**UNIVERSITY OF AGRICULTURE, ABEOKUTA**

**OGUN STATE, NIGERIA**

| | |
|---|---|
| **Course Code** | CSC 303 |
| **Course Title** | ASSEMBLY LANGUAGE PROGRAMMING |
| **Course Lecturer** | Dr. ONASHOGA, S. A. (Mrs.)<br>DEPT. OF COMPUTER SCIENCE<br>UNIVERSITY OF AGRICULTURE,<br>ABEOKUTA, OGUN STATE<br>NIGERIA. |

## COURSE REQUIREMENTS

This is a compulsory course for all students in the University. In view of this, students are expected to participate in all the course activities and have mininmum of 75% attendance to be able to write the final examination.

## COURSE CONTENTS

Binary number systems and other systems, types of encoding, mode of representation of data, e.g. integer, floating, packaged decimal, characteristics, basic structure of the computer instruction set and corresponding machine language, modes of addressing, instruction execution and flow of control programming in assembly language, input and output, subroutines and central sections macros, linkages interfacing, assembly language programmes.

## SECTION ONE

## INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

### 1.1 Programming language

Different programming language are as follows:

**(1) Machine language:**

E.g 5 8 1 3

0101 = 5 , 1000 = 8 , 0001 = 1 , 0011 = 3

**Advantages of machine language**

(A) It uses computer's storage more efficiently.

(B) It takes less time to process in a computer than any other programming language.

**Disadvantages of machine language**

(A) It is time consuming

(B) It is very tedious.

(C) It is subjected to human error.

**(2) LOW LEVEL LANGUAGE(LLL)**

**Advantages of LLL**

(a) It is more efficient than machine language.

(b) It may be useful for security reason.

**(3) HIGH LEVEL LANGUAGE (HLL)** e.g BASIC, Pascal, C++

Note: Compiler is used to convert HLL to machine Language. Every language has a type of translator to itself.

### 1.2 Why learn Assembling Language?

(1) There are still some programming tasks that are best done in Assembling Language for reasons of efficiency and access to machine capabilities not available in HLL.

(2) AL is just a way of expressing the actual native language of the computer, namely machine code, the study of assembly is therefore in a sense a study of the machines and its architecture.

(3) Its study helps to develop a deeper understanding of computer system.

**Advantages of machine language**

It is the easiest form of a program for the machine to understand.

**Disadvantages**

Very tedious, errors prone and time consuming.

Very difficult for human to read.

## 1.3 ASSEMBLY ASPECTS OF ASSEMBLY LANGUAGE PROGRAMMING

This course, Assembly language programming deals with the software aspects of assembly language , assemblers and machine language. It also deals with the hardware aspects of what the computers does to execute programs.

It is an introduction to the study of computer architecture , the interface between hardware and software.

### 1.3.1 COMPUTER ARCHITECTURE

The relationship between hardware (stuff you can touch) and software (programs, code). I can design a computer that has hardware which executes programs in any programming language. For example: a computer that directly executes Pascal.

So, why don't we do just that?

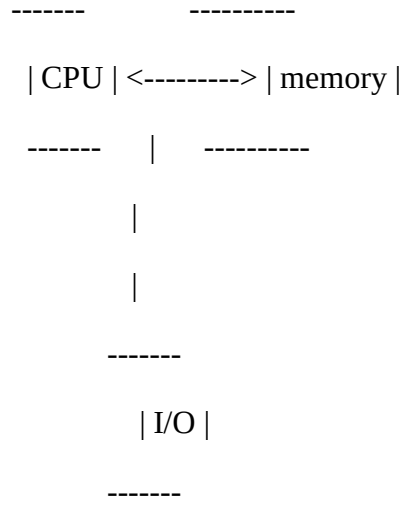
1. From experience (in the engineering community), we know that the hardware that executes HLL programs directly are slower than those that execute a more simple, basic set of instructions.

2. Usability of the machine. Not everyone wants a Pascal machine. ANY high level language can be translated into assembly language.

In this class, in whatever language you are writing programs, it will look like you have a machine that executes those programs directly.

**BASIC COMPUTER OPERATION**

Simplified diagram of a computer system (hardware!)




CPU -- controls the running of programs, executes instructions, makes requests of the memory. CPU stands for central processing unit.

CPU and processor are synonyms (book uses the term processor)

NOTE: Many PC users incorrectly identify the term CPU with whatever is in the box that their display sits on top of. Chances are the real CPU is inside that box, but there will be many more things in there as well.

memory -- where programs and program variables are stored handles requests from the CPU

**Fetch and execute instruction**

How does the CPU execute a program ? CPU executes a program by repeatedly fetching one instruction from memory and executing the instruction in an endless cycle called the Fetch/ Execute Cycle.

To execute a program , CPU implements this cycle in five steps as below:

Table 1: Steps in Fetch Execute Cycle

| General Fetch/ Execute | |
|---|---|
| (1) Initialize CPU | IP = 0 |
| (2) Fetch instruction at IP (PC) | MAR = IP |
| | MDR = Memory [MAR] |
| | IR = MDR |
| (3) PC = next instruction address | IP = IP + 1 |
| (4) Decode instruction | OP = IR operation failed |
| | Addr = IR address failed |
| (5) Execute operation OP | |
| (6) GOTO 2 | |

*Intel calls PC an IP, others call it PC

## SECTION TWO

## DATA REPRESENTATION

### 2.1 NUMBER SYSTEM

### BINARY TO DECIMAL CONVERSION.

The Binary number system uses just two symbols , 0 or 1

4 3 2 1 0

$1\ 1\ 0\ 1\ 1_2$

$2^4 + 2^3 + 0 + 2^1 + 1$

16 + 8 + 2 + 1 = 27.

### DECIMAL TO BINARY CONVERSION.

Repeated division by 2 for this conversion. For example

25 / 2 = 12 r 1 ↑

12 / 2 = 6 r 0

6 / 2 = 3 r 0

3 / 2 = 1 r 1

1 / 2 = r 1

1 1 0 0 1

$25_{10} = 1\ 1\ 0\ 0\ 1_2$

### OCTAL TO DECIMAL CONVERSION:

An octal number can be early converted to its decimal equivalent in multiplying each octal digit by its positional weight.

**For example:** $3\ 7\ 2_8 = 3 * 8^{2\ +}\ 7 * 8^1 + 2 * 8^0$

$= (3 * 64) + (7 * 8) + (2 * 1)$

$= 250_{10}$

**Another example;**

$2\ 4\ .\ 6 = 2 * 8^1 + 4 * 8^0 + 6 * 8^{-1}$

$= 16 + 4 + 0.75$

$= 20.75_{10}$

**DECIMAL TO OCTAL CONVERSION**

Convert $109_{10}$ to base 8

Using repeated division by 8

109/8 = 13 r 5

13/8 = 1 r 5

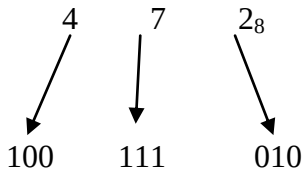
1/8 = 0 r 1

$109_{10} = 155_8$

**OCTAL TO BINARY CONVERSION:**

The conversion is done by converting each octal digit to its 3-bit binary equivalent

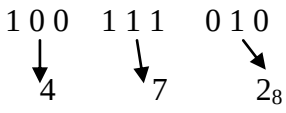
for example:




$472_8 = 100111010_2$

**BINARY TO OCTAL CONVERSION**

To do this, the bits of the binary number are grouped into groups of three bits at the LSB, then each group is converted to its octal equivalent.



**Hexadecimal to Decimal conversion:**

These are numbers to base 16 with 16 possible digit symbols. 0 through 9 plus the letter A B C D E F which are equivalent to 10 11 12 13 14 15 16 decimal.

$3\ 5\ 6 = 3 * 16^2 + 5 * 16^1 + 6 * 16^0$

$= 768 + 80 + 6$

$= 2854_{10}$

$2\ A\ F_{16} = 2 * 16^2 + 10 * 16^1 + 15 * 16^0$

$= 512 + 160 + 15$

$= 687_{10}$

**DECIMAL TO HEXADECIMAL CONVERSION**

$10\ 9_{10}$

Using the repeated division

$109/16 = 6\ r\ 13 = D$

$6 / 16 = 0\ r\ 6$

$109_{10} = 6D_{16}$

$214_{10}$ to hex

$214/16 = 13\ r\ 6$

$13 / 16 = 0\ r\ 13 = D$

### 2.1.1 INTEGER REPRESENTATION

**Integer representation:**

**True magnitude form**

e.g +52 = 00110100

-52 = 10110100

In this representation , the msb represents the sign 0 for +ve and (for –ve , The remaining part represents the magnitude of the number).

The 1's complement form of any binary number is obtained simply by changing each 0 in the number to a 1 and each 1 to a 0.

e.g 1's complement of 1101101 is 0010010

When –ve numbers are represented in 1's complement form, the sign bit is made as 1 and the magnitude as converted from true binary form to its 1's complement.

e.g -52 = 10110100 (true magnitude form)

= 11001011(1's complement form)

2's complement form:

e.g 00110100

11001011 (1's complement add 1 to LSB to form 2's complement )

1

11001100

**Class Work:** Confirm that 2's complement of -419 on a 16-bit machine is FE5D.

Convert $7245_8$ to 8's complement.

7777

7245

0532

\+ 1

0533

## 2.2 FLOATING POINT DATA REPRESENTATION

Convert $384_{10}$ to floating point

$384 = 110000000_2$

normalise = $.11 * 2^9 = .11 * 2^{1001}$

9 = 1001 in binary

Take to hex = $9_h$

Fraction = 11

Exponent = $1001_2 = 9_h$

Sign = 0

Biased exponent = $9_h + 7E_h$

$= 87_h$ to binary

$= 10000111_2$

**Representation**

Sign Exponent fraction

0 10000111 10000000000000000000000

Memory content = $4\ 3\ C0\ 00\ 00_h$

## 2.3 NON-NUMERIC CHARACTER REPRESENTATION

**ASCII code:**

e.g The following is a message encoded in ASCII code , What is the message?

1001000 1000101 1001100 1010000

**solution**

convert the 7-bit code to its hex equivalent, The results are

48 45 4c 50

Now locate these hex values in the table and determine the xter represented by each.

The results are H E L P

The first 3-bits is called the **ZONE bits** and the last 4, **the NUMERIC bits**

For ASCII

| Xters | Zone | Numeric bit |
|---|---|---|
| 0-9 | 011 | 0000-1001 |
| A-0 | 100 | 0001-1111 |
| P-Z | 101 | 0000-1010 |
| a-o | 110 | 0001-1111 |
| p-z | 111 | 0000-1010 |

EBCDIC code

The EBCDIC representation can be summarised as follows

| characters | Zone-Bits | Numeric Bits |
|---|---|---|
| 0-9 | 1111 | 0000-1001 |
| A-I | 1100 | 0001-1001 |
| J-R | 1101 | 0001-1001 |
| S-Z | 1110 | 0001-1000 |

e.g ADE2 in EBCDIC

A - 1100 0001

D - 1100 0100

E - 1100 0101

2 - 1111 0010

ADE2 = 11000001110001001100010111110010

## SECTION THREE

## ERROR IN DATA TRANSMISSION

### 3.1 INTRODUCTION

The movement of binary data and codes from one location to another is most frequent operation performed in digital systems. For example:

- ✓ The reading of instruction codes and data from internal memory as a computer executes a program.
- ✓ The storage and retrieval of data from external memory devices such as magnetic tape and disk.
- ✓ The transmission of information from a computer to a remote user terminal or another computer.

### 3.2 PARITY METHOD

**Parity Bit**

It is an extra bit that is attached to a code group that is being transferred from one location to another. The Bit is made up of 0 or 1 depending on the number of 1s that are contained in the code group.

Even Parity

e.g 1000011 = ASCII character
Thus, it becomes 1<u>1000011</u>
parity bit
added

Odd Parity : The Odd parity is used except that the binary bit is chosen so that the total number of 1s including the parity bit is an odd number.

e.g 1000001 = 11000001
added

It would be apparent that this parity would not work if two bits were in error , because two errors would not change the “oddness” or “evenness” of the number of 1s in the code.

Exercise: A transmitter is sending ASCII coded data to a receiver with an even-parity bit. Show the actual code when the transmitter is sending the message “HELLO”.

## SECTION FOUR

## ASSEMBLY LANGUAGE PROGRAMMING

### 4.1 PROCESS OF ASSEMBLY

The process of creating working assembly language programs involves a number of steps , which I will describe in a general way.

Later , we will see how to carry out these steps in details.

Assembly language is a compiled language , in the sense that assembly language source-code must first be created with a text-editor program, and then the source-code must be compiled. Assembly language compilers are universally call “assemblers”.

Five types of auxiliary programs are commonly used in 8088 assembly language programming . First , as mentioned above, is the text-editor , which is used to type in assembly language source code and then to edit it when errors are discovered . Second is the assembler. The assembler “assembles ” the source code , creating “object” code in the process. The object code is neither executable nor human-readable. The third program is the linker . The linker combines object code modules created by the assembler or by various high-level compilers.

For example, If we wrote a program yesterday to convert hexadecimal numbers to decimal, and we write a program today to convert decimal numbers to hexadecimal, then we may want to write a third programme tomorrow which, when linked with this, reads two-decimal numbers from the keyboard, converts them to hexadecimal, adds them, and writes the back to the screen in decimal. The fourth programme, the loader, is actually built into the operating system and is never explicitly executed. The loader takes the “relocatable” code created by the linker, “loads” it into the memory at the lowest available location and runs it.

### 4.2 INTEL 8088 AND ABOVE CPU REGISTERS

Generally (though not always) when we program in a high-level language we think in terms of the following types of constructs: CONSTANTS numerical, string, or some other quantities whose unchanging actual values are known when the program written VARIABLES quantities (whose initial values may or may not be known) whose values change as the program executes PROCEDURES functions or subroutines which may or may not have arguments and may or may not return answers

None of these items has any real direct equivalent in terms of assembly language. Each, in practice, is a combination of several assembly language features. In assembly language, on the

other hand, much thought goes into the use of the computer's memory (considered as a sequence of bytes or words) and the CPU's *registers*. A register is like a memory location in that it can store a byte (or word) value. [These register sizes apply to CPUs like the 8088, 8086, 8080, Z80, etc. The 68000 CPU has all 4-byte registers. The Z8000 CPU has registers that can be grouped in various ways to contain anything from one byte to 8 bytes. Some TI microprocessors have no registers at all.] However, a register has no address in the computer's memory. Registers are not a part of the computer's memory, but are built into the CPU itself.

Registers are so important in assembly language programming (on microcomputers) for various reasons. First, the variety of instructions using registers tends to be greater than that for operating on values stored at memory locations. Second, these instructions tend to be shorter (i.e., take up less room to store in memory). Third, register-oriented instructions operate faster than memory-oriented instructions since the computer hardware can access a register much faster than a memory location. The 8086-family of microprocessors have a number of registers, *all* of which are partially or totally dedicated to some specific type of use. Here is a list of the registers and their uses. Do not worry if their uses do not seem clear yet. For the present, it suffices for us that the italicized registers are so specialized that they can *only* be used for their special purpose, while the registers in normal type can often be used just like 16-bit (word) memory locations:

1. AX - The accumulator
2. BX - The pointer register
3. CX- The loop counter
4. DX- Used for multiplication and division
5. SI- The "source" string index register
6. DI -The "destination" string index register
7. BP- Used for passing arguments on the stack
8. *SP- The stack pointer*
9. *IP -The instruction pointer*
10. *CS- The "code segment" register*
11. DS -The "data segment" register
12. *SS -The "stack segment" register*
13. *ES- The "extra segment" register*
14. *FLAG- The flag register*

The first seven registers might reasonably be called "general purpose" registers since they can be used rather flexibly to manipulate word values until (or unless) their special functions are needed. AX, BX, CX, and DX are more flexible than the others in that they may be used either as word registers (containing 16-bit values) or as pairs of byte registers (containing 8-bit values). The byte-sized registers gotten this way are known as AL, BL, CL, DL, AH, BH, CH, and DH. For example,

AL contains the less significant byte of AX, while AH contains the more significant byte.

Several of these special register types are common among microprocessors: The accumulator is often a special register which is designated to contain the results of certain arithmetic operations. Many instructions execute faster when operating on the accumulator then they do when operating on other registers, which are in turn faster than operations on memory variables. The 8088 has the 8-bit accumulator AL and the 16-bit accumulator AX. The instruction pointer (or program counter) is a register controlling the execution of programs. Recall that both programs and data are stored in the computer's memory. Most program code is stored in memory in such a way that sequentially executed instructions are actually stored sequentially in memory. The IP (instruction pointer) register contains the address of the next instruction to be executed.

For every instruction fetched from memory, the IP is automatically incremented by the number of bytes in the instruction. The stack pointer (SP) contains the address of the next memory location to the added to the stack. We will discuss stacks later. The flag register contains a number of bit-sized "flags" describing the status and configuration of the CPU. Its main use is in controlling conditional execution of parts of a program.

### 4.3 MODES OF ADDRESSING THE REGISTERS

Table 2 shows different types of addressing mode, with the calculation of how these addresses are generated.

**Table 2: Addressing Modes**

| TYPE | INSTRUCTION | SOURCE | ADDRESS GENERATION | DESTINATION |
|---|---|---|---|---|
| Register | MOV AX,BX | Register BX | | Register AX |
| Immediate | MOV CH, 3AH | Register 3AH | | Register CH |
| Direct | MOV [1234H],AX | Register 3AH | DSx10H+DISP<br>10000H+1234 | Memory address 11234H |
| Register indirect | MOV [BX],CL | Register CL | DSx10H+BX<br>10000H+0300 | Memory address 10300H |
| Base-plus-index | MOV [BX+SI],BP | Register SP | DSx10H+BX+SI<br>10000H+0300H+020 | Memory address 10500H |
| Register relative | MOV CL,[BX+4] | Memory address 10304H | DSx10H+BX+4<br>10000H+0300H+4 | Register CL |
| Base Relative plus-index | MOV ARRAY [BX+SI],DX | Register DX | DSx10H+ARRAY+BX+SI<br>10000H+1000H+0300H+020 | Memory address 11500H |
| Scaled index | MOV [EBX+2xESI], AX | Register AX | DSx10H+EBX+2+ESI<br>10000H+00000300H+000004 | Memory address 10700H |

Notes: EBX = 000000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

## 4.4 ASSEMBLY LANGUAGE INSTRUCTION

### 4.4.1 INSTRUCTION FORMAT

Each statement in a program consists of four parts or fields in the following format:

*Label Opcode Operand Comment*

**Label** – used to store a symbolic name for the memory location that it represents. All labels begin with a letter or one of the following special characters : , $ or ?. A label may be of any length from 1-35 characters .The label appears in a program to identify the name of a memory location for storing data.

**Opcode** - is designed to hold the instruction or opcode e.g ADD

**Operand** – Contains information used by the Opcode e.g in MOV AL, BL instruction has opcode MOV and operands BL and AL

**Comment** - begins with a ';'

### 4.4.2 ASSEMBLY LANGUAGE INSTRUCTION

Types of instructions

i. Transfer instruction

**MOV INSTRUCTION**

Purpose: Data transfer between memory cells, registers and the accumulator.

Syntax: MOV Destination, Source

Where Destination is the place where the data wiil be moved and Source is the place whrere the data is.

ii. Stack Instructions

These instructions allow the use of the stack to store or retrieve data.

Purpose: It recovers a piece of information from the stack

Syntax: POP destination

Examples:

POP

POPF

PUSH

PUSHF

POP instruction

PUSH instruction

The PUSH instruction decreases by two the value of SP and then transfers the content of the source operator to the new resulting address on the recently modified register.

iii. Arithmetic instruction e.g. MUL instruction

Purpose: multiplication with sign

Syntax: MUL source

iv. INC and DEC instructions

Syntax: INC destination ; *destination = destination + 1*

DEC destination ; *destination = destination - 1*

Here are other some simple instructions you should know to get you started:

Instruction Description

ADD* reg/memory, reg/memory/constant ------- Adds the two operands and stores the result into the first operand. If there is a result with carry, it will be set in CF.

SUB* reg/memory, reg/memory/constant ----- Subtracts the second operand from the first and stores the result in the first operand.

AND* reg/memory, reg/memory/constant ----- Performs the bitwise logical AND operation on the operands and stores the result in the first operand.

OR* reg/memory, reg/memory/constant ----- Performs the bitwise logical OR operation on the operands and stores the result in the first operand.

XOR* reg/memory, reg/memory/constant ----- Performs the bitwise logical XOR operation on the operands and stores the result in the first operand. Note that you cannot XOR two memory operands.

MUL reg/memory ----- Multiplies the operand with the Accumulator Register and stores the result in the Accumulator Register.

DIV reg/memory ----- Divides the Accumulator Register by the operand and stores the result in the Accumulator Register.

INC reg/memory ----- Increases the value of the operand by 1 and stores the result in the operand.

DEC reg/memory ---- Decreases the value of the operand by 1 and stores the result

in the operand.

NEG reg/memory ----- Negates the operand and stores the result in the operand.

NOT reg/memory ---- Performs the bitwise logical NOT operation on the operand and stores the result in the operand.

PUSH reg/memory/constant ---- Pushes the value of the operand on to the top of the stack.

POP reg/memory---- Pops the value of the top item of the stack in to the operand.

MOV* reg/memory, reg/memory/constant---- Stores the second operand's value in the first operand.

CMP* reg/memory, reg/memory/constant ---- Subtracts the second operand from the first operand and sets the respective flags. Usually used in conjunction with a JMP, REP, etc.

JMP**---- label Jumps to label.

LEA reg, memory ---- Takes the offset part of the address of the second operand and stores the result in the first operand.

CALL subroutine Calls another procedure and leaves control to it until it returns.

RET Returns to the caller.

INT constant Calls the interrupt specified by the operand.

**SAMPLE PROGRAM ON TRANSFER INSTRUCTION**

In FORTRAN :

A = B + 1

C = D – 1 (where A,B,C,D ARE INTEGERS*2)

ASSEMBLY code equivalent:

**; Version 1**

(A, B, C and D are all defined as DW ?)

```
MOV AX, B ; A = B
MOV A , AX; C = D
MOV AX , D
MOV C, AX; C = D
ADD A, 1; A = A + 1
SUB C, 1; C = C - 1
```

**; Version 2**

```
MOV AX, B
ADD AX, 1
MOV A, AX; A = B + 1
MOV AX, D
SUB AX, 1
MOV C, AX
```

## SECTION FIVE

## ASSEMBLY PROGRAMS

### 5.1 PROGRAMMING OVERVIEW

### 5.1.1 ASSEMBLING LANGUAGE OVERVIEW

An assembly program is written using a simple text editor. Each assembler has specific syntax rules regarding the structure of the source file and the names that are used to represent assembler directives , opcodes , and operands . There are also syntax rules regarding comments in the file.

**Assembler process:**

Create source file using a text editor and save it (.ASM)

Execute commands from a DOS prompt to assembler your text file and create an output hex file with a .HEX extension (e.g ASM51) <filename> [options])

If errors occur during the assembly, edit the source file to correct the syntax error. A listing file (.LST) may be used to see what error the assembler encountered. (e.g, to create a .LST file, use: ASM51<filename>-F)

Once the assembler executes without error, load the .HEX file into a simulator, or into your target hardware (into EPROM , flash , or RAM)

Execute your code and continue the debugging process

### 5.1.2 Getting Started

To program in assembly, you will need some software, namely an assembler and an editor. There is quite a good selection of Windows programs out there that can do these jobs.

An **Assembler** takes the written assembly code and converts it into machine code. Often, it will come with a linker that links the assembled files and produces an executable from it. Windows executables have the .exe extension. Here are some of the popular ones:

1. **MASM** – This is the assembler this tutorial is geared towards, and you should use this while going through this course. Originally by Microsoft, it's now included in the MASM32v8 package, which includes other tools as well. You can get it from http://www.masm32.com/

   **ABOUT MASM**

   - One instruction , declaration per line
   - Comments are anything on a line following ';'

For declaration, MASM has 3 basic types; integer, float(real) and character

In Pascal var sum: integer

*Declaration*

In Pascal: VAR *variablename* : *type*;
In C or C++: *TYPE variablename* e.g. int sum, String sum
In MASM : *Variablename* TYPE *value*

Type is **dd** if integer ; define doubleword (allocates a memory space of 32 bits)

**db** if character; define byte (allocates a memory space of 8bits)

**dd** if floating point;

dw; define word (allocates a memory space of 16 bits, this could be for integer as well )

value is required -- it gives the variable an initial value

-- to explicitly leave value undefined, use the '?' character

EXAMPLES

```
counter dd 0 ; tells the assembler to allocate 32 bits
variable4 dd ?
constant dd 2.71828
letter db 'a' ; tells the assembler to allocate 8 bits
string10 db 'This is a string' , 0  ; null terminated string example

string4 db  'Another string', Oah , 0 ;Oah is the newline character and this is
                string

; null terminated
```

2. **TASM** – Another popular assembler. Made by Borland but is still a commercial product, so you can not get it for free.

3. **NASM** – A free, open source assembler, which is also available for other platforms. It is available at http://sourceforge.net/projects/nasm/. Note that NASM can't assemble most MASM programs and vice versa.

**Editors:** An editor is where you write your code before it is assembled. Editors are personal preferences; there are a LOT of editors around, so try them and pick the one you like.

1. **Notepad** – Comes with Windows; although it lacks many features, it's quick and simple to use.

2. **Visual Studio** – Although it's not a free editor, it has excellent syntax highlighting features to make your code much more readable.

## 5.2 USING DEBUGGER

### WHAT IS A DEBUGGER?

A debugger displays the contents of memory and lets you view registers and variables as they change. You can step through a program one line at a time (called Tracing), making it easier to find logic errors.

### DEBUGGING FUNCTIONS

Some of the most rudimentary functions that any debugger can perform are the following

- Assemble short programs
- View a program's source code along with its machine code
- View the CPU registers and flags
- Trace or execute a program, watching variables for changes
- Enter new values into memory
- Search for binary or ASCII values in memory
- Move a block of memory from one location to another
- Fill a block of memory
- Load and write disk files and sectors

Debug commands may be divided into four categories namely

***Program Creation and Debugging***

i. A Assemble a program using instruction mnemonics

e.g. A 100 ; Assembles at CS:100h

When you press Enter at the end of each line, debug prompts you for the next line of input. Each input line starts with a segment-offset address. To terminate input, press the Enter key on a blank line. For example:

- A 100
  5514:0100 MOV AH, 2
  5514: 0102 MOV DL, 41
  5514: 0104 INT 21
  5514: 0106

| | | |
|---|---|---|
| ii. | G | Execute the program currently in memory |
| iii. | R | Display the contents of registers and flags |
| iv. | P | Proceed past an instruction, procedure, or loop |
| v. | T | Trace a single instruction |
| vi. | U | Disassemble memory into assembler mnemonics |

***Memory Manipulation***

| | | |
|---|---|---|
| i. | C | Compare one memory range with another |
| ii. | D | Dump (display) the contents of memory |
| iii. | E | Enter bytes into memory |
| iv. | F | Fill a memory range with a single value |
| v. | M | Move bytes from one memory range to another |
| vi. | S | Search a memory range for specific value(s) |

***Miscellaneous***

| | | |
|---|---|---|
| i. | H | Perform hexadecimal addition and subtraction |
| ii. | Q | Quit Debug and return to DOS |

***Input-Output***

| | | |
|---|---|---|
| I. | I | input a byte from a port |
| II. | L | load data from disk |
| iii. | O | Send a byte to a port |
| iv. | N | Create a filename for use by the L and W commands |
| v. | W | Write data from memory to disk |

### 5.2.1 ASSEMBLY PROGRAMS

A program that multiplies AX register by 10 , store the result in the BX register

Example 1:

**In Pascal**

```
Var   ax, bx, cx : integer
begin
    bx:= 0;
for cx := 10 downto 1 do bx:= bx + ax
end;
```

We have used **downto** in the **for** loop rather than **to** because in assembler loops with count down are much easier to implement than loops which count up.

```
mov   bx, o ; the BX register holds the running sum
mov   cl, 10 ; this time , use CL as the loop counter
again:
        add bx, ax    ; bx:= bx+ ax
        dec  cl  ; decrement the loop counter
        jnz   again ; repeat only if the loop counter isn't zero
```

In DEBUG , the program would look like this:

```
-A100
4410:0100   MOV   BX ,0
4410:0103   MOV   CL ,10
4410:0105   MOV   BX ,AX
4410:0107   DEC  CL
4410:0109   JNZ   105
4410:010B
```

To unassembled:

```
-U100 , 10A
4410:0100    BB0000          MOV   BX ,0000
4410:0103    B110             MOV   CL ,10
4410:0105    01C3             ADD   BX ,AX
4410:0107    FEC9             DEC   CL
4410:0109    75FA             JNZ   0105
```

**Example 2:** To add 1 to 10 --- A detailed Version

```
.data  ; data segment
answer  label byte
            db ‘the number is ’, 0
i    label   byte
     db    dup(?)
sum label  byte
       db  0
.code
main proc  near
      mov  i, 1
     jmp     L1
     jmp   L2
L2  :      mov ax, i
           add  sum, ax
           inc   i
L1 :    cmp   i,  11
         j L   L2
          push sum
     mov  ax, offset answer
       push  ax
        call  _printf
.end
```

## SECTION SIX

## MACROS AND PROCEDURES

A macros is a group of repetitive instructions on a program which are codified only once and can be used as many times as necessary.

The main difference between a macro and a procedure is that in the macro the passage of parameters is possible and in the procedure it is not, this is only applicable to the TASM – there are other programming languages which do not allow it. At the moment the macro is executed each parameter is substituted by the name or value specified at the time of the call.

We can say then that a procedure is an extension of a determined program, while the macro is a module with specific functions which can be used by different programs.

Another difference between a macro and procedure is the way of calling each one, to call a procedure the use of directive is required , on the other hand the call of macros is done as if it were an assembler instruction.

SYNTAX OF A MACRO

The parts which make a macro are:

Declaration of the macro

Code of the macro

Macro termination directive

The declaration of the macro is done the following way:

NameMacro MACRO [parameter1, parameter2…]

Syntax of a Procedure

There are two types of procedures , The intrasegments, which are found on the same segments of instructions , and the inter-segments which can be stored on different memory segments.

To divert the flow of a procedure (calling it), the following directive is used:

CALL NameOfTheProcedure

The part which make a procedure are:

i. Declaration of the procedure
ii. Code of the procedure
iii. Return directive
iv. Termination of the procedure

For example, if we want a routine which adds two bytes stored in AH and AL each one , and keep the addition in the BX register.

```
Adding Proc Near ; Declaration of the procedure

Mov BX, 0; Content of the procedure

Mov B1, Ah

Mov Ah, 00

Add Bx, Ax

Ret ; Return directive

Add Endp ; End of procedure declaration
```

On the declaration of first word, Adding , corresponds to the name of out procedure , Proc declares it as such and the word Near indicates to the MASM that the procedure is intrasegment.

The Ret directive loads the IP address stored on the stack to original program, lastly , the Add Endp directive indicates the end of the procedure.

**REFERENCES**


1. D.J BRADLEY “ASSEMBLY LANGUAGE PROGRAMS FOR THE IBM PERSONAL COMPUTERS”

2 BERRY BREY ”THE INTEL PROCESSOR 8086/8088, 80186/80188, 80286, 80386, 80486, PENTIUM AND PENTIUM PRO PROCESSOR ARCHITECTURE PROGRAMMING INTERFACE”.