Vidyavardhini's

College of Engineering & Technology

Vasai Road (W)

Department of Computer Engineering

Laboratory Manual
(Student Copy)

Semester	III	**Class**	**S.E**
Course Code	2113116		
Course Name	Computer Organization & Architecture Lab		

Vidyavardhini's College of Engineering & Technology

Vision

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

Mission

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional and societal needs through quality education.

Department Vision:

To evolve as a centre of excellence in the field of Computer Engineering to cater to industrial and societal needs.

Department Mission:

- To provide quality technical education with the aid of modern resources.
- Inculcate creative thinking through innovative ideas and project development.
- To encourage life-long learning, leadership skills, entrepreneurship skills with ethical & moral values.

Program Education Objectives (PEOs):

PEO1: To facilitate learners with a sound foundation in the mathematical, scientific and engineering fundamentals to accomplish professional excellence and succeed in higher studies in Computer Engineering domain

PEO2: To enable learners to use modern tools effectively to solve real-life problems in the field of Computer Engineering.

PEO3: To equip learners with extensive education necessary to understand the impact of computer technology in a global and social context.

PEO4: To inculcate professional and ethical attitude, leadership qualities, commitment to societal responsibilities and prepare the learners for life-long learning to build up a successful career in Computer Engineering.

Program Specific Outcomes (PSOs):

PSO1: Analyze problems and design applications of database, networking, security, web technology, cloud computing, machine learning using mathematical skills, and computational tools.

PSO2: Develop computer-based systems to provide solutions for organizational, societal problems by working in multidisciplinary teams and pursue a career in the IT industry.

Program Outcomes (POs):

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

- **PO12. Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Objectives

1	To make understand the concept of logic gates and their functioning
2	To make use of the simulator to learn the working of logic gates
3	To make understand the various ALU operations
4	To introduce the architecture of 8086 processor
5	To make use of the assembler for Assembly language programming
6	To learn the assembly language programming

Course Outcomes

At the end of the course student will be able to:		PO	Bloom Level
COL.1	Implement number system concept to convert decimal to binary, hexadecimal, and octal.	Implement	Apply (Level 3)
COL.2	Demonstrate the functioning of basic logic gates by designing logic circuits for arithmetic operation.	Demonstrate	Apply (Level 3)
COL.3	Implement arithmetic operations on binary, hexadecimal, and octal numbers using the concepts of the Arithmetic Logic Unit (ALU).	Implement	Apply (Level 3)
COL.4	Make use of TASM, MASM simulator to implement assembly language programs for sorting operations and basic arithmetic operations.	Make, Implement	Apply (Level 3)
COL.5	Implement data transfer techniques such as Programmed I/O, Interrupt-driven I/O, and Direct Memory Access (DMA) using suitable programming constructs.	Implement	Apply (Level 3)
COL.6	Implement the instruction pipelining concept by simulating the pipeline stages of a processor.	Implement	Apply (Level 3)

Mapping of Experiments with Course Outcomes

Sr.No	Experiment	Course Outcomes					
		COL.1	COL.2	COL.3	COL.4	COL.5	COL.6
1	Implement a program to convert hexadecimal, decimal number to binary number	**3**	**--**	**--**	**--**	**--**	**--**
2	Implement a program to obtain 2's complement of a number	**3**	**--**	**--**	**--**	**--**	**--**
3	Design a digital circuit using various logic gates	**--**	**3**	**--**	**--**	**--**	**--**
4	Implement a program to perform addition and subtraction of binary, hexadecimal number	**--**	**--**	**3**	**--**	**--**	**--**
5	Implement restoring division algorithm	**--**	**--**	**3**	**--**	**--**	**--**
6	Implement non-restoring division algorithm	**--**	**--**	**3**	**--**	**--**	**--**
7	Implement 8068 assembly language program for performing various arithmetic operation	**--**	**--**	**--**	**3**	**--**	**--**

8	Implement 8086 assembly language program for performing sorting of array	**--**	**--**	**--**	**3**	**--**	**--**
9	Implement a program to demonstrate various data transfer techniques	**--**	**--**	**--**	**--**	**3**	**--**
10	Implement a program to demonstrate the concept of pipeline	**--**	**--**	**--**	**--**	**--**	**3**

Enter correlation level 1, 2 or 3 as defined below
1: Slight (Low) 2: Moderate (Medium) 3: Substatial (High)

If there is no correlation put "—".

INDEX

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No.1
Implement a program to convert hexadecimal, decimal number to binary number
Date of Performance:
Date of Submission:

Aim: To implement a program that converts numbers from hexadecimal and decimal number systems into their equivalent binary representation.

Objective: To understand and implement the conversion of decimal and hexadecimal numbers into binary format, which is fundamental to data representation in digital systems.

Theory:

In computer systems, data is always processed and stored in binary—a base-2 numeral system that uses only two digits: 0 and 1. However, humans commonly use decimal (base-10), and system-level programs or hardware diagnostics frequently rely on hexadecimal (base-16) due to its compactness and readability.

Number Systems Overview:

Number System	**Base**	**Digits Used**	**Common Usage**
Binary	2	0, 1	Used internally by all computers
Decimal	10	0–9	Used by humans for general calculations
Hexadecimal	16	0–9, A–F	Used in memory addressing, debugging, etc.

Decimal to Binary Conversion:

Decimal to binary conversion involves repeated division by 2:

- Divide the decimal number by 2.
- Store the remainder.
- Repeat the process on the quotient until the quotient is 0.
- The binary number is the remainders read in reverse.

Example:
Convert 13 to binary:

13 ÷ 2 = 6 remainder 1

6 ÷ 2 = 3 remainder 0

3 ÷ 2 = 1 remainder 1

1 ÷ 2 = 0 remainder 1

Binary = 1101

Hexadecimal to Binary Conversion:

Hexadecimal to binary conversion is direct and efficient, as each hex digit maps exactly to a 4-bit binary number.

Hex Digit	**Binary Equivalent**
0	0000
1	0001
...	...
A (10)	1010
F (15)	1111

Example:
Convert Hex 2F to Binary:
2 = 0010, F = 1111 → Binary = 00101111

Why Binary?

- Binary aligns with the ON/OFF (high/low voltage) nature of digital electronics.
- It simplifies the design of hardware logic circuits using gates.
- All information (text, numbers, audio, video) in digital systems is represented in binary format.

Real-World Relevance:

- Microprocessors handle data and instructions in binary format.

- Hexadecimal simplifies representation of large binary numbers in system diagnostics and debugging.
- Understanding conversions is essential for memory management, instruction decoding, and low-level programming.

Solution:

Conclusion: We learned how to convert numbers from decimal and hexadecimal systems into binary. This helped us understand how computers represent and process different number systems internally.

Experiment No. 2
Implement a program to obtain 2's complement of a number
Date of Performance:
Date of Submission:

Aim: To implement a program that computes the 2's complement of a given binary number.

Objective: To understand and apply the concept of 2's complement for representing and manipulating signed binary numbers.

Theory:

In digital systems, particularly in computer architecture, numbers can be either unsigned (only positive) or signed (both positive and negative). While unsigned binary numbers are straightforward, representing negative numbers requires a special system. The 2's complement system is the most widely used method for representing signed integers in binary.

Why Use 2's Complement?

1. **Unified Arithmetic:** Addition, subtraction, and other operations can be performed using the same hardware logic for both positive and negative numbers.
2. **Single Representation of Zero:** Unlike 1's complement, which has two representations of 0 (+0 and -0), 2's complement has only one.
3. **Efficient Hardware Implementation:** Simplifies ALU design in CPUs and supports overflow detection.

How 2's Complement Works:

To find the 2's complement of a binary number:

1. Invert all bits (change 0 to 1 and 1 to 0) — this gives the 1's complement.
2. Add 1 to the result — this gives the 2's complement.

Example:
Let's find the 2's complement of 00010110 (which is +22 in decimal):

- Step 1: Invert all bits → 11101001
- Step 2: Add 1 → 11101010

The result, 11101010, is the 2's complement binary representation of -22.

Applications of 2's Complement:

- Used in Arithmetic Logic Units (ALUs) of processors to perform subtraction.
- Essential in assembly language programming and low-level software development.
- Helps in handling negative values efficiently in hardware design, compilers, and system programming.

Key Points:

- Most Significant Bit (MSB) is used as the sign bit in 2's complement (0 for positive, 1 for negative).
- In an n-bit system, the range of representable numbers is from -2^{n-1} to $(2^{n-1} - 1)$.
  - For 8-bit numbers: Range is −128 to +127.
- Overflow and underflow conditions can occur and must be handled carefully.

Solution:

Conclusion: We understood the concept of 2's complement and how it is used to represent negative numbers in binary. This gave us insight into how subtraction is performed in digital systems.

Experiment No. 3
Design a digital circuit using various logic gates
Date of Performance:
Date of Submission:

Aim: To design and simulate a digital logic circuit using various basic and derived logic gates with the help of Cedar Logic Simulator.

Objective: To understand the working of logic gates and apply them in designing and simulating digital circuits using Cedar Logic Simulator.

Theory:

Introduction to Logic Gates:

Logic gates are the **fundamental building blocks** of digital electronics. These gates perform **Boolean algebra operations** on one or more binary inputs to produce a binary output. Logic gates are implemented using transistors in hardware but can also be simulated virtually using logic simulation software such as **Cedar Logic Simulator**.

There are two categories of gates:

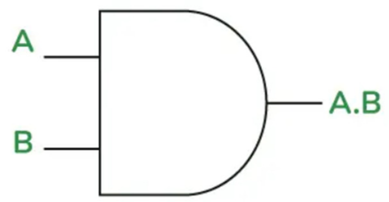
1. **Basic Gates**: AND, OR, NOT
2. **Derived (Universal) Gates**: NAND, NOR, XOR, XNOR

These gates are used to build more complex components like **adders, multiplexers, decoders, flip-flops**, and ultimately **processors and memory units**.

Logic gates are the fundamental building blocks of digital electronics. They perform Boolean operations on binary inputs to produce a binary output. Below are the most commonly used gates:

1. **AND Gate**
   The AND gate outputs 1 only if all inputs are 1.
   Logic Expression: $Y = A \cdot B$

A	B	Y (A · B)
0	0	0
0	1	0
1	0	0
1	1	1

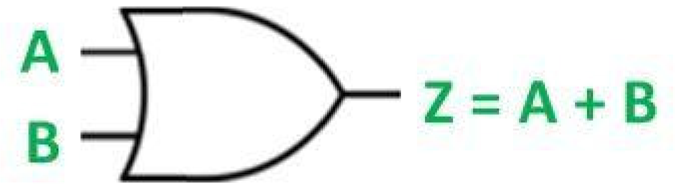


2. OR Gate

The OR gate outputs 1 if any input is 1.

Logic Expression: Y = A + B

A	B	Y (A + B)
0	0	0
0	1	1
1	0	1
1	1	1

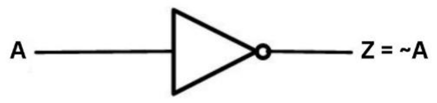


3. NOT Gate

The NOT gate inverts the input (0 becomes 1, and 1 becomes 0).

Logic Expression: Y = ¬A

A	Y (¬A)
0	1
1	0

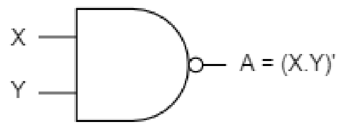


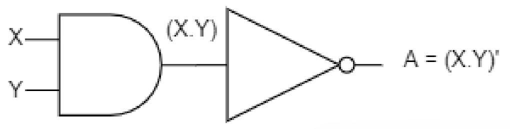
4. NAND Gate

The NAND gate is the inverse of the AND gate.

Logic Expression: Y = ¬ (A · B)

A	B	Y (¬ (A · B))
0	0	1
0	1	1
1	0	1
1	1	0





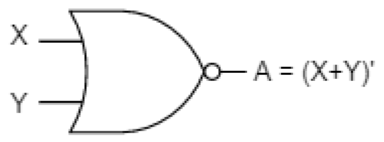
5. NOR Gate

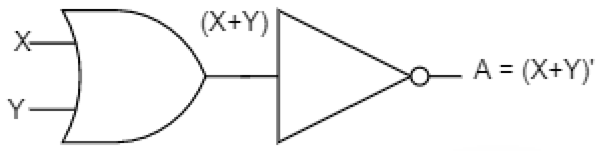
The NOR gate is the inverse of the OR gate.

Logic Expression: Y = ¬ (A + B)

A	B	Y (¬ (A + B))

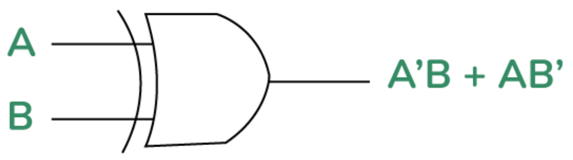
0	0	1
0	1	0
1	0	0
1	1	0





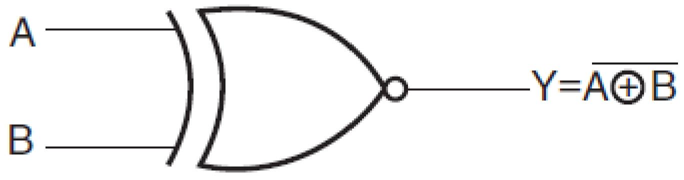
6. **XOR Gate**
The XOR gate outputs 1 only if the inputs are different.
Logic Expression: $Y = A \oplus B$

A	B	Y (A ⊕ B)
0	0	0
0	1	1
1	0	1
1	1	0



7. **XNOR Gate**
   The XNOR (Exclusive-NOR) gate is the complement of the XOR gate. It outputs 1 only if the inputs are the same.
   Logic Expression: Y = ¬ (A ⊕ B) or Y = A ⊙ B

A	B	Y (¬ (A ⊕ B))
0	0	1
0	1	0
1	0	0
1	1	1



$$Y = (\overline{A \oplus B}) = (A.B + \overline{A}.\overline{B})$$

Why Are We Using Cedar Logic Simulator?

Cedar Logic Simulator is used in this practical to help students visually understand and design digital circuits using logic gates. It provides a simple drag-and-drop interface with real-time simulation, making it ideal for beginners. By eliminating hardware constraints, students can focus on learning logic design concepts effectively. It also saves lab time, supports experimentation, and bridges the gap between theoretical learning and practical application.

What is Cedar Logic Simulator?

Cedar Logic Simulator is a free, open-source, **digital logic design and simulation tool**. It allows users to design logic circuits graphically by placing logic gates, connecting them with wires, and observing real-time behavior of the circuit. It's ideal for beginners learning digital logic design.

Installation Steps for Cedar Logic Simulator:

1. **Download Cedar Logic**:
   - Visit the official GitHub page:
   - Or search: "Cedar Logic Simulator GitHub" in your browser.
2. **Download the Installer**:
   - Download the .zip file or Windows installer (CedarLogicSetup.exe).
3. **Install the Software**:
   - Extract the .zip file or run the .exe file.
   - Follow the on-screen instructions to install it.
4. **Launch Cedar Logic**:
   - Open the simulator.
   - Use the toolbar to add gates, switches, and output LEDs.
5. **Begin Simulation**:
   - Build your circuit by dragging gates onto the canvas.
   - Connect them using wires.
   - Add inputs (switches) and outputs (LEDs).
   - Toggle switches to see real-time output.

Practical Applications of Logic Gates:

- Designing **Arithmetic Circuits**: Adders, subtractors.
- Building **Decision-Making Units**: Comparators, encoders, decoders.
- Implementing **Storage and Memory**: Flip-flops, registers.
- Creating **Control Logic** in CPUs and embedded systems.
- Used in **communication systems, robotics, AI processors**, and **IoT devices**.

Solution:

Conclusion: We designed and simulated digital circuits using various logic gates, which helped us visualize and understand how logical operations are performed in hardware.

Experiment No. 4
Implement a program to perform addition and subtraction of binary, hexadecimal number
Date of Performance:
Date of Correction:

Aim: To implement a program that performs addition and subtraction operations on binary and hexadecimal numbers.

Objective: To understand and implement arithmetic operations on binary and hexadecimal numbers commonly used in computer systems.

Theory:

Introduction:

In digital systems and low-level computing, binary (base-2) and hexadecimal (base-16) number systems are widely used. Binary numbers represent the core data format of all computing systems, while hexadecimal offers a compact, human-readable form of binary. Understanding how to perform arithmetic operations (like addition and subtraction) in these systems is essential for tasks like instruction processing, memory addressing, and debugging.

Binary Number System (Base-2):

- Uses only two digits: 0 and 1.
- Each binary digit (bit) represents an increasing power of 2.
- Computers use binary because it maps directly to digital electronics: ON (1) and OFF (0).

Binary Addition Rules:

A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	1	0	1	0
1	1	0	0	1
1	1	1	1	1

Binary Subtraction Rules:

A	B	Borrow In	Difference	Borrow Out
0	0	0	0	0
1	0	0	1	0
0	1	0	1	1
1	1	0	0	0

Subtraction can also be implemented by adding the 2's complement of the subtrahend.

Hexadecimal Number System (Base-16):

- Uses digits 0–9 and letters A–F (A=10 to F=15).
- Each hex digit maps to 4 binary bits (e.g., A = 1010).
- Used in memory addresses, machine-level instructions, and debugging.

Hexadecimal Arithmetic:

- Addition and subtraction follow rules similar to decimal but use base-16.
- If a sum exceeds 15 (F), a carry is generated.
- Subtraction may involve borrowing if the subtrahend digit is greater than the minuend digit.

Examples:

Binary Addition:

  1011 (11)

+ 1101 (13)

= 11000 (24)

Hexadecimal Addition:

  A9

+ 6F

= 118 (carry handled)

Applications:

- Processor ALUs perform these arithmetic operations at the hardware level.
- Memory management often requires binary or hex arithmetic for address calculations.
- Instruction encoding in microcontrollers involves operations in hex.
- Essential in assembly language programming, debugging, and system-level diagnostics.

Solution:

Conclusion: We explored how binary and hexadecimal numbers are used in arithmetic operations and gained hands-on experience in how processors perform basic calculations.

Experiment No. 5
Implement restoring division algorithm
Date of Performance:
Date of Correction:

Aim: To implement a program that performs binary division using the restoring division algorithm.

Objective: To understand and implement the restoring division algorithm used in digital systems for binary division operations.

Theory:

Introduction:

In computer architecture, binary division is a fundamental arithmetic operation performed by the Arithmetic Logic Unit (ALU) of a processor. Unlike multiplication and addition, which are relatively straightforward, division in hardware requires more complex logic.

The Restoring Division Algorithm is one such technique used for implementing binary division in hardware. It simulates long division, similar to how we do it manually, but in binary form using shifting and subtraction.

How the Restoring Division Algorithm Works:

The algorithm operates on unsigned binary integers and performs division by:

- Shifting the dividend left.
- Subtracting the divisor from the partial remainder.
- If the result is negative, it restores the previous value by adding the divisor back (hence the name *restoring*).
- A 0 is placed in the quotient in this case; otherwise, a 1 is placed.

Step-by-Step Explanation:

Let's define:

- A (Accumulator / Remainder) — initialized to 0.
- Q (Dividend) — the number to be divided.
- M (Divisor) — the number by which we divide.

- n — number of bits.

Steps:

1. Initialize A = 0, and Q contains the dividend.
2. Repeat n times:
    - Left shift A and Q (combine them as AQ).
    - A = A − M
    - If A < 0:
        - Set $Q_0 = 0$
        - A = A + M (restore A)
    - Else:
        - Set $Q_0 = 1$
3. After n cycles, Q holds the quotient and A holds the remainder.

Example (4-bit):

Let's divide 13 (1101) by 3 (0011):

- Q = 1101, M = 0011, A = 0000
- Shift AQ left → Subtract M from A
- If result negative → Restore A, set $Q_0 = 0$
- Else → Keep A, set $Q_0 = 1$
- Repeat for 4 cycles
- Final Q = Quotient, A = Remainder

Applications in Computer Architecture:

- Used in hardware-level division in older processors.
- Foundation for understanding non-restoring and SRT division algorithms.
- Useful for teaching binary arithmetic, control logic, and ALU design.
- Important in embedded systems and custom hardware where standard division instructions may not be available.

Although modern CPUs often use faster or more optimized algorithms (like non-restoring or Newton-Raphson), restoring division is crucial for understanding the basics of binary division in digital circuits.

Solution:

Conclusion: We implemented the restoring division algorithm and understood how binary division is performed in hardware using subtraction and shifting techniques.

Experiment No. 6
Implement Non-Restoring division algorithm
Date of Performance:
Date of Correction:

Aim: To implement a program that performs binary division using the non-restoring division algorithm.

Objective: To understand and implement the non-restoring division method for binary division used in digital arithmetic units.

Theory:

Introduction:

In digital systems and computer architecture, division is one of the essential arithmetic operations executed by the **Arithmetic Logic Unit (ALU)**. While restoring division is simple and intuitive, it involves extra steps to restore the original value of the remainder when a subtraction leads to a negative result. To improve performance and reduce hardware complexity, the **non-restoring division algorithm** was developed.

The **non-restoring division algorithm** eliminates the "restore" step used in the restoring division method by keeping track of the sign of the remainder and adjusting the next operation accordingly. This saves hardware time and reduces the number of operations required for division.

Key Concepts:

Let's define:

- **Q (Dividend)**
- **M (Divisor)**
- **A (Accumulator / Remainder)**
- **n (number of bits)**

In non-restoring division:

- If the remainder is **positive**, subtract the divisor.
- If the remainder is **negative**, add the divisor.
- After each operation, shift the partial result left and update the quotient based on the new remainder's sign.

Steps of Non-Restoring Division Algorithm:

1. Initialize A = 0 and Q = dividend, M = divisor.
2. Repeat the following for **n** bits:
    - If A is **positive**, A = A − M
    - If A is **negative**, A = A + M
    - Shift A and Q left (logical shift)
    - If A is **positive**, set $Q_0 = 1$
    - If A is **negative**, set $Q_0 = 0$
3. After n iterations:
    - If A < 0, add M to A (final correction)
4. Q = quotient, A = remainder

Comparison with Restoring Division:

Feature	**Restoring Division**	**Non-Restoring Division**
Handles negative A	By restoring (A = A + M)	By switching to addition/subtraction
Number of operations	More due to restore step	Fewer; no explicit restore required
Hardware complexity	Slightly higher	More efficient for real-time systems

Example:

Let's divide **Q = 1101 (13)** by **M = 0011 (3)**:

- A starts as 0000
- Loop for 4 bits:

- Subtract M from A if $A \geq 0$
- Add M to A if $A < 0$
- Shift A-Q left
- Set $Q_0 = 1$ if $A \geq 0$ else 0

- At the end, A = remainder, Q = quotient

Applications:

- Implemented in **ALUs** of modern processors.
- Used in **hardware accelerators**, **FPGA arithmetic modules**, and **embedded systems**.
- Forms the basis for **faster division algorithms**, including **SRT** and **Booth's** algorithms.
- Important for developing **custom arithmetic units** in VHDL/Verilog and digital simulation platforms.

Solution:

Conclusion: We implemented the non-restoring division algorithm and realized how it improves efficiency by avoiding unnecessary restore steps during division.

Experiment No. 7
Implement 8068 assembly language program for performing various arithmetic operation
Date of Performance:
Date of Correction:

Aim: To write and execute 8086 Assembly Language Programs (ALPs) that perform basic arithmetic operations such as addition, subtraction, multiplication, and division.

Objective: To understand how arithmetic operations are performed at the microprocessor level using 8086 assembly instructions.

Theory:

Introduction to 8086 Microprocessor:

The **Intel 8086** is a 16-bit microprocessor and a foundation for the x86 family of processors. It supports various arithmetic, logical, and control instructions that allow programmers to directly manage the CPU's registers, memory access, and I/O ports.

8086 has:

- 16-bit ALU (Arithmetic Logic Unit)
- 16-bit registers like AX, BX, CX, DX (general-purpose)
- Segment registers: CS, DS, SS, ES
- Instruction Pointer (IP), Flags register
- Supports both byte (8-bit) and word (16-bit) operations

Arithmetic Instructions in 8086:

The 8086 instruction set provides several **arithmetic instructions** that operate on registers or memory operands.

Instruction	**Description**
ADD	Adds two operands
SUB	Subtracts source from destination
INC	Increments operand by 1
DEC	Decrements operand by 1
MUL	Unsigned multiplication

IMUL	Signed multiplication
DIV	Unsigned division
IDIV	Signed division
NEG	Changes sign of the operand (2's complement)

These instructions can work with:

- Register-to-register (e.g., ADD AX, BX)
- Immediate-to-register (e.g., SUB AX, 0005H)
- Memory-to-register or register-to-memory (e.g., ADD AL, [5000H])

Arithmetic Operations Explained:

1. **Addition (ADD)**: Adds values and updates flags like Carry (CF), Overflow (OF), Zero (ZF).
    - Example: ADD AX, BX → AX = AX + BX
2. **Subtraction (SUB)**: Subtracts source from destination.
    - Example: SUB AX, BX → AX = AX - BX
3. **Multiplication (MUL/IMUL)**: Multiplies accumulator with operand.
    - Example: MUL BX → AX = AL * BX (for 8-bit); DX:AX = AX * BX (for 16-bit)
4. **Division (DIV/IDIV)**: Divides accumulator by operand.
    - Result stored in AX (quotient) and DX (remainder) for 16-bit division.

Why Learn Arithmetic in Assembly?

- Provides **low-level control** over hardware.
- Helps understand **how compilers translate high-level arithmetic** into machine code.

- Essential for **embedded systems**, **OS development**, and **microcontroller programming**.
- Used in writing **bootloaders**, **interrupt handlers**, and **system utilities**.

Tools Used:

- **EMU8086 / MASM / TASM** (assemblers and emulators for 8086)
- Simulators allow students to write, run, and debug ALPs.

Solution:

Conclusion: We learned how to write assembly programs for arithmetic operations and understood how low-level instructions work with registers in the 8086 microprocessor.

Experiment No. 8
Implement 8086 assembly language program for performing sorting of array
Date of Performance:
Date of Correction:

Aim: To implement an 8086-assembly language program that sorts an array of numbers in ascending or descending order.

Objective: To understand and apply sorting logic at the assembly language level using 8086 microprocessor instructions.

Theory:

Introduction:

Sorting is a basic yet vital operation in computer science used to organize data for faster searching, better readability, and optimal storage. In high-level languages, we often use built-in functions for sorting, but at the **assembly language** level, the programmer must implement the logic explicitly using low-level instructions and register operations.

In this practical, we will write an assembly program using **8086 microprocessor instructions** to sort an array of numbers. The commonly used sorting technique in assembly is **Bubble Sort** due to its simple logic and suitability for step-by-step implementation using loops and comparisons.

8086 Microprocessor Recap:

The Intel 8086 is a **16-bit processor** with a rich instruction set for:

- **Data transfer**
- **Arithmetic and logical operations**
- **Looping and branching**
- **Memory addressing and pointer manipulation**

In sorting, we use instructions such as:

- MOV – move data
- CMP – compare operands
- JAE, JB, JE, etc. – conditional jumps
- XCHG – swap values
- LOOP – loop control

- Registers: SI, DI, CX, AX, BX, etc.

Sorting Algorithm (Bubble Sort) Logic:

Bubble Sort works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order.

Steps:

1. Load the array into memory.
2. Use a loop to traverse the array.
3. Compare each pair of adjacent elements.
4. Swap them if they are in the wrong order.
5. Repeat this process n-1 times (for n elements).

Example (Ascending Order):
Input Array: [6, 3, 9, 1]
Pass 1: [3, 6, 1, 9]
Pass 2: [3, 1, 6, 9]
Pass 3: [1, 3, 6, 9]

Bubble sort is simple and easy to implement in assembly due to its reliance on **pairwise comparisons** and **adjacent swaps**.

Real-World Applications:

- Understanding how high-level sorting functions work at machine level.
- Developing firmware for devices with limited computing resources.
- Learning how **memory and registers interact** in assembly language.
- Enhancing skills in **pointer arithmetic**, **conditional jumps**, and **loop structures**.
- Essential for **embedded systems**, **OS-level programming**, and **microcontroller firmware**.

Solution:

Conclusion: We implemented a sorting algorithm in 8086 assembly language, which helped us understand the use of loops, comparisons, and memory access at the processor level.

Experiment No. 9
Implement a program to demonstrate various data transfer techniques
Date of Performance:
Date of Correction:

Aim: To implement a program that demonstrates various data transfer techniques used in computer systems for moving data between memory, registers, and I/O devices.

Objective: To understand and implement different data transfer methods used in computer architecture such as programmed I/O, interrupt-driven I/O, and direct memory access (DMA).

Theory:

Introduction:

In computer systems, data transfer refers to the movement of data between different components such as:

- CPU ↔ Memory
- CPU ↔ I/O devices
- Memory ↔ I/O devices

Efficient data transfer is essential for system performance, especially when interacting with high-speed or real-time devices. There are three primary data transfer techniques used in computer architecture:

1. Programmed I/O (Polling Method):

In this method:

- The CPU is in control of all I/O operations.
- The CPU continuously polls (checks) the status of I/O devices.
- When the device is ready, the CPU transfers data between the I/O port and registers/memory.

Pros:

- Simple to implement.

Cons:

- CPU remains busy in checking device status.
- Inefficient for high-speed or multiple I/O devices.

Example in Code:
The program reads or writes data by directly accessing I/O ports or memory addresses.

2. Interrupt-Driven I/O:

In this method:

- The CPU does not poll the I/O devices continuously.
- I/O devices send an interrupt signal to the CPU when they are ready.
- The CPU then pauses current execution and handles the interrupt via an Interrupt Service Routine (ISR).

Pros:

- CPU time is utilized more efficiently.
- More responsive for real-time systems.

Cons:

- Slightly more complex to implement than programmed I/O.

Real-life analogy: A doorbell. Instead of checking the door repeatedly, you get notified when someone is at the door.

3. Direct Memory Access (DMA):

In this method:

- A separate DMA controller is used to transfer data directly between memory and I/O devices without CPU involvement.
- The CPU only initiates the transfer and gets notified once it is complete.

Pros:

- Efficient for bulk data transfer.
- Frees up the CPU to do other tasks during data transfer.

Cons:

- Requires additional hardware (DMA controller).

Use cases: Transferring large data blocks like disk read/write operations, audio/video streaming.

Importance in Computer Architecture:

- These techniques allow efficient utilization of the processor and I/O devices.
- They play a crucial role in system performance, especially in embedded systems, real-time applications, OS kernel development, and device drivers.
- Understanding these methods is essential for designing interrupt controllers, I/O-mapped hardware, and low-level software.

Solution:

Conclusion: We explored and simulated different data transfer methods such as programmed I/O, interrupts, and DMA, and understood their roles in efficient communication between components.

Experiment No. 10
Implement a program to demonstrate the concept of pipeline
Date of Performance:
Date of Correction:

Aim: To implement a program that demonstrates the concept of instruction pipelining in computer architecture.

Objective: To understand and simulate the concept of pipelining used in modern processors to improve instruction throughput and overall CPU performance.

Theory:

Introduction:

In traditional (non-pipelined) processors, each instruction is processed one at a time—fetch, decode, execute, and write-back. This results in significant idle time for various parts of the CPU during each cycle.

Pipelining is a technique that allows overlapping of these stages, like an assembly line, so multiple instructions are in different phases of execution simultaneously. This improves the CPU's throughput (number of instructions completed per unit time) without increasing clock speed.

Pipelining Stages:

A basic instruction pipeline has the following stages:

Stage	**Description**
IF (Fetch)	Instruction is fetched from memory
ID (Decode)	Instruction is decoded, operands identified
EX (Execute)	Operation is performed
MEM	Memory is accessed (if needed)
WB (Write Back)	Result is written to register/memory

In pipelining, while one instruction is in the "Execute" stage, the next may already be in the "Decode" stage, and a third one in the "Fetch" stage.

Visual Example (3-Stage Pipeline):

Clock Cycle	**Instruction 1**	**Instruction 2**	**Instruction 3**
1	**Fetch**		
2	**Decode**	**Fetch**	
3	**Execute**	**Decode**	**Fetch**
4		**Execute**	**Decode**
5			**Execute**

This overlapping results in faster instruction execution overall.

Types of Pipelining:

1. **Instruction Pipelining –** Overlaps fetch-decode-execute stages.
2. **Arithmetic Pipelining –** Used in floating-point units to process parts of arithmetic operations.
3. **Processor-Level Pipelining –** Multiple cores with pipelined execution for parallelism.

Challenges in Pipelining:

- Data Hazards: When an instruction depends on the result of a previous one.
- Control Hazards: Occur due to branch or jump instructions.
- Structural Hazards: When hardware resources are insufficient to execute all stages concurrently.

These can be handled using:

- Stalling
- Forwarding (Bypassing)
- Branch Prediction

Real-World Importance:

- All modern processors (Intel, AMD, ARM) use pipelining to enhance performance.
- Forms the basis of superscalar and parallel processing architectures.
- Essential concept in compiler optimization, instruction scheduling, and microarchitecture design.

Solution:

Conclusion: We simulated instruction pipelining and observed how overlapping stages can improve instruction throughput and processor performance.