LABORATORY MANUAL

UNIT 16
Introduction to Computers 5

UNIT 17
Introduction to Programming 27

UNIT 18
List of Practical Sessions 39
Curriculum Design Committee

Dr. B.D. Acharya  
Dept. of Science & Technology  
New Delhi

Prof. Adimurthi  
School of Mathematics  
TIFR, Bangalore

Prof. Archanaa Aggarwal  
CESP, School of Social Sciences  
JNU, New Delhi

Prof. R. B. Bapat  
Indian Statistical Institute, New Delhi

Prof. M.C. Bhandari  
Dept. of Mathematics  
IIT, Kanpur

Prof. R. Bhatia  
Indian Statistical Institute, New Delhi

Prof. A. D. Dharmadhikari  
Dept. of Statistics  
University of Pune

Prof. O.P. Gupta  
Dept. of Financial Studies  
University of Delhi

Prof. S.D. Joshi  
Dept. of Electrical Engineering  
IIT, Delhi

Dr. R. K. Khanna  
Scientific Analysis Group  
DRDO, Delhi

Prof. Susheela Kumar  
Dept. of Management Studies  
IIT, Delhi

Prof. Veni Madhavan  
Scientific Analysis Group  
DRDO, Delhi

Prof. J.C. Mishra  
Dept. of Mathematics  
IIT, Kharagpur

Prof. C. Musili  
Dept. of Mathematics and Statistics  
University of Hyderabad

Prof. Sankar Pal  
ISI, Kolkata

Prof. A.P. Singh  
PG Dept. of Mathematics  
University of Jammu

Faculty Members  
School of Sciences, IGNOU

Dr. Deepika  
Prof. Poornima Mital  
Dr. Atul Razdan  
Prof. Parvin Sinclair  
Prof. Sujatha Varma  
Dr. S. Venkataraman

Course Design Committee

Prof. C.A. Murthy  
ISI, Kolkata

Prof. S.B. Pal  
IIT, Kharagpur

Dr. B.S. Panda  
IIT, Delhi

Prof. C.E. Veni Madhavan  
IISC, Bangalore

Prof. O.P. Gupta  
Dept. of Financial Studies  
University of Delhi

Prof. S.D. Joshi  
Dept. of Electrical Engineering  
IIT, Delhi

Dr. R. K. Khanna  
Scientific Analysis Group  
DRDO, Delhi

Prof. Susheela Kumar  
Dept. of Management Studies  
IIT, Delhi

Prof. Veni Madhavan  
Scientific Analysis Group  
DRDO, Delhi

Prof. J.C. Mishra  
Dept. of Mathematics  
IIT, Kharagpur

Prof. C. Musili  
Dept. of Mathematics and Statistics  
University of Hyderabad

Prof. Sankar Pal  
ISI, Kolkata

Prof. A.P. Singh  
PG Dept. of Mathematics  
University of Jammu

Faculty Members  
School of Sciences, IGNOU

Dr. Deepika  
Prof. Poornima Mital  
Dr. Atul Razdan  
Prof. Parvin Sinclair  
Prof. Sujatha Varma  
Dr. S. Venkataraman

Block Preparation Team

Dr. S. Venkataraman  
School of Sciences  
IGNOU

Unit 16 of this block is a modified version of Unit 13, Block 4 of PHE-06.
February 2008
©Indira Gandhi National Open University, 2008
ISBN-

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means without written permission from the Indira Gandhi National Open University.

Further information on the Indira Gandhi National Open University courses may be obtained from the University’s office at Maidan Garhi, New Delhi-110 068.

Printed and Published on behalf of Indira Gandhi National Open University, New Delhi, by Director, School of Sciences.
As we have mentioned in the Block introduction of Block 1 of the course, programming cannot be learnt by just reading books. It can be learnt only by constant practice, by creating, compiling and executing programs. Because of this, we have a substantial practical component for this course. You will recall that in the 70% marks earmarked for the term end examination, 50% is for practicals and 20% is for the theory. In the 30% marks for continuous assessment, 20% is for the assignment and 10% is for the practical component. This manual will serve as a guide for the practical component.

Unit 1 of this Block is an introduction to computers to those who do not have a general awareness about working of computers. Even if you are knowledgeable, browse through the Unit, skipping the portions you are familiar with.

In the second Unit we start with a brief introduction to program design. It explains the concept of an algorithm. It also shows the flow charts of some algorithms. We then explain how to compile and run C programs using the GCC compiler and how to use two of the IDEs (Integrated Development Environments) to compile C programs. You are free to use any ANSI compliant compiler you like. Here are some of the choices available with a brief description:

1. The mingwin collection of tools and compilers for the windows. This includes the GCC compiler. The Dev-C++ IDE comes bundled with the GCC compiler from the mingwin collection. The Dev-C++ IDE with the compiler can be downloaded from [www.bloodshed.net](http://www.bloodshed.net).
2. The cygwin collection. This also includes the GCC and you can compile and run most of the linux based programs on this platform.
3. The Micosoft Visual C++ express edition is available for free download.
4. The Watcom C/C++ compiler is also available for download.
5. The DJGPP collection includes the GCC compiler and many of the tools. This is a 32-bit compiler and runs on DOS. This also includes an IDE called Rhide.
6. The Turbo C/C++ compilers can be downloaded from Borland after registering. You can also download the Borland C++ compiler without the IDE and other tools. You may have to use the compiler from the command line. The Turbo C/C++ compilers are pretty old and we do not advise you to use them.

This course has 20 practical sessions of 3 hours each. In Unit 18, we have given the list of programs you have to write in each of the 20 sessions. Instructions regarding practicals are already included in page 18 of the programme guide. In the practicals, for each of the programs that you write afresh, you have to either make a flow chart or an algorithm in pseudocode and show it to the counsellor. You can also use the format we have used in Unit 17 for writing two of the algorithms. After that, you have to write the C program, debug it and run and show the results to your counsellor. The counsellor may suggest changes to the program. After you carry out the changes and the counsellor is satisfied, you have to get the algorithm/flow chart and the C source signed by the counsellor.

Note that you have to do all the sessions marked as ‘Compulsory’. The division into sessions is not rigid. If you finish the programs in one session, you can do the programs for the next session. Also, create a separate folder(directory) to store your programs.
Laboratory Manual
# UNIT 16 INTRODUCTION TO COMPUTERS

<table>
<thead>
<tr>
<th>Structure</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>Objectives</td>
<td></td>
</tr>
<tr>
<td>16.2 Binary, Octal and Hexadecimal Number Systems</td>
<td>5</td>
</tr>
<tr>
<td>Conversion from One System to Another</td>
<td></td>
</tr>
<tr>
<td>Representation of Negative Integers</td>
<td></td>
</tr>
<tr>
<td>16.3 Working of Computers</td>
<td>10</td>
</tr>
<tr>
<td>Execution of Instruction</td>
<td></td>
</tr>
<tr>
<td>16.4 Evolution of Computers</td>
<td>17</td>
</tr>
<tr>
<td>16.5 Computer Software</td>
<td>19</td>
</tr>
<tr>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>Application Software</td>
<td></td>
</tr>
<tr>
<td>16.6 Categories of Languages</td>
<td>21</td>
</tr>
<tr>
<td>Machine Language</td>
<td></td>
</tr>
<tr>
<td>Assembly Language</td>
<td></td>
</tr>
<tr>
<td>High Level Languages</td>
<td></td>
</tr>
<tr>
<td>Fourth Generation Programming Languages</td>
<td></td>
</tr>
<tr>
<td>16.7 Summary</td>
<td>25</td>
</tr>
<tr>
<td>16.8 Solutions/Answers</td>
<td>25</td>
</tr>
</tbody>
</table>

## 16.1 INTRODUCTION

This Unit provides an introduction to computers. You will not be examined on the contents of this Unit. This gives a brief history of computers and gives you an idea of how they work. We advise you to browse through this Unit, skipping all the topics that you are familiar with. We strongly recommend that you read at least the second section of this Unit.

In Sec. 16.2, we will summarise binary, octal and hexadecimal systems. In Sec. 16.3, we will give an introduction to computers and describe how they work. In Sec. 16.4, we will discuss the evolution of computers. In Sec. 16.5, we will discuss what is a software and what are the different kinds of software. In Sec. 16.6, we will discuss computer languages.

### Objectives

After studying this unit, you should be able to

- explain the von Neumann architecture of computers;
- explain the binary, octal and hexadecimal systems;
- convert numbers from one system to another system;
- explain evolution of computers;
- explain the various types of softwares available; and
- explain the various categories of computer languages, their strengths and weaknesses.

## 16.2 BINARY, OCTAL AND HEXADECIMAL NUMBER SYSTEMS

In our daily life we use the decimal system to represent numbers. Recall that, in this system, there are ten symbols available, 0, 1, 2, 3, ..., 9 and each digit has a place value.
For example, in the number 21, 2 has a place value 20. We can also consider 21 as $2 \times 10^1 + 1 \times 10^0$. Similarly, we can consider 223 as $2 \times 10^2 + 2 \times 10^1 + 2 \times 10^0$. In other words, we assign a weight of $10^0$ to the first digit (from the right), a weight of $10^1$ to the second digit from the right and so on. This way, we can represent as large a number as we want using only 10 symbols!

In the case of computers things are different. Since computers are electronic instruments, only two states are available: Either there is a current flow or there isn’t. If we use the presence of current flow to represent 1 and its absence to represent 0, we have two symbols. So, we have to use binary system in computers to represent data in the inner working of the computers. Also, system programmers use octal and hexadecimal systems. In octal system we use 8 symbols 0,1,2,3,…,7 to represent numbers. The weights of the digits are $1 = 8^0$, $8^1$, $8^2 = 64$ etc. In hexadecimal system, we use 16 symbols, 0,1, 2, 3,…,9,A, B, C, D, E, F to represent the digits.

Please note that there is nothing sacrosanct about the decimal system. In fact, Mayans, who lived in South America centuries ago used base 20 system. Babylonians used base 60! They had 60 different symbols! You can see them in Fig. 1.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>21</td>
<td>31</td>
<td>41</td>
<td>51</td>
<td>61</td>
<td>71</td>
<td>81</td>
<td>91</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>22</td>
<td>32</td>
<td>42</td>
<td>52</td>
<td>62</td>
<td>72</td>
<td>82</td>
<td>92</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>23</td>
<td>33</td>
<td>43</td>
<td>53</td>
<td>63</td>
<td>73</td>
<td>83</td>
<td>93</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>24</td>
<td>34</td>
<td>44</td>
<td>54</td>
<td>64</td>
<td>74</td>
<td>84</td>
<td>94</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>25</td>
<td>35</td>
<td>45</td>
<td>55</td>
<td>65</td>
<td>75</td>
<td>85</td>
<td>95</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>26</td>
<td>36</td>
<td>46</td>
<td>56</td>
<td>66</td>
<td>76</td>
<td>86</td>
<td>96</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>27</td>
<td>37</td>
<td>47</td>
<td>57</td>
<td>67</td>
<td>77</td>
<td>87</td>
<td>97</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>28</td>
<td>38</td>
<td>48</td>
<td>58</td>
<td>68</td>
<td>78</td>
<td>88</td>
<td>98</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td>29</td>
<td>39</td>
<td>49</td>
<td>59</td>
<td>69</td>
<td>79</td>
<td>89</td>
<td>99</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>100</td>
</tr>
</tbody>
</table>

Fig. 1: Babylonian symbols.

We will discuss binary, octal and hexadecimal systems in this section. Since we will be representing numbers in various systems, we will specify the system by giving the base as subscript. For example $23_8$ indicates representation is in base 8, $11101_2$ indicates the representation is in base 2. If there is no subscript, the representation in in base 10. Let us now discuss how to convert from one system to another.

### 16.2.1 Conversion from One System to Another

As we mentioned before, we use two symbols, 0 and 1, in binary system. Here, each digit has a weight that is a power of 2. For example, the binary equivalent of 23 is $11101_2$ in binary because

$$1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1 + 2 + 4 + 8 + 16 = 23.$$  

Let us now see how to convert from other systems to the decimal system. We start with conversion from binary to decimal, which is easy. Let us look at an example.

**Example 1:** Convert $110111_2$ to decimal.
Solution: In decimal system, the right most digit is called the **least significant digit** and the left most digit is called the **most significant digit**. In binary representation, the right most digit is also called the **least significant bit (LSB)** and the right most digit is called the **most significant bit (MSB)**. Here is the step-by-step procedure for converting a number from binary to decimal:

1) In the first column of the first row, put the least significant bit. In the second column, put the corresponding weight, which is 1. In the third column, put the face value. (If you are puzzled why we have a column for the face value although it is the same as the digit, please be patient. You will understand the reason when we look at Example 2, where we show how to convert from hexadecimal to decimal.) Multiply the values in the second and third columns and put it in the fourth column.

2) Put the second digit from the right in the first column of the second row. In the second column, put the weight corresponding to the second digit. Note that, we can find the weight for a particular digit by multiplying the weight of the previous digit by the base. Here, base happens to be 2. So, multiplying the previous weight by 2, the weight is 2. In the third column, put the face value. Multiply the second and third columns and put the product in the fourth column. Carry this out for all the digits.

3) Add up all the numbers in the fourth column to get the decimal representation of the number.

* * *

Conversion from octal to decimal and hexadecimal to decimal are also similar as the following example shows.

**Example 2:**

a) Convert 2212\textsubscript{8} to decimal.

b) Convert A314F\textsubscript{16} to decimal.

**Solution:**

a) The procedure is similar to the one we used for the binary system. Note that, in octal system, the base is 8 and the weights are powers of 8.

**Table 2: Conversion from octal to decimal**

<table>
<thead>
<tr>
<th>Digit</th>
<th>Weight</th>
<th>Face value</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>2</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>1162</strong></td>
</tr>
</tbody>
</table>
b) Here, the base is 16 and the weights are powers of 16. Note that, here, A, B, C, D, E, F are used to represent 11, 12, 13, 14 and 15, respectively. Here, the digit is not the same as face value!(See Table 3.)

<table>
<thead>
<tr>
<th>Digit</th>
<th>Weight</th>
<th>Face value</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>256</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>3</td>
<td>4096</td>
<td>3</td>
<td>12288</td>
</tr>
<tr>
<td>A</td>
<td>65536</td>
<td>10</td>
<td>655360</td>
</tr>
</tbody>
</table>

Table 3: Conversion from hexadecimal to decimal

Here are some exercises for you.

E1) Convert the following numbers to decimal system:
   a) 111011₂
   b) 33464₃₈
   c) A123BC₁₆

Converting from decimal to other systems is also easy. Here is an example:

**Example 3:** Convert the following numbers from decimal to the system indicated:

a) Convert 59 to binary.

b) Convert 1162 to octal.

c) Convert 10560444 to hexadecimal.

**Solution:**

a) See Table 4a. We divide the number by 2. The remainder gives the least significant bit. We divide quotient by 2. This will give the next digit. We continue like this till we get 1 as the quotient. The last quotient and the successive remainders give the digits of the number in binary. The last quotient gives the most significant digit and the first remainder gives the least significant digit. We carry out this for the number 59. The calculation is given in Table 4a. So, 59 is 111011₂.

<table>
<thead>
<tr>
<th>(a) Conversion to binary.</th>
<th>(b) Conversion to octal.</th>
<th>(c) Conversion to Hexadecimal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 59</td>
<td>8 1162 → 2</td>
<td>16 10560444 → 11(= C)</td>
</tr>
<tr>
<td>2 29 → 1</td>
<td>8 145 → 1</td>
<td>16 660027 → 10(= B)</td>
</tr>
<tr>
<td>2 14 → 1</td>
<td>8 18 → 2</td>
<td>16 42151 → 3</td>
</tr>
<tr>
<td>2 7 → 0</td>
<td>2</td>
<td>16 2578 → 2</td>
</tr>
<tr>
<td>2 3 → 1</td>
<td></td>
<td>16 161 → 1</td>
</tr>
<tr>
<td>1 → 1</td>
<td></td>
<td>10 (= A)</td>
</tr>
</tbody>
</table>

b) See Table 4b. In this case we divide by 8. As before, we repeatedly divide by 8 till we get a quotient less than 8. The successive remainders and the last quotient will give the digits of the number in octal. The last quotient which is less than 8 will give the most significant digit. So, 1162 is 2212₈.

c) See Table 4c. In this case, we divide by 16. Note that, the last quotient 10 is represented by the letter A. So, 1056044 is A123BC₈ in hexadecimal.

***
Here are some exercises for you to try.

**E2)** Convert the following numbers from decimal to the system indicated:

a) 61 to binary.
b) 113059 to octal.
c) 10560446 to hexadecimal.

It is also easy to convert from binary to octal or hexadecimal. Let us take an example. First, let us see how to convert from binary to octal. Suppose, we want to convert 11111101₂ into octal system. Starting from the right we group the digits in bunches of three:

```
11 111 101
```

The last bunch has only two digits. Now, we find the value of each group in decimal. For example, the value for the first bunch from the right, 101, is 5. Similarly, we find the value of the remaining bunches.

```
11 111 101
3  7  5
```

These values give the digits of 11111101₂ in octal, i.e. the value of the number in octal is 375₈.

We can convert a binary number to hexadecimal similarly. Here, we have to group the digits in bunches of 4. For example, suppose we want to convert the number 111011110₂ to hexadecimal. We start from the right and group the digits in bunches of 4.

```
11 1101 1110
```

As before, we find the value of each bunch.

```
11 1101 1110
3  13 14
```

So, the number in hexadecimal is 3DE₁₆.

**E3)** Convert 11110101₂ into octal and hexadecimal systems.

In all the examples we have discussed, all the integers were positive. In the next subsection, we will discuss how to represent negative numbers.

### 16.2.2 Representation of Negative Integers

As we mentioned earlier, at the lowest level, the CPU recognises only 0s and 1s. How can we store and work with negative numbers? The solution is to reserve the left most bit for the sign; it is 1 if the number is negative and 0 if it is positive.

While this solves the representation problem it still doesn’t solve the problem of working with such numbers. Assuming that we are using 8 bits to store numbers for simplicity, we store 1 as 0000 0001₂ and −2 as 1000 0001₂. But, if we add them, we get 1000 0011₂ which is −3 according to our notation!

One way to get round this problem is the 1’s complement arithmetic. In this representation, we first represent the number without the sign and then complement
Laboratory Manual

each bit; we replace 0 by 1 and 1 by 0. For example, 5 is 0000 0101._2. If we complement the bits we get 1111 1010._2 so −5 is 1111 1010._2. If we add 2 = 0000 0010._2 to −5 = 1111 1010._2, we get 1000 1100 which is −3 because 3 = 0000 0011._2 and if we complement the bits we get 1000 1100._2. So, addition seems to work fine. But what happens when we add 2 and −2? We get 1111 1111._2 which is ‘−0’ in 1’s complement. But, there is no such thing in Mathematics!

The way around this problem is to use 2’s complement arithmetic. In this, we add 1 to the 1’s complement representation. So, −2 = 1111 1010._2 in 1’s complement and 11111101._2 + 1 = 11111110._2 in 2’s complement. Now,

−2 + 2 = 1111 1110._2 + 0000 0010 = 0000 0000._2

with the last carry from got from adding the most significant bits of the numbers discarded. Now, there is only one zero and everything is fine! Here are some exercises for you to check your understanding of 2’s complement arithmetic.

E4) Write −10 and −17 in 2’s complement notation.

E5) Check that 2’s complement representation works fine with multiplication also.

Represent −3 and −5 in 2’s complement notation and multiply them. Check that your answer is correct.

We end this section here. In the next section, we will introduce you to the working of computers.

16.3 WORKING OF COMPUTERS

Computer is defined in the Oxford dictionary as “An automatic electronic apparatus for making calculations or controlling operations that are expressible in numerical or logical terms.”

The definition clearly categorises computer as an electronic apparatus although the first computers were mechanical and electro-mechanical apparatus. The definition also mentions the two major areas of computer application viz. data processing and computer assisted controls/operations. We can also infer from the definition that the computer can perform only Logical operations or Numerical operations.

The concept of computer is quite old. A hypothetical machine was defined in 1935–36 by Alan Turing, which was used for computability theory proofs. This is known as a Turing machine. It consists of an infinitely long ‘tape’ with symbols written at regular intervals. The tape may be infinite in one direction only, with the understanding that the machine will halt if it tries to move off the other end. All computer instruction sets, high level languages and computer architectures, including parallel processors, can be shown to be equivalent to a Turing Machine. Thus, all of them are equivalent to each other in the sense that if a problem can be solved by one of them, it can be solved by all the others. While serving on the BRL (Ballistic Research Laboratories) Scientific Advisory Committee, von Neumann joined the developers of ENIAC, the first tube computer and made some critical contributions. In 1947, while working on the design for the successor machine, EDVAC, von Neumann realised that ENIAC’s lack of a centralised control unit could be overcome to obtain a rudimentary stored program computer. His ideas lead to what is now often called the von Neumann architecture. Most of today’s computer designs are based on the concepts developed by John von Neumann, referred to as the von Neumann architecture. Von Neumann proposed that there should be a unit performing arithmetic and logical operation on the data. This unit
is termed as **Arithmetic Logic Unit (ALU)**. One of the ways to provide instruction to such computer will be by connecting various logic components in such a fashion that they produce the desired output for a given set of inputs. The process of connecting various logic components in specific configuration to achieve desired results is called **programming**. This programming, done by providing instructions within hardware by various connections, is termed as **hard-wired**. But this is a very inflexible process of programming.

Let us now look at the general configuration for arithmetic and logical functions. We need to have a control signal, which directs the ALU to perform a specific arithmetic or logic function on the data. By changing the control signal, we can perform the desired function on the data. We can perform any operation by providing appropriate signals. Thus, for a new operation it is enough to change control signals.

But, how can we supply these control signals? Let us try to answer this from the definition of a program. A program consists of a sequence of steps. Each of these steps requires certain arithmetic or logical or input/output operation to be performed on the data. Therefore, each step may require a new set of control signals. Is it possible for us to provide a unique code for each set of control signals? Well, the answer is ‘yes’. But what do we do with these codes? It is necessary to have a hardware segment, which accepts a code and generates control signals. The unit, which interprets a code to generate respective control signal, is termed as **Control Unit (CU)**. Thus, a program consists of a sequence of codes. Now, this machine is quite flexible, as we only need to provide a new sequence of codes (program) for a new task to be performed. Each code is, in effect, an instruction, for the computer. The hardware interprets each of these instructions and generates respective control signals.

The Arithmetic Logic Unit (ALU) and the Control Unit (CU) together are termed as the **Central Processing Unit (CPU)**. The CPU is the most important component of a computer’s hardware. The control unit interprets instructions and produces the respective control signals. The ALU performs the arithmetic operations such as addition, subtraction, multiplication and division, and the logical operations such as: ‘Is \( A = B \)’ (where A and B are both numeric or alphanumeric data) etc.

All the arithmetic and logical operations are performed in the CPU in special storage areas called **registers**. The size of the register is one of the important considerations in determining the processing capabilities of the CPU. Register size refers to the amount of information (number of bits) that can be held in a register at a time for processing. The larger the register size, the faster may be the speed of processing. A CPU’s
A von Neumann machine has only a single path between the main memory and control unit (CU). This constraint is referred to as von Neumann bottleneck. Several other architectures have been suggested for modern computers.

Now, how can we feed the instructions and data into computers? An external environment supplies the instruction and data, therefore, an input module is needed. The main responsibility of input module is to put the data in the form of signals that can be recognised by the system. Similarly, we need another component, which will report the results in proper format and form. This component is called output module. These components are referred together as input/output (I/O) devices or peripheral devices. Most common input/output devices are keyboard, mouse, monitor and printer. In addition, to transfer the information, the computer system internally needs the system interconnections called Bus which carries data and addresses.

Are these two components sufficient for a working computer? No, because input devices can bring instructions or data only sequentially and a program may not be executed sequentially as jump instructions are normally encountered in programming. In addition, more than one data element may be required at a time. Therefore, a temporary storage area is needed in a computer to store temporarily the instructions and the data. This component is referred to as memory. It was pointed out by von Neumann that the same memory can be used for storing data and instructions. In such cases the data can be treated as data on which processing can be performed, while instructions can be treated as data, which can be used for the generation of control signals.

The memory unit stores all the information in a group of memory cells, also called memory locations, as binary digits. Each memory location has a unique address and can be addressed independently. The contents of the desired memory locations are provided to the CPU by referring to the address of the memory location. The amount of information that can be held in the main memory is known as memory capacity. The processing power is measured in Million Instructions per Second (MIPS). The performance of the CPU was measured in milliseconds (one thousandth of a second) on the first-generation computers, in microseconds (one millionth of a second) on second-generation computers, in nanoseconds (one billionth of a second) on third-generation computers, and is measured in picoseconds (one 1000th of a nanosecond) or femtoseconds (one 1000th of a picosecond) in the later generations.
capacity of the main memory is measured in kilobytes (kB) or Megabytes (MB). One kilobyte equals \(2^{10}\) bytes, which are 1024 bytes (or approximately 1,000 bytes). A megabyte equals \(2^{20}\) bytes, which is approximately little over one million bytes (1,048,576 bytes to be precise). Since the memory on the computer chip is limited, in order to provide extended memory, devices like hard discs, floppy discs, compact discs (CD), magnetic tapes are used as external storage (memory) devices.

The basic function performed by a computer is the execution of a program. A program is a sequence of instructions, which operates on a data to perform certain tasks. In the computers data is represented in binary form by using two symbols 0 and 1, which are called binary digits or bits. But the data which we deal in practice consists of numeric data and characters such as decimal digits 0 to 9, alphabets A to Z, arithmetic operators (e.g. +, −, etc.), relations operators (e.g. =, >, etc.), and many other special characters (e.g.;@,[,], etc.). In general, computers use eight bits to represent a character internally. This allows up to 256 different items to be represented uniquely. This collection of eight bits is called a byte. Thus, one byte is used to represent one character internally. Most computers use two bytes or four bytes to represent numbers (positive and negative) internally. Another term, which is commonly used in computer, is a word. A word may be defined as a unit of information, which a computer can process, or transfer at a time. A word, generally, is equal to the number of bits transferred between the central processing unit and the main memory in a single step. It may also be defined as the basic unit of storage of integer data in a computer. Normally, a word may be equal to 8, 16, 32 or 64 bits. The terms like 32-bit computer, 64-bit computers, etc. basically refer to the word size of the computer.

The von Neumann machine uses stored program concept, i.e., the program and data are stored in the same memory unit. The computers prior to this idea used to store programs and data on separate memories. Entering and modifying these programs were very difficult as they were entered manually by setting switches and plugging and unplugging.

---

E6) State whether following statements are true or false.

i) A byte is equal to 8 bits and can represent a character internally.

ii) Von Neumann architecture specifies different memory for data and instructions. The memory, which stores data, is called data memory, which stores instructions, is called instruction memory.

iii) In Von Neumann architecture each bit of memory can be accessed independently.

iv) A program is a sequence of instructions designed for achieving a task/goal.

v) One MB is equal to 1024 kB.

---

In the next subsection, we discuss how the instructions in a program are executed in a computer.

### 16.3.1 Execution of Instruction

The main aspect in program execution is the execution of an instruction. The key questions, which can be asked in this respect, are: (a) how are the instructions supplied to the computer? and (b) how are they interpreted and executed? We will answer these questions now.

Execution of instructions in von Neumann machine is carried out in a sequential fashion (unless explicitly altered by the program itself) from one instruction to the next.
The program, which is to be executed, is a set of instructions, stored in the memory in form of **machine language** which comprises of ‘0’s and ‘1’s. The central processing unit (CPU) executes the instructions of the program to complete a task. The instruction execution takes place in the CPU registers.

Let us, first discuss few typical registers, which are generally available in the machines. These registers are:

**Memory Address Register (MAR):** It specifies the address of memory location from which data or instruction is to be accessed (for read operation) or to which the data is to be stored (for write operation).

**Memory Buffer Register (MBR):** This register contains the data to be written in the memory (for write operation) or it receives the data from the memory (for read operation).

**Program Counter (PC):** It keeps track of the instruction, which is to be executed next, after the execution of the on-going instruction.

**Instruction Register (IR):** Here the instructions are loaded before their execution.

The simplest model of instruction processing can be a two-step process. The CPU reads (‘fetches’) instructions (codes) from the memory one at a time, and executes or performs the operation specified by this instruction. The fetch operation has to be carried out for all the instructions. It involves reading of an instruction from a memory location to the CPU. The execution of this instruction may involve several operations depending on the nature of the instruction.

The processing needed for a single instruction (fetch and execution) is referred to as instruction cycle. The instruction cycle consists of the fetch cycle and the execute cycle. Program execution terminates if the electric power supply is discontinued or some sort of unrecoverable error occurs; Also, a program can terminate itself.

For fetch cycle, a typical CPU uses a program counter (PC). PC keeps track of the instruction which is to be fetched next. Normally next instruction in sequence is fetched next as programs are executed in sequence.

The fetched instruction is in the form of binary code and is loaded into an instruction register (IR) in the CPU. The CPU interprets the instruction and does the required action. In general, these actions can be divided into the following categories:

**Data Transfer:** From the CPU to memory or from memory to CPU, or from CPU to I/O or I/O to CPU.

**Data Processing:** A logic or arithmetic operation performed by the CPU on the data.

**Sequence Control:** This action may require alteration of sequence of execution. For example, an instruction from location 100 H on execution may specify that the next instruction should be fetched from location 200 H. On execution of such an instruction the Program Counter that was having location value 101 H (the next instruction to be fetched in case where memory word is equal to register size) will be modified to contain a location value 200 H.

Execution of an instruction may involve any combination of these actions. Let us understand the process with an example.

**Example 4:** Let us assume a hypothetical machine which has a 16 bit instructions and data. Each instruction of the machine consists of two components: (a) operation code (op-code) and (b) address of the operand in memory.
The op-code is assumed to be of 4 bits; therefore, remaining 12 bits are for the address of the operand as shown in Fig. 4a. The memory is divided into words of 16 bits. The CPU of this machine contains a register called Accumulator (AC) in addition to the registers given earlier. The AC stores the data temporarily. Fig. 4 shows the data format, which caters for sign bit. The machine can have $2^4 = 16$ possible operation codes. For example, let us assume operation with codes:

- 0001 as “Load the accumulator with the content of memory”
- 0010 as “Store the current value of Accumulator in the memory”
- 0011 as “Add the value from memory to the Accumulator”

This machine can address $2^{12} = 4096$ memory words directly. Remember that in this case the program counter is of 12 bits to accommodate address of the memory where the program resides.

Suppose we want to load the content at the memory location 760 H. 760 H is 0111 0110 0000 in binary. The opcode for loading the memory content on to the Accumulator is 0001. So, the instruction is 0001 0111 0110 0000 in binary.

Note that the first four bits contain the opcode and the next 12 bits contain the address on which the operation is to be carried out. We can write the instruction succinctly as 1760 H in hexadecimal.

Let us assume that three consecutive instructions to be executed are

<table>
<thead>
<tr>
<th>Instruction code</th>
<th>Operation desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0111 0110 0000 (Load accumulator with content at memory location 760 H)</td>
</tr>
<tr>
<td>0011</td>
<td>0111 0110 0001 (Add value from memory 761 H to accumulator)</td>
</tr>
<tr>
<td>0010</td>
<td>0111 0110 0000 (Store content of AC in the memory location 760 H)</td>
</tr>
</tbody>
</table>

The hexadecimal notation for these instructions is:

- 1760H
- 3761H
- 2760H

Let us assume that these instructions are stored in three consecutive memory locations 101 H, 102 H and 103 H and the PC contains a value (101 H), which is the address of first of these instructions as shown in Fig. on the next page.) Since the PC contains value 101 H, so on the next instruction cycle the address stored in PC is passed to MAR which in turn helps in accessing the memory location 101 H and brings its content in MBR which in turn passes it on to IR. The PC is incremented to contain 102 H now. The IR has the value 1760 H that is decoded as “Load the content of address 760 H in the accumulator”. (Refer to Fig. on the following page.) Thus, the accumulator register is loaded with the content of location 760 H that is 0004 H.
Laboratory Manual

Now the instruction 101 H execution is complete, and the next instruction at 102 H (indicated by PC) is fetched and PC is incremented to 103 H. This instruction is 3761 H which implies “add the content of memory location 761 H to the accumulator”. Therefore, accumulator will now contain the sum of its earlier value and the value stored in memory location 761 H. (Refer to Fig. 5c.)

On execution of the instruction at memory location 103 H, PC becomes 104 H; the accumulator results are stored in location 760 H and IR still contains the third instruction. This state is shown in Fig. 5d.

***

Please note that the execution of the instructions in the above example require only data transfer and data processing operations. All the instructions of the example required one memory reference during their execution. Does an instruction require more than one memory reference?

Well, yes! One such instruction is ADD A, B. The execution cycle of this instruction may consist of steps such as:

1. Decode the instruction, which is ADD.
2. Read memory location A (only the contents) into the CPU.
3. Read memory location B (only the contents) into the CPU. (Here, we are
assuring that the CPU has at least two registers for storing memory location contents. The contents of location A and location B need to be written in two different registers.)

4. Add the values of the two above registers.

5. Write back the result from the register containing the sum (in the CPU) to the memory location B.

Thus, in general, the execution cycle for a particular instruction may involve more than one stage and memory references. In addition, an instruction may ask for an I/O operation. For such purpose the ports where the I/O devices are connected are allotted addresses just like any other memory location. When a data from such device is required, the address of the device is placed in the instruction instead of address of a memory location.

E7) What will be the steps involved in interchanging the data in memory location A and memory location B?

After understanding the fundamental operation of a computer, let us now briefly discuss the evolution of computers over the years.

**16.4 EVOLUTION OF COMPUTERS**

You know that, with the growth in microelectronics the IC(integrated circuit) technology evolved rapidly. One of the major milestones in this technology was the very large scale integration (VLSI) where thousands of transistors could be integrated on a single chip. The main impact of VLSI was that, it was possible to produce a complete CPU or main memory or other similar devices on a single IC chip. This implied that mass production of CPU, memory etc. can be done at a very low cost. Let us take a brief review of the important breakthroughs of VLSI technologies.

**Semiconductor Memories:** Initially the IC technology was used for constructing processor, but soon it was realised that same technology can be used for construction of memory. The first memory chip was constructed in 1970 and could hold 256 bits. Although, the cost of this chip was high initially, gradually it is going down. The memory capacity per chip has increased as: 1k, 4k, 16k, 64k, 256k and 1M bytes or even more. Now, a standard PC sold in the market has a memory of 256MB or more. These memories can be random access memory (RAM), read only memory (ROM) or erasable, programmable read only memory (EPROM).

**Microprocessors:** Keeping pace with the electronics as more and more components were fabricated on a single chip, fewer chips were needed to construct a single processor. Intel in 1971 achieved the breakthrough of putting all the components on a single chip. The single chip processor is known as a microprocessor. The Intel 4004 was the first microprocessor. It was a primitive microprocessor designed for a specific application. Intel 8080, which came in 1974, was the first general-purpose microprocessor. It was an 8-bit microprocessor. Motorola is another manufacturer in this area. At present 32- and 64-bit general-purpose microprocessors are already in the market. For example Intel 486 is a 32-bit processor, similarly Motorola’s 68000 is a 32 bit microprocessor. Pentium, which was developed by Intel in 1993, processes integers as long as 64 bits. The VLSI technology is still evolving more and more powerful microprocessors and more storage space now is being put in a single chip.
In Fig. 6, we depict the evolution of Intel microprocessor families, which started with an 8-bit chip to develop to latest available 32/64-bit Pentium chips. The computers are broadly classified as micro-computers, mini-computers, mainframe computers and supercomputers. Although with development in technology the distinction between all these is becoming blurred, yet it is important to classify them as it is sometimes useful to differentiate the key elements and architecture among the classes.

a. Micro-computers

The CPU of a micro-computer is a microprocessor. The micro-computer originated in late 1970’s. The first micro-computers were built around 8-bit microprocessor chips. What do we mean by an 8-bit chip? It means that the chip can retrieve instructions/data from storage, manipulate, and process an 8-bit data at a time or we can say that the chip has a built-in 8-bit data transfer path. 8088 was a 8/16 bit chip i.e. an 8-bit path was used to move data between chip and primary storage (external path), but processing was done within the chip using a 16-bit path (internal path) at a time. 8086 was a 16/16 bit chip i.e. the internal and external paths both were 16 bit wide. Both these chips could support a primary storage capacity of up to 1 MB.

b. Mini-computer

The term mini-computer originated when it was realised that many computing tasks do not require an expensive contemporary mainframe computers but can be solved by a small, inexpensive computer. It is in the middle range of computing, between mainframes and micro computers. Earlier, mini-computers had distinct hardware and operating system features that separated them from the other two types. With advances in technology, the distinction between micro-computers and mini-computers is getting increasingly blurred.

With the advancement in technology the speed, memory size and other characteristics developed and the mini-computer was then used for various stand alone or dedicated applications. The mini-computer was then used as a multi-user system, which can be used by various users at the same time. This was actually the first conceptual realisation of computer networking, where the multiple users were connected using hardware. The concept started from the fact that the CPU performed the processing at much faster...
rates than those required to entire new data and its display. The spare unused time of CPU could be used by other users. Gradually the architectural requirement of mini-computers grew and a 32-bit mini-computer, which was called super-mini, was introduced. The super-mini had more peripheral devices, larger memory and could support more users working simultaneously on the computer in comparison to previous mini-computers. This class of machines are now more commonly used as servers.

c. Mainframes

Mainframe computers are suited to big organisations, to manage high volume applications. They can run multiple operating systems at the same time. They are generally used in business applications. The term mainframe is being gradually replaced by the term enterprise server. Few of popular mainframe series are DEC, IBM, HP, ICL, MEDHA, Sperry, etc. Mainframes are also used as central host computers in distributed systems. Libraries of application programs developed for mainframe computers are much larger than those of the micro- or mini-computers because of their evolution over several decades as families of computing. All these factors and many more make the mainframe computers indispensible even with the popularity of micro-computers.

d. Supercomputers

The upper end of the state of the art mainframe machine is the supercomputer. Super computers are amongst the fastest machines in terms of processing speed and use multiprocessing techniques, where a number of processors are used to solve a problem. There are a number of manufacturers who dominate the market of supercomputers—CRAY (CRAY YMP, CRAY 2), ETA (CDC-ETA 10, ETA 20), IBM 3090 (with vector), NEC (NEC SX-3), Fujitsu (VP Series) and HITACHI (S Series) are same of them. Lately ranges of parallel computing products, which are multiprocessors sharing common buses, have been in use in combination with the mainframe supercomputers. The supercomputers are reaching up to speeds well over 25000 million arithmetic operations per second. India has also developed its indigenous supercomputer PARAM.

Supercomputers are mainly being used for number crunching problems such as weather forecasting, computational fluid dynamics, remote sensing, image processing, biomedical applications, etc.

E8) i) What is a general-purpose machine?
    ii) What are the advantages of IC technology over discrete components?

The processor, memory and peripheral devices comprise the hardware part of the computer. However, all these parts function only because of the software incorporated in the computers. Let us now take brief account of computer software.

16.5 COMPUTER SOFTWARE

Computer software consists of sets of instructions that mould the raw arithmetic and logical capabilities of the hardware units to perform. Computer software can be broadly classified into two categories—Systems Software and Application Software.

Today, there are many languages available for developing program software. These languages are designed keeping in mind some specific areas of applications. Thus, some of the language may be good for writing system program/software while some other for application software.
Laboratory Manual

i) **System Programming Languages**: System programs are designed to make the computer easier to use. An example of system software is an operating system, which consists of many other programs of controlling input/output devices, memory, processor, etc. To write an operating system, the programmer needs instruction to control the computer’s circuitry (hardware part). For example, instructions that move data from one location of storage to a register of the processor. Today languages like C, C++ and C# languages are widely used to develop system software.

ii) **Application Programming Language**: Application programs are designed for specific computer applications, such as payroll processing, inventory control, word processing, etc. To write programs for pay-roll processing or other applications, the programmer does not need to control the basic circuitry of a computer. Instead the programmer needs instructions that make it easy to input data, produce output, do calculations and store and retrieve data. Programming languages that are suitable for such application programs support these instructions but not necessarily the types of instructions needed for development of system programs.

There are two main categories of application programs: **Business programs** and **scientific application programs**. Most programming languages are designed to be good for one category of application but not necessarily for the other, although there are some general-purpose languages that support both types. Business applications are characterised by processing of high volume data but call for simple calculations. Languages which are suitable for business program development must support high volume input, output and storage but need not support complex calculations. On the other hand, programming languages that are designed for writing scientific program contain very powerful instructions for calculations but rather poor instructions for input, output, etc. Amongst traditionally used programming languages, COBOL (Commercial Business Oriented Programming Language) is more suitable for business applications whereas FORTRAN (Formula Translation Language) and C-language are more suitable for scientific applications. Before we discuss more about language let us briefly look at the categories of software viz. system and application software.

16.5.1 **System Software**

**Operating System**

An operating system (OS) is the most important system software and is a must to operate a computer system. An operating system manages computer’s resources very effectively, takes care of scheduling multiple jobs for execution and manages the flow of data and instructions between the input/output units and the main memory. Operating system became a part of computer software with the second-generation computers. Since then operating systems have undergone several revisions and modifications in order to achieve a better utilisation of computer resources. Advances in the field of computer hardware have also helped in the development of more efficient operating systems.

**Language Translator**

A language translator is a system software which translates a computer program written by a user into a machine understandable form.

**Utilities**

Utility programs are those which are very often requested by many application programs. A few examples are:

1) **SORT/MERGE** for sorting large volumes of data and merging them into a single sorted list.
2) Transfer program for transforming contents from one storage medium to another, e.g. disk to tape, tape to disk, etc.

Special Purpose Software

Special purpose program are those which extend the capability of operating systems to provide specialised services to application programs. A few examples are:
1) Spreadsheet software like Excel, Openoffice Calc, etc.
2) Data management software like Oracle, MySQL, SQLServer, etc.

These programs can also be used as stand alone application programs.

16.5.2 Application Software

Application software is written to enable the computer to solve a specific data processing task. There are two categories of application software:

- Software packages that are ready for use, and
- User application program.

A number of powerful application software packages, which do not require significant programming knowledge, have been developed. These are easy to learn and use as compared to the programming languages. Although these packages can perform many general and special functions, there are applications where these packages are not found adequate. In such cases, application program is written to meet the exact requirement. A user application program may be written using one of these packages or a programming language. Some important categories of software packages available are:

- Data Base Management Software
- Word Processing Desktop Publishing (DTP) and presentation Software
- Graphics Software
- Data Communication Software
- Statistical and Operational Research Software

Try the following exercises now.

E9) Explain the following terms in one or two sentences each:
   (a) Operating Systems, and (b) Database Management Software

In the next section, we will discuss different categories of languages for writing software and their advantages and disadvantages.

16.6 CATEGORIES OF LANGUAGES

You can choose any language for writing a program according to the need. But a computer executes programs only after they are represented internally in binary form (sequences of 1s and 0s). Program written in any other languages must be translated to the binary representation of the instructions before they can be executed by the computer. Program written for a computer may be in one of the following categories of languages.
16.6.1 Machine Language

This is a sequence of instructions written in the form of binary numbers consisting of 1s, 0s to which the computer responds directly. You have used this language in Example 1. The machine language was initially referred to as code, although now the term code is used more broadly to refer to any program text.

As you have already learnt, an instruction prepared in any machine language will have at least two parts. The first part is the command or Operation, which tells the computer what function is to be performed. All computers have an operation code for each of its functions. The second part of the instruction is the operand or it tells the computer where to find or store the data that has to be manipulated.

Just as hardware is classified into generations based on technology, computer languages also have a generation classification based on the level of interaction with the machine. Machine language is considered to be the first generation language.

Advantage of Machine Language

It is faster in execution since the computer directly starts executing it.

Disadvantage of Machine Language

It is difficult to understand and develop a program using machine language. Anybody going through this program for checking will have a difficult task understanding what will be achieved when this program is executed. Nevertheless, the computer hardware recognises only this type of instruction code.

16.6.2 Assembly Language

When we employ symbols (letter, digit or special characters) for the operation part, the address part and other parts of the instruction code, this representation is called an assembly language program. This is considered to be the second generation language.

Machine and Assembly languages are referred to as low-level languages since the coding for a problem is at the individual instruction level.

Each machine has got its own assembly language, which is dependent upon the internal architecture of the processor.

An Assembler is a translator, which takes its input in the form of an assembly language program and produces machine language code as its output.

The following program is an example of an assembly language program for adding two numbers \(X\) and \(Y\) and storing the result in some memory location.

\[
\begin{align*}
ld &A, 7 &; \text{Load register} \ A \ \text{with} \ 7 \\
ld &B, 10 &; \text{Load register} \ B \ \text{with} \ 10 \\
add &A, B &; \ A \leftarrow A + B \\
ld & (100), A &; \text{Save the result in the location} \ 100 \\
halt & &; \text{Halt process}
\end{align*}
\]

From this program, it is clear that usage of mnemonics (in our example \(ld\), \(add\), \(halt\) are mnemonics) has improved the readability of our program significantly.

An assembly language program cannot be executed by a machine directly as it is not in a binary form. An assembler is needed in order to translate an assembly language program into the object code executable by the machine.
Introduction to Computers

Advantage of Assembly Language

Writing a program in assembly language is more convenient than in machine language. Instead of binary sequence, as in machine language, it is written in the form of symbolic instructions. Therefore, it gives better readability than the machine language program.

Disadvantages of Assembly Language

Assembly language (program) is specific to particular machine architecture. Assembly languages are designed for specific make and model of a microprocessor. It means that assembly language programs written for one processor will not work on a different processor if it is architecturally different. That is why the assembly language program is not portable.

Assembly language program is not as fast as machine language since it has to be first translated into machine (binary) language code.

16.6.3 High-Level Languages

You must have already heard about the programming languages such as COBOL, FORTRAN, BASIC, PASCAL, JAVA, C, etc. They are called high-level programming languages. The time and cost of creating machine and assembly language was quite high. And this was the prime motivation for the development of high-level languages. The program shown below is written in BASIC to obtain the sum of two numbers.

```
10 LET X = 7
20 LET Y=10
30 LET SUM=X+Y
40 PRINT SUM
50 END
```

The high-level source program must be translated first into the form that machine can understand. This is done by a software called **compiler** which takes the source code as input and produces an output in the machine language code of the machine on which it is to be executed.

During the process of translation, the compiler reads the source program statements one by one and checks the syntax (grammatical) errors. If there is any error, the computer generates a print-out of the errors it has detected. This action is known as **diagnostics**.

There is another type of software, which also does the translation. This is called an interpreter. The compiler and interpreter have different approaches to translation. Table 5 lists the differences between a Compiler and an Interpreter.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Scans the entire program first and then translates it into machine code.</td>
<td>Translates the program line by line.</td>
</tr>
<tr>
<td>2. Converts the entire program to machine code; when all the syntax errors are removed execution takes place.</td>
<td>Each time the program is executed every line is checked for syntax error and then converted to equivalent machine code.</td>
</tr>
<tr>
<td>4. Execution time is less.</td>
<td>Execution time is more.</td>
</tr>
</tbody>
</table>

Advantage of High-level Programming Language: There are four main advantages of high-level programming languages. These are:
Readability: Programs written in these languages are more readable than assembly and machine language.

Portability: Programs could be run on different machines with little or no change. We can, therefore, exchange software leading to creation of program libraries.

Easy debugging: Errors could easily be removed (debugged).

Easy Software development: Software could be easily developed. Commands of programming language are similar to natural languages (English).

16.6.4 Fourth Generation of Programming Languages

The fourth generation of programming languages is not as clearly defined, as are the other earlier generations. Most people feel that a fourth generation language, commonly referred to as 4GL is a high level language that requires significantly fewer instructions to accomplish task than a third generation language does. Thus, a programmer should be able to write a program faster in 4GL than in third generation language.

Most third generation languages are procedural languages. This means that the programmer must specify the steps, that is the procedure, the computer has to follow in a program. By contrast, most fourth generation languages are non-procedural languages. The programmer does not have to give the details of procedure in the program but instead, specifies what is wanted. For example, assume that a programmer needs to display some data on a screen, such as the address of a particular employee (SANTOSH) from the personnel file. In a procedural language, the programmer would have to write a series of instructions in the following steps:

Step 1: Get a record from the personnel file.

Step 2: If this is the record for SANTOSH, display the address.

Step 3: If this is not the record for SANTOSH, go to Step 1.

In a non-procedural language (4GL), however, the programmer would write a single instruction that says:

Get the address of SANTOSH from personnel file.

Major fourth generation languages are used to get information from files and data base, as in the above example and to display or print the information. Fourth generation languages are mostly machine independent. Usually they can be used on more than one type of computer. They are mostly used for office automation or business applications, but not for scientific program. Some fourth generation languages are designed to be easily learnt and used by end users.

E10) State whether the following statements are True or False.

a) Assembly language programs are machine independent.

b) Machine language programs are machine independent.

c) High level languages are machine independent.

d) All programs are machine independent.

e) 4th generation languages are dependent on databases they use.

We close the section here. In the next section, we will summarise our discussion in this Unit.
16.7 SUMMARY

In this Unit we have discussed

1. the von Neumann architecture of computers;
2. the binary, octal and hexadecimal systems;
3. how to convert numbers from one system to another system;
4. the evolution of computers;
5. the various types of softwares available; and
6. the various categories of computer languages, their strengths and weaknesses.

16.8 SOLUTIONS/ANSWERS

E1) a) Digit | Weight | Face value | Product
   1   | 1     | 1          | 1
   0   | 2     | 1          | 0
   1   | 4     | 0          | 4
   1   | 8     | 1          | 8
   1   | 16    | 1          | 16
   1   | 32    | 1          | 32

   61

b) Digit | Weight | Face value | Product
   3   | 1     | 3          | 3
   4   | 8     | 4          | 32
   6   | 64    | 6          | 384
   4   | 512   | 4          | 2048
   3   | 4096  | 3          | 12288
   3   | 32768 | 3          | 98304

   113059

c) Digit | Weight | Face value | Product
   C   | 1     | 12         | 12
   B   | 16    | 11         | 176
   3   | 256   | 3          | 768
   2   | 4096  | 2          | 8192
   1   | 65536 | 1          | 65536
   A   | 1048576 | 10     | 10485760

   10560444

E2) a) 2 161 → 1  b) 8 113059 → 3  c) 16 10560446 → 12 (= E)
   2 30 → 0 8 14132 → 4 16 660027 → 11 (= B)
   2 15 → 1 8 1766 → 6 16 41251 → 3
   2 7 → 1 8 220 → 4 16 2578 → 2
   2 3 → 1 8 27 → 3 16 161 → 1

   1 10(= A)

E3) Grouping in bunches of 3 and finding the value of each bunch, we have
   11 110 1011. So, the given number is 3658 in octal.

   Grouping in bunches of 4 and finding the value of each bunch, we have
   1111101011. So, the number is F516 in hexadecimal.
E4) \(-10 = 1111 0110\) in 2s complement notation. \(-17 = 0001 0010\) in 2’s complement notation.

E5) \(-3 = 1111 1100\) in 1’s complement and 1111 1101 in 2’s complement. \(-5\) is 1111 1011 in 2’s complement. We have worked out the multiplication below. Note that the multiplication is similar to usual multiplication. The first digit in the multiplier 1111 1011 is 1. So, we just copy the multiplicand in the first row of the answer. The next digit in the multiplicand is 0. We do nothing, but when we multiply by the third digit which is 1, we move two digits to the left instead of 1 digit. When we copy the multiplier, we discard all the digits beyond the 8th digit. We then add each column to get the answer.

\[
\begin{array}{ccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 \\
1 & 0 & 1 \\
0 & 1 \\
1 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
\]

Since 0000 1111 is 15 in binary, the 2’s complement arithmetic works for multiplication too!

E6) i) T ii) F iii) T iv) T v) T

E7) 1) Move data from memory location A to temporary register 1.
    2) Move data from memory location B to AC.
    3) Move data from AC to location A.
    4) Move data from temporary register 1 to memory location B.

E8) i) A machine, which can be used for variety of applications and is not modelled only for specific applications. Von Neumann machines are general-purpose machines since they can be programmed for any general application, while a microprocessor based control systems are not general-purpose machines as they are specifically modelled as control systems.

ii) Low cost; increased operating speed; reduction in size of the computers; reduction in power and cooling requirements; and more reliable performance.

Photo Credits.
Eniac Computer from
http://www.library.upenn.edu/exhibits/rbm/mauchly/jwm0-1.html
UNIT 17  INTRODUCTION TO PROGRAMMING

Structure  Page No.
17.1 Introduction  27
    Objectives  
17.2 Design of Programs  27
17.3 Compiling C Programs  31
17.4 Summary  36
17.5 Solutions/Answers  37

17.1  INTRODUCTION

In this Unit, we introduce you to the concept of algorithms and explain the principles of program design in Sec. 17.2. In Sec. 17.3, we will explain how to compile C Programs. We will also explain how to use Dev-C++ and Anjuta for compiling C programs.

Objectives

After studying this unit, you should be able to

• explain the concept of algorithm;
• make flow charts for simple programs;
• compile C programs from the command line; and
• use Anjuta and Dev-C++ for writing C programs.

17.2  DESIGN OF PROGRAMS

You are already familiar with problem solving in Mathematics and you are probably familiar with the steps involved. Even if you do not consciously follow certain prescribed steps, you will carry out the following steps:

1. Understand the problem. This will involve asking the questions like:
   • What are the conditions?
   • What is to be proved or found?
   • Can the problem be solved meaningfully with the conditions given?
2. Try to find some examples of the problem.
3. Try to think of a similar problem that you have solved before.

All these considerations are equally valid in the design of computer programs. The steps in designing a program to solve a particular problem are

1. Understand the problem.
2. Work out the steps involved in solving the problem.
3. Write a program that carries out the steps involved in solving the problem.
4. Check if the program works correctly by trying out the program for those cases that can be verified with manual computation.

The steps 2 and 3 in solving a mathematics problem gets included in step 2 of program design. Also, the steps worked out in step 2 has to be in a specific form. It should be in the form of an algorithm.

Loosely speaking, an algorithm is a step by step procedure for accomplishing certain task. The task could be anything, cooking some dish, calculating income tax, etc.

To understand the concept of algorithm, let us now take an example of a task and write an algorithm for it. Suppose we want to factorise a quadratic polynomial $x^2 - ax + b$ with integer coefficients over integers. Let us assume that $a$ and $b$ are positive for simplicity. How do we do it if we do it by hand? We look for two numbers $p$ and $q$ such that $p + q = a$ and $pq = b$. Another way of achieving the same thing is to look at divisors of $b$. If, for some divisor $d$ of $b$, $d + \frac{b}{d} = a$, $d$ and $\frac{b}{d}$ are the roots of the equation. So, $(x - d)(x - \frac{b}{d}) = x^2 - ax + b$ and we are done. Otherwise, we cannot factorise the polynomial over the integers.

Let us now write this as a step by step procedure:

1. $d \leftarrow 1$. (Take $d = 1$.)

2. [Look for divisors of $b$.] Check whether $d \mid b$. If $d \mid b$, go to Step 3.

   If $d \not\mid b$, check if $d + 1 < b$. If $d + 1 < b$, increase the value of $d$ by 1 and carry out Step 2 for this new value of $d$. If $d + 1 \geq b$, stop; we have already checked all the divisors of $b$ and so we cannot factorise the polynomial over the integers.

3. [Finished?] If $d \mid b$, check if $d + \frac{b}{d} = a$. If yes, the factors are $(x - d)$ and $(x - \frac{b}{d})$. Otherwise $d \leftarrow d + 1$(Increase the value of $d$ by 1.) and go to Step 2.

Let us see how to carry this out for the polynomial $x^2 - 8x + 15$ (See Fig. 1). So, do we have an algorithm? Before we can say that, we have to check whether our procedure has certain characteristics. They are:

**Finiteness** The algorithm has to terminate after a finite number of steps. Our procedure satisfies this because our algorithm will certainly stop after checking all the divisors of $b$, which are finite in number; it may stop earlier if it finds a divisor which is a root of the polynomial.

**Definiteness** The steps involved must be precisely defined. There should be no ambiguity in the meaning. This is so in our procedure.
An algorithm has zero or more inputs; the information we give to the algorithm at the start. In our procedure, the coefficients \( a \) and \( b \) of the polynomial are the input.

**Output**  
An algorithm has one or more outputs. In our procedure, the output could be the factors or the information that we cannot factorise the polynomial over integers.

**Effectiveness**  
This means that the operations performed must be simple enough that anybody can perform them using a paper and pencil in a finite amount of time. As you can see, the operations in our procedure can be performed with pencil and paper.

In the case of computer algorithm, effectiveness means that we should be able to carry out the steps in the procedure on a computer.

Try the following exercises to check your understanding.

E1) Make a chart of the steps involved in factoring the polynomial \( x^2 - 7x + 12 \).

The steps of an algorithm can also be shown using a **flow chart**. Let us look at the flow chart corresponding to the algorithm for factoring a polynomial (See Fig. 2). Note that,

![Flow chart for the polynomial factoring algorithm.](image-url)
in the flow chart, there are different kinds of boxes.

**Parallelograms** These are used for input and output operations.

**Rectangles** These are used to show any processing, arithmetic, or storage operation.

**Diamond** Diamond boxes are used to show questions asked or conditions tested, based on whose answers appropriate exits are taken by the procedure. The exit from the diamond shaped box are labelled with answers to the questions.

**Rounded Rectangles** These are used to show beginning or end point of the procedure.

Let us now look at one more example. We will discuss a simple algorithm to test whether a given natural number is prime or not. We can do this by checking whether any of the natural numbers less than $n$ divide $n$. We can actually do a little better. It is enough to check that the number has no proper divisor less than $\sqrt{n}$. Why is this so? If $n$ is composite, $n = pd$, where $p$ is the smallest prime divisor of $n$. Since $d \neq 1$ (because $n$ is not a prime), it has a prime divisor which is bigger than $p$ by the choice of $p$. So, $n = pd \geq p^2$ or $p \leq \sqrt{n}$. So, if $n$ is composite, it has a divisor not bigger than $\sqrt{n}$. Let us write the steps in the algorithm first.

1. $d ← 2$

2. [Check whether $d$ is smaller than square root of $n$.] Check if $d \leq \sqrt{n}$. If ‘Yes’, go to Step 3. If ‘No’, $n$ is a prime; stop.

3. [Check if $d \mid n$.] If $d \mid n$, $n$ is not a prime; stop. If $d \nmid n$, $d ← d + 1$ and go to step 2.

The flowchart for this algorithm is given in Fig. 3. The examples that we have given are simple ones because we wanted to explain the concept of an algorithm. What happens if the procedure is complicated? In this case the procedure is broken into small manageable parts or modules. We write program instructions for the each of the modules, keeping in mind the role these modules have to play in solving the main problem.

Let us look at an example now. Suppose we want to write a program to solve a system of $n$ simultaneous linear equations in $n$ unknowns using Gauss elimination method. In case you have not studied this before, here is a quick introduction. You can look at Unit 5, Block 2 of the course MTE-10 or any other book in numerical analysis for details. This method is a generalisation of the familiar method of eliminating an unknown between a pair of simultaneous equations. In this we use row operations to reduce the system of equations to triangular form. There are three tasks to be carried out in this method. They are

1. Carrying out row operations.

2. After carrying row operation on the the first $i − 1$ rows, check if there is a row in which the $i$th variable has a non-zero coefficient. If there is no such equation, stop the procedure saying that the system does not have a unique solution.

3. After successfully carrying out the procedure $n − 1$ times check if the coefficient of the $n$th variable is non-zero. If it is, find the solution to the equations. Otherwise, stop with an error message.
When we write the program, we write functions to carry out these tasks. Of course for a complete program, we also need a function that takes the input and another one for giving the output.

In the next section, we will see how to compile C programs.

17.3 COMPILING C PROGRAMS

In this section, we will see how to use the GCC compiler which is free and available for many operating systems. The simplest way to compile a C program using GCC is from the command line. In windows, you can open a command line terminal by clicking on Start ⇒ run and typing ‘command’ in the dialogue box that comes up and pressing the Enter key. See Fig. 4 on the next page.

In most versions of linux you can click the right mouse button on the desktop to get the menu shown in Fig. 5 on the following page. By clicking the left mouse button on the ‘Open Terminal’ in this menu, you can open a linux command line window, also called a terminal in linux.

Your instructor will help you to open a command line terminal.

To compile the first program unit1-prog1.c you have to type

gcc unit1-prog1.c -o unit1-prog1

GCC stands for GNU Compiler Collection.
in the command line and press enter. You will have to type the full path of gcc if it is not on your path in the case of windows. The part `-o unit1-prog1` specifies the name of the output file. Without this, in linux, a file called a.out is created.

![Fig. 4: Opening a command line in windows.](image)

![Fig. 5: Opening a command line window in linux.](image)

You can execute this file by typing `.\a.out` in the command line. If you compile another program without mentioning the output file name, it will overwrite the file created earlier. If you add the part `\-o unit1-prog1`, an executable file with the name unit1-prog1 will be created in the case of linux. You can execute it by typing `.\unit1-prog1`. In windows, a file with name `unit1-prog1.exe` will be created. You can run this by typing the file name `unit1-prog1` and pressing the `Enter` key.

You can add some more options. One of them is `\-Wall`. This asks GCC to print all the warnings. In the initial stages, this option will be helpful in finding the mistakes in your program. In case you want to use the maths library you have to add `\-lm` to the command line. If you want the GCC compiler to use only features specified in ANSI specification, you can add the option `\-ansi` to the command line. Using this option will ensure that your program will compile correctly in any ANSI compliant compiler. For example, if you want to compile the program `unit1-quad.c` which uses the square root function `sqrt()` and the absolute value function `fabs()`, the command is

```
gcc \-Wall \-lm \-ansi unit1-quad.c \-o unit1-quad
```

If you are writing a program, this procedure of compiling from the command line is very tedious. It is more convenient to use an IDE(Integrated Development Environment) for writing programs. IDEs provide a convenient means of writing, compiling and debugging programs. We will discuss two IDEs, one for the linux and one for the windows. For linux, we recommend Anjuta, which provides a very user friendly way of writing and compiling C programs. You can start Anjuta by clicking on
the bottom left corner of the desktop and selecting Anjuta under ‘development’. A window similar to the one in Fig. 6 opens.

![Fig. 6: Anjuta start up window](image)

If you want to open an already existing file, click on ‘Open file’, go to the directory containing your file and open it. If you want to write a new program, click on ‘New file’. A new file opens for you to type your program.

![Fig. 7: Anjuta Window with a program.](image)

After you finish writing the program, press $\text{F9}$ If the program compiles successfully, press $\text{F11}$ to link. If there are any errors in your program, there will be an error message in the message window. You can double click on the message to go to the line containing the error. After you successfully compile your program, you can then press $\text{F3}$ to run the program. The output is similar to the one in Fig. 8 on the following page. To set options of the compiler options click on Settings ⇒ Compiler and
linker settings ...

![Fig. 8: Output window.](image)

In the dialogue that opens, click on the ‘Warnings’ tab and then double click on ‘Wall’ to enable all warnings. See Fig. 9.

![Fig. 9: Setting warnings.](image)

To link the maths library, click on the ‘Libraries’ tab and enter ‘lm’ on the space given. Click ‘+ Add’ afterwards. See Fig. 10 on the facing page.

In windows, you can start Dev-C++ by pressing **START** ⇒ **Programs** → **Blood shed Dev-C++** → **Dev-C++**. The window is similar to Fig. 11 on the next page. To compile and run a file, press **F9**. If you just want to compile, but not run, press **Ctrl** + **F9**. To just run press **Ctrl** + **F10**.
Fig. 10: Linking maths library.

To set the command line options, press Tools ⇒ Compiler Options and enter the options as shown in Fig. 12 on the following page.

Note that, in the case of Dev-C++, you have to add the line `getchar();` before the `return` statement; otherwise the output window will close immediately after running the program and you will not be able to see the output. Also, if your program has a `scanf()` statement, get rid of the trailing `<CR>` you typed after the input using a `getchar()`. See page 48 of Block 2 for the reasons for doing so.

We will now briefly discuss how to programs distributed over multiple files. Let us see
how to compile the 3 files given in exercise 6 of Block 2. The names of the files are unit8-prog9-f1.c, unit8-prog9-f2.c and unit8-prog9-f3.c.

In this case we first create object files using the **-c** option:

```
gcc -c unit8-prog9-f1.c unit8-prog9-f2.c unit8-prog9-f3.c
```

To create an executable file called unit8-prog9, give the command

```
gcc unit8-prog9-f1.o unit8-prog9-f2.o unit8-prog9-f3.o -o unit8-prog9
```

The command is shown in two lines, because the line is too long. You have to type it in one line. Ignore any warnings printed by the compiler. Now, you can run the program by typing ./unit8-prog9. For more complicated projects, you will have to create make files. See page 532 of the book *A Book on C, fourth edition*, by A. Kelley and I. Pohl to learn about make files. This book will be available in your study centre library.

In the next session, we will summarise this unit.

### 17.4 SUMMARY

In this unit we have discussed

1. the concept of algorithm;
2. how to make flow charts for simple programs;
3. how to compile C programs from the command line; and
4. how to use Anjuta and Dev-C++ for writing C programs.

17.5 SOLUTIONS/ANSWERS

E1) See Fig. 13

\[ d = 1 \]
Go to Step 2.

\[ 1 \mid 12 \]
Go to Step 3.

\[ 1 + \frac{12}{1} = 13 \neq 7 \]
Take \( d = 2 < 12 \).
Go to Step 2.

\[ 3 \mid 12 \]
Go to Step 3.

\[ 2 + \frac{12}{2} \neq 7 \]
Take \( d = 3 \).
Go to Step 2.

\[ 2 \mid 12 \]
Go to Step 3.

\[ 3 + \frac{12}{3} = 7 \]
The factorisation is \((x - 3)(x - 4)\)

Fig. 13: Steps involved for \(x^2 - 7x + 12\)
UNIT 18 LIST OF PRACTICAL SESSIONS

In this unit, we list the practical sessions for this course. While it is compulsory to complete those sessions which are marked compulsory, we encourage you to do as many sessions as possible. Apart from the practical sessions, we advise you to compile and run the examples in the Block to get more practice. While we have checked all the programs in the blocks for errors, it is quite possible that some of them still have errors in them. If you find any, please send a mail to the course coordinator pointing the mistake.

For the programs based on numerical analysis, you can refer to the blocks of MTE-10, Numerical Analysis, which is a part of the IGNOU Bachelor’s degree program. The blocks of this course will be available at your program study centre.

Add comments in the program and in functions, documenting what the program or function does. Indent your programs properly so that anyone can read them easily and understand them. If you are working on Linux, the indent program may be available on your computer. For example, if you want to indent your program in the style followed in the book by Kernighan and Ritchie, you can invoke indent using the command

```
indent -kr file_name.c
```

Indent offers many other options. Read the documentation for the program if necessary.

Also, write a friendly interface, wherever required, for the user to interact with the program. Your program should also check if the input is appropriate and print an error if it is not.

Session 1 (Compulsory)

The aim of this session and the next session is to familiarise you with the process of compiling and running C programs. The example programs for the first three sessions are from the first Block, so take it to the study center with you when you go for practicals.

Program 1

1. Compile and run the program unit1-prog1.c given in page 9 of Block 1 from the command line.

2. Open the file unit1-prog1.c in your IDE and compile and run the file. (You have to add the line `getchar();` before the return statement if you are using Dev-C++.)

3. Change the program so that it prints

   **This is very easy!!**

Program 2

1. Compile and run the program unit1-prog8.c in page 11 of Block 1.

2. Change the values of x and y to x = 12, y = 13 and x = 11, y = 14 and get the output.

3. Try giving decimal values x = 1.2, y = 4.1 and x = 0.2 and y = 5 and study the output. How far is your answer correct?
1. Calculate the integral \( \int_{0}^{1} \frac{1}{1 + x^2} \, dx \) by actual integration.

2. Compile and run the program unit1-simpsonf.c in page 17 of Block 1 and compare the values you get.

3. Change the function to a) \( x^2 \) b) \( e^x \). (See the list of functions given in Table 1, page 16, Block 2.) Again, compare the values given by the program with the values you get by actual integration. Note down the difference.

**Session 2 (Compulsory)**

**Program 1**

Insert `printf()` statements in unit3-prog1.c in page 53, Block 1, compile it and run it to check your answer.

**Program 2**

Compile and run the file unit3-prog2.c in page 54 of Block 2 to check your answer for the Exercise E3) of Unit 3.

**Program 3**

Complete the program unit3-prog0.c in page 71 of Block 1, which is given as the solution to exercise E1) of Unit 3, by giving values for the remaining 3 cases. Compile and run the program and check the output.

**Program 4**

In unit3-prog0.c, instead of including the values in the program itself, modify it so that it reads the values of a and b using `scanf()`. Compile and run the program. Give values for a and b covering the 4 cases and check your output.

**Program 5**

Create a program that prompts the user for basic pay and calculates and prints basic + HRA(30% of basic)+DA(46% of basic).

**Session 3 (Compulsory)**

**Program 1**

Write a program similar to unit2-ex7ans.c in page 45 of Block 1 that checks for overflow in multiplication when the two operands are `unsigned long` and when they are `long double`. The program should also print the overflow when it occurs.

**Program 2**

Write a program that prompts for a natural number n and prints the nth Fibonacci number. It should also print an error message if the Fibonacci number is bigger than `ULONG_MAX`. 
Program 3

Write a program that finds all primes less than ULONG_MAX and saves it in a file called primes. You can modify any of the programs we have given in Block 2 for finding primes.

Session 4

Write a C function to determine the number of days between two dates passed to it. (Hint: Find the number of days, say n, between the first date and January 1, 1900; find the number of days, say m, between the second date and January 1, 1900. Your program should return the absolute value of the differences n − m.)

Session 5 (Compulsory)

Program 1

Write a function that evaluates the polynomial by Horner’s method. Horner’s method is as follows (See MTE-10 material if necessary.): If
\[ f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \]
the Horner’s method is given by the equation
\[ f(x) = ((\cdots ((a_n x + a_{n-1}) x + a_{n-2}) x + \cdots + a_1) x + a_0)\]
It should prompt for the coefficients of the polynomial, store the values in an array, prompt the value for which the polynomial is to be evaluated and evaluate and print the value. Your program should be able to evaluate any polynomial of degree less than 20.

Program 2

If \( f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \), define
\[ f_o = \sum_{i \text{ odd}}^{0 \leq i \leq n} a_i x^i, f_e(x) = \sum_{i \text{ even}}^{0 \leq i \leq n} a_i x^i \]
Write functions similar to Horner’s method for evaluating \( f_o(x) \) and \( f_e(x) \). Note that \( f(a) = f_e(a) + f_o(a) \) and \( f(-a) = f_e(a) - f_o(a) \). It is easy to show that, if the polynomial \( f(x) \) has integer coefficients and has a rational root \( \frac{p}{q} \), then \( p \mid a_0 \) and \( q \mid a_n \). In particular, it has an integer root \( a \), then \( a \mid a_0 \). Using this fact and the functions for finding \( f_o(a) \) and \( f_e(a) \) for any \( a \), write a function for checking whether a polynomial \( f(x) \) has integer roots by finding the value of \( f(\pm a) \) for all divisors \( a \) of \( a_0 \).

Program 3

Write a program that computes the value of \( \tan^{-1} x \) using the power series expansion
\[ x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots \]
Compare the values obtained for \( \frac{1}{2} \) and \( \frac{1}{\sqrt{3}} \) with the values obtained from a table or a scientific calculator. Write a program to find the value of \( \pi \) using the relation
\[ \pi = 16 \tan^{-1} \left( \frac{1}{5} \right) - 4 \tan^{-1} \left( \frac{1}{239} \right) \]

Session 6

Write a program that uses Simpson’s one third rule to evaluate definite integrals of function for arbitrary number of intervals. Your program should prompt for the upper
and lower limits of integration and the number of subintervals and evaluate the integral. It should check if the number of intervals is even. Use your program to evaluate the integrals \( \int_{0}^{\pi/2} \sqrt{\sin x} \, dx \) and \( \int_{0}^{1/3} \frac{1}{\sqrt{1-x^2}} \, dx \) for the number of intervals \( n = 10, n = 50, n = 100 \). Compare the values with trapezoidal rule with the values you get from the Simpson’s rule.

**Session 7 (Compulsory)**

**Program 1**

Write a function that solves equations by bisection method. Use it to find a solution of

a) \( x^2 - 5x - 24 = 0 \) \hspace{1cm} b) \( x \log_{10} = 1.2 \)

**Program 2**

Write a program that solves algebraic equations by false position (regula falsi) method. Use it to solve the equations in Program 1 above.

**Session 8 (Compulsory)**

Write a function that solves an intermediate value problem by Runge-Kutta method of \( O(h^4) \) for different values of \( h \). Your program should prompt for the value of \( h \), find the number of iterations required and solve the problem. Use your program to solve the following problems:

1. \( y' = t + y, \ y(0) = 1 \) for \( t = 0.5 \) with \( h = 0.1, h = 0.05 \) and \( h = 0.01 \).
2. \( y' = 2y/t, \ y(1) = 2 \) for \( t = 2 \) with \( h = 0.1, h = 0.05 \) and \( h = 0.01 \).

**Session 9 (Compulsory)**

**Program 1**

In this program you will create a menu using `switch()` statement. Write a currency converter program that converts rupee to dollar, yen or euro or converts any of these currencies to rupee. When the program is run it should print a menu as follows:

**What do you want to do?**

1. Convert rupee to other currencies.
2. Convert other currencies to rupee.

Press 1 or 2 to make your choice. Press any other key to exit.

If the user presses 1, for example, it should display the following menu:

**Convert rupee to**

1. Dollar
2. Yen
3. Euro
Press 1, 2 or 3 to make your choice. Press u to go to the previous menu.

After the user presses 1, 2 or 3, the program should scan the amount in rupee, make the conversion and print the value. It should once again show the first menu. If the user presses any key other than 1, 2 or 3, it should print an error message and ask the user to make a correct choice and print the menu again. The user should be able to exit from the program by pressing any key other than 1 or 2 in the first menu.

Program 2

Consider the problem of estimating the number of real and imaginary roots of a polynomial equation with real coefficients.

\[ f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \]  \hspace{1cm} (1)

We say that a polynomial has a change of sign at a particular term if the sign of that term is different from the next higher degree term. We say that a polynomial has continuation at a particular term if the sign of the term is the same as the next higher degree term. The equation is complete if no coefficient is zero. If \( a_r = 0 \), we say that \( r \)th term is missing. For example, in the polynomial \( 5x^6 - 3x^4 - 4x^3 + 5x^2 - 3x + 1 = 0 \) has changes of sign at the terms \(-3x^4\), \(5x^2\), \(-3x\) and \(1\). It has a continuation at \(-4x^3\). The \( x^5 \) term is missing.

Recall the Descartes Rule of Signs for finding the number of positive and negative roots of a polynomial with real coefficients.

**Theorem 1 (Descartes' rule of signs)** The equation \( f(x) = 0 \) cannot have more positive roots than \( f(x) \) has changes of sign or more negative roots than \( f(-x) \) has changes of sign.

Write a C program that prompts for the coefficients of a polynomial and prints the maximum possible number of real roots. Use it to find the maximum number of real roots of the polynomials.

**Session 10(Compulsory)**

Program 1

In this program you will write function that uses a more efficient method of powering an element in a group. The basic idea behind the algorithm is the fact if \( n = \sum_{i=0}^{k} a_i 2^i \) where each \( a_i \) is 0 or 1, \( a^n = \prod_{a_i=1} a^n2^i \). Here is the algorithm (Source: A Course in Computational Algebraic Number Theory by Henri Cohen, GTM 138, published by Springer):

**Algorithm (Binary Powering):** Given \( a \in G \) and \( n \in \mathbb{Z} \), this algorithm computes \( a^n \) in \( G \). We write 1 for the identity element in \( G \). In the algorithm we write \( \lfloor x \rfloor \) for the largest integer smaller than \( x \). For example \( \lfloor 3.2 \rfloor = 3 \)

1. [Initialise] Set \( y \leftarrow 1 \). If \( n = 0 \) output \( y \) and terminate. If \( n < 0 \), let \( N \leftarrow -n \) and \( z = g^{-1} \). Otherwise, set \( N \leftarrow n \) and \( z \leftarrow g \).

2. [Multiply?] If \( N \) is odd set \( y \leftarrow z \cdot y \).

3. [Halve N] Set \( N \leftarrow \lfloor N/2 \rfloor \). If \( N = 0 \), output \( y \) as the answer and terminate the algorithm. Otherwise, set \( z \leftarrow z \cdot z \) and go to step 2.
Work through the algorithm for $a = 3$ and $n = 5$ manually to understand the algorithm. Then, write a function that implements the algorithm for the group $\left( \mathbb{Z}/p\mathbb{Z} \right)^*$, $p$ a prime, for positive values of $n$ only.

Program 2

Write a program that finds $\text{ord}(a)$, the order of $a \in \mathbb{Z}/p\mathbb{Z}$ for a prime $p$. The program should first factorise $p-1$ using the already computed list of primes in Session 3, say $p-1 = p_1^{n_1}p_2^{n_2} \cdots p_k^{n_k}$. It should find $\text{ord}(a)$ by finding the power of each $p_i$ dividing $\text{ord}(a)$. For example, to find the order of $p_1$ dividing $\text{ord}(a)$ the program should proceed as follows: First, find $b = a^{\frac{p-1}{p_1}}$. Then, find $m_1$ such that $b^{p_1^{m_1}} = 1$ and $b^{p_1^{m_1}-1} \neq 1$. The power of $p_1$ dividing $\text{ord}(a)$ is $p_1^{m_1}$.

Use your program to find the order of 768462011 in $\mathbb{Z}/p\mathbb{Z}$ for $p = 4264967269$. Use the Binary Powering algorithm discussed in Program 1 of this session for taking powers.

Session 11(Compulsory)

Program 1

Write a program for linear regression. Given the values $x_i, y_i = f(x_i), 1 \leq i \leq n$, you have to fit a straight line $Y = aX + b$ for the data. The values of $a$ and $b$ are given by

$$
\begin{align*}
    b &= \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}, \\
    a &= \frac{\sum y_i}{n} - b \frac{\sum x_i}{n}
\end{align*}
$$

Write a program that reads value $n$, $x_i$s and $y_i$s and outputs $a$ and $b$. Use the program to fit a line for the following data.

<table>
<thead>
<tr>
<th>x</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Program 2

Write a program to fit a transcendental curve of the form $Y = ax^b$ for a given set of data. We have

$$
\begin{align*}
    b &= \frac{n \sum \ln x_i \ln y_i - \sum \ln x_i \sum \ln y_i}{n \sum (\ln x_i)^2 - (\sum \ln x_i)^2}, \\
    a &= \text{exp} \left( \frac{\sum \ln y_i}{n} - b \frac{\sum \ln x_i}{n} \right)
\end{align*}
$$

and $a = \text{exp} R$. Use these formulae to fit a curve of the form $y = ax^b$ for the date given below:

<table>
<thead>
<tr>
<th>x</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0.5</td>
<td>2</td>
<td>4.5</td>
<td>8</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Session 12(Compulsory)

In this session, you will create a $n \times n$ array of float using calloc(). Here are the functions for creating and destroying a $n \times n$ square matrix of float.

```c
float **create_square_matrix(int n) {
    float **matrix = calloc(n, sizeof(float *));
    int i;
    if (!matrix) {
        fprintf(stderr, "fatal error\ninsufficient memory\n");
        fprintf(stderr, "from create_matrix\n");
        exit(1);
    }
```
```c
for (i = 0; i < n; i++) {
    matrix[i] = calloc(n, sizeof(float));
    if (!matrix[i]) {
        fprintf(stderr, "fatal error\ninsufficient memory\n");
        fprintf(stderr, "from create_matrix\n");
        exit(1);
    }
}
return matrix;
}

void delete_square_matrix(int n, float **matrix)
{
    int i;
    for (i = 0; i < n; i++)
        free(matrix[i]);
    free(matrix);
}

Use the matrix for creating matrix to create a 3 × 3 matrix of floats and set
m[i][j]=(i+j)/2.0. Print the matrix, then call the delete matrix function and exit
the program. Write a program that solves linear equations by partial pivoting. Note that
you can interchange rows by using a pointer called temp defined as float *temp.
Instead interchanging element by element, you can just swap pointers using temp. Use
your program to solve the set of equations

0.232x_1 + 0.457x_2 + x_3 = 0.248
0.221x_1 + 0.129x_2 + 0.126x_3 = 0.124
0.127x_1 + 0.328x_2 + 0.192x_3 = 2.341

Session 13

Write a program for finding the inverse of a matrix using Gauss-Jordan method. Use
arrays created using calloc() as in the previous session. Use your program to find the
inverse of the matrix

\[
\begin{pmatrix}
2 & 5 & 4 \\
5 & 8 & 5 \\
4 & 5 & 4
\end{pmatrix}
\]

Session 14(Compulsory)

Define a struct for working with natural numbers by
typedef struct RAT{
    long num;
    unsigned long denom;
}rat;

Here, we assume that the denominator is always positive. Write functions for:

1. Reading a rational number and storing it.
2. Removing common factors and reducing a rational number to its smallest form.
3. Adding, multiplying and dividing rational numbers.
4. Printing a rational number.
Session 15

Write a program that maintains a telephone directory in a file. It should use one line for storing each record. Each record should consist of 2 fields, the name field and the telephone number field which are separated by a comma. The program should allow the user to add, delete or display data.

Session 16

Write a C program that administers an objective type test. The questions should be in one file and the correct answers should be in another file. The program should use a struct for storing the question and fseek() to move from question to question in the file. The program should keep a record of questions answered correctly. At the end of the test, it should print the marks as well as the questions that were wrongly answered. (You may find it useful to go through Program 10.12 in page 132 of Block 2).

Session 17

Create a linked list which holds the names of the cities New Delhi, Johannesburg, Rio de Janeiro, Copenhagen and Zurich in the same order, i.e. the head node must contain to New Delhi, the next node must contain Johannesburg and so on. Sort the list in lexicographic order by insertion sort using the function strncmp() for comparison. Afterwards, add the cities Madrid and London in appropriate places. Finally, the program should print the linked list.

Session 18(Compulsory)

Program 1

Write a program that reverses a string using a stack.

Program 2

Write a program that reads an expression from the terminal and checks if the brackets are properly matched.

Session 19(Compulsory)

Implement a queue containing integers as a linked list by implementing functions for creation of the queue, add an element of the queue, remove an element from the queue, to get the element at the top of the queue and the element at the bottom of the queue. Your linked list will have two pointers, one to the top and other to the bottom of the queue.

Session 20(Compulsory)

Create a binary search tree. Insert in the binary tree the words in the sentence ‘All the world is a stage, And all the men and women merely players.’ in the order they occur in the sentence. Then, do an in-order, pre-order and post-order traversals of the trees and list the keys in each node traversed.