COMPUTER ORGANIZATION

IV B.Tech. - I Semester

Department of Computer Science and Engineering

GAYATRI VIDYA PARISHAD COLLEGE OF ENGINEERING FOR WOMEN
www.gvpcew.ac.in(Approved by AICTE & Affiliated to JNTUK)Estd. – 2008
Gandhi Nagar, Madhurawada, Visakhapatnam, Andhra Pradesh 530048.
**Course Objectives & Outcomes**

**Course Objectives:**--

- Comprehensive knowledge of computer system including the analysis and design of components of the system
- Understanding RTL, Micro operations, ALU, Organization of stored program computer, types of instructions and design of basic components of the system
- Illustration of data paths and control flow for sequencing in CPUs, Microprogramming of control unit of CPU
- Illustration of algorithms for basic arithmetic operations using binary and decimal representation
- Description of different parameters of a memory system, organization and mapping of various types of memories
- Describes the means of interaction devices with CPU, their characteristics, modes and introduction multiprocessors.

**Course Outcomes:**--

After completing this Course, the student should be able to:

Understand the architecture of a modern computer with its various processing units. Also the performance measurement of the computer system. In addition to this the management system of computer.

Students have a thorough understanding of the basic structure and operation of a digital computer.

- Able to discuss in detail the operation of the arithmetic unit including the algorithms & implementation of fixed-point and floating-point addition, subtraction, multiplication & division.
- Able to discuss in detail the operation of the arithmetic unit including the algorithms & implementation of fixed-point and floating-point addition, subtraction, multiplication & division.
- Students have a thorough understanding of Micro Program Control
- Students can calculate the effective address of an operand by addressing modes
- Understanding of how a computer performs arithmetic operation of positive and negative numbers.
- Explain the function of each element of a memory hierarchy, Cache memory and its importance.
- Students can understand how cache mapping occurs in computer and can solve various problems related to this.
- Study the hierarchical memory system including cache memories and virtual memory.
- Able to identify and compare different methods for computer I/O
- Able to discuss about advantages of parallel processing, multiprocessors.
1. Syllabus

UNIT-I

COMPUTER ARITHMETIC: Addition and subtraction, multiplication algorithms, Division Algorithms, Floating point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

UNIT-II
REGISTER TRANSFER LANGUAGE AND MICRO-OPERATIONS:

CENTRAL PROCESSING UNIT - Stack organization. Instruction formats. Addressing modes. DATA Transfer and manipulation. Program control. Reduced Instruction set computer

UNIT-III
MICRO PROGRAMMED CONTROL: Control memory, Address sequencing, micro program example, Design of control unit-Hard wired control. Micro programmed control.

UNIT-IV
THE MEMORY SYSTEM: Memory Hierarchy, Main memory, Auxiliary memory, Associative memory, Cache memory, Virtual memory, Memory management hardware

UNIT-V
INPUT-OUTPUT ORGANIZATION : Peripheral Devices, Input-Output Interface, Asynchronous data transfer Modes of Transfer, Priority Interrupt, Direct memory Access, Input –Output Processor (IOP), Serial Communication;

UNIT-VI

Text Books:

Reference Books:
## Lecture Plan

<table>
<thead>
<tr>
<th>Lecture no.</th>
<th>Unit Number</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>I</td>
<td>Introduction to Computer architecture and Organization</td>
</tr>
<tr>
<td>2.</td>
<td>I</td>
<td>Computer Types, functional Units and Basic Operational Concepts, Bus structure, software, performance multi processor and multi computer</td>
</tr>
<tr>
<td>3.</td>
<td>I</td>
<td>Data Types ,Number systems, octal and hexadecimal nos, decimal and alphanumeric representation</td>
</tr>
<tr>
<td>4.</td>
<td>I</td>
<td>Complments-10’s and 9’s Complements; 2’s ans 1’s Complements; Subtraction of unsigned Nos; fixed point representation and Integer representation;</td>
</tr>
<tr>
<td>5.</td>
<td>I</td>
<td>Addition of two numbers in signed magnitude system; 2’s complement addition, subtraction; overflow and overflow detection and decimal fixed point representation</td>
</tr>
<tr>
<td>6.</td>
<td>I</td>
<td>Floating point representation&amp; Error detection codes</td>
</tr>
<tr>
<td>7.</td>
<td>I</td>
<td>Introduction to computer Arithmetic, addition and subtraction algorithm for signed magnitude numbers and hardware implementation; addition and subtraction for signed 2’s complement data</td>
</tr>
<tr>
<td>8.</td>
<td>I</td>
<td>Multiplication algorithm: Hardware implementation for signed magnitude data, hardware &amp; algorithm;</td>
</tr>
<tr>
<td>9.</td>
<td>I</td>
<td>Booth multiplication algorithm; Array multiplier</td>
</tr>
<tr>
<td>10.</td>
<td>I</td>
<td>Division Algorithm: Hardware implementation for signed magnitude data, hardware algorithm and other algorithm.</td>
</tr>
<tr>
<td>11.</td>
<td>I</td>
<td>Floating point Arithmetic operations: Basic considerations, register configurations, addition, subtraction, multiplication and division</td>
</tr>
<tr>
<td>12.</td>
<td>I</td>
<td>Decimal Arithmetic Unit: BCD Adder and BCD Subtraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>14.</td>
<td>I</td>
<td>Hardware and algorithm for Decimal Arithmetic Operations: multiplication and division, floating point operations</td>
</tr>
<tr>
<td>15.</td>
<td>II</td>
<td>Register transfer Language, Register Transfer, Bus and Memory Transfer</td>
</tr>
<tr>
<td>16.</td>
<td>II</td>
<td>Arithmetic Micro operations; Logic Micro operations</td>
</tr>
<tr>
<td>17.</td>
<td>II</td>
<td>Shift micro operations, Arithmetic logic shift unit</td>
</tr>
<tr>
<td>18.</td>
<td>II</td>
<td>Instruction Codes, Computer register</td>
</tr>
<tr>
<td>19.</td>
<td>II</td>
<td>Computer Instruction and format, Instruction Cycle</td>
</tr>
<tr>
<td>20.</td>
<td>II</td>
<td>Memory Reference Instructions, Input-Output and Interrupt</td>
</tr>
<tr>
<td>21.</td>
<td>II</td>
<td>Stack Organization, register Stack, memory stack, Reverse polish notation, Conversion to RPN, Evaluation of arithmetic expression</td>
</tr>
<tr>
<td>22.</td>
<td>II</td>
<td>Instruction Formats, 3-types of CPU organization, 3,2,1,0 address instruction formats, RISC Instructions</td>
</tr>
<tr>
<td>23.</td>
<td>II</td>
<td>Addressing modes; Data Transfer and manipulation instructions</td>
</tr>
<tr>
<td>24.</td>
<td>II</td>
<td>Program Control Instructions, Subroutine call and return instructions, Program interrupts and types of interrupts</td>
</tr>
<tr>
<td>25.</td>
<td>II</td>
<td>RISC Vs CISC; Overlapping Register windows; Berkeley RISCI</td>
</tr>
<tr>
<td>26.</td>
<td>III</td>
<td>Control memory</td>
</tr>
<tr>
<td>27.</td>
<td>III</td>
<td>Address Sequencing</td>
</tr>
<tr>
<td>28.</td>
<td>III</td>
<td>Micro Program Example: computer configuration, instruction format and symbolic microinstructions; The fetch routine, symbolic microprogram and binary microprogram</td>
</tr>
<tr>
<td>29.</td>
<td>III</td>
<td>Design of Control Unit: Hard wired control and Micro programmed control</td>
</tr>
<tr>
<td>30.</td>
<td>IV</td>
<td>Memory Hierarchy, Main memory: RAM and ROM chips, memory address map, memory connection to CPU; Auxiliary memory: Magnetic Disks and Magnetic Tapes</td>
</tr>
<tr>
<td>31.</td>
<td>IV</td>
<td>Associative Memory: Hardware Organization, Match Logic, read operation and write operation</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
<td>Topics</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>32.</td>
<td>IV</td>
<td>Direct mapping, Set-Associative Mapping, Writing into cache and cache initialization</td>
</tr>
<tr>
<td>33.</td>
<td>IV</td>
<td>Virtual memory: Address Space and Memory Space, Address Mapping using pages; Associative memory page table, Page replacement</td>
</tr>
<tr>
<td>34.</td>
<td>IV</td>
<td>Memory management hardware: Segmented page mapping, Numerical example, Memory Protection</td>
</tr>
<tr>
<td>35.</td>
<td>V</td>
<td>Peripheral Devices, I/O Interface: I/O bus and interface modules; I/O vs Memory Bus, Isolated vs memory mapped I/O, Examples of I/O Interface</td>
</tr>
<tr>
<td>36.</td>
<td>V</td>
<td>Asynchronous Data Transfer: Strobe Control, handshaking, asynchronous serial transfer; Asynchronous Commn Interface, First-in-first out buffer</td>
</tr>
<tr>
<td>37.</td>
<td>V</td>
<td>Modes of Transfer: Examples of Programmed I/O, Interrupt initiated I/O software considerations; Priority Interrupt: Daisy Chaining Priority, parallel priority Interrupt, priority encoder,</td>
</tr>
<tr>
<td>38.</td>
<td>V</td>
<td>Interrupt cycle, software routines and initial and final operations, Direct Memory Access: DMA Controller and DMA Transfer</td>
</tr>
<tr>
<td>39.</td>
<td>V</td>
<td>Input Output Processor(IOP): CPU –IOP Communication,</td>
</tr>
<tr>
<td>40.</td>
<td>V</td>
<td>IBM 370 I/o channel, Intel 8089 IOP</td>
</tr>
<tr>
<td>41.</td>
<td>V</td>
<td>Serial Communication: Character oriented protocol,</td>
</tr>
<tr>
<td>42.</td>
<td>V</td>
<td>Transmission example, Data transparency and Bit-oriented protocol</td>
</tr>
<tr>
<td>43.</td>
<td>VI</td>
<td>Parallel Processing</td>
</tr>
<tr>
<td>44.</td>
<td>VI</td>
<td>Pipelining, general considerations</td>
</tr>
<tr>
<td>45.</td>
<td>VI</td>
<td>Arithmetic Pipeline, Instruction Pipeline</td>
</tr>
<tr>
<td>46.</td>
<td>VI</td>
<td>RISC Pipeline; Vector Processing: vector operations, matrix multiplication, memory interleaving,</td>
</tr>
<tr>
<td>47.</td>
<td>VI</td>
<td>superscalar processors and super computers</td>
</tr>
<tr>
<td>48.</td>
<td>VI</td>
<td>Array Processors: Attached Array Processor, SIMD Array Processor</td>
</tr>
<tr>
<td>49.</td>
<td>VI</td>
<td>Characteristics of multi processors</td>
</tr>
<tr>
<td>50.</td>
<td>VI</td>
<td>Interconnection Structure: Timeshared Common Bus, Multiport Memory, Crossbar Switch, Multistage switching network, Hypercube Interconnection</td>
</tr>
<tr>
<td>51.</td>
<td>VI</td>
<td>Inter processor arbitration: serial arbitration procedure, parallel arbitration logic and dynamic arbitration algorithms</td>
</tr>
<tr>
<td>52.</td>
<td>VI</td>
<td>Inter processor communication and synchronization, cache coherence</td>
</tr>
</tbody>
</table>

**Unit – I – Basic structure of computers and Computer arithmetic**


**COMPUTER ARITHMETIC:** Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

**1.1.1. Unit Objectives:**

After reading this Unit, the reader should be able to understand:

- The definition of computer architecture, organization and computer hardware.
- The design aspects of computer hardware and software.
- Functions provided by a digital computer and functional units of a digital computer.
- Processes involved in executing a task, instruction types, connections between memory and processor and data transfer mechanism between memory and processor.
- Processor registers and their functions interrupt and interrupt service routines.
- Bus, single bus, buffer registers, functions of system software and multiprogramming, processor performance and performance parameters.
- Multi Processors and multi computers.
- Data types and their representation, number system and their representation.
- Using complements for subtraction.
- Fixed point and floating point representation of signed numbers.
- Error generation, error detection and error correction codes.
1.1.2. Unit Outcomes:

- Student is able to differentiate the various computer types, hardware Vs software, RISC Vs CISC.

- Student is able to give an outline of functional parts of the computer.

- Student is able to explain basic operational concept of a computer, bus structure & software.

- Student is able to bring out the various performance parameters of a computer.

- Student is able to give representation of various data types in Binary Coded format.

- Student is able to do arithmetic operations on signed, fixed point & floating point numbers using complements.

- Student is able to develop algorithms for arithmetic operations in fixed point & floating point and decimal numbers and design hardware circuits for them.

- Student is able to make error detection codes and build error detection circuits.
1.1.2.5. Lecture-5

**Floatinpoint number representation:**

Fixed point numbers represents integers and floating point numbers represent real numbers.

**Numeric Format:**

A number expressed in scientific notation has a **sign**, a **fraction or significant** (or mantissa) and an **exponent**.

Ex: The number is: -1234.5678

Scientific notation is: \(-1.2345678 \times 10^3\). Here the sign is negative, the significant is 1.2345678 and exponent is 3 and the base is 10. Computers use base as 2.

Disadvantages of Scientific notation:
Most of the numbers can be expressed in many different ways. Ex: $-1.2345678 \times 10^3 = -1234567.8 \times 10^{-3}$ etc. Computers are more efficient and have much simpler hardware if each number is uniquely represented.

Normalization as Solution to the problem:

The floating point number must be normalized, that is, each number's significant is a fraction with no leading zeros. Thus the only valid floating point representation for $-1234.5678$ is $-1.2345678 \times 10^4$. Note IEEE 754 uses an exception for this rule.

Special cases:

The number zero has only zeros in its significand and cannot be normalized. For this reason a special value is assigned to zero. Arithmetic algorithms must explicitly check for zero values and treat them as special cases. $+\infty$ and $-\infty$ also have special representation and require special treatment.

NaN:

NaN means Not a Number. It represents the result of illegal operations, such as $\infty \div \infty$ or taking the square root of a negative number. As with zero and infinity, NaN requires a special treatment in floating point arithmetic algorithms.

A predefined format for computer storage of floating point number:

Each number is stored in its normal form.

Take the number: $X = -1234.5678$; That is $X = X_S \times X_F \times X_E$

$X_S$ is the sign of $X$; $X_F$ is its significand and $X_E$ is its exponent

Since the radix point is located to the left of the most significant bit of the significand, the radix point is not stored. Thus the value $X = -1234.5678$ would be stored as $X_S = 1$, $X_F = 12345678$ and $X_E = 4$

Biasing:

In the above representation for exponent, there is no sign bit for exponent. We can use 2's complement form but prevalent practice is to use biasing.

If $X_E$ has 4 bits, then it can represent 16 items. That is the numbers from -8 to +7. To do this, a set bias value is added to the actual exponent. The result is ten stored in $X_E$. For this the bias should be set to 8.

The smallest possible exponent, -8, is represented as $-8 + \text{bias} = -8 + 8 = 0$ or 0000 in binary.

The largest possible exponent, +7, is represented as $+7 + \text{bias} = +7 + 8 = 15 = 1111$ in binary. The arithmetic algorithms must account for the bias when generating their results.

Characteristics of floating point numbers:
The characteristics are 1. Precision  2. Gap 3. Range

**Precision:**

It characterizes how precise a floating point value can be. *It is defined as the number of bits in the significand*. The greater the number of bits in the significand, the greater is the CPU’s precision and the more precise is it’s value. Many CPUs have 2 representations for floating point numbers. They are called single precision and double precision here double precision has twice the number of bits.

**Gap:**

The gap is the difference between two adjacent values. It’s value depends on the value of the exponent.

Take the number: \( X = .10111010 \times 2^3 \).

It’s adjacent values are: \( .10111001 \times 2^3 \) and \( .10111011 \times 2^3 \).

Each number produce a gap of \( .00000001 \times 2^3 \).

In general the gap for floating point value \( X \) can be expressed as \( 2^{(X_e - \text{precision})} \)

**Range:**

The range of a floating point representation is bounded by it’s smallest and largest possible values.

**Overflow and underflow:**

Overflow occurs when an operation produces a result that can not be stored in computers’s floating point registers. Underflow occurs when an operation produces a result between zero and either the positive or negative smallest possible value.

**IEEE 754 Floating point standard:**

This standard specifies 2 precision for floating point numbers which are called single precision and double precision floating point representations.

**Single Precision Format:**

This format has 32 bits. 1 bit for sign; 8 bits for the exponent; 23 for the significand. The significand also includes an implied 1 to the left of its radix point( except for special values and denormalized numbers).
Error Detection codes:

Information is stored as binary codes and are transmitted by serial or parallel communication. During transmission noise is added to the signal and it may change binary bits in the code from 1 to 0, and vice versa. An error detection code is a **binary code that detects digital errors during transmission**. The detected errors can not be corrected but their presence is indicated.

**Parity bit:**

The most common error detection code used is the parity bit. A parity bit is an extra bit included with a binary message to make the total number of 1’s either odd or even. If the message consists of n bits, then the error detection code consists of n+1 bits. If the bit added to the message makes the sum of 1’s odd in the error detection code, then the scheme is called odd-parity. If the sum of bits is even, the scheme is called even parity scheme.

<table>
<thead>
<tr>
<th>Message xyz</th>
<th>P(odd)</th>
<th>P(even)</th>
<th>Error detection code, odd parity</th>
<th>Error detection code, even parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>1</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td>1</td>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>0</td>
<td>0111</td>
<td>0110</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1</td>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>0</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td>1</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>
Parity Generator and Parity Checker:

Parity Checker:

Parity out from generator

Error

Indication
The circuit arrangement checks the occurrence of error any odd number of times. An even number of errors is not detected.

We note that \( P(\text{even}) \) function is the exclusive OR \( x, y, z \) because it is equal to 1 when either one or all 3 of the variables are equal to 1. The \( P(\text{odd}) \) function is the complement of the \( P(\text{even}) \) function.

Assume at the sending end the message bits and odd parity bit is generated. The EX-OR gates generate \( P(\text{even}) \) function and to generate \( P(\text{odd}) \), the complement of \( P(\text{even}) \) is used.

The 4 bits transmitted has an odd number of 1’s. If an error occurs during transmission, then the number of 1’s become even. Hence parity checker checks for even parity.

1.1.2.6. Lecture 6

COMPUTER ARITHMETIC:

Addition, subtraction, multiplication are the four basic arithmetic operations. Using these operations other arithmetic functions can be formulated and scientific problems can be solved by numerical analysis methods.

Arithmetic Processor:

It is the part of a processor unit that executes arithmetic operations. The arithmetic instructions definitions specify the data type that should be present in the registers used. The arithmetic instruction may specify binary or decimal data and in each case the data may be in fixed-point or floating point form.

Fixed point numbers may represent integers or fractions. The negative numbers may be in signed-magnitude or signed-complement representation. The arithmetic processor is very simple if only a binary fixed point add instruction is included. It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed and floating point representations.

Algorithm:

Algorithm can be defined as a finite number of well defined procedural steps to solve a problem. Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting an algorithm is a flowchart which consists of rectangular and diamond-shaped boxes. The computational steps are specified in the rectangular boxes and the decision steps are indicated inside diamond-shaped boxes from which 2 or more alternate path emerge.

Addition and Subtraction:

3 ways of representing negative fixed point binary numbers:
1. Signed-magnitude representation—used for the representation of mantissa for floating point operations by most computers.

2. Signed-1’s complement

3. Signed-2’s complement—Most computers use this form for performing arithmetic operation with integers

**Addition and subtraction algorithm for signed-magnitude data**

Let the magnitude of two numbers be A & B. When signed numbers are added or subtracted, there are 4 different conditions to be considered for each addition and subtraction depending on the sign of the numbers. The conditions are listed in the table below. The table shows the operation to be performed with magnitude (addition or subtraction) are indicated for different conditions.

<table>
<thead>
<tr>
<th>Sl.No</th>
<th>Operation</th>
<th>Add Magnitudes</th>
<th>Subtract magnitudes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>When A&gt; B</td>
</tr>
<tr>
<td>1</td>
<td>( +A ) + (+B)</td>
<td>+ (A + B)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( +A ) + (-B)</td>
<td></td>
<td>+(A-B)</td>
</tr>
<tr>
<td>3</td>
<td>( -A ) + (+B)</td>
<td></td>
<td>-(A-B)</td>
</tr>
<tr>
<td>4</td>
<td>( -A ) + (-B)</td>
<td></td>
<td>-(A + B)</td>
</tr>
<tr>
<td>5</td>
<td>( +A ) - (+B)</td>
<td></td>
<td>+(A-B)</td>
</tr>
<tr>
<td>6</td>
<td>( +A ) - (-B)</td>
<td></td>
<td>+(A + B)</td>
</tr>
<tr>
<td>7</td>
<td>( -A ) - (+B)</td>
<td></td>
<td>-(A + B)</td>
</tr>
<tr>
<td>8</td>
<td>( -A ) - (-B)</td>
<td></td>
<td>-(A-B)</td>
</tr>
</tbody>
</table>

The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithm for addition and subtraction (from the table above):

**Addition Algorithm:**

When the signs of A and B are identical, add two magnitudes and attach the sign of A to the result. When the sign of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A>B or the
complement of sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

**Subtraction algorithm:**
When the signs of A and B are different, add two magnitudes and attach the sign of A to the result. When the sign of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A>B or the complement of sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

**Hardware Implementation:**
Let A and B are two registers that hold the numbers.

A\,^S and B\,^S are 2, flip-flops that hold sign of corresponding numbers. The result is stored in A and A\,^S .and thus they form Accumulator register.

We need to perform micro operation, A+ B and hence a parallel adder.

A comparator is needed to establish if A> B, A=B, or A<B.

We need to perform micro operations A-B and B-A and hence two parallel subtractor.

An exclusive OR gate can be used to determine the sign relationship, that is, equal or not.

Thus the hardware components required are a magnitude comparator, an adder, and two subtractors.

**Reduction of hardware by using different procedure:**
1. We know subtraction can be done by complement and add.
2. The result of comparison can be determined from the end carry after the subtraction.

We find an adder and a complemeneter can do subtraction and comparison if 2’s complement is used for subtraction.

**Hardware for signed-magnitude addition and subtraction:**

**AVF** Add overflow flip flop. It hold the overflow bit when A & B are added.
Flip flop E—Output carry is transferred to E. It can be checked to see the relative magnitudes of the two numbers.

A-B = A +( -B )= Adding a and 2’s complement of B.

The A register provides other micro operations that may be needed when the sequence of steps in the algorithm is specified.

The complementer passes the contents of B or the complement of B to the Parallel Adder depending on the state of the mode control B. It consists of EX-OR gates and the parallel adder consists of full adder circuits. The M signal is also applied to the input carry of the adder.

When input carry M=0, the sum of full adder is A +B. When M=1, S = A + B’ +1= A – B

Hardware algorithm:

Flow Chart for Add and Subtract operations:

The EX-OR gate provides 0 as output when the signs are identical. It is 1 when the signs are different.

A + B is computed for the following and the sum is stored in EA:

1. When the signs are same and addition operation is required.
2. When the signs are different and subtract operation is required.

The carry in E after addition indicates an overflow if it is 1 and it is transferred to AVF, the add overflow flag

A-B = A+ B’+1 computed for the following:

1. When the signs are different and addition operation is required.
2. When the signs are same and subtract operation is required.

No overflow can occur if the numbers are subtracted and hence AVF is cleared to Zero.

[The subtraction of 2 n-digit un signed numbers M-N ( N≠0) in base r can be done as follows:

1. Add minuend M to the r’s complement of the subtrahend N. This performs M-N =r^n .
2. If M ≥ N, The sum will produce an end carry r which is discarded, and what is left is the result M-N.
3. If M< N, the sum does not produce an end carry and is equal to r-(N-M), which is the r’s complement of the sum and place a negative sign in front.]

A 1 in E indicates that A ≥ B and the number in A is the correct result.

If this number in A is zero, the sign A7 must be made positive to avoid a negative zero.

A 0 in E indicates that A< B. For this case it is necessary to take the 2’s complement of the value in A.

In the algorithm shown in flow chart, it is assumed that A register has circuits for micro operations complement and increment. Hence two complement of value in A is obtained in 2, micro operations. In other paths of the flow chart, the sign of the result is the same as the sign of A, so no change in A7 is required.
However, when \( A < B \), the sign of the result is the complement of original sign of \( A \). Hence, the complement of \( A \) stored in \( A \).

Final Result: \( A \)

**Flow chart for ADD and Subtract operations:**

**Addition and Subtraction with signed-2’s complement Data:**

**Arithmetic Addition:**

This method does not need a comparison or subtraction but only addition and complementation. The procedure is as below:

1. Represent the negative numbers in 2’s complement form.
2. Add the two numbers including the sign bits and discard any carry out of sign bit position.
3. The overflow bit \( V \) is set to 1 if there is a carry into sign bit and no carry out of sign bit or if there is a no carry into sign bit and a carry out of sign bit. Otherwise, it is set to zero.
4. If the result is negative, take the 2’s complement of the result to get a correct negative result.

**Arithmetic Subtraction:**

1. Represent the negative numbers in 2’s complement form.
2. Take the 2’s complement of the subtrahend including the sign bit and add it to the minuend including the sign bit.

3. The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or if there is a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.

4. Discard the carry out of the sign bit position.

Note: A subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed.

Fig: Hardware for Signed 2/s complement for addition/ subtraction.
1.1.2.7.

Lecture-7

**Multiplication Algorithm:**

Hardware implementation of multiplication of numbers in signed – magnitude form:

1. A adder is provided to add two binary numbers and the partial product is accumulated in a register.
2. Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which result in leaving the partial product and the multiplicand in the required relative positions.
3. When the corresponding bit of the multiplier is zero, there is no need to add all zeros to the partial product, since it will not alter its value.

The hardware consists of 4 flipflops, 3 registers, one sequence counter, an adder and complementer.

![Diagram of the multiplication hardware](image)

Q register & Qₐ flip flop : contains multiplier & its sign
Sequence counter : It is set to a value equal to the number of bits in the multiplier
B Register & Bₐ flipflop : It contains the multiplicand & its sign
A Register, E Flip flop : Initialized to ‘0’. Aₐ denotes sign of partial product
EA Register : hold partial product, with carry generated in addition being shifted to E.
Qₙ : Rightmost bit of the multiplier; AQ : will contain the final product.

As AQ represent product register, both Aₐ Qₙ represent the sign of the partial product or product.
The number to be multiplied are stores in memory as n bit sign magnitude numbers and when transferred to register msb bit go to sign flipflop and remaining n-1 bits go to registers. Hence SC is initially set to n-1.

Let the lower order bit of the multiplier in Qₙ tested.
If it is 1, the multiplicand in B is added to the present partial product in A.
If it is a ‘0’, nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and it’s new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0.
The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

**Flowchart for multiply operation:**

**Numerical Example for the above algorithm:**

<table>
<thead>
<tr>
<th>Multiplicand B= 10111</th>
<th>E</th>
<th>A</th>
<th>Q</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier in Q</td>
<td>0</td>
<td>00000</td>
<td>10011</td>
<td>101</td>
</tr>
<tr>
<td>(Q_n = 1); add B</td>
<td>0</td>
<td>10111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>First Partial Product</td>
<td>0</td>
<td>10111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right EAQ</td>
<td>0</td>
<td>01011</td>
<td>11001</td>
<td>100</td>
</tr>
<tr>
<td>(Q_n = 1); add B</td>
<td>1</td>
<td>10111</td>
<td>00010</td>
<td></td>
</tr>
</tbody>
</table>
1.1.2.8. **Lecture-8**

**Booth Multiplication Algorithm:**

Multiplication of signed- 2’s complement integers:

This algorithm uses the following facts.

1. A string of 0’s in the multiplier requires no addition but just shifting.
2. A string of 1’s in the multiplier from bit weight $2^k$ to weight $2^m$ can be treated as $2^{k+1} - 2^m$.

Example: Consider the binary number: 001110 ($+14$)

The number has a string of 1’s from $2^3$ to $2^1$. Hence $k = 3$ and $m = 1$. As other bits are 0’s, the number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$. Therefore the multiplication $M \times 14$, where $M$ is the multiplicand and 14 the multiplier can be done as $M \times 2^4 - M \times 2^1$.

This can be achieved by shifting binary multiplicand $M$ four times to the left and subtracting $M$ shifted left once which is equal to $(M \times 2^4 - M \times 2^1)$.

**Shifting and addition/subtraction rules for multiplicand in Booth’s Algorithm:**

1. The multiplicand is subtracted from the partial product upon encountering the first least significand 1 in a string of 1’s in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0’s in the multiplier.

3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

**Hardware Implementation of Booth Algorithm:**

![Diagram of Booth Algorithm]

Note: Sign bit is not separated from register. QR register contains the multiplier register and $Q_n, Q_{n+1}$ represent the least significant bit of the multiplier in QR. $Q_{n+1}$ is an extra flip flop appended to QR to facilitate a double bit inspection of the multiplier.

AC register and appended $Q_{n+1}$ are initially cleared to 0.

Sequence counter $Sc$ is set to the number $n$ which is equal to the number of bits in the multiplier.

$Q_n, Q_{n+1}$ are to successive bits in the multiplier.

**Example for multiplication using Booth algorithm:**

<table>
<thead>
<tr>
<th>$Q_n, Q_{n+1}$</th>
<th>BR = 1011, $BR' + 1 = 01001$</th>
<th>AC</th>
<th>QR</th>
<th>$Q_{n+1}$</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Initial Subtract BR</td>
<td>00000 01001 01001 00100</td>
<td>10011</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>ashr</td>
<td></td>
<td>11001</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>ashr</td>
<td>00010</td>
<td>01100</td>
<td>1</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>Add BR</td>
<td>10111 11001 11100</td>
<td>10110</td>
<td>0</td>
<td>010</td>
</tr>
<tr>
<td></td>
<td>ashr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>ashr</td>
<td>11110</td>
<td>01011</td>
<td>0</td>
<td>001</td>
</tr>
<tr>
<td>10</td>
<td>Subtract BR</td>
<td>01001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm in flowchart for multiplication of signed 2’s complement numbers.

Array Multiplier:
2-bit by 2-bit Array Multiplier:

Multiplicand bits are \( b_1 \) and \( b_0 \). Multiplier bits are \( a_1 \) and \( a_0 \). The first partial product is obtained by multiplying \( a_0 \) by \( b_1 b_0 \). The bit multiplication is implemented by AND gate. First partial product is made by two AND gates. Second partial product is made by two AND gates. The two partial products are added with two half adder circuits.
Combinational circuit binary multiplier:

A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there bits in the multiplier. The binary output in each level of the AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier and k multiplicand bits, we need j*k AND Gates and (j-1)*k bit adders to produce a product of j+k bits.

4-bit by 3-bit Array Multiplier:
Division Algorithms:

Division Process for division of fixed point binary number in signed–magnitude representation:

Let dividend A consists of 10 bits and divisor B consists of 5 bits.

1. Compare the 5 most significant bits of the dividend with that of divisor.
2. If the 5 bit number is smaller than divisor B, then take 6 bits of the dividend and compare with the 5 bit divisor.
3. The 6 bit number is greater than divisor B. Hence place a 1 for the quotient bit in the sixth position above the dividend. Shift the divisor once to the right and subtracted from the dividend. The difference is called partial remainder.
4. Repeat the process with the partial remainder and divisor. If the partial remainder is equal or greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is small than the divisor, then the quotient bit is zero and no subtraction is needed. The divisor is shifted once to the right in any case.

Hardware Implementation of division for signed magnitude fixed point numbers:

To implement division using a digital computer, the process is changed slightly for convenience.

1. Instead of shifting the divisor to the right, the dividend or the partial remainder, is shifted to the left so as to leave the two numbers in the required relative position.
2. Subtraction may be achieved by adding A (dividend) to the 2’s complement of B(divisor). The information about the relative magnitude is then available from end carry.
3. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost.
4. The divisor is stored in B register and the double length dividend is stored in registers A and Q.
5. The dividend is shifted to the left and the divisor is subtracted by adding its 2’s complement value.
6. If E = 1, it signifies that A ≥ B. A quotient bit is inserted into Qn and the partial remainder is shifted to the left to repeat the process.
7. If E = 0, it signifies that A < B so the quotient Qn remains 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all 5 quotient bits are formed.
8. At the end Q contains the quotient and A the remainder. If the sign of dividend and divisor are alike, the quotient is positive and if unalike, it is negative. The sign of the remainder is the same as dividend.
**Hardware for implementing division of fixed point signed-Magnitude Numbers**

**Example of Binary division with digital hardware:** Divisor B = 10001, \(\overline{B} + 1 = 01111\)

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>A</th>
<th>Q</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dividend:</td>
<td>01110</td>
<td>00000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Shl EAQ</td>
<td>11100</td>
<td>00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add , (\overline{B} + 1)</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E = 1</td>
<td>1</td>
<td>01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Q_n = 1</td>
<td>1</td>
<td>01011</td>
<td>00001</td>
<td>4</td>
</tr>
<tr>
<td>Shl EAQ</td>
<td>0</td>
<td>10110</td>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>Add , (B + 1)</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E = 1</td>
<td>1</td>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Q_n = 1</td>
<td>1</td>
<td>00101</td>
<td>00111</td>
<td>3</td>
</tr>
<tr>
<td>Shl EAQ</td>
<td>0</td>
<td>01010</td>
<td>00110</td>
<td></td>
</tr>
<tr>
<td>Add , (B + 1)</td>
<td>01111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E= 0; Leave Q_n = 0</td>
<td>0</td>
<td>11001</td>
<td>00110</td>
<td></td>
</tr>
<tr>
<td>Add B</td>
<td>10001</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Divide overflow:

When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows:

A divide-overflow condition occurs if the higher order half bits of the dividend constitute a number greater than or equal to the divisor. If the divisor is zero, then the dividend will definitely be greater than or equal to divisor. Hence divide overflow condition occurs and hence the divide-overflow –flip flop will be set. Let the flip flop be called DVF.

Handling DVF:

1. Check if DVF is set after each divide instruction. If DVF is set, then the program branches to a subroutine that takes corrective measures such as rescaling the data to avoid overflow.
2. An interrupt is generated if DVF is set. The interrupt causes the processor to suspend the current program and branch to interrupt service routine to take corrective measure. The most
common corrective measure is to remove the program and type an error message that explains the reasons.

3. The divide overflow can be handled very simply if the numbers are represented in floating point representation.

**Flow chart for divide operation:**

Assumption:

Operands are transferred from memory to registers as n bit words. n-1 bit form magnitude and 1 bit shows the sign.
A divide overflow condition is tested by subtracting the divisor in B from half of the bits of dividend stored in A. If \( v_A \geq B \), the DVF is set and the operation is terminated prematurely. If \( A < B \), no DVF occurs and so the value of dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in \( AQ \) to the left, with the higher order bit shifted into \( E \). If the bit shifted into \( E \) is 1, we know that \( EA \) is greater than \( B \) because \( EA \) consists of a 1 followed by \( n-1 \) bits while \( B \) consists of only \( n-1 \) bits. In this case, \( B \) must be subtracted from \( EA \) and 1 inserted into \( Q_n \) for the quotient bit. Since register A is missing the higher order bit of the dividend (which is in \( E \)), its value is \( EA - 2^{n-1} \). Adding to this value the 2’s complement of \( B \) results in

\[
(\text{EA} - 2^{n-1}) + (2^{n-1} - B) = E - B.
\]

The carry from the addition is not transferred to \( E \) if we want \( E \) to remain a 1.

If the shift left operation inserts a zero into \( E \), the divisor is subtracted by adding it’s 2’s complement value and the carry is transferred into \( E \). If \( E = 1 \), it signifies that \( A \geq B \) and hence \( Q_n \) is set to 1. If \( E = 0 \), it signifies that \( A < B \) and the original number is restored by adding \( B \) to \( A \). In the latter case we leave a 0 in \( Q_n \) (0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After \( n-1 \) times, the quotient magnitude is formed in the register \( Q \) and the remainder is found in register A.

1.1.2.10. Lecture-10
Floating-Point Arithmetic Operations

Many high-level programming languages have a facility for specifying floating-point numbers. The most common way is to specify them by a real declaration statement as opposed to fixed-point numbers, which are specified by an integer declaration statement. Any computer that has a compiler for such high-level programming language must have a provision for handling floating-point arithmetic operations. The operations are quite often included in the internal hardware. If no hardware is available for the operations, the compiler must be designed with a package of floating-point software subroutines. Although the hardware method is more expensive, it is so much more efficient than the software method that floating-point hardware is included in most computers and is omitted only in very small ones.

Basic Considerations

Floating-point representation of data was introduced in Sec. 3-4. A floating-point number in computer registers consists of two parts: a mantissa \( m \) and an exponent \( e \). The two parts represent a number obtained from multiplying \( m \) times a radix \( r \) raised to the value of \( e \); thus

\[ m \times r^e \]

The mantissa may be a fraction or an integer. The location of the radix point and the value of the radix \( r \) are assumed and are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with \( m = 53725 \) and \( e = 3 \) and is interpreted to represent the floating-point number

\[ .53725 \times 10^3 \]
A floating-point number is normalized if the most significant digit of the mantissa is nonzero. In this way the mantissa contains the maximum possible number of significant digits. A zero cannot be normalized because it does not have a nonzero digit. It is represented in floating-point by all 0’s in the mantissa and exponent.

Floating-point representation increases the range of numbers that can be accommodated in a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be \( \pm (2^{47} - 1) \), which is approximately \( \pm 10^{14} \). The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be accommodated is

\[
\pm (1 - 2^{-35}) \times 2^{2047}
\]

This number is derived from a fraction that contains 35 1’s, an exponent of 11 bits (excluding its sign), and the fact that \( 2^{11} - 1 = 2047 \). The largest number that can be accommodated is approximately \( 10^{615} \), which is a very large number. The mantissa can accommodate 35 bits (excluding the sign) and if considered as an integer it can store a number as large as \( (2^{35} - 1) \). This is approximately equal to \( 10^{10} \), which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer may use four words to represent one floating-point number. One word of 8 bits is reserved for the exponent and the 24 bits of the other three words are used for the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers and their execution takes longer and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent. Consider the sum of the following floating-point numbers:

\[
.5372400 \times 10^2 \\
+ .1580000 \times 10^{-1}
\]

It is necessary that the two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the mantissas are stored in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has
the smaller exponent to the right by a number of places equal to the difference between the exponents. After this is done, the mantissas can be added:

\[
\begin{align*}
.5372400 \times 10^2 \\
+.0001580 \times 10^2 \\
\hline
.5373980 \times 10^2
\end{align*}
\]

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

\[
\begin{align*}
.56780 \times 10^5 \\
-.56430 \times 10^5 \\
\hline
.00350 \times 10^5
\end{align*}
\]

A floating-point number that has a 0 in the most significant position of the mantissa is said to have an underflow. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. In the example above, it is necessary to shift left twice to obtain \( .35000 \times 10^3 \). In most computers, a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents. Division is accomplished by dividing the mantissas and subtracting the exponents.

The operations performed with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represented in any one of the three representations: signed-magnitude, signed-2's complement, or signed-1's complement.

A fourth representation employed in many computers is known as a biased exponent. In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation. Consider an exponent that ranges from \(-50\) to \(49\). Internally, it is represented by two digits (without a sign) by adding to it a bias of \(50\). The exponent register contains the number \(e + 50\), where \(e\) is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of \(00\)
to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction of 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range from 49 to 00. The subtraction of 50 gives the negative values in the range of −1 to −50.

The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. As a consequence, a magnitude comparator can be used to compare their relative magnitude during the alignment of the mantissa. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

In the examples above, we used decimal numbers to demonstrate some of the concepts that must be understood when dealing with floating-point numbers. Obviously, the same concepts apply to binary numbers as well. The algorithms developed in this section are for binary numbers. Decimal computer arithmetic is discussed in the next section.

Register Configuration
The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. As a general rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in Fig. 10-14. There are three registers, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.

It is assumed that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa

![Figure 10-14 Registers for floating-point arithmetic operations.](image-url)
whose sign is in A, and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A₁. The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of \( A_s, A, \) and \( a \).

Similarly, register BR is subdivided into \( B_s, B, \) and \( b, \) and QR into \( Q_s, Q, \) and \( q. \) A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. It is assumed that the floating-point numbers are so large that the chance of an exponent overflow is very remote, and for this reason the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so the binary point is assumed to reside to the left of the magnitude part. Integer representation for floating-point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.

The numbers in the registers are assumed to be initially normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands coming from and going to the memory unit are always normalized.

**Addition and Subtraction**

During addition or subtraction, the two floating-point operands are in \( AC \) and \( BR \). The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

A floating-point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be unnormalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory.

The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 10-15. If \( BR \) is equal to zero, the operation is terminated, with the value in the \( AC \) being the result. If \( AC \) is equal to zero, we transfer
Figure 10-15  Addition and subtraction of floating-point numbers.
the content of BR into AC and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas.

The magnitude comparator attached to exponents $a$ and $b$ provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm presented in Fig. 10-2. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop $E$. If $E$ is equal to 1, the bit is transferred into $A_1$ and all other bits of $A$ are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the $AC$ is made zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position $A_1$ is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in $A_1$ is checked again and the process is repeated until it is equal to 1. When $A_1 = 1$, the mantissa is normalized and the operation is completed.

**Multiplication**

The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissas is necessary. The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product. The double-precision answer is used in fixed-point numbers to increase the accuracy of the product. In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product.

The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.
Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents.

The flowchart for floating-point multiplication is shown in Fig. 10-16. The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is

Figure 10-16 Multiplication of floating-point numbers.
terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

The exponent of the multiplier is in \( q \) and the adder is between exponents \( a \) and \( b \). It is necessary to transfer the exponents from \( q \) to \( a \), add the two exponents, and transfer the sum into \( a \). Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

The multiplication of the mantissas is done as in the fixed-point case with the product residing in \( A \) and \( Q \). Overflow cannot occur during multiplication, so there is no need to check for it.

The product may have an underflow, so the most significant bit in \( A \) is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in \( AQ \) is shifted left and the exponent decremented. Note that only one normalization shift is necessary. The multiplier and multiplicant were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01. Therefore, only one leading zero may occur.

Although the low-order half of the mantissa is in \( Q \), we do not use it for the floating-point product. Only the value in the \( AC \) is taken as the product.

**Division**

Floating-point division requires that the exponents be subtracted and the mantissas divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the \( AC \). Remember that the mantissa dividend is a fraction and not an integer. For integer representation, a single-precision dividend must be placed in register \( Q \) and register \( A \) must be cleared. The zeros in \( A \) are to the left of the binary point and have no significance. In fraction representation, a single-precision dividend is placed in register \( A \) and register \( Q \) is cleared. The zeros in \( Q \) are to the right of the binary point and have no significance.

The check for divide-overflow is the same as in fixed-point representation. However, with floating-point numbers the divide-overflow imposes no problems. If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1. For normalized operands this is a sufficient operation to ensure that no mantissa divide-overflow will occur. The operation above is referred to as a **dividend alignment**.

The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require a normalization.

The division algorithm can be subdivided into five parts:

1. Check for zeros.
2. Initialize registers and evaluate the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

The flowchart for floating-point division is shown in Fig. 10-17. The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message. An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.

If the operands are not zero, we proceed to determine the sign of the quotient and store it in Qs. The sign of the dividend in A1 is left unchanged to be the sign of the remainder. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If \( A \geq B \), it is necessary to shift \( A \) once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that \( A < B \).

Next, the divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into \( q \) because the quotient is formed in QR.

The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies \( (n - 1) \) positions to the left of \( A1 \). The remainder can be converted to a normalized fraction by subtracting \( n - 1 \) from the dividend exponent and by shift and decrement until the bit in \( A1 \) is equal to 1. This is not shown in the flow chart and is left as an exercise.
10-6 Decimal Arithmetic Unit

The user of a computer prepares data with decimal numbers and receives results in decimal form. A CPU with an arithmetic logic unit can perform arithmetic microoperations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and
Figure 10-17  Division of floating-point numbers.
output data. When the application calls for a large amount of input-output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers capable of performing decimal arithmetic must store the decimal data in binary-coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic microoperations.

Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, since this process requires special circuits and also takes a longer time to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data. Users can specify by programmed instructions whether they want the computer to perform calculations with binary or decimal data.

A decimal arithmetic unit is a digital function that performs decimal microoperations. It can add or subtract decimal numbers, usually by forming the 9’s or 10’s complement of the subtrahend. The unit accepts coded decimal numbers and generates results in the same adopted binary code. A single-stage decimal arithmetic unit consists of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the augend digit, four inputs for the addend digit, and an input-carry. The outputs include four terminals for the sum digit and one for the output-carry. Of course, there is a wide variety of possible circuit configurations dependent on the code used to represent the decimal digits.

**BCD Adder**

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols $K$, $Z_8$, $Z_4$, $Z_2$, and $Z_1$. $K$ is the carry and the subscripts under the letter $Z$ represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical
<table>
<thead>
<tr>
<th>$K$</th>
<th>$Z_8$</th>
<th>$Z_4$</th>
<th>$Z_2$</th>
<th>$Z_1$</th>
<th>$C$</th>
<th>$S_8$</th>
<th>$S_4$</th>
<th>$S_2$</th>
<th>$S_1$</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>19</td>
</tr>
</tbody>
</table>

and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required.

One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position $Z_8$. To distinguish them from binary 1000 and 1001 which also have a 1 in position $Z_8$, we specify further that either $Z_4$
or $Z_2$ must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage.

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 10-18. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

**Figure 10-18** Block diagram of BCD adder.
A decimal parallel-adder that adds $n$ decimal digits needs $n$ BCD adder stages with the output-carry from one stage connected to the input-carry of the next-higher-order stage. To achieve shorter propagation delays, BCD adders include the necessary circuits for carry look-ahead. Furthermore, the adder circuit for the correction does not need all four full-adders, and this circuit can be optimized.

**BCD Subtraction**

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, the 9's complement cannot be obtained by complementing each bit in the code. It must be formed by a circuit that subtracts each BCD digit from 9.

The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit provided a correction is included. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition. In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111 (decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of a 4-bit binary number $N$ is identical to the subtraction of the number from 1111 (decimal 15). Adding the binary equivalent of decimal 10 gives $15 - N + 10 = 9 - N + 16$. But 16 signifies the carry that is discarded, so the result is $9 - N$ as required. Adding the binary equivalent of decimal 6 and then complementing gives $15 - (N + 6) = 9 - N$ as required.

The 9's complement of a BCD digit can also be obtained through a combinational circuit. When this circuit is attached to a BCD adder, the result is a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables $B_8, B_4, B_2,$ and $B_1$. Let $M$ be a mode bit that controls the add/subtract operation. When $M = 0$, the two digits are added; when $M = 1$, the digits are subtracted. Let the binary variables $x_8, x_4, x_2,$ and $x_1$ be the outputs of the 9's complements circuit. By an examination of the truth table for the circuit, it may be observed (see Prob. 10-30) that $B_1$ should always be complemented; $B_2$ is always the same in the 9's complement as in the original digit; $x_4$ is 1 when the exclusive-OR of $B_2$ and $B_4$ is 1; and $x_8$ is 1 when $B_8B_4B_2 = 000$. The Boolean functions for the 9's complements circuit are

\[
x_1 = B_1' + B_1 \cdot M
\]

\[
x_2 = B_2
\]
\[ x_4 = B_4 M' + (B'_4 B_2 + B_4 B'_2) M \]
\[ x_8 = B_8 M' + B'_8 B'_4 B'_2 M \]

From these equations we see that \( x = B \) when \( M = 0 \). When \( M = 1 \), the \( x \) outputs produce the 9's complement of \( B \).

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in Fig. 10-19. It consists of a BCD adder and a 9's complementer. The mode \( M \) controls the operation of the unit. With \( M = 0 \), the \( S \) outputs form the sum of \( A \) and \( B \). With \( M = 1 \), the \( S \) outputs form the sum of \( A \) plus the 9's complement of \( B \). For numbers with \( n \) decimal digits we need \( n \) such stages. The output carry \( C_{i+1} \) from one stage must be connected to the input carry \( C_i \) of the next-higher-order stage. The best way to subtract the two decimal numbers is to let \( M = 1 \) and apply a 1 to the input carry \( C_i \) of the first stage. The outputs will form the sum of \( A \) plus the 10's complement of \( B \), which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.

1.1.2.12 Lecture-12
Flowcharts can be used for both types of data provided that we interpret the microoperation symbols properly. Decimal numbers in BCD are stored in computer registers in groups of four bits. Each 4-bit group represents a decimal digit and must be taken as a unit when performing decimal microoperations.

For convenience, we will use the same symbols for binary and decimal arithmetic microoperations but give them a different interpretation. As shown in Table 10-5, a bar over the register letter symbol denotes the 9's complement of the decimal number stored in the register. Adding 1 to the 9's complement produces the 10's complement. Thus, for decimal numbers, the symbol \( A \leftarrow A + \bar{B} + 1 \) denotes a transfer of the decimal sum formed by adding the original content \( A \) to the 10's complement of \( B \). The use of identical symbols for the 9's complement and the 1's complement may be confusing if both types of data are employed in the same system. If this is the case, it may be better to adopt a different symbol for the 9's complement. If only one type of data is being considered, the symbol would apply to the type of data used.

Incrementing or decrementing a register is the same for binary and decimal except for the number of states that the register is allowed to have. A binary counter goes through 16 states, from 0000 to 1111, when incremented. A decimal counter goes through 10 states from 0000 to 1001 and back to 0000, since 9 is the last count. Similarly, a binary counter sequences from 1111 to 0000 when decremented. A decimal counter goes from 1001 to 0000.

A decimal shift right or left is preceded by the letter \( d \) to indicate a shift over the four bits that hold the decimal digits. As a numerical illustration consider a register \( A \) holding decimal 7860 in BCD. The bit pattern of the 12 flip-flops is

\[
0111 \ 1000 \ 0110 \ 0000
\]

The microoperation \( d \text{shr} A \) shifts the decimal number one digit to the right to give 0786. This shift is over the four bits and changes the content of the register into

\[
0000 \ 0111 \ 1000 \ 0110
\]

<table>
<thead>
<tr>
<th>Symbolic Designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \leftarrow A + B )</td>
<td>Add decimal numbers and transfer sum into ( A )</td>
</tr>
<tr>
<td>( \bar{B} )</td>
<td>9's complement of ( B )</td>
</tr>
<tr>
<td>( A \leftarrow A + \bar{B} + 1 )</td>
<td>Content of ( A ) plus 10's complement of ( B ) into ( A )</td>
</tr>
<tr>
<td>( Q_i \leftarrow Q_i + 1 )</td>
<td>Increment BCD number in ( Q_i )</td>
</tr>
<tr>
<td>( d \text{shr} A )</td>
<td>Decimal shift-right register ( A )</td>
</tr>
<tr>
<td>( d \text{shl} A )</td>
<td>Decimal shift-left register ( A )</td>
</tr>
</tbody>
</table>
Addition and Subtraction

The algorithm for addition and subtraction of binary signed-magnitude numbers applies also to decimal signed-magnitude numbers provided that we interpret the microoperation symbols in the proper manner. Similarly, the algorithm for binary signed-2’s complement numbers applies to decimal signed-10’s complement numbers. The binary data must employ a binary adder and a complements. The decimal data must employ a decimal arithmetic unit capable of adding two BCD numbers and forming the 9’s complement of the subtrahend as shown in Fig. 10-19.

Decimal data can be added in three different ways, as shown in Fig. 10-20. The parallel method uses a decimal arithmetic unit composed of as many BCD adders as there are digits in the number. The sum is formed in parallel and requires only one microoperation. In the digit-serial bit-parallel method, the digits are applied to a single BCD adder serially, while the bits of each coded digit are transferred in parallel. The sum is formed by shifting the decimal numbers through the BCD adder one at a time. For k decimal digits, this configuration requires k microoperations, one for each decimal shift. In the all serial adder, the bits are shifted one at a time through a full-adder. The binary sum formed after four shifts must be corrected into a valid BCD digit. This correction, discussed in Sec. 10-6, consists of checking the binary sum. If it is greater than or equal to 1010, the binary sum is corrected by adding to it 0110 and generating a carry for the next pair of digits.

The parallel method is fast but requires a large number of adders. The digit-serial bit-parallel method requires only one BCD adder, which is shared by all the digits. It is slower than the parallel method because of the time required to shift the digits. The all serial method requires a minimum amount of equipment but is very slow.
Multiplication

The multiplication of fixed-point decimal numbers is similar to binary except for the way the partial products are formed. A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 and 1 digits. In the binary case, the multiplicand is added to the partial product if the multiplier bit is 1. In the decimal case, the multiplicand must be multiplied by the digit multiplier and the result added to the partial product. This operation can be accomplished by adding the multiplicand to the partial product a number of times equal to the value of the multiplier digit.

The registers organization for the decimal multiplication is shown in Fig. 10-21. We are assuming here four-digit numbers, with each digit occupying four bits, for a total of 16 bits for each number. There are three registers, \( A, B, \) and \( Q \), each having a corresponding sign flip-flop \( A_s, B_s, \) and \( Q_s \).
Registers $A$ and $B$ have four more bits designated by $A_e$ and $B_e$ that provide an extension of one more digit to the registers. The BCD arithmetic unit adds the five digits in parallel and places the sum in the five-digit $A$ register. The end-carry goes to flip-flop $E$. The purpose of digit $A_e$ is to accommodate an overflow while adding the multiplicand to the partial product during multiplication. The purpose of digit $B_e$ is to form the 9's complement of the divisor when subtracted from the partial remainder during the division operation. The least significant digit in register $Q$ is denoted by $Q_L$. This digit can be incremented or decremented.

A decimal operand coming from memory consists of 17 bits. One bit (the sign) is transferred to $B$, and the magnitude of the operand is placed in the lower 16 bits of $B$. Both $B_e$ and $A_e$ are cleared initially. The result of the operation is also 17 bits long and does not use the $A_e$ part of the $A$ register.

The decimal multiplication algorithm is shown in Fig. 10-22. Initially, the entire $A$ register and $B_e$ are cleared and the sequence counter $SC$ is set to a number $k$ equal to the number of digits in the multiplier. The low-order digit of the multiplier in $Q_l$ is checked. If it is not equal to 0, the multiplicand in $B$ is added to the partial product in $A$ once and $Q_l$ is decremented. $Q_l$ is checked again and the process is repeated until it is equal to 0. In this way, the multiplicand in $B$ is added to the partial product a number of times equal to the multiplier digit. Any temporary overflow digit will reside in $A_e$ and can range in value from 0 to 9.

Next, the partial product and the multiplier are shifted once to the right. This places zero in $A_e$ and transfers the next multiplier quotient into $Q_l$. The process is then repeated $k$ times to form a double-length product in $AQ$. 
Division

Decimal division is similar to binary division except of course that the quotient digits may have any of the 10 values from 0 to 9. In the restoring division method, the divisor is subtracted from the dividend or partial remainder as many times as necessary until a negative remainder results. The correct remainder is then restored by adding the divisor. The digit in the quotient reflects the number of subtractions up to but excluding the one that caused the negative difference.

The decimal division algorithm is shown in Fig. 10-23. It is similar to the algorithm with binary data except for the way the quotient bits are formed. The dividend (or partial remainder) is shifted to the left, with its most significant digit placed in $A_r$. The divisor is then subtracted by adding its 10’s complement value. Since $B_r$ is initially cleared, its complement value is 9 as required. The carry in $E$ determines the relative magnitude of $A$ and $B$. If $E = 0$, it signifies
that $A < B$. In this case the divisor is added to restore the partial remainder and $Q_L$ stays at 0 (inserted there during the shift). If $E = 1$, it signifies that $A \geq B$. The quotient digit in $Q_L$ is incremented once and the divisor subtracted again. This process is repeated until the subtraction results in a negative difference which is recognized by $E$ being 0. When this occurs, the quotient digit is not incremented but the divisor is added to restore the positive remainder. In this way, the quotient digit is made equal to the number of times that the partial remainder "goes" into the divisor.
The partial remainder and the quotient bits are shifted once to the left and
the process is repeated k times to form k quotient digits. The remainder is then
found in register A and the quotient is in register Q. The value of E is neglected.

Floating-Point Operations
Decimal floating-point arithmetic operations follow the same procedures as
binary operations. The algorithms in Sec. 10-5 can be adopted for decimal data
provided that the microoperation symbols are interpreted correctly. The multi-
plementation and division of the mantissas must be done by the methods de-
scribed above.

1.1.2.14 Test Questions

4.1.5.a.docx

4.1.5.b.docx  true or false.docx

1.1.2.15 Fill in the blanks type of questions <Minimum of ten>

a. The decimal representation for hex number F3 is ________________.

b. The binary equivalent for the decimal number 41.6875 is ____________________

c. The BCD code for the decimal number 248 is ____________________.

d. For a given number N in base r having n digits, the (r-1)'s complement of N is defined as ________________

e. The 10's complement of a decimal number is obtained by adding _____ to the 9's complement value.
f. When 2 unsigned numbers are added, an overflow is detected from the \textbf{most significant position}. 

g. An overflow for addition/subtraction of two signed numbers is detected when the carry into the sign bit position and carry out of the sign bit position are \textbf{not equal}. 

h. Booth multiplication algorithm is followed when the binary integers are represented in \textbf{signed 2’s complement representation for negative integers}. 

i. When Booth algorithm is used for multiplication, the partial product does not change when the multiplier is identical to the previous multiplier. 

j. Floating point multiplication and division do not require an alignment of the \textbf{mantissa}. 

Answers: (1) 243 (2) 101001.1011 (3) 0010 0100 1000 (4) \( r^n-1-N \) (5) 1 (6) carry out (7) not equal (8) signed 2’s complement representation for negative integers (9) bit, bit (10) mantissa

\textbf{1.1.2.16. Multiple choice questions (Minimum of ten)}

1. Floating point representation is used to store
(A) Boolean values (B) whole numbers (C) real integers (D) integers 
\textbf{Ans: C}

2. In computers, subtraction is generally carried out by
(A) 9’s complement (B) 10’s complement (C) 1’s complement (D) 2’s complement 
\textbf{Ans: D}

3. The circuit used to store one bit of data is known as
(A) Register (B) Encoder (C) Decoder (D) Flip Flop 
\textbf{Ans: D}

4. Which of the following is not a weighted code?
(A) Decimal Number system (B) Excess 3-cod (C) Binary number System (D) None of these 
\textbf{Ans: B}

5. Assembly language
(A) uses alphabetic codes in place of binary numbers used in machine language (B) is the easiest language to write programs (C) need not be translated into machine language
6. The multiplicand register & multiplier register of a hardware circuit implementing booth’s algorithm have (11101) & (1100). The result shall be (A) (812) 10 (B) (-12) 10 (C) (12) 10 (D) (-812) 10

Ans: A

7. What characteristic of RAM memory makes it not suitable for permanent storage? (A) too slow (B) unreliable (C) it is volatile (D) too bulky

Ans: C

8. (2FAOC) 16 is equivalent to (A) (195 084) 10 (B) (001011111010 0000 1100) 2 (C) Both (A) and (B) (D) None of these

Ans: B

9. The average time required to reach a storage location in memory and obtain its contents is called the (A) seek time (B) turnaround time (C) access time (D) transfer time

Ans: C

10. In signed-magnitude binary division, if the dividend is (11100) 2 and divisor is (10011) 2 then the result is (A) (00100) 2 (B) (10100) 2 (C) (11001) 2 (D) (01100) 2

Ans: B

True or False questions

1. EEPROM comes under volatile memory category. ________________

2. Thumb drive or pen drive is semiconductor memory. ________________

3. The control unit generates the appropriate signal at the right moment. ________________

4. While executing a program, CPU brings instruction and data from disk memory. ________________

5. A memory module of capacity 16 * 4, indicates a storage of 128 bits. ________________

6. A memory module of capacity of 1024 locations, the required address bus size is 10. ________________

7. The program counter PC is used to store the address of the next instruction to be fetched from Accumulator. ________________
8. For n-bit signed integer, the range of numbers that can be represented is $-2^{n-1}$ to $2^{n+1}$. 

9. Given a number $N$ in base $r$ having $n$ digits, the $(r-1)'s$ complement of $N$ is defined as

$$ (r^n - 1) - r. $$

10. Floating point representation uses mantissa and an exponent part of radix $R$. 

Answers: (1) false (2) true (3) true (4) false (5) false (6) true (7) false (8) false (9) false (10) true

1.1.2.18. **Review Questions**

4.1.6.a.docx 4.1.6.b.docx 4.1.6.d.docx 4.1.6.e.docx 4.1.6.c.docx

a. Objective type of questions (Very short notes) <Minimum of ten>
b. Analytical type questions <Minimum of ten>
c. Essay type Questions <As per requirements>
d. Problems <As per required Number>
e. Case study <As per required Number>

1.1.2.19. **Skill Building Exercises/Assignments**

a. Take the mother board of a computer and identify CPU, memory, peripheral ICs, BUS etc.
b. Buy the components of a computer, assemble, install the software and make it to function.

Eg:- Prepare a model of something
- Trace something
- Prepare a report on something etc.,

1.1.2.20. **Previous Questions (Asked by JNTUK from the concerned Unit)**

JNTUK questions unit-1.docx

1.1.2.21. **GATE Questions (Where relevant)**
Subject is not in gate syllabus

1.1.2.22. Interview questions (which are frequently asked in a Technical round-Placements)

4.1.10.docx

1.1.2.23. Real-Word (Live) Examples/Case studies wherever applicable

a. List out the intel CPUs in various generation with their specifications. Write how the performance was improved in each generation.

1.1.2.24. Suggested “Expert Guest Lectures” (both from in and outside of the campus)

1.1.2.25. Literature references of Relevant NPTEL Videos/Web/You Tube videos etc.

neptel video ref.docx

1.1.2.26. Any Lab requirements; if so link it to Lab Lesson Plan.

No.

1.1.2.27. Reference Text Books / with Journals Chapters etc.

