Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

LABORATORY MANUAL

Course No.: CSE2214
Course Title: Assembly Language Programming

For the students of 2nd Year, 2nd Semester of
B.Sc. in Computer Science and Engineering program

TABLE OF CONTENTS

COURSE OUTCOMES

After the successful completion of this course, students are expected to be able to:

Sl. No.	COs	POs	Bloom's Taxonomy		
			C	A	P
1.	Execute the fundamental concepts of the microprocessor operation at the address, data, and control level.	1			2
2.	Apply basic instructions and translate a number of small C/C++ programs into assembly language and trace and debug at the assembly level	2			3
3.	Analyze complex problems to show how C/C++ constructs are translated to execute on hardware, simple hardware operations and interrupt handling are crucial building blocks for the Operating Systems and Computer Architecture courses	4			4

PREFFERED TOOL(S)

- emu8086

TEXT/REFERENCE BOOK(S)

- Charles Marut & Ytha Yu, ***Assembly language Programming and Organization of the IBM PC,*** McGraw-Hill, Inc, 1992.

ADMINISTRATIVE POLICY OF THE LABORATORY

- ✓ Students must perform class assessment tasks individually without help of others.
- ✓ Plagiarism is strictly forbidden and will be dealt with punishment.
- ✓ Every student must bring the book in the lab.
- ✓ Mobile should not be in the table or in the hand and phone should be in silent mode.
- ✓ If any emergency call comes then you can receive the call by taking permission from teacher.
- ✓ Lab classes cannot be switched without valid reason and without informing the teachers.
- ✓ Hardcopy assignment must be submitted before teachers enter into the class.
- ✓ Softcopy assignment must be submitted the day before next lab within 8pm in the drive link provided by teacher.

Session 1

Microcomputer Systems

OBJECTIVES:

- Students will be able to learn the components of microcomputer system.
- They will know how to execute an instruction.
- Students will have a brief knowledge about I/O devices.
- They will learn about various types of programming languages.

Components of a Microcomputer System

- Memory

    1. Memory contents can be accessed either by byte or by word.
    2. Byte is 8-bit and word is 16-bit.
    3. Two operations can be performed on memory: read(fetch) and write(store).
    4. Memory can be three types: RAM, ROM and Cache.
    5. There are two types of RAM (Random Access Memory): DRAM: Dynamic RAM and SRAM: Static RAM.
    6. ROM, EPROM, EEPROM, and FLASH memory are examples of ROM (Read Only Memory)
    7. Cache is divided into two types: Internal Cache and External Cache.

- The CPU

    1. CPU consists of two units: Execution Unit (EU) and Bus Interface Unit (BIU).
    2. Execution unit consists of Arithmetic and Logic Unit (ALU) and eight registers for storing data: AX, BX, CX, DX, SI, DI, BP and SP. It also contains FLAGS register.
    3. Bus Interface Unit's registers are: CS, DS, ES, SS and IP holding address of memory locations.

- I/O Ports

    1. There are two types of I/O ports: Serial Port and Parallel Port.
    2. Serial port transfers one bit at a time.
    3. Parallel port transfers 8 or 16 bits at a time.
    4. Examples of I/O ports are Magnetic Disk, Keyboard, Display Monitor and Printers.
    5. Magnetic disk is of two types: Floppy Disk and Hard Disk.
    6. There are three types of printers: Daisy Wheel, Dot Matrix and Laser.

Buses

Processor communicates with memory and I/O devices by using signals that travel along a set of wires called buses. Three kinds of buses: address bus, data bus and control bus.

Instruction Execution (Fetch- Execution Cycle)

- Machine instructions have two parts: Opcode and Operands.
- The steps of executing an instruction are: Fetch and Execution.

Programming Languages

- Machine language: Example: 00100011
- Assembly language: Example: MOV AX, BX
- High Level Language: c = a + b

Compiler

It is used to translate high level instruction to assembly instruction or machine instruction.

Assembler

It is used to translate assembly instruction to machine instruction

Mapping Between Assembly Language and High-Level Language

Instruction Class	C Language	Assembly Language
Data Movement	A=5	MOV A,5

Arithmetic or Logic	B=A+5	MOV AX, A
		ADD AX,5
		MOV B, AX

Data Movement	goto LBL	JMP LBL

Sample Math

Suppose a processor uses 20 bits for an address. How many memory bytes can be accessed?

Answer:

The number of memory bytes will be $2^{20} = 1{,}048{,}576 = 1\text{MB}$

Session 2

Representation of Numbers and Characters

OBJECTIVES:

- Students will come to know about different types of number systems.
- They will be able to perform conversion between number systems.
- They will learn how to perform binary and hexadecimal addition and subtraction.
- They will have a brief idea about how integers and characters are represented in computer.

Number Systems

In computer we use three types of number systems:

1. Decimal Number System (0-9)
2. Binary Number System (0 and 1)
3. Hexadecimal Number System (0-9 and A-F)

Conversion of Binary and Hex to Decimal

Hexadecimal Number, 8A2Dh = Decimal Number, 35373d

Process:

$$8 \times 16^3 + A \times 16^2 + 2 \times 16^1 + D \times 16^0$$

$$= 8 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 13 \times 16^0$$

$$= 35373d$$

Conversion of Decimal to Binary and Hex

Decimal Number, 11172d = Hexadecimal Number, 2BA4h

Process:

$11172 = 698 \times 16 + 4$

$698 = 43 \times 16 + 10(A)$

$43 = 2 \times 16 + 11(B)$

$2 = 0 \times 16 + 2$

Conversion Between Hex and Binary

Binary Number, 1110101010b = Hexadecimal number, 3AAh

0011	1010	1010
3	A	A

Binary and Hex Addition and Subtraction

- **Addition**

5	B	3	9	h
7	A	F	4	h
D	6	2	D	h

- **Subtraction**

D	2	6	F	h
B	A	9	4	h
1	7	D	B	h

One's Complement Representation

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0

Two's Complement Representation

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
															1
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

Integer Representations

- Unsigned Integers

  Represents a magnitude so it is never negative. Largest unsigned integer stored in a byte is 11111111b = FFh =255d. Biggest unsigned integer stored in a word is 1111111111111111b = FFFFh = 65535d. If least significant bit is 0 then number is even otherwise number is odd.

  Example: Addresses of memory locations, ASCII character codes.

- Signed Integers

  It can either be positive or negative. Most significant bit is reserved for sign: 0 for positive and 1 for negative. Negative integers are stored in a computer as two's complement.

Decimal Interpretations

- Unsigned Decimal Interpretation
  - Binary to Decimal Conversion
- Signed Decimal Interpretation
  - If MSB is 0 then Signed Decimal is same as Unsigned Decimal
  - If MSB is 1 take two's complement and convert it to Decimal

Most significant bit of a positive signed integer is 0. So, the leading hex digit of a positive signed integer is 0 - 7h. Integers beginning with 8 - Fh have 1 in their sign bit so they are negative.

For a word, largest positive signed integer is 7FFFh = 32767 and smallest negative signed integer is 8000h = -32768

For a byte, largest positive signed integer is 7Fh = 127 and smallest negative signed integer is 80h = -128

For 0000h - 7FFFh and 00h - 7Fh, signed decimal = unsigned decimal

For 8000h - FFFFh, signed decimal = unsigned decimal - 65536

For 80h - FFh, signed decimal = unsigned decimal - 256

Character Representation

- ASCII Code: Uses seven bits to code each character. ASCII codes from 32-126 are considered to be printable. Others are used for communication control purposes.
- Keyboard: For IBM-PC each key is assigned a unique number called scan code.

Session 3

Organization of the IBM Personal Computers

OBJECTIVES:

- Students will have a brief know about the Intel 8086 family of micro-processors.
- They will acquire knowledge about the organization of 8086 micro-processors and the PC.

Intel 8086 Family

- 8086 and 8088 Microprocessor

  8086 is the first 16-bit microprocessor. It has 16-bit data bus where 8088 has 8-bit data bus. 8086 has faster clock rate than 8088.

- 80286 Microprocessor

  It is 16-bit microprocessor. It is faster than 8086. It can operate in real address mode and protected virtual address mode. In protected mode, it can address 16MB of physical memory. It can treat external storage as physical memory.

- 80386 Microprocessor

  80386 is the first 32-bit microprocessor. 8086 has 32-bit data bus. It has high clock rate. It can execute instructions in fewer clock cycles. It can operate in real and protected mode. In protected mode it can address 4GB of physical memory and 64TB of virtual memory.

- 80486 Microprocessor

  80486 is 32-bit microprocessor. It is the fastest and more powerful processor of the family. It has numeric processor, cache memory and more advanced design. It is three times faster than 80386 running at same clock speed.

Organization of the 8086/8088 Microprocessors

- Registers
  Information inside the microprocessor is stored in registers. There are fourteen 16-bit registers in 8086.

    - Data Registers
      Data registers hold data for an operation. 8086 has four general data registers:

1. Accumulator Register (AX): Used in arithmetic, logic and data transfer instructions. In multiplication and division, one of the numbers involved must be in AX or AL. Input and output operations also required AX and AL.
2. Base Register (BX): Serves as an address register
3. Count Register (CX): CX serves as a loop counter. CL is used as a count in instructions that shift and rotate bits.
4. Data Register (DX): Used in multiplication and division. Also used in I/O operation.

- Address Registers

  Address registers hold the address of an instruction or data. Address registers divided in three types:

  1. Segment Registers

     Segment registers have a very special purpose, pointing at accessible blocks of memory. Segment registers work together with general purpose register to access any memory value.

     - Code Segment (CS): Points at the segment containing the current program.
     - Data Segment (DS): Points at segment where variables are defined. By default, BX, SI and DI registers work with DS segment register.
     - Stack Segment (SS): Points at the segment containing the stack. BP and SP work with SS segment register.
     - Extra Segment (ES): It is up to a coder to define its usage.

  2. Pointer Registers
     - Stack Pointer (SP): Used with SS for accessing the stack segment.
     - Base Pointer (BP): Used primarily to access data on the stack
     - Instruction Pointer (IP): It is used to access instructions. CS contains the segment number of the next instruction and IP contains the offset. IP is updated each time an instruction is executed. An instruction cannot contain IP as an operand.

  3. Index Registers

     - Source Index (SI): Point to memory locations in the data segment addressed by DS. Incrementation of SI give easy access to consecutive memory locations

- Destination Index (DI): Same function as SI. String operations use DI to access memory locations addressed by ES.

- Status Register

  Status register keeps the current status of the processor. In 8086 status register is called FLAGS register. There are two types of Flags:

  1. Status Flags: Reflect the result of an instruction executed by the processor. For example, if an arithmetic operation produces a 0 value as a result then the zero flag (ZF) is set to 1.
  2. Control Flags: Enable or disable certain operations of the processor. For example, if interrupt flag (IF) is set to 0, inputs from the keyboard are ignored by the processor

Memory Segment

8086 is a 16-bit processor using 20-bit address. Addresses are too big to fit in a 16-bit register or memory word. 8086 gets around this problem by partitioning its memory into segments.

- Segment Number

Memory segment is a block of 2^{16} = 64K consecutive memory bytes identified by a segment number starting with 0. Segment number is 16-bit so the highest segment number is FFFFh.

- Offset

Within a segment a memory location is specified by giving an offset. This is the number of bytes from the beginning of segment. With 64-bit segment the offset can be given as a 16-bit number. The first byte in a segment is offset 0 and the last offset in a segment is FFFFh.

Physical Address

A memory location may be specified by providing a segment number and an offset written in the form segment: offset. This form is known as logical address.

A4FB:4872h means offset 4872h within segment A4FBh.

For obtaining 20-bit physical address, 8086 first shifts the segment address four bits to the left and then adds the offset. The physical address for A4FB:4872 is A9822h.

Organization of the PC

- Operating System
- Memory Organization
- I/O Port Addresses
- Start-up Operation

Sample Math

A memory location has physical address 80FD2h. In what segment does it have offset BFD2h.

Answer:

Physical address = Segment* 10h + Offset

Segment * 10h = Physical address – offset

Segment * 10h = 80FD2h – BFD2h

Segment * 10h = 75000h

Segment = 7500h

So, the segment address is 7500h.

Session 4

Introduction to IBM PC Assembly Language

OBJECTIVES:

- Students will come to know the syntax of Assembly language.
- They will acquire knowledge on program data, variables and named constants.
- They will get to know about a few basic instructions of assembly language.
- Students will be able to Translate high-level language to assembly language.
- They will come to know about program structure of assembly language.
- Students will learn the input and output instructions.

Assembly Language Syntax

Statements are of two types: Instruction and Assembler Directive.

Statements can be written in the form: Name Operation_Field Operand(s)_Field Comment

Example: Instruction - START: MOV CX,5; initialize counter

Assembler directive- MAIN PROC

Name Field

Used for instruction labels, procedure names and variable names. Assembler translates name into memory address. Names can be 1 to 31 characters long and may consist of letters, digits or special characters. If period is used, it must be first character. Embedded blanks are not allowed. May not begin with a digit. Not case sensitive.

Operation Field

Contains symbolic (Operation code). Assembler translates op code translated into machine language op code. Examples: ADD, MOV, SUB.
In an assembler directive, the operation field represents pseudo-op code. Pseudo-op code is not translated into machine code; it only tells assembler to do something. Example: PROC pseudo-op is used to create a procedure.

Operand Field

Specifies the data that are to be acted on by the operation. An instruction may have zero, one or more operands. In two-operand instruction, first operand is destination, second operand

is source. For an assembler directive, operand field represents more information about the directive. Examples: NOP, INC AX, ADD WORD1, 2.

Comment Field

Say something about what the statement does. Marked by semicolon in the beginning. Assembler ignores anything after semicolon. It is optional but a good practice.

Program Data

Processor operates only on binary data. Data can be given as a number or a character.

- Numbers
    - Binary
    - Decimal
    - Hexadecimal
- Characters

Named Constants

- Use symbolic name for a constant quantity
- Syntax:
    - name EQU constant

Variables

- Each variable has a data type and is assigned a memory address by the program.
- Possible Values:
    - 8-bit Number Range: Signed (-128 to 127), Unsigned (0 to 255)
    - 16-bit Number Range: Signed (-32,678 to 32767), Unsigned (0 to 65,535)
    - ? - To leave variable uninitialized
- Mainly three types
    - Byte Variables
    - Word Variables
    - Arrays

Data Defining Pseudo-Ops

Pseudo - ops	Description	Bytes	Examples
DB	Define Byte	1	var1 DB 'A' array1 DB 10, 20,30,40
DW	Define Word	2	var2 DW 'AB' array2 DW 1000, 2000
DD	Define Double Word	4	Var3 DD -214743648
DQ	Define Quad Word	8	Var DQ ?
DT	Define Ten Bytes	10	Var DT ?

A Few Basic Instructions

- MOV

  Transfer data between registers, between register and a memory location or move a number directly to a register or a memory location.
  Syntax: MOV destination, source
  Example: MOV AX, WORD1

- XCHG

  Exchange the contents of two registers or register and a memory location.
  Syntax: XCHG destination, source
  Example: XCHG AH, BL

- ADD

  To add contents of two registers, a register and a memory location, a number to a register or a number to a memory location.
  Syntax: ADD destination, source
  Example: ADD WORD1, AX

- SUB

  To subtract the contents of two registers, a register and a memory location, a number from a register or a number from a memory location.
  Syntax: SUB destination, source
  Example: SUB AX, DX

- INC

  To increment one to the contents of a register or memory location.
  Syntax: INC destination
  Example: INC WORD1

- DEC

  To decrement one from the contents of a register or memory location.
  Syntax: DEC destination
  Example: DEC BYTE1

- NEG

  To negate the contents of destination.
  Syntax: NEG destination
  Example: NEG BX

Translation of High-level Language to Assembly Language

- Consider instructions: MOV, ADD, SUB, INC, DEC, NEG
- A and B are two-word variables
- Translate statements into assembly language

Statement	**Translation**
B = A	MOV AX, A MOV B, AX

A = 5 – A	MOV AX, 5
	SUB AX, A
	MOV A, AX
	OR
	NEG A
	ADD A, 5

A = B – 2 x A	MOV AX, B
	SUB AX, A
	SUB AX, A
	MOV A, AX

Program Structure

- Machine programs consists of
    - Code: Contains a program's instructions.
        - Syntax: .CODE name
    - Stack: A block of memory to store stack
        - Syntax: .STACK size
    - Data: Contains all variable definitions
        - Syntax: .DATA

Memory Models

- Determines the size of data and code a program can have.
- Syntax:
    - .MODEL memory_model

Model	Description
SMALL	Code in one segment, data in one segment
MEDIUM	Code in more than one segment, data in one segment
COMPACT	Code in one segment, data in more than one segment
LARGE	Both code and data in more than one segments. No array larger than 64KB
HUGE	Both code and data in more than one segments. Array may be larger than 64KB

The Format of a Code

```
.MODEL SMALL
.STACK 100h
.DATA
;data definition go here
.CODE
MAIN PROC
;instructions go here
MAIN ENDP
;other procedures go here
END MAIN
```

Input and Output Instructions

Function Number	Routine	Function	Code
1	Single Key Input	Input: AH=1 Output: AL=ASCII Code if character is pressed AL=0 if non-character key is pressed	MOV AH,1; input key function INT 21H; ASCII code in AL

2	Single Character Output	Input: AH=2
		DL=ASCII code of the display character or control character
		Output: AL=ASCII code of the display character or control character
			MOV AH,2; display character function
			MOV DL,'?'; character is?
			INT 21H; display character

9	Character String Output	Input: DX=offset address of string. The string must end with a '$' character	MSG DB 'HELLO$'

LEA Instruction

It means Load Effective Address. To get the offset address of the character string in DX we use LEA instruction. Destination is a general register and source is a memory location

- Syntax
    - LEA destination, source
- Example
    - LEA DX, MSG

Sample Question:

Write an assembly code to input an uppercase letter and output the letter in lowercase form.

Sample Input:

```
Enter an uppercase letter: A
```

Sample Output:

```
The lowercase letter is: a
```

Session 5

The Processor Status and the Flags Register

OBJECTIVES:

- Students will be make familiar with the FLAGS register
- They will learn how to determine overflow and the reasons behind overflow occurrence.
- Students will learn how instructions affect the flags.

The FLAGS Register

In 8086, the processor state is implemented as nine individual bits called flags. Each decision made by 8086 is based on the values of these flags. The flags are placed in the FLAGS register. The other bits have no significance.

Two types of flags: Status flags and control flags.

Status flags reflect the result of a computation. They are located in bits 0, 2, 4, 6, 7 and 11.

Control flags enable or disable certain operations of the processor. They are located in bits 8,9 and 10.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Bit	Name	Symbol
0	Carry Flag	CF
2	Parity Flag	PF
4	Auxiliary Carry Flag	AF
6	Zero Flag	ZF
7	Sign Flag	SF
11	Overflow Flag	OF
8	Trap Flag	TF
9	Interrupt Flag	IF
10	Direction Flag	DF

Status Flag

Control Flag

The Status Flags

Carry Flag (CF):

CF = 1 if there is a carry out in the MSB on addition, if there is a borrow into the MSB on subtraction, otherwise CF = 0. It is also affected by shift and rotate instructions.

Parity Flag (PF):

PF = 1 if the low byte of a result has even parity. PF = 0 if the low byte of a result has odd parity.

Auxiliary Carry Flag (AF):

AF = 1 if there is a carry out from 3rd bit on addition or a borrow into 3rd bit on subtraction, otherwise AF = 0. AF is used in BCD operations.

Zero Flag (ZF):

ZF = 1 for a zero result. ZF = 0 for a non-zero result.

Sign Flag (SF):

SF = 1 if the MSB of a result is 1 that means the result is negative. SF = 0 if the MSB of a result is 0 that means the result is positive.

Overflow Flag (OF):

OF = 1 if signed overflow occurred. Otherwise OF = 0.

Overflow

The range of signed numbers that can be represented by a 16-bit word is -32768 to 32767 and 8-bit byte is -128 to 127. The range of unsigned numbers that can be represented by a 16-bit word is 0 to 65535 and 8-bit byte is 0 to 255. If the result of an operation falls out of these range, then overflow occurs and the truncated result that is saved will be incorrect. When we perform an arithmetic operation such as addition there are four possible outcomes:

- No overflow
- Signed overflow only
- Unsigned overflow only
- Both signed and unsigned overflows

Example of Unsigned Overflow Only

Suppose AX contains FFFFh (-1), BX contains 0001h (1). Add the contents of AX and BX.

	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

If it is an unsigned interpretation the correct answer 10000h = 65535 but this is out of range for a word operation. A 1 is carried out of the MSB and the answer stored in AX is 0000h which is wrong so unsigned overflow occurs. But signed overflow does not occur as the stored answer is correct as a signed number.

Example of Signed Overflow Only

Suppose AX contains 7FFFh (32767), BX contains 7FFFh (32767). Add the contents of AX and BX

	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

If it is a signed interpretation the answer is FFFEh = -2 but this incorrect as the result should be 65534. So signed overflow occurs. The unsigned interpretation of FFFEh is 65534, which is the right answer. So unsigned overflow does not occur.

Overflow Indicates by Processor

- The processor sets OF = 1 for signed overflow
- The processor sets CF = 1 for unsigned overflow

Overflow Occur Determination by Processor

- Unsigned overflow

  On addition when there is a carry out in the MSB. This means the result is larger than the biggest unsigned number. On subtraction when there is a borrow in the MSB. This means the correct answer is smaller than 0.

- Signed overflow

  On addition the numbers with the same sign produces result of different sign. On subtraction the result has a different sign than expected. Subtraction of numbers with different signs means addition of number with same sign.

How Instructions Affect the Flags

Instruction	**Affected flags**
MOV/XCHG	None
ADD/SUB	All
INC/DEC	All except CF
NEG	All CF=1 unless result is 0 OF=1 if word operand is 8000h or byte operand is 80h

Sample Math:

ADD AX, BX where AX contains FFFFh and BX contains FFFFh.

Answer

	F	F	F	F	h
+	F	F	F	F	h
1	F	F	F	E	h

- The result stored in AX is FFFEh = 1111 1111 1111 1110

  SF=1 because the MSB is 1.
  PF=0 because there are odd number of 1 in the low byte of the result.
  ZF=0 because the result is non-zero.
  CF=1 because there is a carry out of the MSB on addition.
  OF=0 There is a carry into the MSB and also a carry out.

Session 6

Flow Control Instructions

OBJECTIVES:

- Students will come to know about the unconditional and conditional jumps of assembly language.
- They will be able to programming with high-level language structures.

Label

Labels are needed in situations where one instruction refers to another. Labels end with a colon. Label usually placed on a line by themselves. They refer to the instruction that follows.

The JMP instruction

The JMP instruction causes an unconditional jump. JMP can be used to get around the range restriction of a conditional jump.
Syntax: JMP destination_label

Conditional Jumps

If the conditions for the jump instruction, that is the combination of status flag settings are true, the CPU adjusts the IP to point to the destination label so that the instruction at this label will be executed next. If the jump condition is false, then IP is not altered. If the condition for the jump is true, the next instruction to be executed is the one at destination_label. If the condition is false, the instruction immediately following the jump is done next.

Syntax: Jump_instruction destination_label

Range: The destination label must precede or follow the jump instruction no more than 126 bytes.

There are three categories of conditional jumps-

- Signed conditional jumps
- Unsigned conditional jumps
- Single-flag jumps

Signed Conditional Jumps

Symbol	Description	Condition for Jumps
JG/JNLE	Jump if greater than	ZF=0 and SF=OF
	Jump if not less than or equal to	
JGE/JNL	Jump if greater than or equal to	SF=OF
	Jump if not less than or equal to	
JL/JNGE	Jump if less than	SF<>OF
	Jump if not greater than or equal	
JLE/JNG	Jump if less than or equal	ZF=1 or SF<>OF
	Jump if not greater than	

Unsigned Conditional Jumps

Symbol	Description	Condition for Jumps
JA/JNBE	Jump if above	CF=0 and ZF=0
	Jump if not below or equal	
JAE/JNB	Jump if above or equal	CF=0
	Jump if not below	
JB/JNAE	Jump if below	CF=1
	Jump if not above or equal	
JBE/JNA	Jump if equal	CF=1 or ZF=1
	Jump if not above	

Single-Flag Conditional Jumps

Symbol	Description	Condition for Jumps
JE/JZ	Jump if equal	ZF=1
	Jump if equal to or zero	
JNE/JNZ	Jump if not equal	ZF=0
	Jump if not zero	
JC	Jump if carry	CF=1
JNC	Jump if no carry	CF=0
JO	Jump if overflow	OF=1
JNO	Jump if no overflow	OF=0
JS	Jump if sign negative	SF=1
JNS	Jump if nonnegative sign	SF=0
JP/JPE	Jump if parity even	PF=1
JNP/JPO	Jump if parity odd	PF=0

The CMP Instruction

The jump condition is often provided by the CMP (compare) instruction. This instruction subtracts the source from the destination. The result is not stored. Only the flags are affected. The operands of CMP may not both be memory locations. Destination operand may not be a constant.

Syntax: CMP destination, source

High-level Language Branching Structures

- IF-THEN

Syntax

```
IF condition is true
        THEN
        execute true branch statements
END_IF
```

Example

Replace the number in AX by its absolute value

Pseudocode Algorithm	Assembly Code
IF AX<0 THEN replace AX by –AX END_IF	CMP AX,0 JNL END_IF NEG AX END_IF:

- IF-THEN-ELSE

Syntax

```
IF condition is true
        THEN
        execute true branch statements
ELSE
        execute false branch statements
END_IF
```

Example

Suppose AL and BL contain extended ASCII characters. Display the one that comes first in the character sequence.

Pseudocode Algorithm	Assembly Code

IF AL<=BL
THEN
display the character in AL
ELSE
display the character in BL
END_IF
	MOV AH,2
	CMP AL, BL
	JNBE ELSE_
	MOV DL, AL
	JMP DISPLAY
	ELSE_:
	MOV DL, BL
	DISPLAY:
	INT 21H
	END_IF:

- CASE

Syntax

```
CASE expression
        values_1: statements_1
        values_2: statements_2
         .
         .
         values_n: statements_n
 END_CASE
```

Example
If AX contains a negative number, put -1 in BX; if AX contains 0, put 0 in BX; if AX contains a positive number, put 1 in BX

Pseudocode Algorithm	**Assembly Code**

CASE AX
<0: put -1 in BX
=0: put 0 in BX
>0: put 1 in BX
END_CASE
	CMP AX,0
	JL NEGATIVE
	JE ZERO
	JG POSITIVE
	NEGATIVE:
	MOV BX, -1
	JMP END_CASE
	ZERO:
	MOV BX,0
	JMP END_CASE
	POSITIVE:
	MOV BX,1
	END_CASE:

High-level Language Branching Structures with Compound Conditions

- AND

An AND condition is true if and only if condition_1 and condition_2 are both true.

Syntax

```
condition_1 AND condition_2
```

Example
Read a character and if it is an uppercase letter display it.

Pseudocode Algorithm	Assembly Code

Read a character
IF ('A' >= character) and (character <= 'Z')
THEN
display character
END_IF
	MOV AH,1
	INT 21H
	CMP AL,'A'
	JNGE END_IF
	CMP AL,'Z'
	JNLE END_IF
	MOV DL, AL
	MOV AH,2
	INT 21H
	END_IF:

- OR

An OR condition is true if at least one of condition between condition_1 and condition_2 are true.

Syntax

condition_1 OR condition_2

Example:
Read a character and if it's 'y' or 'Y' display it otherwise terminate the program.

Pseudocode Algorithm	Assembly Code

Read a character
IF (character= 'y') OR (character= 'Y')
THEN
display it
ELSE
terminate the program
END_IF
	MOV AH,1
	INT 21H
	CMP AL, 'y'
	JE THEN
	CMP AL,'Y'
	JE THEN
	JMP ELSE_
	THEN:
	MOV AH,2
	MOV DL, AL
	INT 21H
	JMP END_IF
	ELSE_:
	MOV AH,4CH
	INT 21H
	END_IF:

High-level Language Looping Structures

- FOR
  This is a loop structure in which the loop statements are repeated a known number of times. The counter for the loop is the register CX which is initialized to loop_count. Execution of LOOP instruction causes CX to be decremented automatically.

  Syntax

  LOOP destination_label

  Example:
  Write a count-controlled loop and display a row of 80 stars.

Pseudocode Algorithm	**Assembly Code**

FOR 80 times DO
display '*'
END_FOR
	MOV CX,80
	MOV AH,2
	MOV DL,'*'
	TOP:
	INT 21H
	LOOP TOP

The JCXZ Instruction

It is used before the loop instruction to check the value of CX. If CX is zero, then it helps to terminate the program.

Syntax

JCXZ destination_label

- WHILE
  This is a loop structure which is depend on a condition. The condition is checked at the top of the loop. The loop executes as long as the condition is true.

  Syntax

```
WHILE condition DO
        statements
END_WHILE
```

Example
Write some code to count the number of characters in an input line.

Pseudocode Algorithm	Assembly Code

Initialize count to 0
Read a character
WHILE character <> carriage_return DO
count=count+1
read a character
END_WHILE
	MOV DX,0
	MOV AH,1
	INT 21H
	WHILE_:
	CMP AL,0DH
	JE END_WHILE
	INC DX
	INT 21H
	JMP WHILE_
	END_WHILE:

- REPEAT

  This is a conditional loop structure. The condition is checked after the statements are executed.

  Syntax

  REPEAT
  statements
  UNTIL condition

Example:
Write some code to read characters until a blank is read.

Pseudocode Algorithm	Assembly Code
REPEAT read a character UNTIL character is a blank	MOV AH,1 REPEAT: INT 21H CMP AL,' ' JNE REPEAT END_WHILE:

Sample Question:

1. Write an assembly code to input a word consisting of uppercase and lowercase letters. If there is no uppercase letter the program will output, “No uppercase letters”. And if there is uppercase letter the program will output the first and last uppercase letter.

Sample Input:

```
Enter a word: rhgd
```

Sample Output:

```
No uppercase letter.
```

Sample Input:

```
Enter a word: FIELD
```

Sample Output:

```
The first uppercase letter is: D

The last uppercase letter is: L
```

Session 7

Logic, Shift and Rotate Instructions

OBJECTIVES:

- Students will be able to know the logic, shift and rotate instructions of 8086 assembly language.
- They will be able to write code using logic, shift and rotate instructions.

Logic Instructions

We can change individual bits in the computer by using logic operations. The binary values of 0 and 1 are treated as false and true respectively. When a logic operation is applied to 8 or 16-bit operands, the result is obtained by applying the logic operation at each bit position.

Logic instructions are: AND, OR, XOR and NOT

- AND

  The result of the operation is stored in the destination. Destination must be a register or memory location. Source may be a constant, register or memory location. Memory to memory operations are not allowed

  Effect on flags: SF, ZF and PF reflect the result. CF, OF = 0

  Example: Converting an ASCII digit to a number and conversion of lowercase letter to upper case letter.

  Syntax: AND destination, source

- OR

  The result of the operation is stored in the destination. Destination must be a register or memory location. Source may be a constant, register or memory location. Memory to memory operations are not allowed.

  Effect on flags: SF, ZF and PF reflect the result. CF, OF = 0

  Example: Testing a register for zero and to check the sign of the value.

  Syntax: OR destination, source

- XOR

  The result of the operation is stored in the destination. Destination must be a register or memory location. Source may be a constant, register or memory location. Memory to memory operations are not allowed.

  Effect on flags: SF, ZF and PF reflect the result. CF, OF = 0

  Example: Clearing the value of a register

  Syntax: XOR destination, source

- NOT

  Perform the one's complement operation on the destination. The result of the operation is stored in the destination. Destination must be a register or memory location.

  Effect on flags: There is no effect on the status flags.

  Example: Complement the bits of a register or memory location

  Syntax: NOT destination

Mask

One use of AND, OR and XOR is to selectively modify the bits in the destination. To do is we construct a source bit pattern known as mask. The mask bits are chosen so that the corresponding destination bits are modified in the desired way.

B AND 1 = B	B OR 0 = B	B XOR 0 = B
B AND 0 = 0	B OR 1 = 1	B XOR 1 = -B

The AND instruction can be used to clear specific destination bits while preserving the others.
The OR instruction can be used to set specific destination bits while preserving the others.
The XOR instruction can be used to complement specific destination bits while preserving the others.

The TEST Instruction

The TEST instruction performs an AND operation of the destination with the source but does not change the destination contents. The purpose of the TEST instruction is to set the status flags. The mask should contain 1's in the bit positions to be tested and 0's elsewhere. If destination has 0's in all the tested position, the result will be 0 and so ZF=1.
Effect on flags: SF, ZF, PF reflect the result. CF, OF= 0
Syntax: TEST destination, source

Shift Instructions

The shift instructions shift the bits in the destination operand by one or more positions either to the left or right. For a shift instruction, the bits shifted out are lost. For intel's more advanced processors, a shift instruction also allows the use of an 8-bit constant
Syntax: opcode destination, 1; for a single shift
opcode destination, CL; for a shift of N positions where CL contains N.

In both cases, destination is an 8 or 16-bit register or memory location.

There are two types of shift instruction: Left shift and Right shift.

- Left Shift: The SHL Instruction, The SAL Instruction
- Right Shift: The SHR Instruction, The SAR Instruction

SHL Instruction

The SHL (shift left) instruction shifts the bits in the destination to the left. A 0 is shifted into the rightmost bit position and the MSB is shifted into CF.

Effect on flags: SF, PF, ZF reflect the result. CF= last bit shifted out. OF=1 if result changes sign on last shift.

Example: The SHL instruction on a binary number doubles the value.

Syntax: SHL destination, 1; for a single shift
SHL destination, CL; for a shift of N positions where CL contains N. The value of CL remains the same after the shift operation.

The SAL Instruction

The opcode SAL (shift arithmetic left) is often used in instances where numeric multiplication is intended. SAL instructions generate the same machine code as SHL instruction. Negative numbers can also be multiplied by powers of 2 by left shifts.

Example: If AX is FFFFh (-1), then shifting three times will yield AX= FFF8h (-8).

Overflow

When we treat left shifts as multiplication, overflow may occur. For a single left shift, CF and OF accurately indicate unsigned and signed overflow, respectively. But the overflow flags are not reliable indicators for a multiple left shift. This is because a multiple shift is really a series of single shifts, and CF, OF only reflect the result of the last shift.

Example: If BL contains 80h, CL contains 2 and we execute SHL BL, CL; then CF = OF = 0 even though both signed and unsigned overflow occur.

Example

Write some code to multiply the value of AX by 8. Assume that overflow will not occur.

Solution: To multiply by 8, we need to do three left shifts.

```
MOV   CL, 3; number of shifts to do
SAL   AX, CL; multiply by 8
```

The SHR Instruction

The instruction SHR (shift right) performs right shifts on the destination operand. A 0 is shifted into the MSB position, and the rightmost bit is shifted into CF.
Effect on flags: SF, PF, ZF reflect the result. CF = last bit shifted out. OF = 1 if result changes sign on last shift
Example: The SHR instruction on a binary number halves the value if it is an even number. For odd numbers, a right shift halves it and rounds down to the nearest integer.
Syntax: SHR destination, 1; for a single shift
SHR destination, CL; for a shift of N positions where CL contains N.

The SAR Instruction

The SAR instruction (shift arithmetic right) operates like SHR. The MSB retains its original value.
Effect on flags: SF, PF, ZF reflect the result. CF = last bit shifted out. OF =1 if result changes sign on last shift.
Syntax: SAR destination,1; for a single shift
SAR destination, CL; for a shift of N positions where CL contains N.

Rotate Instructions

The rotate instructions rotate the bits in the destination operand by one or more positions either to the left or right. For a rotate instruction, bits shifted out from one end of the operand are put back into the other end. For intel's more advanced processors, a rotate instruction also allows the use of an 8-bit constant.

Syntax: opcode destination, 1; for a single rotate
opcode destination, CL; for a rotate of N positions where CL contains N.
In both cases, destination is an 8 or 16-bit register or memory location.

Rotate instructions are of two kinds: Left Rotate and right rotate.

- Left Rotate: The ROL Instruction, The RCL Instruction
- Right Rotate: The ROR Instruction, The RCR Instruction

The ROL Instruction

The instruction ROL (rotate left) shifts bits to the left. The MSB is shifted into the rightmost bit. The CF also gets the bit shifted out of the MSB. Destination bits forming a circle, with the least significant bit following the MSB in the circle. In ROL, CF reflects the bit that is rotated out. This can be used to inspect the bits in a byte or word without changing the contents.

Syntax: ROL destination, 1; for a single rotate
ROL destination, CL; for a rotate of N positions where CL contains N

The ROR Instruction
The instruction ROR (rotate right) shifts bits to the right. The rightmost bit is shifted into the MSB and also into the CF. In ROR, CF reflects the bit that is rotated out. This can be used to inspect the bits in a byte or word without changing the contents.

Syntax: ROR destination, 1; for a single rotate
ROR destination, CL; for a rotate of N positions where CL contains N

Example
Use ROL to count the number of 1 bit in BX, without changing BX. Put the answer in AX.

Solution:

```
XOR    AX, AX; AX counts bits
MOV    CX, 16; loop counter
TOP:
ROL    BX,1; CF=bit rotated out
```

```
JNC    NEXT; 0 bit
INC    AX; 1 bit, increment total
NEXT:
LOOP   TOP; loop until done
```

The RCL Instruction

The instruction RCL (Rotate through Carry Left) shifts the bits of the destination to the left. The MSB is shifted into the CF, and the previous value of CF is shifted into the rightmost bit. RCL works like ROL, except that CF is part of the circle of bits being rotated.

Effect on the flags: SF, PF, ZF reflect the result. CF = last bit shifted out. OF = 1 if result changes sign in the last rotation.

Syntax: RCL destination, 1; for a single rotate
RCL destination, CL; for a rotate of N positions where CL contains N

The RCR Instruction

The instruction RCR (Rotate through Carry Right) shifts the bits of the destination to the right. The LSB is shifted into the CF, and the previous value of CF is shifted into the leftmost bit. RCR works like ROR, except that CF is part of the circle of bits being rotated.

Effect on the flags: SF, PF, ZF reflect the result. CF = last bit shifted out.

Syntax: RCR destination, 1; for a single rotate
RCR destination, CL; for a rotate of N positions where CL contains N, OF = 1 if result changes sign in the last rotation.

Session 8

The Stack and Introduction to Procedures

OBJECTIVES:

- Students will come to know about the stack.
- They will learn some terminology of procedures.
- They will be able to use CALL and RET instruction in procedures.

The Stack

A stack is one-dimensional data structure. It is processed in a "last-in, first-out" manner. The most recent addition to the stack is called the top of the stack. A program must set aside a block of memory to hold the stack. We have been doing this by declaring a stack segment; for example,

```
.STACK    100H
```

When the program is assembled and loaded in memory, SS will contain the segment number of the stack segment. For the preceding stack declaration, SP, the stack pointer, is initialized to 100h. This represent the empty stack position. When the stack is non-empty, SP contains the offset address of the top of the stack.

PUSH and PUSHF Instruction

To add a new word to the stack we PUSH it on. The instruction PUSHF, which has no operands, pushes the contents of the FLAGS register onto the stack. Initially, SP contains the offset address of the memory location immediately following the stack segment; the first PUSH decreases SP by 2, making it point to the last word in the stack segment. Because each PUSH decreases SP, the stack grows toward the beginning of memory.

Syntax: PUSH source; where source is a 16-bit register or memory word.

Example: PUSH AX

Execution of PUSH cause the following to happen:

1. SP is decreased by 2.
2. A copy of the source content is moved to the address specified by SS: SP. The source is unchanged.

POP and POPF Instruction

To remove the top item from the stack we POP it. The instruction POPF pops the top of the stack into the FLAGS register

Syntax: POP destination; where destination is a 16-bit register or memory word.

Example: POP BX

Executing POP causes this to happen:

1. The content of SS: SP (the top of the stack) is moved to the destination.
2. SP is increased by 2.

Stack Instructions

There is no effect of PUSH, PUSHF, POP, POPF on the flags. Note that PUSH and POP are word operations, so a byte instruction such as PUSH DL is illegal. A push of immediate data, such as PUSH 2 is also illegal. In addition to the user's program, the operating system uses the stack for its own purposes. For example, to implement the INT 21h functions, DOS saves any registers it uses on the stack and restores them when the interrupt routine is completed. This does not cause a problem for the user because any values DOS pushes onto the stack are popped off by DOS before it returns control to the user's program.

Procedure Declaration

- Syntax

```
name PROC type
        body of the procedure
        RET
 name ENDP
```

Name is the user-defined name of the procedure. The optional operand type is NEAR or FAR (if type is omitted, NEAR is assumed). NEAR means that the statement that calls the procedure is in the same segment as the procedure itself. FAR means that the calling statement is in a different segment. In the following, we assume all procedures are NEAR.

CALL Instruction

To invoke a procedure, the CALL instruction is used. There are two kinds of procedure calls, direct and indirect.

Syntax

Direct Procedure: CALL name, where name is the name of a procedure.

Indirect Procedure: CALL address_expression, where address_expression specifics a register or memory location containing the address of a procedure.

Executing a CALL instruction causes the following to happen:

1. The return address to the calling program is saved on the stack. This is the offset of the next instruction after the CALL statement. The segment: offset of this instruction is in CS: IP at the time the call is executed.
2. IP gets the offset address of the first instruction of the procedure. This transfers control to the procedure.

RET Instruction

The RET (return) instruction causes control to transfer back to the calling procedure. For a NEAR procedure execution of RET causes the stack to be popped into IP. If a pop_value N is specified, it is added to SP, and thus has the effect of removing N additional bytes from the stack. CS: IP now contains the segment: offset of the return address, and control returns to the calling program. Every procedure (except the main procedure) should have a RET someplace. Usually it's the last statement in the procedure

Syntax

RET pop_value; integer argument pop_value is optional.

Sample Question:

Write an assembly code to input two numbers and output the product by applying booth's multiplication.

Sample Input:

```
Enter two numbers: 2 3
```

Sample Output:

```
The product of 2 and 3 is 6
```

Session 9

Multiplication and Division Instructions

OBJECTIVES:

- Students will be able to write code for performing multiplication using MUL and IMUL.
- Students will be able to write code for performing multiplication using DIV and IDIV.
- They will learn how to do sign extension of the dividend.
- They will be able to write decimal input and output procedures.

Signed & Unsigned Multiplication

In binary multiplication, signed and unsigned numbers must be treated differently. For example, we want to multiply the eight-bit numbers 10000000 and 11111111.
Interpreted as unsigned numbers, they represent 128 and 255 respectively. The product is 32640 = 0111111110000000b.
Interpreted as signed numbers, they represent -128 and -1 respectively and the product is 128 = 0000000010000000b.
Because signed and unsigned multiplication lead to different results there are two multiplication instructions: MUL and IMUL. For multiplication of positive numbers MUL and IMUL give the same result.
These instructions multiply bytes or words.
For byte multiplication, one number is contained in the source and the other is assumed to be in AL. The 16-bit product will be in AX. The source may be a byte register or memory byte, but not a constant.
For word multiplication, one number is contained in the source and the other is assumed to be in AX. The most significant 16-bits of the double word product will be in DX, and the least significant 16 bits will be in AX (DX:AX). The source may be a 16-bit register or memory word, but not a constant.

MUL Instructions

MUL (multiply) is used for unsigned multiplication.

Syntax: MUL source

Effect on status flags: SF, ZF, PF and AF undefined. CF/OF is 0 if the upper half of the result is zero otherwise 1.

IMUL Instructions

IMUL (integer multiply) is used for signed multiplication.

Syntax: IMUL source

Effect on status flags: SF, ZF, PF and AF undefined. CF/OF is 0 if the upper half of the result is the sign extension of the lower half otherwise 1.

Example

Suppose AX contains 1 and BX contains FFFFh

Instruction	Decimal Product	Hex Product	DX	AX	CF/OF
MUL BX	65535	0000FFFF	0000	FFFF	0
IMUL BX	-1	FFFFFFFF	FFFF	FFFF	0

Simple Application of MUL and IMUL

Translate the high-level language assignment statement, A= 5 x A - 12 x B into assembly code. Let A and B be word variables and suppose there is no overflow. Use IMUL for multiplication.

```
MOV   AX,5
IMUL  A
MOV    A, AX
MOV    AX,12
IMUL B
SUB   A, AX
```

Signed & Unsigned Division

Signed and unsigned division lead to different results. There are two division instructions: DIV and IDIV. These instructions divide 8 (or 16) bits into 16 (or 32) bits.
The quotient and remainder have the same size as the divisor.
In byte form, the divisor is an 8-bit register or memory byte. The 16-bit dividend is assumed to be in AX. After division, the 8-bit quotient is in AL and the 8-bit remainder is in AH. The divisor may not be a constant.

In word form divisor is a 16-bit register or memory word. The 32-bit dividend is assumed to be in DX:AX, after division, the 16-bit quotient is in AX and the 16-bit remainder is in DX. The divisor may not be a constant.
The effect of DIV and IDIV on the flags is that all status flags are undefined.
It is possible that the quotient will be too big to fit in the specified destination (AL or AX). This can happen if the divisor is much smaller than the dividend. If this happens, the program terminates and the system displays the message "Divide Overflow".

DIV Instruction

DIV (divide) is used for unsigned division.

Syntax: DIV divisor

IDIV Instruction

IDIV (integer divide) is used for signed division. For signed division; the remainder has the same sign as the dividend.

Syntax: IDIV divisor

Example

- Suppose DX contains 0000h, AX contains 0005h, and BX contains 0002h.

Instruction	Decimal Quotient	Decimal Remainder	AX	DX
DIV BX	2	1	0002	0001
IDIV BX	2	1	0002	0001

Sign Extension of the Dividend

Word Division

The dividend is in DX:AX even if the actual dividend will fit in AX. In this case DX should be prepared as follows: For DIV, DX should be cleared. For IDIV, DX should be made the sign extension of AX. The instruction CWD (convert word to double word) will do the extension.

Example: Divide -1250 by 7

Solution:

```
MOV    AX, -1250
CWD
```

```
MOV    BX,7
IDIV   BX
```

Byte Division

The dividend is in AX. If the actual dividend is a byte then AH should be prepared as follows: For DIV, AH should be cleared. For IDIV, AH should the sign extension of AL. The instruction CBW (convert byte to word) will do the extension.

Example: Divide the signed value of the byte variable: XBYTE by -7.

Solution:

```
MOV    AL, XBYTE
CBW
AH  MOV    BL, -7
IDIV BL
```

Sample Question:

Write an assembly code and prepare two procedures INDEC and OUTDEC. You have to take the input by using INDEC and show the output using OUTDEC

Sample Input:

```
Enter a character: A
```

Sample Output:

```
The input character is:  A
```

Session 10

Arrays and Addressing Modes

OBJECTIVES:

- Students will gather knowledge on one-dimensional array and two-dimensional array.
- They will come to know various types of addressing modes.
- They will be able to use the XLAT instruction.

Arrays

It is necessary to treat a collection of values as a group. The advantage of using an array to store the data is that a single name can be given to the whole structure and an element can be accessed by providing an index.

One-dimensional Array

A one-dimensional array is an ordered list of elements all of the same type. By "ordered," we mean that there is a first element, second element, third element and so on. In mathematics, if A is an array, the elements are usually denoted by A [1], A [2], A [3], and so on. The address of the array variable is called the base address of the array.

Byte Array declaration: MSG DB 'abcde'

Word Array declaration: w DW 10, 20, 30, 40, 50, 60

Two-dimensional Array

A two-dimensional array is an array of arrays. We can picture the elements as being arranged in rows and columns. Because memory is one-dimensional, the elements of a two-dimensional array must be stored sequentially.

There are two commonly used ways:

- Row-major order
- Column-major order

The DUP Operator

It is possible to define arrays whose elements share a common initial value by using the DUP (duplicate) operator. This operator causes value to be repeated the number of times specified by repeat_count. DUPs may be nested.

Syntax: repeat_count DUP (value)

Example: GAMMA DW 100 DUP (0)

LINE DB 5, 4, 3 DUP (2, 3 DUP (0), 1) which is equivalent to LINE DB 5,4,2,0,0,0,1,2,0,0,0,1,2,0,0,0,1

Location of Array Elements

The address of an array element may be specified by adding a constant to the base address. Suppose A is an array and S denotes the number of bytes in an element. S = 1 for a byte array, S =2 for a word array. The position of the elements in array A can be determined as A = (N - 1) x S

Addressing Modes

The way an operand is specified is known as its addressing mode. The addressing modes we have used so far are-

1. register mode, which means that an operand is a register
2. immediate mode, when an operand is a constant
3. direct mode, when an operand is a variable.

There are four additional addressing modes for the 8086 which are used to address memory operands indirectly.

4. Register Indirect Mode

In this mode, the offset address of the operand is contained in a register. The register acts as a pointer to the memory location. The operand format is [register]. The register is BX, SI, DI or BP. For BX, SI or DI the operand's segment number is contained in DS. For BP, SS has the segment number.

For example, suppose that SI contains 0100h and the word at 0100h contains 1234h. To execute MOV AX, [SI] the CPU

1. examines SI and obtains the offset address 0100h
2. uses the address DS:0100h to obtain the value 1234h
3. moves 1234h to AX.

5. Based Mode

In this mode, the operand's offset address is obtained by adding a number called a displacement to the contents of a register. The register must be BX or BP. If BX is used, DS contains the segment number of the operand's address. If BP is used, SS has the segment number. Displacement may be the offset address of a variable, a constant

(positive or negative) or the offset address of a variable plus or minus a constant. The syntax of an operand is any of the following equivalent expressions:

[register + displacement], [displacement + register], [register] + displacement, displacement + [register], displacement[register]

6. Indexed Mode

In this mode, the operand's offset address is obtained by adding a number called a displacement to the contents of a register. The register must be SI or Dl. If SI or DI is used, DS contains the segment number of the operand's address. Displacement may be the offset address of a variable, a constant (positive or negative) or the offset address of a variable plus or minus a constant.

The syntax of an operand is any of the following equivalent expressions:

[register + displacement], [displacement + register], [register] + displacement, displacement + [register], displacement[register]

7. Based Indexed

In this mode, the offset address of the operand is the sum of the contents of a base register (BX or BP), the contents of an index register (SI or DI), optionally, a variable's offset address and optionally, a constant (positive or negative). If BX is used, DS contains the segment number of the operand's address. If BP is used, SS has the segment number.

The operand may be written several ways. Four of them are-

variable[base_register] [index_register], [base_register + index_register + variable + constant], variable [base_register + index_register + constant], constant [base_register + index_register + variable]

The PTR Operator

Assembler cannot assemble MOV [BX],1. Because it can't tell whether the destination is the byte pointed to by BX or the word pointed to by BX.

If the destination is supposed to be a byte, we can write MOV BYTE PTR [BX], 1

If the destination is supposed to be a word, we can write MOV WORD PTR [BX], 1

Syntax: type PTR address_expression

The LABEL Pseudo-op

Using LABEL pseudo-op code, we can solve the type conflict

Example

```
MONEY    LABEL WORD
DOLLARS DB   1AH
CENTS   DB   52H
```

This declaration types MONEY as a word variable, and the components DOLLARS and CENTS as byte variables, with MONEY and DOLLARS being assigned the same address by the assembler.

```
MOV   AX, MONEY; AL   = dollars, AH   = cents
```

The XLAT Instruction

- The instruction XLAT (translate) is a no-operand instruction that can be used to convert a byte value into another value that comes from a table.
- The byte to be converted must be in AL, and BX has the offset address of the conversion table.
- The instruction
    1. Adds the contents of AL to the address in BX to produce an address within the table
    2. Replaces the contents of AL by the value found at that address.

Sample Question:

Write an assembly code to sort the following data in ascending order using selection sort algorithm.

Sample Input:

```
2 6 1 9 4
```

Sample Output:

```
The sorted list is: 1 2 4 6 9
```

Session 11

The String Instructions

OBJECTIVES:

- Students will learn about the direction flag.
- They will learn some basic string operations. Such as- move a string, store a string, load a string, scan a string, compare a string.
- Students will have a brief knowledge about the general form of the string. instructions

The Direction Flag

One of the control flags is the direction flag (DF). Its purpose is to determine the direction in which string operations will proceed. These operations are implemented by the two index registers SI and DI. Suppose, for example, that the following string has been declared STRING1 DB 'ABCDE', and this string is stored in memory starting at offset 0200h.
DF =0, SI and DI proceed in the direction of increasing memory addresses from left to right across the string.
If DF = 1, SI and DI proceed in the direction of decreasing memory addresses from right to left.

CLD & STD

To make DF = 0, use the CLD instruction: CLD; clear direction flag
To make DF= 1, use the STD instruction: STD; set direction flag
CLD and STD have no effect on the other flags.

Move Instruction

This instruction copies the contents of the byte addressed by DS:SI, to the byte addressed by ES: DI. The contents of the source byte or word are unchanged. Move instruction permits a memory- memory operation. It also involves the ES register. Move instruction have no effect on the flags. After the byte or word has been moved, both SI and DI are automatically incremented by one or two for byte or word respectively if DF=0 and decremented by one or two for byte or word respectively if DF= 1.

MOVSB Instruction: MOVSB; move string byte
MOVSW Instruction: MOVSW; move string word

The REP Prefix

The REP prefix causes MOVSB to be executed N times. MOVSB moves only a single byte from the source string to the destination string. To move the entire string, first initialize CX to the number N of bytes in the source string and execute

```
REP    MOVSB
```

After each MOVSB, CX is decremented until it becomes 0

Store Instruction

This instruction moves the contents or the AL or AX register to the byte or word addressed by ES: DI respectively. DI is automatically incremented by one or two for byte or word respectively if DF=0 and decremented by one or two for byte or word respectively if DF= 1. Store instruction has no effect on the flags.

STOSB Instruction: STOSB; store string byte
STOSW Instruction: STOSW; store string word

Load Instruction

This instruction moves the byte or word addressed by DS:SI into AL or AX respectively. SI is then automatically incremented by one or two for byte or word respectively if DF=0 and decremented by one or two for byte or word respectively if DF= 1. Load instruction has no effect on the flags. LODSB can be used to examine the characters of a string

LODSB Instruction: LODSB; load string byte
LODSW Instruction: LODSW; load string word

Scan Instruction

This instruction can be used to examine a string for a target byte or word. The target byte or word is contained in AL or AX respectively. Store instruction subtracts the string byte or word pointed to by ES: DI from the contents of AL or AX and uses the result to set the flags. The result is not stored. Afterward, DI is automatically incremented by one or two for byte or word respectively if DF=0 and decremented by one or two for byte or word respectively if DF= 1. All the status flags are affected by store instruction.

SCASB Instruction: SCASB; scan string byte
SCASW Instruction: SCASW; scan string word

REPNZ and REPNE Instruction

If CX is initialized to the number of bytes in the string, these instructions will repeatedly subtract each string byte from AL, update DI and decrement CX until there is a zero result (the target is found) or CX = 0 (the string ends).

REPNZ (repeat while not zero) generates the same machine code as REPNE.

Compare Instruction

This instruction subtracts the byte or word with address ES: DI from the byte or word with address DS:SI and sets the flags. The result is not stored. Afterward, both SI and DI are incremented by one or two for byte or word respectively if DF=0 and decremented by one or two for byte or word respectively if DF= 1. All the status flags are affected by CMPSB

CMPSB Instruction: CMPSB; compare string byte

CMPSB may be used to compare two-character strings to see which comes first alphabetically, or if they are identical, or if one string is a substring of the other.

CMPSW Instruction: CMPSW; compare string word

CMPSW is useful in comparing word arrays of numbers.

REPE and REPZ Instruction

String comparison may be done by attaching the prefix REPE (repeat while equal) or REPZ (repeat while zero) to CMPSB or CMPSW. CX is initialized to the number of bytes in the shorter string, then

```
REPE    CMPSB; compare      string    bytes    while    equal
REPE    CMPSW; compare      string    words    while    equal
```

repeatedly executes CMPSB or CMPSW and decrements CX until There is a mismatch between corresponding string bytes or words or CX =0. The flags are set according to the result of the last comparison.

General Form of the String Instructions

When the assembler encounters one of these general forms, it checks to see the source string is in the segment addressed by DS and the destination string is in the segment addressed by ES. In the case of MOVS and CMPS, if the strings are of the same type; that is, both byte strings or word strings.

Explicit Instruction	Implicit Instruction
MOVS destination_string, source_string	MOVSB
CMPS destination_string, source_string	CMPSB
STOS destination_string	STOS STRING2
LODS source_string	LODS STRING1
SCAS destination_string	SCAS STRING2

An advantage of using the general form of string instructions is that because the operands appear as part of the code, program documentation is improved.

A disadvantage is that only by checking the data definitions is it possible to tell whether a general string instruction is a byte form or a word form.

In fact, the operands specified in a general string instruction may not be the actual operands used when the instruction is executed.

Sample Question:

1. Write an assembly code to input a word consisting of uppercase letter and output the total number of vowels and consonants in the given word.

Sample Input:

```
Enter a word: AUST
```

Sample Output:

```
Number of Vowels: 2
Number of Consonants: 2
```

2. Write an assembly code to input a substring and a main string. Check to see if the substring is the substring of main string.

Sample Input:

```
Enter a Substring: as
Enter Main String: euast
```

Sample Output:

```
The given substring is the
substring of main string.
```

MID TERM EXAMINATION

There will be a 30-minutes written mid-term examination. Different types of questions will be included such as MCQ, mathematics, writing code fragments etc.

FINAL TERM EXAMINATION

There will be a one-hour written examination. Different types of questions will be included such as MCQ, mathematics, write a program etc.