**MODULE NO: 4**

**MODULE TITLE: BASIC PROGRAMMING**

**WRITER: ENGR. ANDREANNE MAE DOLANAS**

To do well in this module, you need to remember the following:

* Pause and pray before starting this module.
* Read and go through the module at your own time and pace.
* Honestly answer the activities.

**OPENING PRAYER**

May God the Father bless us.

May God the Son heal us.

May God the Holy Spirit enlighten us,

and give us eyes to see with,

ears to hear with,

hands to do the work of God with,

feet to walk with,

a mouth to preach the word of salvation with,

and the angel of peace to watch over us and lead us at last,

by our Lord's gift, to the Kingdom.

Amen.

**PRE-ASSESSMENT**

Quiz. To be posted.

**INTRODUCTION: MODULE INTRODUCTION AND FOCUS QUESTION:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Subtopic | Learning Outcome | Estimated Time |
| 4.1 | **Introduction** | Familiarize yourself with the process of programming. | 6 hours |
| 4.2 | **Programming** |
| 4.3 | **Assembly Language programming** |

**A close up of a logo

Description automatically generated**

**STUDY:**

The “vocabulary” of instructions which any particular [microprocessor](https://www.allaboutcircuits.com/textbook/digital/chpt-16/microprocessors/) chip possesses is specific to that model of chip. Unfortunately, there are no standards in place for microprocessor instructions.

When a human programmer develops a set of instructions to directly tell a microprocessor how to do something, they’re programming in the CPU’s own “language.” This language, which consists of the very same binary codes which the Control Unit inside the CPU chip decodes to perform tasks, is often referred to as machine language. While machine language software can be “worded” in binary notation, it is often written in hexadecimal form. But even with hexadecimal notation, these instructions can be easily confused and forgotten. For this purpose, another aid for programmers exists called *assembly language*.

With assembly language, two to four letter mnemonic words are used in place of the actual hex or binary code for describing program steps.

Once a program is developed by a person, it must be written into memory before a microprocessor can execute it. If the program is to be stored in [ROM](https://www.allaboutcircuits.com/textbook/digital/chpt-15/read-only-memory-rom/) (which some are), this can be done with a special machine called a *ROM programmer*, or (if you’re masochistic), by plugging the ROM chip into a breadboard, powering it up with the appropriate voltages, and writing data by making the right wire connections to the address and data lines, one at a time, for each instruction.

CREATING A PROGRAM:

The general steps for writing a program include the following:

* Understand the problem you are trying to solve.
* Design a solution.
* Draw a flow chart.
* Write the pseudocode.
* Write the code.
* Test and debug.
* Test with real users.
* Release the program.
* Iterate the steps for the next version.

Writing code:

Code is written in plain text, so that the compiler can read it. Compilers see formatting characters as syntax errors. A unique file extension is given to the document to indicate the nature of the code. For example, a file created using Python is saved with a .py extension, like 'myprogram.py.' However, the actual content of the file is still just plain text.

A code editor is also called an **integrated development environment**, or **IDE**. An IDE is a software application for formatting your code, checking syntax, as well as running and testing your code. Some IDEs can work with multiple programming languages, while some are very specific for only one language.

One very useful aspect of IDE is known as syntax highlighting. This means elements of the code are shown in different colors based on what they are.

An IDE includes tools for syntax checking, which is similar to checking grammar and spelling. If code contains syntax errors, the program will simply not execute. An IDE identifies exactly where the syntax errors are. Most IDEs also have some form of autocompletion system built in.

**Flowchart symbols:**

Diagram, shape, polygon

Description automatically generated

**STRAIGHT LINE PROGRAMS**

* The simplest type of program.
* The steps follow each other without any alternate routes.

**LOOPING**

* A section of the program will repeat over and over again.
* It can be made to repeat indefinitely or make it stop after a certain number of repetitions or if a condition has been met.

**BRANCHING**

* Allows us to write programs that can do different things at different times.

Diagram

Description automatically generated**Diagram

Description automatically generated**

Diagram

Description automatically generated

**SUBROUTINES**

* Another small program that works along the main one.

Diagram

Description automatically generated

SUBROUTINE

* The act of going to a subroutine is often referred to as calling a subroutine, at the end of which we return to the main program.
* The most important use of a subroutine is that it can be used several times in a program.

**PROGRAMMING LANGUAGES**

After constructing the flowchart, we need a language the computer understands.

**Machine Language**

* the only language the computer understands.
* Consists of 1s and 0s.

**Assembly Language**

* The first step toward a language that is easier for people to work with uses abbreviations to stand for different operations.
* These abbreviations are called mnemonics.
* Machine language and assembly language are low level languages.

The assembly language is called the source code, and for the microprocessor to execute it, it must be converted to machine language or object code. These are several ways to do this:

**Manual Assembly**

Write the program on paper using mnemonics. Then look up each mnemonic on a chart. On the chart there will be a number which is the machine language code for the assembly language mnemonic.

**Assembly with an Assembler or Monitor**

* A monitor is a program that is normally stored in ROM and gives access to the microprocessor’s various registers. It sometimes has in it a simple assembler to change mnemonics into machine code and a disassembler to change machine code back to mnemonics.

**High-level Programming Language**

There are many types of programming languages used but all of them are made of the same elements, which are:

* Programming Environment
* Basic Syntax
* Data Types
* Variables
* Keywords
* Basic Operators
* Decision Making
* Loops
* Numbers
* Characters
* Arrays
* Strings
* Functions
* File I/O

**Programming Environment**

you will need the following setup to start with programming using any programming language.

* A text editor to create computer programs.
* A compiler to compile the programs into binary format.
* An interpreter to execute the programs directly.

**Text Editor**

* A text editor is a software that is used to write computer programs. Your Windows machine must have a Notepad, which can be used to type programs.

**Compiler**

* The computer cannot understand your program directly given in the text format, so we need to convert this program in a binary format, which can be understood by the computer.
* The conversion from text program to binary file is done by another software called Compiler and this process of conversion from text formatted program to binary format file is called program compilation.

Diagram

Description automatically generated

**Interpreter**

* There are other programming languages such as Python, PHP, and Perl, which do not need any compilation into binary format, rather an interpreter can be used to read such programs line by line and execute them directly without any further conversion.

Diagram

Description automatically generated

**ASSEMBLY LANGUAGE**

**Advantages of Assembly Language**

Having an understanding of assembly language makes one aware of −

* How programs interface with OS, processor, and BIOS;
* How data is represented in memory and other external devices;
* How the processor accesses and executes instruction;
* How instructions access and process data;
* How a program accesses external devices.

Other advantages of using assembly language are −

* It requires less memory and execution time;
* It allows hardware-specific complex jobs in an easier way;
* It is suitable for time-critical jobs;
* It is most suitable for writing interrupt service routines and other memory resident programs.

**Addressing Data in Memory**

Let us consider a hexadecimal number 0725H. This number will require two bytes of memory. The high-order byte or most significant byte is 07 and the low-order byte is 25.

The processor stores data in reverse-byte sequence, i.e., a low-order byte is stored in a low memory address and a high-order byte in high memory address. So, if the processor brings the value 0725H from register to memory, it will transfer 25 first to the lower memory address and 07 to the next memory address.

Diagram

Description automatically generated

When the processor gets the numeric data from memory to register, it again reverses the bytes. There are two kinds of memory addresses −

* Absolute address - a direct reference of specific location.
* Segment address (or offset) - starting address of a memory segment with the offset value.

An assembly program can be divided into three sections −

* The **data** section,
* The **bss** section, and
* The **text** section.

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. The syntax for declaring data section is



The **bss** section is used for declaring variables. The syntax for declaring bss section is



The **text** section is used for keeping the actual code. This section must begin with the declaration **global \_start**, which tells the kernel where the program execution begins. The syntax for declaring text section is

Chart

Description automatically generated with medium confidence

**Comments**

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank.





Assembly language programs consist of three types of statements

* Executable instructions or instructions,
* Assembler directives or pseudo-ops, and
* Macros.

The **executable instructions** or simply **instructions** tell the processor what to do. Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.

The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.

**Macros** are basically a text substitution mechanism.

**SYNTAX**

Assembly language statements are entered one statement per line. Each statement follows the following format



The fields in the square brackets are optional.

Following are some examples of typical assembly language statements:

Text, letter

Description automatically generated

**Memory Segments**

A segmented memory model divides the system memory into groups of independent segments referenced by pointers located in the segment registers. Each segment is used to contain a specific type of data. One segment is used to contain instruction codes, another segment stores the data elements, and a third segment keeps the program stack.

We can specify various memory segments as −

* **Data segment** − It is represented by **.data** section and the **.bss**. The .data section is used to declare the memory region, where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

The .bss section is also a static memory section that contains buffers for data to be declared later in the program. This buffer memory is zero-filled.

* **Code segment** − It is represented by **.text** section. This defines an area in memory that stores the instruction codes. This is also a fixed area.
* **Stack** − This segment contains data values passed to functions and procedures within the program.

To speed up the processor operations, the processor includes some internal memory storage locations, called **registers**.

The registers store data elements for processing without having to access the memory. A limited number of registers are built into the processor chip.

**Processor Registers**

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories −

* General registers,
* Control registers, and
* Segment registers.

The general registers are further divided into the following groups −

* Data registers,
* Pointer registers, and
* Index registers.

**Data Registers**

Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways −

* As complete 32-bit data registers: EAX, EBX, ECX, EDX.
* Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
* Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.

Table

Description automatically generated

**AX is the primary accumulator**; it is used in input/output and most arithmetic instructions.

**BX is known as the base register**, as it could be used in indexed addressing.

**CX is known as the count register**, as the ECX, CX registers store the loop count in iterative operations.

**DX is known as the data register**. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

**Pointer Registers**

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers −

* **Instruction Pointer (IP)** − The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
* **Stack Pointer (SP)** − The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
* **Base Pointer (BP)** − The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

**Index Registers**

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers −

* **Source Index (SI)** − It is used as source index for string operations.
* **Destination Index (DI)** − It is used as destination index for string operations.

**Control Registers**

The 32-bit instruction pointer register and the 32-bit flags register combined are considered as the control registers.

Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

The common flag bits are:

* **Overflow Flag (OF)** − It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
* **Direction Flag (DF)** − It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
* **Interrupt Flag (IF)** − It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
* **Trap Flag (TF)** − It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.
* **Sign Flag (SF)** − It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
* **Zero Flag (ZF)** − It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
* **Auxiliary Carry Flag (AF)** − It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
* **Parity Flag (PF)** − It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
* **Carry Flag (CF)** − It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a *shift* or *rotate* operation.

**Segment Registers**

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments −

* **Code Segment** − It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
* **Data Segment** − It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
* **Stack Segment** − It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

In assembly programming, a program needs to access the memory locations. All memory locations within a segment are relative to the starting address of the segment. A segment begins in an address evenly divisible by 16 or hexadecimal 10. So, the rightmost hex digit in all such memory addresses is 0, which is not generally stored in the segment registers.

The segment registers stores the starting addresses of a segment. To get the exact location of data or instruction within a segment, an offset value (or displacement) is required. To reference any memory location in a segment, the processor combines the segment address in the segment register with the offset value of the location.

Most assembly language instructions require operands to be processed. An operand address provides the location, where the data to be processed is stored.

When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source. Source contains either the data to be delivered (immediate addressing) or the address (in register or memory) of the data. Generally, the source data remains unaltered after the operation.

The three basic modes of addressing are −

* Register addressing
* Immediate addressing
* Memory addressing

**Register Addressing**

In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

Text

Description automatically generated with medium confidence

As processing data between registers does not involve memory, it provides fastest processing of data.

**Immediate Addressing**

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

Text

Description automatically generated

**Direct Memory Addressing**

When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required. This way of addressing results in slower processing of data.

To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value. This offset value is also called **effective address**.

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name. The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

In direct memory addressing, one of the operands refers to a memory location and the other operand references a register.



**Direct-Offset Addressing**

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data

Text

Description automatically generated

**The MOV Instruction**

We have already used the MOV instruction that is used for moving data from one storage space to another. The MOV instruction takes two operands. The syntax of the MOV instruction is



Text

Description automatically generated

Please note that:

* Both the operands in MOV operation should be of same size
* The value of source operand remains unchanged

**ANALYSIS:**

A picture containing table

Description automatically generated

**RESEARCH:**

Create a program using assembly language following the steps mentioned in the module. Explain what the goal of the program is and how it works.

**ACTION: ASSESSMENT:**

Quiz. To be posted.

**ENDING PRAYER: Glory be…**

**REFERENCES:**

Brown, Jerald A. Malvino, Albert Paul. Digital Computer Electronics 3rd ed. McGraw Hill.

https://study.com/academy/lesson/how-to-write-a-program-coding-testing-debugging.html

https://www.tutorialspoint.com/computer\_programming/index.htm

https://www.tutorialspoint.com/assembly\_programming/index.htm