a
102	62	b
103	63	c
104	64	d

- No matter what the actual Endianness of the hardware, single-byte character data byte order resembles Big-Endian byte order
  - the left-most byte goes first

Example

- **Question** : Consider a computer that has a byte-addressable memory organized in 32-bit words according to the big-endian scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the name "Johnson" has been entered

Solution:

- ASCII code for J=01001010 , o=01101111 , h=01101000 , n=01101110, s=01110011
- Hence 'Johnson'= 4A 6F 68 6E 73 6F 6E

Big-endian scheme

Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 4A6F686E and 736F6EXX.

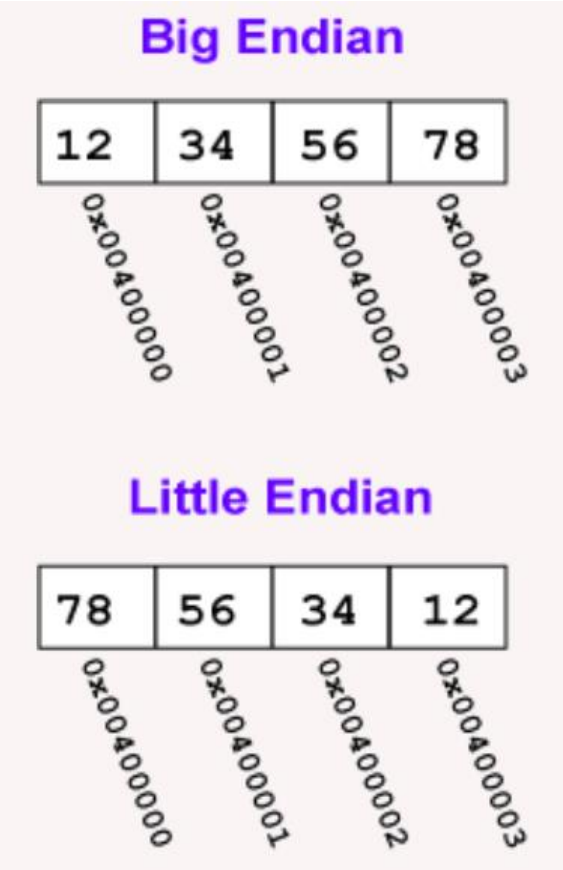
1000	4A	6F	68	6E	1003
1004	73	6F	6E	XX	1007

Little-endian scheme

Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 6E686F4A and XX6E6F73. Byte 1007

1000	4A	6F	68	6E	1003
1004	73	6F	6E	XX	1007

- Example: Store 0x12345678 in (a) little and (b) big endian form



Word Alignment

- Aligned Address : Word addresses begin at a byte address that is a multiple of no. of bytes in a word
- Example
  - If word length = 32 bits =4 bytes,

    aligned word addresses are 0, 4, 8, 12, ….
  - If word length =64 bits= 8 bytes,

    aligned word addresses are 0, 8, 16, 24, ….
  - If word length =16 bits = 2 bytes,

    aligned word addresses are _ _ _ _ _ _

Contd..

- Unaligned addresses : Word addresses begin at an **Arbitrary** byte address

- But Aligned address is commonly used

Accessing Numbers, Characters, Char. strings

- Number : occupies one word
  - Accessed by specifying word address
- Character : occupies one byte
  - Accessed by specifying byte address
- String : Variable length
  - Specify the address of first byte (for 1st character)
  - Successive characters found in successive locations
  - Length : indicted by special control character.
    or separate register indicating length

Memory Operations

- Two basic operations
  - Load (Read or Fetch)
    - copy content of a memory location to processor
    - Memory contents remain unchanged
  - Store (or Write)
    - Transfer an information item from processor to a memory location
    - Destroys the former contents of that location
- Processor sends the address of desired mem loc
- Read/write one word/byte at a time
- Processor reg. – capable of holding a word used as source/ destination of transfer.
- When a Byte is to be transferred
  - byte located in lower-order position in the reg. is transferred

Instructions & Instruction Sequencing

A computer must have instructions to perform four types of operations:

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

- Two types of notations are used to represent the instructions
  - Register Transfer Notation (RTN)
  - Assembly Language Notation

REGISTER TRANFER NOTATION

- Locations involved in Data transfer are:

Location	Symbolic names
Memory	LOC, A, NUM
Processor register	R0, R5
I/O sub system registers	DATAIN, OUTSTATUS

- A name stands for hardware address of a location
- Content of a location – indicated by Square brackets around the name eg, [LOC] , [R0], [DATAIN]

RTN - Examples

- **R1 ← [LOC]**

  Means that the contents of location named LOC is transferred into processor register R1.

- **R3 ← [R1] + [R2]**

  Add the contents of register R1 and R2, and then place the sum into register R3

- RHS represents a value, LHS- name of a location where the value is to be placed (overwritten)

ASSEMBLY LANGUAGE NOTATION

- **Move LOC,R1**

  Means that the contents of location LOC is transferred into processor register R1.

- **Add R1,R2,R3**

  Add the contents of register R1 and R2, and then place the sum into Register R3.

BASIC INSTRUCTION TYPES

- In a high level language statement

  C = A+ B

Variables A, B and C refer to distinct memory location addresses

- contents of these locations represent the values of the three variables
- RTN for the above action: *C* ← *[A] + [B]*

3 Address Instruction

- A 3-address m/c instruction is represented as:

  Add A,B,C

  - A & B are called the *source* operands
  - C is called the *destination* operand
  - Add is the operation

- General format:

  Operation Source1,Source2,Destination

Contd…

- With k bit address :

  3*k bits – for addressing 3 operands

  some bits (say x) – to denote operation

- With 32 bit (4 byte) address:
  - 12 bytes - for addressing 3 operands
  - 1 or 2 bytes- to denote operation
  - So, 3 address instruction needs around 14 bytes
  - Hence. It is too large to fit in one word

- Alternative - Use a sequence of simpler instructions for the same task

2 Address Instruction

- General format:

  Operation Source,Destination

- Example 1:

  Add A,B

- Equivalent RTN

  B ← [A] + [B]

  - **B** is both source & destination
  - Original content of B is destroyed

2 Address Instruction

- Example 2:

  Move B,C

  Copies content of location B to location C

- Equivalent RTN

  C ← [B]

3 addr instruction using 2 addr instructions

- Implement **Add A,B,C** using 2 address instruction

  Move B,C

  Add A,C

- For usual word length and addr. size, even 2 addr. instruction cannot be normally fitted into one word

One-address instruction

- Instruction specifies only one operand
- Second operand will be in a register ; called the ***accumulator***
- Example

Add A

- Add the content of memory location A to content of accumulator
- Place sum back into accumulator

Contd..

- **Load A**

  Copies the contents of memory location A into the accumulator

  Location A is source and Acc is destination

- **Store A**

  Copies the contents of accumulator into memory location A

  Acc is source and A is destination

Representing C←[A]+[B] using 1-addr instructions

Load A
Add B
Store C

- Generalizations of 1-address load, store, and add instructions

  Load A,Ri (where Ri is a gen. purpose register, i.e. Acc)
  Store Ri,A
  Add A,Ri

Instructions using register operands

- Even one addr. Instruction may not fit into one word
- Solution:
  - Use registers to hold operands:
  - Benefits are:
    - Faster operation
    - Few registers – few bits to address them
    - Ex: 32 registers – 5 bit register addr

Contd..

Examples:

- ***Add Ri,Rj***
- ***Add Ri,Rj,Rk***

  - Ri and Rj contain source operands
  - Rj is also destination in first instruction
  - Rk is destination in second instruction

- Fits normally into one word

Contd..

- Instruction for data transfer between different locations

  Move Source, Destination

- Move data from memory to processor register

  Move Loc,Ri (same as ***Load Loc,Ri*** **)**

- Move data from processor register to memory

  Move Ri,Loc **(**same as ***Store Ri, Loc*****)**

- Move is used instead of Load/Store

Instruction sequence for C←[A]+[B]

- Using Move instead of Load/Store
- Arithmetic operation is allowed only on register operands

Move A,Ri

Move B,Rj

Add Ri,Rj

Move Rj,C

Instruction sequence for C←[A]+[B]

Arithmetic operation is allowed with one operand in memory and the other in register

Move A,Ri

Add B,Ri

Move Ri,C

Zero Address Instructions

- Locations of all operands in an instruction are defined implicitly
- In such machines operands are stored in ***Stack***
- Ex: **Add**
  - Means add two operands available in stack
  - Result is stored in Top Of Stack

Write 3, 2, and 1 address based assembly code for E= (A+B)*C – (D+E)*F

- 3-address based assembly code

ADD R1, A, B	R1<- [A] + [B]
MUL R2, R1, C	R2<- [R1] * [C]
ADD R3, D, E	R3<- [D] – [E]
MUL R3, R3, F	R3<- [R3] * [F]
SUB E, R2, R3	E<- [R2] - [R3]

- 2-address based assembly code

MOVE R1, A	R1<- [A]
ADD R1, B	R1<- [R1] + [B]
MUL R1, C	R1<- [R1] * [C]
MOVE R2, D	R2<- [D]
ADD R2, E	R2<- [R2] + [E]
MUL R2, F	R2<- [R2] * [F]
SUB R1, R2	R1<- [R1] - [R2]
MOVE E, R1	E<- [R1]

Write 3, 2, and 1 address based assembly code for E= (A+B)*C – (D+E)*F

- 1-address based assembly code
  - Note Ri is accumulator

LOAD D	Ri<- [D]
ADD E	Ri<-[Ri] + [E]
MUL F	Ri<- [Ri] * [F]
STORE T	T<- [Ri]
LOAD A	Ri<- [A]
ADD B	Ri<- [Ri] + [B]
MUL C	Ri<- [Ri] * [C]
SUB T	Ri<-[Ri] – [T]
STORE E	E<- [Ri]

Instruction Formats

- Three-Address Instructions
  - ADD R1, R2, R3 R1 ← R2 + R3
- Two-Address Instructions
  - ADD R1, R2 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

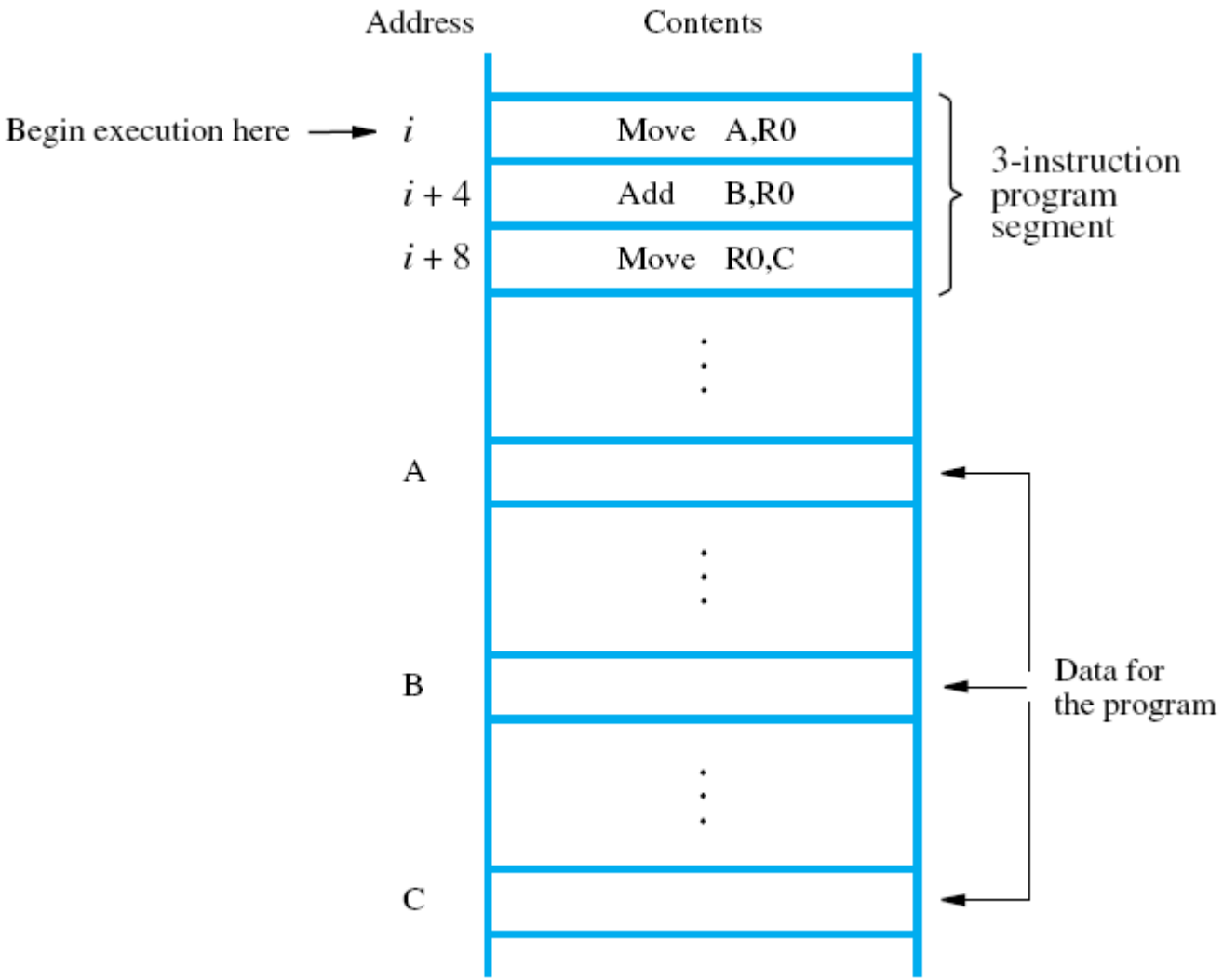
Instruction Execution & Straight Line Sequencing

Consider a pgm. for C←[A]+[B]

Assumptions on m/c features :

- Allows only one memory operand per instruction
- Has a number of registers
- Word length=32 bits
- Memory is byte addressable
- Each Instruction takes 4 bytes

Program C←[A]+[B] in memory



Straight Line Sequencing

- Place the address of first instruction (i) into PC
- Fetch & Execute this instruction
- Increment PC by 4 (i.e, i+4), to point to the next instruction
- Thus fetch & execute instructions one at a time in the **order of increasing addresses**

Executing a given instruction

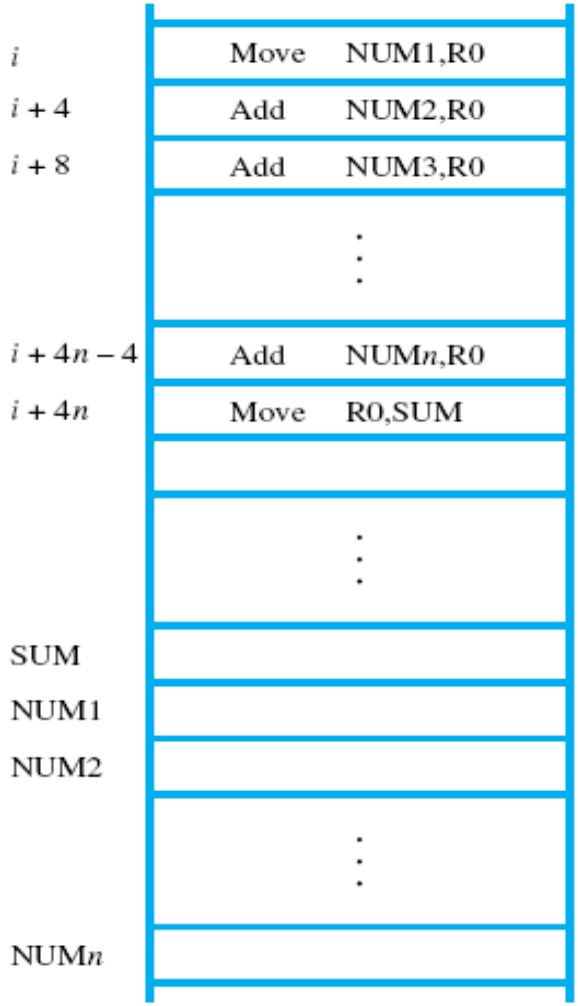
- 2 phase procedure to execute an instruction
  - Phase-1 : **Instruction Fetch**
  - Phase-2 : **Instruction Execute**

- **Instruction Fetch Phase :**
  - The instruction is fetched from the memory location whose address is in PC
  - Place this instruction in the *instruction register* (IR)

Contd…

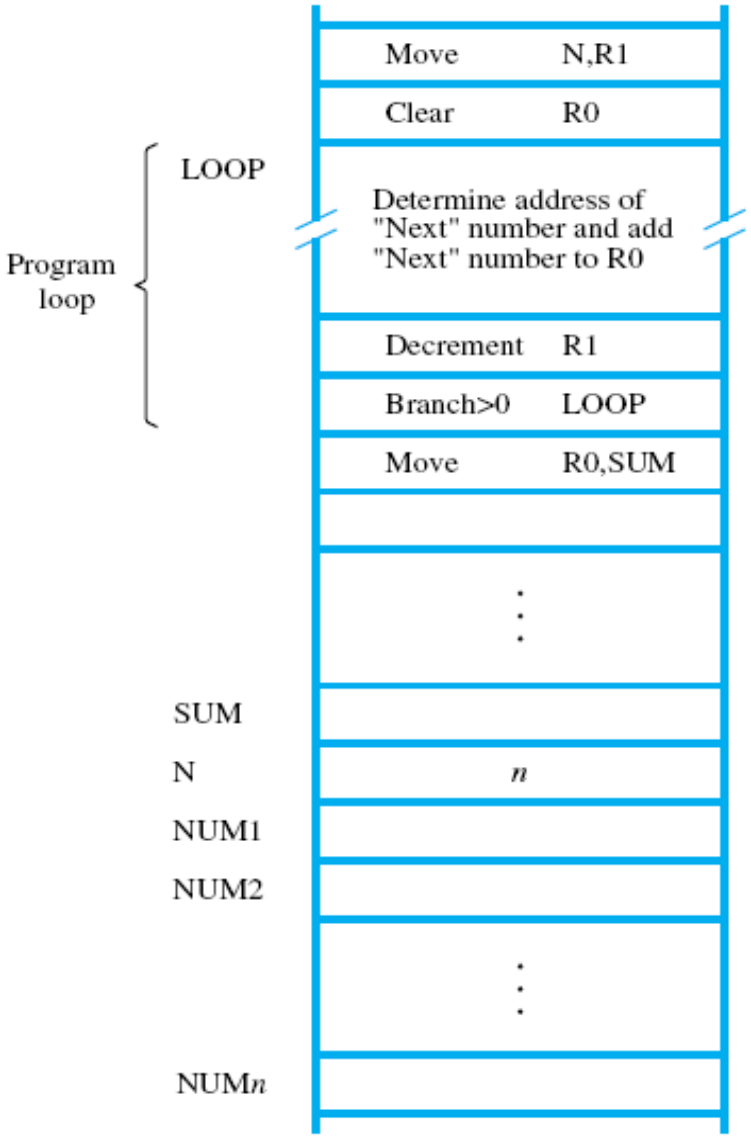
- **Instruction Execution Phase :**
  - **The operation code of IR is decoded by control unit**
  - **Specified operation is performed using ALU circuits**
  - **Operation may also involve fetching operands from memory or Registers**
  - **Result is stored in destination location**
- **Increment the contents of PC to point to next instruction**

A straight-line pgm. to add a list of n numbers



Branching

Using a loop to add *n* numbers.



Contd..

- **Program Loop** : Straight line sequence of instructions executed as many times as needed

  - **LOOP** – Where the pgm. loop begins

  - **Branch > 0** – Where the pgm. loop ends

  - **Register R1** – used as the counter
    - determines the number of loop iterations
    - decremented by 1 each time through the loop

Contd..

- **Branch** instruction : Loads a new address to PC instead of the next sequential address
- Branch target : The instruction at the new address
- Conditional Branch : Causes branch only if the specified condition is satisfied
- Else execute next instruction in sequential order

Contd…

- **Branch > 0 LOOP** : a conditional branch
- Branch to location LOOP if the result of preceding instruction (ie value in R1) > 0
- Branch doesn't occur when value in R1=0 (ie at n^{th} pass), and next instruction (Move) is executed

Condition Codes

- **Condition code flags** : Indicate the data condition or status after an arithmetic/logical operation
- A **Flag** is a one bit storage unit (a flip flop)
- Flag is **set** to 1 or **cleared** to zero based on the result of operation
- **Condition Code Register / Status Register** : is a processor register which accommodates a group of Flags

Commonly Used Flags of CCR

- **N** (negative) : Set to 1 if the result of arithmetic/logical operation is negative; otherwise, cleared to 0
- **Z** (zero) : Set to 1 if the result of arithmetic or logical operation is 0; otherwise, cleared to 0
- N & Z flags may also be affected by data transfer instructions such as Move, Load, or Store
- Enables conditional branching based on the value and sign of the operand that was moved

Contd..

- **V** (overflow) : Set to 1 if arithmetic overflow occurs, else cleared to 0

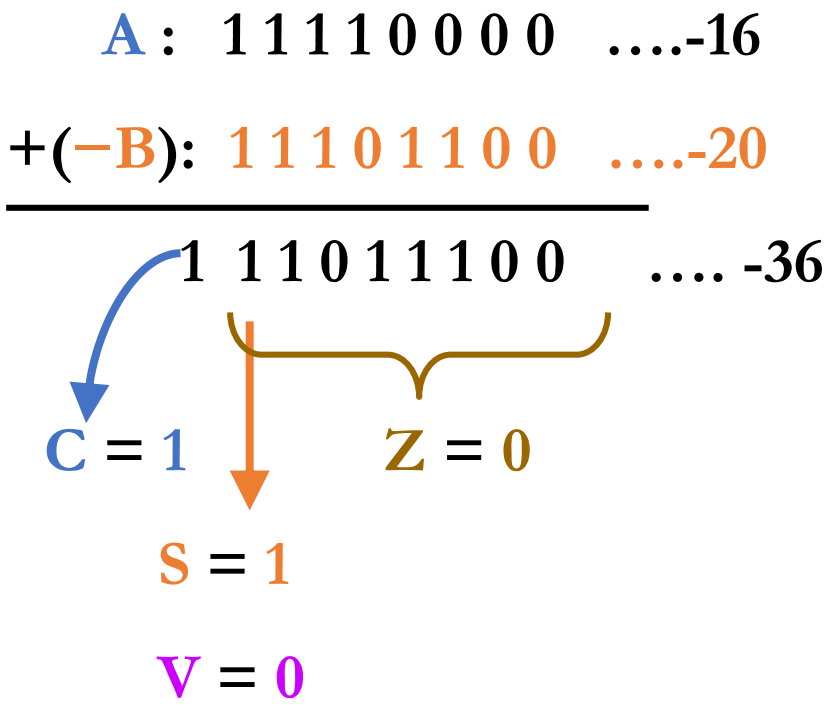
- Programmer can test V flag and branch to appropriate routine to correct the problem
  - Ex: BranchIfOverflow instruction

- **C** (carry) : Set to 1 if a carry-out results from MSB position during an arithmetic operation, else cleared to 0

- Branch > 0 - means branch if neither N nor Z flag is set to 1

Conditional Branch Instructions

- Example: A + (-B)
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



Acknowledgement
Internet

PPT edited by:
Raju K., Assoc. Prof.
CSE Dept, NMAMIT