WESLEY UNIVERSITY ONDO, NIGERIA

Faculty of Natural and Applied Sciences

Department of Computer Science

Course Code:	CCS 225
Course Title:	Foundation of Sequential Programming
Status:	Core
Semester:	Second
Mode of Delivery:	Classroom Lectures & Practical Sessions

Course Lecturer

N.A Udoh (MNCS)
Contact: +2347060553660
Email: nicholas.udoh@wesleyuni.edu.ng

Course Description

This course introduces the fundamental principles of sequential programming, including structured problem solving, algorithm development, and logical program execution. It emphasizes variables, control structures, functions, and debugging techniques necessary for developing efficient software solutions.

Course Objectives

- Understand sequential program execution
- Develop algorithms
- Use loops and conditionals
- Apply modular programming
- Debug and test programs

Learning Outcomes

- Design structured programs
- Write executable programs
- Apply logical reasoning
- Debug simple applications
- Demonstrate problem-solving skills

Assessment

ASSESSMENT TYPE	SCORE
Group Work Project (GWP)	15%
Collaborative Review (CR)	15%
Examination	70%
Total	100%

CONTENTS **PAGE**

Module 1 **…….1**

Unit 1 Principles of Programming …….....1
Unit 2 Evolution of Programming Languages …….28

Module 2 **…….....50**

Unit 1 Basic Machine Architecture ……50
Unit 2 Data Representation and Storage ……65
Unit 3 Operations on Data ……..84
Unit 4 Machine Instructions ……..... ……102

Module 3 **…….....117**

Unit 1 Block Structured Languages ……..117
Unit 2 Specification and Translation of Programming Languages….. 140

MODULE 1

Unit 1 PRINCIPLES OF PROGRAMMING

Unit 2 EVOLUTION OF PROGRAMMING LANGUAGES

UNIT 1: PRINCIPLES OF PROGRAMMING

CONTENTS

1.0 INTRODUCTION
2.0 OBJECTIVES
3.0 MAIN CONTENT

3.1 WHAT IS PROGRAMMING

3.2 PROGRAM DESIGN AND SPECIFICATION

3.2.1 Program Development Life Cycle

3.3 SEQUENTIAL PROGRAM STRUCTURES

3.3.1 STORAGE

3.3.2 DATA DECLARATION

3.3.3 INPUT AND OUTPUT

3.3.4 OPERATIONS ON DATA

3.3.5 CONTROL

3.3.5.1 The sequence structure

3.3.5.2 Decision Structure or Selection Structure

3.3.5.3 Repetition or Iteration Structure

4.0 CONCLUSION

5.0 SUMMARY

6.0 TUTOR-MARKED ASSIGNMENT

7.0 REFERENCES/FURTHER READINGS

1.0 INTRODUCTION

This unit serves as introduction to computer programming by presenting the fundamental concepts and terminology of programming. Skills in designing and writing simple computer programs are developed, programming concepts and terminology for identification and writing of basic programs using constructs, such as variables and constants are to be learnt.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- Understand what programming is all about
- Know the steps to be followed in designing programs
- Describe sequential programming
- Understand the basic structures in which sequential program components are constructed

3.0 MAIN CONTENT

3.1 WHAT IS PROGRAMMING

Computers don't understand natural language like English they have to be instructed in special computer languages before they can perform any task. The process of developing series of sequence of instructions known as programs to be given to computer so as to perform a specific task is known as programming. Though computer is viewed as a super machine, it cannot solve

any problem that it has not been directed to solve by humans (Programmers). It is better said that the intelligence of computer is derived from the intelligence of man (garbage in garbage out).

A programmer requires some basic facilities and tools to be able to design, develop, test, implement or maintain computer programs. These facilities and tools include text editors, compilers, interpreters, diagnostic tools, etc. Text editors provide the basic means of creating and modifying text files that is open, view, and edit plain text. Examples include Notepad, E-TextEditor, GNU Emacs, EditPlus, gedit, Textpad, UltraEdit, etc. A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. Examples of interpreted languages are Perl, Python and Matlab. Diagnostic tools are for detecting error messages in a programmer's source code that refer to statements or syntax that the compiler cannot understand. These tools for editing, compiling, debugging, etc. programs are sometimes integrated in one graphical user interface. This integrated tool is known as Integrated Development Environment (IDE) and it is used for rapidly developing computer programs. Examples are Visual Studio, NetBeans, JBuilder, Eclipse etc.

3.2 PROGRAM DESIGN AND SPECIFICATION

Program design applies to the development or production of computer programs. A basic knowledge of program design is needed to write programs of reasonable quality. A program specification is usually part of a system specification, which defines the whole system.

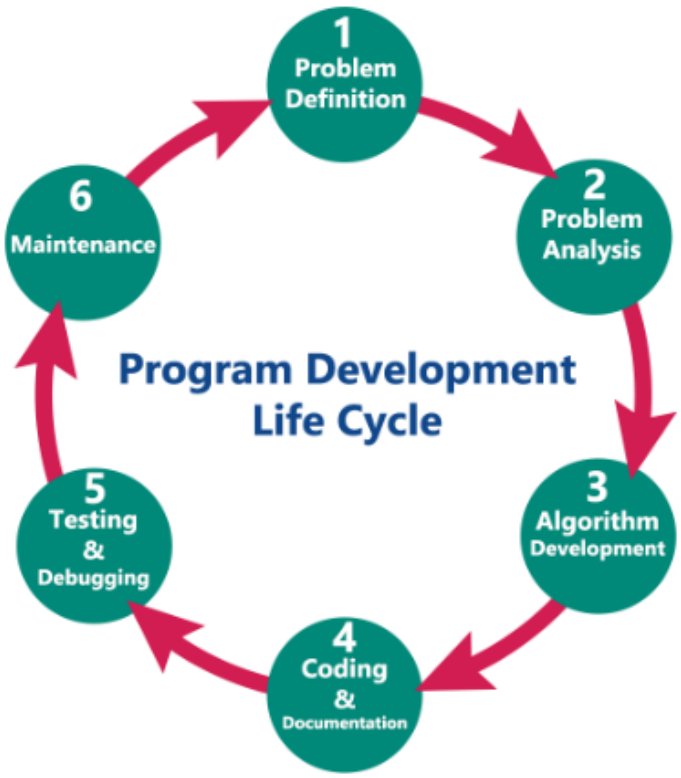
The aims of program design are summarized as follows:

i. Reliability: The program can be depended upon always to do what is supposed to do.
ii. Maintainability: The program will be easy to change or modify when the need arises
iii. Readability: The program will be easy for a programmer to read and understand
iv. Performance: The program causes the tasks to be done quickly and efficiently
v. Storage saving: The program is not allowed to be unnecessarily long to achieve memory efficiency.

3.2.1 Program Development Life Cycle

When a computer program is to be developed using any programming language, a sequence of steps is followed. These steps are called phases in program development. There is need to carefully follow this sequence of steps to successfully develop correct computer programs. The process associated with this computer program development is called program development life cycle (PDLC). The program development life cycle is a set of steps or phases that are used to develop a program in any programming language. PDLC is a systematic way of developing quality software. It provides an organized plan for breaking down the task of program development into manageable chunks, each of which must be successfully completed before moving on to the next phase. The program development process is divided into the steps discussed below:

This cycle is divided into the following six (6) steps.



1. Problem Definition

   The first step is to define the problem. In major software projects, this is a job for system analyst, who provides the results of their work to programmers in the form of a *program specification*. The program specification defines the data used in program, the processing

that should take place while finding a solution, the format of the output and the user interface.

In this phase, the problem statement is defined and the boundaries of the problem are decided. In this phase there is need to understand the problem statement, what is the requirement, and what should be the output of the problem solution. These are defined in this first phase of the program development life cycle. The developers must obtain the program requirements from the users and document the requirements. Typically, a standard form is used to develop the requirements.

A programmer is usually given specification of what proposed program is required to do. The programmer must then design and implement the program so that it meets the specification. A program that meets its specification is said to be correct. An important factor in determining program specification is to produce requirements specification. A system analyst will discuss the requirements specification with the users. A *requirement* is simply a statement of what the system must do or what characteristics it needs to have. During a systems development project, requirements will be created that describe what the business needs (*business requirements*); what the users need to do (*user requirements*); what the software should do (*functional requirements*); characteristics the system should have (*nonfunctional requirements*); and how the system should be built (*system requirements*). Although this list of requirement.

2. Problem Analysis

   During analysis, a programmer review specifications to fully understand what the software should do. The analysis of the problem to be solved involves having the basic understanding of the problem, identification and designing of inputs and outputs and identification of any suitable solution model. The requirements like variables, functions, etc. to solve the problem are determined in this phase. That means the required resources to solve the problem defined in the problem definition phase are gathered in this phase.

3. Algorithm Development

   During this phase, a step by step procedure to solve the problem using the specification given in the previous phase is developed. This phase is very important for program

development. That means we write the solution in step by step statements. Program design starts by focusing on the main goal that the program is trying to achieve and then breaking the program into manageable components, each of which contributes to this goal. This approach of program design is called *top-bottom program design* or *modular programming*. The first step involve identifying *main routine*, which is the one of program's major activity. From that point, programmers try to divide the various components of the main routine into smaller parts called *modules*. For each module, programmer draws a conceptual plan using an appropriate program design tool to visualize how the module will do its assign job.

The various program design tools are described below:

i. **Algorithms**

An algorithm is a step-by-step description of how to arrive at a solution in the easiest way. Algorithms are not restricted to computer world only. In fact, we use them in everyday life.

ii. **Flowcharts**

A flowchart is a diagram that shows the logic of the program. For example:

iii. **Decision tables**

A Decision table is a special kind of table, which is divided into four parts by a pair of horizontal and vertical lines.

iv. **Pseudo-code**

A pseudo-code is another tool to describe the way to arrive at a solution. They are different from algorithm by the fact that they are expressed in program language like constructs.

4. Coding and Documentation

This phase uses a programming language to write or implement the actual programming instructions for the steps defined in the previous phase. In this phase, we construct the actual program. That means we write the program to solve the given problem using programming languages like C, C++, Java, etc., Coding the program means translating an algorithm into specific programming language. The technique of programming using only well-defined control structures is known as Structured programming. Programmer

must follow the language rules, violation of any rule causes error. These errors must be eliminated before going to the next step.

5. Testing and Debugging

   After removal of syntax errors, the program will execute. However, the output of the program may not be correct. This is because of logical error in the program. A logical error is a mistake that the programmer made while designing the solution to a problem. So the programmer must find and correct logical errors by carefully examining the program output using *Test data*. Syntax error and Logical error are collectively known as *Bugs*. The process of identifying errors and eliminating them is known as *Debugging*. During this phase, there is need to check whether the code written in the previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test whether it is providing the desired output or not.

6. Maintenance

After testing, the software project is almost complete. The structure charts, pseudo-codes, flowcharts and decision tables developed during the design phase become documentation for others who are associated with the software project. This phase ends by writing a manual that provides an overview of the program's functionality, tutorials for the beginner, in-depth explanations of major program features, reference documentation of all program commands and a thorough description of the error messages generated by the program.

The program is actively used by the users. If there is need for any enhancements, all the phases are to be repeated to make the enhancements. That means in this phase, the solution (program) is used by the end-user. The program is deployed (installed) at the user's site. Here also, the program is kept under watch till the user gives a green signal to it.
Even after the software is completed, it needs to be maintained and evaluated regularly. In software maintenance, the programming team fixes program errors and updates the software.

3.3 SEQUENTIAL PROGRAM STRUCTURES

Sequential programming is when the algorithm to be solved consists of operations one after the other. A sequential program explicitly waits in-line, for the expected events in various places in the execution path. First cooking dinner, then eating, and then washing the dishes is one sequence. First eating, then washing the dishes, and then cooking is a much less sensible sequence.
The ideas of sequence are the ideas with which almost every introduction to programming begins. Most books compare a program to a recipe or a sequence of instructions, along the lines of:

to go to work:

get dressed
eat breakfast
catch the bus

In sequential composition, different program components execute in sequence on all processors. Based on the number of microprocessors, computers can be classified into Sequential computers and Parallel computers. Any task complete in sequential computers is with one microcomputer only. Most of the computers we see today are sequential computers where in any task is completed sequentially instruction after instruction from the beginning to the end. The parallel computer is relatively fast. New types of computers that use a large number of processors. The processors perform different tasks independently and simultaneously thus improving the speed of execution of complex programs dramatically. Parallel computers match the speed of supercomputers at a fraction of the cost.

Sequential program structures are forms in which program components are constructed, organized and interrelated. In learning a programming language, there is need to learn about two important aspects of the language: its syntax and semantics. The syntax of a language is the grammatical rules that govern the ways in which words, symbols, expressions and statements may be formed and combined. The semantics of a language are the rules that govern its meaning. In the case of computer language, meaning is defined in terms of what happens when the program is executed.

The main features of the computer are still visible in features of the programming language. These features are storage, input and output, operation on data, and control.

3.3.1 STORAGE

Computers require a set of instructions and data to be stored in their memory to perform a specific task. This stored program concept was originated ever since the invention of Charles Babbage's difference engine in 1822. In programming languages, data are identified by name rather than by their location addresses in main storage. The names that associate stored data values are called identifiers because an identifier is the name by which the data value may be identified. An identifier is a constant if it is always associated with the same data value and it is a variable if its associated data value is allowed to change. Changing a variable's value implies changing what is stored.

When using names in programs, care must be taken to specify whether the names are literals or identifiers. When names or letters are used literally they are called literals, and they are distinguished from identifiers by placing them within quotation marks. So the instruction PRINT "N" means print the letter N, and the instruction PRINT N means print the value associated with N.

3.3.2 DATA DECLARATION

A variable is a symbolic name assigned to a data item by the programmer. At any particular time, a variable will stand for one particular data, called the value of a variable, which may change from time to time during a computing process. The value of a variable may change many times during the execution of a program. A variable is usually given a name by the programmer. The variable must be declared that is to state its data type and its Value.

A data type is a classification of data, which can store a specific type of information. Data types are primarily used in computer programming in which variables are created to store data. There are a number of traditional data types found in most languages. The act of defining a variable is called data declaration. Declarations provide information about the name and type of data objects needed during program execution. Every language supports a set of primitive data types.

Usually these include integer, real, Boolean, and character or string. A language standard determines the minimum set of primitive types that the language compiler must implement. There are two types of declaration, implicit declaration and explicit declaration. Implicit declaration or default declaration are those declaration which is done by compiler when no explicit declaration or user defined declaration is mentioned. For example in 'Perl' compiler implicitly understand that:

$abc ='astring' is a string variable and

$abc=7; is an integer variable.

In explicit declaration, user explicitly defined the variable type. For example:

Float A, B;

In this example it specifies that it is of float type variable which has name A & B.

Purpose of Declarations:

i. Choice of storage representation: Translator determine the best storage representation of data types that is why it needs to know primarily the information of data type and attribute of a data object.
ii. Storage Management: It helps the computer to make best use of memory for data object by providing its information so that computer can allocate the optimum size of memory for the data.

A variable name does not have an associated value until it has been assigned one. In computer programming, initialisation is the assignment of an initial value for a data object or variable. The manner in which initialisation is performed depends on programming language, as well as type, storage class, etc., of an object to be initialized.

3.3.3 INPUT AND OUTPUT

Programming languages have special functions for dealing with input and output. Common names for these functions are input, read, get, accept, output, write, print, put and display. This is illustrated using the following simple Java program:

ComputeArea.java

```
1 public class ComputeArea {
2 public static void main (String[] args) {
3 double radius; // Declare radius
4 double area; // Declare area
5
6 // Assign a radius
7     radius = 20; // New value is radius
8
9 // Compute area
10     area = radius * radius * 3.14159;
11
12 // Display results
13     System.out.println("The area for the circle of radius " +
14       radius + " is " + area);
15   }
16 }
```

Variables such as radius and area correspond to memory locations. Every variable has a name, a type, a size, and a value. Line 3 declares that radius can store a double value. The value is not defined until a value is assigned. Line 7 assigns 20 into radius. Similarly, line 4 declares variable area, and line 10 assigns a value into area.

Java uses System.out to refer to the standard output device and System.in to the standard input device. By default the output device is the display monitor, and the input device is the keyboard. To perform console output, you simply use the println method to display a primitive value or a string to the console.

3.3.4 OPERATIONS ON DATA

The operation that may be applied to data items of various types were discussed in previous chapter. It remains to examine how these operations are incorporated into programs. Operations are expressed in the form of statements. The simplest statement is the assignment statement. It

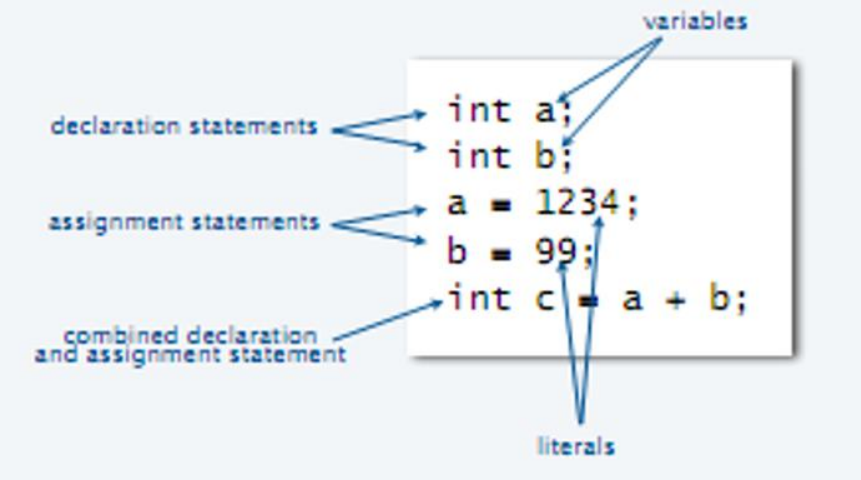
consists of a variable name, followed by the assignment operator (=), followed by some sort of expression. The assignment operation has the form:

variable = expression

The assignment operation is used to assign a name to a value. Thus it is used whenever there is need to keep track of a value that is needed later. Some typical uses include:

- initialize a variable (count = 0)
- increment/decrement a counter (count = count + 1)
- accumulate values (sum = sum + item)
- capture the result of a computation (y = 3*x + 4)
- swap two values (t = x; x = y; y = t)

The assignment operator is not commute i.e. x = e is not the same as e = x.



3.3.5 CONTROL

In the problem-solving phase of computer programming, you will be designing algorithms. This means that you will have to be conscious of the strategies you use to solve problems in order to apply them to programming problems. These algorithms can be designed though the use of flowcharts or pseudocode. To implement an algorithm, it should be described in an understandable form. The descriptions are called constructs. The key to better algorithm design and thus to programming, lies in properly defining the control structures. These control structures can be grouped into three constructs namely the sequence structure, the decision structure or

selection structure and the repetition or iteration structure

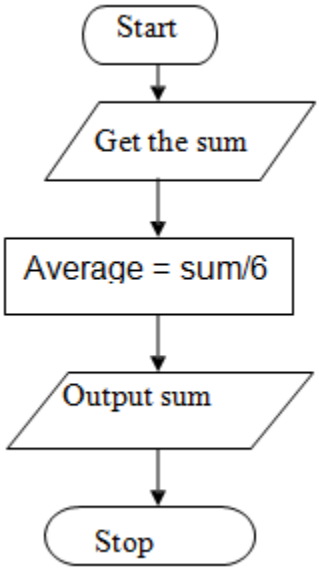
3.3.5.1 The sequence structure

The first type of control structures is called the sequence structure. This structure is the most elementary structure. The sequence structure is a case where the steps in an algorithm are constructed in such a way that, no condition step is required. The sequence structure is the logical equivalent of a straight line.

Example 1: Suppose you are required to design an algorithm for finding the average of six numbers, and the sum of the numbers is given. The pseudocode will be as follows:

Start
Get the sum
Average = sum / 6
Output the average
Stop

The corresponding flowchart will appear as follows:



Example 2: This is the pseudo-code required to input three numbers from the keyboard and output the result.

Use variables: sum, number1, number2, number3 of type integer
Accept number1, number2, number3
Sum = number1 + number2 + number3
Print sum
End program

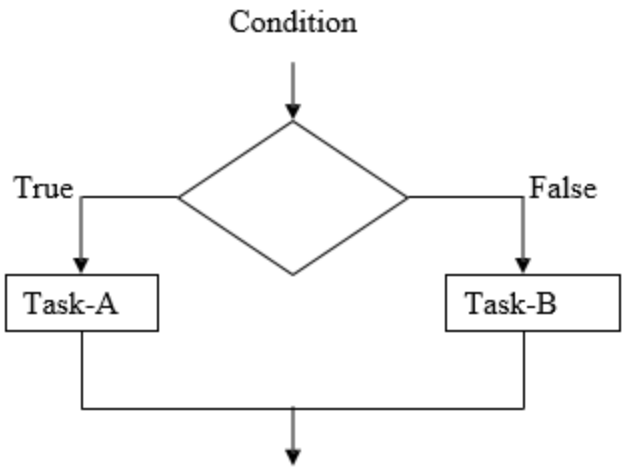
Example 3: The following pseudo-code describes an algorithm which will accept two numbers from the keyboard and calculate the sum and product displaying the answer on the monitor screen.

Use variables sum, product, number1, number2 of type real
display "Input two numbers"
accept number1, number2
sum = number1 + number2
print "The sum is ", sum
*product = number1 * number2*
print "The Product is ", product
end program

3.3.5.2 Decision Structure or Selection Structure

The decision structure or mostly commonly known as a selection structure, is case where in the algorithm, one has to make a choice of two alternatives by making decision depending on a given condition.

Selection structures are also called *CASE* selection structures when there are two or more alternatives to choose from. This structure can be illustrated in a flowchart as follows:



In pseudocode form:

If condition is true

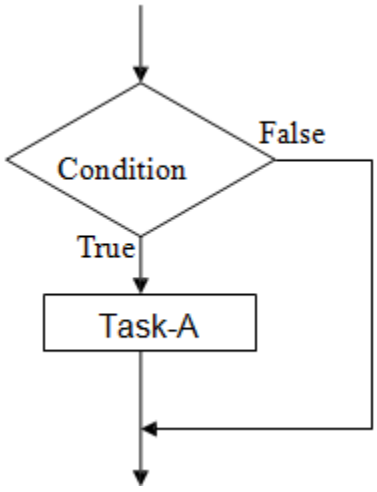
Then do task A

else

Do Task-B

In this example, the condition is evaluated, if the condition is true Task-A is evaluated and if it is false, then Task-B is executed

A variation of the construct of the above figure is shown below



The above structure implies the following:

If condition is true then

Do Task-A

In this case, if condition is false, nothing happens. Otherwise Task-A is executed. The selection requires the following

- Choose alternative actions as a result of testing a logical condition
- Produce code to test a sequence of logical tests

There are many occasions where a program is required to take alternative actions. For example, there are occasions where there is need to take action according to the user choice. All computer languages provide a means of selection. Usually it is in the form of If statement and our pseudo-code is no exception to this. The if statement together with logical operators will be used to test for true or false as shown below.

If a = b
print "a = b"

The action is only taken when the test is true.

The logical operators used in pseudo-codes are

=	is equal to
>	is greater than
<	is less than
>=	is greater than or equal
<=	is less than or equal
<>	is not eaqual to

Example 4: The following shows how the selection control structure is used in a program where a user chooses the options for multiplying the numbers or adding them or subtracting.

Use variables: choice, of the type character
ans, number1, number2, of type integer
display "choose one of the following"
display "m for multiply"
display "a for add"
display "s for subtract"

accept choice

display "input two numbers you want to use"

accept number1, number2

*if choice = m then ans = number1 * number2*

if choice = a then ans = number1 + number2

if choice = s then ans = number1 - number2

display ans

Compound Logical Operators

There are many occasions when there is need to extend the conditions that are to be tested. Often there are conditions to be linked.

In everyday language there is a statement like '*If I had the time and the money I would go on holiday'*. The '*and'* means that both conditions must be true before any action is taken. Another statement is '*I am happy to go to the theatre or the cinema'*. The logical link this time is *or*. Conditions in if statements are linked in the same way. Conditions linked with and only result in an action when all conditions are true. For example, if a >b and a > c then display "a is the largest". Conditions linked with an 'or' lead to an action when either or both are true.

Example 5: The program is to input an examination mark and test it for the award of a grade. The mark is a whole number between 1 and 100. Grades are awarded according to the following criteria:

>=	80 Distinction
>=	60 Merit
>=	40 Pass
<	40 fail

The pseudo-code is

Use variables: mark of type integer

If mark >= 80 display "distinction"

If mark >= 60 and mark < 80 display "merit"

If mark >= 40 and mark < 60 display "pass"

If mark < 40 display "fail"

An if statement on its own is often not the best way of solving problems. A more elegant set of conditions can be created by adding an else statement to the if statement. The else statement is used to deal with situations as shown in the following examples.

Example 6: A person is paid at top for category 1 work otherwise pay is at normal rate.

```
If
        the work is category 1
        pay-rate is top
Else
        pay-rate is normal
```

The else statement provides a neat way of dealing with alternative condition. In pseudo- code this can be written as:

```
If
        work = cat1
then
        p-rate: = top
Else
        p-rate = normal
```

The following example illustrate the use of if … else statements in implementing double alternative conditions.

```
If
        salary < 50000
then
```

```
        Tax = 0
Else
    If
        salary > 50000 AND salary < 100000
    then
        Tax = 50000 * 0.05
   Else
       Tax = 100000 * 0.30
```

The case statement

Repeating the if … else statements a number of times can be somewhat confusing. An alternative method provided in a number of languages is to use a selector determined by the alternative conditions that are needed. This is called a **case statement**.

Example 7: The following program segment outputs a message to the monitor screen describing the insurance available according to a category input by the user.

```
Use variables: category of type character
Display    "input    category"
Accept category
If category = U
    Display "insurance is not available"
    Else
        If category = A then
           Display "insurance is double"
        Else
            If category = B then
               Display  "insurance  is  normal"
        Else
            If category = M then
```

Display "insurance is medically dependent"
Else
Display "entry invalid"

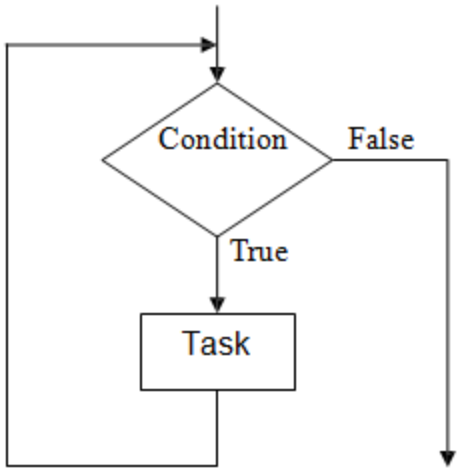
This can be expressed in a case statement as follows:

Use variables: category of type character
Display "input category"
Accept category
DO case of category
CASE category = U
Display "insurance not available"
CASE category = A
Display "insurance is double"
CASE category = B
Display "insurance is normal"
CASE category = M
Display "insurance is medically dependent"
OTHERWISE
Display "entry is invalid"
ENDCASE

Instead of using the word *otherwise*, one can use *else*.

3.3.5.3 Repetition or Iteration Structure

A third structure causes the certain steps to be repeated.



The Repetition structure can be implemented using

- Repeat Until Loop
- The While Loop
- The For Loop

Any program instruction that repeats some statement or sequence of statements a number of times is called an iteration or a loop. The commands used to create iterations or loops are all based on logical tests. There three constructs for iterations or loops in our pseudo- code.

The Repeat Until loop.

The syntax is

REPEAT

A statement or block of statements

UNTIL a true condition

Example 8: A program segment repeatedly asks for entry of a number in the range 1 to 100 until a valid number is entered.

REPEAT

DISPLAY "Enter a number between 1 and 100"

ACCEPT number

UNTIL number < 1 OR number > 100

Example 9. A survey has been carried out to discover the most popular sport. The results will be typed into the computer for analysis. Write a program to accomplish this.

```
REPEAT
    DISPLAY "Type in the letter chosen or Q to finish"
    DISPLAY "A: Athletics"
    DISPLAY "S: Swimming"
    DISPLAY "F: Football"
    DISPLAY "B: Badminton"
    DISPLAY "Enter data"
    ACCEPT letter
    If letter = 'A' then
       Athletics = athletics + 1
    If letter = 'S' then
       Swimming = Swimming + 1
    If letter = 'F' then
       Football = Football + 1
    If letter = 'B' then
       Badminton = Badminton + 1
UNTIL letter = 'Q'
DISPLAY "Athletics scored", athletics, "votes"
DISPLAY "Swimming scored", swimming, "votes"
DISPLAY "Football scored", football, "votes"
DISPLAY "Badminton scored", Badminton, "votes"
```

The WHILE loop

The second type of iteration to be considered is the while iteration. This type of conditional loop tests for terminating condition at the beginning of the loop. In this case no action is performed at all if the first test causes the terminating condition to evaluate as false.

The syntax is

WHILE (a condition is true)

A statement or block of statements

ENDWHILE

Example 10: A program segment to print out each character typed at a keyboard until the character 'q' is entered.

```
WHILE letter <> 'q'
    ACCEPT letter
    DISPLAY "The character you typed is", letter
ENDWHILE
```

Example 11: Write a program that will output the square root of any number input until the number input is zero.

In some cases, a variable has to be initialised before execution of the loop as shown in the following example.

```
Use variable: number of type real
DISPLAY "Type in a number or zero to stop"
ACCEPT number
WHILE number <> 0
    Square = number * number
    DISPLAY "The square of the number is", square
    DISPLAY "Type in a number or zero to stop"
    ACCEPT number
ENDWHILE
```

The FOR Loop

The third type of iteration, that can be used when the number of iterations is known in advance, is a for loop. This, in its simplest form, uses an initialisation of the variable as a starting point, a stop condition depending on the value of the variable. The variable is incremented on each iteration until it reaches the required value.

The pseudo-code syntax will be:

FOR *(starting state, stopping condition, increment)*

Statements

ENDFOR

Example 12.

FOR (n = 1, n <= 4, n + 1)

DISPLAY "loop", n

ENDFOR

The fragment of code will produce the output

Loop 1

Loop 2

Loop 3

Loop 4

In the example, n is usually referred to as the loop variable, or counting variable, or index of the loop. The loop variable can be used in any statement of the loop. The variable should not be assigned a new value within the loop, which may change the behaviour of the loop.

Example 13: Write a program to calculate the sum and average of a series of numbers. The pseudo-code solution is:

Use variables: n, count of the type integer

Sum, number, average of the type real

DISPLAY "How many numbers do you want to input"

ACCEPT count

SUM = 0

FOR (n = 1, n <= count, n + 1)

DISPLAY "Input the number from your list"

ACCEPT number

SUM = sum + number

ENDFOR

Average = sum / count

DISPLAY "The sum of the numbers is ", sum

DISPLAY "Average of the numbers is ", average

Flowcharts have been used in this section to illustrate the nature of the three control structures. These three are the basic control structures out of which all programs are built. Beyond this, flowcharts serve the programmer in two distinct ways: as problem solving tools and as tools for documenting a program.

Example 14

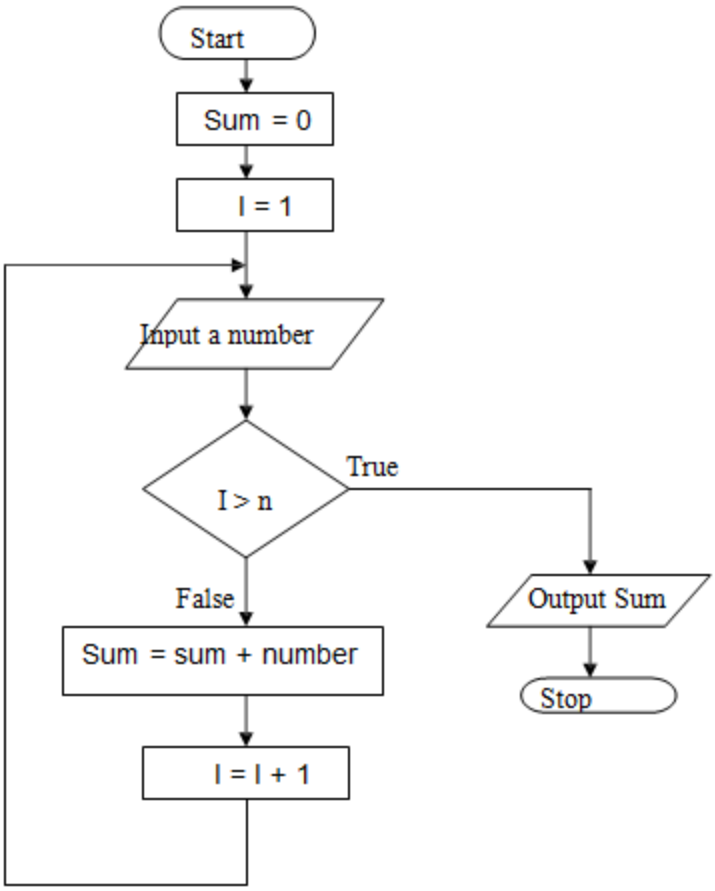
Design an algorithm and the corresponding flowchart for finding the sum of n numbers.

Pseudocode Program

Start
Sum = 0
Display "Input value n"
Input n
For(I = 1, n, 5) Input a
value
Sum = sum + value
ENDFOR
Output sum Stop

In this example, 'I' is used to allow counting of the numbers for the addition. 'I' is compared with 'n' to check whether the numbers have been exhausted or not in order to stop the computation of the sum (or to stop the iteration structure). In such a case, 'I' is referred to as a counter.

The corresponding flowchart will be as follows:



4.0 CONCLUSION

In this unit, it has been demonstrated that ability to write simple computer programs depends on the understanding of the concept of programming. Also, the ability to go through and carefully adhere to the rules and processes of program development life cycle will ensure the development of correct and reliable programs. The discussion on program structures in this unit will also ensure that students learning how to write computer programs will have sound understanding of problem solving through the understanding of the basic control structures.

5.0 SUMMARY

In discussing the basic programming concept, the importance of program design and specification processes have been emphasized in this unit. The forms in which program components are constructed, organized and interrelated namely storage, data declaration, input and output mechanism, operations on data and more importantly control structures

were properly discussed. The three control structures emphasized in this unit are sequence structures, decision structures and repetition structures.

6.0 TUTOR-MARKED ASSIGNMENT

1. What is Programming?
2. Explain Program Development Life Cycle (PDLC) in details
3. Differentiate between an identifier and a variable
4. Explain what sequential programs entail
5. Explain the following constructs in the implementation of algorithms
    i. The sequence structure
    ii. Decision Structure
    iii. Repetition Structure
6. Design an algorithm and the corresponding flowchart for finding the sum of the numbers 2, 4, 6, 8, …, n
7. Using flowcharts, write an algorithm to read 100 numbers and then display the sum.
8. Write an algorithm to read two numbers then display the largest.
9. Write an algorithm to read two numbers then display the smallest

7.0 REFERENCES/FURTHER READINGS

1. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).
2. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).
3. PROG0101. (2019). Fundamentals *of Programming Chapter 2: Programming Languages.* FTMS College Kualar Lumpur, Malaysia. Retrieved online at *https:// ftms.edu.my* on 20th November, 2021.
4. Johnew Zhang. (2012). *CS 241 Notes : Foundations of Sequential Programming*
5. Chris Thomson. (2013). *CS 241: Foundations of Sequential Programs.* Winter 2013, University of Waterloo
6. Matt Fredrikson and Andre Platzer (2014). *Lecture Notes on Sequential Programs and Compositional Reasoning.* Carnegie Mellon University

UNIT 2: EVOLUTION OF PROGRAMMING LANGUAGES

CONTENTS

1.0 INTRODUCTION

2.0 OBJECTIVES

3.0 MAIN CONTENTS

3.1 Computers and Numbers

3.1.1 Decimal numbers

3.1.2 Binary Numbers

3.1.3 Octal Numbers

3.1.4 Base 16 (Hexadecimal)

3.1.5 Converting Between Number Bases

3.1.5.1 Converting from Base 10 to Any Base

3.1.5.2 Converting from Any Base to Base 10 (Decimal)

3.1.5.3 Hexadecimal to Binary

3.1.5.4 Binary to Hexadecimal

3.1.5.5 Conversions between Other Bases

3.2 Programming Language Classifications

3.2.1 Low Level Languages (LLL)

3.2.1.1 Machine Language

3.2.1.2 Assembly Language

3.2.1.3 High Level Language (HLL)

3.3 Generations of Programming Language

4.0 CONCLUSION

5.0 SUMMARY

6.0 TUTOR-MARKED ASSIGNMENT

7.0 REFERENCES/FURTHER READINGS

1.0 INTRODUCTION

To communicate with computers, data must be converted into forms more readily acceptable to computers. Computers understands only a simple language that consists of 1s and 0s, with a 1 representing the presence of electrical signal in the signal path while a 0 represents the absence of electrical signal. Instructions are therefore coded into computer's memory as 0s and 1s. This method of instructing computer is called machine language. As the computer operates using a program coded in 0s and 1s, it is highly laborious for a programmer to write a program in 0s and 1s. Programmers find it easier to write programs in a language approaching that of English. This language is called high level language. Over the years, computer languages have evolved from machine language to high-level languages.

In this unit, in discussing the evolution of programming languages, we will study computer and number system as a basis of communicating with the computer, the machine language, assembly language and high level languages.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- discuss the uses of number bases in computing
- perform conversion between different number bases
- explain what machine language entails
- describe the composition of an assembly language statement
- describe the motivation that led to the development of programming languages in high level language from machine language.
- discuss the structure of sequential programs.

3.0 MAIN CONTENT

3.1 COMPUTERS AND NUMBERS

When digital computers store and process data, they make use of numbers in base two. Several other number bases also have uses in computing and so the general idea of number bases together with the methods for converting from one base to another must be developed.

3.1.1 Decimal numbers

Decimal numbers also known as denary numbers or number to base 10 are the numbers in everyday use because ten is the basis of the number system. To write a number in decimal, we make use of the ten digit symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Let say we have on our hands the decimal number 2,153. Let's have a look at just what the number means:

T	H	T	U
2	1	5	3

Basically, it means 2 Thousands, 1 Hundred, 5 Tens and 3 Units. This could also be expressed in the powers of 10 as follows:

Number	2	1	5	3	
Place Holder	10^3	10^2	10^1	10^0	
Result	2×10^3 = 2, 000	1×10^2 = 100	5×10^1 = 50	3×10^0 = 3	Total = 2,153

In the decimal system the place-holder for each digit is a power of 10 so that moving from right to left, in the table, corresponds to an increase in magnitude by a factor of 10 at every step.

3.2.2 Binary Numbers

The binary, or base 2, number system uses the two digits 0 and 1 to represent numbers and is of particular importance in computing. In a computer's memory elements can be in one of two states, OFF or ON corresponding to the digits 0 and 1 respectively. These elements represent one binary digit or bit. All internal processing and calculations in computing are done in binary.

We have seen that in base 10 every number can be written as a weighted sum of powers of 10. In an analogous manner for base 2 we use a weighted sum of powers of 2 to express numbers. The place-holder for each digit is therefore a power of 2 and moving

from right to left corresponds to an increase in magnitude by a factor of 2 at every step. For example, consider the following table.

Place-Holder	2^3	2^2	2^1	2^0
Weight	8	4	2	1
Binary digit	1	1	0	1

The binary number in the table, 1101 is sometimes written with the subscript "2" to indicate that it is a base 2 number, i.e. 1101_2. To obtain the decimal representation of 1101 we multiply each binary digit by its column's weight and sum the values. Starting from the right,

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 1 + 0 + 4 + 8 = 13.$$

Hence $1101_2 = 13_{10}$.

3.2.3 Octal Numbers

Octal numbers are numbers to base 8. There are eight symbols used in the octal system, 0, 1, 2, 3, 4, 5, 6, and 7. Its place holder increase in powers of 8. Octal numbers are used as a shorthand for binary. Octal used to be popular when computers employed 12-bit, 24-bit or 36-bit words for data and addressing. However, as modern computers all use 16-bit, 32-bit or 64-bit words octal is rarely used nowadays. Consider the table given below:

Place Holder	8^2	8^1	8^0
Weight	64	8	1
Octal Numbers	1	5	5

The octal number in the table, 155 is sometimes written with the subscript "8" to indicate that it is a base 8 number, i.e. 155_8. To obtain the decimal representation of 155 we multiply each octal digit by its column's weight and sum the values. Starting from the right,

$$5 \times 8^0 + 5 \times 8^1 + 1 \times 82 = 5 + 40 + 64 = 109_{10}$$

Hence, $155_8 = 109_{10}$

3.2.4 Base 16 (Hexadecimal)

The hexadecimal (often called hex) or base 16 number system uses sixteen symbols,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, to represent numbers. The first ten digits are the same as in the decimal system while the remaining six, A to F, correspond to the numbers from 10 to 15 respectively.

A computer carries out all its operations in binary but as numbers become large the binary representation requires increasingly more digits (0's and 1's) and becomes difficult for humans to read and write. For this reason computers often display information, such as memory addresses, in hexadecimal as their format is more compact.

In base 16 we use a weighted sum of powers of 16 to express numbers. The place-holder for each digit is therefore a power of 16 and moving from right to left corresponds to an increase in magnitude by a factor of 16 at every step. Consider the table given below:

Place-Holder	16^3	16^2	16^1	16^0
Weight	4096	256	16	1
Hex digit	1	2	B	F

The hex number in the table, 12BF can be written with the subscript "16", to indicate that it is a base 16 number, i.e. $BF12_{16}$.

To obtain the decimal representation of $BF12_{16}$, we multiply each hex digit by its column's weight, noting that B represents 11 and F corresponds to 15, and sum the values, i.e.

$$15 \times 16^0 + 11 \times 16^1 + 2 \times 16^2 + 1 \times 16^3 = 15 + 176 + 512 + 4096 = 4799.$$

Hence, $12BF_{16} = 4799_{10}$

3.2.5 Converting Between Number Bases

In this section we look at converting integers between different number systems. While the main focus will be on the bases most commonly used in computing, i.e. 2 (binary), 10 (decimal) and 16 (hex) we also present some results for other bases including octal (base 8). The ability to convert back and forth between different bases is a fundamental skill required of anyone working in the area of computing.

3.1.5.1 Converting from Base 10 to Any Base

Converting from base 10 (decimal) to any other base is easy. Start with the decimal number to be converted and repeatedly divide by the new base number retaining the remainder at each step. We shall illustrate with some examples.

(i). Base 10 (Decimal) to Base 2 (Binary)

Example 1: Convert the decimal number 475 to a binary number.

Solution

Start by dividing 475 by 2 and keep the remainder. Repeat the process until we can no longer perform a division.

475 / 2 = 237, remainder 1
237 / 2 = 118, remainder 1
118 / 2 = 59, remainder 0
59 / 2 = 29, remainder 1
29 / 2 = 14, remainder 1
14 / 2 = 7, remainder 0
7 / 2 = 3, remainder 1
3 / 2 = 1, remainder 1
1 / 2 = 0, remainder 1

Now read the binary number from the bottom to the top: 111011011.

Hence $475_{10} = 111011011_2$

(ii). Base 10 (Decimal) to Base 16 (Hexadecimal)

Example 2: Convert the decimal number 795 to a hex number.

Solution

Start by dividing 795 by 16 and keep the remainder. Repeat the process until we can no longer perform a division.

795 / 16 = 49, remainder 11 (= B in hex)

49 / 16 = 3, remainder 1

3 / 16 = 0, remainder 3

Now read the hex number from the bottom to the top: 31B.

Hence $795_{10} = 31B_{16}$

(iii). Base 10 (Decimal) to Base 8 (Octal)

Example 3: Convert the decimal number 5361 to an octal number.

Solution

Start by dividing 5361 by 8 and keep the remainder. Repeat the process until we can no longer perform a division. The octal number system is similar to decimal except that it only uses the eight digits from 0 to 7.

5361 / 8 = 670, remainder 1

670 / 8 = 83, remainder 6

83 / 8 = 10, remainder 3

10 / 8 = 1, remainder 2

1 / 8 = 0, remainder 1

Now read the octal number from the bottom to the top:

Hence, $5361_{10} = 12361_8$

3.1.5.2 Converting from Any Base to Base 10 (Decimal)

Converting to base 10 (decimal) from any other base is also fairly straightforward. We shall consider the place value method. The method is based on the "place values" of the digits in the number being converted.

Let $N_b = x_1 x_2 x_3 \ldots x_m$ be a base b number with m digits.

To convert to base 10 we calculate as follows:

$$N_{10} = x_m b^0 + x_{m-1} b^1 + x_{m-2} b^2 + \ldots + x_1 b^{m-1}$$

(i). Binary to Decimal

Example 4:

(a). Convert the binary number 11001 to a decimal number.

(b). Convert the binary number 11011101 to a decimal number.

Solution

(a) The place values of digits in a binary number are powers of 2. To convert 11001 proceed as follows:

$$11001 = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4$$
$$= 1 + 0 + 0 + 8 + 16 = 25.$$

Hence, $11001_2 = 25_{10}$.

As base 2 only uses the numbers 0 and 1 this approach essentially involves adding the non-zero place values together.

(b) From the right, adding the place values, corresponding to the non-zero digits, in 11011101 gives:

$$1 + 0 + 4 + 8 + 16 + 0 + 64 + 128 = 221.$$

Hence, $11001_2 = 221_{10}$.

(ii). Hexadecimal to Decimal

Example 5:

(a). Convert the hexadecimal number 3B2 to a decimal number.

(b). Convert the hexadecimal number 4BAE to a decimal number.

Solution

(a) The place values of digits in a hex number are powers of 16. To convert 3B2 to its decimal representation, starting from the right, multiply each digit in 3B2 by the appropriate power of 16.

$$3B2_{16} = 2 \times 16^0 + 11 \times 16^1 + 3 \times 16^2$$
$$= 2 \times 1 + 11 \times 16 + 3 \times 256$$
$$= 2 + 176 + 768 = 946.$$

Hence, $3B2_{16} = 946_{10}$.

(b)

$$4BAE_{16} = 14 \times 16^0 + 10 \times 16^1 + 11 \times 16^2 + 4 \times 16^3$$
$$= 14 \times 1 + 10 \times 16 + 11 \times 256 + 4 \times 4096$$
$$= 19374.$$

Hence, $4BAE_{16} = 19374_{10}$.

(iii). Octal (Base 8) to Decimal

Example 6: Convert the octal number 7630 to a decimal number

Solution

The place values of digits in octal numbers are powers of 8. To convert 7630 to its decimal representation, starting from the right, multiply each digit in 7630 by the appropriate power of 8.

$$7630_8 = 0 \times 8^0 + 3 \times 8^1 + 6 \times 8^2 + 7 \times 8^3$$
$$= 0 \times 1 + 3 \times 8 + 6 \times 64 + 7 \times 512$$
$$= 3992$$

Hence, $7630_8 = 3992_{10}$

3.1.5.3 Hexadecimal to Binary

As hexadecimal is base 16 = 2^4, and binary is base 2 = 2^1, every digit in a hex number can be replaced by its four bit binary equivalent.

Example 7: Convert the hexadecimal number 3C7D to a binary number.

Solution

Replace each hexadecimal number with its 4-bit binary equivalent.

Hex	3	C	7	D
Binary	0011	1100	0111	1101

Hence, $3C7D_{16} = 0011\ 1100\ 0111\ 1101_2$.

3.1.5.4 Binary to Hexadecimal

Example 8: Convert the binary number 1111 1100 0100 1110 to a hexadecimal number.

Solution

• Starting from the right-hand side-split the number into groups of four. If necessary pad on the left with zeros to obtain a group of four.

• Convert each group of four to its decimal equivalent using the binary placeholder weightings, i.e. 1, 2, 4 and 8. For example, in the table below, the group of four on the right gives,

8 + 4 + 2 + 0 = 14. Hence, $1110_2 = 14_{10}$

Convert each decimal number to its hex equivalent, e.g. $14_{10} = E_{16}$

Binary	1111	1100	0100	1110
Decimal	15	12	4	14
Hex	F	C	4	E

Hence, $1111\ 1100\ 0100\ 1110_2 = FC4E_{16}$.

3.1.5.5 Conversions between Other Bases

Here we present some examples of how the methods described earlier can be applied to conversions between other bases.

(i). Octal to Binary

As octal is base 8 = 2^3 and binary is base 2 =2^1, every digit in an octal number can be replaced by its three bit binary equivalent.

Example 9: Convert the octal number 761 to a binary number.

Solution

Replace each octal digit with its 3-bit binary equivalent.

Octal	7	6	1
Binary	111	110	001

Hence, $761_8 = 111110001_2$.

(ii). Binary to Octal

Example 10: Convert the binary number 1110101000101 to an octal number.

Solution

• Starting from the right hand side split the number into groups of three. If necessary pad on the left with zeros to obtain a group of three.

• Convert each group of three to its octal equivalent using the binary placeholder weightings, i.e. 1, 2 and 4. For example, on the right we have, 4 + 0 + 1 = 5.

Hence, $101_2 = 5_8$

Binary	001	110	101	000	101
Octal	1	6	5	0	5

Hence, $1110101000101_2 = 16505_8$.

(iii). Hexadecimal to Octal

Example 11: Convert the hexadecimal number 8B6E to an octal number.

Solution

One method is to convert the hex number to binary and then convert from binary to octal.

Write each hex digit as a four bit binary number.

Hex	8	B	6	E
Binary	1000	1011	0110	1110

Starting from the right, split the binary representation into groups of three. Pad the leftmost triple with zeros if required.

Binary	001	000	101	101	101	110
Octal	1	0	5	5	5	6

Hence, $8B6E_{16} = 105556_{8}$.

(iv). Octal to Hexadecimal

Example 12: Convert the octal number 6473 to a hex number.

Solution

All we have to do is reverse the process in the previous example.

Write each octal digit as a three bit binary number.

Octal	6	4	7	3
Binary	110	100	111	011

Starting from the right, split the binary representation into groups of four. Pad the leftmost group with zeros if required.

Convert each decimal number to its hex equivalent, e.g. $1011_2 = 11_{10}$.

Convert each binary number to its decimal equivalent, e.g. $11_{10} = B_{16}$.

Binary	1101	0011	1011
Decimal	13	3	11
Hex	D	3	B

Hence, $6473_8 = D3B_{16}$.

3.3 Programming Language Classifications

Programming language can be classified as Low Level Language (LLL) or High Level Language (HLL). The binary machine language is usually defined as the lowest level, whereas the highest level might be human language such as English

3.2.1 Low Level Languages (LLL)

Low-level languages are designed to operate and handle the entire hardware and instructions set architecture of a computer directly. A program written in a low level language can be made to run very quickly, and with a very small memory requirement when compared with the equivalent program in a high-level language. However, they are considered difficult to use, due to the numerous technical details such as the computer instruction set architecture which must be remembered. Low-level programming languages are sometimes divided into two categories: Machine Language and Assembly language.

3.2.1.1 Machine Language

You have already seen how numbers are handled by computers. And machine language is only a lot of numbers that we have just discussed. The difference is that whereas we were thinking of these numbers as just numbers, machine language treats them as more than such. A particular number when used in machine language, will cause the CPU to perform a particular activity or instruction. For example: $8B, decimal 139 or binary 1000 1011 could cause the CPU to add two numbers together. Nonetheless, this is what machine language is all about. The name says it all! It is language for machines. Each manufacturer of the different CPUs has designed a different language for its product.

A machine language is a programming language in which the instruction are in a form that allows the computer to perform them immediately, without any further translation being required.

Machine language is the sequence of bits (machine code) that directly controls a processor, causing it to add, compare, or move data from one place to another. The computer microprocessor can process directly the machine codes without a previous transformation. Writing programs at this level is an enormously tedious task and also requires memorizing or looking up numerical codes for every instruction that is used.

Instructions in machine language are in the form of a binary code, also called machine code, and are called machine instructions. Machine instructions are stored in the same way as data, and each instruction corresponds directly to a hardware facility on the machine for which it is written.

At this stage you may be asking yourself - if this is what machine language is all about, why bother? The reason you should bother is because of the benefits of machine language, these are:

i. Faster execution of the program
ii. More efficient use of memory
iii. Shorter programs (in memory)
iv. Freedom from the operating system

All of the above benefits are a direct result of programming in a language that the CPU can understand without having to have it translated first. When you program in high level language, the operating system is a machine language program that is being run by the machine. The program could be described like this:

"Next
Look at the next instruction
Translate it into a series of machine language instructions
Perform each instruction
Store the result if required
Goto Next"

Programming in high level language can be up to 60 times slower than a program written directly in machine language! This is because translation takes time, and also the resulting machine language instructions generated are usually less efficient. However, it would also be admitted that programming in machine language does have drawbacks.

The main disadvantages of machine language are:

i. Programs are more difficult to read and debug
ii. Impossible to adapt to other computers
iii. Longer programs (in instructions)
iv. Arithmetic calculations difficult

It is not advocated that you write program you want in machine language instead of high level language. A program used for calculating averages is a simple one when written in high level language, but a machine language version would take a lot more effort. The study of machine language today is good for the understanding of how computers handles data and instructions.
Further discussion on machine language and machine instructions in particular will be discuss in details in Module 2.

3.2.1.2 Assembly Language

Quite obviously if machine language could only be represented by numbers, very few people would want to write programs in machine language. After all, who would make sense of a program which looked like:

```
0 0 1 0 0 0 0 1    or   $21
0 0 0 0 0 0 0 0    or   $00
0 1 0 0 0 0 0 0    or   $40
etc.
```

Fortunately, we can invent a series of names for each of these numbers. Each computer manufacturer normally devices a low-level language that corresponds closely to the particular machine language used by that manufacturer. This language is called an Assembly language. Assembly language is just such a representation of machine language, enabling it to be read by humans. The main difference between assembly language and machine language is that assembly language is one level higher than machine language. It is more easily read by humans than machine language, but on the other hand, computers can't read assembly language

Assembly language can be converted directly into machine code by a program. Such a program is called an assembler. You can see that this is a program which performs the rather boring task of

translating your assembly language program into a sequence of machine language instructions that the CPU will understand i.e. into binary numbers. The manufacturer provides the Assembler which translate the Assembly language into machine code. For each assembly language instruction there is an identical machine language instruction, and vice versa; i.e. there is a one-to-one relationship between them. We can therefore say that assembly language is equivalent to machine language. A program written in assembly language is called the Source Program. The translated program in machine code is called the Object Program. Assembly languages differ, since the features of each assembly language depend on the particular computer on which it is used.

Assembly language uses structured commands called mnemonics as substitutions for numbers allowing humans to read the code easier than looking at binary. For example, at this stage, the instruction:

INC A

may not mean much to you but at least you can read it. If you were told "INC" is a standard mnemonic for increment (INCrement) and that A is a variable, then by simply looking at the instruction you can get a feel for what is happening. The same instruction in machine language is

0100 1100

Now obviously you can also "read" that instruction in the sense that you can read the number, but it doesn't mean much unless you have a table to look up or, your brain is capable of functioning like a computer. Although easier to read than binary, assembly language is a difficult language. The problem with assembly language is that it requires a high level of technical knowledge, and it's slow to write. Typically, one machine instruction is represented as one line of assembly code. Mnemonic codes are used in place of machine codes, e.g. using LDA 5 in place of 0000000000000101. Symbolic addresses are frequently used instead of actual machine addresses e.g. using LDA N where N stands for the address which can be assigned a numerical value at a more convenient time. This is only scratching the surface of assembly language. You can do a lot more than write mnemonics instead of numbers for instructions. A line of an assembly language program can be divided into the following sections:

1. COMMENTS

As you have probably already guessed, a comment is a few words which don't affect the actual program but is there to remind you, or tell other people that may look at your program, exactly what you are doing in this section of the program.

2. OPERANDS

We said before that there was an instruction that added two numbers together but we didn't say what these numbers were. This is what the operand does. It can tell the CPU which numbers to use or, in other cases the mnemonic tells the CPU what instruction to execute and the operand tells the CPU what to use that instruction with.

3. MNEMONICS

This has been discussed earlier.

4. LABELS - Labels let you give a name to a line and other things such as complete programs, constants, variables, etc.

The program given below shows assembly language program to add two numbers A & B.

Program code	**Description**
READ A	It reads the value of A.
ADD B	The value of B is added with A.
STORE C	The result is store in C.
PRINT C	The result in 'C' is printed.
HALT	Stop execution.

3.2.1.3 High Level Language (HLL)

High Level languages have replaced machine and assembly language in all areas of programming. Programming languages were designed to be high level if it is independent of the underlying machine. High-level languages (also known as problem-oriented languages) enable a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and farther from machine languages. High level languages are portable (machine independent) as it can be run on different machines with little or no change. High-level languages provide a richer set of

instructions and support, making the programmer's life even easier. High level languages use translator programs such as compiler and interpreter to convert it into a machine language program. The followings are popular examples of high level languages:

A. Pascal

• A high-level programming language developed by Niklaus Wirth in the late 1960s.

• The language is named after Blaise Pascal, a seventeenth-century French mathematician who constructed one of the first mechanical adding machines.

• It is a popular teaching language.

Example:

```
Program HelloWorld(output);
begin
end.
writeLn('Hello, World!')
```

B. **C**

• Developed by Dennis Ritchie at Bell Labs in the mid-1970s.

• C is much closer to assembly language than are most other high-level languages.

• The first major program written in C was the UNIX operating system.

• The low-level nature of C, however, can make the language difficult to use for some types of applications.

Example:

```
#include <stdio.h>
int main(void)
{
printf("hello, world\n");
return 0;
}
```

C. C++

• A high-level programming language developed by Bjarne Stroustrup at Bell Labs.
• C++ adds object-oriented features to its predecessor, C.
• C++ is one of the most popular programming language for graphical applications, such as those that run in Windows and Macintosh environments.

Example:

```
#include <iostream>
int main()
{
std::cout << "Hello World!" << std::endl;
return 0;
}
```

D. BASIC

• Short for **Beginner's All-purpose Symbolic Instruction Code**.
• Developed in the 1950s for teaching University students to program and provided with every self-respecting personal computer in the 1980s,
• BASIC has been the first programming language for many programmers.
• It is also the foundation for Visual Basic.

Example:

```
PRINT "Hello world!"
```

E. Visual Basic

• A programming language and environment developed by Microsoft.
• Based on the BASIC language, Visual Basic was one of the first products to provide a graphical programming environment and a paint metaphor for developing user interfaces.

Example:

```
MsgBox "Hello, World!“
```

F. JAVA

• A high-level programming language developed by Sun Microsystems.

• Java was originally called *OAK*, and was designed for handheld devices and set-top boxes.

• Oak was unsuccessful so in 1995 Sun changed the name to Java and modified the language to take advantage of the burgeoning World Wide Web.

• Java is a general purpose programming language with a number of features that make the language well suited for use on the World Wide Web.

Example:

```
/* * Outputs "Hello, World!" and then exits */
public class HelloWorld {
}
}
public static void main(String[] args) {
System.out.println("Hello, World!");
```

3.3 Generations of Programming Language

The first generation languages, or 1GL, are low-level languages that are machine language.

The second generation languages, or 2GL, are also low-level languages that generally consist of assembly languages.

The third generation languages, or 3GL, are high-level languages such as C.

The fourth generation languages, or 4GL, are languages that consist of statements similar to statements in a human language. Fourth generation languages are commonly used in database programming and scripts.

The fifth generation languages, or 5GL, are programming languages that contain visual tools to help develop a program. A good example of a fifth generation language is Visual Basic

4.0 CONCLUSION

Programming languages have evolved from the machine instructions based on binary number system to a better readable instructions of assembly language and then to the more user friendly human method of constructing computer instructions known as high level languages.

5.0 SUMMARY

This unit has discussed number systems namely base 2, base 8 base 10 and base 16 in relation to computing. Conversion between number bases was also discussed. The relevance of number bases to machine language was emphasized. Subsequently, the evolution of programming languages from machine language to high level languages was discussed. A brief survey of few examples of high level programming languages was done. A summary of the generations of programming languages capped the discussion in this unit.

6.0 TUTOR-MARKED ASSIGNMENT

1. Convert these decimal numbers to (a) Octal (b) Binary (c) Hex
    i. 22
    ii. 751
    iii. 1453

2. Convert these octal numbers to (a) Decimal (b) Binary (c) Hexadecimal
    i. 73
    ii. 152
    iii. 1453
3. Convert these Hexadecimal numbers to decimal
    i. 6A
    ii. 2C8
    iii. 347
4. Discuss the advantages and disadvantages of machine language
5. Explain why machine language instructions are executed faster than high level languages instructions.

6. Explain the component of assembly language.
7. Explain the features of Java that make the language well suited for use on the World Wide Web.

7.0 REFERENCES/FURTHER READING

7. John, Vander Reyden (1983). *Dragon Machine Language for absolute Beginners*. Melbourne House (Publishers) Ltd, United Kingdom (1st ed).
8. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).
9. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).
10. PROG0101. (2019). Fundamentals *of Programming Chapter 2: Programming Languages.* FTMS College Kualar Lumpur, Malaysia. Retrieved online at *https:// ftms.edu.my* on 20th November, 2021.

MODULE 2

Unit 1 BASIC MACHINE ARCHITECTURE

Unit 2 DATA STORAGE IN COMPUTER

Unit 3 OPERATIONS ON DATA

Unit 4 MACHINE INSTRUCTIONS

UNIT 1: BASIC MACHINE ARCHITECTURE

CONTENTS

4.0 INTRODUCTION
5.0 OBJECTIVES
6.0 MAIN CONTENT

3.1 COMPUTER ARCHITECTURE MODELS

3.2 LEVELS WITHIN COMPUTER ARCHITECTURE

3.3 MICROPROCESSOR
3.3.1 FUNCTIONS OF THE PROCESSOR

3.4 MEMORY SYSTEM
3.4.1 MEMORY HIERARCHY

4.0 CONCLUSION

5.0 SUMMARY

6.0 TUTOR-MARKED ASSIGNMENT

7.0 REFERENCES/FURTHER READING

1.0 INTRODUCTION

In this unit, we consider the basic architectural features common to all systems where the popular Von Neumann and the system bus models will be introduced. As the hardware support computer programs by providing the operations the software requires, the understanding of basic computer architecture is therefore important in discussing computer programs. However, only the processor which is responsible for data processing and also the memory where data and instructions reside are emphasized in this unit.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- list the subsystems of a computer
- explain the interconnection of subsystems and explain different bus systems
- explain the hierarchic nature of computer systems
- describe the role of the central processing unit (CPU) in a computer
- describe the role of the memory in a computer.

3.0 MAIN CONTENT

3.1 COMPUTER ARCHITECTURE MODELS

The style of construction and organization of the many parts of a computer system are its architecture. Conventional digital computers have a common form that is attributed to Von Neumann. The Von Neumann's model consists of five major components as illustrated in Figure 1 below.

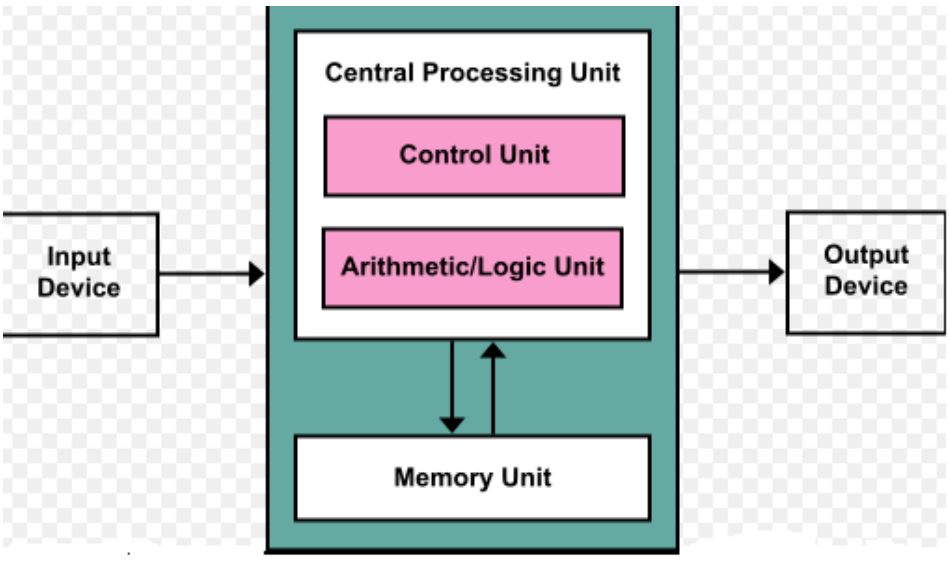


Figure 1**:** Von Neumann's model of Conventional Digital Computers

The input unit provides instructions and data to the system which are subsequently stored in the memory unit. The instructions and data are processed by the Arithmetic and logic Unit (ALU) under the direction of the Control Unit (CU). The results are sent to the output unit. The ALU and the CU are often referred to collectively as the central processing unit (CPU). The CPU controls the operation of the computer and performs its data processing functions.

Although the Von Neumann's model prevails in modern computers, it has been streamlined using system bus model as shown in Figure 2 below

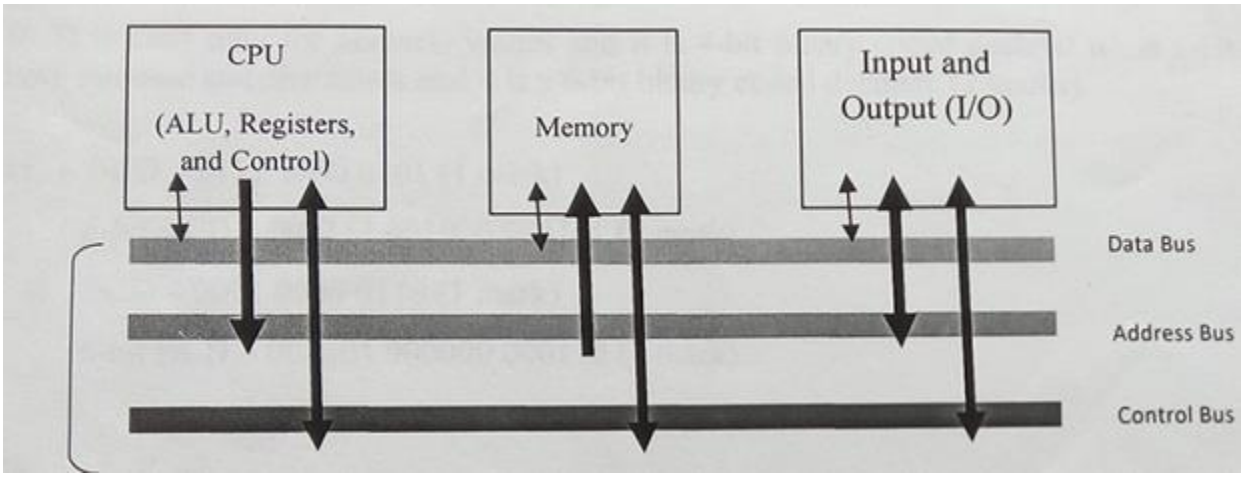


Figure 2: System Bus Model

This model partitions a computer system into three subunits: CPU, Memory, and Input/output (I/O) units. This refinement of the Von Neumann's model combines the ALU and the CU into one functional unit, the CPU. The input and output units are also combined into a single I/O unit. The model shows the communications among the components which are by means of a shared pathway called the system bus. A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Although there are many different bus designs, on any bus the lines can be classified into three functional groups: data, address, and control lines. The data bus carries the information being transmitted. Typically, the number of lines in a data bus is referred to as the width of the data bus. Since each line can carry only 1 bit at a time, the number of lines determines how many bits can be transferred at a time. The width of the data bus is a key factor in determining overall system performance. For example, if the data bus is 8 bits wide, and each instruction is 16 bits long, then the CPU must access the memory module twice during each instruction cycle.

The address bus identifies where the information is being sent of fetched. The address lines are used to designate the source of destination of the data on the data bus. For example, if the CPU wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines. Clearly, the width of the address bus determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports.

The control bus describes the manner in which information is being sent. It controls the access to the use of the data and address buses. Since the data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and timing information between system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed.

In addition, there may be power distribution lines that supply power to the attached modules.

3.2 LEVELS WITHIN COMPUTER ARCHITECTURE

A computer is a complex system; contemporary computers contain millions of elementary electronic components. How, then, can one clearly describe them? The key is to recognize the hierarchic nature of most complex systems, including the computer. A hierarchic system is a set of interrelated subsystem, each of the latter, in turn, hierarchic in structure until we reach some lowest level of elementary subsystem. The hierarchic nature of complex system is essential to both their design and their description. The designer need only deal with a particular level of the system at a time. At each level, the system consists of a set of components and their interrelationships. The behavior at each level depends only on a simplified, abstracted characterization of the system at the next lower level. At each level, the designer is concerned with structure and function. Figure 3 shows the seven levels in the computer, from the user level down to the transistor level.

User Level: Application Programs
High Level High Level Languages
Assembly Language / Machine Code
Microprogrammed / Hardwired Control
Functional Units (Memory, ALU, *etc.*)
Logic Gates
Low Level Transistors and Wires

Figure 3: Levels of machines in the computer hierarchy.

As we progress from the top level downward, the levels become less “abstract” and more of the internal structure of the computer shows through. These levels are discussed below.

User or Application-Program Level

The applications layer is the language of the computer as seen by the end user. We are most familiar with the user, or application program level of the computer. At this level, the user interacts with the computer by running programs such as word processors, spreadsheet programs, or games. Here the user sees the computer through the programs that run on it, and little (if any) of its internal or lower-level structure is visible.

High Level Language Level

Anyone who has programmed a computer in a high level language such as C, Pascal, FORTRAN, or Java, has interacted with the computer at this level. Here, a programmer sees only the language, and none of the low-level details of the machine. At this level the programmer sees the data types and instructions of the high-level language, but needs no knowledge of how those data types are actually implemented in the machine. It is the role of the compiler to map data types and instructions from the high-level language to the actual computer hardware. Programs written in a high-level language can be re-compiled for various machines that will (hopefully) run the same and provide the same results regardless of which machine on which they are compiled and run.

Assembly Language/Machine Code Level

This controls the way in which all software uses the hardware layers. Programming with 1s and 0s is tedious and error prone. As a result, one of the first computer programs written was the assembler, which translates ordinary language mnemonics such as MOVE Data, Acc, into their corresponding machine language 1s and 0s. This language, whose constructs bear a one-to-one relationship to machine language, is known as assembly language.

Hardwired Control (Machine layer)

It is the **control unit** that effects the register transfers. It does so by means of **control signals** that transfer the data from register to register, possibly through a logic circuit that transforms it in some way. The control unit interprets the machine instructions one by one, causing the specified register transfer or other action to occur. Hardwired control units have the advantages of speed and component count, but until recently were exceedingly difficult to design and modify.

Functional Unit (Microprogrammed Layer)

The microprogrammed layer interprets the machine language instructions from the machine layer and directly causes the digital logic elements to perform the required operations. The register transfers and other operations implemented by the control unit move data in and out of "functional units. Functional units include internal CPU registers, the Arithmetic Logic Unit (ALU), and the computer's main memory.

Logic Gates (Digital Logic layer)

The level at which any semblance of the computer's higher-level functioning is visible is at the logic gate and transistor levels. It is from logic gates that the functional units are built, and from transistors that logic gates are built. The logic gates implement the lowest-level logical operations upon which the computer's functioning depends.

Physical Layer

At the very lowest level, a computer consists of electrical components such as transistors and wires, which make up the logic gates, but at this level the functioning of the computer is lost in details of voltage, current, signal propagation delays, quantum effects, and other low-level matters.

3.3 MICROPROCESSOR

A computer consists of five functionally independent main parts input, memory, Arithmetic and Logic Unit (ALU), output and control unit. This is illustrated in Figure 4

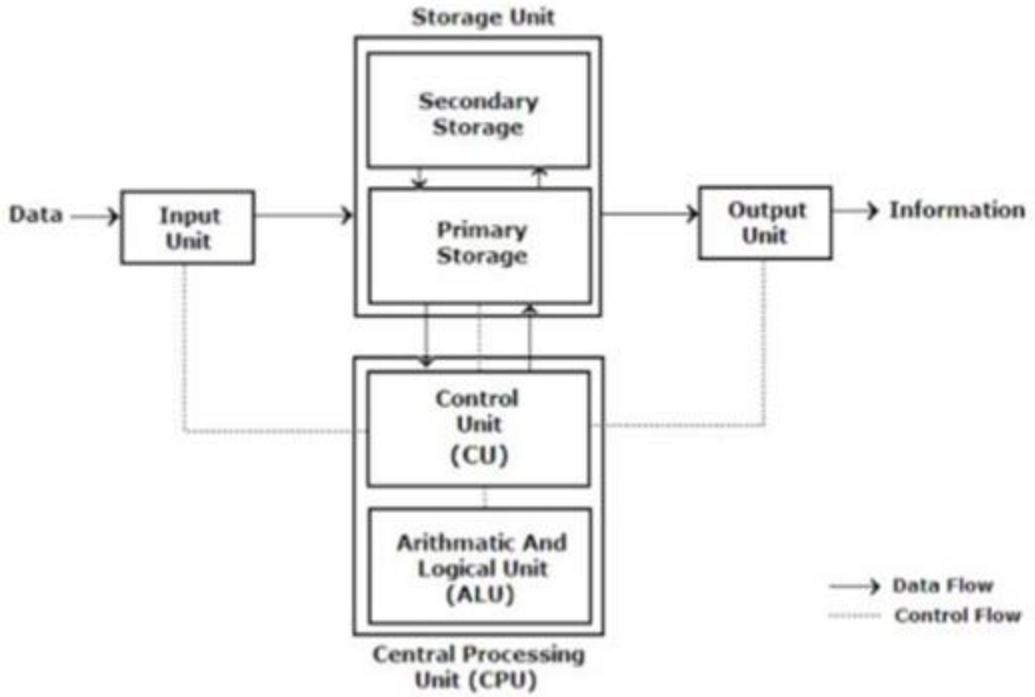


Figure 4: Block Diagram of a Computer

The ALU and the Control unit constitutes the central processing unit. The CPU and some other components connected to it form the microprocessor shown in Figure 5.

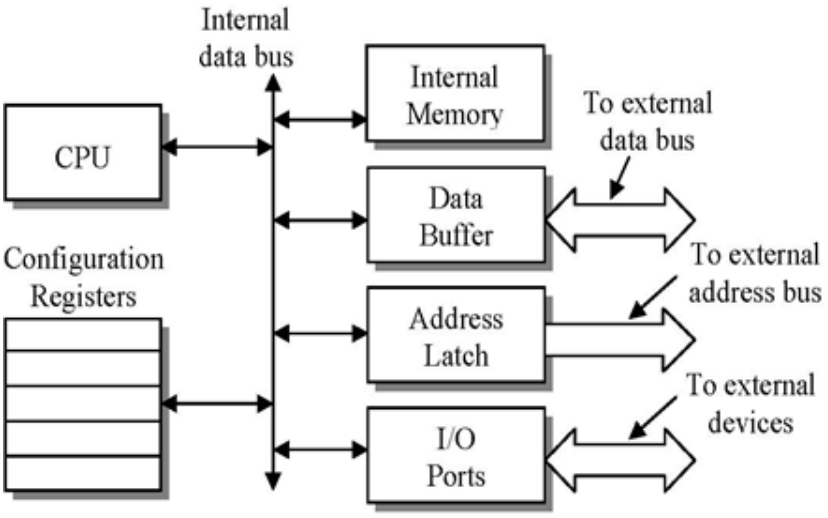


Figure 5: Generic Block Diagram of Processor Internals

The following is a description of the components of the processor.

A. **CENTRAL PROCESSING UNIT (CPU)**

This is the brain of the processor. The execution of all instructions occurs inside the CPU along with the computation required to determine addressing. In most architectures it has three parts: an Arithmetic and Logic Unit (ALU), a Control Unit (CU), and a set of registers (Figure 6).

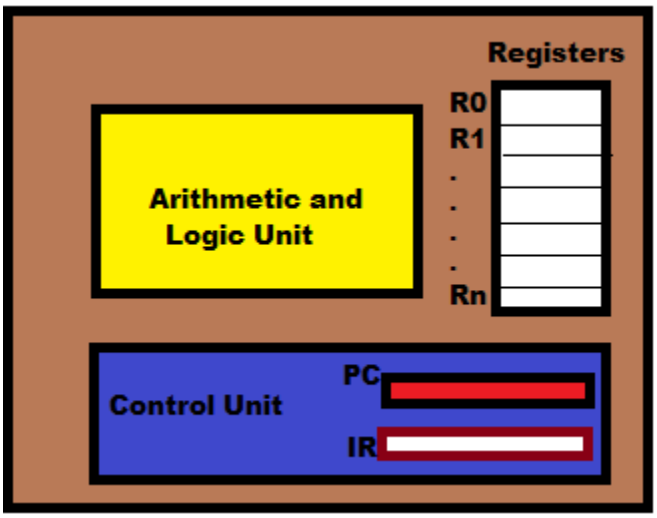


Figure 6: Functional Parts of the Central Processing Unit (CPU)

a. The Arithmetic and Logic Unit

The arithmetic and logic unit performs logic, shift, and arithmetic operations on data.

i. Logic Operations: Logic operations include NOT, AND, OR, NAND, NOR, XOR etc. These operations treat the input data as bit patterns and the result of the operation is also a bit pattern. Detail discussion is in Module 2 Unit 3.

ii. Shift Operations: Shift operations include logical shift and arithmetic shift operations. Logical shift operations are used are used to shift bit patterns to the left or right, while logical arithmetic operations are applied to integers to divide or multiply integers by two. Detail discussion is in Module 2 Unit 3.

iii. Arithmetic Operation: Arithmetic operations involve adding, subtracting, multiplying and dividing integers or floating numbers. More discussion in Module 2 Unit 3

b. Registers

Registers are fast stand-alone storage locations that hold data temporarily. Multiple registers are needed to facilitate the operations of the CPU. These include:

i. Data Registers: Computers use dozens of registers inside the CPU to speed up their operations, because complex operations are done using hardware and this requires several registers to hold the intermediate results. Data registers are named R0 to Rn in Figure 6.

ii. Instruction Registers: Computers not only stores data in their memory but also programs. The CPU is responsible for fetching instructions one by one from the memory, storing them in the instruction register (IR in figure 6), decoding them, and executing them.

iii. Program Counter: Another common register in the CPU is the program counter (PC). The program counter keeps track of the instruction currently being executed. After execution of the instruction, the counter is incremented to point to the address of the next instruction in memory.

c. Control Unit

The third part of the CPU is the control unit and it is the nerve of the computer. The control unit controls the operation of each subsystem. Controlling is achieved through signals sent from the control unit to other subsystems.

B. **INTERNAL MEMORY**

A small, but extremely quick memory. It is used for any internal computations that need to be done fast without the added overhead of writing to external memory. It is also used for storage by processes that are transparent to the applications, but necessary for the operation of the processor.

C. **DATA BUFFER**

This buffer is a bi-directional device that holds outgoing data until the memory bus is ready for it or incoming data until the CPU is ready for it. This circuitry also provides signal conditioning ensuring the output signals are strong enough and the fragile internal components of the CPU are protected.

D. **ADDRESS LATCH**

This group of latches maintains the address that the processor wishes to exchange data with on the memory bus. It also provides signal conditioning and circuit protection for the CPU.

E. **I/O PORTS**

These ports represent the device interfaces that have been incorporated into the processor's hardware.

F. **CONFIGURATION REGISTERS**

A number of features of the processor are configurable. These registers contain the flags that represent the current configuration of the processor. These registers might also contain addressing information such as which portions of memory are protected and which are not.

3.3.1 FUNCTIONS OF THE PROCESSOR

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. The processor controls the input of data and its transfer into main storage, processes data, and then sends the result to the output unit. As already been indicated the processor is connected to other elements of the computer by means of buses. The functions of the processor are:

i. To control the use of main storage to store data and instructions
ii. To control the sequence of operations
iii. To give commands to all parts of the computer system
iv. To carry out processing

3.4 MEMORY SYSTEM

A simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU has been considered. In this simple model, the memory system is a linear array of bytes, and the CPU can access each memory location in a constant amount of time. The memory system is divided into two, the primary memory and the secondary memory. The classifications of computer memory is shown in Figure 7.

1. **Primary memory: -** Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being

executed. The memory contains a large number of semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site called word.

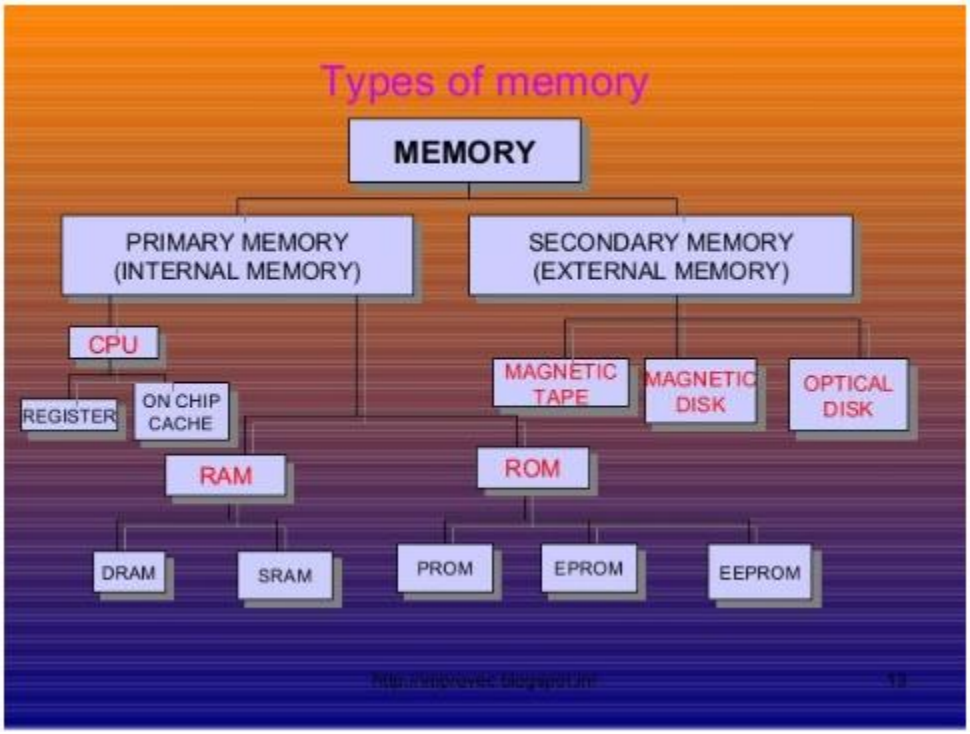


Figure 7: Types of Memory

2. **Secondary memory: -** Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently. Examples: - Magnetic disks & tapes, optical disks (i.e. CD-ROM's), floppies etc.

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as *cores.* The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. The most common semiconductor memory is referred to as *random-access memory* (RAM). A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage.

3.4.1 Memory Hierarchy

In practice, a memory system is a hierarchy of storage devices as illustrated in Figure 8 with different capacities, costs, and access times.

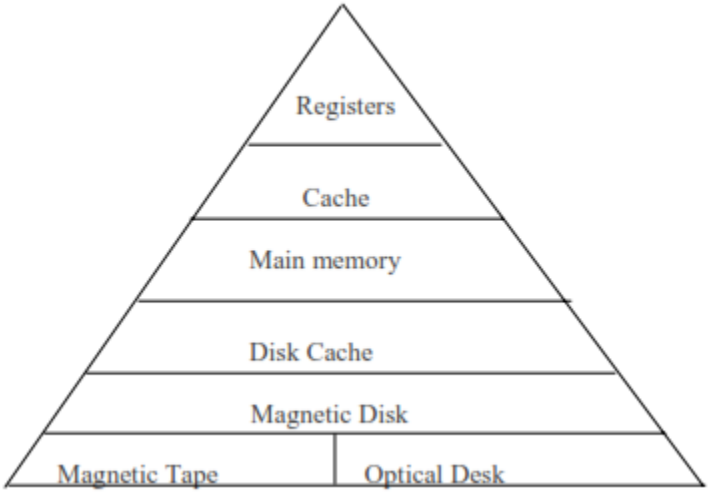


Figure 8: Memory Hierarchy

CPU registers hold the most frequently used data. Small, fast cache memories nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because well-written programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called buses. Each transfer of data between the CPU and memory is accomplished with a series of steps called a bus transaction. A read transaction transfers data from the main memory to the CPU. A write transaction transfers data from the CPU to the main memory.

4.0 CONCLUSION

In this unit, we have discussed the basic computer machine architecture and discussed two models that is the Von Neumann and the system bus models. The hierarchic nature of computer systems, the roles of the central processing unit (CPU) and that of the memory in a computer as support for computer programs were discussed.

5.0 SUMMARY

In this unit we have seen that the parts that make up a computer can be divided into three broad categories or subsystems: the central processing unit (CPU), the memory and the input/output subsystem. The interconnection of the three subsystems plays an important role, because information needs to be exchanged between these subsystems. The CPU and the memory are normally connected by three groups of connections, each called a bus: data bus, address bus, and control bus. The central processing unit performs operations on data. It has three parts: an arithmetic logic unit (ALU), a control unit, and a set of registers. Memory is a collection of storage locations. The two broad types of memory discussed here are the primary memory and the secondary memory. The memory hierarchy of the computer system was also discussed.

6.0 TUTOR-MARKED ASSIGNMENT

1. The basic elements of computer are essentially the same for all digital computers. However, there are variations in construction that reflect the differing ways in which computers are used. Compare and contrast the System Bus and Von Neumann's models in view of discussing the basic architectural features common to all systems.
2. There are a number of levels at which the construction and organisation of computer system is studied. Explain these levels briefly
3. Explain the generic block diagram of computer processor internals
4. Differentiate between the digital level and the physical level of Computer architecture
5. At the top of the memory hierarchy are the registers and caches. Explain these two memory devices

6. Explain the following components of a typical central Processing Unit (CPU)
    i. Control Unit
    ii. Arithmetic Logic Unit
    iii. Instruction Decoder
7. There are five basic ROM types, state their respective features, merits and demerits.

7.0 REFERENCES/FURTHER READINGS

1. William Stallings (2013). *Computer Organization and Architecture: Designing For Performance.* Pearson Education, Inc., publishing as Prentice Hall (9th ed).

2. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).
3. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).
4. Tanenbaum, Andrew S. (1993) *Structural Computer Organisation*. India: Prentice Hall. (3rd ed).

UNIT 2: DATA REPRESENTATION AND STORAGE

CONTENTS

1.0 INTRODUCTION

2.0 OBJECTIVES

3.0 MAIN CONTENTS

3.1 STORING DATA AS BIT PATTERNS

3.2 STORING NUMBERS

3.2.1 Storing Integers

3.2.1.1 Unsigned Representation

3.2.1.2 Overflow

3.2.1.3 Signed Representation

3.2.1.4 Two's Complement Representation

3.2.1.5 Comparison of the three Systems

3.2.2 Storing Reals

3.2.2.1 Fixed-Point Representation

3.2.2.2 Floating-Point Number Representation

3.3 STORING TEXT

3.4 STORING AUDIO

3.5 STORING IMAGES

3.6 STORING VIDEOS

4.0 CONCLUSION

5.0 SUMMARY

6.0 TUTOR-MARKED ASSIGNMENT

7.0 REFERENCES/FURTHER READINGS

1.0 INTRODUCTION

A computer is a programmable data processing machine and since all operations on data must be performed by the computer's hardware, there is need to understand the representations of data

within the computer and also the nature and operations performed on data. Data comes in different forms including numbers, text, audio, image and video. All data types are transformed into a uniform representation when they are stored in a computer and transformed back into their original form when retrieved. This representation is called bit pattern. This unit discusses data representation and storage or computer processing.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- list five different data types used in a computer
- describe how different data is stored inside the computer as bit patterns
- describe how integers are stored in a computer using unsigned, sign-and-magnitude, and two's complement formats
- describe how reals are stored inside the computer using floating-point format
- describe how text is stored in a computer using one of the various encoding systems.
- describe how audio is stored in a computer using sampling technique.
- describe how image is stored in a computer using PIXELS
- describe the basic principle of storing video.

3.0 MAIN CONTENTS

3.1 STORING DATA AS BIT PATTERNS

A bit (binary digit) is the smallest unit of data that can be stored in a computer and has a value of 0 or 1. A bit represents the state of a device that can take one of two states. A convention can be established to represent the 'ON' state as 1 and the 'OFF' state as 0, or vice versa. Computers use various two-state devices to store data.

A bit pattern, a sequence or a string of bits can be used to represent different types of data. With a text editor, the character 'A' typed on the keyboard can be stored as an 8-bit pattern 01000001. The same 8-bit pattern can represent the number 65, a part of an image, part of a song, or part of a video.

To further discuss operations on data, this section discusses how different data types, numbers, text, audio, image and video are stored inside a computer.

3.2 STORING NUMBERS

The numerical data types are coded for storage purposes. The basic types of representation are:

i. Binary Coded Decimal (BCD) Representation: in which each decimal digit in the number is coded separately. BCD is a 4-bit code used for coding numeric values only as follows

Decimal Digit	0	1	2	3	4	5	6	7	8	9
BCD Code	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

For Example, the number 2875 is coded thus:

Decimal Digit	2	8	7	5
BCD Code	0010	1000	0111	0101

ii. Pure Binary Representation: A number is changed into the binary system before being stored in the computer's memory. This type of representation addresses how to store signed and unsigned numbers and also how to show the decimal point. There are several ways the computer handles the sign issue, for the decimal point, computers use two different representations: fixed-point and floating-point. The first is used to store a number as an integer while the second is used to store a number as a real.

3.2.1 Storing Integers

Integers are whole numbers together with negatives. For example . . . –3, –2, –1, 0, 1, 2, 3 . . . are set of integers. An integer can also be thought of as a number in which the position of the decimal point is fixed. Fixed-point representation is used to store an integer. In this representation, the decimal point is assumed but not stored. However, a user or program may store an integer as a real with the fractional part set to zero. To use computer memory more efficiently, unsigned and signed integers are stored inside computer differently.

3.2.1.1 Unsigned Representation

An unsigned integer is an integer that can never be negative and can take only '0' or positive values. A computer with *n*-bit storage location stores an unsigned integer by converting it to binary and then, if the number of bits is less than *n* 0s are added to the left of the binary integer so that there is a total of n bits. If the number of bits is greater than n, the integer cannot be stored. A condition referred to as overflow will occur.

Example 1: Store 7 in an 8-bit memory location using unsigned representation

Solution:

First change the integer to binary then add five 0s to make a total of eight bits

Change 7 to binary	111
Add five bits at the left	00000111

Example 2:

Store 258 in a 16-bit memory location

Solution:

Change 258 to binary	100000010
Add seven bits at the left	0000000100000010

3.2.1.2 Overflow

Due to size limitations, the range of integers that can be represented is limited. In an n-bit memory location only an unsigned integer between 0 and $2^n - 1$ can be stored. In a memory location that can only hold 4-bits, an integer larger than $2^4 - 1 = 15$ cannot be stored. Overflow happens when for example, there is 11 in a 4-bit memory location and then try to add 9 to the integer. The minimum number of bits required to represent the decimal 20 is five bits. In other words, $20 = 10100_2$, so the computer drops the leftmost bit and keeps the rightmost four bits (0100) the result is now presented as 4 instead of 20 due to overflow error.

3.2.1.3 Signed Representation

In this method, the available range for unsigned integers (0 and $2^n - 1$) is divided into two equal halves. The first half represents positive integers, the second half, represents negative integers. For example, if n is 4, the range is 0000 to 1111. This range is divided into two halves: 0000 to 0111 and 1000 to 1111. The bit patterns are then assigned to negative and positive integers (Figure 1).

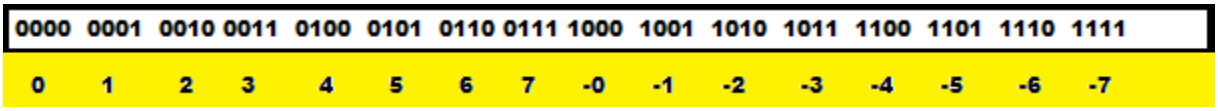


Figure 1: Sign-and-magnitude Representation of 4-bit integers

In signed representation, the leftmost bit defines the sign of the integer. If it is 0, the integer is positive, if it is 1, the integer is negative. There are two 0s in sign-and-magnitude representation: +0 and -0.

Example 3:

a. Store +28 in an 8-bit memory location using sign –and-magnitude representation.
b. Store -28 in an 8-bit memory location using sign –and-magnitude representation.

Solution

a. The integer is changed to 7-bit binary then add 0 as the eighth leftmost bit to make it 8-bit number

Change 28 to 7-bit binary	0011100
Add the sign and store	00011100

b. The integer is changed to 7-bit binary then add 1 as the eighth leftmost bit to make it 8-bit number

Change 28 to 7-bit binary	0011100
Add the sign and store	10011100

3.2.1.4 Two's Complement Representation

Almost all computers use two's complement representation to store a signed integer in an n-bit memory location. T*wo's complement* numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00\ldots000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01\ldots111_2 = 2^{N-1} - 1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10\ldots000_2 = -2^{N-1}$. And -1 is written as all ones: $11\ldots111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the overall number is interpreted differently for two's complement numbers and sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called *taking the two's complement*. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. Another method is to copy bits from the right until a 1 is copied, then invert the rest of the bits. The two's complement representation is useful to find the representation of a negative number or to determine the magnitude of a negative number.

To store an integer in two's complement representation, the computer follows the step below:

i. The integer is changed to an n-bit binary
ii. If the integer is positive or zero, it is stored as it is; if it is negative, the computer takes the two's complement of the integer and then stores it.

To retrieve an integer in two's complement representation, the computer follows the steps below:

i. If the leftmost bit is 1, the computer applies the two's complement operation to the integer. If the leftmost bit is 0, no operation is applied
ii. The computer changes the integer to decimal

Example: 4

a. Store +28 in an 8-bit memory location using two's complement representation.
b. Store -28 in an 8-bit memory location using two's complement representation.

Solution:

a. The integer is positive, so after decimal to binary conversion, no more action is needed. Note that three extra 0s are added to the left of the integer to make it eight bits.

Change 28 to 8-bit binary	00011100

b. The integer is negative, so after decimal to binary conversion, the computer applies the two's complement operation on the integer.

Change 28 to 8-bit binary	0 0 0 1 1 1 0 0
Apply two's complement operation	1 1 1 0 0 1 0 0

Example 5:

Two's Complement Representation of a Negative Number

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution

Start with $+ 2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

Example 6:

Value of Negative Two's Complement Numbers

Find the decimal value of the two's complement number 1001_2.

Solution

1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding *N*-bit numbers, the carry out of the *N*th bit (i.e., the $N + 1^{th}$ result bit) is discarded.

Example 7:

Adding Two's Complement Numbers

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.

Solution

(a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.

Subtraction is performed by taking the two's complement of the second number, then adding.

Example 8:

Subtracting Two's Complement Numbers

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

Solution

(a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

The two's complement of 0 is found by inverting all the bits (producing $11\ldots111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate −0. Zero is considered positive because its sign bit is 0.

Like unsigned numbers, *N*-bit two's complement numbers represent one of 2^N possible values. However the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: −8 to 7. In general, the range of an *N*-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no −0. The most negative number $10\ldots000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01\ldots111_2$) and adding 1, which produces $10\ldots000_2$, the weird number, again. Hence, this negative number has no positive counterpart.

Adding two *N*-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate

overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 9:

Adding Two's Complement Numbers with Overflow

Compute $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution

$4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and −3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively.

3.2.1.5 Comparison of the three Systems

Table 1 shows a comparison between unsigned, two's complement, and sign-and-magnitude integers. A 4-bit memory location can store an unsigned integer between 0 and 15, and the same location can store two's complement signed integers between -8 and +7.

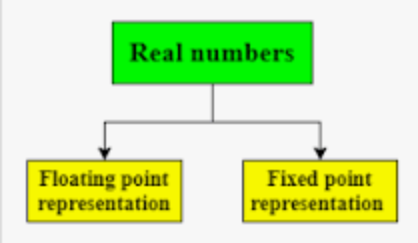
Table 1: Summary of Integer Representation

Contents of Memory	**Unsigned**	**Sign-and-magnitude**	**Two's Complement**
0000	0	0	+0
0001	1	1	+1
0010	2	2	+2

0011	3	3	+3
0100	4	4	+4
0101	5	5	+5
0110	6	6	+6
0111	7	7	+7
1000	8	-0	-8
1001	9	-1	- 7
1010	10	-2	- 6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

3.2.2 Storing Reals

A real is a number that include fractions/values after the decimal point. For example, 231.54 is a real number. Real numbers are represented either as fixed point number representation or floating-point number representation. In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

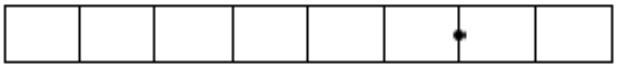


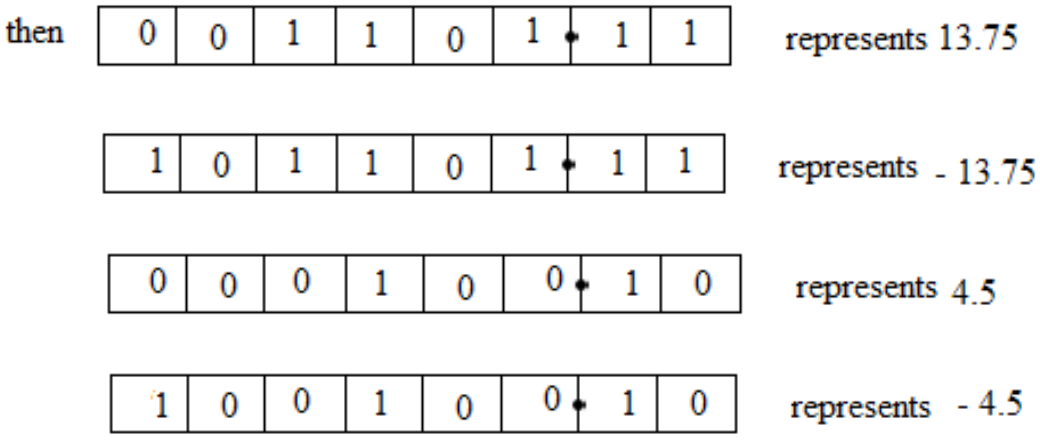
3.2.2.1 Fixed-Point Representation

This representation has fixed number of bits for integer part and for fractional part. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

For a fixed-point number representation the programmer requires a computer-storage location of sufficient size to store all the digits of the number.

For an 8-bit representation, if the programmer assumes the point to be:





In using fixed-point number representation, the result may not be accurate or it may not have the required precision. For example in a decimal system, assume a fixed-point representation with two digits at the right of the decimal point and fourteen digits at the left of the decimal point, for a total of sixteen digits is used. The precision of a real number in this system is lost if a decimal number such as 1.00792 is represented; the system stores the number as 1.00. Also, assume a fixed-point number representation with six digits to the right of the decimal point and ten digits to the left of the decimal point for a total of sixteen digits. The accuracy of a real number in this system is lost if a decimal number such as 631254378943.43 is represented; the system stores the number as 1254378943.43. Therefore, real numbers with very large parts or very small fractional parts should not be stored in fixed-point representation.

3.2.2.2 Floating-Point Number Representation

The solution for maintaining accuracy or precision is to use floating-point representation. This representation allows the decimal point to float that is, there can be different numbers of digits to the left or right of the decimal point. The range of real numbers that can be stored using this method increases tremendously. Numbers with large integer parts or small fractional parts can be stored in memory. In floating-point number representation, a number is made of three section:

sign	shifter	Fixed-point number

The first section is the sign, either positive or negative, the second section shows how many places the decimal point should be shifted to the right or left to form the actual number while the third section is a fixed-point representation in which the position of the decimal is fixed.

To make the fixed part of the representation uniform, the floating-point method uses only one none-zero digit to the left of the decimal point. This is called normalisation. In the decimal system, this digit can be 1 to 9, while in the binary system it can only be 1. After a binary number is normalized, only three pieces of information about the number are stored: sign, exponent, and mantissa. The sign can be stored using 1 bit (0 or 1). The exponent (power of 2) defines the shifting of the decimal point. The power can be negative or positive. A new representation call the Excess system is used to store the exponent. In the Excess system, both the positive and negative integers are stored as unsigned integers. To represent a positive or negative, a positive integer called a bias

is added to each number to shift them uniformly to the non-negative side. The value of this bias is $2^{m-1} - 1$, where m is the size of the memory location to store the exponent. The mantissa is the binary integer to the right of the decimal point. It defines the precision of the number. The mantissa is stored in fixed-point notation. The mantissa is a fractional part that together with the sign, is treated like an integer stored in sign-and-magnitude representation.

The Institute of Electrical and Electronics Engineers (IEEE) has defined several standards for storing floating-point numbers. The two most common are the single precision and double precision formats. The single precision format uses a total of 32 bits to store a real number in floating-point representation. The 'sign' occupies one bit (0 for positive and 1 for negative), the 'exponent' occupies eight bits using a bias o 127, and the mantissa uses twenty-three bits (unsigned number). This standard is referred to as Excess_127 because the bias is 127. Double precision format uses a total of 64 bits to store a real number in floating-point representation. The sign occupies one bit, the exponent occupies eleven bits using a bias of 1023, and the mantissa uses fifty-two bits. The standard is referred to as Excess_1023 because the bias is 1023.

A real number can be stored in one of the IEEE standard floating-point format using the following procedure:

a. Store the sign as either 0 or 1 (depending on if the number is positive or negative)
b. Change the number to binary if not in binary
c. Normalise the number
d. Find the values of exponent and mantissa
e. Concatenate the sign, exponent and mantissa

Example 10:

Show the Excess_127 representation of the decimal number 5.75

Solution

a. The sign is positive, so sign = 0
b. Decimal to binary conversion: 5.75 = 101.11
c. Normalisation: $101.11 = 1.0111 \times 2^2$

d. Exponent = 2 + 127 = 129 = $(10000001)_2$, mantissa = 0111. Nineteen zeroes are added at the right of the mantissa to make it 23 bits
e. The presentation is shown below

Number	**Normalised Value**	**Stored Value**		
		sign	**Exponent**	**Mantissa**
5.75 = $(101.11)_2$	1.0111 x 2^2	0	10000001	01110000000000000000

The number is stored as 01000000101110000000000000000000

Example 11:

Show the Excess_127 representation of the decimal number -161.875

Solution

a. The sign is negative, so sign = 1
b. Decimal to binary conversion: 161.875 = 10100001.111
c. Normalisation: 10100001.111= 1.0100001111 x 2^7
d. Exponent = 7 + 127 = 134 = $(10000110)_2$, mantissa = 0100001111. Thirteen zeroes are added at the right of the mantissa to make it 23 bits
e. The presentation is shown below

Number	**Normalised Value**	**Stored Value**		
		sign	**Exponent**	**Mantissa**
161.875 = 10100001.111	1.0100001111 x 2^7	1	10000110	01000011110000000000000

The number is stored as 11000011001000011110000000000000

Example 12:

The bit pattern $(11001010000000000111000100001111)_2$ is stored in memory in Excess_127 format. Show what the value of the number is in decimal notation.

Solution

a. The first bit represent the sign, the next eight bits represents the exponent, and the remaining 23 bits represents the mantissa;
b. The first bit represent the sign, the next eight bits represents the exponent, and the remaining 23 bits represents the mantissa;
c. The sign is negative;
d. The exponent $(10010100)_2 \equiv 148_{10}$
e. Therefore, the shifter = 148 – 127 = 21
f. Denormalisation gives $1.00000000111000100001111 \times 2^{21}$
g. The binary number is 100000000111000100011.11
h. Conversion from binary to decimal: 100000000111000100011.11 = 2,104,378.75 (absolute value)
i. The number is -2,104,378.75

3.3 STORING TEXT

Character data, sometimes referred to as "string" data, may consist of any digits, letters of the alphabet or symbols which, the internal coding system of the computer is capable of representing. Any sequence of symbols used to represent an idea is therefore referred to as text data type. Different sets of bit patterns have been designed to represent text symbols. Each set is called a code, and the process of representing symbols is called coding. Some popular codes are:

1. ASCII

The American National Standards Institute (ANSI) developed a code called American Standard Code for Information Interchange (ASCII). This code uses seven bit for each symbol. This means that $2^7 = 128$ different symbols can be defined in this code. ASCII codes are widely used throughout the computer industry.

2. EBCDIC

Extended Binary Coded Decimal Interchange Code (EBCDIC) is sometimes called 8-bit ASCII. There are 256 characters in the EBCDIC character set.

3. Unicode

Unicode uses 32 bits and can therefore represent up to 2^{32} = 4,294,967,296 symbols. Different sections of the code are allocated to symbols from different languages in the world. Some part of the code are used for graphical and special symbols.

3.4 STORING AUDIO

Audio is a representation of sound. Sound can also be given a binary coded representation suitable for storage as data in a computer. Audio is an example of analog data. Even if it is possible to measure all its values in a period of time, these values cannot be stored in the computer's memory as an infinite memory location will be needed. If the all the values of an audio signal over an interval cannot be recorded, some of them can be recorded. A finite number of points of the analog audio signal is selected and their values measured and recorded to represent the entire audio signal. The sampling rate determines how well an audio signal is processed, the higher the better. In simple cases the input device is a combination of a microphone and a digitising sound sampler. It is the latter that produces a binary coded representation of the sound picked up by the microphone.

The dominant standard for storing audio is MP3 (short for MPEG layer 3). This standard is a modification of the MPEG (Motion Picture Experts Group) compression method used for video. It uses 44,100 samples per second and 16 bits per sample.

3.5 STORING IMAGES

Computer-generated images can be stored in several different formats and differing resolutions. But all computer files, whether numbers, words or graphics, are stored as digital information. What computers do is translate the image into digital code for storage and then interpret the file back into an image for display. How the computer does this involves the manner in which the image was created and the code and formats needed for making a graphic image file and then creating a graphic image display. Images are stored as 1's and 0's! To store an image on a computer, the image is first broken down into tiny elements called **PIXELS.** The **smallest** element in a picture or image is called Pixel (in short of **Pic**ture **El**ement = **Pixel**). The number of pixels is the product of height and width of an image (In other words, we can say it's calculated using image resolution). If your image resolution is 1020 x 800 (width x height), the total number of pixels is 816,000. Now, for the computer to store the image, each pixel is represented by a binary value.

For every pixel, an average color is found and a binary value is assigned. For monochrome (two-color) image, only 1 bit is needed to represent each pixel. 0 for white and 1 for black. For colored images, each pixel is represented by multiple bits, one combination per shade. The number of bits allocated for each pixel color is called **color depth** or **bit depth** (in simple words, how many bits represent each pixel). If the color depth of an image is 8-bit, the image contains 256 colors. The most common color depths you see are 8-bit (2^8 = 256 colors), 16-bit (2^{16} = 65,536 colors) and 24-bit (2^{24} = 16.7 million colors). Larger color depth allows more shades and different colors. You can find the color depth or bit depth of your image from your image properties.

Computers store graphic information in several formats. Postscript is one. There are also JPEGs (pictures for computer screens), TIFFs (quality images for printing presses), PICTs (simple line drawings) and MOVs (movie video files), among many others. New formats are constantly developed to address growing graphic image needs. Each format uses different types of digital data to represent, store and display the graphic image.

3.6 STORING VIDEOS

Video is a representation of images called frames over time. A movie consists of a series of frames shown one after the other to create the illusion of motion. In other words, video is the representation of information that changes in space (single image) and in time (a series of images). So, if we know how to store an image inside a computer, we also know how to store video: each image or frame is transformed into a set of bit patterns and stored.

Today video is normally compressed using a common video compression technique known as MPEG (Moving Picture Experts Group) format.

4.0 CONCLUSION

In this unit, we have discussed the five different data types used in a computer namely numbers, texts, image, audio and videos. This unit has demonstrated how this data types are represented and stored as bit patterns for processing in a computer.

5.0 SUMMARY

Data comes in different forms, including numbers, texts, image, audio and videos. All data types are transformed into a uniform representation called a bit pattern. A number is changed

to the binary system before being stored in computer memory. There are two ways to handle the decimal point: fixed-point and floating-point. Integers can be taught of as numbers in which the position of the decimal point is fixed: the decimal point is at the right of the least significant bit. An unsigned integer is an integer that can never be negative. One of the methods used to store a signed integer is the sign-and-magnitude format. In this format, the leftmost bit is used to show the sign and the rest of the bits define the magnitude. Sign and magnitude are separated from each other. Almost all computers use the two's complement representation to store a signed integer in an n-bit memory location. In two's complement representation, the leftmost bit defines the sign of the integer, but sign and magnitude are not separated from each other.

A real is number with an integer part and a fractional part. Real numbers are stored in the computer using floating-point representation. In floating-point representation, a number is made up of three segments: a sign, a shifter and a fixed-point number.

Text can also be represented with bit pattern. Different sets of bit patterns (codes) have been designed to represent text symbols; ASCII, EBCDIC, and UNICODE were discussed.

Storage of image is done by scanning and then stored as Picture Elements known as PIXELS.

Video is the representation of images that changes in space and time. Video is stored using MPEG format.

6.0 TUTOR-MARKED ASSIGNMENT

1. Name five types of data that a computer can process.
2. How is bit pattern length related to the number of symbols the bit pattern can represent?
3. What steps are needed to convert audio data to bit pattern?
4. Compare and contrast the representation of positive integers in unsigned, sign-and-magnitude format and two's complement format.
5. Compare and contrast the representation of negative integers in sign-and-magnitude format and two's complement format.
6. Compare and contrast the representation of zero sign-and-magnitude two's complement, and Excess formats.
7. Find the representation of -4_{10} as a 4-bit two's complement number.
8. Compute (a) $-5_{10} + 3_{10}$ and (b) $-6_{10} + 6_{10}$ using two's complement numbers.

9. Store +20 in an 8-bit memory location using sign –and-magnitude representation.
10. Store -20in an 8-bit memory location using sign –and-magnitude representation.
11. The bit pattern $(11001010000000000111000100011011)_2$ is stored in memory in Excess_127 format. Show what the value of the number is in decimal notation.
12. Show the Excess_127 representation of the decimal number -161.875.

7.0 REFERENCES/FURTHER READING

1. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2^{nd} ed).
2. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5^{th} ed).
3. Hamer, F., Horan R. & Lavelle, M. (2005). *Basic Engineering: Binary Numbers 2.*

UNIT 3: OPERATIONS ON DATA

CONTENTS

1.0 INTRODUCTION
2.0 OBJECTIVES
3.0 MAIN CONTENTS
3.1 LOGIC OPERATION
3.1.1 Logic Gates
3.1.2 Logic Operations at Pattern Level
3.1.3 Applications of Logic Operation
3.2 SHIFT OPERATIONS
3.2.1 Logical Shift Operations
3.2.2 Arithmetic Shift Operations
3.3 ARITHMETIC OPERATIONS
3.3.1 Binary Addition
3.3.2 Binary Subtraction
3.3.3 Binary multiplication
3.3.4 Binary Division
3.3.5 Two's Complement Arithmetic
4.0 CONCLUSION
5.0 SUMMARY
6.0 TUTOR-MARKED ASSIGNMENT
7.0 REFERENCES/FURTHER READINGS

1.0 INTRODUCTION

After the discussion on data representation and storage, it is time to discuss the operation on stored data. The focus of this unit is the operation of the arithmetic logic unit (ALU) of the central processing unit (CPU). Operations on data can be divided into three broad categories: logic operations, shift operations and arithmetic operations. Therefore, this unit discusses the logic, shift and arithmetic operations of the ALU.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- list the three operations performed on data
- perform unary and binary logic operations on bit patterns
- distinguish between logic shift operations and arithmetic shift operations
- perform logic shift operations on bit patterns
- perform addition and subtraction on integers stored in two's complement format
- perform addition and subtraction on integers stored in sign-and-magnitude format
- explain some application of logical and shift operations such as setting, unsetting and flipping specific bits.

3.0 MAIN CONTENTS

3.1 LOGIC OPERATION

In the previous chapters, it was explained that data and instructions are coded and stored in binary form. This chapter explains how the computer handles binary data and instructions at the digital logic level of the machine. Logic operations refer to those operations that apply the same basic operation on individual bits of a pattern, or on two corresponding bits in two patterns.

Data and instructions are transmitted between the various parts of the processor or between the processor and the peripherals by mean of pulse trains. Various tasks are performed by passing pulse trains through electronic switches called gates. Each gate is an electronic circuit that may have provision for receiving or sending several pulses at once. Each gate normally performs some simple function like AND, OR, NOT, etc.

3.1.1 Logic Gates

In logic operations, gates are represented by symbols and inputs and outputs are represented by arrowed lines labelled by letters. There are several ways of representing logic functions:

– Symbols to represent the gates;

– Truth tables defines the values of the output for each possible inputs; and

– Boolean algebra to manipulate bits in order to minimize the number of gates needed for a logic operation.

The basic gates are:

1. The NOT gate is a unary operator. It takes only one input and the output is the complement of the input. A NOT gate is also called an 'inverter'

NOT Gate

Symbol

Truth-table

a	y
0	1
1	0

Boolean

$y = \overline{a}$

y is only TRUE if *a* is FALSE

Circle (or 'bubble') on the output of a gate implies that it as an inverting (or complemented) output.

2. The AND gate is a binary operator. It takes two inputs and the output is 1 if both inputs are 1s otherwise the output is 0. In Boolean algebra AND is represented by a dot**.**

AND Gate

Symbol

Truth-table

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Boolean

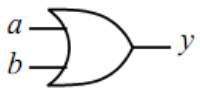
$y = a.b$

y is only TRUE only if *a* is TRUE and *b* is TRUE

3. The OR gate is also a binary operator. The output is I if either of the input is 1 or both inputs are 1s. The output is 0 only when both inputs are 0s. In Boolean algebra OR is represented by a plus sign +

OR Gate

Symbol



Truth-table

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

Boolean

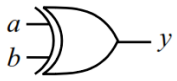
$y = a + b$

• *y* is TRUE if *a* is TRUE or *b* is TRUE (or both)

4. The XOR gate is also a binary operator. The XOR gate can be viewed as a comparator testing non-equivalence of the inputs. If the two inputs are the same the output of XOR gate will be 0 but if they are different, the output will be 1. In Boolean algebra XOR is represented by a $\oplus$ sign

EXCLUSIVE OR (XOR) Gate

Symbol



Truth-table

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Boolean

$y = a \oplus b$

• *y* is TRUE if *a* is TRUE or *b* is TRUE (but not both)

5. The NAND gate in the complement of the AND gate. The output is 1 if either or both inputs are 0 otherwise, the output will be 0.

NOT AND (NAND) Gate

Symbol

Truth-table

a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

Boolean

$y = \overline{a.b}$

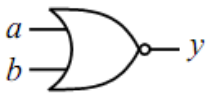
• *y* is TRUE if *a* is FALSE or *b* is FALSE (or both)

• *y* is FALSE only if *a* is TRUE and *b* is TRUE

6. The NOR gate is the complement of the OR gate. The output is 0 if either or both inputs are 1 otherwise, the output will be 1.

NOT OR (NOR) Gate

Symbol



Truth-table

a	b	y
0	0	1
0	1	0
1	0	0
1	1	0

Boolean

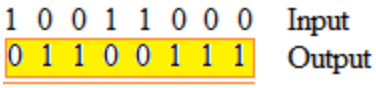
$y = \overline{a + b}$

• *y* is TRUE only if *a* is FALSE and *b* is FALSE

• *y* is FALSE if *a* is TRUE or *b* is TRUE (or both)

3.1.2 Logic Operations at Pattern Level

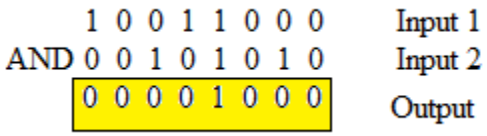
The operators can be applied to n-bit pattern. The same properties are exhibited by the operators as when they are applied to individual bit. For example, the use of NOT operator on the bit pattern 1 0 0 1 1 0 0 0 as input will produce:



Example 1:

Use the AND operator on the bit patterns 1 0 0 1 1 0 0 0 and 0 0 1 0 1 0 1 0.

Solution



3.1.3 Applications of Logic Operation

1. Complementing: Logic operation can be applied to complement bit pattern. Applying a NOT operator to a pattern or bits changes every 0 to 1 and vice versa. This is sometimes referred to as one's complement.
2. Unsetting Specific Bits: Logic operation can be performed to unset specific bit in a pattern. Applying an AND gate to specific bit in a pattern can unset the bit. To unset any bit, 0 bits are applied to the specific bits. The second input in this case is called a mask. The 0 bit in the mask unset the corresponding bits in the first input, the 1 bit in the mask leave the corresponding bits in the first input unchanged.

   Example 2:

   Use a mask to unset the five leftmost bits of a pattern 1 0 1 0 0 1 1 0.

   Solution

   The mask is 0 0 0 0 0 1 1 1. The result of applying the mask is:

   Note that the rightmost bits are unchanged due to 1 bits mask.
3. Setting Specific Bits: Logic operation can be performed to set specific bit in a pattern. This means that a bit is forced to 1. Applying an OR gate to specific bit in a pattern can set the bit. To set any bit, 1 bits are applied to the specific bits. The second input is also a mask.

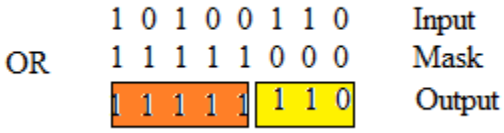
The 1 bit in the mask set the corresponding bits in the first input, the 0 bit in the mask leave the corresponding bits in the first input unchanged.

Example 3:

Use a mask to set the five leftmost bits of a pattern 1 0 1 0 0 1 1 0.

Solution

The mask is 1 1 1 1 1 0 0 0. The result of applying the mask is:



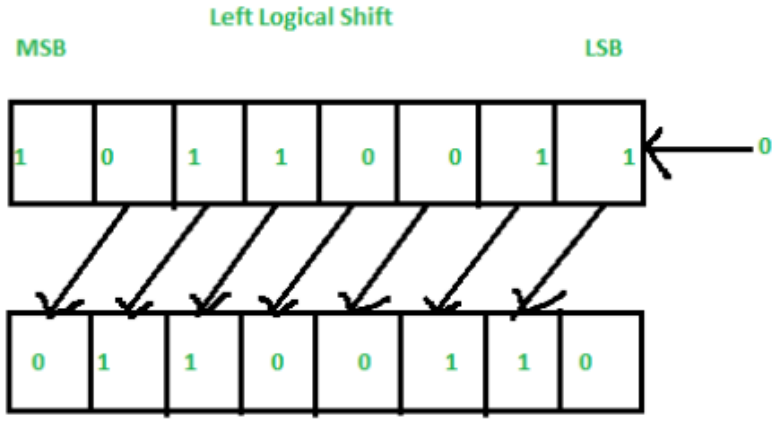
3.2 SHIFT OPERATIONS

Shift operations move the bits in a pattern, changing the positions of the bits. This operation can move bits to the left or to the right. There are two categories of shift operations these are logical shift operations and arithmetic shift operations.

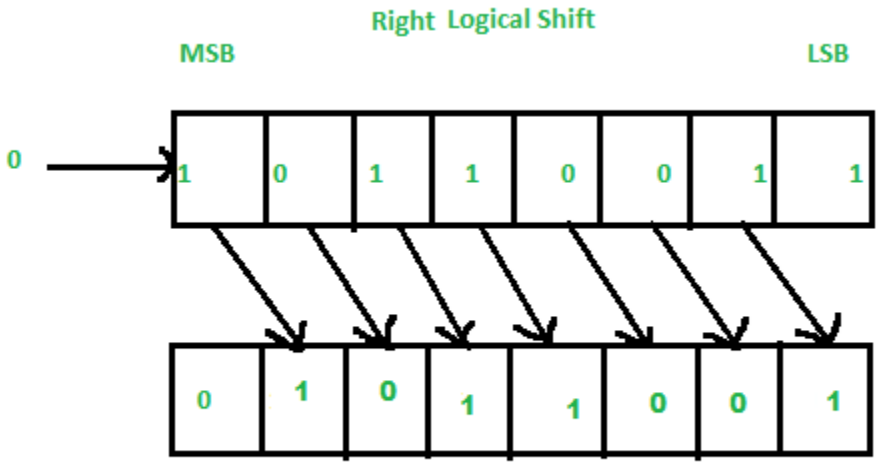
3.2.1 Logical Shift Operations

These operations are used for serial transfer of information. A logical shift operation is applied to a pattern that does not represent a signed number. There are three types of logical shift operations, these are logical shift left, logical shift right and circular shift

A. **Logical Shift Left:** Logical shift left operation moves each bit to the left one by one. The Empty least significant bit (LSB) is filled with zero and the most significant bit (MSB) is rejected.

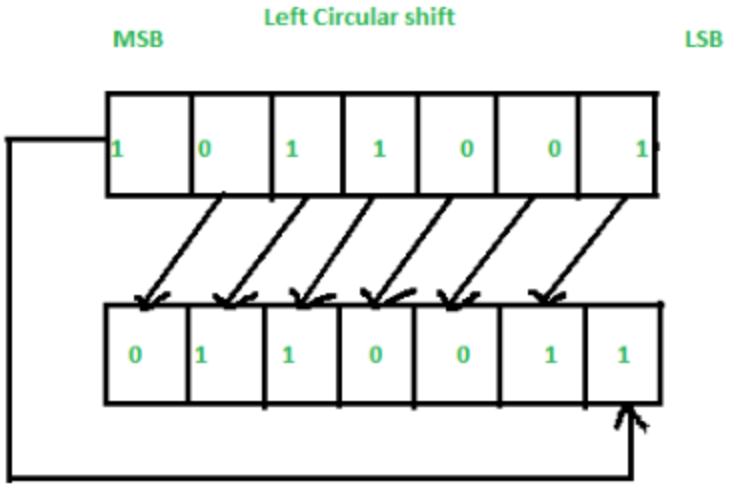


B. **Logical Shift Right:** Logical shift right operation moves each bit to the right one by one and the least significant bit (LSB) is rejected and the empty MSB is filled with zero.

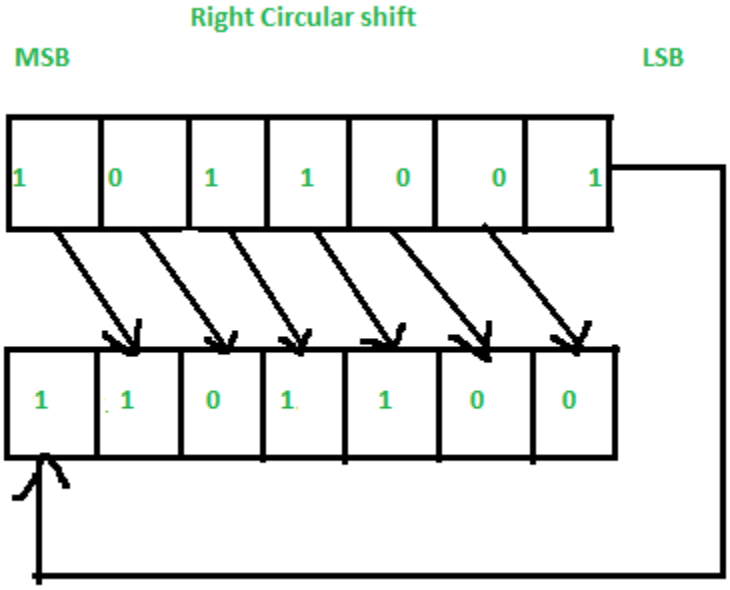


C. Circular shift: The circular shift circulates the bits in the sequence of the register around the both ends without any loss of information. There are two types:

   i. **Circular Left Shift:** This shifts each bit one position to the left. The leftmost bit circulates and become the rightmost bit.



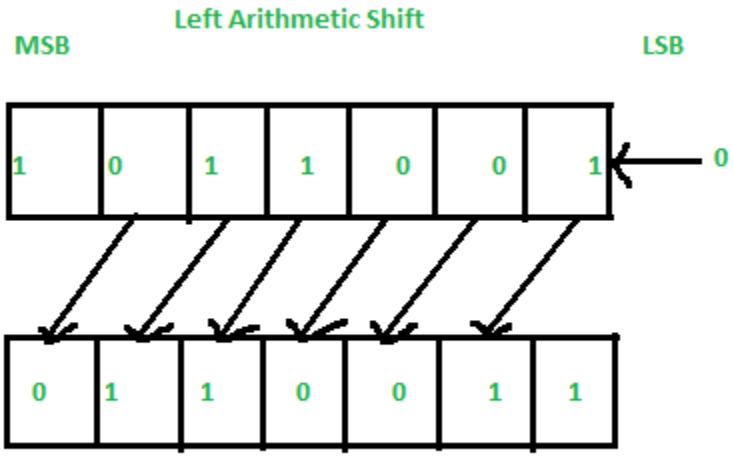
ii. **Circular Right Shift:** This shifts each bit one position to the right. The rightmost bit circulates and become the leftmost bit.



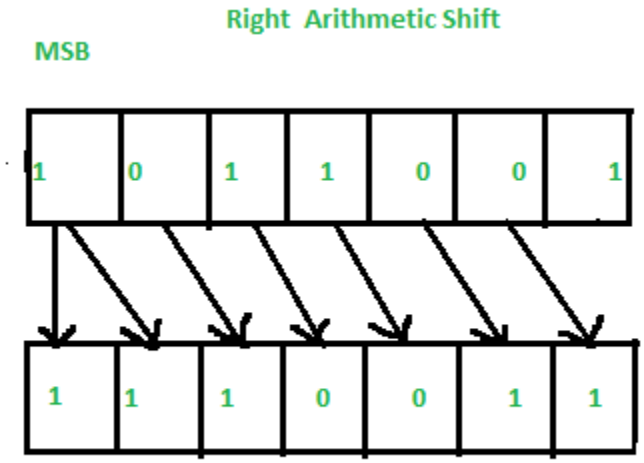
3.2.2 Arithmetic Shift Operations

Since the logical shifts do not work for signed numbers, there is another kind of shifts called arithmetic shifts for signed numbers. An arithmetic shift operation shifts a signed binary number to the left or to the right position. In an arithmetic shift-left, it multiplies a signed binary number by 2 and in an arithmetic shift-right, it divides the number by 2.

a) Arithmetic Left shift: This operation works just like logical shift left, the only difference is that it deals with signed numbers. As long as the sign bit is not changed by the shift, the result will be correct (i.e., will be multiplied by 2). Arithmetic Left shift operation moves each bit to the left one by one. The empty least significant bit (LSB) is filled with zero and the most significant bit (MSB) is rejected. Same as the Left Logical Shift.



b) Arithmetic Right shift: This operation does NOT shift the sign bit: the new bits entering on the left are copies of the sign bit. Arithmetic Right shift operation moves each bit to the right one by one and the least significant bit is rejected and the empty MSB is filled with the value of the previous MSB.

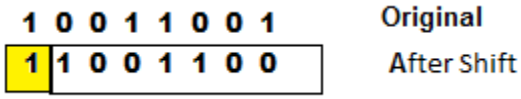


Example 4:

Use an arithmetic right shift operation on the bit pattern 1 0 0 1 1 0 0 1. The pattern is an integer in two's complement format.

Solution

The solution is shown below:



The leftmost bit is retained and also copied to its right neighbor bit.

3.3 Arithmetic Operations

Arithmetic operations involve adding, subtracting, multiplying, and dividing. These operations are applied to integers and floating-point numbers.

3.3.1 Binary Addition

Binary addition is performed in the same way as addition in the decimal-system and is, in fact, much easier to master. Assuming binary addition is performed between two variables, say X and Y, since both X and Y can take only the role 0 and 1, the possible input and output combinations may be arranged as follows:

0 + 0 = 0

0 + 1 =1

1 + 0 = 1

1 + 1 = 1 0

This table represents a standard binary addition, except for the last entry. When both' X and Y represents 1"s, the value of X + Y is 1 0. Put into words, the last rule states that binary one + binary one = binary two = binary "one zero"

3.3.2 Binary Subtraction

Binary subtraction is just as simple as addition. Subtraction of one bit from another obey the following four basic rules

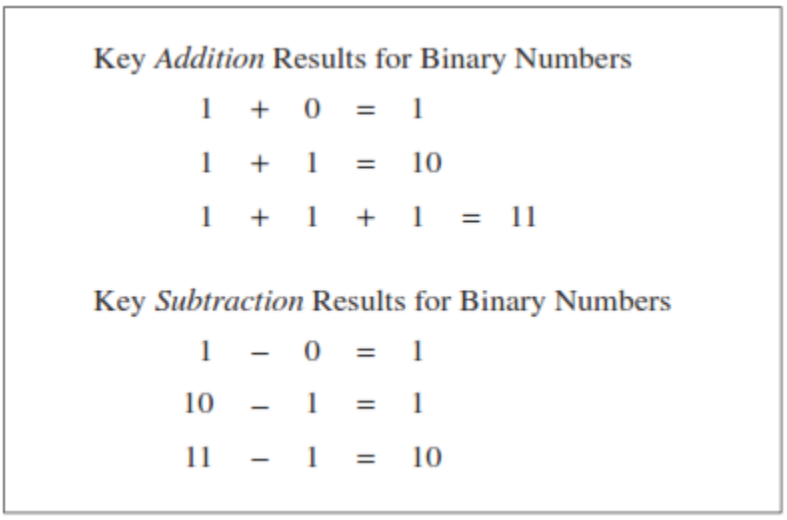
0 – 0 = 0

1 – 1 =0

1 – 0 = 1

10 – 1 = 1 with a transfer (borrow) of 1.

When performing subtraction, it is sometimes necessary to borrow from the next higher-order column. It will be necessary to borrow when a 1 is to be subtracted from a 0. In this case a 1 is borrowed from the next higher-order column, which leaves a 0 in that column.



3.3.3 Binary multiplication

Binary multiplication is performed in the same manner as decimal multiplication. It is much easier, since there are only two possible results of multiplying two bits. The Binary multiplication obeys the four basic rules.

$0 \times 0 = 0$

$0 \times 1 = 0$

$1 \times 0 = 0$

$1 \times 1 = 1$

Example 5:

Multiply the following binary numbers.

(a) $101 \times 1\ 1$

(b) 1101×10

(c) 1010 × l 0 1

(d) 1 0 1 1 × 1 0 1 0

Solution:

(a)
```
   101
   1 1 ×
  ----
   101
  101
 ----
 1111
```

(b)
```
  11101
    1 0 ×
  -----
   0000
  1101
 -----
 11010
```

(c)
```
   1010
    101 ×
  -----
   1010
  0000
 1010
 ------
 110010
```

(d)
```
    1011
    1010 ×
  -----
    0000
   1011×
  0000×
 1011×
 -------
 1101110
```

Multiplication of fractional number is performed in the same way as with fractional numbers in the decimal numbers.

Example 6:

Perform the binary multiplication 0.01 × 11.

Solution:

```
    0.01
  1 1 ×
  ------
    01
  01×
  ------
  0.11
```

3.3.4 Binary Division

Division in the binary number system employees the same procedure as division in the decimal system, as will be seen in the following examples.

Example 7:

Perform the following binary division

(a) 110 ÷ 11

(b) 1100 ÷ 11

Solution

(a)

```
      10
   ------
11 | 110
     11
     ---
      0 0
      0 0
     ---
      0 0
     ---
```

(b)

```
      100
   ------
11 | 11000
     11
     ---
      0 0
      0 0
     ---
        0 0
        0 0
       ---
          0 0
```

Binary division problems with remainders are also treated the same as in the decimal system, as illustrates the following example.

Example 8:

Perform the following binary division:

(a) 1111 ÷ 110

(b) 1100 ÷ 101

Solution

(a)

```
      10. 1
    ---------
110 | 1 11 1.00
      1 10
      ----
        1 10
        1 10
        ----
        0 0 0
```

(b)

```
      10. 011
    ----------
110 | 1 100.00
      1 01
      ----
        100
        000
        ----
         1000
          101
          ---
           110
           101
           ---
             1
```

3.3.5 Two's Complement Arithmetic

The issue of representing integers as binary strings in a computer as discussed in previous chapter is based on two's complement by most computers. One of the advantage of two's complement representation is that there is no difference between addition and subtraction. When the subtraction operation is encountered, the computer simply changes it to an addition operation, but makes two's complement of the second number. In other words:

$A - B \equiv A + (\bar{B} + 1)$

Where $(\bar{B} + 1)$ is the two's complement of B.

This means only addition is needed for discussion. Adding numbers in two's complement is like adding the numbers in decimal. The addition is done column by column, and if there is a carry, it is propagated to the next column. However, the carry is discarded in the last column.

Example 9:

Two integers A and B are stored in two's complement format. Show how B is added to A and the result stored in R.

A = 0 0 0 1 0 0 0 1 B = 0 0 0 1 0 1 1 0

Solution

The operation is adding. A is added to B and the result is stored in another location R.

```
      1              Carry
  0 0 0 1 0 0 0 1    A
+ 0 0 0 1 0 1 1 0    B
  0 0 1 0 0 1 1 1    R
```

Check: (+17) + (+22) = +39

Example 10:

Two integers A and B are stored in two's complement format. Show how B is added to A and the result stored in R.

A = 0 0 0 1 1 0 0 0 B = 1 1 1 0 1 1 1 1

Solution

The operation is adding. A is added to B and the result is stored in another location R.

```
1 1 1 1 1          Carry
  0 0 0 1 1 0 0 0  A
+ 1 1 1 0 1 1 1 1  B
  0 0 0 0 0 1 1 1  R
```

Check: (+24) + (-17) = +7

Example 11:

Two integers A and B are stored in two's complement format. Show how B is subtracted from A and the result stored in R.

A = 0 0 0 1 1 0 0 0 B = 1 1 1 0 1 1 1 1

Solution

The operation is subtraction. A is added to $(\overline{B} + 1)$ and the result is stored in another location R.

The two's complement of B is 0 0 0 1 0 0 0 1

Therefore, the operation is thus:

			1					Carry
	0	0	0	1	1	0	0	0 A
+	0	0	0	1	0	0	0	1 $(\overline{B}+1)$
	0	0	1	0	1	0	0	1 R

Check: (+24) + (-17) = +7

4.0 CONCLUSION

The arithmetic logic unit of the central processing unit performs three operations on data. These operations are logic operations, shift operations and arithmetic operations. The differences among these operations and how the different operations are performed were demonstrated through solved examples. Also, the applications of logical and shift operations such as setting, unsetting and flipping specific bits were explained.

5.0 SUMMARY

Operations on data can be divided into three broad categories: logic operations, shift operations and arithmetic operations. Logic operations refer to those operations that apply the same basic operation to individual bits of a pattern or to two corresponding bits in two patterns. Shift operations move the bits in the pattern. Arithmetic operations involve adding, subtracting, multiplying, and dividing.

6.0 TUTOR-MARKED ASSIGNMENT

1. What is the difference between logic operations and arithmetic operations?
2. What is the difference between unary operations and binary operations?
3. Explain when the results of the following logic gates are true:
    i. **AND**
    ii. **OR**
    iii. **XOR**
    iv. **NOR**
7. Use the AND operator on the bit patterns 1 0 0 1 1 1 0 0 and 0 0 1 0 1 0 1 1.
8. Use a mask to unset the five leftmost bits of a pattern 1 0 1 0 0 1 0 0.
9. Use a mask to set the five leftmost bits of a pattern 1 0 1 0 0 1 0 0.
10. Use an arithmetic right shift operation on the bit pattern 1 0 0 0 0 1 1 1. The pattern is an integer in two's complement format.
11. Two integers A and B are stored in two's complement format. Show how B is added to A and the result stored in R.
12. Two integers A and B are stored in two's complement format. Show how B is subtracted from A and the result stored in R.

7.0 REFERENCES/FURTHER READINGS

1. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).
2. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).
3. Hamer, F., Horan R. & Lavelle, M. (2005). *Basic Engineering: Binary Numbers 2*

UNIT 4: MACHINE INSTRUCTION

CONTENTS

1.0 INTRODUCTION
2.0 OBJECTIVES
3.0 MAIN CONTENTS
3.1 INSTRUCTION FORMAT
3.2 ADDRESS FORMAT
3.2.1 Three-address machines
3.2.2 Two-address machines
3.2.3 One address machine (Accumulator machines)
3.2.4 Zero-address machines (stack machines)
3.3 Instruction Cycle
3.3.1 Fetch cycle
3.3.2 Decode instruction cycle
3.3.3 Execute Cycle
4.0 CONCLUSION
5.0 SUMMARY
6.0 TUTOR-MARKED ASSIGNMENT
7.0 **REFERENCES/FURTHER READINGS**

1.0 INTRODUCTION

This unit concentrates on computer machine language and consequently covers several aspects of machine operations. The purpose of this unit is to demonstrate the various features of machine language. The unit also explains how a computer can carry out instructions presented to it in machine language. It is believe that the basic programming skill acquired in module 1 will be supplemented in this unit.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- explain the composition of machine language instruction format
- explain the part of the instruction format that deals with specifying the address of operands that is the address format
- explain the different methods of specifying the address format
- discuss how to implement the high level statement in the different methods address format
- explain the cycle through which the processor executes each instruction.

3.0 MAIN CONTENTS

3.1 INSTRUCTION FORMAT

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. A machine instruction has several components, this is illustrated in Figure 1. The instruction format is the size and arrangement of these components. Two major components are the function code also called ‘opcode’, which specifies the function or operation performed, and the operand addresses, which specify the locations of the operands used.

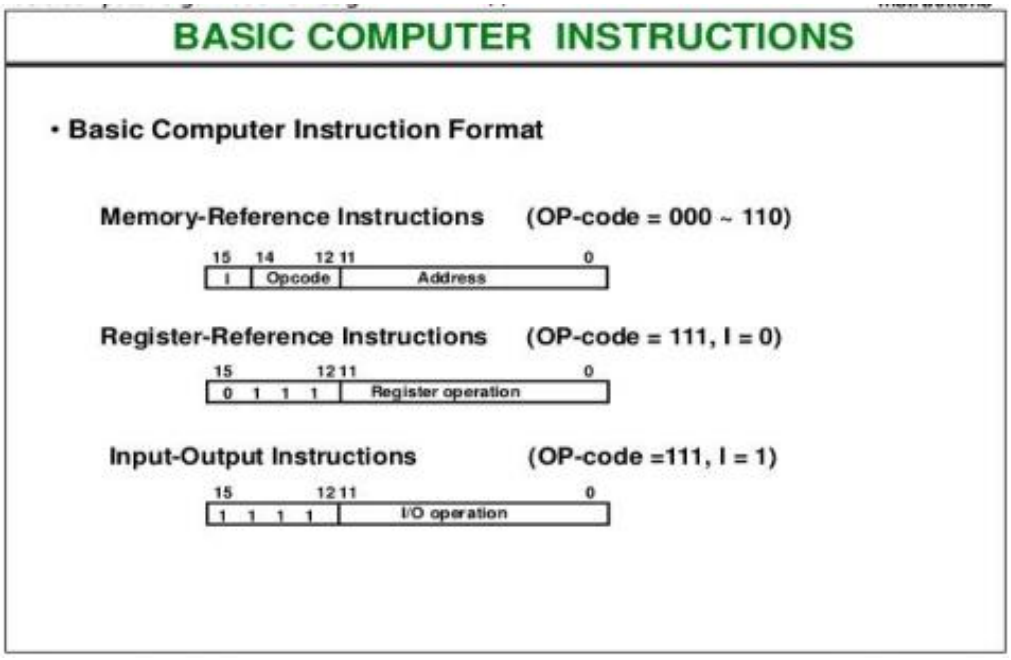


Figure 1: Basic Computer Instruction Format

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. Operations specified by computer instructions are executed on some data stored in memory or processor registers, Operands residing in processor registers are specified with a register address. Each instruction initiates a sequence of micro operations that fetch operands from registers or memory, possibly perform arithmetic, logic, or shift operations, and store results in registers or memory

Consider the following statement in a high level language (C for instance):

a = a + b + a * c;

The meaning of this statement is: take the value of ‘a’ and multiply it with ‘c’, then add ‘a’ and ‘b’ to the result; the result is assigned to the variable ‘a’.

The sequence of operations (single statement per operation) that evaluates the statement is:

t = a * c;

a = a + b;

a = a + t;

It is important to note that, in order to understand what the above sequence does, one has to know how sequencing works that is execution of instructions one after the other and what every operation

does. The above sequence points out why computers are said to be sequential. In the example above, ‘t’ stands for temporary, an intermediate variable that holds the value of the first operation; in this case the value of multiplication a * c could not be assigned to ‘a’ because ‘a’ is also needed for the second operation.

In executing the above statement, the question is to know which operation is to be performed first. The spontaneous and yet not very correct answer is multiplication; it is not simple multiplication because the statement specify not only the operation to be performed but also where the result is to be stored (this is a computer statement, not a simple mathematical equality). The proper name is therefore multiply_and_store, while for the second statement the proper name would be add_and_store. Multiplication and addition are binary operations; multiply_and_store and add_and _store are ternary operations.

The operands which specify the values for the binary operation (which is a part of the ternary operation) are called source operands. The operand that specifies where the result is to be stored is called the destination operand.

Operands may be constants like in the following example:

a = b * 4

where the value in operand b is multiplied with 4 (the other operand) and to assign the result to the operand a. However, in most cases generic names are used because the value is not known but where it is stored that is its address.

3.2 ADDRESS FORMAT

This is the part of the instruction format that deals with specifying the address of operands. The main methods are:

- 3-address machines;
- 2-address machines;
- 1-address machines;
- 0-address machines.

In an n-address machine the maximum number of operands is n. The convention is that the destination is the first operand in the instruction. This a commonly used convention though not generally accepted. It is consistent with the assignment statements in high level languages. The

other used convention, listing the destination after the source operands, is coherent with our verbal description of operations.

3.2.1 Three-address machines

In a 3-address machine all three operands are explicit in each instruction. Each instruction specifies the address of two operands and gives a further address for the results of the operation

The general format of an instruction is:

operation dest, op1, op2

where:

• operation is the name of the operation to be performed;

• dest is the destination operand, the place where the result will be stored;

• op1 and op2 are the two source operands.

Thus the meaning of:

ADD r2, r1, r0 is to add the value stored in register r1, with the value stored in register r0, and put the result in the register r2.

In the example above all operands were held in registers the reason for discussing the addresses is that registers can be seen as a special part of the memory, very fast and very close to the CPU;

Example 1:

What is the meaning of the following instruction?

ADD x, y, z

Solution

Add the value of variable y to the value of variable z and then store the result in the memory location corresponding the variable x.

3.2.2 Two-address machines

Each instruction specifies the address of two operands. The result of the operation would replace one of the two operands. It is an improvement over the three-address machine.

The general format of instructions is:

operation dest, op

where:

• operation is the name of the operation to be performed

• dest designates the name of one source operand and the name of the destination

• op is the name of the second source operand

Thus the meaning of an instruction like:

ADD r1, r2

is to add the values stored in the registers r1 and r2, and to store the result r1.

There is an advantage in having two-address instructions as compared with three-address instructions; the instructions are shorter, which is important when preserving memory. Moreover shorter instructions might be fetched faster (in the case instructions are wider than the data-path, multiple accesses to memory are required). There is a drawback however with two-address instructions: one of the source operands is destroyed; as a result extra moves are sometimes necessary to preserve the operands that will be needed later.

Example 2:

Show how to implement the high level statement

a = a + b + a * c on a 3-address machine and then on a 2-address machine. Both machines are 8 bit register-register machines with 32 general purpose registers and a 16 bit addresses. The values of variables a, b, and c are stored in r1, r2 and r3 respectively. In any case calculate the number of clock cycles necessary if every memory access takes two clock cycles and the execution phase of an instruction takes one clock cycle.

Solution

For the 3-address machine:

MUL r4, r1, r3 # 3 * 2 + 1 clock cycles

ADD r1, r1, r2 # 3 * 2 + 1

ADD r1, r1, r4 # 3 * 2 + 1

This sequence requires 21 clock cycles to complete; each instruction has a fetch phase that takes three (3 bytes/instruction) times two clock cycles (2 clock cycles per memory access), plus an execution phase which is one clock cycle long.

For the 2-address machine:

MOV r4, r1 # 2 * 2 + 1 clock cycles

MUL r4, r3 # 2 * 2 + 1

ADD r1, r2 # 2 * 2 + 1

ADD r1, r4 # 2 * 2 + 1

The sequence requires 20 clock cycles to complete; it is slightly faster than the implementation of the same statement on the 3-address machine. The two address machine requires 10 bits (5 + 5) to encode the two operands and the example assumes an instruction is 16 bit wide.

3.2.3 One address machine (Accumulator machines)

In a 1-address machine the accumulator is implicitly both a source operand and the destination of the operation. The instruction has only to specify the second source operand. The format of an instruction is:

operation op

where:

• operation is the name of the operation to be performed

• op is a source or a destination operand. Example of source or destination operand is the accumulator.

Thus the meaning of:

ADD a

is to add the value of variable a to the content of the accumulator, and to leave the result in the accumulator. The accumulator is a register which has a special position in hardware and in software. Instructions are very simple and the hardware is also very simple.

Example 3:

Show how to implement the statement

a = a + b + a * c using an accumulator machine.

Answer:

LOAD a # bring the value of a in accumulator

MUL c # a * c

```
ADD  b      # a * c + b
ADD  a      # a * c + b + a
STO  a      # store the final result in memory
```

Due to its simplicity, only one operand has to be explicitly specified, accumulator machines present compact instruction sets. The problem is that the accumulator is the only temporary storage: memory traffic is at the highest for accumulator machines compared with other approaches.

3.2.4 Zero-address machines (stack machines)

How is it possible to have a machine without explicit operands instructions? This is possible if the locations of the operands and the result to be stored are known. A stack is a memory (sometimes called LIFO = Last-In-First-Out) defined by two operations PUSH and POP: PUSH moves a new item from the memory into the stack while POP gets the last item that was pushed into the stack. The formats of operations on a stack machine are:

operation

PUSH op

POP op

where:

• operation indicates the name of the operation to be performed. Operation always acts on the value(s) at top of the stack

• op is the address in the main memory where the value to be pushed/popped is located.

A stack machine has two memories: an unstructured one, we call it the main memory, where instructions and data are stored, and a structured one, the stack where access is allowed only through predefined operations (PUSH/POP).

Example 4:

Show how to implement the statement

a = a + b + a * c using a stack machine.

Solution

```
PUSH a      # push the value of a;
PUSH c      # push the value of c
MUL         # multiply the two values on top of the stack
PUSH b
ADD
PUSH a
ADD
POP  a    # store the result back in memory at the address where a is located.
```

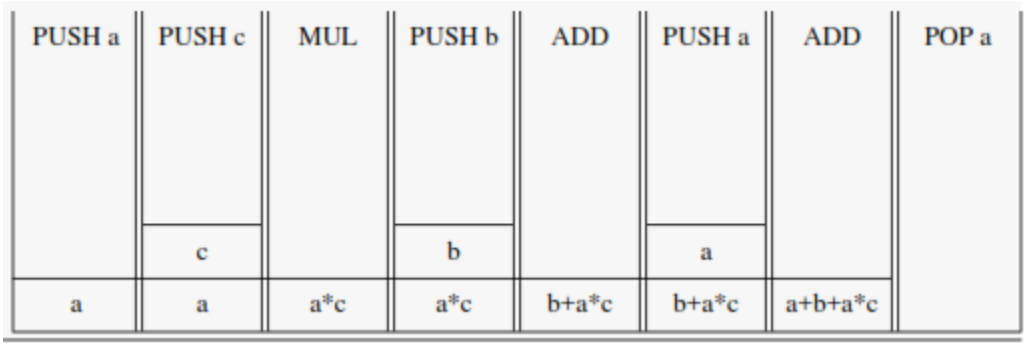
Whenever an operation is performed, the source operands are popped from the stack, the operation is performed, and the result is pushed into the stack

Example 5:

Show the content of the stack while implementing the statement:

a = a + b + a * c

Solution



3.3 Instruction Cycle

A computer instruction is a binary code that specifies a sequence of micro operations for the computer. Instruction codes together with data are stored in memory. The computer reads each

instruction from memory and places it in a control register. The control then interprets the binary code of the instructions and proceeds to execute it by issuing a sequence of micro operations.
A program that exists inside a computer's memory unit consists of a series of instructions.
The instruction cycle (also known as the fetch–decode–execute cycle, or simply the fetch-execute cycle) is the cycle that the central processing unit (CPU) follows from boot-up until the computer has shut down in order to process instructions. Figure 2 illustrated this cycle.

The processor executes these instructions through a cycle for each instruction. In a basic computer, each instruction cycle consists of the following phases:
Instruction fetch: fetch instruction from memory
Decode the instruction: what operation to be performed.
Read the effective address from memory
Execute the instruction

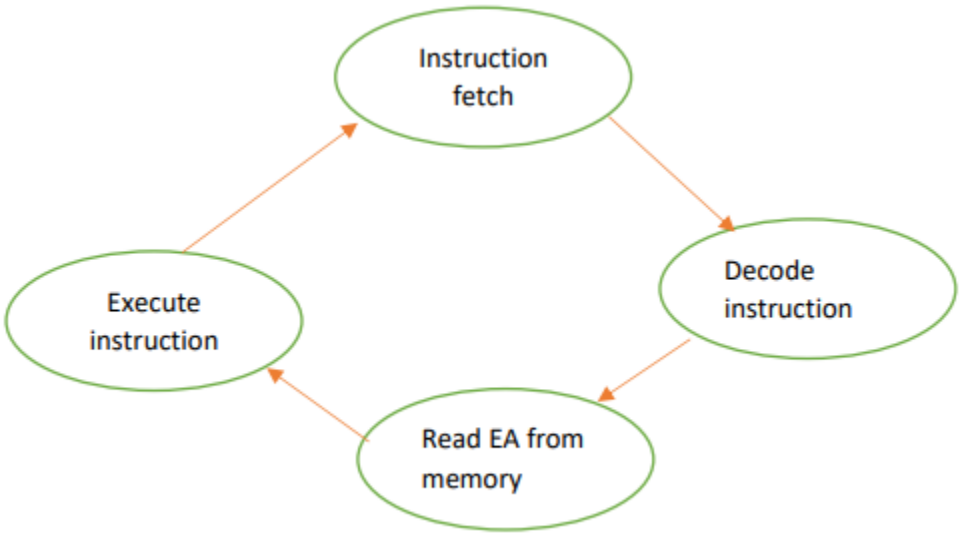


Figure 2: Instruction Cycle

Registers involved in each instruction cycle as shown in Figure 3 are:

i. Memory address registers (MAR): It is connected to System Bus address lines. It specifies the address of a read or write operation in memory.

ii. Memory Buffer Register (MBR): Also called Memory Data Register (MDR). It is connected to the system bus Data Lines. It holds the memory value to be stored, or the last value read from the memory.
iii. Program Counter (PC): Holds the address of the next instruction to be fetched.
iv. Instruction Register (IR): Holds the last instruction fetched.

3.3.1 Fetch cycle

The address of the next instruction to execute is in the Program Counter (PC) at the beginning of the fetch cycle.

Step 1: The address in the program counter is transferred to the Memory Address Register (MAR), as this is the only register that is connected to the system bus address lines.

Step 2: The address in MAR is put on the address bus, now a Read order is provided by the control unit on the control bus, and the result appears on the data bus and is then copied into the memory buffer register. Program counter is incremented by one, to get ready for the next instruction. These two acts can be carried out concurrently to save time.

Step 3: The content of the MBR is moved to the instruction register (IR).

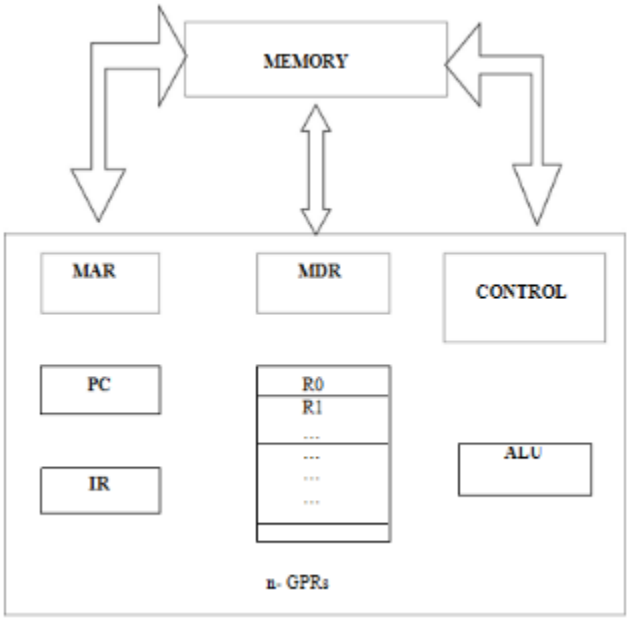


Figure 3: Registers Involved In Instruction Cycle

3.3.2 Decode instruction cycle

During the instruction fetch, the opcode in the instruction is decoded by the control unit (CU). The CU now "knows" which instruction it should execute, and can therefore output a sequence of levels and pulses to set up paths and effect the desired register transfers.

3.3.3 Execute Cycle

The CPU executes the instruction by reading values from registers, performing arithmetic or logical functions on them, and writing the result into a register. The result is stored in main memory or is sent to an output device

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory. In high-level languages, the compiler is responsible for translating high-level operations into low-level operations that access registers. This translation will be discussed in module 3.

The cycle of execution is summarized in Figure 4 below.

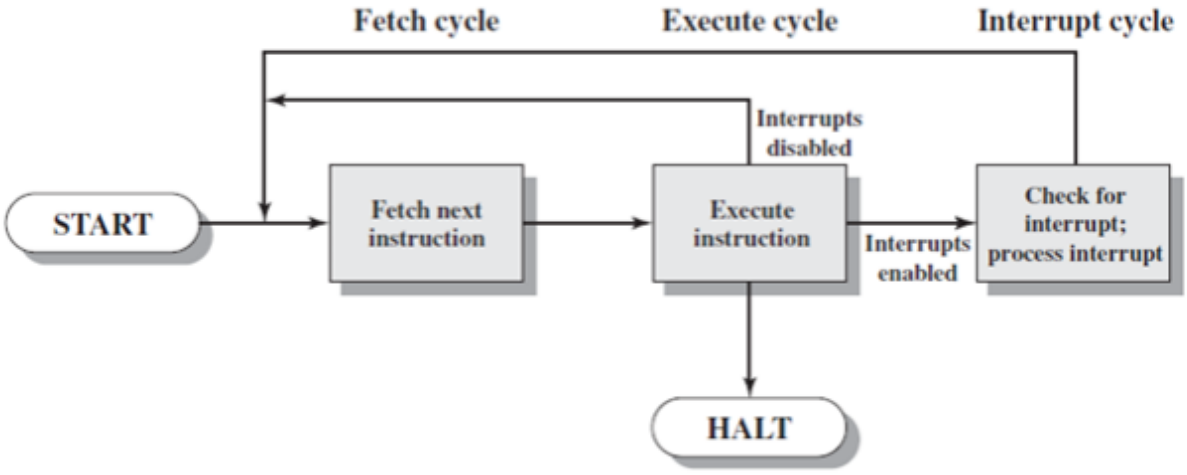


Figure 4: Instruction Cycle with Interrupt

Programs reside in the memory and usually get these through the Input unit. Execution of the program starts when the program counter is set to point at the first instruction of the program. The contents of the program counter are transferred to MAR and a Read Control Signal is sent to the memory. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR. Now contents of MDR are transferred to the IR and now the instruction is ready to be decoded and executed. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands. An operand in the memory is fetched by sending its address to MAR and initiating a read cycle. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU. After one or two such repeated cycles, the ALU can perform the desired operation. If the result of this operation is to be stored in the memory, the result is sent to MDR. Address of location where the result is stored is sent to MAR and a write cycle is initiated. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal. An interrupt is a request signal from an input and output device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine. The Diversion may change the internal stage of the processor so its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue

4.0 CONCLUSION

We have discussed the composition of machine language instruction format. The two major components of the instruction format are the function code also called ‘opcode’, which specifies the function or operation performed, and the operand addresses, which specify the locations of the operands used. The main methods of addressing in the instruction formats which specifies the address of operands that is 3-address machines, 2-address machines, 1-address machines, and 0-address machines were discussed. The instruction cycle for the execution of instructions was also explained. This unit has therefore shown how statements in high level programming language which is translated into machine language is implemented.

5.0 SUMMARY

This unit covered several aspects of machine instructions. The instruction format which is the size and arrangement of the component of machine instructions and the address format which is the addressing part of the instruction format were discussed. The main methods of address formats that is the 3-address format, 2-address format, 1-address format, and 0-address format were discussed with examples of how an high level language statement are processed in each of the format.

The instruction cycle was also discussed. The first operation in the cycle is the fetch cycle where the address of the next instruction to be executed in the Program Counter (PC) is retrieved from memory. Then decode cycle where, the opcode in the instruction is decoded by the control unit (CU). The last operation of the cycle is the execute cycle where the CPU executes the instruction by reading values from registers, performing arithmetic or logical functions on them, and writing the result into a register. It was also discussed that the operations of the cycle may be interrupted by a required service from an input and output device. In this case, the CPU will have to save the state of the operation so that it can resume it after the interrupt servicing has ended.

6.0 TUTOR-MARKED ASSIGNMENT

1. What is an instruction in machine language?
2. Differentiate between 2-operand and 3-operand address format
3. What is the meaning of the following instruction?
   ADD x, y, z
4. Show how to implement the high level statement
   a = a + 2b + 3a * c on a 3-address machine and then on a 2-address machine
5. Implement the high level statement:
   a = (a * b) + 2 * a * c
   on a three-address machine. Assume that variables a, b, and c are in registers R1, R2, and R3 respectively.
6. Show the content of the stack while implementing the statement: a = (a * b) + 2 * a * c
7. Discuss the instruction cycle

8. Discuss all registers involved in each instruction cycle

7.0 REFERENCES/FURTHER READINGS

William Stallings (2013). *Computer Organization and Architecture: Designing For Performance.* Pearson Education, Inc., publishing as Prentice Hall (9th ed).

Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).

French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).

Tanenbaum, Andrew S. (1993) *Structural Computer Organisation*. India: Prentice Hall. (3rd ed).

MODULE 3: PROGRAMMING LANGUAGES

Unit 1: BLOCK STRUCTURED LANGUAGES

Unit 2: SPECIFICATION AND TRANSLATION OF PROGRAMMING LANGUAGES

UNIT 1: BLOCK STRUCTURED LANGUAGES

CONTENTS

7.0 INTRODUCTION
8.0 OBJECTIVES
9.0 MAIN CONTENT

- **3.1 Programming language paradigm**
  - **3.1.1 Overview of the imperative paradigm**
  - **3.1.2 Overview of the Functional Paradigm**
  - **3.1.3 Overview of the logic paradigm**
  - **3.1.4 Overview of the object-oriented paradigm**
- **3.2 Subprograms**
- **3.3 Block Structured Programming**
  - **3.3.1 Names**
  - **3.3.2 Denotable Objects**
  - **3.3.3 Blocks**
  - **3.3.4 Environment**
- **3.4 Parameter passing**
  - **3.4.1 Parameter Passing Techniques**
    - **3.4.1.1 Pass by Value**
    - **3.4.1.2 Pass by reference**

4.0 CONCLUSION

5.0 SUMMARY

6.0 TUTOR-MARKED ASSIGNMENT

7.0 REFERENCES/FURTHER READINGS

1.0 INTRODUCTION

In module 1, we discussed the basic principles of programming and stated that designing a sound and correct computer programs involves following strictly the principles of program development life cycle. A good program could be developed when a good technique, model, pattern or paradigm is followed. A common term used to explain techniques, approaches or framework on which programming languages are developed is known as programming paradigm. This unit will present an overview of the main programming language paradigms. The paradigms presented are imperative, functional, logic, and object-oriented programming. Subprograms which are important aspect of program structure will be discuss. The two types of subprograms: functions and procedures will be discussed as well. Block structure approach which is the building blocks from which programs are constructed for efficiency and clarity to ensure quality of programs in programming languages is also part of the discussion in this unit. Finally, the unit will discuss parameter passing and its techniques. The main methods of parameter passing that will be discuss is the pass by value and pass by reference.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- explain what is meant by programming language paradigm
- distinguish between four computer programming language paradigms
- describe subprograms and explain some of the basic terminologies in subprograms
- list the general characteristics of subprograms
- explain block structured programming
- explain the parameter passing in programming language
- explain two methods of parameter passing in programming language.

3.0 MAIN CONTENT

3.1 Programming language paradigm

Paradigm means an example that serves as pattern, approach or model. Programming paradigm therefore means a pattern that serves as a school of thoughts for programming of computers. It is the preferred approach to programming that a language supports and also a classification of programming languages based on their features even though most popular languages support multiple paradigms. A good programmer might write great software using any programming paradigm

Programming paradigm presents programming techniques that are related to an algorithmic idea for solving a particular class of problems. Examples of these techniques are ‘divide and conquer', 'program development by stepwise refinement' and programming style. The details of these techniques are dealt with in Software engineering. A programming style that is of concern in this unit is block structure. Before we delve fully into that we will briefly enumerate the four main programming paradigms. These are the imperative paradigm, the functional paradigm, the logical paradigm, and the object-oriented paradigm. More details will be in another course titled “survey of programming languages”.

3.1.1 Overview of the imperative paradigm

The word 'imperative' can be used both as an adjective and as a noun. As an adjective it means 'expressing a command or plea'. In other words, asking for something to be done. As a noun, an imperative is a command or an order. For example

“First do this and next do that”

The 'first do this and next do that' is a short phrase which really in a nutshell describes the spirit of the imperative paradigm. The basic idea is the command, which has a measurable effect on the program state. The phrase also reflects that the order to the commands is important. 'First do that, then do this' would be different from 'do this, then do that'. It is the oldest but still the dominant paradigm. It is closest to the actual mechanical behavior of a computer as its languages were abstractions of assembly language. It is based on commands that update variables held in storage. Variables and assignment commands constitute a simple but useful abstraction from the memory fetch and update of machine instruction sets. Imperative programming languages can be

implemented very efficiently and is still dominant because it is related to the nature and purpose of programming. As programs are written to model real-world processes affecting real world objects, imperative programs model such processes.

In imperative paradigm also called procedural paradigm, we can think of a program as an active agent that manipulates passive objects. We encounter many passive objects in our daily life: a stone, a book, a lamp, and so on. A passive object cannot initiate an action by itself, but it can receive actions from active agents. A program in an imperative paradigm is an active agent that uses passive objects that we refer to as data or data items. Examples of an imperative language are FORTRAN (FORmula TRANslation), COBOL (Common Business-Oriented Language), Pascal, C, and Ada.

3.1.2 Overview of the Functional Paradigm

In the functional paradigm, a program is considered a mathematical function. In this context, a function is a black box that maps a list of inputs to a list of outputs (Figure 1). For example, "*summation"* can be considered as a function with *n* inputs and only one output.

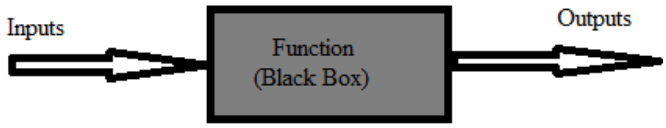


Figure 1: A function in a functional Language

The function takes the *n* inputs and adds them and created the sum.

Functional programming is in many respects a simpler and cleaner programming paradigm than the imperative one. The reason is that the paradigm originates from a purely mathematical discipline: the theory of functions. As described in section 3.1.1, the imperative paradigm is rooted in the key technological ideas of the digital computer, which are more complicated, and less 'clean' than mathematical function theory. Functional programming evaluates an expression and use the resulting value for something. Below we characterize the most important, overall properties of the functional programming paradigm.

i. It is based on mathematics and the theory of functions

ii. The values produced are non-mutable that is it is impossible to change any constituent of a composite value. As a remedy, it is possible to make a revised copy of composite value.
iii. Functions are full-fledged data just like numbers, lists, ...
iv. The language fits well with computations driven by needs.

Examples of functional language are LISP (LISt Programming), Scheme, Haskell, F#, etc.

3.1.3 Overview of the logic paradigm

The logic paradigm is dramatically different from the other three main programming paradigms. It uses the principle of logical reasoning to answer queries. It based on formal logic. The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation. For example, the famous rule of deduction in logic is:

If (A is B) and (B is C), then (A is C)

One famous logic language is PROLOG (PROgramming in LOGic)

3.1.4 Overview of the object-oriented paradigm

The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger. The underlying, and somewhat deeper reason to the success of the object-oriented paradigm is probably the conceptual anchoring of the paradigm. An object-oriented program is constructed with the outset in concepts, which are important in the problem domain of interest. An object-oriented program he theory of concepts, and models of human interaction with real world phenomena.

There are four key object-oriented program concepts, these are data abstraction, encapsulation, inheritance, and polymorphism. Data Abstraction focuses on the essential characteristics of some object which yields clearly defined boundaries. It is relative to the perspective of the viewer. Encapsulation is the compartmentalisation of structure and behaviour so that the details of an object’s implementation are hidden. Inheritance allow classes to use parent classes’ behavior and structure. It improves reliability and manageability and allows code reusability. Polymorphism

ensures that different implementations can be hidden behind a common interface. This means we can define several operations with the same name that can do different things in related classes.

We will now describe the most important properties of object-oriented programming as seen as a school of thought in the area of computer programming.

i. Data as well as operations are encapsulated in objects
ii. Information hiding is used to protect internal properties of an object
iii. Objects interact by means of message passing, a metaphor for applying an operation on an object. In most object-oriented languages, objects are grouped in classes.
iv. Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects.
v. Classes represent concepts whereas objects represent phenomena.
vi. Classes are organized in inheritance hierarchies

Some object-oriented programming languages include C++,JAVA, C#, etc.

3.2 Subprograms

The subprogram may be used to describe a component part of a program. Used loosely, the term may merely refer to any set of statements forming part of a program used for a specific task. However, a properly constructed subprogram should be self-contained, perform well-defined operations on well-defined data and have an internal structure that is independent of the program in which it is contained. When a subprogram has all these properties, it is sometimes called a program module. Subprograms are the fundamental building blocks of programs and are therefore among the most import concepts in programming language design.

The general characteristics of **Subprogram are:**

i. A subprogram has a single entry point.
ii. The caller is suspended during execution of the called subprogram, which implies that there is **only one** subprogram in execution **at any given time**.
iii. Control always returns to the caller when the called subprogram's execution terminates

Basic terminologies in Subprograms

i. A subprogram **definition** is a description of the actions of the subprogram abstraction.
ii. A subprogram **call** is an explicit request that the called subprogram be executed.
iii. A subprogram is said to be **active** if, after having been called, it has begun execution but has not yet completed that execution.
iv. A **subprogram header** is the first line of the definition, serves several definitions:
   - It specifies that the following syntactic unit is a subprogram definition of some particular kind.
   - The header provides a name for the subprogram.
   - May optionally specify a list of parameters.

Consider the following header examples:

FORTRAN:

```
Subroutine Adder(parameters)
```

Ada

```
procedure Adder(parameters)
```

C

```
void Adder(parameters)
```

v. The **parameter profile** (sometimes called the signature) of a subprogram is the number, order, and types of its formal parameters.
vi. The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type.
vii. A subprogram **declaration** provides the protocol, but not the body, of the subprogram.
viii. A **formal parameter** is a dummy variable listed in the subprogram header and used in the subprogram.
ix. An **actual parameter** represents a value or address used in the subprogram call statement.

There are two distinct categories of subprograms, these are procedures and functions.

Procedures: Procedures provide user-defined parameterized computation statements. Any defined way of carrying out some actions may be called a procedure. Programming procedures are defined operations on defined data and may be used as program components. The computations are enacted by single call statements.

Functions: Many high-level programming languages have inbuilt functions such as those that perform mathematical computations, such as Sine or absolute value. Another example are functions that convert from one numeric type to another such as an "Int" function, which has a real argument and evaluates to the largest integer not greater than the real, e.g. Int(3.6) evaluates to 3. There are functions that covert from characters to their integer ordinal value or vice versa e.g. a function with argument "A" that evaluates to 65 (ASCII charcter set)

A function is a piece of code identified by name, it is given a local environment of its own and is able to exchange information with the rest of the code using parameters. This concept translates into two different linguistic mechanisms. The first, definition (or declaration) of function, and its use (or call). In the program segment shown in Figure 2, the first five lines constitute the definition of the function named foo, whose local environment is composed from three names n, a, and tmp. The first line is the header, while the remaining lines constitute the body of the function. The last two lines are the uses (or calls) of foo.

```
int foo (int n, int a) {
   int tmp=a;
   if (tmp==0) return n;
   else return n+1;
}
...
int x;
x = foo(3,0);
x = foo(x+1,1);
```

Figure 2: Definition and use of a function

A function exchanges information with the rest of the program using three principal mechanisms: parameters, return value, nonlocal environment.

3.3 Block Structured Programming

An insistence on structured programming can directly contribute to the overall quality of programs and the achievement of many design aims. Structured programs are not only more comprehensible they are also much easier to test. Aids to good program design include using meaningful identifiers in programs, using subprograms and procedures, indenting of code to highlight its structure, and restricting the size of subprograms to manageable lengths. Block structured languages provide these aids to improve the quality of programs. Some terminologies used in block structured languages are discussed below.

3.3.1 Names

When we declare a new variable in a program:

int fie;

or we define a new function:

```
int foo( ){
   fie = 1;
}
```

We introduce new names, such as fie and foo to represent an object (a variable and a function in our example). The character sequence fie can be used every time that we want to refer to the new variable, just as the character sequence foo allows us to call the function that assigns to fie the value. A name is therefore nothing more than a sequence of characters used to represent, or denote, another object. In most languages, names are formed of identifiers.

The use of names implements a first, elementary, data abstraction mechanism.
For example, when, in an imperative language, we define a name using a variable, we are introducing a symbolic identifier for a memory location; therefore we are abstracting from the low-level details of memory addresses. If, then, we use the assignment command:

fie = 2;

the value 2 will be stored in the location reserved for the variable named ‘fie’. At the programming level, the use of the name avoids the need to bother with whatever this location is. The

correspondence between name and memory location must be guaranteed by the implementation. We will use the term environment to refer to that part of the implementation responsible for the associations between names and the objects that they denote.

3.3.2 Denotable Objects

The objects to which a name can be given are called denotable objects. Even if there are considerable differences between programming languages, the following is a non-exhaustive list of possible denotable objects:

• Objects whose names are defined by the user: variables, formal parameters, procedures (in the broad sense), user-defined types, labels, modules, user-defined constants, exceptions.

• Objects whose names are defined by the programming language: primitive types, primitive operations, predefined constants.

The association (or binding) between a name and an object it denotes can therefore be created at various times. Some names are associated with objects during the design of a language, while other associations are introduced only when a program is executed.

Not all associations between names and denotable objects are fixed once and for all at the start of program execution. Many can vary during execution. To be able to understand how these associations behave, we need to introduce the concept of environment. Definition 4.1 (Environment) The set of associations between names and denotable objects which exist at runtime at a specific point in the program and at a specific time during execution, is called the (referencing) environment. Usually, when we speak of environments, we refer only to associations that are not established by the language definition. The environment is therefore that component of the abstract machine which, for every name introduced by the programmer and at every point in the program, allows the determination of what the correct association is. Note that the environment does not exist at the level of the physical machine. The presence of the environment constitutes one of the principle characteristics of high-level languages which must be simulated in a suitable fashion by each implementation of the language. A declaration is a construct that allows the introduction of an association in the environment. High-level languages often have explicit declarations, such as:

```
int x;
int f (){
   return 0;
}
type T = int;
```

The first is a declaration of a variable, the second of a function named f, the third is declaration of a new type, T, which coincides with type int). Some languages allow implicit declarations which introduce an association in the environment for a name when it is first used. The denoted object's type is deduced from the context in which the name is used for the first time.

As we will see in detail below, there are various degrees of freedom in associations between names and denotable objects. First of all, a single name can denote different objects in different parts of the program. Consider the following code segment:

```
{int fie;
 fie = 2;
    {char fie;
     fie = a;
    }
}
```

The outermost name fie denotes an integer variable, while the inner one is of type character.

While different names for the same object are used in different environments, no particular problems arise. The situation is more complicated when a single object is visible using different names in the same environment. This is called aliasing and the different names for the same object called aliases. If the name of a variable passed by reference to a procedure is also visible inside the same procedure, we have a situation of aliasing. Other aliasing situations can easily occur using pointers. If X and Y are variables of pointer type, the assignment X = Y allows us to access the same location using both X and Y.

Let us consider, for example, the following fragment of C program where, as we will do in the future, we assume that write(Z) is a procedure which allows us to print the value of the integer variable Z:

```
int *X, *Y;          // X,Y pointers to integers
X = (int *) malloc (sizeof (int));
                     // allocate heap memory
*X = 5;              // * dereference
Y=X;                 // Y points to the same object as X
*Y=10;
write(*X);
```

The names X and Y denote two different variables, which, however, after the execution of the assignment command X=Y, allow to access the same memory location (therefore, the next print command will output the value 10).

3.3.3 Blocks

Almost all important programming languages today permit the use of blocks, a structuring method for programs introduced by ALGOL60. Block structuring is fundamental to the organisation of the environment. A block is a textual region of the program, identified by a start sign and an end sign, which can contain declarations local to that region (that is, which appear within the region).

The start- and end-block constructs vary according to the programming language: begin ... end for languages in the ALGOL family, braces {...} for C and Java, round brackets (...) for LISP and its dialects, let ... in ... end in ML, etc. Moreover, the exact definition of block in the specific programming language can differ slightly from the one given above. In some cases, for example, one talks about block only when there are local declarations. Often, though, blocks have another important function, that of grouping a series of commands into a syntactic entity which can be considered as a single (composite) command. These distinctions, however, are not relevant as far as we are concerned. We will, therefore, use the definition given above and we distinguish two cases:

Block associated with a procedure: This is a block associated with declarations local to a procedure. It corresponds textually to the body of the procedure itself, extended with the declarations of formal parameters.

In-line block: This is a block which does not correspond to a declaration of procedure and which can appear (in general) in any position where a command can appear.

3.3.4 Environment

The environment changes during the execution of a program. However, the changes occur generally at two precise times: on the entry and exit of a block. The block can therefore be considered as the construct of least granularity to which a constant environment can be associated. A block's environment, meaning by this terminology the environment existing when the block is executed, is initially composed of associations between names declared locally to the block itself. In most languages allowing blocks, blocks can be nested; that is, the definition of one block can be wholly included in that of another. An example of nested anonymous blocks is shown in the program segment below:

```
{int fie;
 fie = 2;
    {char fie;
     fie = a;
    }
}
```

The overlapping of blocks so the last open block is not the first block to be closed is never permitted. In other words, a sequence of commands of the following kind is not permitted in any language:

open block A;

open block B;

close block A;

close block B;

Different languages vary, then, in the type of nesting they permit. In C, for example, blocks associated with procedures cannot be nested inside each other (that is, there cannot be procedure declarations inside other procedures), while in Pascal and Ada this restriction is not present. Block nesting is an important mechanism for structuring the environment. There are mechanisms that allow the declarations local to a block to be visible in blocks nested inside it.

Remaining informal for the time being, we say that a declaration local to a block is visible in another block when the association created by such a declaration is present in the second block. Those mechanisms of the language which regulate how and when the declaration is visible are called visibility rules. The canonical visibility rule for languages with blocks states that a declaration local to a block is visible in that block and in all blocks listed within it, unless there is a new declaration of the same name in that same block. In this case, in the block which contains the redefinition, the new declaration hides the previous one.

In the case in which there is a redefinition, the visibility rule establishes that only the last name declared will be visible in the internal block, while in the exterior one there is a visibility hole. The association for the name declared in the external block will be, in fact, deactivated for the whole of the interior block (containing the new declaration) and will be reactivated on exit from the inner block. Note that there is no visibility from the outside inwards. Every association introduced in the environment local to a block is not active (or rather the name that it defines is not visible) in an exterior block which contains the interior one. Analogously, if we have two blocks at the same nesting level, or if neither of the two contains the other, a name introduced locally in one block is not visible in the other.

The definition just given, although apparently precise, is insufficiently so to establish with precision what the environment will be at an arbitrary point in a program.
We will assume this rule for the rest of this section, while the next will be concerned with stating the visibility rules correctly.

In general we can identify three components of an environment. The environment associated with a block is formed of the following components:
Local environment: This is composed of the set of associations for names declared locally to the block. In the case in which the block is for a procedure, the local environment contains also the associations for the formal parameters, given that they can be seen, as far as the environment is concerned, as locally declared variables.
Non-local environment: This is the environment formed from the associations for names which are visible from inside a block but which have not been declared locally.

Global environment: Finally, there is the environment formed from associations created when the program's execution began. It contains the associations for names which can be used in all blocks forming the program.

The environment local to a block can be determined by considering only the declarations present in the block. We must look outside the block to define the non-local environment. The global environment is part of the non-local environment. Names introduced in the local environment can be themselves present in the non-local environment. In such cases, the innermost (local) declaration hides the outermost one. The visibility rules specify how names declared in external blocks are visible in internal ones. In some cases, it is possible to import names from other, separately defined modules. The associations for these names are part of the global environment.

We will now consider the program segment below where, for ease of reference, we assume that the blocks can be labelled. The labels behave as comments as far as the execution is concerned.

```
A:{int a =1;

   B:{int b = 2;
      int c = 2;

      C:{int c =3;
         int d;
         d = a+b+c;
         write(d)
      }

      D:{int e;
         e = a+b+c;
         write(e)
      }
   }
}
```

Let us assume that block A is the outermost. It corresponds to the main program. The declaration of the variable 'a' introduces an association in the global environment.
Inside block B two variables are declared locally (b and c). The environment for B is therefore formed of the local environment, containing the association for the two names (b and c) and from the global environment containing the association for 'a'.

Inside block C, 2 local variables (c and d) are declared. The environment of C is therefore formed from the local environment, which contains the association for the two names (c and d) and from the non-local environment containing the same global environment as above, and also the association for the name ‘b’ which is inherited from the environment of block B. Note that the local declaration of ‘c’ in block C hides the declaration of ‘c’ present in block B. The print command present in block C will therefore print the value 6.

In block D, finally, we have a local environment containing the association for the local name ‘e’, the usual global environment and the non-local environment, which, in addition to the association for a contains the association for the names ‘b’ and ‘c’ introduced in block B. Given that variable ‘c’ has not been internally re-declared, in this case, therefore, the variable declared in block B remains visible and the value printed will be 5. Note that the association for the name ‘d’ does not appear in the environment non-local to D, given that this name is introduced in an exterior block which does not contain D. The visibility rules, indeed, allows only the inheritance of names declared in exterior blocks from interior ones and not vice versa.

3.4 Parameter passing

One way that a non-local method program can gain access to the data that it is to process is through parameter passing. Data that passed through parameters are accessed through names that are local to the subprogram. A subprogram with parameter access to the data it is to process is a parameterized computation. It can perform its computation on whatever data it receives through its parameters. Parameter is a special kind of variable, used in subprogram to refer to one of the pieces of data provided as input to the subprogram. These pieces of data are called arguments. An ordered list of parameters is usually included in the definition of a subprogram, so that, each time the subprogram is called, its arguments for that call can be assigned to the corresponding parameters (Figure 3).

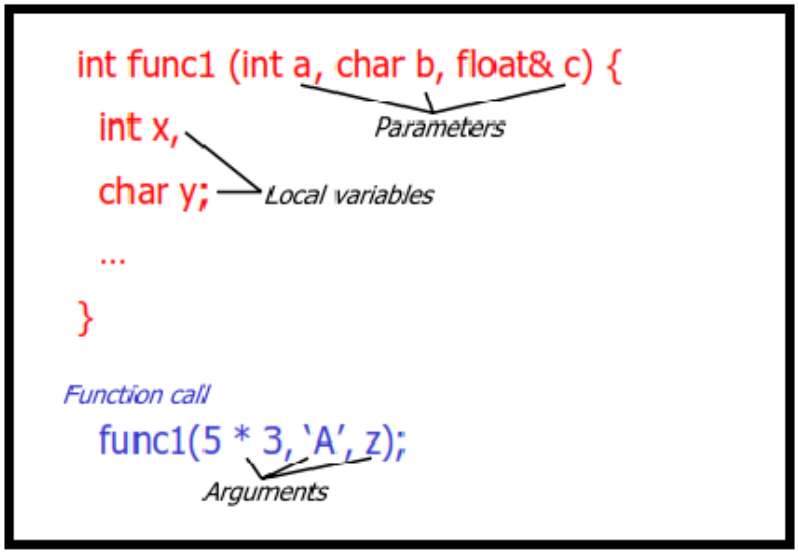


Figure 3: Assigning Arguments to the corresponding parameters

A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram. Subprograms call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. An actual parameter represents a value or address used in the subprogram call statement. Consider the following Figure 4:

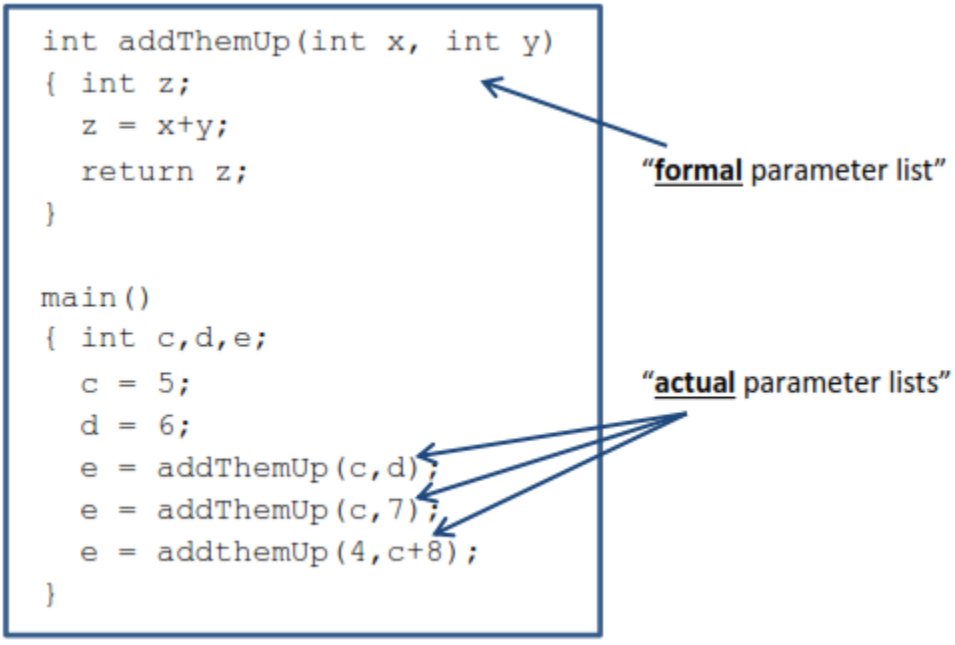


Figure 4: Difference between Formal and Actual Parameters

We distinguish between formal parameters, which appear in the definition of a function, and actual parameters, which appear, instead, in the call. The formal parameters are always names which, as far as the environment is concerned, behave as declarations local to the function itself. They behave, in particular, as bound variables, in the sense that their consistent renaming has no effect on the semantics of the function. For example, the function foo in Figure 2 and that in Figure 5 are indistinguishable, even though this second one has different names for its formal parameters.

```
int foo (int m, int b){
   int tmp=b;
   if (tmp==0) return m;
   else return m+1;
}
```

Figure 5: Renaming of formal parameters

The way in which actual parameters are paired with formal parameters, and the semantics which results from this, is called the parameter passing discipline. According to what is now traditional terminology, a specific mode is composed of the kind of communication that it supports, together with the implementation that produces this form of communication. The mode is fixed when the function is defined and can be different for each parameter; it is fixed for all calls of the function. From a strictly semantic viewpoint, the classification of the type of communication permitted by a parameter is simple. From a subprogram's viewpoint, three parameter classes can be discerned:

• Input parameters.
• Output parameters.
• Input/output parameters.

A parameter is of input type if it allows communication which is only in the direction from the caller to the function (the "callee"). It is of output type if it permits communication only in direction from the callee to the caller. Finally, it is input/output when it permits bidirectional communication.

Note that this is a linguistic classification, part of the definition of the language; it is not derived from the use to which parameters are put. An input/output parameter remains that way even if it is used only in a unidirectional fashion (e.g., from caller to callee).

3.4.1 Parameter Passing Techniques

The main techniques for parameter passing are Pass by Value, Pass by Reference, Pass by Pointer pass by name and pass by results. Of the techniques that we will discuss, the first two (by value and by reference) are the most important and are widely used. The others are little more than variations on the theme of call by value. An exception is call by name, which we will discuss last. Although call by name is no longer used as a parameter-passing mechanism, nevertheless, it allows us to present a simple case of what it means to "pass an environment" into a procedure.

3.4.1.1 Pass by Value

Pass by value Call by value is a mode that corresponds to an input parameter. The local environment of the procedure is extended with an association between the formal parameter and a new variable. The actual parameter can be an expression. When called, the actual parameter is evaluated and its r-value that is the value of the contents of variable (or result of expression) obtained and associated with the formal parameter. On termination of the procedure, the formal parameter is destroyed, as is the local environment of the procedure itself. During the execution of the body, there is no link between the formal and the actual parameter. There is no way of making use of a value parameter to transfer information from the callee to the caller.

Figure 6 shows a simple example of passing by value. Like in C, C++, Pascal and Java, when we do not explicitly indicate any parameter-passing method for a formal parameter, it is to be understood that parameter is to be passed by value. The variable y never changes its value (it always remains 1). During the execution of foo, x assumes the initial value 2 by the effect of passing the parameter. It is then incremented to 3, finally it is destroyed with the entire activation record for foo.

```
int y = 1;
void foo (int x) {
   x = x+1;
   }
...
y = 1;
foo(y+1);
   // here y =  1
```

Figure 6: Pass By Value

Passing by value is a very simple mechanism with clear semantics. It is the default mechanism in many languages (e.g., Pascal) and is the only way to pass parameters in C and Java.

3.4.1.2 Pass by reference

Pass by reference (also called by variable) implements a mechanism in which the parameter can be used for both input and output. Used when we want to return more than one value from a function.

The actual parameter must be an expression with l-value that is the location of variable. At the point of call, the l-value of the actual parameter is evaluated and the procedure's local environment is extended with an association between the formal parameter and the actual parameter's l-value (therefore creating an aliasing situation). The most common case is that in which the actual parameter is a variable. In this case, the formal and the actual are two names for the same variable. At the end of the procedure, the connection between the formal parameter and the actual parameter's l-value is destroyed, as is the environment local to the procedure. It is clear that call by reference allows bidirectional communication: each modification of the formal parameter is a modification of the actual parameter.

Figure 7 shows a simple example of call by reference (which we have notated in the pseudocode with the reference modifier). During the execution of foo, x is a name for y. Incrementing x in the body is, to all effects, the incrementing of y. After the call, the value of y is therefore 1.

```
int y = 0;
void foo (reference int x) {
   x = x+1;
   }
y=0;
foo(y);
   // here y = 1
```

Figure 7: Pass By Reference

It can be seen that, as shown in Figure 8, the actual parameter need not necessarily be a variable but can be an expression whose l-value is determined at call time. In a way similar to the first case, during the execution of foo, x is a name for v[1] and the increment of x in the body, is an increment of v[1]. After the call, the value of v[1] is, therefore, 2.

```
int[] V = new V[10];
int i=0;
void foo (reference int x) {
   x = x+1;
   }
...
V[1] = 1;
foo(V[i+1]);
   // here V[1] = 2
```

Figure 8: Another example of Pass by Reference

Pass by reference is a low-level operation. It is possible in Pascal (var modifier) and in many other languages. It has been excluded from more modern languages.

10.0 CONCLUSION

In this unit, what is meant by programming language paradigm and the differences between four computer programming language paradigms were discussed. We described the concept of subprograms and explain some of the basic terminologies in subprograms. Some general characteristics of subprograms were also listed.

The idea of block structured programming was explained and the unit concluded on how programming languages handles parameter passing in program structures. Discussions on the two main methods of parameter passing in programming language concluded the unit.

11.0 SUMMARY

This unit has discussed the different approaches in different programming language to give an insight on which language may be appropriate for any particular problem. The development of good programs has been emphasized in module 1, techniques in program structures such as usage of subprograms and block structured languages to enhance the development of good programs were discussed in this unit. In programming principle, a function that is not local to a method cannot gain access to data in such method. A way by which non-local method program can gain access to the data that it is to process is through parameter passing. The main methods of parameter passing: pass by value and pass by reference were discussed.

12.0 TUTOR-MARKED ASSIGNMENT

1. List four common computer language programming paradigms
2. Compare and contrast, a procedural paradigm with an object-oriented paradigm
3. Explain the main concept of object-oriented programming
4. What is subprogram? Explain its two basic types.
5. Described the concept of subprograms
6. Explain the following terminologies in subprograms
    i. subprogram definition
    ii. subprogram call
    iii. active subprogram
    iv. subprogram header
    v. parameter profile
7. Explain the differences in use between actual parameters and formal parameters
8. If the subprogram *calculate (A, B, S, P)* accepts the value of A and B and calculates their sum S and product P, which variable do you pass by value and which one by reference?

13.0 REFERENCES/FURTHER READINGS

5. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).
6. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).
7. Cooke, D. A. (2003). *Concise Introduction to Computer Languages*. Pacific Grove, CA: Brooks/Cole.
8. Tucker, A. and Noonan, R. (2002). *Programming Languages: Principles and Paradigms.* McGraw-Hill.
9. Sebester, R. (2006). *Concepts of programming languages.* Addison Wesley.
10. Pratt, T. W. and Zelkowitz, M. V. (1999). *Programming Languages: Design and implementation* Prentice Hall (3rd ed).

UNIT 2: SPECIFICATION AND TRANSLATION OF PROGRAMMING LANGUAGES

CONTENTS

4.0 INTRODUCTION
5.0 OBJECTIVES
6.0 MAIN CONTENTS
3.1 PROGRAMMING LANGUAGE SPECIFICATION
3.1.1 Forms o**f Programming Language Specification**
3.1.1.1 SYNTAX
3.1.1.2 SEMANTICS
3.1.2 Programming language reference
3.2 PROGRAMMING LANGUAGE TRANSLATION
3.2.1 Assembler
3.2.1.1 Types of Assembler
3.2.2 Compiler
3.2.2.1 Different Phases of Compilation
3.2.3 Interpreter
7.0 CONCLUSION
8.0 SUMMARY
9.0 TUTOR-MARKED ASSIGNMENT
10.0 REFERENCES/FURTHER READINGS

1.0 INTRODUCTION

This unit can be grouped into two parts; in the first part we will consider the of programming languages while the second part will be for discussion on programming language translation. The specification of programming language is the definition of such programming language. It is a

description of the syntax and semantics of the language so that the language will be used without any ambiguity. Specification of programming language also involves programming language reference which is all about the documentation manual for the language.

For programming language translation, we will consider the three main translator discussed in module 1: the assembler, the compiler, and the interpreter. We will discuss the briefly the processes of translation involved in each of the translator. This unit will serve as a foundation for courses like Compiling Techniques and Programming with Assembly language.

2.0 OBJECTIVES

By the end of the unit, you will be able to:

- explain what is meant by programming language specification
- explain the forms of programming language specification
- describe the syntax and semantics of a programming language
- explain programming language reference or language reference manual as the part of the documentation associated with most programming languages
- explain programming language translation and translators
- describe an assembler
- describe a compiler and explain different phases of compilation
- describe an interpreter and states its advantages and disadvantages.

3.0 MAIN CONTENTS

3.1 PROGRAMMING LANGUAGE SPECIFICATION

In computing, a programming language specification (or standard or definition) is a documentation artifact that defines a programming language so that users and implementors (language translators) can agree on what programs in that language mean. Specifications are typically detailed and formal, and primarily used by implementors, with users referring to them in case of ambiguity; the C++ specification is frequently cited by users, for instance, due to the complexity. Related documentation includes a programming language reference, which is intended expressly for users, and a programming language rationale, which explains why the specification is written as it is; these are typically more informal than a specification.

Not all major programming languages have specifications, and languages can exist and be popular for decades without a specification. Perl (through Perl 5) is a notable example of a language without a specification, while PHP was only specified in 2014, after being in use for 20 years. A language may be implemented and then specified, or specified and then implemented, or these may develop together, which is usual practice today. This is because implementations and specifications provide checks on each other: writing a specification requires precisely stating the behavior of an implementation, and implementation checks that a specification is possible, practical, and consistent. Writing a specification before an implementation has largely been avoided since ALGOL 68 (1968), due to unexpected difficulties in implementation when implementation is deferred. However, languages are still occasionally implemented and gain popularity without a formal specification: an implementation is essential for use, while a specification is desirable but not essential (informally, "code talks"). ALGOL 68 was the first (and possibly one of the last) major language for which a full formal definition was made before it was implemented.

3.1.1 Forms o**f Programming Language Specification**

A programming language specification can take several forms, including the following:

- **i.** Formal Language Specification: Language Specifications consist of two parts: The syntax of a programming language is the part of the language definition that says what programs look like; their form and structure. The semantics of a programming language is the part of the language definition that says what programs do; their behavior and meaning. An explicit definition of the syntax and semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., the approach taken for the C language), or a formal semantics (e.g., the Standard ML and Scheme specifications). A notable example is the C language, which gained popularity without a formal specification, instead being described as part of a book, The C Programming Language (1978), and only much later being formally standardized in ANSI C (1989).
- **ii.** A description of the behavior of a compiler (sometimes called "translator") for the language (e.g., the C++ language and FORTRAN). The syntax and semantics of the

language has to be inferred from this description, which may be written in natural or a formal language.

iii. A model implementation, sometimes written in the language being specified (e.g., Prolog). The syntax and semantics of the language are explicit in the behavior of the model implementation.

3.1.1.1 SYNTAX

The syntax of a programming language is usually described using a combination of the following two components:

i. A regular expression describing its lexemes, and
ii. A context-free grammar which describes how lexemes may be combined to form a syntactically correct program.

A context-free grammar basically consists of a finite set of grammar rules. In order to define grammar rules, we assume that we have two kinds of symbols: the terminals, which are the symbols of the alphabet underlying the languages under consideration, and the non-terminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \rightarrow a$, where A is a single nonterminal, and the right-hand side a is a string of terminal and/or nonterminal symbols.

A context-free grammar is a quadruple G = (V, S, P, S)
Where
• V is a finite set of symbols called the vocabulary (or set of grammar symbols);
• $S \subseteq V$ is the set of terminal symbols (for short, terminals);
• $S \in (V - S)$ is a designated symbol called the start symbol;
• $P \subseteq (V - S) \times V$ is a finite set of productions (or rewrite rules, or rules).

3.1.1.2 SEMANTICS

Formulating a rigorous semantics of a large, complex, practical programming language is a daunting task even for experienced specialists, and the resulting specification can be difficult for anyone but experts to understand. The following are some of the ways in which programming language semantics can be described; all languages use at least one of these description methods, and some languages combine more than one:

A. Natural language: Description by human natural language.

Most widely used languages are specified using natural language descriptions of their semantics. This description usually takes the form of a reference manual for the language. These manuals can run to hundreds of pages, e.g., the print version of The Java Language Specification, 3rd Ed. is 596 pages long.

B. Formal semantics: Description by mathematics.

Formal semantics are grounded in mathematics. As a result, they can be more precise and less ambiguous than semantics given in natural language. However, supplemental natural language descriptions of the semantics are often included to aid understanding of the formal definitions. For example, The ISO Standard for Modula-2 contains both a formal and a natural language definition on opposing pages.

Programming languages whose semantics are described formally can reap many benefits. For example:

i. Formal semantics enable mathematical proofs of program correctness;
ii. Formal semantics facilitate the design of type systems, and proofs about the soundness of those type systems. In programming languages, a type system is a logical system comprising a set of rules that assigns a property called a type to the various constructs of a computer program, such as variables, expressions, functions or modules.
iii. Formal semantics can establish unambiguous and uniform standards for implementations of a language.

Automatic tool support can help to realize some of these benefits. For example, an automated theorem prover or theorem checker can increase a programmer's (or language designer's) confidence in the correctness of proofs about programs (or the language itself). The power and

scalability of these tools varies widely: full formal verification is computationally intensive, rarely scales beyond programs containing a few hundred lines and may require considerable manual assistance from a programmer; more lightweight tools such as model checkers require fewer resources and have been used on programs containing tens of thousands of lines; many compilers apply static type checks to any program they compile.

C. Reference implementations: Description by computer program

A reference implementation is a single implementation of a programming language that is designated as authoritative. The behavior of this implementation is held to define the proper behavior of a program written in the language. This approach has several attractive properties. First, it is precise, and requires no human interpretation: disputes as to the meaning of a program can be settled simply by executing the program on the reference implementation (provided that the implementation behaves deterministically for that program).

On the other hand, defining language semantics through a reference implementation also has several potential drawbacks. Chief among them is that it conflates limitations of the reference implementation with properties of the language. For example, if the reference implementation has a bug, then that bug must be considered to be an authoritative behavior. Another drawback is that programs written in this language may rely on quirks in the reference implementation, hindering portability across different implementations.

Nevertheless, several languages have successfully used the reference implementation approach. For example, the Perl interpreter is considered to define the authoritative behavior of Perl programs. In the case of Perl, the open-source model of software distribution has contributed to the fact that nobody has ever produced another implementation of the language, so the issues involved in using a reference implementation to define the language semantics are moot.

D. Test suites: Description by examples of programs and their expected behaviors.

While few language specifications start off in this form, the evolution of some language specifications has been influenced by the semantics of a test suite (e.g. in the past the specification of Ada has been modified to match the behavior of the Ada Conformity Assessment Test Suite).

Defining the semantics of a programming language in terms of a test suite involves writing a number of example programs in the language, and then describing how those programs ought to behave — perhaps by writing down their correct outputs. The programs, plus their outputs, are called the "test suite" of the language. Any correct language implementation must then produce exactly the correct outputs on the test suite programs.

The chief advantage of this approach to semantic description is that it is easy to determine whether a language implementation passes a test suite. The user can simply execute all the programs in the test suite, and compare the outputs to the desired outputs. However, when used by itself, the test suite approach has major drawbacks as well. For example, users want to run their own programs, which are not part of the test suite; indeed, a language implementation that could only run the programs in its test suite would be largely useless. But a test suite does not, by itself, describe how the language implementation should behave on any program not in the test suite; determining that behavior requires some extrapolation on the implementor's part, and different implementors may disagree. In addition, it is difficult to use a test suite to test behavior that is intended or allowed to be nondeterministic.

Therefore, in common practice, test suites are used only in combination with one of the other language specification techniques, such as a natural language description or a reference implementation.

3.1.2 Programming language reference

Documentation is any communicable material that is used to describe, explain or instruct regarding some attributes of an object, system or procedure, such as its parts, assembly, installation, maintenance and use. In computing, a programming language reference or language reference manual is part of the documentation associated with most mainstream programming languages. It is written for users and developers, and describes the basic elements of the language and how to use them in a program. For a command-based language, for example, this will include details of every available command and of the syntax for using it. The reference manual is usually separate and distinct from a more detailed programming language specification meant for implementors of the language rather than those who simply use it to accomplish some processing task.

There may also be a separate introductory guide aimed at giving newcomers enough information to start writing programs, after which they can consult the reference manual for full details. Frequently, however, a single publication contains both the introductory material and the language reference.

3.2 PROGRAMMING LANGUAGE TRANSLATION

Programming language translation is the conversion of statements written in one language to statements in another language, e.g. converting assembly language to machine code. A translator is a program that performs this translation. There are three types of translator: assemblers, interpreters and Compilers.

3.2.1 Assembler

The assembler translates mnemonic operation codes into machine code, and symbolic addresses into machine addresses (Figure 1).

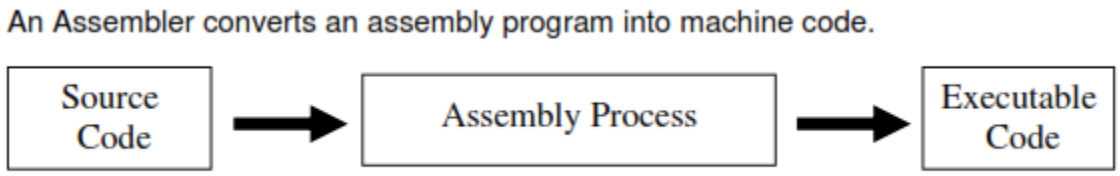


Figure 1: Translation by Assembler

Translation of source codes (assembly language statements) to object codes (machine codes) needs:

i. Translation of mnemonic opcodes to equivalent machine codes e.g. STL to 14
ii. Translation of symbolic labels to equivalent machine address e.g. RETADR to 1033
iii. Building of machine instructions in proper format.
iv. Conversion of data constants into internal machine representation, such as EOF to 454F46
v. Writing the object program and the assembly listing.

The above translation of object codes to source codes is done by the assembler in two parts: analysis and synthesis. Analysis means to take them into parts. Then synthesis means to put it together. First, to analyze if it is a valid program and then transform into data structures. In the transformation to data structures, the following steps are carried out:

a. split program into lines (get line, readline);
b. lexical analysis (scanning);
c. context-free analysis (parsing);
d. context-sensitive analysis (semantic analysis)

Secondly the machine binary codes will be created from the result of analysis. In the middle, there is an intermediate representation.

3.2.1.1 Types of Assembler

a. Load-and-Go Assembler:

   Load-and-go assembler generates their object code in memory for immediate execution. No object program is written out and no loader is needed. It is useful in a system with frequent program development and testing. Programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

b. One-Pass Assemblers

Assign addresses to all statements in source code and save values (addresses) assigned to labels for subsequent usage and then process directives.

One-Pass Assemblers generate their object code in memory for immediate execution just like load-and-go assemblers. External storage for the intermediate file between two passes is slow or is inconvenient to use because they require that all areas be defined before they are referenced (forward reference) this is possible, although inconvenient, to do so for data items. This is the problem of forward reference in one-pass assembler.

Forward reference in one pass assembler makes the assembler to omit the operand address if the symbol has not yet been defined. The assembler stores this undefined symbol and indicates that it is undefined. It then adds the address of this operand address to a list of forward references

associated with the assembler's entry. When the definition for the symbol is encountered, it scans the reference list and inserts the address. At the end of the program, it reports the error if there are still entries indicated as undefined symbols.

c. Two Pass Assemblers

Two pass assembler translate instructions by converting labels to addresses, generating values defined by BYTE and WORD, processing the directives not done in pass one and then writing the object code to output device

For a two pass assembler, forward references in symbol definition are not allowed, symbol definition must be completed in pass 1. Prohibiting forward references in symbol definition is not a serious inconvenience.

3.2.2 Compiler

A compiler translates a program written in one high level language, the source code into another language which is the object code. Most compilers are organized into three stages: a front end, an optimizer, and a back end. The front end is responsible for understanding the program. It makes sure the program is valid and transforms it into an intermediate representation, a data structure used by the compiler to represent the program. The optimizer improves the intermediate representation to increase the speed or reduce the size of the executable which is ultimately produced by the compiler. The back end converts the optimized intermediate representation into the output language of the compiler.

Compilation is a different process, where a compiler reads in a program, but instead of running the program, the compiler translates it into some other language, such as bytecode or machine code. The translated code may either be directly executed by hardware, or serve as input to another interpreter or another compiler Figure 2.

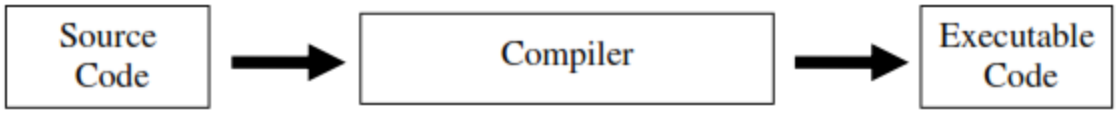


Figure 2: Compilation Process

3.2.2.1 Different Phases of Compilation

The major phases of compilation process are lexical analysis, Syntactic analysis, Semantic analysis, Code Optimisation, and Code generation.

A. Lexical analysis

The aim of lexical analysis is to read the symbols (characters) forming the program sequentially from the input and to group these symbols into meaningful logical units, which we call tokens. For example, the lexical analyser of C or Java, when presented with the string x = 1 + y++; will produce 7 tokens: the identifier x, the assignment operator =, the number 1, the addition operator +, the identifier foo, the auto increment operator ++ and finally the command termination token.

B. **Syntactic analysis**

Once the list of tokens has been constructed, the syntactic analyser (or parser) seeks to construct a derivation tree for this list. This is, clearly, a derivation tree in the grammar of the language. Each leaf of this tree must correspond to a token from the list obtained by the scanner (Figure 3).

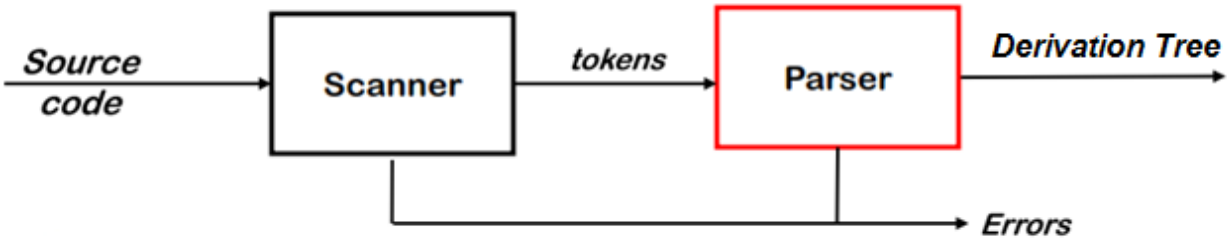


Figure 3: Scanning and Parsing

C. **Semantic analysis**

The derivation tree (which represents the syntactic correctness of the input string) is subjected to checks of the language's various context-based constraints. As we have seen, it is at this stage that

declarations, types, number of function parameters, etc., are processed. As these checks are performed, the derivation tree is augmented with the information derived from them and new structures are generated.

D. Code Optimisation

The code obtained from the preceding phases by repeatedly traversing the derivation tree is fairly inefficient. There are many optimisations that can be made before generating object code. Typical operations that can be performed are:

i. Removal of useless code (dead code removal). That is, removal of pieces of code that can never be executed because there is no execution sequence that can reach them.
ii. In-line expansion of function calls. Some function (procedure) calls can be substituted by the body of the associated function, making execution faster. It also makes other optimisations possible.
iii. Subexpression factorisation. Some programs compute the same value more than once. If, and when, this fact is discovered by the compiler, the value of the common subexpression can be calculated once only and then stored.
iv. Loop optimisations. Iterations are the places where the biggest optimisations can be performed. Among these, the most common consists of removing from inside a loop the computation of subexpressions whose value remains constant during different iterations.

E. **Code generation**

Starting with optimised intermediate form, the final object code is generated. There follows, in general, a last phase of optimisation which depends upon the specific characteristics of the object language. In a compiler that generates machine code, an important part of this last phase is register assignment (decisions as to which variables should be stored in which processor registers). This is a choice of enormous importance for the efficiency of the final program.

3.2.3 Interpreter

An Interpreter is also a program that translates high-level source code into executable code. However the difference between a compiler and an interpreter is that an interpreter translates one

line at a time and then executes it: no object code is produced, and so the program has to be interpreted each time it is to be run. If the program performs a section code 1000 times, then the section is translated into machine code 1000 times since each line is interpreted and then executed.

Interpretation is a method of executing a program. The program is read as input by an interpreter, which performs the actions written in the program Figure 4.

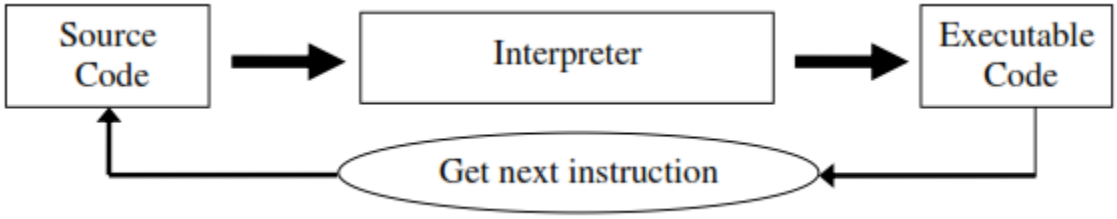


Figure 4: Interpretation Process

An interpreter is composed of two parts: a parser and an evaluator. After a program is read as input by an interpreter, it is processed by the parser. The parser breaks the program into language components to form a parse tree. The evaluator then uses the parse tree to execute the program.

Advantages of an Interpreter

1. Good at locating errors in programs
2. Debugging is easier since the interpreter stops when it encounters an error.
3. If an error is deducted there is no need to retranslate the whole program

Disadvantages of an Interpreter

1. Rather slow
2. No object code is produced, so a translation has to be done every time the program is running.
3. For the program to run, the Interpreter must be present

4.0 CONCLUSION

Programming language specification has been described as a definition that is necessary for the implantation of the programming language by users. The forms of a programming language

specification are formal Language Specification which represents its syntax and semantics, description of the behavior of a compiler, and model implementation. The syntax of a programming language is usually described using a combination of regular expressions and context free grammars. Programming language semantics can be described by natural language, formal semantics which is a description by mathematics, reference implementation that is, description by computer program, and test suites which are description by examples of programs and their expected behaviors. Another characteristic of programming language specification is the programming language reference which is written for users and developers, and describes the basic elements of the language and how to use them in a program.

As we already know that computer cannot execute instruction other than its native language, any instruction given to it other than machine language needs to be translated to machine codes. The three types of translator: assemblers, interpreters and Compilers were discussed in this unit. The assembler translates mnemonic operation codes into machine code

A compiler translates a program written in one high level language, the source code into another language which is the object code. An Interpreter is also a program that translates high-level source code into executable code. However the difference between a compiler and an interpreter is that an interpreter translates one line at a time and then executes it: no object code is produced.

5.0 SUMMARY

In this unit, we have discussed the relevance of programming language specification. The forms of programming language specification, the syntax and semantics of a programming language, programming language reference or language reference manual as the part of the documentation associated with most programming languages.

The discussion also covers programming language translation and translators where assemblers, compilers and interpreters were explained.

6.0 TUTOR-MARKED ASSIGNMENT

1. Explain what is meant by programming language specification.
2. Explain the forms of programming language specification

3. Describe the syntax and semantics of a programming language
4. Explain programming language reference or language reference manual
5. Explain programming language translation
6. Describe an assembler
7. Describe a compiler and explain different phases of compilation
8. Describe an interpreter and states its advantages and disadvantages
9. The box below shows part of a high-level language computer program. Part A shows the program before it has been translated. Part B shows the program after it has been translated.

A	B
While colour = 'Red' do Writeln (Colour, make);	1010 1101 1011 1111 1111 1001 1101 1110 1101 1101

a. Compilers, interpreters and assemblers are all translation programs. Which one of the three translators would be necessary to translate program A into program B.
b. What general name is given to the code, similar to B, produced after translation?

10. Interpreters and assemblers differ in the way they translate computer programs and the type of programs they translate. State two such differences.

7.0 REFERENCES/FURTHER READINGS

1. Forouzan, B. and Mosharaf, F. (2011). *Foundations of Computer Science.* BookPower United Kingdom (2nd ed).
2. French C. S. (1996). *Computer Science*. BookPower United Kingdom (5th ed).
3. Cooke, D. A. (2003). *Concise Introduction to Computer Languages*. Pacific Grove, CA: Brooks/Cole
4. Tucker, A. and Noonan, R. (2002). *Programming Languages: Principles and Paradigms.* McGraw-Hill.
5. Sebester, R. (2006). *Concepts of programming languages.* Addison Wesley.

6. Pratt, T. W. and Zelkowitz, M. V. (1999). *Programming Languages: Design and implementation* Prentice Hall (3rd ed).
7. Johnew Zhang (2012). *CS 241 Notes: Foundations of Sequential Programming*. Available online
8. Maurizio, Gabbrielli and Simone, Martini (2010). *Programming Languages: Principles and Paradigms.* Springer-Verlag London Limited. Available online
9. https://en.wikipedia.org/wiki/Programming_language_specification