UNIT 1 MICROPROCESSOR ARCHITECTURE

Structure **Page No.**

1.0 Introduction 5
1.1 Objectives 5
1.2 Microcomputer Architecture 5
1.3 Structure of 8086 CPU 7
  1.3.1 The Bus Interface Unit
  1.3.2 Execution Unit (EU)
1.4 Register Set of 8086 11
1.5 Instruction Set of 8086 13
  1.5.1 Data Transfer Instructions
  1.5.2 Arithmetic Instructions
  1.5.3 Bit Manipulation Instructions
  1.5.4 Program Execution Transfer Instructions
  1.5.5 String Instructions
  1.5.6 Processor Control Instructions
1.6 Addressing Modes 29
  1.6.1 Register Addressing Mode
  1.6.2 Immediate Addressing Mode
  1.6.3 Direct Addressing Mode
  1.6.4 Indirect Addressing Mode
1.7 Summary 33
1.8 Solutions/Answers 33

1.0 INTRODUCTION

In the previous blocks of this course, we have discussed concepts relating to CPU organization, register set, instruction set, addressing modes with a few examples. Let us look at one microprocessor architecture in regard of all the above concepts. We have selected one of the simplest processors 8086, for this purpose. Although the processor technology is old, all the concepts are valid for higher end Intel processor. Therefore, in this unit, we will discuss the 8086 microprocessor in some detail.

We have started the discussion of the basic microcomputer architecture. This discussion is followed by the details on the components of CPU of the 8086 microprocessor. Then we have discussed the register organization for this processor. We have also discussed the instruction set and addressing modes for this processor. Thus, this unit presents exhaustive details of the 8086 microprocessor. These details will then be used in Assembly Programming.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the features of the 8086 microprocessor;
- list various components of the 8086 microprocessor; and
- identify the instruction set and the addressing modes of the 8086 microprocessor.

1.2 MICROCOMPUTER ARCHITECTURE

The word micro is used in microscopes, microphones, microwaves, microprocessors, microcomputers, microprogramming, microcodes etc. It means small. A

microprocessor is an example of VLSI bringing the whole processor to a single small chip. With the popularity of distributed processing, the emphasis has shifted from the single mainframe system to independently working workstations or functioning units with their own CPU, RAM, ROM and a magnetic or optical disk memory. Thus, the advent of the microprocessor has transformed the mainframe environment to a distributed platform.

Let us recapitulate the basic components of a microprocessor:

Figure 1: Components of a Microcomputer

Please note the following in the above figure:

- ROM stores the boot program.
- The path from CPU to devices is through Buses. But what would be the size of these Buses?

Bus Sizes

1. The Address bus: 8085 microprocessor has 16 bit lines. Thus, it can access up to 2^{16} = 64K Bytes. The address bus of 8086 microprocessor has a 20 bits address bus. Thus it can access upto 2^{20} = 1M Byte size of RAM directly.

2. Data bus is the number of bits that can be transferred simultaneously. It is 16 bits in 8086.

Microprocessors

The microprocessor is a complete CPU on a single chip. The main advantages of the microprocessor are:

- compact but powerful;
- can be microprogrammed for user's needs;
- easily programmable and maintainable due to small size; and
- useful in distributed applications.

A microprocessor must demonstrate:

- More throughput
- More addressing capability
- Powerful addressing modes
- Powerful instruction set
- Faster operation through pipelining
- Virtual memory management.

However, RISC machine do not agree with above principles.

Some of the most commercially available microprocessors are: Pentium, Xeon, G4 etc.

The assembly language for more advanced chips subsumes the simplest 8086/ 8088 assembly language. Therefore, we will confine our discussions to Intel 8086/8088 assembly language. You must refer to the further readings for more details on assembly language of Pentium, G4 and other processors.

All microprocessors execute a continuous loop of fetch and execute cycles.

```
while (1)
{

        fetch (instruction); ,
        execute (using date);
}
```

1.3 STRUCTURE OF 8086 CPU

The 8086 microprocessor consists of two independent units:

1. The Bus Interface unit, and
2. The Execution unit.

Please refer to Figure 2.

Figure 2: The CPU of INTEL 8086 Microprocessor

The word independent implies that these two units can function parallel to each other. In other words they may be considered as two stages of the instruction pipeline.

1.3.1 The Bus Interface Unit

The BIU (Bus Interface Unit) primarily interacts with the system bus. It performs almost all the activities relating to fetch cycle such as:

- Calculating the physical address of the next instruction
- Fetching the instruction
- Reading or writing data memory or I/O port from memory or Input/ Output.

The instruction/ data is then passed to the execution unit. This BIU consists of:

(a) The Instruction Queue

The instruction queue is used to store the instruction "bytes" fetched. Please note two points here: that it is (1) A Byte (2) Queue. This is used to store information in byte form, with the underlying queue data structure. The advantage of this queue would only be if the next expected instructions are fetched in advance, thus, allowing a pipeline of fetch and execute cycles.

(b) The Segment Registers

These are very important registers of the CPU. Why? We will answer this later. In 8086 microprocessor, the memory is a byte organized, that is a memory address is byte address. However, the number of bits fetched is 16 at a time. The segment registers are used to calculate the address of memory location along with other registers. A segment register is 16 bits long.

The BIU contains four sixteen-bit registers, viz., the CS: Code Segment, the DS: Data Segment, the SS: Stack Segment, and the ES: Extra Segment. But what is the need of the segments: Segments logically divide a program into logical entities of Code, Data and Stack each having a specific size of 64 K. The segment register holds the upper 16 bits of the starting address of a logical group of memory, called the segment. But what are the advantages of using segments? The main advantages of using segments are:

- Logical division of program, thus enhancing the overall possible memory use and minimise wastage.
- The addresses that need to be used in programs are relocatable as they are the offsets. Thus, the segmentation supports relocatability.
- Although the size of address, is 20 bits, yet only the maximum segment size, that is 16 bits, needs to be kept in instruction, thus, reducing instruction length.

The 8086 microprocessor uses overlapping segments configuration. The typical memory organization for the 8086 microprocessor may be as per the following figure.

Figure 3: Logical Organisation of Memory in INTEL 8086 Microprocessor

Although the size of each segment can be 64K, as they are overlapping segments we can create variable size of segments, with maximum as 64K. Each segment has a specific function. 8086 supports the following segments:

As per model of assembly program, it can have more than one of any type of segments. However, at a time only four segments one of each type, can be active.

The 8086 supports 20 address lines, thus supports 20 bit addresses. However, all the registers including segment registers are of only 16 bits. So how may this mapping of 20 bits to 16 bits be performed?

Let us take a simple mapping procedure:

The top four hex digits of initial physical address constitute segment address.

You can add offset of 16 bits (4 Hex digits) from 0000h to FFFFh to it . Thus, a typical segment which starts at a physical address 10000h will range from 10000h to 1FFFFh. The segment register for this segment will contain 1000H and offset will

range from 0000h to FFFFh. But, how will the segment address and offset be added to calculate physical address? Let us explain using the following examples:

Example 1 (In the Figure above)

The value of the stack segment register (SS) = 6000h
The value of the stack pointer (SP) which is Offset = 0010h

Thus, Physical address of the top of the stack is:

SS	6	0	0	0	0	Implied zero
SP	+ 0	0	1	0		
Physical Address	6	0	0	1	0	

This calculation can be expressed as:
Physical address = SS (hex) × 16 + SP (hex)

Example 2

The offset of the data byte = 0020h
The value of the data segment register (DS) = 3000h
Physical address of the data byte

DS	3	0	0	0	0	Implied Zero
Offset	+ 0	0	2	0		
Physical Address	3	0	0	2	0	

This calculation can be expressed as physical address = DS (Hex) × 16 + Data byte offset (hex).

Example 3

The value of the Instruction Pointer, holding address of the instruction = 1234h
The value of the code segment register (CS) = 448Ah
Physical address of the instruction

CS	4	4	8	A	0	ImpliedZero
IP	+ 1	2	3	4		
Physical Address	4	5	A	0	4	

Physical Address = CS (Hex) × 16 + IP

(c) Instruction Pointer

The instruction pointer points to the offset of the current instruction in the code segment. It is used for calculating the address of instruction as shown above.

1.3.2 Execution Unit (EU)

Execution unit performs all the ALU operations. The execution unit of 8086 is of 16 bits. It also contains the control unit, which instructs bus interface unit about which memory location to access, and what to do with the data. Control unit also performs decoding and execution of the instructions. The EU consists of the following:

(a) Control Circuitry, Instruction Decoder and ALU

The 8086 control unit is primarily micro-programmed control. In addition it has an instruction decoder, which translates an instruction into sequence of micro operations. The ALU performs the required operations under the control of CU which issues the necessary timing and control sequences.

(b) Registers

All CPUs have a defined number of operational registers. 8086 has several general purpose and special purpose registers. We will discuss these registers in the following sections.

1.4 REGISTER SET OF 8086

The 8086 registers have five groups of registers. These groupings are done on the basis of the main functions of the registers. These groups are:

General Purpose Register

8086 microprocessors have four general purpose registers namely, AX, BX, CX, DX. All these registers are 16 – bit registers. However, each register can be used as two general-purpose byte registers also. These byte registers are named AH and AL for AX, BH and BL for BX, CH and CL for CX, and DH and DL for DX. The H in register name represents higher byte while L represents lower byte of the 16 bits registers. These registers are primarily used for general computation purposes. However, in certain instruction executions they acquire a special meaning.

AX register is also known as accumulator. Some of the instructions like divide, rotate, shift etc. require one of the operands to be available in the accumulator. Thus, in such instructions, the value of AX should be suitably set prior to the instruction.

BX register is mainly used as a base register. It contains the starting base location of a memory region within a data segment.

CX register is a defined counter. It is used in loop instruction to store loop counter.

DX register is used to contain I/O port address for I/O instruction.

You will experience their usage in various assembly programs discussed later.

Segment Registers

Segment Registers are used for calculating the physical address of the instruction or memory. Segment registers cannot be used as byte registers.

Pointer and Index Registers

The 8086 microprocessor has three pointer and index registers. Each of these registers is of 16 bit and cannot be accessed byte wise. These are Base Pointer (BP), Source Index (S1) and Destination Index (DI). Although they can be used as general purpose registers, their main objective is to contain indexes. BP is used in stack segment, SI in Data segment and DI in Extra Data segment.

Special Registers

A Last in First Out (LIFO) stack is a data structure used for parameter passing, return address storage etc. 8086 stack is 64K bytes. Base of the stack is pointed to by the stack segment (SS) register while the offset or top of the stack is stored in Stack Pointer (SP) register. Please note that although the memory in 8086 has byte addresses, stack is a word stack, which is any push operation will occupy two bytes.

Flags Register

A flag represents a condition code that is 0 or 1. Thus, it can be represented using a flip- flop. 8086 employs a 16-bit flag register containing nine flags. The following table shows the flags of 8086.

Flags	Meaning	Comments
Conditional Flags represent result of last arithmetic or logical instruction executed. Conditional flags are set by some condition generated as a result of the last mathematical or logical instruction executed. The conditional flags are:		
CF	Carry Flag	1 if there is a carry bit
PF	Parity Flag	1 on even parity 0 on odd parity
AF	Auxiliary Flag	Set (1) if auxiliary carry for BCD occurs
ZF	Zero Flag	Set if result is equal to zero
SF	Sign Flag	Indicates the sign of the result (1 for minus, 0 for plus)
OF	Overflow Flag	set whenever there is an overflow of the result
Control flags, which are set or reset deliberately to control the operations of the execution unit. The control flags of 8086 are as follows:		
TF	Single step trap flag	Used for single stepping through the program
IF	Interrupt Enable flag	Used to allow/inhibit the interruption of the program
DF	String direction flag	Used with string instruction.

☞ Check Your Progress 1

1. What is the purpose of the queue in the bus interface unit of 8086 microprocessors?

   ……

   ……

   ……

2. Find out the physical addresses for the following segment register: offset

   (a) SS:SP = 0100h:0020h
   (b) DS:BX = 0200h:0100h
   (c) CS:IP = 4200h:0123h

3. State True or False. | T | F |

(a) BX register is used as an index register in a data segment. ☐

(b) CX register is assumed to work like a counter. ☐

(c) The Source Index (SI) and Destination Index(DI) registers in 8086 can also be used as general registers. ☐

(d) Trag Flag (TR) is a conditional flag. ☐

1.5 INSTRUCTION SET OF 8086

After discussing the basic organization of the 8086 micro-processor, let us now provide an overview of various instructions available in the 8086 microprocessor. The instruction set is presented in the tabular form. An assembly language instruction in the 8086 includes the following:

Label: Op-code Operand(s); Comment

For example, to add the content of AL and BL registers to get the result in AL, we use the following assembly instruction.

NEXT: ADD AL,BL ; AL ← AL + BL

Please note that NEXT is the label field. It is giving an identity to the statement. It is an optional field, and is used when an instruction is to be executed again through a LOOP or GO TO. ADD is symbolic op-code, for addition operation. AL and BL are the two operands of the instructions. Please note that the number of operands is dependent upon the instructions. 8086 instructions can have zero, one or two operands. An operand in 8086 can be:

1. A register
2. A memory location
3. A constant called literal
4. A label.

We will discuss the addressing modes of these operands in section 1.6.

Comments in 8086 assembly start with a semicolon, and end with a new line. A long comment can be extended to more than one line by putting a semicolon at the beginning of each line. Comments are purely optional, however recommended as they provide program documentation. In the next few sections we look at the instruction set of the 8086 microprocessor. These instructions are grouped according to their functionality.

1.5.1 Data Transfer Instructions

These instructions are used to transfer data from a source operand to a destination operand. The source operand in most of the cases remains unchanged. The operand can be a literal, a memory location, a register, or even an I/O port address, as the case may be. Let us discuss these instructions with the following table:

MNEMONIC	DESCRIPTION	EXAMPLE

MOV des, src	des ← src; Both the operands should be byte or word. **src** operand can be register, memory location or an immediate operand **des** can be register or memory operand. Restriction: Both source and destination cannot be memory operands at the same time.	MOV CX,037AH
		; CX register is initialized
		; with immediate value
		; 037AH.
		
		MOV AX,BX
		; AX←BX

PUSH operand	Pushes the operand into a stack. SP ← SP – 2; value [TOS] ← operand. Initialise stack segment register, and the stack pointer properly before using this instruction. No flags are effected by this instruction. The operand can be a general purpose register, a segment register, or a memory location. Please note it is a word stack and memory address is a byte address, thus, you decrement by 2. Also you decrement as SP is initialised to maximum offset and condition of stackful is a zero offset (so it is a reversed stack)	PUSH BX ; decrement stack pointer ; by; two, and copy BX to ; stack. ; decrement stack pointer ; by two, and copy ; BX to stack
POP des	POP a word from stack. The **des** can be a general-purpose register, a segment register (except for CS register), or a memory location. Steps are: des← value [TOS] SP ← SP + 2	POP AX ; Copy content for top ; of stack to AX.
XCHG des, src	Used to exchange bytes or words of src and des. It requires at least one of the operands to be a register operand. The other can be a register or memory operand. Thus, the instruction cannot exchange two memory locations directly. Both the operands should be either byte type or word type. The segment registers cannot be used as operands for this instruction.	XCHG DX,AX ; Exchange word in DX ; with word in AX
XLAT	Translate a byte in AL using a table stored in the memory. The instruction replaces the AL register with a byte from the lookup table. This instruction is a complex instruction.	Example is available in Unit 3.
IN accumulator, port address	It transfers a byte or word from specified port to accumulator register. In case an 8-bit port is supplied as an operand then the data byte read from that part will be transferred to AL register. If a 16-bit port is read then the AX will get 16 bit word that was read. The port address can be an immediate operand, or contained in DX register. This instruction does not change any flags.	IN AL,028h ; read a byte from port ; 028h to AL register
OUT port address, Accumulator	It transfers a byte or word from accumulator register to specified port. This instruction is used to output on devices like the monitor or the printer.	
LEA register, source	Load "effective address" (refer to this term in block 2, Unit 1 in addressing modes) of operand into specified 16 – bit register. Since, an address is an offset in a segment and maximum can	LEA BX, PRICES ; Assume PRICES is ; an array in the data ; segment. The ; instruction loads the

	be of 16 bits, therefore, the register can only be a 16-bit register. LEA instruction does not change any flags. The instruction is very useful for array processing.	; offset of the first byte of ; PRICES directly into ; the BX register.
LDS des-reg	It loads data segment register and other specified register by using consecutive memory locations.	LDS SI, DATA ; DS←content of memory ; location DATA & ; DATA + 1 ; SI ← content of ; memory locations ; DATA + 2 & DATA + ; 3
LES des-reg	It loads ES register and other specified register by using consecutive memory locations. This instruction is used exactly like the LDS except in this case ES & other specified registers are initialized.	
LAHF	Copies the lower byte of flag register to AH. The instruction does not change any flags and has no operands.	
SAHF	Copies the value of AH register to low byte of flag register. This instruction is just the opposite of LAHF instruction. This instruction has no operands.	
PUSHF	Pushes flag register to top of stack. SP ← SP – 2; stack [SP] ← Flag Register.	
POPF	Pops the stack top to Flag register. Flag register ← stack [SP] SP ← SP + 2	

1.5.2 Arithmetic Instructions

MNEMONIC	DESCRIPTION	EXAMPLE

ADD	Adds byte to byte, or word to word. The source may be an immediate operand, a register or a memory location. The rules for operands are the same as that of MOV instruction. To add a byte to a word, first copy the byte to a word location, then fill up the upper byte of the word with zeros. This instruction effects the following flags: AF, CF, OF, PF, SF, ZF.	ADD AL,74H
		; Add the number 74H to
		; AL register, and store the
		; result back in AL
		ADD DX,BX
		; Add the contents of DX to
		; BX and store the result in ;
		DX, BX remains
		; unaffected.

ADC des, src	Add byte + byte + carry flag, or word + word + carry flag. It adds the two operands with the carry flag. Rest all the details are the same as that of ADD instruction.	
INC des	It increments specified byte or word operand by one. The operand can be a register or a memory location. It can effect AF, SF, ZF, PF, and OF flags. It does not affect the carry flag, that is, if you increment a byte operand	INC BX ; Add 1 to the contents of ; BX register INC BL ; Add 1 to the contents of ; BL register

	having 0FFH, then it will result in 0 value in register and no carry flag.	
AAA	ASCII adjusts after addition. The data entered from the terminal is usually in ASCII format. In ASCII 0-9 are represented by codes 30-39. This instruction allows you to add the ASCII codes instead of first converting them to decimal digit using masking of upper nibble. AAA instruction is then used to ensure that the result is the correct unpacked BCD.	ADD AL,BL ; AL=00110101, ASCII 05 ; BL=00111001, ASCII 09 ; after addition ; AL = 01101110, that is, ; 6EH- incorrect ; temporary result AAA ; AL = 00000100. ; Unpacked BCD for 04 ; carry = 1, indicates ; the result is 14
DAA	Decimal (BCD) adjust after addition. This is used to make sure that the result of adding two packed BCD numbers is adjusted to be a correct BCD number. DAA only works on AL register.	; AL = 0101 1001 (59 ; BCD) ; BL = 0011 0101 (35 ; BCD) ADD AL, BL ; AL = 10001101 or ; 8Eh (incorrect BCD) DAA ; AL = 1001 0100 ; ≡ 94 BCD : Correct.
SUB des, src	Subtract byte from byte, or word from word. (des ← des – src). For subtraction the carry flag functions as a borrow flag, that is, if the number in the source is greater than the number in the destination, the borrow flag is to set 1. Other details are equivalent to that of the ADD instruction.	SUB AX, 3427h ; Subtract 3427h from AX ; register, and store the ; result back in AX
SBB des, src	Subtract operands involving previous carry if any. The instruction is similar to SUB, except that it allows us to subtract two multibyte numbers, because any borrow produced by subtracting less-significant byte can be included in the result using this instruction.	SBB AL,CH ; subtract the contents ; of CH and CF from AL ; and store the result ; back in AL.
DEC src	Decrement specified byte or specified word by one. Rules regarding the operands and the flags that are affected are same as INC instruction. Please note that if the contents of the operand is equal to zero then after decrementing the contents it becomes 0FFH or 0FFFFH, as the case may be. The carry flag in this case is not affected.	DEC BP ; Decrement the contents ; of BP ; register by one.
NEG src	Negate - creates 2's complement of a given number, this changes the sign of a number. However, please note that if you apply this instruction on operand having value –128 (byte operand) or –32768 (word operand) it will result in overflow condition. The overflow (OF) flag will be set to	NEG AL ; Replace the number in ; AL with it's 2's ; complement

	indicate that operation could not be done.	
CMP des,src	It compares two specified byte operands or two specified word operands. The source and destination operands can be an immediate number, a register or a memory location. But, both the operands cannot be memory locations at the same time. *The comparison is done simply by internally subtracting the source operand from the destination operand.* The value of source and the destination, operands is not changed, but the flags are set to indicate the results of the comparison.	CMP CX,BX ; Compare the CX register ; with the BX register ; In the example above, the ; CF, ZF, and the SF flags ; will be set as follows. ; CX=BX 0 1 0; result of ; subtraction is zero ; CX>BX 0 0 0; no borrow ; required therefore, CF=0 ; CX<BX 1 0 1 ; subtraction require ; borrow, so CF=1
AAS	ASCII adjust after subtraction. This instruction is similar to AAA (ASCII adjust after addition) instruction. The AAS instruction works on the AL register only. It updates the AF and CF flags, but the OF, PF, SF and the ZF flags remain undefined.	; AL = 0011 0101 ASCII 5 ; BL = 0011 1001 ASCII 9 SUB AL,BL ; (5-9) result: ; AL= 1111 1100 = - 4 in ; 2's complement, CF = 1 AAS ;result: ; AL = 0000 0100 = ; BCD 04, ; CF = 1 borrow needed.
DAS	Decimal adjust after subtraction. This instruction is used after subtracting two packed BCD numbers to make sure the result is the packed BCD. DAS only works on the AL register. The DAS instruction updates the AF, CF, SF, PF and ZF flags. The overflow (OF) is undefined after DAS.	; AL=86 BCD ; BH=57 BCD SUB AL,BH ; AL=2Fh, CF =0 DAS ; Results in AL = 29 BCD

MUL src	This is an unsigned multiplication instruction that multiplies two bytes to produce a word operand or two words to produce a double word such as:
	AX ← AL* src (byte multiplication src is also byte)
	DX or AX ← AX * src (word multiplication is two word).
	This instruction assumes one of the operand in AL (byte) or AX (word): the src operand can be register or memory operand. If the most significant word of the result is zero then, the CF and the OF flags are both made zero. The AF, SF, PF, ZF flags are not defined after the MUL instruction. If you want to multiply a byte with a word, then first convert byte to a word operand.
		MOV AX,05; AX=05
		MOV CX,02; CX=02
		MUL CX
		; results in DX=0
		; AX=0Ah

AAM	ASCII adjust after multiplication. Please note that two ASCII numbers cannot be multiplied directly. To multiply first convert the ASCII	; AL=0000 0101 unpacked ; BCD 05 ; BH=0000 1001 unpacked ; BCD 09

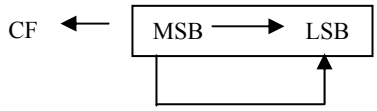
	number to numeric digits by masking off the upper nibble of each byte. This leaves unpacked BCD in the register. AAM instruction is used to adjust the product to two unpacked BCD digits in AX after the multiplication has been performed. AAM defined by the instruction while the CF, OF and the AF flags are left undefined.	MUL BH ; AX=AL * BH=002Dh AAM ; AX=00000100 00000101 ; BCD 45 : Correct result
DIV src	This instruction divides unsigned word by byte, or unsigned double word by word. For dividing a word by a byte, the word is stored in AX register, divisor the src operand and the result is obtained in AH : remainder AL: quotient. It can be represented as: AH: Remainder AL: Quotient } ← AX/ src Similarly for double word division by a word we have DX: Remainder AX: Quotient } ← DX:AX/ src A division by zero result ın run tıme error. The divisor src can be either in a register or a memory operand.	; AX = 37D7h = 14295 ; decimal ; BH = 97h = 151 decimal DIV BH ; AX / BH quotient ; AL = 5Eh = 94 ; decimal RernainderAH = ; 65h = 101 ; decimal
IDIV	Divide signed word by byte or signed double word by word. For this division the operand requirement, the general format of the instruction etc. are all same as the DIV instruction. IDIV instruction leaves all flags undefined.	; AL = 11001010 = -26h = ; - 38 decimal ; CH = 00000011 = + 3h = ; 3 decimal ; According to the operand ; rules to divide by a byte ; the number should be ; present in a word register, ; i.e. AX. So, first convert ; the operand in AL to word ; operand. This can be done ; by sign extending the ; AL register, ; this makes AX ; 11111111 11001010. ; (Sign extension can also ; be done with the help of ; an instruction, discussed ; later) IDIV CH ; AX/CH ; AL = 11110100 = - 0CH ; = -12 Decimal ; AH = 11111110 = -02H = ; - 02 Decimal ; Although the quotient is ; actually closer to -13 ; (-12.66667) than -12, but ; 8086 truncates the result ; to give -12.
AAD	ASCII adjust after division. The BCD numbers are first unpacked, by	; AX= 0607 unpacked ; BCD for 6

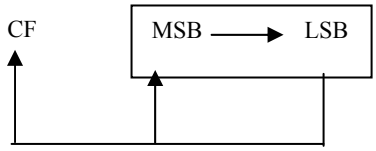
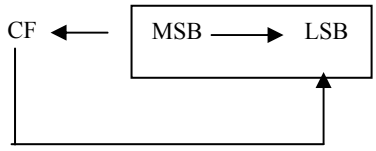
	masking off the upper nibble of each byte. Then ADD instruction is used to convert the unpacked BCD digits in AL and AH registers to adjust them to equivalent binary prior to division. Such division will result in unpacked BCD quotient and remainder. The PF, SF, ZF flags are updated, while the AF, CF, and the OF flags are left undefined.	; and 7 CH = 09h AAD ; adjust to binary before ; division AX= 0043 = ; 043h = 67 Decimal DIV CH ; Divide AX by unpacked ; BCD in CH ; AL = 07 unpacked BCD ; AH = 04 unpacked BCD ; PF = SF = ZF = 0
CBW	Fill upper-byte or word with copies of sign bit of lower bit. This is called sign extension of byte to word. This instruction does not change any flags. This operation is done with AL register in the result being stored in AX.	; AL = 10011011 = -155 ; decimal AH = 00000000 CBW ;convert signed ; byte in AL to signed ; word in AX = 11111111 ; 10011011 = -155 decimal
CWD	Fill upper word or double word with sign bit of lower word. This instruction is an extension of the previous instruction. This instruction results in sign extension of AX register to DX:AX double word.	; DX : 0000 0000 0000 0000 ; AX : 1111 0000 0101 0001 CWD ; DX:AX = 1111 1111 1111 1111: ; 1111 0000 0101 0001

1.5.3 Bit Manipulation Instructions

These instructions are used at the bit level. These instructions can be used for testing a zero bit, set or reset a bit and to shift bits across registers. Let us look into some such basic instructions.

MNEMONIC	DESCRIPTION	EXAMPLE
NOT des	Complements each bit to produce 1's complement of the specified byte or word operand. The operand can be a register or a memory operand.	; BX = 0011 1010 0001 0000 NOT BX ; BX = 1100 0101 1110 1111
AND des, src	Bitwise AND of two byte or word operands. The result is des ← des AND src. The source can be an immediate operand a register, or a memory operand. The destination can be a register or a memory operand. Both operands cannot be memory operands at the same time. The CF and the OF flags are both zero after the AND operation. PF, SF and ZF area updated, Afis left undefined.	; BH = 0011 1010 before AND BH, 0Fh ; BH = 0000 1010 ; after the AND operation
OR des, src	OR each corresponding bits of the byte or word operands. The other operands rules are same as AND. des ← des OR src	; BH = 0011 1010 before OR BH, 0Fh ; BH = 0011 1111 after
XOR des,src	XOR each corresponding bit in a byte or word operands rules are two same as AND and OR. des ← Des + src	; BX = 00111101 01101001 ; CX = 00000000 11111111 XOR BX,CX ; BX=0011110110010110 ; Please note, that the bits in ; the lower byte are inverted.

TEST des, src	AND the operands to update flags, but donot change operands value. It can be used to set and test conditions.CF and OF are both set to zero, PF, SF and ZF are all updated, AF is left undefined after the operation.	; AL = 0101 0001 TEST AL, 80h. ; This instruction would ; test if the MSB bit of the AL ; register is zero or one. After ; the TEST operation ZF will ; be set to 1 if the MSB of AL ; is zero.
SHL/SAL des, count	Shift bits of word or byte left, by count. It puts zero(s) in LSB(s). MSB is shifted into the carry flag. If more than one bits are shifted left, then the CF gets the most recently moved MSB. If the number of bits desired to be shifted is only 1, then the immediate number. 1 can be written as one of the operands. However, if the number of bits desired to be shifted is more than one, then the second operand is put in CL register.	SAL BX, 01 ; if CF = 0 ; BX = 1000 1001 ; result : CF = 1 ; BX = 0001 0010
SHR des, count	It shifts bits of a byte or word to register put zero in MSB. LSB is moved into CF.	SHR BX,01 ; if CF = 0 ; BX = 1000 1001 ; result: CF = 1 ; BX = 0100 0100 MOV CL, 02 SHR BX, CL ; with same BX, the ; result would be ; CF = 0 ; BX = 0010 0100
SAR des, count	Shift bits of word or byte right, but it retains the value of new MSB to that of old MSB. This is also called arithmetic shift operation, as it does not change the MSB, which is sign bit of a number.	; AL=0001 1101 = +29 ; decimal, CF = 0 SAR AL, 01 ; AL = 0000 1110 = +14 ; decimal, CF = 1 ; OF = PF = SF = ZF = 0 ; BH = 1111 0011 = -13 ; decimal SAR BH,01 ; BH = 1111 1001 = -7 ; decimal, CF =1 ; OF = ZF = 0 ; PF = SF = 1
ROL des, count	Rotate bits of word or byte left, MSB is transferred to LSB and also to CF. Diagrammatically, it can be represented as:  The operation is called rotate as it circulates bits. The operands can be register or memory operand.	
ROR des, count	Rotate bits of word or byte right,	; CF = 0,

	LSB is transferred to MSB and also to CF. The same can be represented diagrammatically as follows: 	; BX = 0011 1011 0111 0101 ROR BX, 1 ; results ; CF = 1, ; BX = 1001 1101 1011 1010
RCL des, count	Rotate bits of words or byte left, MSB to CF and CF to LSB. The operation is circular and involves carry flag in rotation. 	
RCR des, count	Rotate bits of word or byte right, LSB to CF and CF to MSB. This instruction rotates left.	`

☞ Check Your Progress 2

1. Point out the error/ errors in the following 8086 assembly instruction (if any)?

   a. PUSHF AX
   b. MOV AX, BX
   c. XCHG MEM_WORD1, MEM_WORD2
   d. AAA BL, CL
   e. IDIV AX, CH

2. State True or False in the context of 8086 assembly language. | T | F |

   (a) LEA and MOV instruction serve the same purpose. The only difference between the two is the type of operands they take. ☐

   (b) NEG instruction produces 1's complement of a number. ☐

   (c) MUL instruction assumes one of the operands to be present in the AL or AX register. ☐

   (d) TEST instruction performs an OR operation, but does not change the value of operands. ☐

   (e) Suppose AL contains 0110 0101 and CF is set, then instructions ROL AL and RCL AL will produce the same results. ☐

1.5.4 Program Execution Transfer Instructions

These instructions are the ones that causes change in the sequence of execution of instruction. This change can be through a condition or sometimes may be unconditional. The conditions are represented by flags. For example, an instruction may be jump to an address if zero flag is set, that is the last ALU operation has resulted in zero value. These instructions are often used after a compare instruction, or some arithmetic instructions that are used to set the flags, for example, ADD or SUB. LOOP is also a conditional branch instruction and is taken till loop variable is below a certain count.

Please note that a "/" is used to separate two mnemonics which represent the same instruction.

MNEMONIC	DESCRIPTION	EXAMPLE
CALL proc 1	This function results in a procedure/ function call. The return address is saved on the stack. There are two basic types of CALLS. NEAR or Intra-Segment calls: if the call is made to a procedure in the same segment as the calling program. FAR or Inter segment call: if the call is made to a procedure in the segment, other than the calling program. The saved return address for NEAR procedure call is just the IP. For FAR Procedure call IP and CS are saved as return address. A procedure can also be called indirectly, by first initializing some 16-bit register, or some other memory location with the new addresses as follows.	CALL procl CALL proc2 The new instruction address is determined by name declaration proc1 is a near procedure, thus, only IP is involved. proc2 involves new CS: IP pair. On call to proc1 stack ← IP IP ← address offset of proc1 on call to proc2 Stack [top] ← CS Stack [top] ← IP CS ← code segment of proc2 IP ← address offset of proc2 Here we assume that procl is defined within the same segment as the calling procedure, while proc2 is defined in another segment. As far as the calling program is concerned, both the procedures have been called in the same manner. But while declaring these procedures, we declare procl as NEAR procedure and proc2 as FAR procedure, as follows: procl PROC NEAR proc2 PROC FAR LEA BX, procl ; initialize BX with the ; offset of the procedure ; procl CALL BX ; CALL procl indirectly ; using BX register
RET number	It returns the control from	RET 6

	procedure to calling program. Every CALL should be a RET instruction. A RET instruction, causes return from NEAR or FAR procedure call. For return from near procedure the values of the instruction pointer is restored from stack. While for far procedure the CS:IP pair get is restored. RET instruction can also be followed by a number.	; In this case, 8086 ; increments the stack ; pointer by this number ; after popping off the IP ; (for new) or IP and CS ; registers (for far) from ; the stack. This cancels ; the local parameters, or ; temporary parameters ; created by the ; programmer. RET ; instruction does not ; affect any flags.
JMP Label	Unconditionally go to specified address and get next instruction from the label specified. The label assigns the instruction to which jump has to take place within the program, or it could be a register that has been initialised with the offset value. JMP can be a NEAR JMP or a FAR jump, just like CALL.	JMP CONTINUE ; CONTINUE is the label ; given to the instruction ; where the control needs ; to be transferred. JMP BX ; initialize BX with the ; offset of the instruction, ; where the control needs ; to be transferred.

Conditional Jump	All the conditional jumps follow some conditional statement, or any instruction that affects the flag.	MOV CX, 05
		MOV BX, 04
		CMP CX, BX
		; this instruction will set
		; various flags like the ZF,
		; and the CF.
		JE LABEL1
		; conditional jump can
		; now be applied, which
		; checks for the ZF, and if
		; it is set implying CX =
		; BX, it makes
		; a jump to LABEL1,
		; otherwise the control
		; simply falls
		; through to next
		; instruction
		; in the above example as
		; CX is not equal to BX
		; the jump will not take
		; place and the next
		; instruction to conditional
		; jump instruction will be
		; executed. However, if
		; JNE (Jump if not equal
		; to) or JA (Jump if
		above), ; or JAE (Jump
		above or
		; equal) jump instructions
		; if applied instead of JE,
		; will cause the conditional
		; jump to occur.

	All the conditional jump instructions which are given below are self explanatory.	
JA/JNBE	Jump if above / Jump if not below nor equal	

JAE/JNB	Jump if above or equal/ Jump if not below	
JB/JNAE	Jump if below/ Jump if not above nor equal	
JBE/JNA	Jump if below or equal/ Jump if not above	
JC	Jump if carry flag set	
JE/JZ	Jump if equal / Jump if zero flag is set	
JNC	Jump if not carry	
JNE/JNZ	Jump if not equal / Jump if zero flag is not set	
JO	Jump if overflow flag is set	
JNO	Jump if overflow flag is not set	
JP/JPE	Jump if parity flag is set / Jump if parity even	
JNP/JPO	Jump if not parity / Jump if parity odd	
JG/JNLE	Jump if greater than / Jump if not less than nor equal	
JA/JNL	Jump if above / Jump if not less than	
JL/JNGE	Jump if less than / Jump if not greater than nor equal	
JLE/JNG	Jump if less than or equal to / Jump if not greater than	
JS	Jump if sign flag is set	
JNS	Jump if sign flag is not set	
LOOP label	This is a looping instruction of assembly. The number of times the looping is required is placed in CX register. Each iteration decrements CX register by one implicitly, and the Zero Flag is checked to check whether to loop again. If the zero flag is not set (CX is zero) greater than the control goes back to the specified label in the instruction, or else the control falls through to the next instruction. The LOOP instruction expects the label destination at offset of –128 to +127 from the loop instruction offset.	; Let us assume we want to ; add 07 to AL register, ; three times. MOV CX,03 ; count of iterations L1: ADD AL,07 LOOP Ll ; loop back to Ll, ; until CX ; becomes equal to zero ; Loop affects no flags.
LOOPE/ LOOPZ label	Loop through a sequence of instructions while zero flag = 1 and CX is not equal to zero. There are two ways to exit out of the loop, firstly, when the count in the CX register becomes equal to zero, or when the quantities that are being compared become unequal.	Let us assume we have an array of 20 bytes. We want to see if all the elements of that array are equal to 0FFh or not. To scan 20 elements of the array, we loop 20 times. And we come out of the loop, when either the count of iterations has become equal to 20, or in other words CX register has

		decremented to zero, which means all the elements of the array are equal to 0FFh, or an element in the array is found which is not equal to 0FFh. In this case, the CX register may still be greater than zero, when the control comes out. This can be coded as follows: (Please note here that you might not understand everything at this place, that is because you are still not familiar with the various addressing modes. Just concentrate on the LOOPE instruction):
		
		MOV BX, OFFSET ARRAY
		; Point BX at the start
		; of the ARRAY
		DEC BX ; put number of
		; array elements in CX
		MOV CX,10
		L1: INC BX ; point to
		; next element in array
		CMP [BX],0FFh
		; compare array element
		; with 0FFh
		LOOPE L1
		; When the control comes
		; out of the loop, it has
		; either scanned all the
		; elements and found them
		; to be all equal to 0FFh, or
		; it is pointing to the first
		; non-0FFh, element in the
		; array.

LOOPNE/LOOPNZ label	This instruction causes Loop through a sequence of instructions while zero flag = 0 and CX is not equal to zero. This instruction is just the opposite of the previous instruction in its functionality.	
JCXZ label	Jump to specified address if CX =0. This instruction will cause a jump, if the value of CX register is zero. Otherwise it will proceed with the next instruction in sequence.	This instruction is useful when you want to check whether CX is zero even prior to entering into a loop. Please note that LOOP instruction executes the loop at least once before decrementing and checking the value of CX register. Thus, CX=0 will execute the loop once and decrement the CX register,

		making it 0FFFFh, which is non zero. This will cause FFFFh times execution of loop. To avoid such type of conditions you can proceed as follows: JCXZ SKIP _LOOP ; if CX is already 0, skip ; loop L1: SUB [BX],07h INC BX LOOP L1 ; loop until CX=0 SKIP_LOOP: ……..

In addition to these instructions, there are other interrupt handling instructions also, which too transfer the control of the program to some specified location. We will discuss these instructions in later units.

1.5.5 String Instructions

These are a very strong set of 8086 instructions as these instructions process strings, in a compact manner, thus, reducing the size of the program by a considerable amount. "String" in assembly is just a sequentially stored bytes or words. A string often consists of ASCII character codes. A subscript B following the instruction indicates that the string of bytes is to be acted upon, while "W" indicates that it is the string of words that is being acted upon.

MNEMONIC	DESCRIPTION	EXAMPLES
REP	This is an instruction prefix. It causes repetition of the following instruction till CX becomes zero. REP. It is not an instruction, but it is an instruction prefix that causes the CX register to be decremented. This prefix causes the string instruction to be repeated, until CX becomes equal to zero.	REP MOVSB STR1, STR2 The above example copies byte by byte contents. The CX register is initialized to contain the length of source string REP repeats the operation MOVSB that copies the source string byte to destination byte. This operation is repeated until the CX register becomes equal to zero.
REPE/REPZ	It repeats the instruction following until CX =0 or ZF is not equal to one. REPE/REPZ may be used with the compare string instruction or the scan string instruction. REPE causes the string instruction to be repeated, till compared bytes or words are equal, and CX is not yet decremented to zero.	
REPNE/REPNZ	It repeats instruction following it until CX =0 or ZF is equal to 1. This comparison here is just inverse of REPE except for CX, which is checked to be equal to zero.	
MOVS/MOVSB/ MOVSW	It causes moving of byte or word from one string to another. This	Assumes both data and extra segment start at address 1000

	instruction assumes that:
	• Source string is in Data segment.
	• Destination string is in extra data segment
	• SI stores offset of source string in extra segment
	• DI stores offset of destination string is in data segment
	• CX contains the count of operation
	A single byte transfer requires;
	• One byte transfer from source string to destination
	• Increment of SI and DI to next byte
	• Decrement count register that is CX register
		in the memory. Source string starts at offset 20h and the destination string starts at offset 30h. Length of the source string is 10 bytes. To copy the source string to the destination string, proceed as follows:
		MOV AX,1000h
		MOV DS,AX
		; initialize data segment and
		MOV ES,AX
		; extra segment
		MOV SI,20h
		MOV DI,30h
		; load offset of start of
		; source string to SI
		; Load offset of start of
		; destination string to DI
		MOV CX,10
		; load length of string to CX
		; as counter
		REP MOVSB
		; Decrement CX and
		; MOVSB until
		; CX =0
		; after move SI will be one
		; greater than offset of last
		; byte in source string, DI
		; will be one greater than
		; offset of last destination
		; string. CX will be equal
		; to zero.

CMPS/CMPSB/ CMPSW	It compares two string bytes or words. The source string and the destination strings should be present in data segment and the extra segment respectively. SI and DI are used as in the previous instruction. CX is used if more than one bytes or words are to be compared, however for such a case appropriate repeating prefix like REP, PEPE etc. need to be used.	MOV CX,10 MOV SI,OFFSET SRC_STR ; offset of source ; string in SI MOV DI, OFFSET DES_STR ; offset of destination ; string in DI REPE CMPSB ; Repeat the comparison of ; string bytes until ; end of string or until ; compared bytes are not ; equal.

SCAS/SCASB/ SCASW	It scans a string. Compare a string byte with byte in AL or a string word with a word in AX. The instruction does not change the operands in AL (AX) or the operand in the string. The string to be scanned must be present in the extra segment, and the offset of the string must be contained in the DI register. You can use CX if operation is to be repeated using REP prefixes.	MOV AL, 0Dh
		; Byte to be scanned
		; for in AL
		MOV DI,OFFSET DES_STR
		MOV CX,10
		REPNE SCAS DES_STR
		; Compare byte inDES_STR
		; with byte in AL register
		; Scanning is repeated while ;
		the bytes are not equal and ;
		it is not end of string. If a
		; carriage return 0Dh is
		; found, ZF = DI will point ;

		at the next byte after the ; carriage return. If a ; carriage return is not ; found then, ZF = 0 and ; CX = 0. SCASB or ; SCASW can be used to ; explicitly state whether ; the byte comparison or the ; word comparison is ; required.
LODS/LODSB/ LODSW	It loads string byte into AL or a string word into AX. The string byte is assumed to be pointed to by SI register. After the load, the SI pointer is automatically adjusted to point to the next byte or word as the case may be. This instruction does not affect any flag.	MOV SI,OFFSET SRC_STR LODS SRC_STR ; LODSB or LODSW can ; be used to indicate to the ; assembler, explicitly, ; whether it is the byte that ; is required to be loaded or ; the word.
STOS/STOSB/ STOSW	It stores byte from AL or word from AX into the string present in the extra segment with offset given by DI. After the copy, DI is automatically adjusted to point to the next byte or word as per the instruction. No flags are affected.	MOVDI,OFFSET DES_STR STOSB DES_STR

1.5.6 Processor Control Instructions

The objectives of these instructions are to control the processor. This raises two questions:

How can you control processor, as this is the job of control unit?
How much control of processor is actually allowed?

Well, 8086 only allows you to control certain control flags that causes the processing in a certain direction, processor synchronization if more than one processors are attached through LOCK instruction for buses etc.

Note: Please note that these instructions may not be very clear to you right now. Thus, some of these instructions have been discussed in more detail in later units. You must refer to further readings for more details on these instructions.

MNEMONIC	DESCRIPTION	EXAMPLE
STC	It sets carry flag to 1.	
CLC	It clears the carry flag to 0.	
CMC	It complements the state of the carry flag from 0 to 1 or 1 to 0 as the case may be.	CMC; Invert the carry flag
STD	It sets the direction flag to 1. The string instruction moves either forward (increment SI, DI) or backward (decrement SI, DI) based on this flag value. STD instruction does not affect any other flag. The set direction flag causes strings to move from right to left.	
CLD	This is opposite to STD, the string	CLD

	operation occurs in the reverse direction.	; Clear the direction flag
		; so that the string pointers
		; auto-increment.
		MOV AX,1000h
		MOV DS, AX
		; Initialize data segment
		; and extra segment
		MOV ES, AX
		MOV SI, 20h
		; Load offset of start of
		; source string to SI
		MOV DI,30h
		; Load offset of start of
		; destination string to DI
		MOV CX,10
		; Load length of string to
		; CX as counter
		REP MOVSB
		; Decrement CX and
		; increment
		; SI and DI to point to next
		; byte, then MOVSB until
		; CX = 0

There are many process control instructions other than these; you may please refer to further reading for such instructions. These instructions include instructions for setting and closing interrupt flag, halting the computer, LOCK (locking the bus), NOP etc.

1.6 ADDRESSING MODES

The basic set of operands in 8086 may reside in register, memory and immediate operand. How can these operands be accessed through various addressing modes? The answer to the question above is given in the following sub-section. Large number of addressing modes help in addressing complex data structures with ease. Some specific Terms and registers roles for addressing:

Base register (BX, BP): These registers are used for pointing to base of an array, stack etc.
Index register (SI, DI): These registers are used as index registers in data and/or extra segments.
Displacement: It represents offset from the segment address.

Addressing modes of 8086

Mode	**Description**	**Example**
Direct	Effective address is the displacement of memory variable.	
Register Indirect	Effective address is the contents of a register.	[BX]
		[SI]
		[DI]
		[BP]
Based	Effective address is the sum of a base register and a displacement.	LIST[BX] (OFFSET LIST + BX)
		[BP + 1]
Indexed	Effective address is the sum of an index register and a displacement.	LIST[SI]
		[LIST +DI]
		[DI + 2]
Based Indexed		[BX + SI]

	Effective address is the sum of a base and an index register.	[BX][DI] [BP + DI]
Based Indexed with displacement	Effective address is the sum of a base register, an index register, and a displacement.	[BX + SI + 2]

1.6.1 Register Addressing Mode

Operand can be a 16-bit register:

Addressing Mode	Description	Example
AX, BX, CX, DX, SI, DI,BP,IP,CS,DS,ES,SS Or it may be AH, AL, BH, BL, CH, CL, DH, DL	In general, the register addressing mode is the most efficient because registers are within the CPU and do not require memory access.	MOV AL,CH MOV AX,CX

1.6.2 Immediate Addressing Mode

An immediate operand can be a constant expression, such as a number, a character, or an arithmetic expression. The only constraint is that the assembler must be able to determine the value of an immediate operand at assembly time. The value is directly inserted into the machine instruction.

MOV AL,05

Mode	Description	Example

Immediate	Please note in the last examples the expression (2 + 3)/5, is evaluated at assembly time.	MOV AL,10
		MOV AL,'A'
		MOV AX,'AB'
		MOV AX, 64000
		MOV AL, (2 + 3)/5

1.6.3 Direct Addressing Mode

A direct operand refers to the contents of memory at an address implied by the name of the variable.

Mode	Description	Example

DIRECT	The direct operands are also called as relocatable operands as they represent the offset of a label from the beginning of a segment. On reloading a program even in a different segment will not cause change in the offset that is why we call them relocatable. Please note that a variable is considered in Data segment (DS) and code label in code segment (SS) by default. Thus, in the example, COUNT, by	MOV COUNT, CL
		; move CL to COUNT (a
		; byte variable)
		MOV AL,COUNT
		; move COUNT to AL
		JMP LABEL1
		; jump to LABEL1
		MOV AX,DS:5
		; segment register and
		; offset
		MOV BX,CSEG:2Ch
		; segment name and offset
		MOV AX,ES:COUNT
		; segment register and
		; variable.

	default will be assumed to be in data segment, while LABEL 1, will be assumed to be in code segment. If we specify, as a direct operand then the address is non-relocatable. Please note the value of segment register will be known only at the run time.	; The offsets of these ; variables are calculated ; with respect to the ; segment name (register) ; specified in the ; instruction.

1.6.4 Indirect Addressing Mode

In indirect addressing modes, operands use registers to point to locations in memory. So it is actually a register indirect addressing mode. This is a useful mode for handling strings/ arrays etc. For this mode two types of registers are used. These are:

- Base register BX, BP
- Index register SI, DI

BX contain offset/ pointer in Data Segment
BP contains offset/ pointer in Stack segment.
SI contains offset/pointer in Data segment.
DI contains offset /pointer in extra data segment.

There are five different types of indirect addressing modes:

1. Register indirect
2. Based indirect
3. Indexed indirect
4. Based indexed
5. Based indexed with displacement.

Mode	Description	Example

Register indirect	Indirect operands are particularly powerful when processing list of arrays, because a base or an index register may be modified at runtime.	MOV BX, OFFSET ARRAY
		; point to start of array
		MOV AL,[BX]
		; get first element
		INC BX
		; point to next
		MOV DL,[BX]
		; get second element
		The brackets around BX signify that we are referring to the contents of memory location, using the address stored in BX.
		In the following example, three bytes in an array are added together:
		MOV SI,OFFSET ARRAY
		; address of first byte
		MOV AL,[SI]
		; move the first byte to AL
		INC SI
		; point to next byte
		ADD AL,[SI]
		; add second byte
		INC SI
		; point to the third byte
		ADD AL,[SI]
		; add the third byte

Based Indirect and Indexed Indirect	Based and indirect addressing modes are used in the same manner. The contents of a register are added to a displacement to generate an effective address. The register must be one of the following: SI, DI, BX or BP. If the registers used for displacement are base registers, BX or BP, it is said to be base addressing or else it is called indexed addressing. A displacement is either a number or a label whose offset is known at assembly time. The notation may take several equivalent forms. If BX, SI or DI is used, the effective address is usually an offset from the DS register; BP on the other hand, usually contains an offset from the SS register.	; Register added to an offset
		MOV DX, ARRAY[BX]
		MOV DX,[DI + ARRAY]
		MOV DX,[ARRAY + SI]
		; Register added to a constant
		MOV AX,[BP + 2]
		MOV DL,[DI – 2] ; DI + (-2)
		MOV DX,2[SI]

Mode	Description	Example
Based Indexed	In this type of addressing the operand's effective address is formed by combining a base register with an index register.	MOV AL,[BP] [SI] MOV DX,[BX + SI] ADD CX,[DI] [BX] ; Two base registers or two ; index registers cannot be ; combined, so the ; following would be ; incorrect: MOV DL,[BP + BX] ; error : two base registers MOV AX,[SI + DI] ; error : two index registers
Based Indexed with Displacement	The operand's effective address is formed by combining a base register, an index register, and a displacement.	MOV DX,ARRAY[BX][SI] MOV AX, [BX + SI + ARRAY] ADD DL,[BX + SI + 3] SUB CX, ARRAY[BP + SI] Two base registers or two index registers cannot be combined, so the following would be incorrect: MOV AX,[BP + BX + 2] MOV DX,ARRAY[SI + DI]

☞ Check Your Progress 3

State True or False. T F

1. CALL instruction should be followed by a RET instruction. ☐

2. Conditional jump instructions require one of the flags to be tested. ☐

3. REP is an instruction prefix that causes execution of an instruction until CX value become 0. ☐

4. In the instruction MOV BX, DX register addressing mode has been used. ☐

5. In the instruction MOV BX,ES:COUNTER the second operand is a direct operand. ☐

6. In the instruction ADD CX, [DI] [BX] the second operand is a based index operand, whose effective address is obtained by adding the contents of DI and BX registers. ☐

7. The instruction ADD AX,ARRAY [BP + SI] is incorrect. ☐

1.7 SUMMARY

In this unit, we have studied one of the most popular series of microprocessors, viz., Intel 8086. It serves as a base to all its successors, 8088, 80186, 80286, 80486, and Pentium. The successors of 8086 can be directly run on any successors. Therefore, though, 8086 has become obsolete from the market point of view, it is still needed to understand advanced microprocessors.

To summarize the features of 8086, we can say 8086 has:

- a 16-bit data bus
- a 20-bit address bus
- CPU is divided into Bus Interface Unit and Execution Unit
- 6-byte instruction prefetch queue
- segmented memory
- 4 general purpose registers (each of 16 bits)
- instruction pointer and a stack pointer
- set of index registers
- powerful instruction set
- powerful addressing modes
- designed for multiprocessor environment
- available in versions of 5Mhz and 8Mhz clock speed.

You can refer to further readings for obtaining more details on INTEL and Motorola series of microprocessors.

1.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1. It improves execution efficiency by storing the next instruction in the register queue.

2. a) 0100 × 10h (- 16 in decimal) + 0020h
   = 01000h + 0020h
   = 01020h

   b) 0200h × 10h + 0100h
   = 02000h + 0100h
   = 02100h

   c) 4200h × 10h + 0123

= 42000h + 0123h
= 42123h

3. a) False b) True c) True d) False

Check Your Progress 2

1. (a) PUSHF instructions do not take any operand.
   (b) No error.
   (c) XCHG instruction cannot have two memory operands
   (d) AAA instruction performs ASCII adjust after addition. It is used after an ASCII Add. It does not have any operands.
   (e) IDIV assumes one operand in AX so only second operand is needed to be specified.

2. (a) False
   (b) False
   (c) True
   (d) False
   (e) False

Check Your Progress 3

1. False
2. True
3. True
4. True
5. True
6. True
7. False

UNIT 2 INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

Structure **Page No.**

2.0	Introduction	35
2.1	Objectives	35
2.2	The Need and Use of the Assembly Language	35
2.3	Assembly Program Execution	36
2.4	An Assembly Program and its Components	41
	2.4.1 The Program Annotation	
	2.4.2 Directives	
2.5	Input Output in Assembly Program	45
	2.5.1 Interrupts	
	2.5.2 DOS Function Calls (Using INT 21H)	
2.6	The Types of Assembly Programs	51
	2.6.1 COM Programs	
	2.6.2 EXE Programs	
2.7	How to Write Good Assembly Programs	53
2.8	Summary	55
2.9	Solutions/Answers	56
2.10	Further Readings	56

2.0 INTRODUCTION

In the previous unit, we have discussed the 8086 microprocessor. We have discussed the register set, instruction set and addressing modes for this microprocessor. In this and two later units we will discuss the assembly language for 8086/8088 microprocessor. Unit 1 is the basic building block, which will help in better understanding of the assembly language. In this unit, we will discuss the importance of assembly language, basic components of an assembly program followed by discussions on the program developmental tools available. We will then discuss what are COM programs and EXE programs. Finally we will present a complete example. For all our discussions, we have used Microsoft Assembler (MASM). However, for different assemblers the assembly language directives may change. Therefore, before running an assembly program you must consult the reference manuals of the assembler you are using.

2.1 OBJECTIVES

After going through this unit you should be able to:

- define the need and importance of an assembly program;
- define the various directives used in assembly program;
- write a very simple assembly program with simple input – output services;
- define COM and EXE programs; and
- differentiate between COM and EXE programs.

2.2 THE NEED AND USE OF THE ASSEMBLY LANGUAGE

Machine language code consists of the 0-1 combinations that the computer decodes directly. However, the machine language has the following problems:

- It greatly depends on machine and is difficult for most people to write in 0-1 forms.
- DEBUGGING is difficult.
- Deciphering the machine code is very difficult. Thus program logic will be difficult to understand.

To overcome these difficulties computer manufacturers have devised English-like words to represent the binary instruction of a machine. This symbolic code for each instruction is called a mnemonic. The mnemonic for a particular instruction consists of letters that suggest the operation to be performed by that instruction. For example, ADD mnemonic is used for adding two numbers. Using these mnemonics machine language instructions can be written in symbolic form with each machine instruction represented by one equivalent symbolic instruction. This is called an assembly language.

Pros and Cons of Assembly Language

The following are some of the advantages / disadvantages of using assembly language:

- Assembly Language provides more control over handling particular hardware and software, as it allows you to study the instructions set, addressing modes, interrupts etc.
- Assembly Programming generates smaller, more compact executable modules: as the programs are closer to machine, you may be able to write highly optimised programs. This results in faster execution of programs.

Assembly language programs are at least 30% denser than the same programs written in high-level language. The reason for this is that as of today the compilers produce a long list of code for every instruction as compared to assembly language, which produces single line of code for a single instruction. This will be true especially in case of string related programs.

On the other hand assembly language is machine dependent. Each microprocessor has its own set of instructions. Thus, assembly programs are not portable.

Assembly language has very few restrictions or rules; nearly everything is left to the discretion of the programmer. This gives lots of freedom to programmers in construction of their system.

Uses of Assembly Language

Assembly language is used primarily for writing short, specific, efficient interfacing modules/ subroutines. The basic idea of using assembly is to support the HLL with some highly efficient but non–portable routines. It will be worth mentioning here that UNIX mostly is written in C but has about 5-10% machine dependent assembly code. Similarly in telecommunication application assembly routine exists for enhancing efficiency.

2.3 ASSEMBLY PROGRAM EXECUTION

An assembly program is written according to a strict set of rules. An editor or word processor is used for keying an assembly program into the computer as a file, and then the assembler is used to translate the program into machine code.

There are 2 ways of converting an assembly language program into machine language:

1) Manual assembly

2) By using an assembler.

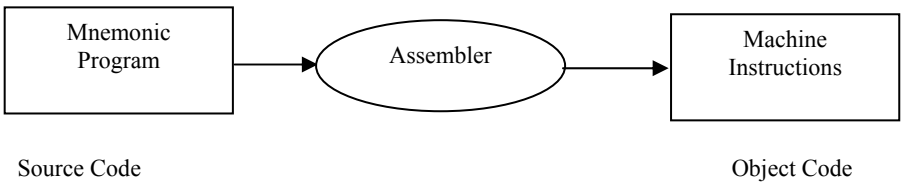
Manual Assembly

It was an old method that required the programmer to translate each opcode into its numerical machine language representation by looking up a table of the microprocessor instructions set, which contains both assembly and machine language instructions. Manual assembly is acceptable for short programs but becomes very inconvenient for large programs. The Intel SDK-85 and most of the earlier university kits were programmed using manual assembly.

Using an Assembler

The symbolic instructions that you code in assembly language is known as - Source program.

An assembler program translates the source program into machine code, which is known as object program.



The steps required to assemble, link and execute a program are:

Step 1: The assembly step involves translating the source code into object code and generating an intermediate .OBJ (object file) or module.

The assembler also creates a header immediately in front of the generated .OBJ module; part of the header contains information about incomplete addresses. The .OBJ module is not quite in executable form.

Step 2: The link step involves converting the .OBJ module to an .EXE machine code module. The linker's tasks include completing any address left open by the assembler and combining separately assembled programs into one executable module.

The linker:

- combines assembled module into one executable program
- generates an .EXE module and initializes with special instructions to facilitate its subsequent loading for execution.

Step 3: The last step is to load the program for execution. Because the loader knows where the program is going to load in memory, it is now able to resolve any remaining address still left incomplete in the header. The loader drops the header and creates a program segment prefix (PSP) immediately before the program is loaded in memory.

Figure 2: Program Assembly

All this conversion and execution of Assembly language performed by Two-pass assembler.

Two-pass assembler: Assemblers typically make two or more passes through a source program in order to resolve forward references in a program. A forward reference is defined as a type of instruction in the code segment that is referencing the label of an instruction, but the assembler has not yet encountered the definition of that instruction.

Pass 1: Assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, name of data fields and programs labels and their relative location (offset) within the segment.

Pass 1 determines the amount of code to be generated for each instruction.

Pass 2: The assembler uses the symbol table that it constructed in Pass 1. Now it knows the length and relative position of each data field and instruction, it can complete the object code for each instruction. It produces .OBJ (Object file), .LST (list file) and cross reference (.CRF) files.

Tools required for assembly language programming

The tools of the assembly process described below may vary in details.

Editor

The editor is a program that allows the user to enter, modify, and store a group of instructions or text under a file name. The editor programs can be classified in 2 groups.

- Line editors
- Full screen editors.

Line editors, such as EDIT in MS DOS, work with the manage one line at a time. Full screen editors, such as Notepad, Wordpad etc. manage the full screen or a paragraph at a time. To write text, the user must call the editor under the control of the operating system. As soon as the editor program is transferred from the disk to the system memory, the program control is transferred from the operating system to the editor program. The editor has its own command and the user can enter and modify text by using those commands. Some editor programs such as WordPerfect are very easy to use. At the completion of writing a program, the exit command of the editor program will save the program on the disk under the file name and will transfer the control to the operating system. If the source file is intended to be a program in the 8086 assembly language the user should follow the syntax of the assembly language and the rules of the assembler.

Assembler

An assembly program is used to transfer assembly language mnemonics to the binary code for each instruction, after the complete program has been written, with the help of an editor it is then assembled with the help of an assembler.

An assembler works in 2 phases, i.e., it reads your source code two times. In the first pass the assembler collects all the symbols defined in the program, along with their offsets in symbol table. On the second pass through the source program, it produces binary code for each instruction of the program, and give all the symbols an offset with respect to the segment from the symbol table.

The assembler generates three files. The object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program. It is created only when your program has been successfully assembled with no errors. The errors that are detected by the assembler are called the symbol errors. For example,

MOVE AX1, ZX1 ;

In the statement, it reads the word MOVE, it tries to match with the mnemonic sets, as there is no mnemonic with this spelling, it assumes it to be an identifier and looks for its entry in the symbol table. It does not even find it there therefore gives an error as undeclared identifier.

List file is optional and contains the source code, the binary equivalent of each instruction, and the offsets of the symbols in the program. This file is for purely

documentation purposes. Some of the assemblers available on PC are MASM, TURBO etc.

Linker

For modularity of your programs, it is better to break your program into several sub routines. It is even better to put the common routine, like reading a hexadecimal number, writing hexadecimal number, etc., which could be used by a lot of your other programs into a separate file. These files are assembled separately. After each file has been successfully assembled, they can be linked together to form a large file, which constitutes your complete program. The file containing the common routines can be linked to your other program also. The program that links your program is called the linker.

The linker produces a link file, which contains the binary code for all compound modules. The linker also produces link maps, which contains the address information about the linked files. The linker however does not assign absolute addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. This form a program is said to be relocatable because it can be put anywhere in memory to be run.

Loader

Loader is a program which assigns absolute addresses to the program. These addresses are generated by adding the address from where the program is loaded into the memory to all the offsets. Loader comes into action when you want to execute your program. This program is brought from the secondary memory like disk. The file name extension for loading is .exe or .com, which after loading can be executed by the CPU.

Debugger

The debugger is a program that allows the user to test and debug the object file. The user can employ this program to perform the following functions.

- Make changes in the object code.
- Examine and modify the contents of memory.
- Set breakpoints, execute a segment of the program and display register contents after the execution.
- Trace the execution of the specified segment of the program and display the register and memory contents after the execution of each instruction.
- Disassemble a section of the program, i.e., convert the object code into the source code or mnemonics.

In summary, to run an assembly program you may require your computer:

- A word processor like notepad
- MASM, TASM or Emulator
- LINK.EXE, it may be included in the assembler
- DEBUG.COM for debugging if the need so be.

Errors

Two possible kinds of errors can occur in assembly programs:

a. Programming errors: They are the familiar errors you can encounter in the course of executing a program written in any language.
b. System errors: These are unique to assembly language that permit low-level operations. A system error is one that corrupts or destroys the system under which the program is running - In assembly language there is no supervising

interpreter or compiler to prevent a program from erasing itself or even from erasing the computer operating system.

2.4 AN ASSEMBLY PROGRAM AND ITS COMPONENTS

Sample Program

In this program we just display:

```
Line     Offset
Numbers          ———————— Source Code ————————————
0001             DATA SEGMENT
0002     0000            MESSAGE DB "HAVE A NICE DAY!$"
0003             DATA ENDS
0004             STACK SEGMENT
0005                     STACK 0400H
0006             STACK ENDS
0007             CODE SEGMENT
0008                     ASSUME   CS: CODE, DS: DATA    SS: STACK
0009     Offset  Machine   Code
0010     0000    B8XXXX          MOV AX, DATA
0011     0003    8ED8            MOV DS, AX
0012     0005    BAXXXX          MOV DX, OFFSET MESSAGE
0013     0008    B409            MOV AH, 09H
0014     000A    CD21            INT  21H
0015     000C    B8004C          MOV AX,  4C00H
0016     000F    CD21            INT   21H
0017                             CODE ENDS
0018                             END
```

The details of this program are:

2.4.1 The Program Annotation

The program annotation consists of 3 columns of data: line numbers, offset and machine code.

- The assembler assigns line numbers to the statements in the source file sequentially. If the assembler issues an error message; the message will contain a reference to one of these line numbers.

- The second column from the left contains offsets. Each offset indicates the address of an instruction or a datum as an offset from the base of its logical segment, e.g., the statement at line 0010 produces machine language at offset 0000H of the CODE SEGMENT and the statement at line number 0002 produces machine language at offset 0000H of the DATA SEGMENT.

- The third column in the annotation displays the machine language produce by code instruction in the program.

Segment numbers: There is a good reason for not leaving the determination of segment numbers up to the assembler. It allows programs written in 8086 assembly language to be almost entirely relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message "Have a nice day$" somewhere in memory. It is located in the DATA SEGMENT. Since the

characters are stored in ASCII, therefore it will occupy 15 bytes (please note each blank is also a character) in the DATA SEGMENT.

Missing offset: The xxxx in the machine language for the instruction at line 0010 is there because the assembler does not know the DATA segment location that will be determined at loading time. The loader must supply that value.

Program Source Code

Each assembly language statement appears as:

{identifier} Keyword {{parameter},} {;comment}.

The element of a statement must appear in the appropriate order, but significance is attached to the column in which an element begins. Each statement must end with a carriage return, a line feed.

Keyword: A keyword is a statement that defines the nature of that statement. If the statement is a directive then the keyword will be the title of that directive; if the statement is a data-allocation statement the keyword will be a data definition type. Some examples of the keywords are: SEGMENT (directive), MOV (statement) etc.

Identifiers: An identifier is a name that you apply to an item in your program that you expect to reference. The two types of identifiers are name and label.

1. Name refers to the address of a data item such as counter, arr etc.
2. Label refers to the address of our instruction, process or segment. For example MAIN is the label for a process as:

```
MAIN PROC FAR
A20:   BL,45 ; defines a label A20.
```

Identifier can use alphabet, digit or special character but it always starts with an alphabet.

Parameters: A parameter extends and refines the meaning that the assembler attributes to the keyword in a statement. The number of parameters is dependent on the Statement.

Comments: A comment is a string of a text that serves only as internal document action for a program. A semicolon identifies all subsequent text in a statement as a comment.

2.4.2 Directives

Assembly languages support a number of statements. This enables you to control the way in which a source program assembles and list. These statements, called directives, act only when the assembly is in progress and generate no machine-executable code. Let us discuss some common directives.

1. **List:** A list directive causes the assembler to produce an annotated listing on the printer, the video screen, a disk drive or some combination of the three. An annotated listing shows the text of the assembly language programs, numbers of each statement in the program and the offset associated with each instruction and each datum. The advantage of list directive is that it produces much more informative output.

2. **HEX:** The HEX directive facilitates the coding of hexadecimal values in the body of the program. That statement directs the assembler to treat tokens in the

source file that begins with a dollar sign as numeric constants in hexadecimal notation.

3. **PROC Directive:** The code segment contains the executable code for a program, which consists of one or more procedures defined initially with the PROC directive and ended with the ENDP directive.

```
Procedure-name    PROC    FAR    ; Beginning of Procedure
Procedure-name    ENDP    FAR    ; End Procedure
```

4. **END DIRECTIVE:** ENDS directive ends a segment, ENDP directive ends a procedure and END directive ends the entire program that appears as the last statement.

5. **ASSUME Directive:** An .EXE program uses the SS register to address the base of stack, DS to address the base of data segment, CS to address base of the code segment and ES register to address the base of Extra segment. This directive tells the assembler to correlate segment register with a segment name. For example,

```
ASSUME  SS: stack_seg_name, DS: data_seg_name,  CS: code_seg_name.
```

6. **SEGMENT Directive:** The segment directive defines the logical segment to which subsequent instructions or data allocations statement belong. It also gives a segment name to the base of that segment.

The address of every element in a 8086 assembly program must be represented in segment - relative format. That means that every address must be expressed in terms of a segment register and an offset from the base of the segmented addressed by that register. By defining the base of a logical segment, a segment directive makes it possible to set a segment register to address that base and also makes it possible to calculate the offset of each element in that segment from a common base.

An 8086 assembly language program consists of logical segments that can be a code segment, a stack segment, a data segment, and an extra segment.

A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

CODE SEGMENT

The logical program segment is named code segment. When the linker links a program it makes a note in the header section of the program's executable file describing the location of the code segment when the DOS invokes the loader to load an executable file into memory, the loader reads that note. As it loads the program into memory, the loader also makes notes to itself of exactly where in memory it actually places each of the program's other logical segments. As the loader hands execution over to the program it has just loaded, it sets the CS register to address the base of the segment identified by the linker as the code segment. This renders every instruction in the code segment addressable in segment relative terms in the form CS: xxxx.

The linker also assumes by default that the first instruction in the code segment is intended to be the first instruction to be executed. That instruction will appear in memory at an offset of 0000H from the base of the code segment, so the linker passes that value on to the loader by leaving an another note in the header of the program's executable file.

The loader sets the IP (Instruction Pointer) register to that value. This sets CS:IP to the segment relative address of the first instruction in the program.

STACK SEGMENT

8086 Microprocessor supports the **Word stack**. The stack segment parameters tell the assembler to alert the linker that this segment statement defines the program stack area.

A program must have a stack area in that the computer is continuously carrying on several background operations that are completely transparent, even to an assembly language programmer, for example, a real time clock. Every 55 milliseconds the real time clock interrupts. Every 55 ms the CPU is interrupted. The CPU records the state of its registers and then goes about updating the system clock. When it finishes servicing the system clock, it has to restore the registers and go back to doing whatever it was doing when the interruption occurred. All such information gets recorded in the stack. If your program has no stack and if the real time clock were to pulse while the CPU is running your program, there would be no way for the CPU to find the way back to your program when it was through updating the clock. 0400H byte is the default size of allocation of stack. Please note if you have not specified the stack segment it is automatically created.

DATA SEGMENT

It contains the data allocation statements for a program. This segment is very useful as it shows the data organization.

Defining Types of Data

The following format is used for defining data definition:

Format for data definition:

{Name} <Directive> <expression>
Name - a program references the data item through the name although it is optional.
Directive: Specifying the data type of assembly.
Expression: Represent a value or evaluated to value.

The list of directives are given below:

Directive	Description	Number of Bytes
DB	Define byte	1
DW	Define word	2
DD	Define double word	4
DQ	Define Quad word	8
DT	Define 10 bytes	10

DUP Directive is used to duplicate the basic data definition to 'n' number of times

```
ARRAY    DB    10 DUP (0)
```

In the above statement ARRAY is the name of the data item, which is of byte type (DB). This array contains 10 duplicate zero values; that is 10 zero values.

EQU directive is used to define a name to a constant

```
CONST    EQU    20
```

Type of number used in data statements can be octal, binary, haxadecimal, decimal and ASCII. The above statement defines a name CONST to a value 20.

Some other examples of using these directives are:

```
TEMP      DB    0111001B      ; Binary value in byte operand
                                  ; named temp
VALI      DW    7341Q         ; Octal value assigned to word
                                  ; variable
Decimal   DB    49            ; Decimal value 49 contained in
                                  ;  byte variable
HEX       DW    03B2AH        ; Hex decimal value in word
                                  ; operand
ASCII     DB    'EXAMPLE'     ; ASCII array of values.
```

☞ Check Your Progress 1

1. Why should we learn assembly language?

   ……

   ……

   ……

2. What is a segment? Write all four main segment names.

   ……

   ……

   ……

3. State True or False. | T | F |

   (a) The directive DT defines a quadword in the memory ☐

   (b) DUP directive is used to indicate if a same memory location is used by two different variables name. ☐

   (c) EQU directive assign a name to a constant value. ☐

   (d) The maximum number of active segments at a time in 8086 can be four. ☐

   (e) ASSUME directive specifies the physical address for the data values of instruction. ☐

   (f) A statement after the END directive is ignored by the assembler. ☐

2.5 INPUT OUTPUT IN ASSEMBLY PROGRAM

A software interrupt is a call to an Interrupt servicing program located in the operating system. Usually the input-output routine in 8086 is constructed using these interrupts.

2.5.1 Interrupts

An **interrupt** causes interruption of an ongoing program. Some of the common interrupts are: keyboard, printer, monitor, an error condition, trap etc.

8086 recognizes two kinds of interrupts: **Hardware** interrupts and **Software** interrupts.

Hardware interrupts are generated when a peripheral Interrupt servicing program requests for some service. A software interrupt causes a call to the operating system. It usually is the **input-output** routine.

Let us discuss the software interrupts in more detail. A software interrupt is initiated using the following statements:

```
INT     number
```

In 8086, this interrupt instruction is processing using the **interrupt vector table (IVT).** The IVT is located in the first 1K bytes of memory, and has a total of 256 entities, each of 4 bytes. An entry in the interrupt vector table is identified by the number given in the interrupt instruction. The entry stores the address of the operating system subroutine that is used to process the interrupt. This address may be different for different machines. Figure 1 shows the processing of an interrupt.

Figure 1: Processing of an Interrupt

The interrupt is processed as:

Step 1: The number field in INT instruction is multiplied by 4 to find its entry in the interrupt vector table. For example, the IVT entry for instruction INT 10h will be found at IVT at an address 40h. Similarly the entry of INT 3h will be placed at 0Ch.

Step 2: The CPU locates the interrupt servicing routine (ISR) whose address is stored at IVT entry of the interrupt. For example, in the figure above the ISR of INT 10h is stored at location at a segment address F000h and an offset F065h.

Step 3: The CPU loads the CS register and the IP register, with this new address in the IVT, and transfers the control to that address, just like a far CALL, (discussed in the unit 4).

Step 4: IRET (interrupt return) causes the program to resume execution at the next instruction in the calling program.

Keyboard Input and Video output

A Keystroke read from the keyboard is called a console input and a character displayed on the video screen is called a console output. In assembly language, reading and displaying character is most tedious to program. However, these tasks were greatly simplified by the convenient architecture of the 8086/8088. That

architecture provides for a pack of software interrupt vectors beginning at address 0000:0000.

The advantage of this type of call is that it appears static to a programmer but flexible to a system design engineer. For example, INT 00H is a special system level vector that points to the "recovery from division by zero" subroutine. If new designer come and want to move interrupt location in memory, it adjusts the entry in the IVT vector of interrupt 00H to a new location. Thus from the system programmer point of view, it is relatively easy to change the vectors under program control.

One of the most commonly used Interrupts for Input /Output is called DOS function call. Let us discuss more about it in the next subsection:

2.5.2 DOS Function Calls (Using INT 21H)

INT 21H supports about 100 different functions. A function is identified by putting the function number in the AH register. For example, if we want to call function number 01, then we place this value in AH register first by using MOV instruction and then call INT 21H:

Some important DOS function calls are:

DOS Function Call	Purpose	Example

AH = 01H	For reading a single character from keyboard and echo it on monitor. The input value is put in AL register.	To get one character input in a variable in data segment you may include the following in the code segment:
		MOV AH,01
		INT 21H
		MOV X, AL
		(Please note that interrupt call will return value in AL which is being transferred to variable of data segment X. X must be byte type).

AH = 02H	This function prints 8 bit data (normally ASCII) that is stored in DL register on the screen.	To print a character let say '?' on the screen we may have to use following set of commands:
		MOV AH, 02H;
		MOV DL, '?'
		INT 21H

AH = 08H	This is an input function for inputting one character. This is same as AH = 01H functions with the only difference that value does not get displayed on the screen.	Same example as 01 can be used only difference in this case would be that the input character wouldn't get displayed
		MOV AH, 08H
		INT 21H
		MOV X, AL

AH = 09H	This program outputs a string whose offset is stored in DX register and that is terminated using a $ character. One can print newline, tab character also.	To print a string "hello world" followed by a carriage return (control character) we may have to use the following assembly program segment.

Example of AH = 09H	CR EQU ODH
	; ASCII code of carriage return.
	DATA SEGMENT
	STRING DB 'HELLO WORLD', CR, '$'
	DATA ENDS
	CODE SEGMENT
	:
	MOV AX, DATA
	MOV DS, AX
	MOV AH, 09H
	MOV DX, OFFSET STRING
	; Store the offset of string in DX register.
	INT 21H	

AH = 0AH	For input of string up to 255 characters. The string is stored in a buffer.	Look in the examples given.
AH = 4CH	Return to DOS	

Some examples of Input

(i) Input a single ASCII character into BL register without echo on screen

```
CODE SEGMENT

   MOV   AH, 08H  ; Function 08H
   INT   21H      ; The character input in AL is
   MOV   BL, AL   ; transfer to BL
   :
CODE ENDS
```

(ii) Input a Single Digit for example (0,1, 2, 3, 4, 5, 6, 7, 8, 9)

```
CODE SEGMENT
            …
;   Read a single digit in BL register with echo. No error check in the Program
            MOV   AH,  01H

            INT   21H
; Assuming that the value entered is digit, then its ASCII will be stored in AL.
; Suppose the key pressed is 1 then ASCII '31' is stored in the AL. To get the
; digit 1 in AL subtract the ASCII value '0' from the AL register.
; Here it store      0 as ASCII 30,
; 1 as 31,  2 as 32…….9 as 39
; to store 1 in memory subtract 30 to get 31 – 30  = 1
            MOV BL, AL
            SUB   BL, ' 0'  ;  ' 0'  is digit 0 ASCII
                  ; OR
            SUB    BL,  30H
; Now BL contain the single digit 0 to 9
; The only code missing here is to check whether the input is in the specific
; range.
…
CODE ENDS.
```

(iii) Input numbers like (10, 11……..99)

```
; If we want to store 39, it is actually 30 + 9
; and it is 3 × 10 + 9
; to input this value through keyboard, first we input the tenth digit e.g., 3 and
```

```
; then type 9
        MOV   AH, 08H
        INT   21H
        MOV  BL, AL ; If we have input 39 then, BL will first have character
; 3, we can convert it to 3 using previous logic that is 33 – 30 = 3.
        SUB   BL, '0'
        MUL   BL, AH          ; To get 30 Multiply it by 10.
                              ; Now BL Store 30
                              ; Input another digit from keyboard
        MOV   AH,  08H
        INT   21H;
        MOV   DL, AL          ; Store AL in DL
        SUB   DL, '0'         ;    (39 – 30) = 9.
; Now BL contains the value: 30 and DL has the value 9 add them and get the
; required numbers.
        ADD   BL, DL
; Now BL store 39.  We have 2 digit value in BL.
```

Let us try to summarize these segments as:

```
CODE SEGMENT
;  Set DS register
  MOV AX, DATA    ; ┐ boiler plate code to set the DS register so that the
  MOV  DS, AX     ; ┘ program can access the data segment.

; read first digit from keyboard
        MOV    AH,  08
        INT    21H
        MOV    BL, AL
        SUB    BL, '0'
        MUL    BL,  10H
; read second digit from keyboard
        MOV    AH, 08H
        INT    21H
        MOV    DL, AL
        SUB    DL, '0'
; DL = 9 AND BL = 30
        SUM     BL, DL
; now BL store 39
CODE ENDS.
```

Note: Boilerplate code is the code that is present more or less in the same form in every assembly language program.

Strings Input

```
CODE SEGMENT
        …
        MOV AH, 0AH   ; Move 04 to AH register
        MOV DX, BUFF          ; BUFF must be defined in data segment.
        INT   21H
…..
CODE ENDS
DATA  SEGMENT
        BUFF  DB      50      ; max length of string,
                              ; including CR, 50 characters
              DB      ?       ; actual length of string not known at present
              DB 50 DUP(0)    ; buffer having 0 values
DATA ENDS.
```

Explanation

The above DATA segment creates an input buffer BUFF of maximum 50 characters. On input of data 'JAIN' followed by enter data would be stored as:

50	4	J	A	I	N	#

Examples of Display on Video Monitor

(1) **Displaying a single character**

```
; display contents of BL register (assume that it has a single character)
MOV  AH,  02H
MOV  DL, BL.
INT    21 H.
```

Here data from BL is moved to DL and then data display on monitor function is called which displays the contents of DL register.

(2) **Displaying a single digit (0 to 9)**

Assume that a value 5 is stored in BL register, then to output BL as ASCII value add character '0' to it

```
ADD  BL, '0'
MOV  AH,  02H
MOV  DL, BL
INT    21H
```

(3) **Displaying a number (10 to 99)**

Assuming that the two digit number 59 is stored as number 5 in BH and number 9 in BL, to convert them to equivalent ASCII we will add '0' to each of them.

```
ADD  BH, '0'
ADD  BL, '0'
MOV  AH, 02H
MOV  DL, BH
INT    21H
MOV  DL, BL
INT    21H
```

(4) **Displaying a string**

```
MOV  AH, 09H
MOV  DX, OFFSET   BUFF
INT    21H
```

Here data in input buffer stored in data segment is going to be displayed on the monitor.

A complete program:

Input a letter from keyboard and respond. "The letter you typed is ___".

```
CODE SEGMENT
;       set the DS register
                MOV  AX, DATA
                MOV DS,  AX
;       Read Keyboard
                MOV  AH,  08H
                INT   21H
;       Save input
                MOV  BL, AL
;       Display first part of Message
                MOV AH, 09H
                MOV DX, OFFSET MESSAGE
                INT  21 H
;       Display character of BL register
                MOV  AH,  02H
                MOV DL, BL
                INT  21 H
;       Exit to DOS
                MOV AX, 4C00H
                INT 21H
CODE ENDS

DATA SEGMENT
        MESSAGE DB "The letter you typed is $"
DATA ENDS
END.
```

2.6 THE TYPES OF ASSEMBLY PROGRAMS

Assembly language programs can be written in two ways:

COM Program: Having all the segments as part of one segment
EXE Program: which have more than one segment.

Let us look into brief details of these programs.

2.6.1 COM Programs

A COM (Command) program is the binary image of a machine language program. It is loaded in the memory at the lowest available segment address. The program code begins at an offset 100h, the first 1K locations being occupied by the IVT.

A COM program keeps its code, data, and stack segments within the same segment. Since the offsets in a physical segment can be of 16 bits, therefore the size of COM program is limited to 2^{16} = 64K which includes code, data and stack. The following program shows a COM program:

```
; Title add two numbers and store the result and carry in memory variables.
; name of the segment in this program is chosen to be CSEG

CSEG SEGMENT
        ASSUME CS:CSEG, DS:CSEG, SS:CSEG
        ORG       100h
START:MOV  AX, CSEG          ; Initialise data segment
        MOV DS, AX           ; register using AX
        MOV AL, NUM1         ; Take the first number in AL
```

```
        ADD AL, NUM2          ; Add the 2nd number to it
        MOV RESULT, AL        ; Store the result in location RESULT
        RCL AL, 01            ; Rotate carry into LSB
        AND AL, 00000001B     ; Mask out all but LSB
        MOV CARRY, AL         ; Store the carry result
        MOV AX,4C00h
        INT 21h
        NUM1 DB      15h      ; First number stored here
        NUM2 DB      20h      ; Second number stored here
        RESULT DB    ?        ; Put sum here
        CARRY DB     ?        ; Put any carry here
CSEG ENDS
END START
```

These programs are stored on a disk with an extension .com. A COM program requires less space on disk rather than equivalent EXE program. At run-time the COM program places the stack automatically at the end of the segment, so they use at least one complete segment.

2.6.2 EXE Programs

An EXE program is stored on disk with extension .exe. EXE programs are longer than the COM programs, as each EXE program is associated with an EXE header of 256 bytes followed by a load module containing the program itself. The EXE header contains information for the operating system to calculate the addresses of segments and other components. We will not go into such details in this unit.

The load module of EXE program consists of up to 64K segments, although at the most only four segments may be active at any time. The segments may be of variable size, with maximum size being 64K.

We will write only EXE programs for the following reasons:

- EXE programs are better suited for debugging.
- EXE-format assembler programs are more easily converted into subroutines for high-level languages.
- EXE programs are more easily relocatable. Because, there is no ORG statement, forcing the program to be loaded from a specific address.
- To fully use multitasking operating system, programs must be able to share computer memory and resources. An EXE program is easily able to do this.

An example of equivalent EXE program for the COM program is:

```
; ABSTRACT        this program adds 2 8-bit numbers in the memory locations
;                 NUM1 and NUM2. The result is stored in the
;                 memory location RESULT. If there was a carry
;                 from the addition it will be stored as 0000 0001 in
;                 the location CARRY
; REGISTERS       Uses CS, DS, AX
DATA SEGMENT
        NUM1   DB    15h      ; First number
        NUM2   DB    20h      ; Second number
```

```
        RESULT DB     ?        ; Put sum here
        CARRY  DB     ?        ; Put any carry here
DATA ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA             ; Initialise data segment
        MOV DS, AX             ; register using AX
        MOV AL, NUM1           ; Bring the first number in AL
        ADD AL, NUM2           ; Add the 2nd number to AL
        MOV RESULT, AL         ; Store the result
        RCL AL, 01             ; Rotate carry into Least Significant Bit (LSB)
        AND AL, 00000001B      ; Mask out all but LSB
        MOV CARRY, AL          ; Store the carry
        MOV AX, 4C00h          ; Terminate to DOS
        INT 21h
CODE ENDS
        END START
```

2.7 HOW TO WRITE GOOD ASSEMBLY PROGRAMS

Now that we have seen all the details of assembly language programming, let us discuss the art of writing assembly programs in brief.

Preparation of writing the program

1. Write an algorithm for your program closer to assembly language. For example, the algorithm for preceding program would be:

   get NUM1
   add NUM2
   put sum into memory at RESULT
   position carry bit in LSB of byte
   mask off upper seven bits
   store the result in the CARRY location.

2. Specify the input and output required.

   input required - two 8-bit numbers
   output required - an 8-bit result and a 8-bit carry in memory.

3. Study the instruction set carefully. This step helps in specifying the available instructions and their format and constraints. For example, the segment registers cannot be directly initialized by a memory variable. Instead we have to first move the offset for segment into a register, and then move the contents of register to the segment register.

You can exit to DOS, by using interrupt routine 21h, with function 4Ch, placed in AH register.

It is a nice practice to first code your program on paper, and use comments liberally. This makes programming easier, and also helps you understand your program later. Please note that the number of comments do not affect the size of the program.

After the program development, you may assemble it using an assembler and correct it for errors, finally creating exe file for execution.

☞ Check Your Progress 2

State True or False

		T	F
1.	For input/ output on Intel 8086/8088 machine running on DOS require special routines to be written by the assembly programmers.		
2.	Intel 8086 processor recognises only the software interrupts.		
3.	INT instruction in effect calls a subroutine, which is identified by a number.		
4.	Interrupt vector table IVT stores the interrupt handling programs.		
5.	INT 21H is a DOS function call.		
6.	INT 21H will output a character on the monitor if AH register contains 02.		
7.	String input and output can be achieved using INT 21H with function number 09h and 0Ah respectively.		
8.	To perform final exit to DOS we must use function 4CH with the INT 21H.		
9.	Notepad is an editor package.		
10.	Linking is required to link several segments of a single assembly program.		
11.	Debugger helps in removing the syntax errors of a program.		
12.	COM program is loaded at the 0th location in the memory.		
13.	The size of COM program should not exceed 64K.		
14.	A COM program is longer than an EXE program.		
15.	STACK of a COM program is kept at the end of the occupied segment by the program.		
16.	EXE program contains a header module, which is used by DOS for calculating segment addresses.		
17.	EXE program cannot be easily debugged in comparison to COM programs.		
18.	EXE programs are more easily relocatable than COM programs.		

2.8 SUMMARY

We summarize the complete discussion in the following flow chart.

2.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. (a) It helps in better understanding of computer architecture and work in machine language.
   (b) Results in smaller machine level code, thus result in efficient execution of programs.
   (c) Flexibility of use as very few restrictions exist.

2. A segment identifier a group of instructions or data value. We have four segments.
   1. Data segment 2. Code segment 3. Stack segment 4. Extra Segment

3. (a) False
   (b) False
   (c) True
   (d) True
   (e) False
   (f) True

Check Your Progress 2

1. False
2. False
3. True
4. False
5. True
6. True
7. True
8. True
9. True
10. False
11. False
12. False
13. True
14. False
15. True
16. True
17. False
18. True

2.10 FURTHER READINGS

1. Yu-Cheng Lin, Genn. A. Gibson, "*Microcomputer System the 8086/8088 Family"* 2nd Edition, PHI.
2. Peter Abel, "*IBM PC Assembly Language and Programming*", 5th Edition, PHI.
3. Douglas, V. Hall, "*Microprocessors and Interfacing*", 2nd edition, Tata McGraw-Hill Edition.
4. Richard Tropper, *"Assembly Programming 8086",* Tata McGraw-Hill Edition.
5. M. Rafiquzzaman, "*Microprocessors, Theory and Applications: Intel and Motorala",* PHI.

UNIT 3 ASSEMBLY LANGUAGE PROGRAMMING (PART – I)

Structure **Page No.**

3.0 Introduction 57
3.1 Objectives 57
3.2 Simple Assembly Programs 57
  3.2.1 Data Transfer
  3.2.2 Simple Arithmetic Application
  3.2.3 Application Using Shift Operations
  3.2.4 Larger of the Two Numbers
3.3 Programming With Loops and Comparisons 63
  3.3.1 Simple Program Loops
  3.3.2 Find the Largest and the Smallest Array Values
  3.3.3 Character Coded Data
  3.3.4 Code Conversion
3.4 Programming for Arithmetic and String Operations 69
  3.4.1 String Processing
  3.4.2 Some More Arithmetic Problems
3.5 Summary 75
3.6 Solutions/ Answers 75

3.0 INTRODUCTION

After discussing a few essential directives, program developmental tools and simple programs, let us discuss more about assembly language programs. In this unit, we will start our discussions with simple assembly programs, which fulfil simple tasks such as data transfer, arithmetic operations, and shift operations. A key example here will be about finding the larger of two numbers. Thereafter, we will discuss more complex programs showing how loops and various comparisons are used to implement tasks like code conversion, coding characters, finding largest in array etc. Finally, we will discuss more complex arithmetic and string operations. You must refer to further readings for more discussions on these programming concepts.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- write assembly programs with simple arithmetic logical and shift operations;
- implement loops;
- use comparisons for implementing various comparison functions;
- write simple assembly programs for code conversion; and
- write simple assembly programs for implementing arrays.

3.2 SIMPLE ASSEMBLY PROGRAMS

As part of this unit, we will discuss writing assembly language programs. We shall start with very simple programs, and later graduate to more complex ones.

3.2.1 Data Transfer

Two most basic data transfer instructions in the 8086 microprocessor are MOV and XCHG. Let us give examples of the use of these instructions.

; Program 1: This program shows the difference of MOV and XCHG instructions:

```
DATA  SEGMENT
        VAL    DB      5678H           ; initialize variable VAL
DATA  ENDS

CODE  SEGMENT
                ASSUME      CS:     CODE, DS: DATA
MAINP:          MOV         AX, 1234H       ; AH=12 & AL=34
                XCHG        AH, AL          ; AH=34 & AL=12
                MOV         AX, 1234H       ; AH=12 & AL=34
                MOV         BX, VAL         ; BH=56 & BL=78
                XCHG        AX, BX          ; AX=5678 & BX=1234
                XCHG        AH, BL          ; AH=34, AL=78, BH=12, & BL=56
                MOV         AX, 4C00H       ; Halt using INT 21h
                INT         21H
CODE  ENDS
END   MAINP
```

Discussion:

Just keep on changing values as desired in the program.

; Program 2: Program for interchanging the values of two Memory locations
; input: Two memory variables of same size: 8-bit for this program

```
DATA SEGMENT
        VALUE1  DB          0Ah             ; Variables
        VALUE2  DB          14h
DATA ENDS
CODE SEGMENT
        ASSUME CS:CODE, DS:DATA
        MOV AX, DATA                        ; Initialise data segments
        MOV DS, AX                          ; using AX
        MOV AL, VALUE1                      ; Load Value1 into AL
        XCHG VALUE2,AL                      ; exchange AL with Value2.
        MOV VALUE1,AL                       ; Store A1 in Value1
        INT 21h                             ; Return to Operating system
        CODE ENDS
END
```

Discussion:

The question is why cannot we simply use XCHG instruction with two memory variables as operand? To answer the question let us look into some of constraints for the MOV & XCHG instructions:

The MOV instruction has the following constraints and operands:

- CS and IP may never be destination operands in MOV;
- Immediate data value and memory variables may not be moved to segment registers;
- The source and the destination operands should be of the same size;
- Both the operands cannot be memory locations;
- If the source is immediate data, it must not exceed 255 (FFh) for an 8-bit destination or 65,535 (FFFFh) for a 16-bit destination.

The statement MOV AL, VALUE1, copies the VALUE1 that is 0Ah in the AL register:

```
AX : 00  0A  <------------------  0A    (VALUE1)
     AH AL                        14    (VALUE2)
```

The instruction, XCHG AL, VALUE2 ; exchanges the value of AL with VALUE2

Now AL and VALUE2 contains and values as under:

```
AX :  00  14  <-------+           0A    (VALUE1)
                      +-------->  0A    (VALUE2)
```

The statement, MOV VALUE1, AL ;, now puts the value of AL to VALUE1.

Thus the desired exchange is complete

```
AX :  00  14  ------------------>  14    (VALUE1)
                                  0A    (VALUE2)
```

Other statements in the above program have already been discussed in the preceding units.

3.2.2 Simple Arithmetic Application

Let us discuss an example that uses simple arithmetic:

; Program 3: Find the average of two values stored in
; memory locations named FIRST and SECOND
; and puts the result in the memory location AVGE.

; Input : Two memory variables stored in memory locations FIRST and SECOND

```
; REGISTERS                  ; Uses DS, CS, AX, BL
; PORTS                      ; None used
DATA          SEGMENT
        FIRST     DB  90h    ; FIRST number,        90h is a sample value
        SECOND  DB  78h      ; SECOND number,       78h is a sample value
        AVGE     DB  ?       ; Store average here
DATA      ENDS
CODE      SEGMENT
        ASSUME CS:CODE, DS:DATA
START:    MOV   AX, DATA          ; Initialise data segment, i.e. set
          MOV   DS, AX            ; Register DS to point to Data Segment
          MOV   AL, FIRST         ; Get first number
          ADD   AL, SECOND        ; Add second to it
          MOV   AH, 00h           ; Clear all of AH register
          ADC   AH,  00h          ; Put carry in LSB of AH
          MOV   BL,  02h          ; Load divisor in BL register
          DIV   BL                ; Divide AX by BL. Quotient in AL,
                                  ; and remainder in AH
          MOV   AVGE, AL          ; Copy result to memory
CODE      ENDS
        END   START
```

Discussion:

An add instruction cannot add two memory locations directly, so we moved a single value in AL first and added the second value to it.

Please note, on adding the two values, there is a possibility of carry bit. (The values here are being treated as unsigned binary numbers). Now the problem is how to put

the carry bit into the AH register such that the AX(AH:AL) reflects the added value. This is done using ADC instruction.

The ADC AH,00h instruction will add the immediate number 00h to the contents of the carry flag and the contents of the AH register. The result will be left in the AH register. Since we had cleared AH to all zeros, before the add, we really are adding 00h + 00h + CF. The result of all this is that the carry flag bit is put in the AH register, which was desired by us.

Finally, to get the average, we divide the sum given in AX by 2. A more general program would require positive and negative numbers. After the division, the 8-bit quotient will be left in the AL register, which can then be copied into the memory location named AVGE.

3.2.3 Application Using Shift Operations

Shift and rotate instructions are useful even for multiplication and division. These operations are not generally available in high-level languages, so assembly language may be an absolute necessity in certain types of applications.

; Program 4: Convert the ASCII code to its BCD equivalent. This can be done by simply replacing the bits in the upper four bits of the byte by four zeros. For example, the ASCII '1' is 32h = 0011 0010B. By making the upper four bits as 0 we get 0000 0010 which is 2 in BCD. The number obtained is called unpacked BCD number. The upper four bits of this byte is zero. So the upper four bits can be used to store another BCD digit. The byte thus obtained is called packed BCD number. For example, an unpacked BCD number 59 is 00000101 00001001, that is, 05 09. The packed BCD will be 0101 1001, that is 59.

The algorithm to convert two ASCII digits to packed BCD can be stated as:

Convert first ASCII digit to unpacked BCD.
Convert the second ASCII digit to unpacked BCD.

Decimal	ASCII	BCD
5	00110101	00000101
9	00111001	00001001

Move first BCD to upper four positions in byte.

0101 0000	Using Rotate Instructions

Pack two BCD bits in one byte.

	0101 0000 0000 1001	
Pack	0101 1001	Using OR

;The assembly language program for the above can be written in the following manner.

```
; ABSTRACT          Program produces a packed BCD byte from 2 ASCII
                  ; encoded digits. Assume the number as 59.

                  ; The first ASCII digit (5) is loaded in BL.
                  ; The second ASCII digit (9) is loaded in AL.
                  ; The result (packed BCD) is left in AL.
```

```
; REGISTERS          ; Uses CS, AL, BL, CL
; PORTS              ; None used
CODE        SEGMENT
            ASSUME      CS:CODE
START:      MOV  BL,    '5'     ; Load first ASCII digit in BL
            MOV  AL,    '9'     ; Load second ASCII digit in AL
            AND  BL,    0Fh     ; Mask upper 4 bits of first digit
            AND  AL,    0Fh     ; Mask upper 4 bits of second digit
            MOV  CL,    04h     ; Load CL for 4 rotates
            ROL  BL,    CL      ; Rotate BL 4 bit positions
            OR   AL,    BL      ; Combine nibbles, result in AL contains 59
                                ; as packed BCD
CODE        ENDS
            END         START
```

Discussion:

8086 does not have any instruction to swap upper and lower four bits in a byte, therefore we need to use the rotate instructions that too by 4 times. Out of the two rotate instructions, ROL and RCL, we have chosen ROL, as it rotates the byte left by one or more positions, on the other hand RCL moves the MSB into the carry flag and brings the original carry flag into the LSB position, which is not what we want.

Let us now look at a program that uses RCL instructions. This will make the difference between the instructions clear.

; Program 5: Add a byte number from one memory location to a byte from the next memory location and put the sum in the third memory location. Also, save the carry flag in the least significant bit of the fourth memory location.

```
; ABSTRACT           : This program adds 2-8-bit words in the memory locations
;                    : NUM1 and NUM2. The result is stored in the memory
;                    : location RESULT. The carry bit, if any will be stored as
;                    : 0000 0001 in the location CARRY

; ALGORITHM:
;          get NUM1
;          add NUM2 in it
;          put sum into memory location RESULT
;          rotate carry in LSB of byte
;          mask off upper seven bits of byte
;          store the result in the CARRY location.
;
; PORTS              : None used
; PROCEDURES         : None used
; REGISTERS          : Uses CS, DS, AX
;
DATA        SEGMENT
      NUM1        DB    25h     ; First number
      NUM2        DB    80h     ; Second number
      RESULT      DB    ?       ; Put sum here
      CARRY       DB
DATA        ENDS
CODE        SEGMENT
      ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA              ; Initialise data segment
      MOV DS, AX                ; register using AX
      MOV AL, NUM1              ; Load the first number in AL
      ADD AL, NUM2              ; Add 2nd number in AL
```

```
        MOV RESULT, AL          ; Store the result
        RCL AL, 01              ; Rotate carry into LSB
        AND AL, 00000001B       ; Mask out all but LSB
        MOV CARRY, AL           ; Store the carry result
        MOV  AH, 4CH
        INT 21H
CODE    ENDS
END     START
```

Discussion:

RCL instruction brings the carry into the least significant bit position of the AL register. The AND instruction is used for masking higher order bits, of the carry, now in AL.

In a similar manner we can also write applications using other shift instructions.

3.2.4 Larger of the Two Numbers

How are the comparisons done in 8086 assembly language? There exists a compare instruction CMP. However, this instruction only sets the flags on comparing two operands (both 8 bits or 16 bits). Compare instruction just subtracts the value of source from destination without storing the result, but setting the flag during the process. Generally only three comparisons are more important. These are:

Result of comparison	**Flag(s) affected**
Destination < source	Carry flag = 1
Destination = source	Zero flag = 1
Destination > source	Carry = 0, Zero = 0

Let's look at three examples that show how the flags are set when the numbers are compared. In example 1 BL is less than 10, so the carry flag is set. In example 2, the zero flag is set because both operands are equal. In example 3, the destination (BX) is greater than the source, so both the zero and the carry flags are clear.

Example 1:

```
        MOV BL, 02h
        CMP BL, 10h         ; Carry flag = 1
```

Example 2:

```
        MOV AX, F0F0h
        MOV DX, F0F0h
        CMP AX,DX           ; Zero flag = 1
```

Example 3:

```
        MOV BX, 200H
        CMP BX, 0           ; Zero and Carry flags = 0
```

In the following section we will discuss an example that uses the flags set by CMP instruction.

☞ Check Your Progress 1

State True or False with respect to 8086/8088 assembly languages. | T | F |

1. In a MOV instruction, the immediate operand value for 8-bit destination cannot exceed F0h. ☐
2. XCHG VALUE1, VALUE2 is a valid instruction. ☐
3. In the example given in section 3.2.2 we can change instruction DIV BL with a shift. ☐
4. A single instruction cannot swap the upper and lower four of a byte register. ☐
5. An unpacked BCD number requires 8 bits of storage, however, two unpacked BCD numbers can be packed in a single byte register. ☐
6. If AL = 05 and BL = 06 then CMP AL, BL instruction will clear the zero and carry flags. ☐

3.3 PROGRAMMING WITH LOOPS AND COMPARISONS

Let us now discuss a few examples which are slightly more advanced than what we have been doing till now. This section deals with more practical examples using loops, comparison and shift instructions.

3.3.1 Simple Program Loops

The loops in assembly can be implemented using:

- Unconditional jump instructions such as JMP, or
- Conditional jump instructions such as JC, JNC, JZ, JNZ etc. and
- Loop instructions.

Let us consider some examples, explaining the use of conditional jumps.

Example 4:

```
        CMP   AX,BX        ; compare instruction: sets flags
        JE    THERE        ; if equal then skip the ADD instruction
        ADD   AX, 02       ; add 02 to AX

THERE:  MOV   CL, 07       ; load 07 to CL
```

In the example above the control of the program will directly transfer to the label THERE if the value stores in AX register is equal to that of the register BX. The same example can be rewritten in the following manner, using different jumps.

Example 5:

```
        CMP   AX, BX       ; compare instruction: sets flags
        JNE   FIX          ; if not equal do addition
        JMP   THERE        ; if equal skip next instruction
FIX:    ADD   AX, 02       ; add 02 to AX
```

```
THERE:  MOV    CL, 07
```

The above code is not efficient, but suggest that there are many ways through which a conditional jump can be implemented. Select the most optimum way.

Example 6:

```
        CMP    DX, 00          ; checks if DX is zero.
        JE     Label1          ; if yes, jump to Label1 i.e. if ZF=1

Label1:----                    ; control comes here if DX=0
```

Example 7:

```
        MOV    AL, 10          ; moves 10 to AL
        CMP    AL, 20          ; checks if AL < 20 i.e. CF=1
        JL     Lab1            ; carry flag = 1 then jump to Lab1

        Lab1: ------           ; control comes here if condition is satisfied
```

LOOPING

; Program 6: Assume a constant inflation factor that is added to a series of prices
; stored in the memory. The program copies the new price over the old price. It is
; assumed that price data is available in BCD form.

; The algorithm:

```
;Repeat
;          Read a price from the array
;          Add inflation factor
;          Adjust result to correct BCD
;          Put result back in array
;         Until all prices are inflated

; REGISTERS: Uses DS, CS, AX, BX, CX
; PORTS      : Not used
ARRAYS      SEGMENT
            PRICE        DB      36h, 55h, 27h, 42h, 38h, 41h, 29h, 39h
ARRAYS      ENDS
CODE        SEGMENT
            ASSUME CS:CODE, DS:ARRAYS
START:      MOV   AX, ARRAYS  ; Initialize data segment
            MOV   DS, AX      ; register using AX
            LEA   BX, PRICES  ; initialize pointer to base of array
            MOV   CX, 0008h   ; Initialise counter to 8 as array have 8
                              ; values.
DO_NEXT:    MOV   AL, [BX]    ; Copy a price to AL. BX is addressed in
                              ; indirect mode.
            ADD   AL, 0Ah     ; Add inflation factor
            DAA               ; Make sure that result is BCD
            MOV   [BX], AL    ; Copy result back to the memory
            INC   BX          ; increment BX to make it point to next price
            DEC   CX          ; Decrement counter register
            JNZ   DO_NEXT     : If not last, (last would be when CX will
                              ; become 0) Loop back to DO_NEXT
            MOV   AH, 4CH     ; Return to DOS
            INT   21H
CODE        ENDS
            END   START
```

Discussion:

Please note the use of instruction: LEA BX,PRICES: It will load the BX register with the offset of the array PRICES in the data segment. [BX] is an indirection through BX and contains the value stored at that element of array. PRICES. BX is incremented to point to the next element of the array. CX register acts as a loop counter and is decremented by one to keep a check of the bounds of the array. Once the CX register becomes zero, zero flag is set to 1. The JNZ instruction keeps track of the value of CX, and the loop terminates when zero flag is 1 because JNZ does not loop back. The same program can be written using the LOOP instruction, in such case, DEC CX and JNZ DO_NEXT instructions are replaced by LOOP DO_NEXT instruction. LOOP decrements the value of CX and jumps to the given label, only if CX is not equal to zero.

Let us demonstrate the use of LOOP instruction, with the help of following program:

; Program 7: This following program prints the alphabets (A-Z)

```
; Register used : AX, CX, DX

CODE  SEGMENT
        ASSUME : CS:CODE.
MAINP:  MOV   CX,  1AH      ; 26 in decimal  = 1A in  hexadecimal Counter.
        MOV   DL, 41H       ; Loading DL with ASCII hexadecimal of  A.
NEXTC:  MOV   AH, 02H       ; display result character in DL
        INT   21H           ;  DOS interrupt
        INC   DL            ; Increment  DL for next char
        LOOP  NEXTC         ;  Repeat until CX=0.(loop automatically decrements
                            ;  CS and checks whether it is zero or not)
        MOV   AX, 4C00H     ;  Exit DOS
        INT   21H           ;  DOS Call
CODE ENDS
END MAINP
```

Let us now discuss a slightly more complex looping program.

; Program 8: This program compares a pair of characters entered through keyboard.
; Registers used: AX, BX, CX, DX

```
DATA SEGMENT
        XX  DB  ?
        YY  DB  ?
DATA ENDS

CODE SEGMENT
        ASSUME    CS:  CODE,   DS: DATA
MAINP:  MOV  AX,  DATA          ; initialize data
        MOV  DS,  AX            ; segment using AX
        MOV  CX,  03H           ; set counter to 3.
NEXTP:  MOV  AH,  01H           ; Waiting for user to enter a char.
        INT  21H
        MOV  XX, AL             ; store the 1st input character in XX
        MOV  AH,  01H           ; waiting for user to enter second
        INT  21H                ;          character.
        MOV  YY, AL             ; store the character to YY
        MOV  BH, XX             ; load first character in BH
        MOV  BL, YY             ; load second character in BL
        CMP  BH,  BL            ; compare the characters
        JNE  NOT_EQUAL          ;
```

```
EQUAL:        MOV  AH, 02H          ; if characters  are equal then control
              MOV  DL,  'Y'                 ;  will execute this block and
               INT    21H                    ; display 'Y'
              JMP  CONTINUE                 ;  Jump to continue loop.

NOT_EQUAL: MOV  AH, 02H                     ; if characters  are not equal then
                                            control
               MOV  DL,  'N''               ;  will execute this block and
               INT  21 H                    ;  display 'N'

CONTINUE :    LOOP NEXT P                   ; Get the next character
               MOV AH, 4C H                 ; Exit to DOS
               INT  21 H
CODE ENDS
END MAINP
```

Discussion:

This program will be executed, at least 3 times.

3.3.2 Find the Largest and the Smallest Array Values

Let us now put together whatever we have done in the preceding sections and write down a program to find the largest and the smallest numbers from a given array. This program uses the JGE (jump greater than or equal to) instruction, because we have assumed the array values as signed. We have not used the JAE instruction, which works correctly for unsigned numbers.

; Program 9: Initialise the **smallest** and the **largest** variables as the first number in
; the array. They are then compared with the other array values one by one. If the
; value happens to be smaller than the assumed smallest number or larger than the
; assumed largest value, the **smallest** and the **largest** variables are changed with the
; new values respectively. Let us use register DI to point the current array value and
; LOOP instruction for looping.

```
DATA          SEGMENT
              ARRAY        DW     -1, 2000, -4000, 32767, 500,0
              LARGE        DW     ?
              SMALL        DW     ?
DATA          ENDS
END.

CODE          SEGMENT
              MOV    AX,DATA
              MOV    DS,AX                  ; Initialize DS
              MOV    DI, OFFSET ARRAY       ; DI points to the array
              MOV    AX, [DI]               ; AX contains the first element
              MOV    DX, AX                 ; initialize large in DX register
              MOV    BX, AX                 ; initialize small in BX register
              MOV    CX, 6                  ; initialize loop counter
A1:           MOV    AX, [DI]               ; get next array value
              CMP    AX, BX                 ; Is the new value smaller?
              JGE    A2                     ; If greater then (not smaller) jump to
                                            ; A2, to check larger than large in DX
              MOV    BX, AX                 ; Otherwise it is smaller so move it to
                                            ; the smallest value  (BX register)
              JMP    A3                     ; as it is small, thus no need
                                            ; to compare it with the large so jump
                                            ; to A3 to continue or terminate loop.
```

```
A2:     CMP   AX, DX        ; [DI] = large
        JLE   A3            ; if less than it implies not large so
                            ; jump to A3
                            ; to continue or terminate
        MOV   DX, AX        ; otherwise it is larger value, so move
                            ; it to DX that store the large value
A3:     ADD   DI, 2         ; DI now points to next number
        LOOP  A1            ; repeat the loop until CX = 0
        MOV   LARGE, DX
        MOV   SMALL, BX     ; move the large and small in the
                            ; memory locations
        MOV   AX, 4C00h
        INT   21h           ; halt, return to DOS
CODE    ENDS
```

Discussion:

Since the data is word type that is equal to 2 bytes and memory organisation is byte wise, to point to next array value DI is incremented by 2.

3.3.3 Character Coded Data

The input output takes place in the form of ASCII data. These ASCII characters are entered as a string of data. For example, to get two numbers from console, we may enter the numbers as:

Enter first number	1234
Enter second number	3210
The sum is	4444

As each digit is input, we would store its ASCII code in a memory byte. After the first number was input the number would be stored as follows:

The number is entered as:

31	32	33	34	hexadecimal storage
1	2	3	4	ASCII digits

Each of these numbers will be input as equivalent ASCII digits and need to be converted either to digit string to a 16-bit binary value that can be used for computation or the ASCII digits themselves can be added which can be followed by instruction that adjust the sum to binary. Let us use the conversion operation to perform these calculations here.

Another important data format is packed decimal numbers (packed BCD). A packed BCD contains two decimal digits per byte. Packed BCD format has the following advantages:

- The BCD numbers allow accurate calculations for almost any number of significant digits.
- Conversion of packed BCD numbers to ASCII (and vice versa) is relatively fast.
- An implicit decimal point may be used for keeping track of its position in a separate variable.

The instructions DAA (decimal adjust after addition) and DAS (decimal adjust after subtraction) are used for adjusting the result of an addition of subtraction operation on

packed decimal numbers. However, no such instruction exists for multiplication and division. For the cases of multiplication and division the number must be unpacked. First, multiplied or divided and packed again. The instruction DAA and DAS has already been explained in unit 1.

3.3.4 Code Conversion

The conversion of data from one form to another is needed. Therefore, in this section we will discuss an example, for converting a hexadecimal digit obtained in ASCII form to binary form. Many ASCII to BCD and other conversion examples have been given earlier in unit 2.

Program 10:

```
;       This program converts an ASCII input to equivalent hex digit that it represents.
;       Thus, valid ASCII digits are 0 to 9, A to F and the program assumes that the
;       ASCII digit is read from a location in memory called ASCII. The hex result is
;       left in the AL. Since the program converts only one digit number the AL is
;       sufficient for the results. The result in AL is made FF if the character in ASCII
;       is not the proper hex digit.
; ALGORITHM
;       IF number <30h THEN error
;       ELSE
;       IF number <3Ah THEN Subtract 30h (it's a number 0-9)
;       ELSE (number is >39h)
;       IF number <41h THEN error (number in range 3Ah-40h which is not a valid
;        A-F character range)
;       ELSE
;       IF number <47h THEN Subtract 37h for letter A-F 41-46 (Please note
;       that 41h – 37h = Ah)
;       ELSE            ERROR
;
; PORTS           : None used
; PROCEDURES      : None
; REGISTERS       : Uses CS, DS, AX,
;
DATA        SEGMENT
            ASCII   DB 39h          ; Any experimental data
DATA        ENDS
CODE        SEGMENT
        ASSUME CS:CODE, DS:DATA
START:      MOV     AX, DATA        ; initialise data segment
            MOV     DS, AX          ; Register using AX
            MOV     AL, ASCII       ; Get the ASCII digits of the number
                                    ; start the conversion
            CMP     AL, 30h         ; If the ASCII digit is below 30h then it is not
            JB      ERROR           ; a proper Hex digit
            CMP     AL, 3Ah         ; compare it to 3Ah
            JB      NUMBER          ; If greater then possibly a letter between A-F
            CMP     AL, 41h         ; This step will be done if equal to or above
                                    ; 3Ah
            JB      ERROR           ; Between 3Ah and 40h is error
            CMP     AL, 46h
            JA      ERROR           ; The ASCII is out of 0-9 and A-F range
            SUB     AL, 37h         ; It's a letter in the range A-F so convert
            JMP     CONVERTED
NUMBER:     SUB     AL, 30h         ; it is a number in the range 0-9 so convert
            JMP     CONVERTED
```

```
ERROR:      MOV   AL, 0FFh      ; You can also display some message here
CONVERTED:  MOV   AX, 4C00h
            INT   21h           ; the hex result is in AL
CODE        ENDS
            END   START
```

Discussions:

The above program demonstrates a single hex digit represented by an ASCII character. The above programs can be extended to take more ASCII values and convert them into a 16-bit binary number.

☞ Check Your Progress 2

1. Write the code sequence in assembly for performing following operation:

   Z = ((A – B) / 10 * C) * * 2

   ……

   ……

2. Write an assembly code sequence for adding an array of binary numbers.

   ……

   ……

3. An assembly program is to be written for inputting two 4 digits decimal numbers from console, adding them up and putting back the results. Will you prefer packed BCD addition for such numbers? Why?

   ……

   ……

4. How can we implement nested loops, for example,

   ```
   For (i = 1 to 10, step 1)
           { for (j = 1 to, step 1)
                   add 1 to AX}
   ```

   in assembly language?

   ……

   ……

3.4 PROGRAMMING FOR ARITHMETIC AND STRING OPERATIONS

Let us discuss some more advanced features of assembly language programming in this section. Some of these features give assembly an edge over the high level language programming as far as efficiency is concerned. One such instruction is for string processing. The object code generated after compiling the HLL program containing string instruction is much longer than the same program written in assembly language. Let us discuss this in more detail in the next subsection:

3.4.1 String Processing

Let us write a program for comparing two strings. Consider the following piece of code, which has been written in C to compare two strings. Let us assume that ‘str1’ and ‘str2’ are two strings, initialised by some values and ‘ind’ is the index for these character strings:

```
for (ind = 0; ( (ind <9) and (str1[ind] = = str2[ind]) ), ind + +)
```

The intermediate code in assembly language generated by a non-optimising compiler for the above piece may look like:

```
        MOV     IND, 00                 ; ind : = 0
L3:     CMP     IND, 08                 ; ind < 9
        JG      L1                      ; not so; skip
        LEA     AX, STR1                ; offset of str1 in AX register
        MOV     BX, IND                 ; it uses a register for indexing into
                                        ; the array
        LEA     CX, STR2                ; str2 in CX
        MOV     DL, BYTE PTR CX[BX]
        CMP     DL, BYTE PTR AX[BX]             ; str1[ind] = str2[ind]
        JNE     L1                              ; no, skip
        MOV     IND, BX
        ADD     IND, 01
L2:     JMP     L3                              ; loop back
L1:
```

What we find in the above code: a large code that could have been improved further, if the 8086 string instructions would have been used.

; Program 11: Matching two strings of same length stored in memory locations.
; REGISTERS : Uses CS, DS, ES, AX, DX, CX, SI, DI

```
DATA        SEGMENT
            PASSWORD    DB      'FAILSAFE'      ; source string
            DESTSTR     DB      'FEELSAFE'      ; destination string
            MESSAGE     DB      'String are equal $'
DATA        ENDS
CODE        SEGMENT
        ASSUME  CS:CODE, DS:DATA, ES:DATA
            MOV     AX, DATA
            MOV     DS, AX                  ; Initialise data segment register
            MOV     ES, AX                  ; Initialise extra segment register
; as destination string is considered to be in extra segment. Please note that ES is also
; initialised to the same segment as of DS.
            LEA     SI, PASSWORD            ; Load source pointer
            LEA     DI, DESTSTR             ; Load destination pointer
            MOV     CX, 08                  ; Load counter with string length
            CLD                     ; Clear direction flag so that comparison is
                                    ; done in forward direction.

            REPE    CMPSB           ; Compare the two string byte by byte
            JNE     NOTEQUAL        ; If not equal, jump to NOTEQUAL
            MOV     AH, 09          ; else display message
            MOV     DX, OFFSET MESSAGE ;
            INT     21h             ; display the message
NOTEQUAL:MOV        AX, 4C00h       ; interrupt function to halt
            INT     21h
CODE        ENDS
            END
```

Discussion:

In the above program the instruction CMPSB compares the two strings, pointed by SI in Data Segment and DI register in extra data segment. The strings are compared byte by byte and then the pointers SI and DI are incremented to next byte. Please note the last letter B in the instruction indicates a byte. If it is W, that is if instruction is CMPSW, then comparison is done word by word and SI and DI are incremented by 2,

that is to next word. The REPE prefix in front of the instruction tells the 8086 to decrement the CX register by one, and continue to execute the CMPSB instruction, until the counter (CX) becomes zero. Thus, the code size is substantially reduced.

Similarly, you can write efficient programs for moving one string to another, using MOVS, and scanning a string for a character using SCAS.

3.4.2 Some More Arithmetic Problems

Let us now take up some more practical arithmetic problems.

Use of delay loops

A very useful application of assembly is to produce delay loops. Such loops are used for waiting for some time prior to execution of next instruction.

But how to find the time for the delay? The rate at which the instructions are executed is determined by the clock frequency. Each instruction takes a certain number of clock cycles to execute. This, multiplied by the clock frequency of the microprocessor, gives the actual time of execution of a instruction. For example, MOV instruction takes four clock cycles. This instruction when run on a microprocessor with a 4Mhz clock takes 4/4, i.e. 1 microsecond. NOP is an instruction that is used to produce the delay, without affecting the actual running of the program.

Time delay of 1 ms on a microprocessor having a clock frequency of 5 MHz would require:

$$1 \text{ clock cycle} = \frac{1}{5\text{MHz}}$$

$$= \frac{1}{5 \times 10^6} \text{ Seconds}$$

Thus, a 1-millisecond delay will require:

$$= \frac{1 \times 10^{-3}}{\left(\frac{1}{5 \times 10^6}\right)} \text{ clock cycles}$$

= 5000 clock cycles.

The following program segment can be used to produce the delay, with the counter value correctly initialised.

```
        MOV     CX, N          ; 4 clock cycles  N will vary depending on
                               ; the amount of delay required

DELAY:          NOP            ; 3 cycles
                NOP            ; 3 cycles
                LOOP DELAY     ; 17 or 5
```

LOOP instruction takes 17 clock cycles when the condition is true and 5 clock cycles otherwise. The condition will be true, 'N' number of times and false only once, when the control comes out of the loop.

To calculate 'N':

Total clock cycles = clock cycles for MOV + N(2*NOP clock cycles + 17) – 12 (when CX = 0)

5000 = 4 + N(6 + 17) – 12
N = 5000/23 = 218 = 0DAh

Therefore, the counter, CX, should be initialized by 0DAh, in order to get the delay of 1 millisecond.

Use of array in assembly

Let us write a program to add two 5-byte numbers stored in an array. For example, two numbers in hex can be:

	20	11	01	10	FF
	FF	40	30	20	10
1	1F	51	31	31	1F

Carry

Let us also assume that the numbers are represented as the lowest significant byte first and put in memory in two arrays. The result is stored in the third array SUM. The SUM also contains the carry out information, thus would be 1 byte longer than number arrays.

```
; Program 12: Add two five-byte numbers using arrays
; ALGORITHM:
;           Make count = LEN
;           Clear the carry flag
;           Load address of NUM1
;           REPEAT
;               Put byte from NUM1 in accumulator
;               Add byte from NUM2 to accumulator + carry
;               Store result in SUM
;               Decrement count
;               Increment to next address
;           UNTIL count = 0
;           Rotate carry into LSB of accumulator
;           Mask all but LSB of accumulator
;           Store carry result, address pointer in correct position.
; PORTS             : None used
; PROCEDURES        : None used
; REGISTERS         : Uses CS, DS, AX, CX, BX, DX

DATA    SEGMENT
        NUM1    DB      0FFh,   10h     ,01h    ,11h    ,20h
        NUM2    DB      10h,    20h,    30h,    40h     ,0FFh
        SUM     DB      6DUP(0)
DATA    ENDS
        LEN     EQU     05h     ; constant for length of the array

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV     AX, DATA        ; initialise data segment
        MOV     DS, AX          ; using AX register
        MOV     SI, 00          ; load displacement of 1st number.
                                ; SI is being used as index register
        MOV     CX, 0000        ; clear counter
        MOV     CL, LEN         ; set up count to designed length
        CLC                     ; clear carry. Ready for addition
AGAIN:  MOV     AL, NUM1[SI]    ; get a byte from NUM1
        ADC     AL, NUM2[SI]    ; add to byte from NUM2 with carry
```

```
            MOV     SUM[SI], AL   ; store in SUM array
            INC     SI
            LOOP    AGAIN         ; continue until no more bytes
            RCL     AL, 01h       ; move carry into bit 0 of AL
            AND     AL, 01h       ; mask all but the 0th bit of AL
            MOV     SUM[SI], AL   ; put carry into 6th byte
FINISH:     MOV     AX, 4C00h
            INT     21h
CODE        ENDS
            END     START
```

;Program 13: A good example of code conversion: Write a program to convert a
; 4-digit BCD number into its binary equivalent. The BCD number is stored as a
; word in memory location called BCD. The result is to be stored in location HEX.

```
; ALGORITHM:
;           Let us assume the BCD number as 4567
;           Put the BCD number into 4, 16bit registers
;           Extract the first digit (4 in this case)
;           by masking out the other three digits. Since, its place value is 1000.
;           So Multiply by 3E8h (that is 1000 in hexadecimal) to get 4000 = 0FA0h
;           Extract the second digit (5)
;           by masking out the other three digits.
;           Multiply by 64h (100)
;           Add to first digit and get 4500 = 1194h
;           Extract the third digit (6)
;           by masking out the other three digits (0060)
;           Multiply by 0Ah (10)
;           Add to first and second digit to get 4560 = 11D0h
;           Extract the last digit (7)
;           by masking out the other three digits (0007)
;           Add the first, second, and third digit to get 4567 = 11D7h
; PORTS       : None used
; REGISTERS: Uses CS, DS, AX, CX, BX, DX

THOU        EQU     3E8h          ; 1000 = 3E8h
DATA        SEGMENT
            BCD     DW    4567h
            HEX     DW    ?       ; storage reserved for result
DATA        ENDS

CODE        SEGMENT
        ASSUME CS:CODE, DS:DATA
START:      MOV     AX, DATA      ; initialise data segment
            MOV     DS, AX        ; using AX register
            MOV     AX, BCD       ; get the BCD number AX = 4567
            MOV     BX, AX        ; copy number into BX; BX = 4567
            MOV     AL, AH        ; place for upper 2 digits in AX = 4545
            MOV     BH, BL        ; place for lower 2 digits in BX = 6767
                                  ; split up numbers so that we have one digit
                                  ; in each register
            MOV     CL, 04        ; bit count for rotate
            ROR     AH, CL        ; digit 1 (MSB) in lower four bits of AH.
                                  ; AX = 54 45
            ROR     BH, CL        ; digit 3 in lower four bits of BH.
                                  ; BX = 76 67
            AND     AX, 0F0FH     ; mask upper four bits of each digit.
                                  ; AX = 04 05
```

```
        AND     BX, 0F0FH     ; BX = 06 07
        MOV     CX, AX        ; copy AX into CX so that can use AX for
                              ; multiplication CX = 04 05

        ; CH contains digit 4 having place value 1000, CL contains digit 5
        ; having place value 100, BH contains digit 6 having place value 10 and
        ; BL contains digit 7 having unit place value.
        ; so obtain the number as CH × 1000 + CL × 100 + BH  × 10 + BL

        MOV     AX, 0000H     ; zero AH and AL
                              ; now multiply each number by its place
                              ; value
        MOV     AL, CH        ; digit 1 to AL for multiply
        MOV     DI, THOU      ; no immediate multiplication is allowed so
                              ; move thousand to DI
        MUL     DI            ; digit 1 (4)*1000
                              ; result in DX and AX. Because BCD digit
                              ; will not be greater than 9999, the result will
                              ; be in AX only. AX = 4000
        MOV     DH, 00H       ; zero DH
        MOV     DL, BL        ; move BL to DL, so DL = 7
        ADD     DX, AX        ; add AX; so DX = 4007
        MOV     AX, 0064h     ; load value for 100 into AL
        MUL     CL            ; multiply by digit 2 from CL
        ADD     DX, AX        ; add to total in DX.  DX now contains
                              ; (7 +  4000 + 500)
        MOV     AX, 000Ah     ; load value of 10 into AL
        MUL     BH            ; multiply by digit 3 in BH
        ADD     DX, AX        ; add to total in DX; DX contains
                              ;  (7 + 4000 + 500 +60)
        MOV     HEX, DX       ; put result in HEX for return
        MOV     AX, 4C00h
        INT     21h
CODE    ENDS
        END     START
```

☞ Check Your Progress 3

1. Why should we perform string processing in assembly language in 8086 and not in high-level language?

   ……

   ……

   ……

2. What is the function of direction flag?

   ……

   ……

   ……

3. What is the function of NOP statement?

   ……

   ……

   ……

3.5 SUMMARY

In this unit, we have covered some basic aspects of assembly language programming. We started with some elementary arithmetic problems, code conversion problems, various types of loops and graduated on to do string processing and slightly complex arithmetic. As part of good programming practice, we also noted some points that should be kept in mind while coding. Some of them are:

- An algorithm should always precede your program. It is a good programming practice. This not only increases the readability of the program, but also makes your program less prone to logical errors.
- Use comments liberally. You will appreciate them later.
- Study the instructions, assembler directives and addressing modes carefully, before starting to code your program. You can even use a debugger to get a clear understanding of the instructions and addressing modes.
- Some instructions are very specific to the type of operand they are being used with, example signed numbers and unsigned numbers, byte operands and word operands, so be careful !!
- Certain instructions except some registers to be initialised by some values before being executed, example, LOOP expects the counter value to be contained in CX register, string instructions expect DS:SI to be initialised by the segment and the offset of the string instructions, and ES:DI to be with the destination strings, INT 21h expects AH register to contain the function number of the operation to be carried out, and depending on them some of the additional registers also to be initialised. So study them carefully and do the needful. In case you miss out on something, in most of the cases, you will not get an error message, instead the 8086 will proceed to execute the instruction, with whatever junk is lying in those registers.

In spite of all these complications, assembly languages is still an indispensable part of programming, as it gives you an access to most of the hardware features of the machine, which might not be possible with high level language. Secondly, as we have also seen some kind of applications can be written and efficiently executed in assembly language. We justified this with string processing instructions; you will appreciate it more when you actually start doing the assembly language programming. You can now perform some simple exercises from the further readings.

In the next block, we take up more advanced assembly language programming, which also includes accessing interrupts of the machine.

3.6 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. False 2. False 3. True 4. True 5. True 6. False

Check Your Progress 2

1.
```
MOV     AX, A       ; bring A in AX
SUB     AX, B       ; subtract B
MOV     DX, 0000h   ; move 0 to DX as it will be used for word division
MOV     BX, 10      ; move dividend to BX
IDIV    BX          ; divide
IMUL    C           ; ( (A-B) / 10 * C) in AX
IMUL    AX          ; square AX to get (A-B/10 * C) * * 2
```

2. Assuming that each array element is a word variable.

```
        MOV     CX, COUNT     ; put the number of elements of the array in
                              ; CX register
        MOV     AX, 0000h     ; zero SI and AX
        MOV     SI, AX
; add the elements of array in AX again and again
AGAIN:  ADD AX, ARRAY[SI]     ; another way of handling array
        ADD     SI, 2         ; select the next element of the array
        LOOP    AGAIN         ; add all the elements of the array. It will
                                terminate when CX becomes zero.
        MOV     TOTAL, AX     ; store the results in TOTAL.
```

3. Yes, because the conversion efforts are less.
4. We may use two nested loop instructions in assembly also. However, as both the loop instructions use CX, therefore every time before we are entering inner loop we must push CX of outer loop in the stack and reinitialize CX to the inner loop requirements.

Check Your Progress 3

1. The object code generated on compiling high level languages for string processing commands is, in general, found to be long and contains several redundant instructions. However, we can perform string processing very efficiently in 8086 assembly language.

2. Direction flag if clear will cause REPE statement to perform in forward direction. That is, in the given example the strings will be compared from first element to last.

3. It produces a delay of a desired clock time in the execution. This instruction is useful while development of program. A collection of these instructions can be used to fill up some space in the code segment, which can be changed with new code lines without disturbing the position of existing code. This is particularly used when a label is specified.

UNIT 4 ASSEMBLY LANGUAGE PROGRAMMING (PART-II)

Structure **Page No.**

4.0	Introduction	77
4.1	Objectives	77
4.2	Use of Arrays in Assembly	77
4.3	Modular Programming	80
	4.3.1 The stack	
	4.3.2 FAR and NEAR Procedures	
	4.3.3 Parameter Passing in Procedures	
	4.3.4 External Procedures	
4.4	Interfacing Assembly Language Routines to High Level Language Programs	93
	4.4.1 Simple Interfacing	
	4.4.2 Interfacing Subroutines With Parameter Passing	
4.5	Interrupts	97
4.6	Device Drivers in Assembly	99
4.7	Summary	101
4.8	Solutions/ Answers	102

4.0 INTRODUCTION

In the previous units, we have discussed the instruction set, addressing modes, and other tools, which are needed to develop assembly language programs. We shall now use this knowledge in developing more advanced tools. We have divided this unit broadly into four sections. In the first section, we discuss the design of some simple data structures using the basic data types. Once the programs become lengthier, it is advisable to divide them into small modules, which can be easily written, tested and debugged. This leads to the concept of modular programming, and that is the topic of our second section in this unit. In the third section, we will discuss some techniques to interface assembly language programs to high level languages. We have explained the concepts using C and C ++ as they are two of the most popular high-level languages. In the fourth section we have designed some tools necessary for interfacing the microprocessor with external hardware modules.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- implement simple data structures in assembly language;
- write modular programs in assembly language;
- interface assembly program to high level language program; and
- analyse simple interrupt routines.

4.2 USE OF ARRAYS IN ASSEMBLY

An array is referencing using a base array value and an index. To facilitate addressing in arrays, 8086 has provided two index registers for mathematical computations, viz. BX and BP. In addition two index registers are also provided for string processing, viz. SI and DI. In addition to this you can use any general purpose register also for indexing.

An important application of array is the tables that are used to store related information. For example, the names of all the students in the class, their CGPA, the list of all the books in the library, or even the list of people residing in a particular area can be stored in different tables. An important application of tables would be character translation. It can be used for data encryption, or translation from one data type to another. A critical factor for such kind of applications is the speed, which just happens to be a strength of assembly language. The instruction that is used for such kind of applications is XLAT.

Let us explain this instruction with the help of an example:

Example 1:

Let us assume a table of hexadecimal characters representing all 16 hexadecimal digits in table:

```
HEXA    DB    '0123456789ABCDEF'
```

The table contains the ASCII code of each hexadecimal digit:

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Contents	30	31	32	33	34	35	36	37	38	39	41	42	43	44	45	46

(all value in hexadecimal)

If we place 0Ah in AL with the thought of converting it to ASCII, we need to set BX to the offset of HEXA, and invoke XLAT. You need not specify the table name with XLAT because it is implicitly passed by setting BX to the HEXA table offset. This instruction will do the following operations:

It will first add BX to AL, generating an effective address that points to the eleventh entry in the HEXA table.

The content of this entry is now moved to the AL register, that is, 41h is moved to AL.

In other words, XLAT sets AL to 41h because this value is located at HEXA table offset 0Ah. Please note that the 41h is the ASCII code for hex digit A. The following sequence of instructions would accomplished this:

```
MOV   AL,   0Ah              ; index value
MOV   BX,   OFFSET HEXA      ; offset of the table HEXA
XLAT
```

The above tasks can be done without XLAT instruction but it will require a long series of instructions such as:

```
MOV     AL, 0Ah              ; index value
MOV     BX, OFFSET HEXA      ; offset of the table HEXA
PUSH    BX                   ; save the offset
ADD     BL, AL               ; add index value to table
                             ; HEXA offset
MOV     AL, [BX]             ; retrieve the entry
POP     BX                   ; restore BX
```

Let us use the instruction XLAT for data encoding. When you want to transfer a message through a telephone line, then such encoding may be a good way of preventing other users from reading it. Let us show a sample program for encoding.

PROGRAM 1:

; A program for encoding ASCII Alpha numerics.

; ALGORITHM:
; create the code table
; read an input string character by character
; translate it using code table
; output the strings

```
DATA        SEGMENT
            CODETABLE       DB 48 DUP (0)       ; no translation of first
                                                ; 48 ASCII
                            DB '4590821367'     ; ASCII codes 48 –
                                                ; 57 ≡ (30h – 39h)
                            DB 7 DUP (0)        ; no translation of
                                                 these 7 characters
                            DB   'GVHZUSOBMIKPJCADLFTYEQNWXR'
                            DB 6 DUP (0)        ; no translation
                            DB   'gvhzusobmikpjcadlftyeqnwxr'
                            DB 133 DUP (0)      ; no translation of remaining
                                                ; character
DATA        ENDS

CODE        SEGMENT
            MOV     AX, DATA
            MOV     DS, AX              ; initialize DS
            MOV     BX, OFFSET CODETABLE        ; point to lookup table
GETCHAR:
            MOV     AH, 06              ; console input no wait
            MOV     DL, 0FFh            ; specify input request
            INT     21h                 ; call DOS
            JZ      QUIT                ; quit if no input is waiting
            MOV     DL, AL              ; save character in DL
            XLAT    CODETABLE           ; translate the character
            CMP     AL, 0               ; translatable?
            JE      PUTCHAR             ; no : write it as is.
            MOV     DL, AL              ; yes : move new character
                                        ; to DL
PUTCHAR:
            MOV     AH, 02              ; write DL to output
            INT     21h
            JMP     GETCHAR             ; get another character
QUIT:       MOV     AX, 4C00h
            INT     21h
CODE        ENDS
END
```

Discussion:

The program above will code the data. For example, a line from an input file will be encoded:

A SECRET Message (Read from an input file)
G TUHFUY Juttgou (Encoded output)

The program above can be run using the following command line. If the program file name is coding.asm

coding infile > outfile

The infile is the input data file, and outfile is the output data file.
You can write more such applications using 8086 assembly tables.

☞ Check Your Progress 1

1. Write a program to convert all upper case letters to lower case.
   ……
   ……
   ……

2. **State True or False** | T | F |

   a. Table handling cannot be done without using XLAT instruction. ☐

   b. In XLAT instruction AX register contains the address of the first entry of the table. ☐

   c. In XLAT instruction the desired element value is returned in AL register. ☐

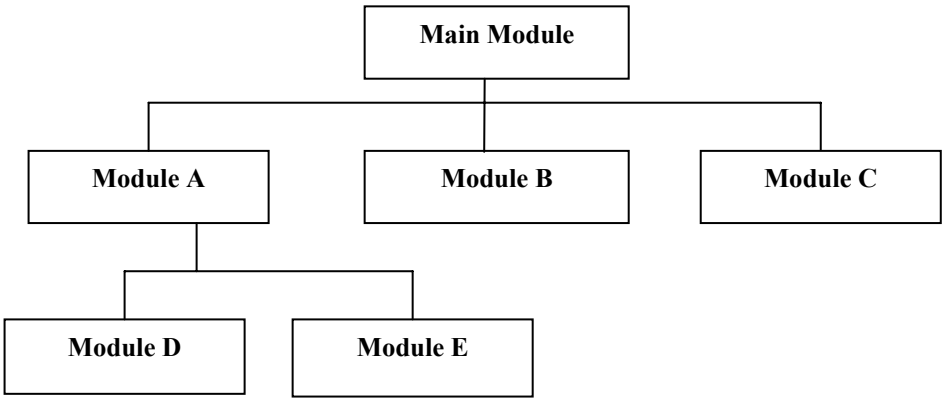
4.3 MODULAR PROGRAMMING

Modular programming refers to the practice of writing a program as a series of independently assembled source files. Each source file is a modular program designed to be assembled into a separate object file. Each object file constitutes a module. The linker collects the modules of a program into a coherent whole.

There are several reasons a programmer might choose to modularise a program.

1. Modular programming permits breaking a large program into a number of smaller modules each of more manageable size.
2. Modular programming makes it possible to link source code written in two separate languages. A hybrid program written partly in assembly language and partly in higher level language necessarily involves at least one module for each language involved.
3. Modular programming allows for the creation, maintenance and reuse of a library of commonly used modules.
4. Modules are easy to comprehend.
5. Different modules can be assigned to different programs.
6. Debugging and testing can be done in a more orderly fashion.
7. Document action can be easily understood.
8. Modifications may be localised to a module.

A modular program can be represented using hierarchical diagram:



The advantages of modular programming are:

1. Smaller, easier modules to manage
2. Code repetition may be avoided by reusing modules.

You can divide a program into subroutines or procedures. You need to CALL the procedure whenever needed. A subroutine call transfers the control to subroutine instructions and brings the control back to calling program.

4.3.1 The Stack

A procedure call is supported by a stack. So let us discuss stack in assembly. Stacks are Last In First Out data structures, and are used for storing the return addresses of the procedures and for parameter passing and saving the return value.

In 8086 microprocessor a stack is created in the stack segment. The SS register stores the offset of stack segment and SP register stores the top of the stack. A value is pushed in to top of the stack or taken out (poped) from the top of the stack. The stack segment can be initialized as follows:

```
STACK_ SEG SEGMENT STACK
        DW 100          DUP (0)
        TOS LABEL       WORD
STACK_SEG ENDS

CODE SEGMENT
            ASSUME CS:CODE, SS:STACK_SEG
            MOV   AX, STACK_SEG
            MOV   SS,AX                 ; initialise stack segment
            LEA   SP,TOP                ; initialise stack pointer
CODE  ENDS
            END
```

The directive STACK_SEG SEGMENT STACK declares the logical segment for the stack segment. DW 100 DUP(0) assigns actual size of the stack to 100 words. All locations of this stack are initialized to zero. The stacks are identified by the stack top and that is why the Label Top of Stack (TOS) has been selected. Please note that the stack in 8086 is a WORD stack. Stack facilities involve the use of indirect addressing through a special register, the stack pointer (SP). SP is automatically decremented as items are put on the stack and incremented as they are retrieved. Putting something on to stack is called a PUSH and taking it off is called a POP. The address of the last element pushed on to the stack is known as the top of the stack (TOS).

Name	Mnemonics	Description
Push onto the stack	PUSH SRC	SP←SP – 2 SP+1 and SP location are assign the SRC
Pop from the stack	POP DST	DST is a assigned values stored at stack top SP← SP + 2

4.3.2 Far and Near Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, that is, they require extra code to

join them together in such a way that they can communicate with each other. This extra code is sometimes referred to as linkage overhead.

A procedure call involves:

1. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.

2. The registers used by the procedures need to be stored before their contents are changed and then restored just before the procedure is finished.

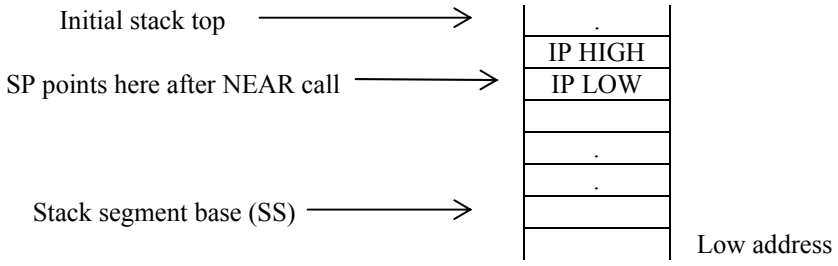
3. A procedure must have a means of communicating or sharing data with the procedures that call it, that is parameter passing.

Calls, Returns, and Procedures definitions in 8086

The 8086 microprocessor supports CALL and RET instructions for procedure call.

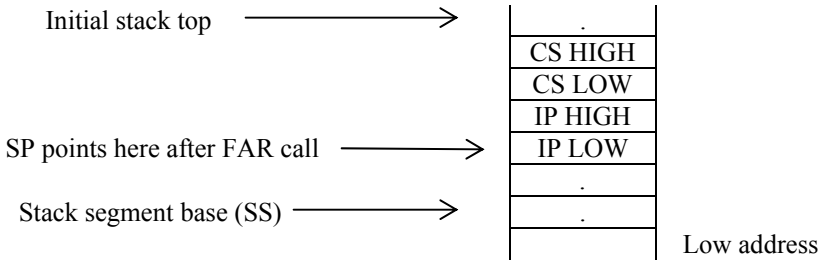
The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initialized IP with the address of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds of procedure call. These are FAR and NEAR calls.

The NEAR procedure call is also known as Intrasegment call as the called procedure is in the same segment from which call has been made. Thus, only IP is stored as the return address. The IP can be stored on the stack as:



Please note the growth of stack is towards stack segment base. So stack becomes full on an offset 0000h. Also for push operation we decrement SP by 2 as stack is a word stack (word size in 8086 = 16 bits) while memory is byte organised memory.

FAR procedure call, also known as intersegment call, is a call made to separate code segment. Thus, the control will be transferred outside the current segment. Therefore, both CS and IP need to be stored as the return address. These values on the stack after the calls look like:



When the 8086 executes the FAR call, it first stores the contents of the code segment register followed by the contents of IP on to the stack. A RET from the NEAR procedure. Pops the two bytes into IP. The RET from the FAR procedure pops four bytes from the stack.

Procedure is defined within the source code by placing a directive of the form:

<Procedure name> PROC <Attribute>

A procedure is terminated using:

<Procedure name> ENDP

The <procedure name> is the identifier used for calling the procedure and the <attribute> is either NEAR or FAR. A procedure can be defined in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case the <attribute> code NEAR should be used as the procedure and code are in the same segment. For the latter two cases the <attribute> must be FAR.

Let us describe an example of procedure call using NEAR procedure, which contains a call to a procedure in the same segment.

PROGRAM 2:

Write a program that collects in data samples from a port at 1 ms interval. The upper 4 bits collected data same as mastered and stored in an array in successive locations.

```
; REGISTERS          :Uses CS, SS, DS, AX, BX, CX, DX, SI, SP
; PROCEDURES         : Uses WAIT

DATA_SEG      SEGMENT
        PRESSURE          DW        100       DUP(0)      ; Set up array of 100 words
        NBR_OF_SAMPLES          EQU       100
        PRESSURE_PORT EQU     0FFF8h                      ; hypothetical input port
DATA_SEG      ENDS

STACK_SEG SEGMENT STACK
        DW          40                  DUP(0)            ; set stack of 40 words
        STACK_TOP   LABEL WORD
STACK_SEG   ENDS

CODE_SEG     SEGMENT
       ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:    MOV     AX, DATA_SEG                ; Initialise data segment register
          MOV     DS, AX
          MOV     AX, STACK_SEG               ; Initialise stack segment register
          MOV     SS, AX
          MOV     SP, OFFSET STACK – TOP      ; initialise stack pointer  top of
                                              ; stack
          LEA     SI, PRESSURE                ; SI points to start of array
                                              ; PRESSURE
          MOV     BX, NBR_OF_SAMPLES          ; Load BX with number
                                              ; of samples
          MOV     DX, PRESSURE_PORT           ; Point DX at input port
                                              ; it can be any A/D converter or
                                              ; data port.

READ_NEXT: IN          AX, DX                 ; Read data from port
                                              ; please note use of IN instruction
             AND       AX, 0FFFH              ; Mask upper 4 bits of AX
             MOV       [SI], AX               ; Store data word in array
             CALL      WAIT                   ; call procedures wait for delay
```

```
            INC     SI              ; Increment SI by two as dealing with
            INC     SI              ; 16 bit words and not bytes
            DEC     BX              ; Decrement sample counter
            JNZ     READ_NEXT       ; Repeat till 100
                                    ; samples are collected
STOP:       NOP
WAIT        PROC    NEAR
            MOV     CX, 2000H       ; Load delay value
                                    ; into CX
HERE:       LOOP    HERE            ; Loop until CX = 0
            RET
WAIT        ENDP
CODE_SEG    ENDS
            END
```

Discussion:

Please note that the CALL to the procedure as above does not indicate whether the call is to a NEAR procedure or a FAR procedure. This distinction is made at the time of defining the procedure.

The procedure above can also be made a FAR procedure by changing the definition of the procedure as:

```
WAIT        PROC FAR
            .
            .
WAIT        ENDS
```

The procedure can now be defined in another segment if the need so be, in the same assembly language file.

4.3.3 Parameter Passing in Procedures

Parameter passing is a very important concept in assembly language. It makes the assembly procedures more general. Parameter can be passed to and from to the main procedures. The parameters can be passed in the following ways to a procedure:

1. Parameters passing through registers
2. Parameters passing through dedicated memory location accessed by name
3. Parameters passing through pointers passed in registers
4. Parameters passing using stack.

Let us discuss a program that uses a procedure for converting a BCD number to binary number.

PROGRAM 3:

Conversion of BCD number to binary using a procedure.

Algorithm for conversion procedure:

Take a packed BCD digit and separate the two digits of BCD.
Multiply the upper digit by 10 (0Ah)
Add the lower digit to the result of multiplication

The implementation of the procedure will be dependent on the parameter-passing scheme. Let us demonstrate this with the help of three programs.

Program 3 (a): Use of registers for parameter passing: This program uses AH register for passing the parameter.

We are assuming that data is available in memory location. BCD and the result is stored in BIN

```
;REGISTERS          : Uses CS, DS, SS, SP, AX
;PROCEDURES         : BCD-BINARY

DATA_SEG            SEGMENT
        BCD         DB 25h                      ; storage for BCD value
        BIN         DB ?                        ; storage for binary value
DATA_SEG            ENDS
STACK_SEG           SEGMENT     STACK
                    DW          200 DUP(0)          ; stack of 200 words
        TOP_STACK   LABEL       WORD
STACK_SEG           ENDS

CODE_SEG            SEGMENT
        ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:  MOV         AX, DATA_SEG                ; Initialise data segment
        MOV         DS, AX                      ; Using AX register
        MOV         AX, STACK_SEG               ; Initialise stack
        MOV         SS, AX                      ; Segment register. Why
                                                ; stack?
        MOV         SP, OFFSET TOP_STACK        ; Initialise stack pointer
        MOV         AH, BCD
        CALL        BCD_BINARY                  ; Do the conversion
        MOV         BIN, AH                     ; Store the result in the
                                                ; memory
                    :
                    :
; Remaining program can be put here
;PROCEDURE          : BCD_BINARY - Converts BCD numbers to binary.
;INPUT              : AH with BCD value
;OUTPUT             : AH with binary value
;DESTROYS           : AX

BCD_BINARY    PROC NEAR

        PUSHF                       ; Save flags
        PUSH    BX                  ; and registers used in procedure
        PUSH    CX                  ; before starting the conversion
                                    ; Do the conversion
        MOV     BH, AH              ; Save copy of BCD in BH
        AND     BH, 0Fh             ; and mask the higher bits. The lower digit
                                    ; is in BH
        AND     AH, 0F0h            ; mask the lower bits. The higher digit is in AH
                                    ; but in upper 4 bits.
        MOV     CH, 04              ; so move upper BCD digit to lower
        ROR     AH, CH              ; four bits in AH
        MOV     AL, AH              ; move the digit in AL for multiplication
        MOV     BH, 0Ah             ; put 10 in BH
        MUL     BH                  ; Multiply upper BCD digit in AL
                                    ; by 0Ah in BH, the result is in AL
        MOV     AH, AL              ; the maximum/ minimum number would not
                                    ; exceed 8 bits so move AL to AH
        ADD     AH, BH              ; Add lower BCD digit to MUL result
; End of conversion, binary result in AH
                POP     CX                  ; Restore registers
                POP     BX
                POPF
```

```
                RET                     ; and return to calling program
BCD_BINARY      ENDP
CODE_SEG        ENDS
                END     START
```

Discussion:

The above program is not an optimum program, as it does not use registers minimally. By now you should be able to understand this module. The program copies the BCD number from the memory to the AH register. The AH register is used as it is in the procedure. Thus, the contents of AH register are used in calling program as well as procedure; or in other words have been passed from main to procedure. The result of the subroutine is also passed back to AH register as returned value. Thus, the calling program can find the result in AH register.

The advantage of using the registers for passing the parameters is the ease with which they can be handled. The disadvantage, however, is the limit of parameters that can be passed. For example, one cannot pass an array of 100 elements to a procedure using registers.

Passing Parameters in General Memory

The parameters can also be passed in the memory. In such a scheme, the name of the memory location is used as a parameter. The results can also be returned in the same variables. This approach has a severe limitation. It is that you will be forced to use the same memory variable with that procedure. What are the implications of this bound? Well in the example above we will be bound that variable BCD must contain the input. This procedure cannot be used for a value stored in any other location. Thus, it is a very restrictive method of procedural call.

Passing Parameters Using Pointers

This method overcomes the disadvantage of using variable names directly in the procedure. It uses registers to pass the procedure pointers to the desired data. Let us explain it further with the help of a newer version of the last program.

Program 3 (c) version 2:

```
DATA_SEG        SEGMENT
                BCD             DB          25h     ; Storage for BCD test value
                BIN             DB          ?       ; Storage for binary value
DATA_SEG        ENDS

STACK_SEG SEGMENT   STACK
                    DW      100 DUP(0)      ; Stack of 100 words
        TOP_STACK LABEL   WORD
STACK_SEG  ENDS

CODE_SEG    SEGMENT
            ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:      MOV     AX, DATA_SEG            ; Initialize data
            MOV     DS, AX                  ; segment using AX register
            MOV     AX, STACK_SEG           ; initialize stack
            MOV     SS, AX                  ; segment. Why stack?
            MOV     SP, OFFSET TOP_STACK    ; initialize stack pointer
; Put pointer to BCD storage in SI and DI prior to procedure call.
            MOV     SI, OFFSET BCD          ; SI now points to BCD_IN
            MOV     DI, OFFSET BIN          ; DI points BIN_VAL
                                            ; (returned value)
            CALL    BCD_BINARY              ; Call the conversion
```

```
                                        ; procedure
        NOP                             ; Continue with program
                                        ; here

; PROCEDURE       : BCD_BINARY Converts BCD numbers to binary.
; INPUT           : SI points to location in memory of data
; OUTPUT          : DI points to location in memory for result
; DESTROYS        : Nothing

BCD_BINARY    PROC    NEAR
        PUSHF                   ; Save flag register
        PUSH    AX              ; and AX registers
        PUSH    BX              ; BX
        PUSH    CX              ; and CX
        MOV     AL, [SI]        ; Get BCD value from memory
                                ; for conversion
        MOV     BL, AL          ; copy it in BL also
        AND     BL, 0Fh         ; and mask to get lower 4 digits
        AND     AL, 0F0h        ; Separate upper 4 bits in AL
        MOV     CL, 04          ; initialize counter CL so that upper digit
        ROR     AL, CL          ; in AL can be brought to lower 4 bit
                                ; positions in AL
        MOV     BH, 0Ah         ; Load 10 in BH
        MUL     BH              ; Multiply upper digit in AL by 10
                                ; The result is stored in AL
        ADD     AL, BL          ; Add lower BCD digit in BL to result of
                                ; multiplication
; End of conversion, now restore the original values prior to call. All calls will be in
; reverse order to save above. The result is in AL register.
        MOV     [DI], AL        ; Store binary value to memory
        POP     CX              ; Restore flags and
        POP     BX              ; registers
        POP     AX
        POPF
        RET
BCD_BINARY      ENDP
CODE_SEG        ENDS
        END     START
```

Discussion:

In the program above, SI points to the BCD and the DI points to the BIN. The instruction MOV AL,[SI] copies the byte pointed by SI to the AL register. Likewise, MOV [DI], AL transfers the result back to memory location pointed by DI.

This scheme allows you to pass the procedure pointers to data anywhere in memory. You can pass pointer to individual data element or a group of data elements like arrays and strings. This approach is used for parameters passing to BIOS procedures.

Passing Parameters Through Stack

The best technique for parameter passing is through stack. It is also a standard technique for passing parameters when the assembly language is interfaced with any high level language. Parameters are pushed on the stack and are referenced using BP register in the called procedure. One important issue for parameter passing through stack is to keep track of the stack overflow and underflow to keep a check on errors. Let us revisit the same example, but using stack for parameter passing.

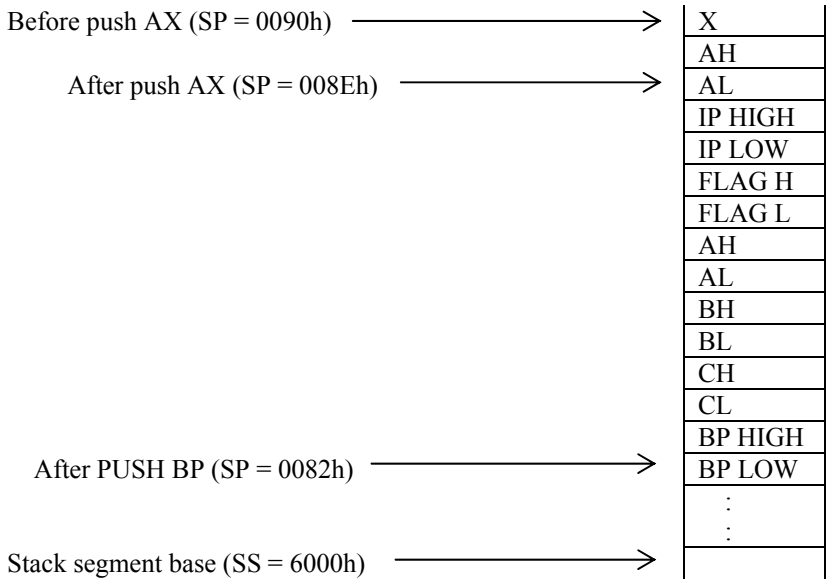
PROGRAM 3: Version 3

```
DATA_SEG    SEGMENT
            BCD         DB      25h     ; Storage for BCD test value
            BIN         DB      ?       ; Storage for binary value
DATA_SEG    ENDS

STACK_SEG   SEGMENT     STACK
            DW          100 DUP(0)      ; Stack of 100 words
            TOP_STACK   LABEL   WORD
STACK_SEG   ENDS

CODE_SEG    SEGMENT
        ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:      MOV     AX, DATA                ; Initialise data segment
            MOV     DS, AX                  ; using AX register
            MOV     AX, STACK-SEG .         ; initialise stack segment
            MOV     SS, AX                  ; using AX register
            MOV     SP, OFFSET TOP_STACK    ; initialise stack pointer
            MOV     AL, BCD                 ; Move BCD value into AL
            PUSH    AX                      ; and push it onto word stack
            CALL    BCD_BINARY              ; Do the conversion
            POP     AX                      ; Get the binary value
            MOV     BIN, AL                 ; and save it
            NOP                             ; Continue with program
; PROCEDURE         : BCD_BINARY Converts BCD numbers to binary.
; INPUT             : None - BCD value assumed to be on stack before call
; OUTPUT            : None - Binary value on top of stack after return
; DESTROYS          : Nothing

BCD_BINARY  PROC    NEAR
            PUSHF                           ; Save flags
            PUSH    AX                      ; and registers : AX
            PUSH    BX                      ; BX
            PUSH    CX                      ; CX
            PUSH    BP                      ; BP. Why BP?
            MOV     BP, SP                  ; Make a copy of the
                                            ; stack pointer in BP
            MOV     AX, [BP+ 12]            ; Get BCD number from
                                            ; stack. But why it is on
; BP+12 location? Please note 5 PUSH statements + 1 call which is intra-segment (so
; just IP is stored) so total 6 words are pushed after AX has been pushed and since it is
; a word stack so the BCD value is stored on 6 × 2 = 12 locations under stack. Hence
; [BP + 12] (refer to the figure given on next page).
            MOV     BL, AL          ; Save copy of BCD in BL
            AND     BL, 0Fh         ; mask lower 4 bits
            AND     AL, F0H         ; Separate upper 4 bits
            MOV     CL, 04          ; Move upper BCD digit to low
            ROR     AL, CL          ; position BCD digit for multiply location
            MOV     BH, 0Ah           ; Load 10 in BH
            MUL     BH                ; Multiply upper BCD digit in AL by 10
                                      ; the result is in AL
            ADD     AL, BL            ; Add lower BCD digit to result.
            MOV     [BP + 12], AX     ; Put binary result on stack
            ; Restore flags and registers
            POP     BP
            POP     CX
            POP     BX
            POP     AX
```

```
                POPF
                RET
BCD_BINARY              ENDP
CODE_SEG                ENDS
END         START
```

Discussion:

The parameter is pushed on the stack before the procedure call. The procedure call causes the current instruction pointer to be pushed on to the stack. In the procedure flags, AX, BX, CX and BP registers are also pushed in that order. Thus, the stack looks to be:



The instruction MOV BP, SP transfers the contents of the SP to the BP register. Now BP is used to access any location in the stack, by adding appropriate offset to it. For example, MOV AX, [BP + 12] instruction transfers the word beginning at the 12th byte from the top of the stack to AX register. It does not change the contents of the BP register or the top of the stack. It copies the pushed value of AH and AL at offset 008Eh into the AX register. This instruction is not equivalent to POP instruction.

Stacks are useful for writing procedures for multi-user system programs or recurvise procedures. It is a good practice to make a stack diagram as above while using procedure call through stacks. This helps in reducing errors in programming.

4.3.4 External Procedures

These procedures are written and assembled in separate assembly modules, and later are linked together with the main program to form a bigger module. Since the addresses of the variables are defined in another module, we need segment combination and global identifier directives. Let us discuss them briefly.

Segment Combinations

In 8086 assembler provides a means for combining the segments declared in different modules. Some typical combine types are:

1. PUBLIC: This combine directive combines all the segments having the same names and class (in different modules) as a single combined segment.
2. COMMON: If the segments in different object modules have the same name and the COMMON combine type then they have the same beginning address. During execution these segments overlay each other.

3. STACK: If the segments in different object modules have the same name and the combine type is STACK, then they become one segment, with the length the sum of the lengths of individual segments.

These details will be more clear after you go through program 4 and further readings.

Identifiers

a) **Access to External Identifiers:** An external identifier is one that is referred in one module but defined in another. You can declare an identifier to be external by including it on as EXTRN in the modules in which it is to be referred. This tells the assembler to leave the address of the variable unresolved. The linker looks for the address of this variable in the module where it is defined to be PUBLIC.

b) **Public Identifiers:** A public identifier is one that is defined within one module of a program but potentially accessible by all of the other modules in that program. You can declare an identifier to be public by including it on a PUBLIC directive in the module in which it is defined.

Let us explain all the above with the help of the following example:

PROGRAM 4:

Write a procedure that divides a 32-bit number by a 16-bit number. The procedure should be general, that is, it is defined in one module, and can be called from another assembly module.

```
; REGISTERS             :Uses CS, DS, SS, AX, SP, BX, CX
; PROCEDURES            : Far Procedure SMART_DIV
DATA_SEG    SEGMENT     WORD      PUBLIC
            DIVIDEND    DW        2345h, 89AB     ; Dividend =
                                                  ; 89AB2345H
            DIVISOR     DW        5678H           ; 16-bit divisor
            MESSAGE     DB        'INVALID DIVIDE', '$'
DATA_SEG    ENDS

MORE_DATA   SEGMENT     WORD
            QUOTIENT    DW        2  DUP(0)
            REMAINDER   DW        0
MORE_DATA ENDS

STACK_SEG   SEGMENT     STACK
            DW          100 DUP(0)            ; Stack of 100 words
            TOP – STACK LABEL     WORD        ; top of stack pointer
STACK_SEG   ENDS

PUBLIC      DIVISOR

PROCEDURES  SEGMENT     PUBLIC                ; SMART_DIVis declared as an
        EXTRN SMART_DIV: FAR                  ; external label in procedure
                                              ; segment of  type FAR
PROCEDURES  ENDS
; declare the code segment as PUBLIC so that it can be merged with other PUBLIC
; segments
CODE_SEG    SEGMENT     WORD      PUBLIC
        ASSUME CS:CODE, DS:DATA_SEG, SS:STACK SEG
START:      MOV     AX, DATA_SEG              ; Initialize data segment
            MOV     DS, AX                    ; using AX register
            MOV     AX, STACK_SEG             ; Initialize stack segment
```

```
            MOV     SS, AX                      ; using AX register
            MOV     SP, OFFSET TOP_STACK        ; Initialize stack pointer
            MOV     AX, DIVIDEND                ; Load low word of
                                                ; dividend
            MOV     DX DIVIDEND + 2             ; Load high word of
                                                ; dividend
            MOV     CX, DIVISOR                 ; Load divisor
            CALL SMART_DIV
; This procedure returns Quotient in the DX:AX pair and Remainder in CX register.
; Carry bit is set if result is invalid.
            JNC     SAVE_ALL                    ; IF carry = 0, result valid
            JMP     STOP                        ; ELSE carry set, don't
                                                ; save result
            ASSUME DS:MORE_DATA                 ; Change data segment
SAVE_ALL:   PUSH DS                             ; Save old DS
            MOV     BX, MORE_DATA               ; Load new data segment
            MOV     DS, BX                      ; register
            MOV     QUOTIENT, AX                ; Store low word of
                                                ; quotient
            MOV     QUOTIENT + 2, DX            ; Store high word of
                                                ; quotient
            MOV     REMAINDER, CX               ; Store remainder
            ASSUME DS:DATA_SEG
            POP     DS                          ; Restore initial DS
            JMP     ENDING
STOP:       MOV     DL, OFFSET MESSAGE
            MOV     AX, AH 09H
            INT     21H
ENDING:     NOP
CODE_SEG    ENDS
            END     START
```

Discussion:

The linker appends all the segments having the same name and PUBLIC directive with segment name into one segment. Their contents are pulled together into consecutive memory locations.

The next statement to be noted is PUBLIC DIVISOR. It tells the assembler and the linker that this variable can be legally accessed by other assembly modules. On the other hand EXTRN SMART_DIV:FAR tells the assembler that this module will access a label or a procedure of type FAR in some assembly module. Please also note that the EXTRN definition is enclosed within the PROCEDURES SEGMENT PUBLIC and PROCEDURES ENDS, to tell the assembler and linker that the procedure SMART_DIV is located within the segment PROCEDURES and all such PROCEDURES segments need to be combined in one. Let us now define the PROCEDURE module:

```
; PROGRAM    MODULE       PROCEDURES

; INPUT      : Dividend - low word in AX, high word in DX, Divisor in CX
; OUTPUT     : Quotient  - low word in AX, high word in DX. Remainder in CX
             ; Carry     - carry flag is set if try to divide by zero
; DESTROYS   : AX, BX, CX, DX, BP, FLAGS
DATA_SEG     SEGMENT      PUBLIC      ; This block tells the assembler that
EXTRN DIVISOR:WORD                    ; the divisor is a word variable and is
DATA_SEG     ENDS                     ; external to this procedure. It would be
                                      ; found in segment named DATA_SEG
PUBLIC       SMART_DIV                ; SMART_DIV is available to
                                      ; other modules. It is now being defined
```

```
                                        ; in PROCEDURES SEGMENT.
PROCEDURES      SEGMENT   PUBLIC
SMART_DIV       PROC      FAR
        ASSUME CS:PROCEDURES, DS:DATA_SEG
        CMP     DIVISOR, 0              ; This is just to demonstrate the use of
                                        ; external variable, otherwise  we can
                                        ; check it through CX register which
                                        ; contains the divisor.
        JE      ERROR_EXIT              ; IF divisor = 0, exit procedure
        MOV     BX, AX                  ; Save low order of dividend
        MOV     AX, DX                  ; Position high word for lst divide
        MOV     DX, 0000h               ; Zero DX
        DIV     CX                      ; DX:AX/CX, quotient in AX,
                                        ; remainder in DX
        MOV     BP, AX                  ; transfer high order of final result to BP
        MOV     AX, BX                  ; Get back low order of dividend. Note
                                        ; DX contains remainder so DX : AX is
                                        ; the actual number
        DIV     CX                      ; DX:AX/CX, quotient in AX,
                                        ; 2nd remainder that is final remainder
                                        ; in DX
        MOV     CX, DX                  ; Pass final remainder in CX
        MOV     DX, BP                  ; Pass high order of quotient in DX
                                        ; AX contains lower word of quotient
        CLC                             ; Clear carry to indicate valid result
        JMP     EXIT                    ; Finished
ERROR_EXIT: STC                         ; Set carry to indicate divide by zero
EXIT:   RET
SMART_DIV       ENDP
PROCEDURES      ENDS
                END
```

Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the main, therefore the statement EXTRN DIVISOR:WORD is necessary for informing assembler that this data name is found in some other segment. The data type is defined to be of word type. Please not that the DIVISOR is enclosed in the same segment name as that of main that is DATA_SEG and the procedure is in a PUBLIC segment.

☞ Check Your Progress 2

T	F

1. State True or False

(a) A NEAR procedure can be called only in the segment it is defined. ☐

(b) A FAR call uses one word in the stack for storing the return address. ☐

(c) While making a call to a procedure, the nature of procedure that is NEAR or FAR must be specified. ☐

(d) Parameter passing through register is not suitable when large numbers of parameters are to be passed. ☐

(e) Parameter passing in general memory is a flexible way of passing parameters. ☐

(f) Parameter passing through pointers can be used to pass a group of data elements. ☐

(f) Parameter passing through stack is used whenever assembly language programs are interfaced with any high level language programs. ☐

(h) In multiuser systems parameters should be passed using pointers. ☐

(i) A variable say USAGE is declared in a PROCEDURE segment, however it is used in a separate module. In such a case the declaration of USAGE should contain EXTRN verb. ☐

(i) A segment if declared PUBLIC informs the linker to append all the segments with same name into one. ☐

2. Show the stack if the following statements are encountered in sequence.

   a) Call to a NEAR procedure FIRST at 20A2h:0050h
   b) Call to a FAR procedure SECOND at location 3000h:5055h
   c) RETURN from Procedure FIRST.

4.4 INTERFACING ASSEMBLY LANGUAGE ROUTINES TO HIGH LEVEL LANGUAGE PROGRAMS

By now you can write procedures, both external and internal, and pass parameters, especially through stack, let us use these concepts, to see how assembly language can be interfaced to some high level language programs. It is very important to learn this concept, because then you can combine the advantages of both the types of languages, that is, the ease of programming of high level languages and the speed and the scope of assembly language. Assembly language can be interfaced with most of the high level languages like C, C + + and database management systems.

What are the main considerations for interfacing assembly to HLL? To answer that we need to answer the following questions:

- How is the subroutine invoked?
- How are parameters passed?
- How are the values returned?
- How do you declare various segments so that they are consistent across calling program?

The answer to the above questions are dependent on the high level language (HLL). Let us take C Language as the language for interfacing. The C Language is very useful for writing user interface programs, but the code produced by a C compiler does not execute fast enough for telecommunications or graphics applications. Therefore, system programs are often written with a combination of C and assembly language functions. The main user interface may be written in C and specialized high speed functions written in assembly language.

The guidelines for calling assembly subroutines from C are:

(i) Memory model: The calling program and called assembly programs must be defined with the same memory model. One of the most common convention that makes NEAR calls is .MODEL SMALL, C

(ii) The naming convention normally involve an underscore (_) character preceding the segment or function name. Please note, however, this underscore is not used while making a call from C function. Please be careful about case sensitivity.

You must give a specific segment name to the code segment of your assembly language subroutine. The name varies from compiler to compiler. Microsoft C, and Turbo C require the code segment name to be_TEXT or a segment name with suffix_TEXT. Also, it requires the segment name _DATA for the data segment.

(iii) The arguments from C to the assembly language are passed through the stack. For example, a function call in C:

```
function_name (arg1, arg2, ..., argn) ;
```

would push the value of each argument on the stack in the reverse order. That is, the argument *argn* is pushed first and *arg1* is pushed last on the stack. A value or a pointer to a variable can also be passed on the stack. Since the stack in 8086 is a word stack, therefore, values and pointers are stored as words on stack or multiples of the word size in case the value exceeds 16 bits.

(iv) You should remember to save any special purpose registers (such as CS, DS, SS, ES, BP, SI or DI) that may be modified by the assembly language routine. If you fail to save them, then you may have undesirable/ unexplainable consequences, when control is returned to the C program. However, there is no need to save AX, BX, CX or DX registers as they are considered volatile.

(v) Please note the compatibility of data types:

char	Byte (DB)
int	Word (DW)
long	Double Word (DD)

(vi) Returned value: The called assembly routine uses the followed registers for returned values:

char	AL
Near/ int	AX
Far/ long	DX : AX

Let us now look into some of the examples for interfacing.

4.4.1 Simple Interfacing

The following is a sample of the coding, used for procedure interfacing:

```
PUBLIC CUROFF
        _TEXT SEGMENT WORD PUBLIC 'CODE'
                ASSUME CS:_TEXT
                _CUROFF PROC NEAR     ; for small model
        :
        :
```

The same thing can be written using the newer simplified directives in the following manner:

```
PUBLIC CUROFF
.MODEL small,C
.CODE
CUROFF PROC
        :
        :
```

This second source code is much cleaner and easier to read. The directives .MODEL and .CODE instruct the assembler to make the necessary assumptions and adjustments so that the routine will work with a small model of C program. (Please

refer to Assembler manuals on details on models of C program. The models primarily differ in number of segments).

PROGRAM 5:

Write an assembly function that hides the cursor. Call it from a C program.

```
            . PUBLIC CUROFF
            . MODEL small,C
            . CODE
CUROFF  PROC
      MOV     AH,3                ; get the current cursor position
      XOR     BX,BX               ; empty BX register
      INT     10h                 ; use int 10hto do above
      OR      CH,20h              ; force to OFF condition
      MOV     AH,01               ; set the new cursor values
      INT     10h
      RET
CUROFF ENDP
            END
```

For details on various interrupt functions used in this program refer to further readings.

The C program to test this routine is as follows:

```
        # include < stdio.h
        void curoff(void);
        void main()

{

      printf("%s\n, "The cursor is now turning off);
      curoff();

}
```

You can write another procedure in assembly language program to put the cursor on. This can be done by replacing OR CH,20h instruction by AND CH,1Fh. You can call this new function from C program to put the cursor on after the curoff.

4.4.2 Interfacing Subroutines With Parameter Passing

Let us now write a C program that calls the assembly program for parameter passing. Let us extend the previous two programs such that if on function call 0 is passed then cursor is turned off and if 1 is passed then cursor is turned on. The calling C program for such may look like:

```
        # include < stdio.h
        void cursw(int);
        void main()
        {
                printf("%s\n", "the cursor is now turning off");
                cursw(0); /* call to assembly routine with 0 as parameter
                getchar();
                printf("%s\n","the cursor is now turning on");
                cursw(l); /* call to assembly routine with parameter as1.
        }
```

The variables in C or C + + are passed on the stack.

Let us now write the assembly routine:

PROGRAM 6:

Write a subroutine in C for toggling the cursor using old directives.

```
;
; use small memory model for C – near code segment

_DATA       SEGMENT WORD    'DATA'
   CURVAL   EQU             [BP+4]          ; parameters
_DATA       ENDS

_TEXT       SEGMENT         BYTE  PUBLIC        'CODE'
DGROUP      GROUP           _DATA
            ASSUME          CS:_TEXT,     DS:DGROUP,  SS:DGROUP
            PUBLIC          _CURSW
_CURSW      PROC            NEAR
            PUSH            BP              ; BP register of caller is saved
            MOV             BP, SP          ; BP is pointing to stack now
            MOV             AX, CURVAL
            CMP             AX, 0H
            JZ              CUROFF          ; Execute code for cursor off
            CMP             AX, 01H
            JZ              CURON           ; Execute code for cursor on
            JMP             OVER            ; Error in parameter, do nothing
CUROFF:                                     ; write code for curoff
        :
        :
            JMP             OVER
CURON:                                      ; write code for curon
        :
        :
OVER:       POP     BP
            RET
_CURSW      ENDP
_TEXT       ENDS
            END
```

Why the parameter is found in [BP+4]? Please look into the following stack for the answer.

Parameter (0 or 1)	BP + 4
Return Address	BP + 2
Old value	BP + 0

PROGRAM 7:

Write a subroutine in C that toggles the cursor. It takes one argument that toggles the value between on (1) and off (0) using simplified directives:

```
            PUBLIC CURSW
            .MODEL small, C
            .CODE
CURSW PROC switch:word

      MOV       AX,SWITCH           ; get flag value
      XOR       AX,AX               ; test zero / nonzero
                :
                :
          // routine to test the switch and accordingly
```

```
        switch off or switch on the cursor //
                        :
                        :
        CURSW ENDP
                        END
```

In a similar manner the variables can be passed in C as pointers also. Values can be returned to C either by changing the variable values in the C data segment or by returning the value in the registers as given earlier.

4.5 INTERRUPTS

Interrupts are signals that cause the central processing unit to suspend the currently executing program and transfer to a special program called an interrupt handler. The interrupt handler determines the cause of the interrupt, services the interrupt, and finally returns the control to the point of interrupt. Interrupts are caused by events external or internal to the CPU that require immediate attention. Some external events that cause interrupts are:

- Completion of an I/O process
- Detection of a hardware failure

An 8086 interrupt can occur because of the following reasons:

1. Hardware interrupts, caused by some external hardware device.
2. Software interrupts, which can be invoked with the help of INT instruction.
3. Conditional interrupts, which are mainly caused due to some error condition generated in 8086 by the execution of an instruction.

When an interrupt can be serviced by a procedure, it is called as the Interrupt Service Routine (ISR). The *starting addresses* of the interrupt service routines are present in the first 1K addresses of the memory (Please refer to Unit 2 of this block). This table is called the interrupt vector table.

How can we write an Interrupt Servicing Routine? The following are the basic but rigid sequence of steps:

1. Save the system context (registers, flags etc. that will be modified by the ISR).
2. Disable the interrupts that may cause interference if allowed to occur during this ISR's processing
3. Enable those interrupts that may still be allowed to occur during this ISR processing.
4. Determine the cause of the interrupt.
5. Take the appropriate action for the interrupt such as – receive and store data from the serial port, set a flag to indicate the completion of the disk sector transfer, etc.
6. Restore the system context.
7. Re-enable any interrupt levels that were blocked during this ISR execution.
8. Resume the execution of the process that was interrupted on occurrence of the interrupt.

MS-DOS provides you facilities that enable you to install well-behaved interrupt handlers such that they will not interfere with the operating system function or other interrupt handlers. These functions are:

Function	Action
Int 21h function 25h	Set interrupt vector
Int 21h function 35h	Get interrupt vector
Int 21h function 31h	Terminate and stay residents

Here are a few rules that must be kept in mind while writing down your own Interrupt Service Routines:

1. Use Int 21h, function 35h to get the required IVT entry from the IVT. Save this entry, for later use.
2. Use Int 21h, function 25h to modify the IVT.
3. If your program is not going to stay resident, save the contents of the IVT, and later restore them when your program exits.
4. If your program is going to stay resident, use one of the terminate and stay resident functions, to reserve proper amount of memory for your handler.

Let us now write an interrupt routine to handle "division by zero". This file can be loaded like a COM file, but makes itself permanently resident, till the system is running.

This ISR is divided into two major sections: the initialisation and the interrupt handler. The initialisation procedure (INIT) is executed only once, when the program is executed from the DOS level. INIT takes over the type zero interrupt vector, it also prints a sign-on message, and then performs a terminate and "stay resident exit" to MS-DOS. This special exit reserves the memory occupied by the program, so that it is not overwritten by subsequent application programs. The interrupt handler (ZDIV) receives control when a divide-by-zero interrupt occurs.

```
CR          EQU     ODH         ; ASCII carriage return
LF          EQU     0Ah         ; ASCII line feed
BEEP        EQU     07h         ; ASCII beep code
BACKSP      EQU     08h         ; ASCII backspace code

CSEG SEGMENT PARA PUBLIC  'CODE'
        ORG 100h
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG

INIT PROC NEAR
        MOV     DX,OFFSET ZDIV      ; reset interrupt 0 vector
                                    ; to address of new
                                    ; handler using function 25h, interrupt
        MOV     AX, 2500h           ; 0 handles divide-by-zero
        INT     21h
        MOV     AH,09               ; print identification message
        INT     21h
                                    ; DX assigns paragraphs of memory
                                    ; to reserve
        MOV     DX,((OFFSET PGM_LEN + 15)/16) + 10h
        MOV     AX,3100h            ; exit and stay resident
        INT     21h                 ; with a return code = 0
INIT    ENDP

ZDIV PROC FAR                       ; this is the zero-divide
                                    ; hardware interrupt handler.
        STI                         ; enable interrupts.
        PUSH    AX                  ; save general registers
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
```

```
        MOV     AX,CS
        MOV     DS,AX
        MOV     DX,OFFSET WARN          ; Print warning "divide by
        MOV     AH, 9                   ; zero "and" continue or
        INT     21h                     ; quit?"

ZDIV1:  MOV     AH,1                    ; read keyboard
        INT     21h
        CMP     AL, 'C'                 ; is it 'C' or 'Q'?
        JE      ZDIV3                   ; jump it is a 'C'.
        CMP     AL, 'Q'

        JE      ZDIV2                   ; jump it's a'Q'
        MOV     DX, OFFSET BAD          ; illegal entry, send a
        MOV     AH,9                    ; beep, erase the bad char
        INT     21h                     ; and try again
        JMP     ZDIV1

ZDIV2:  MOV     AX, 4CFFh               ; user wants to abort the
        INT     21h                     ; program, return with
                                        ; return code = 255
ZDIV3:  MOV     DX,OFFSET CRLF          ; user wishes to continue
        MOV     AH,9                    ; send CRLF
        INT     21h
        POP     ES                      ; restore general registers
        POP     DS                      ; and resume execution
        POP     BP
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        IRET
 ZDIV   ENDP

SIGNON          DB              CR, LF, 'Divide by zero interrupt'
                DB              'Handler Installed'
                DB              CRLF,'$'
WARN            DB              CR, LF, 'Divide by zero detected:'
                DB              CR, LF 'Quit or Continue (C/Q) ?'
                DB              '$'
BAD             DB              BEEP, BACKSP, ",BACKSP,'$'
CRLF            DB              CR,LF,$'
PGM_LEN         EQU $-INIT
CSEG    ENDS
                END
```

4.6 DEVICE DRIVERS IN ASSEMBLY

Device drivers are special programs installed by the config.sys file to control installable devices. Thus, personal computers can be expanded at some future time by the installation of new devices.

The device driver is .com file organized in 3 parts.

1) The leader
2) The strategy procedure

3) The interrupt procedure

The driver has either .sys or .exe extension and is originated at offset address 0000h.

The Header

The header contains information that allows DOS to identify the driver. It also contains pointers that allow it to chain to other drivers loaded into the system.

The header section of a device driver is 18 bytes in length and contains pointers and the name of the driver.

Following structure of the header:

```
CHAIN DD -1          : link to next driver
ATTR DW 0            : driver attribute
STRT DW START        : address of strategy
INTER DW INT         : address if interrupt
DNAME DB 'MYDRIVER'  : driver name.
```

The first double word contains a –1 that informs DOS this is the last driver in the chain. If additional drivers are added DOS inserts a chain address in this double word as the segment and offset address. The chain address points to the next driver in the chain. This allows additional drivers installed at any time.

The attribute word indicates the type of headers included for the driver and the type of device the driver installs. It also indicates whether the driver control a character driver or a block device.

The Strategy Procedure

The strategy procedure is called when loaded into memory by DOS or whenever the controlled device request service. The main purpose of the strategy is to save the request header and its address for use by the interrupt procedure.

The request header is used by DOS to communicate commands and other informations to the interrupt procedure in the device driver

The request header contains the length of the request header as its first byte. This is necessary because the length of the request header varies from command to command. The return status word communicate information back to DOS from the device driver.

The initialise driver command (00H) is always executed when DOS initialises the device driver. The initialisation commands pass message to the video display indicating that the driver is loaded into the system and returns to DOS the amount of memory needed by the driver. You may only use DOS INT 21H functions 00H. You can get more details on strategy from the further readings.

The Interrupt Procedure

The interrupt procedure uses the request header to determine the function requested by DOS. It also performs all functions for the device driver. The interrupt procedures must respond to at least the initialised driver command (00H) and any other commands required to control the device operated by the device driver. You must refer to the further readings for more details and examples of device drivers.

☞ Check Your Progress 3

State True or False

	T	F
(a) Assembly language routines cannot be interfaced with BASIC programs.	☐	
(b) The key issue in interfacing is the selection of proper parameter passing method.	☐	
(c) The value of arguments to be passed are pushed in the stack in reverse order.	☐	
(d) AX, BX, CX or DX registers need not be saved in interfacing of assembly programs with high level language programs.	☐	
(e) Hardware interrupts can be invoked with the help of INT function.	☐	

2. What are the sequences of steps in an interrupt service routine?

……

……

……

4.7 SUMMARY

In the above module, we studied some programming techniques, starting from arrays, to interrupts.

Arrays can be of byte type or word type, but the addressing of the arrays is always done with respect to bytes. For a word array, the address will be incremented by two for the next access.

As the programs become larger and larger, it becomes necessary to divide them into smaller modules called procedures. The procedures can be NEAR or FAR depending upon where they are being defined and from where they are being called. The parameters to the procedures can be passed through registers, or through memory or stack. Passing parameters in registers is easier, but limits the total number of variables that can be passed. In memory locations it is straight forward, but limits the use of the procedure. Passing parameters through stack is most complex out of all, but is a standard way to do it. Even when the assembly language programs are interfaced to high level languages, the parameters are passed on stack.

Interrupt Service Routines are used to service the interrupts that could have arisen because of some exceptional condition. The interrupt service routines can be modified- by rewriting them, and overwriting their entry in the interrupt vector table.

This completes the discussion on microprocessors and assembly language programming. The above programming was done for 8086 microprocessor, but can be tried on 80286 or 80386 processors also, with some modification in the assembler directives. The assembler used here is MASM, Microsoft assembler. The assembly language instructions remain the same for all assemblers, though the directives vary from one assembler to another. For further details on the assembler, you can refer to their respective manuals. You must refer to further readings for topics such as Interrupts, device drivers, procedures etc.

4.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. We will give you an algorithm using XLAT instruction. Please code and run the program yourself.

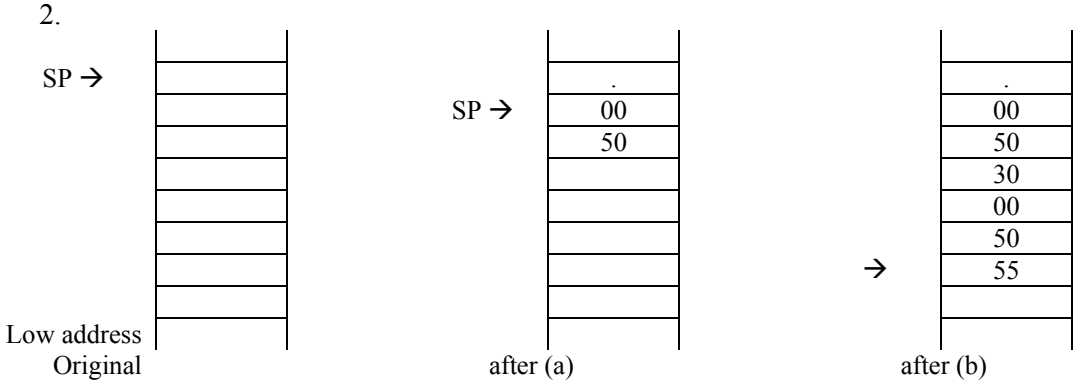
   - Take a sentence in upper case for example 'TO BE CONVERTED TO LOWER CASE' create a table for lower case elements.
   - Check that the present ASCII character is an alphabet in upper case. It implies that ASCII character should be greater than 40h and less than 58h.
   - If it is upper case then subtract 41h from the ASCII value of the character. Put the resultant in AL register.
   - Set BX register to the offset of lower case table.
   - Use XLAT instruction to get the required lower case value.
   - Store the results in another string.

2. (a) False (b) False (c) True

Check Your Progress 2

1. (a) True (b) False (c) False (d) True (e) False (f) True (g) True (h) False (i) False (j) True.

2.



(c) The return for FIRST can occur only after return of SECOND. Therefore, the stack will be back in original state.

Check Your Progress 3

1. (a) False (b) False (c) True (d) True (e) False

2.
   - Save the system context
   - Block any interrupt, which may cause interference
   - Enable allowable interrupts
   - Determine the cause of interrupt
   - Take appropriate action
   - Restore system context
   - Enable interrupts which were blocked in Step 2