Indira Gandhi
National Open University
School of Computer and
Information Sciences

MCS-202
Computer Organisation

Block

4

Microprocessor and Advanced Architectures

FACULTY OF THE SCHOOL

Prof P. V. Suresh, Director
Dr Akshay Kumar
Dr Sudhansh Sharma
Prof. V. V. Subrahmanyam
Mr Mangala Prasad Mishra

PROGRAMME/COURSE DESIGN COMMITTEE

Shri Sanjeev Thakur
Amity School of Computer Sciences,
Noida Shri Amrit Nath Thulal
Amity School of Engineering and Technology
New Delhi
Dr. Om Vikas(Retd),
Ministry of ICT, Delhi
Shri Vishwakarma
Amity School of Engineering and
Technology New Delhi
Prof (Retd) S. K. Gupta, IIT Delhi
Prof. T.V. Vijay Kumar, SC&SS, JNU,
New Delhi
Prof. Ela Kumar, CSE, IGDTUW, Delhi
Prof. Gayatri Dhingra, GVMITM, Sonipat
Sh. Milind Mahajani
Impressico Business Solutions, Noida, UP
Prof. V. V. Subrahmanyam
SOCIS, New Delhi
Prof. P. V. Suresh
SOCIS, IGNOU, New Delhi
Dr. Shashi Bhushan
SOCIS, IGNOU, New Delhi
Shri Akshay Kumar,
SOCIS, IGNOU, New Delhi
Shri M. P. Mishra,
SOCIS, IGNOU, New Delhi
Dr. Sudhansh Sharma,
SOCIS, IGNOU, New Delhi

BLOCK PREPARATION TEAM

Dr Zahid Raja *(Content Editor)*
Jawaharlal Nehru University
New Delhi

Dr Akshay Kumar *(Course Writer – Unit 13)*
SOCIS, IGNOU
Delhi

Dr Gaurav Verma (*Course Writer - Unit 16)*
Electronics and Communication Engineering,
NIT, Kurukshetra

Unit 14 and Unit 15 has been
adopted from MCS012: Block 4
Unit 2, Unit 3 and Unit 4.

(Language Editor) School of Humanities
IGNOU

Course Coordinator: Mr Akshay Kumar

PRINT PRODUCTION

August, 2021

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110068.

Printed at :

BLOCK 4 INTRODUCTION

In the previous blocks, we have discussed about computer organizations, the number systems, memory and input output organization, instruction set of a computer, addressing modes, the micro-operations and the control unit of a computer system.

This block presents an example micro-processor, as an example of computer architecture. We will discuss the microprocessor architecture and its programming. Our main emphasis would be on Intel 8086/ 8088 microprocessor. The newer microprocessors may use the concepts covered for 8086 microprocessors.

This block is divided into four units. We start with the introduction to microprocessors, with special emphasis on 8086 microprocessors. Unit 1 also gives a brief introduction to the Instruction Set and the addressing modes of the 8086 micro-processor. Taking this as the base, in Unit 2 we get on to the Introduction of Assembly Language Programming. In this unit, we will also give a brief account of various tools required to develop and execute an Assembly Language Program. In Unit 3, a detailed study of Assembly Language, its programming techniques along with several examples have been taken up. Unit 4 presents a brief discussion on some of the advanced architectures.

This block gives you only some details about 8086 microprocessor and assembly language programming. For complete details on this Intel series of microprocessors, you may refer to the further readings, given below. You may also study the advanced architectures from the further readings given in Block 1.

FURTHER READINGS FOR THE BLOCK

1. IBM PC Assembly Language and Programming, Fifth Edition, Peter Abel.
2. Douglas V. Hall: Microprocessors and Interfacing – Programming and Hardware by – McGraw Hill – 1986.
3. Peter Norton & John Socha: Assembly Language book for IBM PC-Prentice Hall of India, 1989.
4. Yu-Cheng Liu, A. Gibson: Micro-computer Systems: The 8086/ 8088 family – Prentice Hall of India, 1986.
5. Douglas V. Hall; Microprocessors and Digital Systems 2/e, McGraw Hill 1986.
6. William B. Giles, Assembly Language Programming for the Intel 80xxx family, Maxwell Macmillan International editions 1991.

UNIT 13 MICROPROCESSOR ARCHITECTURE

Structure **Page No.**

13.0 INTRODUCTION

In the previous three blocks of this course, you have gone through the concept of data representation, logic circuits, memory and I/O organisation, instruction set architecture, micro-operations, control unit etc. of a computer system. The processor of a general purpose computer consists of an instruction set, which uses a set of addressing modes. The control unit of a processor uses a set of registers and arithmetic logic unit to process these instructions. This unit present details of a micro-processor, in the content of all the above concepts. We have selected a simple micro-processors 8086, for the discussion. Although the processor technology is old, all the concepts are valid for higher end Intel processor family, which are commonly referred to as x86 family. This block does not attempt to make you an expert assembly programmer, however, you will be able to write reasonably good assembly programs . This unit discusses the 8086 microprocessor in some detail. This unit will introduce you to block diagram of components of 8086 microprocessor. This is followed by discussion on the register organization for this processor. Some useful instructions and addressing modes of this processor are also discussed in this unit. Please note the concepts discussed in this unit may be useful in writing good Assembly Programs.

13.1 OBJECTIVES

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

- explain the role of various components of 8086 microprocessor;
- illustrate the use of segmentation in 8086 microprocessor;
- use some of the important instruction of 8086 microprocessor
- illustrate the use of different types of addressing modes of 8086 microprocessor.

13.2 STRUCTURE OF 8086 MICROPROCESSOR

A microprocessor contains one or more processing unit on a single chip. Today's processors contain multiple processing cores in a single chip, therefore are called multi-core processors. A computer system consists of a micro-processor, memory unit and input/output interfaces, internal and external connection structure, such as buses; and several input/output devices. The bus size of a processor is a very important design parameter. For example, the address bus of a processor, generally, can determine the size of the physical main memory. The data bus determines the size of the data that can be transferred from the memory to the processor registers.

The size of address bus of 8086 micro-processor is 20 bits and data bus is 16 bits. Thus, 8086 micro-processor has 2^{20} = 1M Byte base memory. From this memory, about 640 KB was part of base RAM and remaining was used as ROM.

8086 micro-processor was designed as a complex instruction set computer with the basic objectives of supporting more instructions, addressing modes and more throughput. Present day multi-core processors are far more powerful than 8086 micro-processor, but objective of this block is to introduce some of the basic features of micro-processor and assembly language programming. For the basic discussion this processor is good example.

A microprocessor executes a sequence of machine instruction, which can be represented as the following notional program:

```
repeat  execution of <instruction cycle>
        {
                fetch(instruction);
                execute(instruction)
                        decode instruction
                        fetch operand;
                        execute the desired operation on data
                if (interrupt) process it and return to program execution;
        }
```

The 8086 microprocessor consists of two independent units (refer to Figure 13.1):

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

These units can function independently, therefore, they can function as two stages instruction pipeline. Components of these two units as shown in Figure 13.1 are explained in the following sections.

13.3.1 The Bus Interface Unit

The Bus Interface Unit (BIU) is responsible for external communication though the system bus. It has a dedicated address adder circuit, which takes input from segment registers and special registers of the processor. This unit also has an instruction stream queue of 6 byte length and is used to store the instruction, which is to be executed. The main tasks of this unit are:

- Computing the physical address of the instruction or data or input/output port from/ to the information is to be read or written into.
- This unit then reads or writes the information from the physical address as computed above.
    - If an instruction is fetched, it is stored in the instruction stream queue.

- Data is fetched into a general purpose register.
- In case of writing the data value of a selected register is written into a desired memory location or I/O port .

An instruction queue is useful only if more than one instructions are fetched simultaneously, which may be used for instruction pipelining involving the stages of instruction fetch and instruction execution.

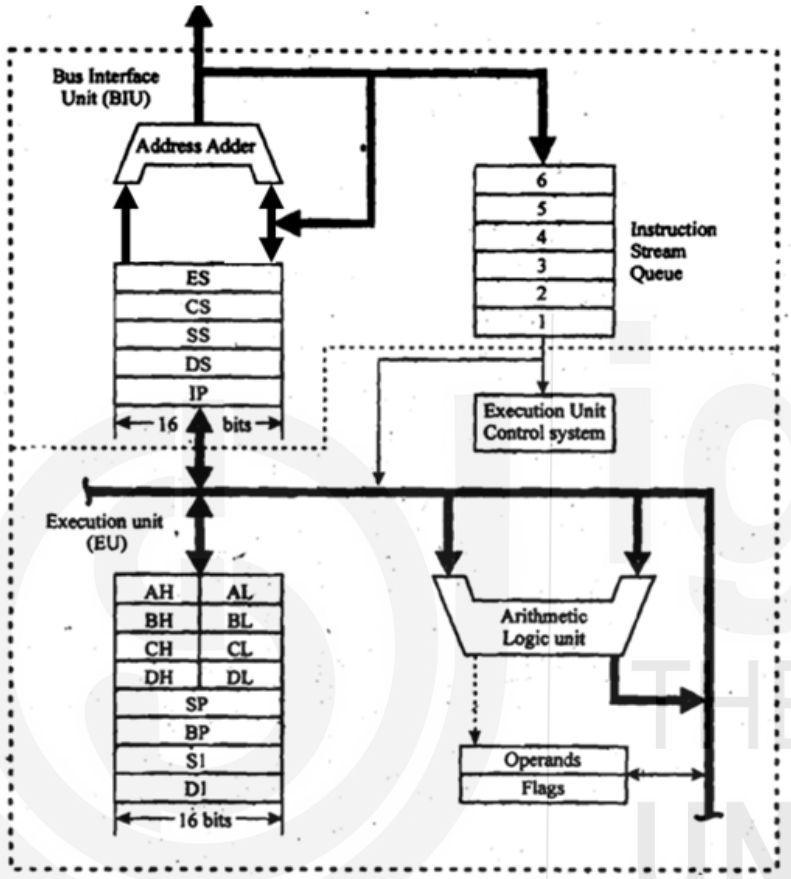


Figure 13.1: Structure of Intel 8086 Microprocessor

The Segment Registers

8086 microprocessor uses a very interesting concept of segmentation. As stated earlier that a 8086 microprocessor has 20 bit address but and 16 bit data bus. Now, assume that direct addressing scheme is used to address an operand in the memory. Since, the size of data is 16 bits, therefore, an efficient system will have a direct memory address of 16 bits, as this is the amount of data that can be fetched in one memory read operation. However, the address of an operand has to be 20 bits long (size of address bus). Thus, the two design options are either fetch two data words, which will be every inefficient; or use a concept of segmentation, which was specifically designed to this microprocessor.

The BIU of this microprocessor has four specific segment registers, namely CS: Code Segment register, DS: Data Segment register, SS: Stack Segment register, and ES: Extra Segment register. All of these registers are 16 bit long. Why segmentation? Segmentation divides the 1 MB memory of the computer associated with this microprocessor into logical overlapping segments of 64 KB. A program can have several code, data and stack segments. However, a maximum of four segments, one each of each type, may be available for accessing data and instructions at a specific

time, as there are four segment registers only. Thus, a program can consist of logical segments of code, data, stack etc. Thus, address of a data byte stored in the memory consists of a double *Segment Register (16 bits): Offset Register (16 bits)* Pair. How does this segmented addressing better than fetching two words? In the segmented scheme an address included in an instructions consist of only the 16 bit memory address, thus, a segment can be a maximum size of 2^{16} = 64 KB only. In addition, as the size of segment register is 16 bits, therefore, there can be 2^{16} = 65536 number of segments. Please note that these segments will be overlapping as the size of base memory for this processor is only 1 MB. Figure 13.2 shows the memory organisation of 8086 microprocessor. Thus, a segment register is loaded with the address of current segments and offset is used to represent data within that segment. Thus, instruction just needs to store 16 bit address only. The address adder computes the 20 bit physical address from the *Segment Register (16 bits): Offset Register (16 bits)* Pair. Please note all the content in Figure 13.2 is in hexadecimal notation.

Segment Start Address	Offset Address	Physical Memory Address	Byte content of Memory
0000 **Offsets in this segment: 0000 to 001F** **Segment size (in byte) = 0020 in hexadecimal** **Which is 32 in decimal**	0000	00000	52
	0001	00001	AA
	0002	00002	24
	…	…	…
	000E	0000E	CF
	000F	0000F	32
	0010	00010	32
	0011	00011	66
	…	…	…
	001E	0001E	--
	001F	0001F	--
0002 **Offsets in this segment: 0000 to 00DF** **Segment size (in byte) = 00E0 in hexadecimal** **Which is 224 in decimal** **0000 to 00DF** **Segment size = 00E0 in hexadecimal** **Which is 224 in decimal**	0000	00020	49
	0001	00021	23
	0002	00022	1F
	…	…	…
	000E	0002E	11
	000F	0002F	42
	0010	00030	34
	0011	00031	CD
	…	…	…
	00FE	0011E	AB
	00FF	0011F	AB
	…	…	…
	FFFE	1001E	DD
	FFFF	1001F	EE
…	…	…	…
Unused memory		**FFFFC**	--
		FFFFD	--
		FFFFE	--
		FFFFF	--

Figure 13.2: Memory Organisation of Intel 8086 microprocessor

Figure 13.2 shows two hypothetical segments (just for illustration) in the 1MB memory using hexadecimal notation. Assume that one segment data is of 30 bytes, thus it can be accommodated in the segment of size 32 Bytes. Please note that segment start address for this segment is 0000h and offset of these locations are 0000 to 001Fh. Therefore, the second segment can start from the physical memory address 00020h. The second segment is assumed to be of 64 KB and starting from physical memory 00020h. Therefore, it has segment starting address as 0002h and offset values

from 0000h to FFFFh. An interesting fact about the memory of 8086 processor is that, although a single byte has an address, but in a single memory access two bytes are transferred though data bus. For example, access to an even memory offset 0000h will transfer bytes at offset 0000h and 0001h. However, in case, you try to access an odd memory offset like 0013h, then the bytes 0012h and 0013h would be transferred to the processor.

Now, the question is given the segment starting address of 16 bits and segment offset of 16 bits, how will you compute the physical address? The designers of 8086 used an address adder to compute physical address. The addition is performed as follows:

Given: Segment Address 0002h; and offset say 0001h

Physical address is computed by shifting segment address to left by one hexadecimal digit (appending 0 as the lowest hexadecimal digit and add the offset in the result).

The Segment Address (hexadecimal)		0	0	0	2
Shift left and add zero in least significant digit	0	0	0	2	0
Add the offset		0	0	0	1
Resulting 20 bit physical address	**0**	**0**	**0**	**2**	**1**

Given: Segment Address 0002h; and offset say FFFFh. The physical address will be computed as:

The Segment Address (hexadecimal)		0	0	0	2
Shift left and add zero in least significant digit	0	0	0	2	0
Add the offset		F	F	F	F
Resulting 20 bit physical address	**1**	**0**	**0**	**1**	**F**

Please note that F+2 will be 15+2=17, so addend is 1 and carry is 1. Also when you add carry 1 to F, it will be 16, which is addend is 0 and carry is 1

What are the advantages of segmentation?

The following are the main advantages of segmentation.

- Since segment can overlap, therefore, in case a segment is smaller than 64 KB, next segment can start almost immediately from where the segment ends, thus, memory wastage is minimized.
- A program stores only the offset within a segment, thus, program code can be relocated to another segment, if the need so be. Thus, segmentation supports writing of reloadable programs.
- Instruction uses only 16 bit address field instead of 20 bits of address bus, thus, reducing the overall size of instructions and program.

Use of Segment Registers

As discussed ealier, 8086 microprocessor has four segment registers, they are used with specialized registers that store to compute physical address. These pairs are:

(a) Code Segment (CS) register and Instruction Pointer (IP) register, which is the offset of the next instruction to be executed, to compute the address of the next instruction to be fetched. The following example explains their use.

An assumed Code Segment(CS) Address (hexadecimal)		0	1	A	D
Shift left and add zero in least significant digit	0	1	A	D	0
Assume that IP contains an offset 1A3Ch		1	A	3	C
Resulting 20 bit physical address	**0**	**3**	**5**	**0**	**C**

Please note D+3= 13+3=16, therefore, carry = 1 and sum hex digit =0

A+A+carry(1) = 21, so carry = 1 and sum = 21-16=5

(b) Stack Segment (SS) register and Stack Pointer (SP) register, which points to the top of the stack in the stack segment, to compute the address of the top of the stack. The following example explains their use.

An assumed Stack Segment(SS) Address (hexadecimal)		F	1	1	D
Shift left and add zero in least significant digit	F	1	1	D	0
Assume that SP contains an offset 0110h		0	1	1	0

Resulting 20 bit physical address	**F**	**1**	**2**	**E**	**0**

(c) Data Segment (DS) register and Offset to compute the address of the data to be fetched. The following example explains their use.

An assumed Data Segment(DS) Address (hexadecimal)		A	5	8	3
Shift left and add zero in least significant digit	A	5	8	3	0
Assume that data offset is A021h		A	0	2	1
Resulting 20 bit physical address	**A**	**F**	**8**	**5**	**1**

(d) Extra Segment (ES) register and offset to compute the address of extra data segment (in case two data segments are used at the same time).

13.3.2 Execution Unit

The execution unit of the 8086 micro-processor consists of a set of general purpose and special purpose registers, an internal data and control bus, arithmetic and logic unit (ALU) and flags registers. The size of ALU of the 8086 processor is 16 bits. The execution unit control system decodes and performs the required operation on the data. It has the following main components:

Control Circuitry for Instruction Decode and operand specification and ALU

The 8086 processor uses a micro-programmed control unit, which decodes the instruction and executes it as per the micro-program stored in the control memory. The control unit is also responsible for generating the control timing sequences. ALU performs the operation on the data as instructed by the control unit.

Registers

8086 has several kinds of registers, which includes general purpose, special purpose registers and a special flag register. The next section explains the role of different registers of 8086 micro-processor.

13.3.3 Register Set

The 8086 registers have five different categories of registers. The following table explains the role of these registers.

Register Category	Register Name and Size	Special Purpose, if any
Segment Registers	CS (16 bits)	For storing the base address of code segment
	DS (16 bits)	For storing the base address of data segment
	SS (16 bits)	For storing the base address of stack segment
	ES (16 bits)	For storing the base address of extra data segment
General Purpose Register: Can be used for any computation, in addition they are used for specific purpose as stated in this table	AX - 16 bits ; it consists of two byte register AH, which stores the higher byte and AL, which stores the lower byte	It is also called accumulator register. It can store the results of addition or subtraction operation; for some instructions like multiplication and division it store one of the operand.
	BX - 16 bits ; it consists of BH and BL	It is also called base register. It stores the base location of a memory array.
	CX - 16 bits ; it consists of CH and CL	It is also called counter register. It can be used for keeping count in looping instructions

	DX - 16 bits ; it consists of DH and DL	This register can be used for I/O operation.
Pointer and Index Registers: These registers can also be used as general purpose registers	BP (16 bits)	Base Pointer register used in stack segment
	SI (16 bits)	Source Index register used in data segment
	DI (16 bits)	Destination Index register used in extra data segment
Special Register	SP	Stack Pointer register, points to the top of the stack.
Flags Register	It consists of 16 flags set by the last ALU operations. Each flag is 1 bit logn	Some of the important flags are carry flag (CF), Parity Flag (PF), Auxiliary Flag (AF), Zero flag (ZF), Sign Flag (SF), Overflow Flag (OF), Interrupt Enable flag (IF) and other control flags.

☞ Check Your Progress 1

1. What are the different components of Bus Interface unit and what are their uses?
   ……
   ……
   ……
2. Compute the physical address for a 8086 microprocessor for the following:

   (a) CS:IP = 0111h:0020h
   (b) DS:BX = 0211h:0100h
   (c) SS:SP = 42AAh:0123h
   ……
   ……
   ……
3. What is the role of a flag register in 8086 microprocessor? Can it be used as general purpose register?
   ……
   ……
   ……

13.4 INSTRUCTION SET OF 8086

In the previous sections, the basic structure of 8086 micro-processor was discussed. This section presents the instruction set of this micro-processor. This section presents only few instructions of each type, as idea is to present example of basic instructions, which you may require to write assembly programs in the next two units.

Interestingly, the 8086 instructions can be 1 byte to 6 byte long. A general format of instructions presented here is as under:

Label:	**Operation mnemonic**	**Operand(s)**	**; Comment**
LOOP:	**ADD**	**AX, DX**	**; AX← AX + DX**

Label is optional and is used to identify an instruction. It may be used if a subgroup of instructions are to be repeated. Operation mnemonic identifies the operation to be performed. These instructions, depending on the operation, may have zero, one or two

operands. *It may be noted that in 8086 micro-processor, if an instruction has two operands, then at least one of the operand has to be a register operand; or in other words both the operands cannot be memory operands*. This restriction is due to limits on the size of instruction. In addition, an operand address may use several addressing modes, which are discussed in section 1.5. The comments in the instruction are optional and are written after a semi colon (;) symbol. The example of an addition instruction is shown, where LOOP: is the label, ADD operation is to be performed on operand AX an DX. Please note that the comment in this case states the nature of addition instruction in 8086 microprocessor. The 16 bit contents of AX and DX are added and by the ALU and result is stored in AX register. In addition, this operation will also set the flags register.

The following are some of the important functional groups of the 8086 instructions.

13.4.1 Data Transfer Instructions

Data transfer instructions are required for moving the data between a pair of source and destination. Following are some of the more useful 8086 data transfer instructions.

MOV instruction: MOV destination, source (transfers source data to destination)

This instruction transfers the data from source to destination. The source or destination can be a general purpose register, immediate operand, memory location or I/O port. However, it may be noted that both the source and destination cannot be memory location in one instruction.

Example:

To move an immediate operand 2F1Ch into DX register, you can use the following instruction:

```
MOV DX, 2F1Ch
```

To move content of DX register into AX register, you can use the following instruction:

```
MOV AX,  DX
```

PUSH and POP instructions: PUSH operand or POP destination

PUSH and POP instructions are used to transfer a word (2 bytes) to and from the sword stack of 8086 microprocessor. The stack in 8086 microprocessor grows from a higher memory address to lower memory address as shown in Figure 13.3

	Offset	Stack content
	0000	
	0001	
	…	
	00FC	
	00FD	
SP	00FE	AB
-	00FF	BD
Initial Stack State		

	Offset	Stack content
	0000	
	0001	
	…	
SP	00FC	FF
	00FD	AA
	00FE	AB
-	00FF	BD
Let AX = AAFFh After PUSH AX		

	Offset	Stack content
	0000	
	0001	
	…	
	00FC	
	00FD	
SP	00FE	AB
-	00FF	BD
After POP DX DX = AAFFh		

Figure 13.3: Stack after one PUSH and one POP instructions.

Please note the following about Figure 13.3

- The stack is a word stack and the operands of PUSH and POP instructions are word operands. These two instructions does not affect the flag registers.

- The maximum size of this stack segment is 0100h having offsets 0000h to 00FFh. The stack segment register value is not shown.
- In 8086 microprocessor, the stack grows from higher offset to lower offset. The stack would be empty if SP contains 0100h. Stack is full when SP is 0000h.
- The PUSH instruction causes the decrementing the stack pointer by a value 2 (as stack is a word stack and the offset is an address of a byte), i.e. SP=SP-2, and then the word operand of the PUSH instruction is put in the stack locations pointed to by the SP.
- POP instruction results in moving the content at the stack location into the destination register, specified by the instruction. This is followed by incrementing the stack pointer register value by 2, i.e. SP=SP+2.

PUSHF and POPF instructions: The PUSHF instruction is used for pushing the current flags register on to the stack, while POPF pops the content at the top of the stack to fags register.

Other data transfer instructions

There are a number of other data transfer instructions. These instruction and their purpose is given in the following table:

MNEMONIC	DESCRIPTION
XCHG destination, source	Exchanges bytes of words of source and destination. At least one operand should be a register operand.
XLAT	This is a complex instruction, which translates a byte of AL register using a lookup table. This instruction uses AL register as the operand. An example of this instruction is given in Unit 15.
LEA register, source	This instruction results in loading of 16 bit effective address of source operand to the specified register operand. This instruction is useful for array index manipulation.
IN accumulator, port address OUT port address, Accumulator	This instruction is used to transfers a byte or word from a specified Input port to accumulator register. The instruction can use DX register as implied operand for port address. The port address can also be an immediate operand. This instruction can be used to transfer a byte or word, which is in accumulator register to specified output port address of an output devices, such as monitor or printer
LDS/LES	These instructions and used to loading data segment/extras data segment respectively along with one specified registers. Details on these instructions are beyond the scope of this unit.
LAHF/SAHF	The LAHF loads the low byte of flags register to AH register, while SAHF stores value of AH register to low byte of flags register.

13.4.2 Arithmetic Instructions

8086 microprocessor has a large number of arithmetic instructions. These instructions are explained below:

ADD and ADC instructions: ADD destination, source and ADC destination, source

The purpose of ADD instruction is to simply add the two operands and the result of addition is stored in *destination*. The source operand can be a general purpose register, immediate operand, memory location etc. the destination may be register operand or memory location. Also both the operands should not be memory operand in an

instruction. It may be noted that both source and destination operand should either be byte operands or word operands. This instruction causes changes in several flags of the flags register. Some of these flags are: carry flag, overflow flag, sign flag, zero flag etc. ADC instruction in addition to adding the source and destination also adds the Carry flag of the flags register.

.Example:

To add an immediate operand 2F1Ch into DX register, you can use the following instruction:

```
ADD DX, 2F1Ch
```

To add content of BL register into AL register, you can use the following instruction:

```
MOV AL,  BL ; the result of addition will be in AL register
```

INC and DEC instructions: INC destination and DEC destination

The purpose of INC instruction to increment the destination operand by 1, while DEC instruction decrements the destination operand by 1. The operand can a memory or register operand of byte or word type. the two operands and the result of addition is stored in *destination*. This instruction causes changes in several flags of the flags register. Some of these flags are: overflow flag, sign flag, zero flag etc. ADC instruction in addition to adding the source and destination also adds the Carry flag of the flags register.

Example:

To increment the AL register content by a value 1, you may use the instruction:

```
INC AL
```

To decrement the value of BL register by 1, you may use the instruction:

```
DEC  BL
```

AAA and DAA instructions:
The AAA instruction performs ASCII adjust after addition, whereas DAA instruction performs decimal, i.e. binary coded decimal, adjust after addition. These two instructions does not have any operand expect the implied operand, which is AL register. 8086 microprocessor allows you to add two decimal digits (0 to 9) stored in ASCII format, unpacked BCD format (which consists of single BCD digit in a byte) or packed BCD format (which consists of two BCD digits in a byte). AAA is used to adjust the results, if you have added ASCII digits or unpacked BCD. DAA is used to adjust the results, if you have added two packed BCD numbers. The following examples explains these two instructions.

Example:
Consider the AL register has ASCII digit '7' and BL contains ASCII '6'. You want to add these two values to get an answer 13 in decimal. One of the way would be to convert these operand into binary and perform the addition and convert the results back to desired format. Other way will be to use AAA as follows:

```
                ; Given AL = 00110111₂ and BL =  00110110₂
ADD AL, BL      ; Result would be 01101101₂  (incorrect sum)
AAA             ; Result would be adjusted by adding 0110 in lower
                ; 4 bits and setting the AF and CF flags as AL is greater than
                ; 9. The upper four bits will be made 00000011₂. The CF
                ; indicates that the result is 13.
Example for DAA
                ; Given AL =    00010111₂ (packed BCD 17)
                ; and     BL =  01010110₂ (packed BCD 56)
ADD AL, BL      ; Result        01101101₂  (sum 6D is incorrect)
DAA             ; As lower 4 bits > 9, so adjust lower 4 bits by adding 0110
```

; in it. Thus, the result will adjusted to 01110011_2

MUL, DIV and IDIV instructions: MUL source, DIV source and IDIV source

MUL and DIV instructions are unsigned multiplication and unsigned division instructions respectively. IDIV is a signed division instruction. The source can be a memory or register operand, which contains either byte data or word data. For these instructions one of the operand is assumed to be AL register (if data is of byte type) or AX register (if data is of word type). The result of MUL instruction is stored in AX register (if data is of one byte) or DX and AX pair (if data is of word type). Thus, symbolically the MUL instructions can be represented as:

AX ← AL × source (if source is 8 bit data)
DX, AX ← AX, sources (if source is 16 bit data)

In case in this instruction, if most significant bit of the result is 0, then carry and overflow flags are set to 0. In case a byte is to be multiplied with a word operand, then you must first convert the byte operand to a word operand using instructions like CBW given later in the unit.

The result of DIV and IDIV instructions for byte operands is stored as AH stores remainder and AL stores quotient of division, or for word operands DX stores the remainder and AX stores the quotient. in AH register (if data is of one byte) or DX and AX pair (if data is of word type). Thus, symbolically the MUL instructions can be represented as:

AH (Remainder) AL (Quotient) ← AL / source (if source is 8 bit data)
DX (Remainder) AX (Quotient) ← AX / source (if source is 16 bit data)

In the division operation a 0 value in the source register will result in run time error.

Example:

Assume that AL register contains 11h and BL register contains 02h.
Multiplication and division instructions will give following results:

```
MUL BL      ; Result 11h × 02h = 22h; The AH = 00h and AL=22h
DIV  BL     ; Result 11h / 02h = Remainder in AH= 01h and
            ; Quotient in AL 08h
```

CMP instructions: compares destination and source operands

This is a very interesting instruction used for comparing two operands. This instruction only sets the flag by subtracting source from the destination operand (both byte or both word). Both the source and destination operands cannot be memory operands at the same time. This operation may set carry flag zero flag, sign flag etc. The following example explains how flags may be set by this operand. This instruction only changes the flags, no operand value is changed.

Example:

Instruction	Flags if AX= CX	Flags if AX > CX	Flags if AX < CX
CMP AX, CX	CF=0; ZF=1; SF=0	CF=0; ZF=0; SF=0	CF=1; ZF=0; SF=1

Other arithmetic instructions

Some of the other instructions are given below:

SUB destination, source	This instruction subtract source from destination. The carry flag in subtraction is a borrow flag.
SUB destination, source	Subtracts with previous borrow, if any.
NEG source	Creates the 2's complement of source number.

AAS, DAS	Works in a similar manner as AAA and DAA, except they operate after subtraction operation.
AAM, AAD	Works in a similar manner as AAA, except the operation is multiplication and division respectively.
CBW, CWD	These instructions convert byte to word or word to double word respectively. The value of sign bit is filled in the upper byte or word as the case may be. For CBW operand is in AL register and resulting word is in AX register; whereas for CWD the operand is in AX register and the double word is in DX, AX pair.

13.4.3 Bit Manipulation Instructions

The Bit manipulation instructions are used to manipulate the bit wise data. These instructions are very useful in performing logical operation on the data. The following are some of the bit manipulation instructions:

NOT, AND, OR, XOR instructions:

These are logical instructions of 8086 microprocessor. NOT instruction takes only one operand, while all other instruction have destination and source operands. The operands can be memory or register operands and both the operands cannot be memory operands in a single instruction.

Example:

Let AL= 00111010_2 and BL=11011100_2

NOT AL ; the result in AL would be 11000101_2

AND AL, BL ; the result in AL would be 00011000_2

OR AL, BL ; the result in AL would be 11111110_2

XOR AL, BL ; the result in AL would be 11100110_2

TEST destination, source instruction:

This instruction performs the AND operation on the two operands, but does not changes the operands value. This instruction clears the carry and overflow flags and sign flag and zero flags are set as per the operand.

SHL and SHR; SAL and SAR; ROL and ROR; RCL and RCR instructions:

All these eight instructions are shift instructions with small difference. All these instructions take two operands *destination and count.* The count specifies the count of times the bits of the destination operand are to be shifted. The alphabet L or R at the end of instruction mnemonic specifies Left shift or Right shift respectively. Count sometimes can be stored in CL register.

Following diagram and example explains these shift operations, assuming that data is of byte type and the count of shift is by one bit:

CF	AL Register Value								SHL is shift left; SAL is arithmetic Shift left.
0	1	0	0	0	0	0	1	1	Initial Value
1	0	0	0	0	0	1	1	0	After execution of SHL AL or SAL AL
	←								Direction of shift

In SHL or SAL instruction 0 is put at the least significant bit (shown in green colour) and all the bits of the operand are shifted towards the left by 1 bit. The most significant bit is shifted to carry flag.

CF	AL Register Value								SHR is logical shift right.
1	1	1	0	0	0	0	1	1	Initial Value

1	0	1	1	0	0	0	0	1	After execution of SHR AL
	→								Direction of shift

All the bits of the byte are shifted towards the right. The most significant bit gets the value 0 and least significant bit is pushed to the carry flag (shown in green colour).

CF	AL Register Value								SAR is arithmetic shift right.
0	1	1	0	0	0	0	1	1	Initial Value
1	1	1	1	0	0	0	0	1	After execution of SAR AL
	→								Direction of shift

In the arithmetic shift right, all the bits are shifted towards the right. The most significant bit, which is a sign bit retains the same sign (please see the 1's in the left most position in the diagram above) and least significant bit is pushed to the carry flag (shown in green colour).

CF	AL Register Value								ROL is rotate left;.
0	1	0	0	0	0	0	1	1	Initial Value
1	0	0	0	0	0	1	1	1	After execution of ROL AL
	←								Direction of shift

In the Rotate shift left, all the bits are shifted towards the left. The most significant bit is shifted to CF as well as rotated to least significant bit, as shown above (in green colour).

.

CF	AL Register Value								ROR is rotate right;
0	1	0	0	0	0	0	1	1	Initial Value
1	1	1	0	0	0	0	0	1	After execution of ROR AL
	→								Direction of shift

In the Rotate shift right, all the bits are shifted towards the right. The least significant bit is shifted to CF as well as rotated to most significant bit, as shown above (in green colour).

CF	AL Register Value								RCL is rotate left with carry;
0	1	0	0	0	0	0	1	1	Initial Value
1	0	0	0	0	0	1	1	0	After execution of RCL AL
	←								Direction of shift

In the Rotate shift left with carry, all the bits are shifted towards the left. The most significant bit is shifted to CF, and the CF is rotated to the least significant bit (shown in blue colour)

CF	AL Register Value								RCR is rotate right with carry;
0	1	0	0	0	0	0	1	1	Initial Value
1	0	1	0	0	0	0	0	1	After execution of RCR AL
	→								Direction of shift

In the Rotate shift right with carry, all the bits are shifted towards the right. The least significant bit is rotated to CF (shown in blue colour), and the CF is shifted to the most significant bit.

☞ Check Your Progress 2

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

   a. POPF DX
   b. MOV BX
   c. XCHG MEM_BYTE1, MEM_BYTE2

d. DAA BL, CL
e. DIV AX, CH

2. Compare the different types of shift instructions of 8086 micro-processor.

…….
…….
…….

3. How can you check two operands are equal or not?

…….
…….
…….

13.4.4 Program Execution Transfer Instructions

In general, the execution of instructions of a program is sequential. However, there are certain instructions that results in execution of a different set of instructions. Some of these instructions including call to a procedure, return from a procedure, unconditional and conditional jump instructions etc. All these instructions contain an operand, which is the address of the next instruction which is to be executed as a consequence of execution of this instruction. The conditional jump uses flags register to determine if the jump is to be performed or not. Subroutine call instruction stores the return address. This section explains some of the important program execution transfer instructions.

CALL and RET instructions.

Call and return instructions are used form calling a procedure and once execution of the execution of the procedure is over RET instruction brings the control to the next instruction after the CALL instruction. In 8086 microprocessor there are two types of calls, viz. NEAR call and FAR call. The near call is within the same segment, whereas FAR call is to a different segment. A call instruction has the following basic format:

CALL <address of procedure>

Now, the question is how to recognize, if it is a NEAR or FAR procedure call? This is resolved by the assembler from the declaration of the procedure, which is created as a NEAR or FAR procedure. An example, explaining this is discussed in Unit 15 of this Block. A call to the procedure can be made using the CALL instruction. For example, if the name of a procedure in a separate code segment is procdeure1, then the following call instruction will be used:

CALL *procedure1* ;

This instruction will cause the execution for following sequence of operations:

1. If, it is a FAR procedure, then, present CS and IP should be saved as return address on the top of the stack, otherwise only IP will be stored on the stack.
   SP=SP-2; SS[SP]←CS; // This step will not be required in NEAR procedure
   SP=SP-2; SS[SP]←IP;
2. The CS will be loaded with the code segment address of *procedure1* and IP will be loaded with the offset of *procedure1*.
   CS= CS of *procedure1*; // This step will not be required in NEAR procedure
   IP = Offset of first instruction of *procedure1*;
3. The next instruction as per CS:IP value updated in step 2 will be executed next.

A procedure ends in a return instruction (RET). It causes the called procedure to return to the calling program. The following sequence of actions are performed by the RET instruction.

1. Perform the following actions:.

CS ← SS[SP] ; SP=SP+2;; // NOT performed in NEAR procedure
IP ← SS[SP] ; SP=SP+2;

2. The next instruction as per CS:IP value updated in step 1 will be executed next.

Jump instructions:

8086 micro-processor have instructions for unconditional and conditional jump instructions. The unconditional jump can be to NEAR or FAR label. It only requires one operand, which is the address, specified using a Label, of the next instruction to be executed. The format of this instruction is given below:

JMP Label

There are number of unconditional jump instructions. An unconditional jump instruction checks various flag register to determine, if the jump is to be taken or not. One of the mist common instruction to set flags prior to conditional jump instruction is the CMP instruction, which has been explained in section 13.4.2. Following table lists some of the conditional jump instructions along with the condition, when the jump will be taken.

Instruction	**Condition if the prior instruction is CMP AX, BX or any other arithmetic instruction**
JA/JNBE	Jump if AX > BX
JAE/JNB	Jump if AX >= BX
JB/JNAE	Jump if AX < BX
JBE/JNA	Jump if AX <= BX
JC	Jump if carry flag is set
JE/JZ	Jump if AX = BX
JNC	Jump if no carry
JNE/JNZ	Jump if AX ≠ BX
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 is even
JNP/JPO	Jump if parity flag is not set ; Jump if parity is odd
JG/JNLE	Jump if AX > BX
JA/JNL	Jump if AX > BX
JL/JNGE	Jump if AX < BX
JLE/JNG	Jump if AX <= BX
JS	Jump if sign flag is set
JNS	Jump if sign flag is not set
JCXZ	Jump to specified address if CX =0

The instruction JCXZ is a very useful instruction, when CX register is used as a counter.

Loop instructions:

A loop instruction (LOOP label) uses *CX register as a counter register*. The label in the loop instcution should be in the range -128 to +127. Prior to a loop instruction, the looping count value should be moved to CX register. The Loop instruction decrements the CX register and checks if CX register has zero value. If CX is not zero, then loop instruction takes the program back to the instruction, which is specified by the label of that instruction. In case, CX is zero then the loop is terminated, i.e., the next instruction after the loop instruction is executed in sequence. 8086 micro-processor has a number of loop instruction, which differ in condition the condition of loop termination. The following table lists some of these instructions, which may be used

later Units. There are many other such instructions for looping, a discussion on them is beyond the scope of this unit.

Instruction	Loop Termination
LOOP label	When CX register is zero.
LOOPE/ LOOPZ label	When a value being checked is unequal OR when CX register becomes zero.
LOOPNE/LOOPNZ label	When the value being checked becomes equal or CX register become zero.

Example: Let us assume you have byte array of 40h bytes. Write an assembly program segment that check if each of these elements have a value 00F0h.

Solution: Please note the two conditions - the first condition is that each element should be equal to 00F0h and the second condition is loop is to be executed 40h times. Thus, LOOPE instruction would be used, but prior to that you need to set different registers. The program segment for looping is shown below:

```
        ; Assume that the name of the array is BYTECOST
        MOV BX, OFFSET BYTECOST ; This instruction will cause the BX register to
                                ; point at the first element of byte array BYTECOST.
        DEC BX                  ; Decrementing the value of BX register by one.
                                ; This will cause BX to point to one byte prior to
                                ; BYTECOST array. Why is this instruction?
                                ; This is due to specific loop instructions below.
        MOV CX,40h              ; Initialise the loop counter to size of array
L1:     INC BX                  ; Move to the next element in the array.
        CMP [BX],0F0h           ; Compare the array element to 0F0h
        LOOPE L1                ; Loop if the present array element is equal to
                                ; 0F0h as per CMP instruction and CX is not zero.
```

It may be noted that LOOPE instruction will automatically decrement the value of the counter CX register.

In addition to the program execution control transfer, there are string instructions which are useful for string matching. Such instructions were specially designed for 8086 microprocessor, so that it can perform faster string comparisons. Some of these instructions are discussed in the next section.

13.4.5 String Instructions

String based instructions in 8086 were added to allow faster processing of strings based operations. A string is a sequence of ASCII characters in a 8086 micro-processor. Most of the string instructions use a subscript B to indicate that each character in the string is of byte type and a subscript W indicates that each character of the string is of the size of a word (16 bits). Following are the some of the string instructions.

REP, REPE/ REPZ and REPNE/REPNZ:

These keywords are used before any string instructions to repeat the following instruction to a number of times as specified in CX register. REP prefix repeats the instruction and decrements the CX register by 1, till CX becomes zero. The REPE/REPZ prefix repeats the instruction till either CX becomes zero or ZF becomes 0, whereas REPNE/REPNZ prefix repeats the instruction till either CX becomes zero or ZF becomes 1.

MOVS/MOVSB/MOVSW instruction:

This instruction moves data from one byte string to another byte string. This string operator uses several registers implicitly. The source string is assumed to be in data segment, indexed by SI register, whereas the destination string is assumed to be extra data segment indexed by DI register. CX is used as counter register. On transfer of one byte data from sources string to destination, automatically results in increment of SI and DI registers, and decrement of CX register.

Example: Assume that both data segment and extra data segment registers start from segment address 00FFh and a byte string of length 0100h starting at an offset 0400h is to be coped at an offset 0600h. Write the program segment to show this transfer.

; Assuming data segment and extra data segments registered are already initialised.

```
MOV CX, 0100h        ; Initialize counter CX
MOV SI, 0400h        ; Initialise SI
MOV DI, 0600h        ; Initialise DI
REP MOVSB            ; Will perform transfer till CX is 0.
                     ; SI and DI will be incremented after one byte is transferred
```

Other string instructions:

The following table shows other string instructions.

Instruction	Purpose
CMPS/CMPSB/ CMPSW	This instruction compares two byte or word strings, use of CX, SI and DI remains the same as MOVS. It is recommended to use REPE in this case.
SCAS/SCASB/ SCASW	This instruction compares a string with a value in AL or AX register for a byte or word string respectively. The string to be scanned is assumed to be in extra data segment. This instruction uses CX and DI registers, when REP prefix is used.
LODS/LODSB/ LODSW	This instruction is used to load a byte or word of a string pointed to by SI register into AL or AX registers respectively.
STOS/STOSB/ STOSW	This instruction is used to store a byte of word from AL or AX registers respectively into a location pointed by DI register.

13.4.6 Processor Control Instructions

These instructions are used to change certain parameters that are under the control of the programmer. You can control some of the flags, which may alter the conditional jump and direction of string manipulation. The following table briefly lists some of the most used processor control instructions. You may refer to the further readings for more details on such instructions.

Instruction	Purpose
STC	This instruction sets the carry flag.
CLC	This instruction clears the carry flag.
CMC	This instruction complements or inverts the state of the carry flag.
STD	This instruction sets the direction flag (DF=1), so the SI and DI are decremented automatically.
CLD	This instruction clears the direction flag (DF=0), so the SI and DI are incremented automatically.

There are many other process control instructions. You may refer to further readings to know more about these instructions.

13.5 ADDRESSING MODES

The 8086 micro-processor supports many operating modes to address the operands. These are – Immediate addressing mode, Register addressing mode, direct addressing mode and Indirect addressing modes. These addressing modes are explained in the following sections.

13.5.1 Immediate Addressing Mode

Immediate addressing allows an operand to be part of the instruction. The 8086 assembly language allows you to even use expressions as part of the instructions; however, these expressions should be computable at assembly time to produce a constant value. Some of the examples of immediate source operand are given below:

```
MOV AL, 45h          ; move immediate value 45h to AL
MOV AL, 'a'          ; move immediate ASCII character value 'a' to AL
MOV AX, 'ba'  ; move immediate ASCII characters values to AX register
MOV AL, (5+3)*2      ; move 16 to AL register.
```

13.5.2 Register Addressing Mode

A register addressing mode allows a any of the register of 8086 to be made as an Operand. However, some special registers cannot be used in every instruction. 8086 microprocessor may allow two register operands in a single instruction. Register operands can be 16 bit registers or 8 bit registers as shown below:

16 bit Register operands: AX,BX,CX,DX,SI,DI,BP,IP,CS,DS,ES,SS

8 bit Register operands: AH, AL, BH, BL, CH, CL, DH, DL

Register operand, in general, allows faster execution of instructions. Some example of instructions using register operands is given below:

```
MOV AL, BL   ; Move the content of BL register to AL register
             ; Both the register operands are of 8 bits
MOV DS, AX   ; loads data segment register using 16 bit AX register
```

13.5.3 Direct Addressing Mode

A direct addressing mode, in general, specifies a memory location as an operand in an instruction. An 8086 instruction can have a maximum of one memory operand. Interestingly, the 8086 memory address is of 16 bits only and contains the offset of an address; therefore, these addresses are called relocatable addresses. If a program is to reloaded in a different memory segment, it will just require to change the segment register and not the offset. Thus, programs can be relocated to any segment, without changing the instructions. Following are some of the examples of direct addressing mode:

```
MOV CL, LoopCount; loads the content of Loopcount location to CL register.
                 ; Segment register used will be data segment (DS)
JMP Loop         ; Jump to address specified by the label loop.
                 ;Please note that segment register for Loop
                 : would be the code segment.
```

13.5.4 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 and arrays. For this mode two types of registers are used. These are:

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

BX register contains the offset of the location in Data Segment, whereeas BP register points to the base of the stack segment register. The index registers SI and DI also contains offset in the Data Segment and Extra data Segment respectively.

These registers can be combined to create several indirect addressing modes. These are:

Register indirect: In this addressing mode the register contains the address of the data. In general, the type of register as stated above determines the segment in which the data is to be accessed. Examples of this mode are:

```
MOV AL, [DI] ;  Move the byte at the memory location ES:DI  to AL.
MOV AL, [BX];  Move the byte at the memory location DS:BX to AL.
```

Based indirect: In this addressing mode a base register and a displacement are added to compute the offset of address of data in the related segment. Example of this mode are:

```
MOV AL, [BX+2] ;  Move the byte at the memory location DS:BX+2 to AL.
```

Indexed indirect: In this addressing mode an index register and a displacement are added to compute the offset of address of data in the related segment. Example of this mode are:

```
MOV AL, [DI+2] ;  Move the byte at the memory location ES:DI+2 to AL.
```

There are two more such indirect addressing modes, viz. Based Indexed and Based Indexed with displacement, however, they are rarely used and are not explained in this Unit.

☞ Check Your Progress 3

1. Why are CALL and RET statements used?

……
……

2. What are the different types of Jump instructions? Why are they needed?

……
……
……

3. What are the different implicit operations of LOOP instruction?

……
……
……

4. Why do you need the string instructions?

……
……
……

5. Give one example each of each type of addressing mode of 8086 micro-processor.

……
……
……

13.6 SUMMARY

In this unit, you have gone through the basic architecture of 8086 microprocessor. This architecture was a creative design and used many interesting concepts related to enhancing the speed of instruction processing. First of these is the concept of use of segment registers to reduce 20 bit physical address to a 16 bit offset address, reducing the size of instruction using direct addressing, second faster string processing by using two separate segments to speed up string operations such as matching, third use of pipelining by designing two sections in CPU, fourth use of instruction queue for pre-fetching instructions and so on. 8086 assembly language forms the basis of Intel instruction sets of advanced processors and may help you appreciate the assembly language of those processors.

Some of the key features of this processor include:

- It has 20 bit address bus, therefore, base memory is 1 MB
- It has 16 bit data bus, thus can fetch two bytes simultaneously
- It has four segment rgisters that along with other pointer registers converts 16 bit offsets to 20 bit physical address.
- It has large number of instructions of different types, which allows writing of powerful assembly programs.

Please refer to the further reading for more details on 8086 assembly language programming.

13.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1. 8086 microprocessor has a Bus interface unit, which is a dedicated unit to compute memory address for reading/ writing a byte or word from/to the memory. It consists of a dedicated adder circuit, which converts 16 bit offset and content of 16 bit segment register to a 20 bit physical address. It also has a 6 byte instruction queue, which can store more than one instruction at a time. The execution unit performs the arithmetic. logical, shift, call, test and many other operations on data. It also contains registers, which store data and temporary results.

2. (a)

CS (in hexadecimal)		0	1	1	1
Shift left by one Hexadecimal digit	0	1	1	1	0
IP (in hexadecimal)		0	0	2	0
Physical address (Hexadecimal)	**0**	**1**	**1**	**3**	**0**

(b)

DS (in hexadecimal)		0	2	1	1
Shift left by one Hexadecimal digit	0	2	1	1	0
BX (in hexadecimal)		0	1	0	0
Physical address (Hexadecimal)	**0**	**2**	**2**	**1**	**0**

(c)

SS (in hexadecimal)		4	2	A	A
Shift left by one Hexadecimal digit	4	2	A	A	0
SP (in hexadecimal)		0	1	2	3
Physical address (Hexadecimal)	**4**	**2**	**B**	**C**	**3**

3. Flag register is used to store all the flag bits, which are generated as a result of last instruction. Some of these flags are sign flag, carry flag, overflow flag etc. Flag register cannot be used as a general purpose register.

Check Your Progress 2

1. (a) POPF instruction does not take any explicit operand.
   (b) Move instruction has a source and destination
   (c) An instruction cannot have two memory operands in 8086 microprocessor
   (d) DAA instruction has an implicit operand only. No explicit operand is to specified.
   (e) In DIV instruction you need to specify one operand only. The other operand is explicit.

2. SHL is shift left instruction and identical to arithmetic shift left instruction. Compare the different types of shift instructions of 8086 micro-processor. However, SHR and SAL differ and different input is added to the left most bit. Rotate instruction ROL and ROR just rotates the word/byte, whereas RCL and RCR also rotate the sign bit. (Please refer to section 13.4.3).

3. Perform test instruction on the operands (please make sure both the operands are not memory operand). If it sets the zero flag, then both the operands are same; otherwise they are different.

Check Your Progress 3

1. CALL statement calls a subroutine, i.e. the next instruction to be executed by the processor should be the first instruction of the subroutine. Since on completion of the subroutine execution the next instruction of the calling program is to execute therefore the return address is stored by the CALL instruction. RET instruction just brings the control back the the next instruction after CALL instruction in the calling program.
2. There are primarily two types of jump instructions: unconditional jump and conditional jumps. The unconditional jump instruction causes a compulsory jump to specified label. There are a number of conditional jump instructions, where a jump is taken if the related condition is fulfilled; else next instruction in sequence is executed.
3. Loop instruction in each iteration decrements CX register, and checks the value of CX. In case it is not zero, you go back to the Label from where the loop started. However, if the CX register is zero, the next instruction in sequence is executed.
4. String instructions in 8086 microprocessor are specially designed for efficient execution of string operations. For example, to match two strings, one string each be put in DS and ES with DS:SI pointing to first string and ES:DI pointing to second string. String length is put in CX register. The string matching instruction on using REPE command will compare the first byte and will increment SI and DI; and decrement CX. Thus, you do not need to write lengthy program for string matching, which includes all the operation as given above.
5. Immediate Operand

```
   MOV AL, (9+7)*2        ; move 32 to AL register.
Register Addressing
   MOV AL, DL             ; move DL to AL register.
Direct Addressing
   MOV AL, X              ; move content of byte location X to AL register.
Register Addressing
   MOV AH, [BX]           ; move content of location, whose address is
                          ; DS:BX to AL register.
```

UNIT 14 INTRODUCTION TO ASSEMBLY LANGUAGE PROGRAMMING

Structure **Page No.**

14.0 INTRODUCTION

In the previous unit, you have gone through the basic concepts of 8086 microprocessor, which included the 8086 structure, segmentation, register set, instructions and addressing modes. This unit present a basic framework for writing assembly language programs for 8086 microprocessor. In this unit, you will learn about the importance, basic components and development tools of assembly language programming. The Input/Output to an assembly language program is a complex process. This unit discusses the Input/Output to assembly program by using interrupts. This unit also discussed about different kinds of Assembly programs, viz. COM programs and EXE programs. Finally,the unit presents an example assembly program. An assembly program consists of assembler directives and instructions of 8086 microprocessor. This program is assembled using an assembler program. Several such assembler programs exist, which use different assembler directives. We have used the assembler directives, as used in Microsoft Assembler (MASM). However, these directives may be different for different assemblers. Therefore, before running an assembly program you must consult the reference manuals of the assembler you are using and change directives accordingly.

14.1 OBJECTIVES

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

- define the need and importance of an assembly program;
- usebasic directives for writing an assembly program;
- use interrupts to perform input/ output in an assembly program;
- define and differentiate between COM and EXE programs.

14.2 THE NEED AND USE OF THE ASSEMBLY LANGUAGE

The computer instructions are a sequence of 0's and 1's. These sequencesconsist of instruction operation code, addressing modes and operand addresses. The instructions of the programs written in the machine language are directly decoded by processing unit. However, you may have to face the following problems, if youprogram using machine language:

- Machine Language depends on machine instruction set and is difficult for most people to write in 0-1 forms.
- Debugging or correcting a machine language program is difficult.
- Deciphering the machine code is very difficult. Thus, program logic of programs written in machine language will be difficult to understand.

To overcome these difficulties computer manufacturers have devised English-like words to represent the binary instructionsof 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 better 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 execution module. An assembly instruction is directing translated to a machine instruction, therefore, assembly programs are highly optimized. 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 the compilers produce a long list of code for every instruction as compared to assembly language, which produces single line code for a single instruction. Further, complex instructions of a computer, like string instruction of 8086 highly that are highly optimized, can be used while writing an assembly program, making program faster. On the other hand, unlike high level languages,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 to write good logic of a program

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 High level languages with some highly efficient but non–portable routines. It will be worth mentioning here that UNIX is mostly written in C, a high-level language, but it has

about 5-10% machine dependent assembly code. In addition, many telecommunications applicationsuse assembly routinesto enhancethe efficiency.

14.3 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 anassembler program is used to translate the assembly language program into machine code. 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 (refer to Figure 14.1).

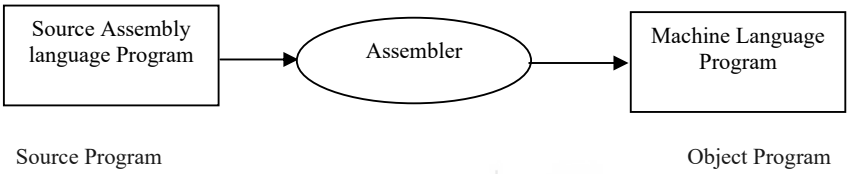


Figure 14.1: Use of assembler

Subsequently the object program is linked and an executable program is created. These steps are explained below:

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 generates a header in front of the.OBJ module; part of the header contains information about incomplete addresses in the object module. The .OBJ module is not an executable form.

Step 2: The link step involves converting the .OBJ module to an .EXE machine code module. The linkercompletes any address left open by the assembler and combines separately assembled programs into one executable module. In addition, it also initializes the .EXE module with special instructions to facilitate its subsequent loading of the .EXE program into the computer memory for execution.

Step 3: The last step is to load the program for execution. Because the loader knows where the program is going to be loaded in the memory, it is able to resolve all the remaining incomplete addresses in the header. The loader drops the header and creates a program segment prefix just before the program is loaded in memory.

These steps are shown in Figure 14.2

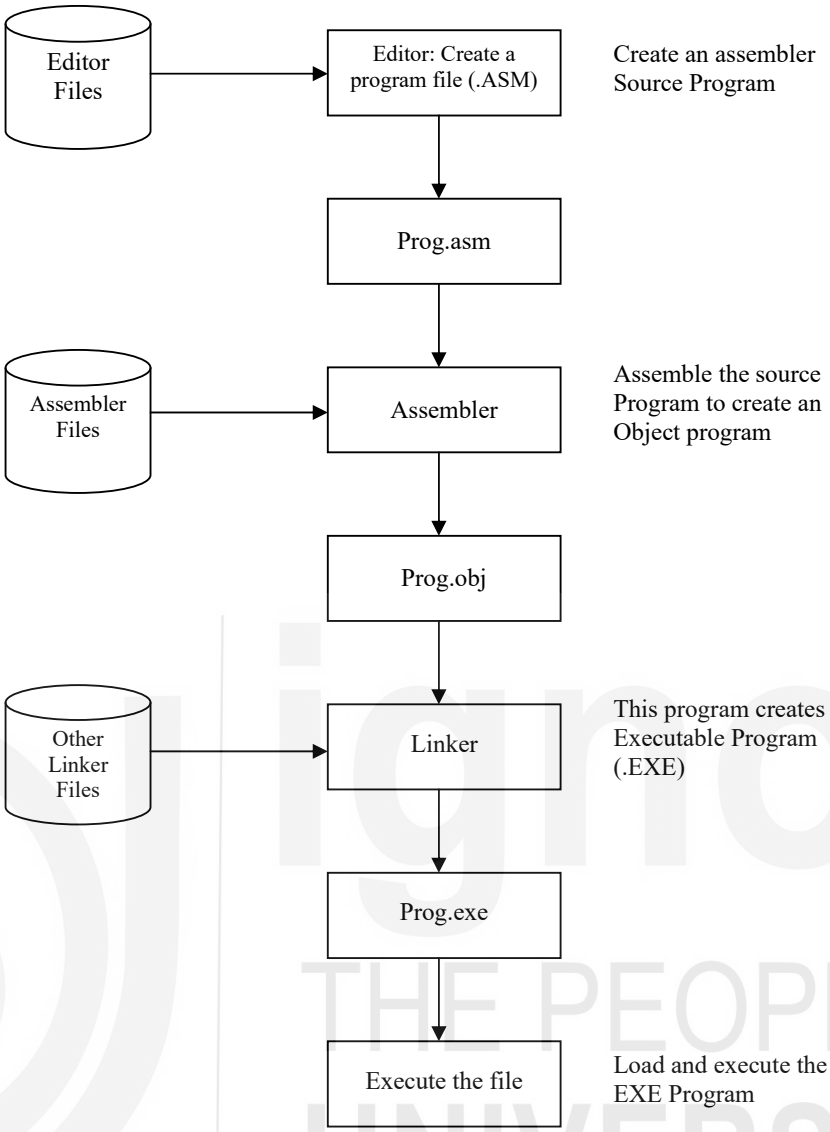


Figure 14.2: Program Assembly

Tools required for assembly language programming: Following are some of the basic tools needed to create assembly program. A modern-day assembler may contain several of these tools.

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 program createsan ASCII file. A common line editor program is NOTEPAD in Windows; vi editor in UNIX etc. An editor program may be part of assembler itself. You should use proper syntax of the assembly instructions to create an 8086-assembly program.

Assembler: An assembly program consists of assembly language instructions, which consists of assembly language mnemonics. The editor program, as defined above, is used to input the assembly language program and save it as a text file. An assembler is a program that converts the assembly language program, stored in a text file, into an equivalent machine language program. In general, this conversion is performed in two phases: first the assembler reads the assembly language file to collect various symbols used by the program along with their offsets in symbol table. On the second pass, it produces binary code for each instruction of the program and assignsan offset to all the symbols in the symbol table with respect to the segment base.

The assembler generates three files when your program gets successfully assembled with no errors. These three files are the object file, the list file and cross reference file. The object file contains the binary code for each instruction in the program.The errors that are detected by the assembler are called the symbol errors. For example, in the following statement the mnemonic MOVE is compared by assembler to all the mnemonics of the mnemonic set. It fails to get a match following which it assumes MOVE to be an identifier and looksfor its entry in thesymbol table.It does not find it there too, therefore gives an error "undeclared identifier".

```
MOVE AX1, ZX1 ;
```

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 historical assemblers available on PCs are MASM, TURBO assembler etc.

Linker: For better modularity programs are broken into several sub routines. It is even better to design common routine, like reading a hexadecimal number, writing hexadecimal number etc., which could be used by a lot of other programs.These common routines can be put into files and 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 linked file, which contains the binary code for all component modules. The linker also produces link map, which contains the address information about the linked files. The linker, however, does not assign absolute or physical addresses to your program. It only assigns continuous relative addresses to all the modules linked starting from the zero. Since these programs uses just relative addresses, they can be loaded in any physical memory address. Thus, these programs are called relocatable programs.

Loader: The basic purpose of the loader program is to convert the logical or relative addresses assigned by linker to absolute or physical memory addresses. This task is performed, while loader loads the linked program into the physical memory for execution. The linked program is brought from the secondary memory, like disk, to the computermemory for execution The file name extension of the files 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 contents of the register after the execution.
- It traces the execution of the specified segment of the program and displays the register and memory contents after the execution of each instruction.
- Disassemble a section of the program, i.e., converts 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, Emulator of 8086, or any other assembler
- Linker, which may be included in the assembler
- Debugger for debugging, if the need so be.

Errors

Two possible kinds of errors can occur in assembly programs:

a. Programming errors: They are the errors you can encounter in the course of executing a program written in any high-levellanguage, like syntax errors and semantic error
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. Therefore, assembly program should be tested in a safe mode.

14.4 AN ASSEMBLY PROGRAM AND ITS COMPONENTS

In this section, a simple assembly program is shown and its various components are explained. Consider the following program:

```
Line    Offset
Numbers          Source Code
0001             DATA SEGMENT
0002    0000     MESSAGE DB "Assembly LanguageProgramming$"
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      MachineCode    Assembly Instructions
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
```

Program1: A simple 8086 assembly language program

The details of this program are explained in section 14.4.1.

14.4.1 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 names: Segment name is specified by the programmer. It allows programs written in 8086 assembly language to be relocatable. They can be loaded practically anywhere in memory and run just as well. Program1 has to store the message "Assembly Language Programming$" somewhere in memory. It is located in the DATA SEGMENT. Since the characters are stored in ASCII, therefore it will occupy 30 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 astatement must appear in the appropriate order, but significance is attached to the column in which an element begins. Each statement must end with a new line character.

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 the name of an item,given by you, 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

   the statement
   A20: BL,45 ; defines a label A20.

Identifier can use alphabet, digit or special character.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.

14.4.2 Directives

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

1. **HEX:** The HEX directive facilitates the coding of hexadecimal values in the body of the program. This statement directs the assembler to treat related tokens in the source file as numeric constants in hexadecimal notation.

2. **PROC Directive:**PROC directive can be used in the code segment of an assembly program to create a procedure. You can use more than one PROC directives in a code segment. A PROC directive marks the start of a procedure. The end of a procedure is marked by the ENDP directive. The following example shows the start and end of a procedure named CheckZero.

```
CheckZero PROC  NEAR  ; Beginning of a Procedure in same segment
   …
CheckZero ENDP  NEAR  ; Marks the end of the Procedure CheckZero
```

3. **END DIRECTIVE:** There are three different END directives. These are:
   (i) ENDS Directive: This directive marks the completion of a segment. Thus, every segment used by you must have an ENDS directive.
   (ii) ENDP directive:As stated in point 2 it is used to mark the end of a procedure.
   (iii) END directive: It marks the end of the entire program. Any statement after this directive is ignored by the assembler.

4. **ASSUME Directive:** The purpose of assume directive is to identify the possible use of various segments defined in an assembly program. For example, if in your assembly program you have defined segments named STRING, CODE and STACK, which are to be used as data segment, code segment and stack segment respectively, then you can use the following ASSUME directive statement

   ```
   ASSUME  CS: CODE, DS: STRING; SS: STACK
   ```

5. **SEGMENT Directive:** The segment directive defines the segment name. A segment directive makes it possible to set a segment register to address the base address of a segment register (Please refer to discussion on segment register in Unit 13). All the offsets in a segment are computed from a base address of a segment.A segment directive indicates to assemble all statements following it in a single source file until an ENDS directive.

CODE SEGMENT

The code segment contains the code of the program, which may include procedures and sometimes other segments too. Linker marks the code segment in a program in a header. This header is used by the operating system when it invokes the loader to load an executable file of the program into memory. The loader reads this header for setting the CS register. A physical memory address is represented as CS: xxxx, where xxxx represents the offset in the code segment. In general, the first instruction of the code segment is assumed as the first instruction to be executed, therefore, is put at an offset of 0000H. The instruction pointer (IP) register is used to mark the offset of an instruction in code segment. The CS: IP pair is thus used to specify physical address of an instruction in a program that is being executed.

STACK SEGMENT

8086Microprocessor 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 issues a real time clock interrupts after every 55 milliseconds. 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 before the occurrence of interrupt. All such information gets recorded in the stack. Please note if you have not specified the stack segment it is automatically created.Why is stack segment essential? Consider your program is being executed by CPU, and a clock pulse need service, then if the system has no stack, then your CPU will not be able to return to your program again after serving of the clock pulse.

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: Specifies the data type to assemble.
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; i.e. 10 zero values.

EQU directive is used to define a name to a constant

```
CONST    EQU    20
```

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

Some other examples of using these directives are:

BINDB0111001B	; Binary value in byte operand named temp
OCT DW 7341Q	; Octal value assigned to word variable named VALI
DECDB49	; Decimal value 49 contained in byte variable names DEC
HEXDW03B2AH	; Hexadecimal value a is stored in a word operand HEX.

ASCDB 'EXAMPLE'	; Array of ASCII values is stored in variable ASCI.

☞ 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.	☐	

14.5 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 performed using these interrupts.

14.5.1 Interrupts

An **interrupt** causes interruption of an ongoing program. Some of the common interrupts are caused by devices like keyboard, printer, monitor, an error condition, etc.

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

Hardware interrupts are generated by a device that requests for some service. A software interrupt causes a call to the operating system. It usually is the **input-output** routine.

In 8086 software interrupts can be used for input-output of data. A software interrupt is initiated using the following statements:

```
INT     number
```

In 8086, this interrupt instruction is processing using an**interrupt vector table (IVT).** The IVT is located in the first 1K bytes of memory, and has a total of 256 entries,

each of 4 bytes. 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 14.3 shows the processing of an interrupt.

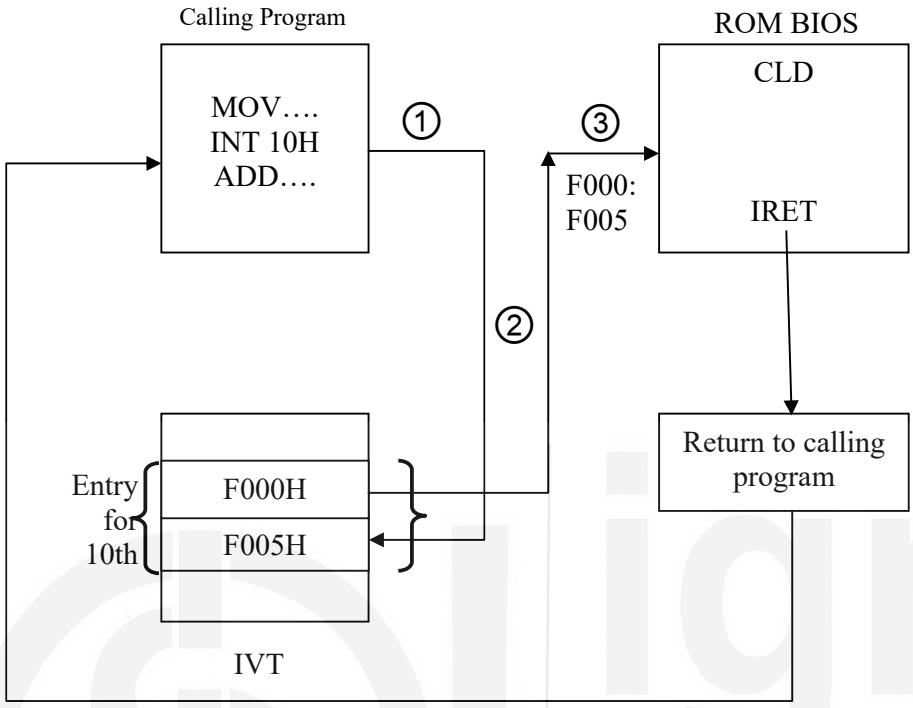


Figure 14.3: Processing of an Interrupt

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 10Hwould be found at IVT at an address 40H. Similarly, the entry of INT 3H will be at an address 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 address (CS: IP) as F000h:F065h

Step 3: The CPU loads the CS register and the IP register, with this new address in the IVT, and starts instruction execution process for that instruction.

Step 4: IRET (interrupt return) causes the program to resume execution of the next instruction of the program, which was being executed prior to interrupt servicing.

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 a character is a difficult program. However, these tasks were simplified by the architecture of the 8086. This architecture provides a pack of software interrupt vectors beginning at address 0000h:0000h,i.e., start of IVT.

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

14.5.2 DOS Function Calls (Using INT 21H)

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

Some important DOS function calls are given in the following table:

DOS Function Call	Purpose and Example
AH = 01H	This function called is used for reading a single character from keyboard and displaying it on monitor. The input value is put in AL register. For example, to read a character in a memory location X, you may use the following code fragment: `MOV AH,01H ; load AH register with the function value 01h` `INT 21H;` call the interrupt to read a character in AL `MOV X, AL;` Load the read character in memory location X

AH = 02H	This function prints an 8-bit data (normally ASCII), which is stored in DL register, on the screen. For example, to print a character '?' on the monitor, you may write the following code fragment:
	`MOV AH,02H ; load AH register with the function value 02h`
	`MOV DL, '?' ; Move the character to be displayed in DL`
	`INT 21H;` call the interrupt to display the character in DL

AH = 08H	This function is also used to input a single character into AL register, except that the character does not get displayed on the monitor. For example, to read a character in a memory location X, you may use the following code fragment: `MOV AH,08H ; load AH register with the function value 08h` `INT 21H;` call the interrupt to read a character in AL `MOV X, AL;` Load the read character in memory location X

AH = 09H	This function outputs a string whose offset is stored in DX register. The string is terminated by using a $ character. You can use this function to print newline character, tab character etc. For example, to print a string "Hello World", you may use the following code fragment:
	`DATA SEGMENT`
	`STRING DB 'HELLO WORLD','$'`
	`DATA ENDS`
	`CODE SEGMENT`
	…

<table>
<tr><td></td><td>

```
MOV AX, DATA; Put offset of Data Segment to AX.
  MOV DS, AX; Initialize data segment register using AX
  MOV AH,09H; load AH with the function value 09h
  MOV DX, OFFSET STRING; Store the offset of STRING in DX
INT 21H    ; Call interrupt 21H to display the STRING
  …
```

</td></tr>
<tr><td>AH = 0AH</td><td>
For input of string up to 255 characters. The string is stored in a buffer. For example, the following data and code fragment will input a string having a maximum length of 50 bytes. First, you need to define these parameters in the data segment, as given below:

```
DATA SEGMENT
     BUFFDB50
     DB?
     DB 50 DUP(0)
DATA ENDS
```

The name of the data segment, as given above, is DATA. It consists of total 52 bytes locations named BUFF. The first location of BUFF stores the decimal value 50, which is the maximum size of the string that can be stored in this buffer. The second location, marked with '?', will be used to store the actual size of the string, once it is read in the buffer. The remining 50 bytes at present are initialized as 0. These bytes will contain the string once it is read. The code segment, which will perform the string read operation is given below:

```
CODE SEGMENT
     …
     MOV AH, 0AH    ; Move 0A to AH register
     MOV DX, OFFSET BUFF; DX contains offset of BUFF
     INT   21H          ; Call interrupt 21h
      …
CODE ENDS
```

For the given code (complete the other necessary directives and statements) and data segment, if you input a value "Parv", then it will be stored in the BUFF as given below:

50	4	P	a	r	v	0	0	…	0

</td></tr>
<tr><td>AH = 4CH</td><td>This function call returns control back to the operating system.</td></tr>
</table>

Some examples of Input

(i) Input a single ASCII character to BL register, without displaying it on screen

```
CODE SEGMENT
  :
  MOV   AH, 08H   ; AH is loaded with function 08H
```

```
  INT   21H        ;Issue the Interrupt 21h
  MOV   BL, AL     ;  Transfer the input obtained in AL to BL
  :
CODE ENDS
```

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

```
CODE SEGMENT
  :
  MOV   AH,  01H
  INT   21H
  MOV   BL, AL
  SUB   BL,'0'   ; or use SUBBL,30H
  :
CODE ENDS.
```

Description: First you move the value of function 01H to AH register, next you call the Interrupt 21H. Execution of these two statements will result in input of a single character in AL register and display of that character on the monitor. You expect this character to be any digit amongst 0 to 9 (you can check this with additional assembly code, if needed.). The ASCII equivalent values of these digits '0' to digit '9' are 30H to 39H. The input value first is moved from AL to BL register and thereafter you either subtract '0', which is ASCII value of digit 0. Alternatively, can subtract 30H from BL, which is same as the hexadecimal value of digit '0'. This subtraction will result in binary equivalent value of the input digit in the BL register. For example, if you had input digit '8' as input, then BL register will contain: 38H-30H= 08H. This binary value then can be used for arithmetic operations.

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

One of the methods to input two-digit number would be to input two single digit numbers and using the place value of these digits convert them to a two-digit binary equivalent number. For example, to input a two-digit number 48, you will first input digit 4, convert it to binary and then input digit 8, which is also converted to binary. Now, perform the following computation to get an equivalent binary number into the specific register:

4*10+8 = 48

The assembly code for the same is given below:

```
CODE SEGMENT
  :
  MOV   AH,08H     ; Function 08H
  INT   21H        ; Interrupt 21H
  MOV   BL, AL     ; Move the value to BL register
  SUB   BL,'0'     ; Subtract '0' to get binary in BL
  MOV   AH, 08H    ; Now, start input of second digit
  INT   21H        ;
  MOV   DL, AL     ; Store AL in DL register
  SUB   DL, '0'    ; get the second binary digit in DL
  MOV   AL, 0AH    ; Move value 10 in AL register
  MUL   BL         ; Multiply AL by BL.
  ADD   BL, DL
  :
CODE ENDS.
```

Description: The code first input the two digits in BL and DL registers respectively, for example, an input 4 will be moved to BL and 8 will be moved to DL register, where they are converted to equivalent binary using subtraction. Next, BL is multiplied by 10, which is moved to AL register and added with the

value of DL, resulting in 4*10+8. This output is stored as binary value in BL register.

Examples of output on Display unit

(i) **Displaying a single character**

The following code displays the ASCII equivalent character of the data stored in BL register.

```
CODE SEGMENT
  :
  MOV   AH,02H      ; The single character output function
  MOV   DL, BL      ; Move the character to DL
  INT   21H
  :
CODE ENDS.
```

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

The following program first convers the binary value to equivalent decimal digit and then outputs it on the monitor. For example, if BL register contains a value '0000 0111', which is 07H, it will be converted to digit 7 by adding 30H or '0' and moved to DL register so that interrupt instruction displays this value on the monitor. The following code will perform the display as stated above:

```
CODE SEGMENT
  :
  ADD BL, '0'
  MOV DL, BL
  MOV AH,02H
  INT   21H
  :
CODE ENDS.
```

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

Assuming that the two-digit number 48 is stored as number 4 in BH and number 8 in BL. Each of these digits is converted to its equivalent ASCII digit by adding 30H or '0'. This is followed by displaying the equivalent ASCII of the digits respectively to output the given number.

```
CODE SEGMENT
  :
  ADD   BH, '0'
  ADD   BL, '0'
  MOV   AH, 02H
  MOV   DL, BH
  INT   21H
  MOV   DL, BH
  INT   21H
  :
  CODE ENDS
```

(iv) **Displaying a string**

Assume that a string is stored in the data segment with the label ST1. To display this string the following code can be used:

```
DATA SEGMENT
  ST1 DB "Output a string$"
```

```
DATA ENDS
CODE SEGMENT
      :
      MOV DX, OFFSET ST1
      MOV AH, 09H
       INT   21H
      :
    CODE ENDS.
```

An example with Input and output both:

To write a program that accepts an input character from the keyboard and respond. "The input is _".

```
DATA SEGMENT
      MESSAGE DB "The input is$"
DATA ENDS
CODE SEGMENT
    MOV  AX, DATA; Move data segment address to AX
    MOV DS,  AX  ; Initialize DS register
    MOV  AH,  08H ; Set function for character read
    INT  21H      ; Read character in AL
    MOV  BL, AL   ; Move input to BL
    MOV AH, 09H   ; Function to display strings
    MOV DX, OFFSET MESSAGE ; Move offset of string to DX
    INT  21 H         ; Display string named MESSAGE
    MOV  AH,  02H; Function to display character
    MOV DL, BL        ; Move character to DL
    INT  21 H         ; Display character
    MOV AX, 4C00H     ; Move 4CH to AH (DOS function call)
    INT 21H           ; Exit to DOS
CODE ENDS
END.
```

☞ Check Your Progress 2:

Q1: List the interrupts that can be used to input one character.

.....

.....

Q2: What is the output of following code segment, assume that BL register contains the binary value $0000\ 0010_2$

```
CODE SEGMENT
      :
      ADD BL, '0'
      MOV DL, BL
       MOV AH,02H
       INT   21H
      :
    CODE ENDS.
```

.....

.....

Q3: Name the interrupt used to exit to operation to operating system.

.....

.....

14.6 THE TYPES OF ASSEMBLY PROGRAMS

Assembly language programs can be written in two ways:

- COM Programs: A .COM program has only one physical segment, which includes all the different segments.
- EXE Program: An EXE programconsists of separate segments.

Let us look into brief details of these programs.

14.6.1 COM Programs

A COM (Command) program is the binary image of a machine language program. It is loaded in the memory at the lowestavailable segment address. In 8086 micro-processor, a COM program code begins at an offset 100h, as the first 1K locations are occupied by the interrupt vector table (IVT).

All the segments of a COM program are kept in the same segment, i.e., its code segment, data segment and stack segments are within the same segment. Since the offsets in a physical segment can be of size 16 bits, therefore the size of COM program is limited to 2^{16} = 64K which includes code segment, data segment and stack segment.The following program is a COM program, which adds two numbers. The program stores the result of addition and carry bit of addition in memory variables.

```
    CSEG SEGMENT
         ASSUME CS:CSEG, DS:CSEG, SS:CSEG
         ORG 100h      ;  Segment starts at address 0100h
START:   MOV AX, CSEG ;Move the segment address to AX
         MOV DS, AX    ;  Initialize Data segment using AX
         MOV AL, NUM1  ;  Transfer first operand to AL
         ADD AL, NUM2  ;  Add second operand to AL
         MOV RES, AL   ;  Store the result in AL to location RES
         RCL AL, 01    ;  Rotate AL by 1 bit to get carry into LSB
         AND AL, 00000001B;  Mask out all bits except the LSB
         MOV CARRY, AL      ;  Store the carry bit into location CARRY
         MOV AX,4C00h
         INT 21h
         NUM1DB   15h  ;  The first operand
         NUM2 DB  20h  ;  The Second operand
         RES DB   ?    ;  Stores the sum
         CARRY DB ?    ;  Stores the carry bit
   CSEG ENDS
   END START
```

The program initializes the data segment CSEG, which is also the code and stack segment too. The two operands are stored in memory locations NUM1 and NUM2 are added and the result is stored in the location RES. In order to store the carry bit first a rotate with carry instruction is executed, followed my masking out the upper 7 bits. This causes only the carry bit to remain in the AL register. This carry bit is then moved to the location CARRY. Finally, the program exits to DOS using Interrupt 21H.

The COM programs are stored on a disk with an extension ***.com***. A COM program uses less disk space in comparison to an 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.

14.6.2 EXE Programs

An EXE program is stored on the disk with extension ***.exe***. EXE programs are longer than the equivalent 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 compute the address of various segments and other components related to offsets. A detailed discussion on segment header is beyond the scope of this Unit.

The load module of EXE program consists of several segments of length up to 64K. It may be noted that in 8086 microprocessor a maximum of four segments may be active at any time. These segments can be of variable sizes, with the maximum size being 64K.

In the subsequent Units, you will be learning to write EXE programs only as:

- EXE programs are better suited for debugging.
- Assembled EXE programs can be easily linked to subroutines of high-level languages.
- EXE programs are easily to relocate in the memory, as they do not contain any ORG statement. It may be noted that ORG statement forces a program to be loaded from a specific memory 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 EXE program, which is equivalent to the COM program given in the previous section is given below:

```
        DATA SEGMENT
            NUM1DB   15h  ;  The first operand
            NUM2 DB  20h  ;  The Second operand
            RES DB   ?    ;  Stores the sum
            CARRY DB ?    ;  Stores the carry bit
        DATA END
        CODE SEGMENT
            ASSUME CS:CODE, DS:DATA
START:      MOV AX, DATA ;Move the segment address to AX
            MOV DS, AX    ;  Initialize Data segment using AX
            MOV AL, NUM1 ;  Transfer first operand to AL
            ADD AL, NUM2 ;  Add second operand to AL
            MOV RES, AL   ;  Store the result in AL to location RES
            RCL AL, 01    ;  Rotate AL by 1 bit to get carry into LSB
            AND AL, 00000001B;  Mask out all bits except the LSB
            MOV CARRY, AL     ;  Store the carry bit into location CARRY
            MOV AX,4C00h
            INT 21h
    CODE ENDS
    END START
```

14.7 HOW TO WRITE GOOD ASSEMBLY PROGRAMS

This section defines the art of writing good assembly programs. A good program requires a clear description or problem and the algorithm that is being used for solving the problem. The following are some of the advices, which may help you in writing good assembly programs.

1. Write an algorithm for your program closer to assembly language. For example, the algorithm for preceding program would be:
   - Assuming that both the numbers, NUM1 and NUM2 are in the memory.
     - Put first number from memory to AL
     - Add second number from memory to AL
     - Store the result in some memory location
   - Position carry bit in Least significant bit (LSB) of a byte
     - mask off upper seven bits
     - store the result in the CARRY location.

2. Specify the input and output of the program.

   Input: Two 8-bit numbers, in two different memory locations
   Output: An 8-bit result and an 8-bit carry in memory locations

3. Study the instruction set carefully: Study the available set of instructions, their format and their limitations. For example, the limitation of the move instruction is that it cannot move an immediate operand to a segment register. Thus, the segment address is first moved to a register, say AX, which is then used to initialize the segment register. cannot be directly initialized by a memory variable.
4. You can exit to DOS, by using interrupt routine 21h, function 4Ch. Therefore,04CH is placed in AH register followed by INT 21H instruction. This will result in exit to DOS.
5. 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 does not affect the size of your program.
6. You may assemble your program using an assembler, which helps you in removing the syntax errors. It also helps in creating an ***.exe*** file for execution.

☞ Check Your Progress 3

Q1: When would you use ***.com*** program?

.....

.....

.....

.....

Q2: Why are EXE programs preferred over .COM programs?

.....

.....

.....

.....

Q3: State True or False

	T	F
(i) Input/output on Intel 8086/8088 machine running on DOS require special functions to be written by the assembly programmers.	☐	
(ii) Intel 8086 processor recognizes only the software interrupts.	☐	

(iii) INT instruction in effect calls a subroutine, which is identified by a number. ☐

(iv) Interrupt vector table IVT stores the interrupt handling programs. ☐

(v) INT 21H is a DOS function call. ☐

(vi) INT 21H will output a character on the monitor if AH register contains 02. ☐

(vii) String input and output can be achieved using INT 21H with function number 09h and 0Ah respectively. ☐

(viii) To perform final exit to DOS we must use function 4CH with the INT 21H. ☐

(ix) Notepad can be used as an editor package. ☐

(x) Linking is required to link several segments of a single assembly program. ☐

(xi) Debugger helps in removing the syntax errors of a program. ☐

(xii) COM program is loaded at the 0^{th} location in the memory. ☐

(xiii) The size of COM program should not exceed 64K. ☐

(xiv) A COM program is longer than an EXE program. ☐

(xv) STACK of a COM program is kept at the end of the occupied segment by the program. ☐

(xvi) EXE program contains a header module, which is used by DOS for calculating segment addresses. ☐

(xvii) EXE program cannot be easily debugged in comparison to COM programs. ☐

(xviii) EXE programs are more easily relocatable than COM programs. ☐

14.8 SUMMARY

This unit introduces you to some of the basic concept of 8086 programming,especially input/output. 8086 microprocessor uses an interrupt vector table (IVT) that points to the address of the interrupt servicing programs of 8086 micro-processor. One of the most important interrupts being interrupt 21H, which is used for input/output and several different functions. An IVT provides a flexible design environment, as you can change the interrupt service program without much efforts. This unit discusses some of the important functions of INT 21H. This unit also differentiatesbetween COM & EXE program that are used in 8086 micro-processor.

14.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. (a) It helps in better understanding of computer architecture and 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 identifies 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

Q1: Interrupt 21H with functions 01H, 08H and 0AH
Q2: Output will be the digit 2.
Q3: Interrupt 21H with function 4C

Check Your Progress 3

Q1: The COM programs are of size less than 64K. When you require small fast functions, you may use COM programs.
Q2: Due to better structure and possibility of linking with the high level language programs. In addition, if your environment supports multiprogramming then EXE programs can be easily relocated
Q3:

(i) False
(ii) False
(iii) True
(iv) False
(v) True
(vi) True
(vii) True
(viii) True
(ix) True
(x) False
(xi) False
(xii) False
(xiii) True
(xiv) False
(xv) True
(xvi) True
(xvii) False
(xviii) True

14.10 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 15 ASSEMBLY LANGUAGE PROGRAMMING

Structure

15.0 INTRODUCTION

After discussing about the directives, program developmental tools and Input / Output in assembly language programming, let us discuss more about assembly language programs. In this unit, we will first discuss the simple assembly programs, which performs simple tasks such as data transfer, arithmetic operations, and shift operations. A key example would be to find the larger of two numbers. Thereafter, you will go through more complex programs showing how loops and various comparisons are used to implement tasks like code conversion, coding characters, finding largest in array etc. This unit also discusses more complex arithmetic and string operations and modular programming. You must refer to further readings for more details on these programming concepts.

15.1 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;
- write simple assembly programs for implementing arrays;
- explain the use of stack in parameter passing; and
- use modular programming in assembly language

15.2 SIMPLE ASSEMBLY PROGRAMS

In this unit, first simple assembly programs are discussed and later more complex programs are written. In this section several simple assembly programs are explained.

15.2.1 Data Transfer

Data transfer in one of the most fundamental operation. 8086 microprocessor has two basic data transfer instructions,viz. MOV and XCHG. These instructions are explained with the help of simple example.

Program 1: Write a program using 8086 assembly language to exchange a data word stored in a memory location with the value stored in BX register and interchanges the value of AH and AL registers.

Directives Statement	**Discussion**
DATA SEGMENT VAL DW4321H DATA ENDS	The data segment stores a variable VAL, which stores a data word.
CODE SEGMENT ASSUME CS:CODE,DS:DATA MAINP:MOV AX, DATA MOV DS, AX MOVAX, 8765H XCHG AH,AL MOVBX, 8765H XCHG BX, VAL MOVAX, 4C00H INT21H CODE ENDS END MAINP	Use of assume directive to correlate segment registers with segment names and explicitly initialize data segment register. Move 8765H to AX register. Result: AX=6587H Move 8765H to BX register. Result: BX=4321H and VAL=8765H Return to operating system using Interrupt 21h with function 4C.

Program 2: Write an 8086-assembly program that interchanges the values of two Memory locations.

Directives Statement	**Discussion**
DATA SEGMENT VAL1 DB 25h VAL2 DB 65h DATA ENDS	Define two variables VAL1 and VAL2 consisting of byte values.

CODE SEGMENT
ASSUMECS:CODE,DS:DATA
MOV AX, DATA
MOV DS, AX
MOV AL, VAL1
XCHG VAL2,AL
MOV VAL1,AL
MOV AX, 4C00H
INT 21h
CODE ENDS
END
	Use of assume directive to correlate segment registers with segment names and explicitly initialize data segment register.
	Load the variable VAL1into AL
	Exchange AL with variable VAL2
	Now, move the AL to variable VAL1
	Return to operating system using Interrupt 21h with function 4C.

In Program 2, why have we not used XCHG VAL1, VAL2 instruction directly? To answer this question,you should look into the constraints for the MOV instructions, which are given below:

- 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, VAL1, copies the VAL1 that is 25h to the AL register:
- The instruction, XCHG AL, VAL2exchanges the value of AL register (25h) with VAL2(65h). Thus, after the execution of this instruction AL will contain 65h and VAL2 will contain 25h. VAL1 at this time will also contain 25h only.
- Finally, the instruction MOV VALUE1, AL will put the value 65h into VAL1.

15.2.2 Simple Arithmetic Application

Let us discuss an example that uses simple arithmetic operations.

Program 3: Find the average of two-bytevalues stored in the memory locations named as FIRST and SECOND. The result of the operation can be stored in a third memory location named MEAN.

Discussion: The program should have two memory variables stored in memory locations FIRST and SECOND and a third location for storing the mean value.An add instruction cannot add two memory locations directly, so you are required to move a single value in AL first and then add the second value to it.In addition, on adding the two-byte values, there is a possibility of a carry bit. Assuming that problem is addressing two unsigned binary numbers, the problem is how to put the carry bit into the AH register such that the AX(AH:AL) reflects the added byte values. 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 addition, 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 mean value, you can divide the sum given in AX by 2. 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.

Directives Statement	**Discussion**
DATA SEGMENT FIRST DB 90h SECOND DB 78h MEAN DB ? DATA ENDS	Three variables
CODE SEGMENT ASSUME CS:CODE, DS:DATA	

START: MOV AX, DATA
MOV DS, AX
MOV AL, FIRST
ADD AL, SECOND
MOV AH, 00h
ADC AH, 00h
MOV BL, 02h
DIV BL
MOV MEAN,AL
MOV AX, 4C00H
INT 21H
CODE ENDS
END START
	Initialize data segment
	
	Get FIRST number in AL
	Add SECOND number to AL
	Clear AH register
	Put carry in AH
	Load divisor (2) in BL register
	Divide AX by BL. Quotient in AL and remainder in AH; and copy the result to memory and return to operating system.

15.2.3 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: Write a program in 8086 assembly language to convert a two-digit ASCII code to an equivalent packed BCD number. You may assume that these two ASCII digits are stored in AL and BL registers.
Discussion: to its BCD equivalent. An ASCII digit is of 8 bit length. The lower four bits of an ASCII digit represents the equivalent BCD value of the ASCII digit. For example, ASCII digit '3' is 00110011_2, so if we replace the upper four bits by 0s, you will get 00000011_2, which is equal to BCD digit 03. The number so 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 39 is 00000011 00001001, that is, 03 09. The packed BCD will be 0011 1001, that is 39. Thus, the algorithm to convert two ASCII digits to packed BCD can be stated as:
Input: The two-digit ASCII number
Output: Two-digit packed BCD number
Process:

Convert the ASCII digits to unpacked BCD. An example is given in the table below:

Digit	ASCII	Unpacked BCD
3	00110011	00000011
9	00111001	00001001

Move most significant BCD digit to upper four bit positions in byte by using rotate instruction as shown below:

0000 0011	This is a unpacked BCD of digit 3.
00110000	Use Rotate Instructions to get this

Pack the bits of rotated BCD digit with the least significant BCD digit bits, as shown below to create a packed two-digit BCD number in a byte.

Rotated value of digit 3	0011 0000
The unpacked digit 9	0000 1001
Use OR operator	0011 1001

Display or store the results in a desired location or register.
The assembly language program for the above can be written in the following manner.

Directives Statement	Discussion
DATA SEGMENT HighDigit DB '3' LowDigit DB '9' PackedBCD DB ? DATA ENDS	The data segment stores the ASCII value of the two digits, the assumed digits are '3' and '9'.

CODE SEGMENT
ASSUME CS:CODE, DS: DATA
START: MOV AX, DATA
MOV DS, AX
MOVBL, HighDigit
MOVAL, LowDigit
ANDBL, 0Fh
ANDAL, 0Fh
MOVCL, 04h
ROLBL, CL
ORAL, BL
MOV PackedBCD, AL
MOV AX 4C00H
INT 21H
CODE ENDS
END START
	Initialize data segment register
	
	Move the Higher digit (3) in BL
	Move the lower digit (9) to AL
	Mask upper 4 bits of BL
	Mask upper 4 bits of AL
	Move the rotate count to CL
	Rotate BL register using CL
	OR to get the packed BCD in AL
	Store the result in Packed BCD
	Return to Operating system

Discussion on Program 4:

8086 does not have any instruction to swap upper and lower four bits in a byte, therefore you need to use the rotate instruction4 times. You can choose any of the two rotate instructions, ROL and RCL. In this example, 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. Rest the entire program proceeds as per the algorithm.

Program 5: Write a program using 8086 assembly language that adds two binary numbers (assume the number are of byte type) stored in the consecutive memory locations. The result of the addition and carry, if any are also stored in the memory locations.

Directives Statement	Discussion
DATA SEGMENT NUM1DB 25h NUM2 DB 80h RES DB ? CARY DB ? DATA ENDS	First number contains 25h Second number contains 80h Will store sum of the two numbers Will store carry bit, if any

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
START:MOV AX, DATA
MOV DS, AX MOV AL, NUM1
ADD AL, NUM2
MOV RES,AL RCL AL, 01 AND AL, 00000001B MOV CARY, AL
MOV AH, 4CH
INT 21H
CODEENDS
ENDSTART
	Initialize data segment register
	
	Load the first number in AL, addthe second number in AL and store the result into RES
	Rotate AL with carry, to bring carry bit to the least significant bit, AND it with 00000001_2 to mask out all bits except the least significant bit. Store the carry bit to CARY and return to the operating system.

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 AL.

15.2.4 Larger of the Two Numbers

How are the comparisons done in 8086 assembly language? There exists a compare instruction CMP destination, source. However, this instruction only sets the flags on comparing two operands both of which should be either of 8 bits or 16 bits. Compare instruction just subtracts the value of the source operand from the destination operand without storing the result, but setting the flag during the process. In general,the following three comparisons may be able to address most of the comparison operations:

Instruction: CMP destination, source

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

The following examples show how the flags are set when the numbers are compared.

Example 1:

```
MOV BL, 02h          ;Move 02h to BL
CMP BL, 10h          ; Compare BL with 10h. Sets carry flag = 1
```

As the value of BL is less than 10h, the carry flag would be set as borrow would be needed to subtract 10h from BL.

Example 2:

```
MOV AX,F0F0h     ;  Same value is moved to AX
MOV DX, F0F0h    ;  and BX
CMP AX,DX        ;On comparison, it sets Zero flag = 1
```

The zero flag is set as both the operands are same.

Example 3:

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

The destination register (BX) contains a value greater than the source (0), so both the zero and the carry flags are cleared.

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 15.2.2 you can change instruction DIV BL with a shift instruction. ☐
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. ☐

15.3 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.

15.3.1 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 given 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 assembly code given above is not efficient, but suggests that there are many ways through which a conditional jump can be implemented. You should select the most optimum way based on your program requirements.

Example 6:

```
      CMP   DX, 00            ; checks if DX is zero.
      JE    Label1            ; if yes, jump to Label1 i.e., if ZF=1
      …
Label1:other instructions     ; 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    Label1            ; carry flag = 1 then jump to Lab1
        …
Label1:  other instructions     ; control comes here if condition is satisfied
```

LOOPING

Program 6: Write a program using 8086 assembly language that computes the new prices from series of prices data stored in the memory. You may assume a constant inflation factor that is added to each old price value. Also assume that all the prices are given in the BCD form.

Discussion:

Input: A list of prices stored in the memory and a constant inflation factor

Output: The new prices

Process:

Repeat the following steps
- Read a price (in BCD) from the input array
- Add inflation factor
- Adjust result to correct BCD
- Put result back in the same array

Until all prices are converted to new price

Directives Statement	**Discussion**

`ARRAYSSEGMENT`
`  PRICES DB 25h, 35h, 45h, 65h, 75h`
`ARRAYS ENDS`
	The data segment is named ARRAYS and consist of a list of 5 PRICES.

`CODESEGMENT`
`ASSUME CS:CODE, DS:ARRAYS`
`START:MOVAX, ARRAYS`
`MOVDS, AX`
`LEA   BX, PRICES`
`MOV   CX, 0005h`
 	
`DO_NEXT:MOVAL,[BX]`
`ADD   AL, 0Ah`
 	
`DAA`
`MOV[BX], AL`
 	
`INC   BX`
`DEC   CX`
 	
`JNZ   DO_NEXT`
`MOV   AH,4CH`
`INT 21H`
`CODE  ENDS`
`END START`
	Initialize data segment. Please note the use of name ARRAYS
	
	Move address of variable PRICES to BX register and move 5 to CX as PRICES has 5 values.
	Load the first value from array to AL and add constant 0Ah, which is assumed as inflation factor
	Since input is BCD, DAA adjusts the addition, and results are stored in the PRICES again.
	BX is incremented to point to next value and counter CX is decremented
	If the decrement operation does not result in zero, then jump is taken to DO_NEXT label, else all the values of PRICES has been processed, so program exits to Operating system.

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 named `ARRAYS`. [BX] is an

indirection through BX and points to the value stored at that element of array named 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. The LOOP instruction is demonstrated with the help of following program:

Program 7: Write a program using 8086 assembly language that prints the alphabets A-Z on the screen. This program is written using LOOP statement.

Directives Statement	**Discussion**

CODE SEGMENT
ASSUMECS:CODE
MAINP:MOVCX,1AH
MOV DL, 41H
NEXTC: MOV AH, 02H
INT 21H
INC DL
LOOP NEXTC
 	
MOV AX, 4C00H
INT21H CODE
ENDS
END MAINP
	1AH=26 (number of alphabets to be displayed)
	41H is hexadecimal equivalent of ASCII 'A'.
	Function 02H of Interrupt 21h is used to display the character stored in DL.
	Increment DL to next alphabet value
	Loop instruction will decrement CX by 1 and check if CX=0, if not it loops to label NEXTC to print remining characters.
	Once all the characters are printed, the program returns to the operating system.

Let us now discuss slightly more complex program in the next section.

15.3.2 Find the Largest and the Smallest Array Values

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

Program 8: Write a program using 8086 assembly language to find the largest and smallest numbers in an array.
Discussion: Initialize the **SMALL** and the **LARGE** variables as the first number in the array. They are then compared with the other array values one by one. If thevalue happens to be smaller than the assumed smallest number or larger than theassumed largest value, the **SMALL** and the **LARGE** variables are changed by this new value respectively. Let us use register DI to point the current array value andLOOP instruction for looping.

Directives Statement	**Discussion**
DATA SEGMENT ARRAY DW -1, 2000, -4000,32767, 500,0 LARGE DW ? SMALL DW ?	Data segment includes a total of 6 signed values. You need to find the largest and the smallest among these values. The largest and smallest values will be stored in variables LARGE and SMALL

DATA ENDS	respectively.

CODESEGMENT
ASSUME CS:CODE, DS:DATA
START:MOVAX, DATA
MOV DS,AX
MOV DI, OFFSET ARRAY
MOV AX, [DI]
MOV DX, AX
MOV BX, AX
MOV CX, 6
	Initialize the data segment register using AX
	Move offset of ARRAY of data segment to DI and move the array element pointed by DI to AX register. DX and BX registers are used to store the largest and smallest respectively. The first value of array is moved in both these registers. Since the size of array is 6, so move 6 to CX register.

A1: MOV AX, [DI] CMP AX, BX JGE A2	The element pointed to by DI is moved to AX, which is comparted with BX. In case, AX is greater than or equal to BX, which means AX is not the smallest. The program jumps to label A2.
MOV BX, AX JMP A3	This instruction will be executed, if the condition as above is false, which means AX is smallest, in this case the value of AX will be moved to BX, which contains new smallest value now. The program will then jump to label A3.
A2: CMP AX, DX JLE A3 MOV DX, AX	This statement will be executed, if junp is taken to label A2. The value in AX will be compared to largest in DX, if it is less or equal to the program will jump to label A3, otherwise the AX is largest value, so it will be moved to DX register.

A3:ADD DI, 2
LOOP A1
MOV LARGE, DX
MOV SMALL, BX
MOV AX, 4C00h
INT 21h
CODE ENDS
END START
	Next, DI is incremented by 2, so that it points to the next data word in the memory. The LOOP instruction decrements CX and checks if it is zero, if not then the steps from label A1 are repeated. Once the CX becomes zero, the DX is moved to LARGE and BX is moved to SMALL, as they contain the largest and smallest values respectively. Finally, program returns to the operating system.

Point to Note: 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.

15.3.3 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, you may enter the numbers as:

Enter first number	1234
Enter second number	3210

As each digit is input, you 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 stored as:

31	32	33	34	hexadecimal values stored in memory
1	2	3	4	equivalent 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.

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.

Let us discuss the process of conversion of ASCII digits to equivalent binary number, which can be used for computation.

15.3.4 Code Conversion

The conversion of data from one form to another is required in programs, which involve input of numbers. Therefore, in this section we will discuss an example, for converting a hexadecimal digit obtained in ASCII form to its equivalent binary form. An example showing ASCII to BCD conversion has already been explained as part of this unit.

Program 9: Write a program in 8086 assembly language to convert an ASCII input to equivalent hexadecimal value that it represents. The valid ASCII digits for this conversion are the digits 0 to 9 and alphabets A to F. The program assumes that the ASCII digit is read from a location in memory called ASCII. The hexadecimal or binary result isleft 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 ASCIIis not the proper hex digit.
Algorithm
Input: An ASCII digit
Output: The hexadecimal equivalent of number, if it is valid
Process:

IfASCII digitis in the range 30h to 39h it represents hex digit 0 to 9.
If ASCII digit is in the range 41h to 46h it represents hex digit A to F.
For any other value of ASCII, it does not represent a hex digit.

Directives Statement	**Discussion**

`DATASEGMENT`
`  ASCIIDB 39h`
`DATA ENDS`
	ASCII variable contains an ASCII digit.

`CODESEGMENT` `ASSUME CS:CODE, DS:DATA`	

START: MOV AX, DATA
MOV DS, AX
MOV AL, ASCII
CMP AL, 30h
JB ERROR
	Initialize data segment using AX
	
	Get the ASCII digits in AL register and compare it with 30h. If it is less than 30h, it is not a valid digit. So go to label ERROR

CMP AL, 3Ah
JAE ATOF
SUB AL, 30h
JMP CONVERTED
	These instructions will be executed only if the ASCII digit is 30h or more. In case the ASCII digit is equal to or above 3Ah, you jump to ATOF, otherwise the ASCII is in range 30h to 39h. So convert it and move to label CONVERTED.

ATOF: CMP AL, 41h JB ERROR CMP AL, 46h JA ERROR SUB AL, 37h JMP CONVERTED	You will be here if ASCII is greater than or equal to 3Ah. Check if it is below 41h, if yes go to label ERROR. Next, check if the ASCII digit is above 46h, if yes go to label ERROR. Otherwise convert the ASCII to hex digit equivalent by subtracting 37h. Next, jump to CONVERTED.
ERROR: MOV AL, 0FFh	Error is detected when AL has FF, which is moved to it.
CONVERTED: MOV AX, 4C00h INT 21h CODEENDS ENDSTART	Otherwise, AL contains the converted hex digit so the program returns to operating system.

Discussions:

The above program demonstrates a conversion of a single ASCII character to equivalent hexadecimal digit represented by that 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 BCD addition for such numbers? Why?

   ……

4. How can you implement nested loops, such as given below?

   for (i = 1 to 10, step 1)
   { for (j = 1 to 5, step 1)
   add 1 to AX}

   in assembly language?

   ……

   ……

15.4 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 (HLL) 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. The following section discuss a program of string processing:

15.4.1 String Processing

Let us write a program for comparing two strings.

Program 10: Write a Program using 8086 assembly language to match two strings of same length stored in two separate memory locations.

Directives Statement	**Discussion**
DATASEGMENT PASSWORDDB'FAILSAFE' DESTSTR DB'FEELSAFE' MESSAGEDB'String are equal$' DATA ENDS	 The source string The destination string The message to be displayed if strings are the same

CODESEGMENT
ASSUMECS:CODE,DS:DATA,ES:DATA
MOV AX, DATA
MOV DS, AX
MOV ES, AX
 	
LEASI, PASSWORD
LEA DI, DESTSTR
MOV CX, 08
CLD
 	
REPECMPSB
JNE NTEQ
MOV AH, 09
MOV DX, OFFSET MESSAGE
INT 21h
NTEQ: MOV AH 4CH
INT 21H
CODE ENDS
END
	The string matching requires two segments for data, viz. data segment for source string and extra data segment for destination string. Thus, DS and ES are initialized using AX.
	
	The offset of PASSWORD and DESTSTR are stored in SI and DI respectively. As both the strings are of 8 bytes, CX=8. The direction flag, which determines the direction of string processing is cleared.
	
	Repeat the compare string operation byte by byte and move to label NTEQ, if they are not equal at any character. The branch will not be taken, if the strings are equal. In that case the MESSAGE is printed on the screen. Finally, the program returns to the operating system.

Discussion:

In the program given above, 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 the next word respectively. 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, when string instructions are used.

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

15.4.2 Some More Arithmetic Problems

Let us now take up some more practical arithmetic problems.

Use of array in assembly

An array is referenced 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 processing, viz. SI and DI. You can also use any general-purpose register for indexing.

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

Carry in	0	0	0	1	0
	20	11	01	10	FF
	FF	40	30	20	10
1	1F	51	31	31	1F
Carry out					

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 11: Write a program in 8086 assembly language to add two five-byte numbers using arrays.
Algorithm:
Input: two arrays of 5 bytes each.
Output: an array of sum of size 6 bytes
Process:

Repeat the following steps till all the elements of array (5) are added
Load the byte of first array in AL
Add the corresponding byte of second array in AL with carry
Store the result in a memory array
Increment to next bytes
Rotate carry into LSB of accumulator
Mask all but LSB of accumulator
Store carry result in memory

Directives Statement	**Discussion**
DATASEGMENT NUM1DB0FFh,10h,01h,11h,20h NUM2DB10h,20h,30h,40h,0FFh SUMDB 6DUP(0) DATAENDS	Two arrays of size 5 each, SUM will store the addition and overall carry out bit

CODESEGMENT
ASSUME CS:CODE, DS:DATA
START: MOVAX, DATA
MOVDS, AX
MOVSI, 00
MOVCX, 05h
CLC
 	
AGAIN: MOV AL, NUM1[SI]
ADC AL, NUM2[SI]
MOV SUM[SI], AL
INC SI
LOOPAGAIN
 	
RCL AL, 01h
AND AL, 01h
MOVSUM[SI], AL
MOV AX, 4C00h
INT 21h
CODE ENDS
ENDSTART
	Initialize segment register
	
	SI register is being used as index register, in the array, therefore, is initialized to 0. CX is initialized to the size of arrays and CLC clears the carry bit
	
	First an element of array NUM1 is moved to AL and then the corresponding element of NUM2 and previous carry is added in AL. The result is stored in the memory and SI is incremented to the next element of the arrays. These operations are repeated for all the elements of arrays.
	
	Next, the final carry is rotated to LSB of AL and the higher bits are masked out. This final carry is also stored in the 6th element of SUM. Finally, program returns to operating system.

A good example of code conversion involving arithmetic is discussed next.

Program 12: Write a program using 8086 assembly programming language to convert a 4-digit BCD number into its binary equivalent. The BCD number can be stored as a word in memory location called BCD. The result is to be stored in location HEX.

The procedure to perform this number is explained with the help of following example.

Assume a 4-digit BCD number, say 4567, which is stored in a word in memory. To convert this number, you should extract each BCD digit separately and perform the following operation:

The binary number = 4 × (1000 or 3E8h) + 5 × (100 or 64h) + 6 × (10 or Ah) + 7

Directives Statement	**Operation**
THOUEQU 3E8h DATASEGMENT BCDDW4567h HEXDW? DATAENDS	Constant THOU is equal to 1000, i.e. 3E8h
CODESEGMENT ASSUME CS:CODE, DS:DATA START: MOVAX, DATA MOVDS, AX MOVAX, BCD MOVBX, AX MOVAL, AH MOVBH, BL MOVCL, 04 RORAH, CL RORBH, CL ANDAX, 0F0FH ANDBX, 0F0FH MOV CX, AX	Initialize data segment Register. AX = 4567 BX = AX = 4567 AL = AH = 45 BH = BL = 67 CL = 4, as 4-bit rotation will be used AH = 54 due to 4-bit rotation BH= 76 due to 4-bit rotation AX=5445 AND 0F0Fh = 0405 BX= 7667 AND 0F0Fh = 0607 AX is moved to CX so that AX can be used for

MOV AX, 0000H
MOVAL, CH MOVDI,
THOU MULDI
MOVDH, 00H
MOVDL, BL
ADD DX, AX
MOVAX, 0064h
MULCL
ADDDX, AX MOVAX,
000Ah
MULBH
ADDDX, AX
MOVHEX, DX
MOVAX, 4C00h
INT21h
CODEENDS
ENDSTART
	other operations. CX = AX = 0405
	
	AX=0
	AL=CH=04
	DI=1000
	AX= 04×1000 = 0FA0h
	DH=0
	DL=BL=07, thus DX= 0007
	DX=0FA0h+0007h=0FA7h
	AX=0064h
	CL=05; AX=5×100 = 01F4h
	DX= 0FA7h+01F4h= 119Bh
	AX=000A
	BH=6; AX=6×10=003Ch
	DX=119Bh+003Ch=11D7h
	Move this value to location HEX
	Return to operating system

☞ 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 REPE statement?
   ……
   ……
   ……

15.5 MODULAR PROGRAMMING

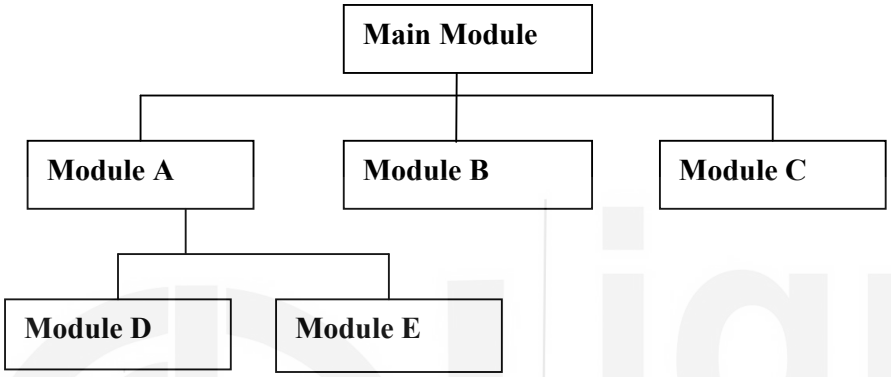
Modular programming refers to the practice of writing a program as a series of independently assembled source files. Each source file is a modular program designed to be assembled into a separate object file. Each object file constitutes a module. The linker collects the modules of a program into a coherent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

There are several reasons a programmer might choose to modularize 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.
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 localized to a module.

A modular program can be represented using hierarchical diagram:



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

15.5.1 The Stack

A procedure call is supported by a stack. Stack is a Last In First Out (LIFO) data structure. A stack in assembly language can be used for storing the return addressof procedures, for parameter passing and for storing the value returned by the procedure.

In 8086 microprocessor a stack is created in the stack segment. The SS register stores the base of stack segment and SP register stores the position of 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 of 8086 is a word stack. In order to use stack, first the stack segment register is initialized, as given below:

`STACK_SEG SEGMENT STACK`
`  DW 100 DUP (0)`
`  TOS LABEL WORD`
`STACK_SEG ENDS`
	Declaration of the stack segment.
	Assign 100-word locations to stack
	TOS is a word label to the top of stack.
	End of stack segment

`CODE SEGMENT`
`ASSUME CS:CODE,SS:STACK_SEG`
`  MOV  AX, STACK_SEG`
`  MOV  SS,AX`
`  LEA  SP,TOP`
`  …`
`CODE  ENDS`
`  END`
	Just like a data segment, the SS register is initialized to the base of stack segment.
	
	The SP register is loaded with the maximum offset of the stack, represented

The directive STACK_SEG SEGMENT STACK declares the logical segment for the stack segment. DW 100 DUP(0) assigns an actual size of the stack to 100 words. All locations of this stack are initialized to zero. The label TOS defines the initial top ofthis empty stack. Please note that the stack in 8086 is a WORD stack. The stack grows from a higher offset to a lower offset.The top position of stack uses an indirect addressing mechanism through a special register called the stack pointer (SP). SP initially is made to points to a label TOS. SP is automatically decremented,whenan itemis put on the stack (called PUSH operation) and incremented as an item is retrieved from the stack (called POP operation). SP points to the address of the last element pushed on to the stack. The following table explains the PUSH and POP instructions of 8086 microprocessor

Name	Mnemonics	Description
Push a word value SRC onto the stack	PUSH SRC	SP←SP – 2 ; (decrement SP to the next word) Put the word SRC into word pointed to by present value of SP and SP+1 ;
Pop a word value from the stack in DST	POP DST	Retrieve the word stored on stack top to DST ; SP← SP + 2 ;

15.5.2 FAR and NEAR Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, i.e., 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:

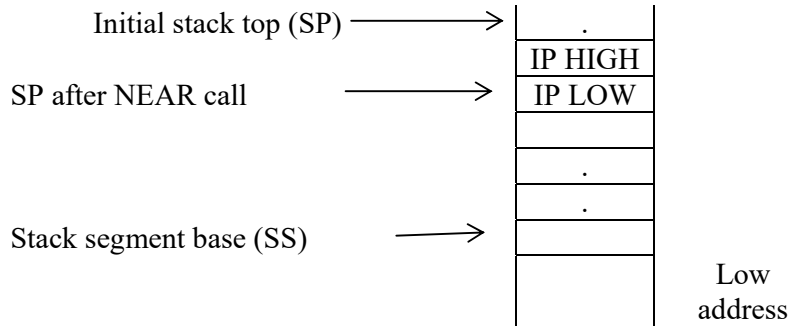
1. Unlike branch instructions, a procedure call must save the address of the next instruction to be executed of the calling program, so that after completion of execution of the procedure, the procedure can return the control to the calling program.
2. The registers used by the procedures need to be saved before their contents are changed by the procedure. These saved register values are used to restore the contents of the registers when the execution returns to the calling program.
3. A procedure must have 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.

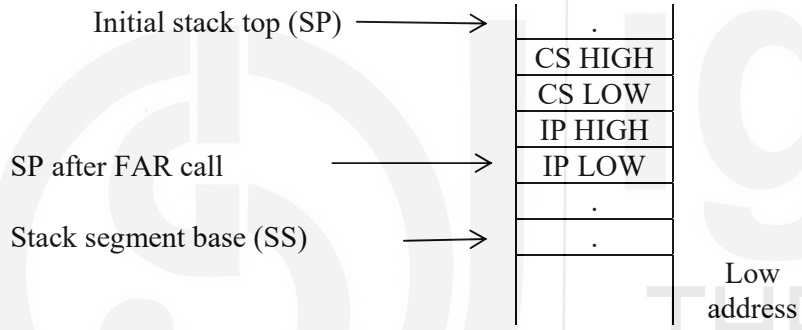
The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initialized IP with the address of the first instruction of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds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

The NEAR procedure call is also known as Intra-segment 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 on the top of the stack. The IP can be stored on the stack as:



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

15.5.3 Parameter Passing

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.

However, in this Unit we will discuss parameter passing using a stack with the help of an example.

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.

Program 13: Write a program using 8086 assembly language, which has a procedure to convert a two-digit packed BCD number to an equivalent binary number. Use stack for parameter passing.

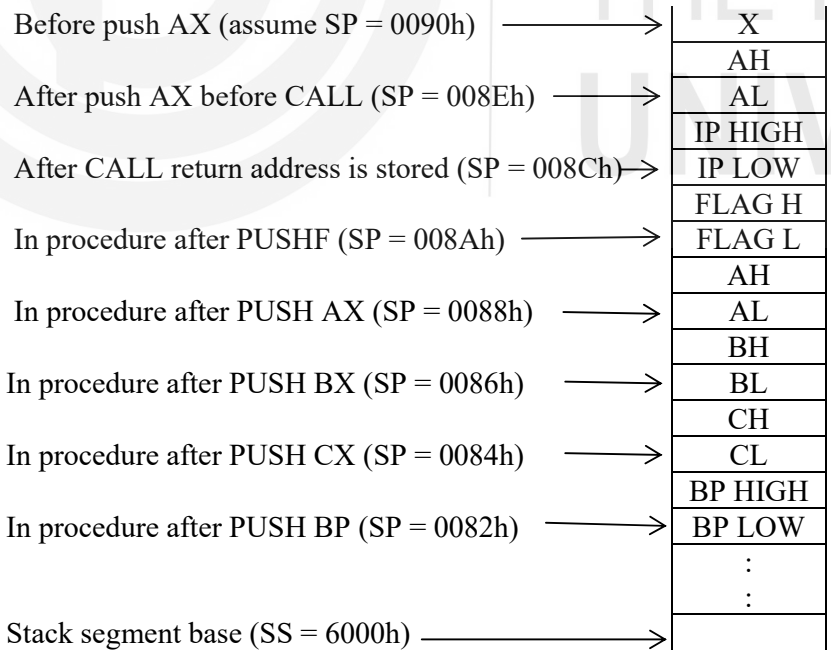
Discussion: The logic of conversion of packed BCD number to binary can be done in two simple steps. First convert the packed BCD digits to unpacked BCD digits and then multiply each digit with place value.

Directives Statement	**Discussion**
DATA_SEGSEGMENT BCDDB25h BINDB? DATA_SEGENDS	Storage for BCD value Storage for binary value
STACK_SEGSEGMENT STACK DW100DUP(0) TOP_STACKLABELWORD STACK_SEGENDS	Stack of 100 words Label for stack top

CODE_SEGSEGMENT
ASSUME CS:CODE_SEG,
DS:DATA_SEG, SS:STACK_SEG
START:MOVAX, DATA
MOVDS, AX
MOVAX, STACK-SEG
MOV SS, AX
MOV SP, OFFSET TOP_STACK
MOVAL, BCD
PUSH AX
 	
CALL BCD_BINARY
POPAX
MOV BIN,AL
MOV AH, 4CH
INT 21H
 	
; PROCEDURE : BCD_BINARY
BCD_BINARY PROC NEAR
; Store the registers
PUSHF
	Initialize data segment
	
	Initialize stack segment
	
	Initialize stack pointer
	Move BCD value into ALand push it onto word stack as parameter and call the procedure. The procedure returns binary value in AX register, which is moved to AL and the program returns to operating system.
	The procedure to convert BCD value received in AX register to binary value. But, first all the registers used by the procedure

PUSHAX PUSHBX PUSHCX PUSHBP MOVBP, SP MOVAX,[BP+ 12] MOVBL, AL ANDBL, 0Fh ANDAL, F0H MOVCL, 04 RORAL, CL MOVBH, 0Ah MULBH ADDAL, BL MOV [BP + 12], AX ; Restore flags and registers POPBP POPCX POPBX POPAX POPF RET BCD_BINARY ENDP CODE_SEG ENDS END START	and flags register is pushed in the stack. Next, the value of stack top is moved to BP register. The stack location [BP+12] contains the BCD value, which is moved to AX =0025h. BL=AL=25h BL= 25h AND 0Fh = 05h AL= 25h AND F0h = 20h CL=04 AL= 02h BH=0Ah (or 10) AX= 02×10 = 0014h AL=14h+05h=19h Move this binary value to stack Restore all the registers to their original content, restore is in reverse order of storage and return to the calling program End of procedure End of code segment End of the file

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 would be as follows:



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. Since the BP contains SP value, which is 0082h, therefore, BP+12 would be 0082h + 000Ch = 008Eh. This address contains the value of AX, which was pushed prior to call to the procedure. Please recall this pushed value was the BCD value, which is to be converted (this is the parameter value). Thus, this instruction copies the BCD parameter value at offset 008Eh into the AX register in the procedure. This instruction is not equivalent to POP instruction.

Stacks are useful for writing procedures for multi-user system programs or recursive 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.

15.5.4 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, you need segment combination and global identifier directives to write such programs. Let us discuss them briefly.

Concept of Segment Combination

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 name (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 having the length, as the sum of the lengths of individual segments.

For more details, you may refer to the further readings.

Use of 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.

The following example explains the use of external procedures in 8086 microprocessor.

Program 14: Write a program using 8086 assembly procedure that divides a 32-bit number by a 16-bit number. The procedure should be defined in one module, and other modules should be able to call this procedure.

The procedure is named a SMART_DIV procedure and first the calling program to this external procedure is given below:

Directives Statement	Discussion

DATA_SEGSEGMENT WORD PUBLIC DIVIDENDDW2345h,89ABh DIVISORDW5678h MESSAGEDB'INVALID','$' DATA_SEGENDS MORE_DATASEGMENTWORD QUOTIENT DW2DUP(0) REMAINDERDW 0 MORE_DATA ENDS STACK_SEGSEGMENTSTACK DW100 DUP(0) TOP_STACKLABEL WORD STACK_SEG ENDS PUBLICDIVISOR	Public data segment 32-bit dividend 16-bit divisor Message in case division is invalid This segment is valid only for this module as it is not shared. Stack segment of 100 words Label to stack top DIVISOR is a public variable
PROCEDURESSEGMENTPUBLIC EXTRN SMART_DIV: FAR PROCEDURESENDS	PPOCEDURES segment is a PUBLC division, it contains an external FAR procedure

CODE_SEGSEGMENTWORDPUBLIC
ASSUME CS:CODE_SEG
DS:DATA_SEG, SS:STACK SEG
START: MOVAX, DATA_SEG
MOVDS, AX
MOVAX, STACK_SEG
MOVSS, AX
MOVSP, OFFSET TOP_STACK
MOVAX, DIVIDEND
MOV DX, DIVIDEND + 2
MOVCX, DIVISOR
 	
CALL SMART_DIV
JNC SAVE_ALL
 	
JMP STOP
 	
ASSUME DS:MORE_DATA
SAVE_ALL:PUSH DS
MOV BX, MORE_DATA
MOV DS, BX
MOV QUOTIENT, AX
MOV QUOTIENT + 2, DX
MOV REMAINDER, CX
ASSUMEDS:DATA_SEG
POP DS
JMP ENDING
STOP: MOV DL, OFFSET MESSAGE
MOV AH, 09H
INT 21H
 	
ENDING: MOV AH, 4Ch
INT 21H
CODE_SEGENDS
ENDSTART
	It declares the code segment as PUBLIC so that it can be merged with other PUBLIC segments.
	Initially, DATA_SEG is used as the data segment
	There is only one stack segment for this main program
	
	AX is loaded with lower word (2345h), DX with higher word (89ABh) of the DIVIDEND; and DIVISOR is loaded in CX.
	Procedure is Called
	In case Carry flag is NOT set all the values are saved
	Unconditional jump to label STOP is executed.
	New data segment is assumed and initialized. The old DS is pushed to stack.
	
	The values of QUOTIENT and REMAINDER are saved.
	
	After saving the data segment is restored and unconditional jump is taken to end of program
	This code will be executed in case of Carry Flag is set, it will show the message INVALID to show that division was invalid.
	
	Finally, program will terminate

Discussion on the calling program:

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 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. The statement 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. Please also note that in case the procedure SMART_DIV encounters an error, such as division by zero, it sets carry flag, which is checked in the calling program to put the results in the memory or display an error message.

Let us now define the PROCEDURE module:

Directives Statement	**Discussion**
Input: Dividend is 2 words input. The low word is input in AX and high word is input in DX register The divisor is input in CX register. **Output:** The Quotient is returned in DX:AX pair and remainder is returned in CX register. In case, divisor is zero, then Carry Flag is set to indicate that division is incorrect.	

`DATA_SEGSEGMENTPUBLIC`
`EXTRN DIVISOR:WORD`
`DATA_SEG ENDS`
 	
`PUBLIC SMART_DIV`
	This declaration informs assembler that DIVISOR is a word variable and isexternal to this procedure. It also indicates that DIVISOR can be found in public segment DATA_SEG
	
	The SMART_DIV defined in this module is PUBLIC, i.e. it is available to other modules also.

`PROCEDURESSEGMENTPUBLIC`
`SMART_DIVPROCFAR`
`ASSUME CS:PROCEDURES,`
`DS:DATA_SEG`
 	
`CMPDIVISOR, 0`
`JEERROR_EXIT`
 	
`MOVBX, AX`
`MOVAX, DX`
`MOVDX, 0000h`
`DIV CX`
 	
`MOVBP, AX`
`MOVAX, BX`
 	
	It declares the PROCEDURES segment as PUBLIC so that it can be merged with other PUBLIC segments with the same name.
	
	The divisor is compared to 0, to check division by 0. You can also check it using CX. In case, it is same then jump to label ERROR_EXIT.
	
	AX containing lower dividend word is moved to BX and higher divided word is moved from DX to AX. The DX is emptied. The DX: AX now contains (0000h:89ABh). This is divided by divisor in CX. Leaving remainder in DX and quotient in AX. The quotient of higher word division is moved to BP and AX is loaded with lower word of dividend.
	
	The DX:AX pair is divided by CX.

DIVCX MOVCX, DX MOVDX, BP CLC JMPEXIT ERROR_EXIT: STC EXIT: RET SMART_DIV ENDP PROCEDURESENDS	Please note DX already contained the reminder of earlier division. The quotient of this division is stored in AX and remainder in DX The final remainder is transferred from DX to CX and the higher word division quotient saved in BO is moved to DX. Thus, DX:AX contains the quotient and CX remainder of this SMART_DIV procedure. The carry flag is cleared and control jumps to label EXIT This code is executed in case divisor was zero, the STC will set the Carry flag. The procedure returns.

Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the calling program, 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 calling program that is DATA_SEG and the procedure SMART_DIV is in a PUBLIC PROCEDURES segment.

☞Check Your Progress 4

	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 stack is used whenever assembly language programs are interfaced with any high level language programs.		☐
(e)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.

15.6 SUMMARY

This Unit presents some programs written in 806 assembly language. The programs cover elementary arithmetic problems, code conversion problems, use of arrays and

jump statements in assembly, the use of near and far procedure, highlighting the use of stack in procedure calls. Some of the important points presented in this unit are:

- An algorithm should 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.
- 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 requires you to initialize certain registers prior to their use in program, for example, LOOP expects the counter value to be contained in CX register, string instructions expect DS:SI to be initialized 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 etc. Study such requirements 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 you have also seen some assembly programs can be very efficient in comparison of HLL programs, for example, the assembly programs of string processing are very efficient. You should write assembly programs from the further readings.

15.7 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, 0Ah     ; move dividend to BX
    IDIV    BX          ; divide DX:AX by BX. The quotient will be in AX
    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 and is stored in data segment.

```
        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 repeatedly
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. You 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 you 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, you can perform string processing very efficiently in 8086 assembly language.

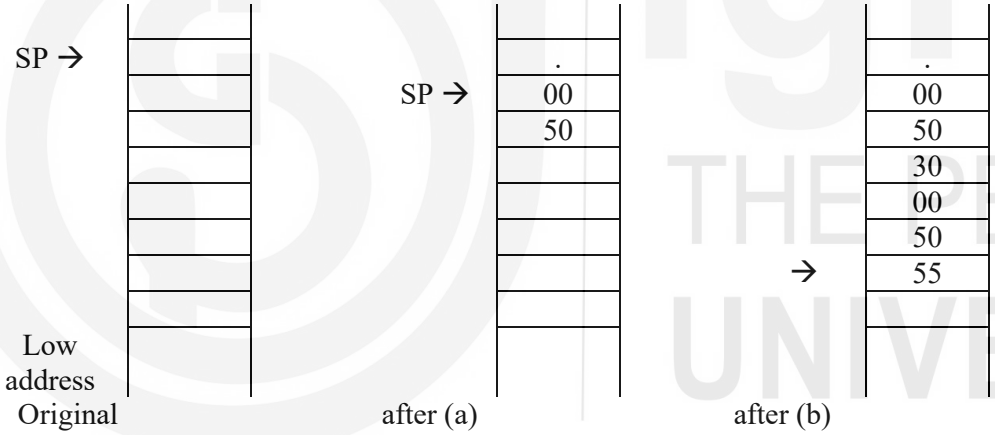
2. Direction flag if clear will cause REPE statement to perform in forward direction, i.e. comparison would be from first element to last.

3. It repeats the instruction after this instruction.

Check Your Progress 4

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

2.



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

UNIT 16 ADVANCED ARCHITECTURES

Structure **Page Nos.**

16.0 INTRODUCTION

The previous Units of this course discuss about the basic computer architecture of a computer system, including the assembly language. Architecture design is the first step in life cycle of a processor. It is a very crucial step as the performance of processor majorly depends on the design chosen. For instance, if you choose a non-pipelined architecture for your processor, you are compromising its performance for simplicity. On the other hand, if you choose an architecture involving multiple cores, you can increase your processor performance exponentially but at the same time its complexityincreases drastically. So, choosing an architecture that suits our application is a must. For that you should have an understanding of various types of architectures and their implementations in detail.

In this Unit, we will discuss some advanced architectures and design methodologies used to achieve higher performance of a computer system.

16.1 OBJECTIVES

After going through this Unit, you will be able to:

- define the concept of parallelism in both uniprocessor and multi-processor system.
- define various techniques used to implement pipelining in a processor.
- explain the concept of multiprocessor systems and interrelated details.
- define how processors communicate with each other.
- define the concept of Cache Coherence and
- define various types of multi-core processors.

16.2 NEED OF ADVANCED ARCHITECTURES AND PARALLEL PROCESSING

The fast-paced IT industry demands high-end architectural designs to support its high-performance applications. To meet these needs computer architectures are going through numerous changes constantly. One such implementation is parallel processing which is used to achieve higher speeds.

Parallel Processing includes a set of techniques that can be used to increase processing speed and latency, which is defined for the data rate of data transfer, of a computational system. These techniques enable simultaneous processing of data as opposed to the conventional sequential processing. Due to parallel execution of instructions, there is a significant increase in throughput. *Throughput* is the measure of computation and is defined in terms of number of instructions that can be executed by a given processor in a given interval of time. Increase in throughput implies to increase in speed of operation. To this extent it must benoted that the need of parallel processing is to increase the performance of a system. The techniques involved in inducing parallel processing in a system widely vary in methodology and resources used. However, the aim of all these techniquesis to process data concurrently. For example, in a processor, one instruction can be fetched from memory and another instruction can be executed by the ALU in the same clock cycle. Another example can be a system using two processors for parallel execution of instructions. In the former example, the Processor is using Pipelining as a technique to speed up the execution of instructions. We will discuss pipelining in the next sections in detail. The later example of two processors is used to achieve parallel processing. Here, performance of the system increases at the expense of increased complexity and cost.

16.3 PARALLELISM IN UNI-PROCESSOR SYSTEMS

Parallel execution of several program can be done on a single processor system. Such parallelism can be implemented using the hardware as well as software. In this unit, we will focus only on the hardware related parallelism in a uni-processor system.

As far as hardware-based parallelism is concerned several techniques are used on uni-processor system to increase the throughout of instruction execution. Some of these techniques are:

- Several processors contain multiple units to perform various arithmetic functions, such as multiple adder circuits unit as designed in Block 1. These units' speedup the execution of various functions that can be done in parallel.
- Using the memory organizations, such as interleaved memory, to speed up the data access operations. In addition, the processor operations can be overlapped with memory operations, for example, 8086 micro-processor
- Use of pipelining in the processor. Which is explained in detailed in this section.

The concept of pipelining is one of the major aspects of Parallelism in Uniprocessors, let us first answer the question "What is Pipelining?".

Pipelining is a technique in which we divide the whole task into subtasks and execute them concurrently. Each subtask is processed in different segment of the processor. These segments are interconnected to each other in such a way that the result of one segment is passed to another segment. Output is obtained after the data is processed through all the segments. The characteristic feature of pipelining is that several processes can be running in different segments at the same time. Typical related example of pipelining can be an assembly linein industries, such as automobile manufacturing, which fabricate automobile in a step-by-step manner in different segments. In the following sub-section, the concepts of arithmetic and instruction pipeline are explained.

16.3.1 Arithmetic Pipeline

In Arithmetic Pipelining an arithmetic operation is divided into a sequence of segments and computations of several arithmetic operationscan be done concurrently. Therefore, in this technique, there may be a possibility that a segment has produced data for the next segment, which is still processing the data of earlier operation. This will result in overlapping of data in different segments. To overcome this problem,you can use registers between segments to store data while next segment is still computing. Adding registers between stages can increase complexity but this step improves the reliability of the system.

Arithmetic pipelining is preferred in those systems in which same operation is to be performed on different data sets multiple times. The arithmetic pipeline can be used in computers used for high-end scientific computations to implement the floating point number arithmetic etc.

To describe Arithmetic Pipelining, let us consider apipeline that adds two floating-pointnumbers. Let us consider two numbers represented in floating point format.

$$A = X \times 10^{x}$$

$$B = Y \times 10^{y}$$

Here, X, Y are the mantissa of numbers A, B respectively and x,y are the exponents of A and B respectively. In order to find the sum of the two floating point numbers A and B, the following sequence of steps are required to be computed:

Step1: Compare the exponents of the given floating point numbers A and B, i.e. compare the values of x and y.

Step2: Align the mantissa of the number having smaller exponent. It may be noted that this process may result in unnormalized mantissa representation.

Step3: Perform the addition of mantissas to obtain resultant mantissa, while exponent of the bigger number is chosen as the exponent of the result.

Step4: If the resultant mantissa is not normalized, then normalize the result.

The following example explains the process of addition of floating-point numbers.

Example: Add the following two floating point numbers using the process as explained above:

$A = 0.5565 \times 10^{4}$ and $B = 0.166 \times 10^{3}$

Step1: The exponent of A, which is 4, is greater than that of exponent of B, which is 3.

Step2: In this step, you should align the mantissa of thesmaller numbers by modifying the B as follows (please note that A remains unchanged). Please note after alignment exponent of both A and B is same, however, the mantissa of B is unnormalized.

$B = 0.0166 \times 10^4$

Step3: Add the two numbers as follows:

$$Result = A + B = 0.5565 \times 10^4 + 0.0166 \times 10^4 = 0.5731 \times 10^4$$

Step4: Since the result is already normalized, this step will not perform any operation. Thus, the final result of the addition operation is 0.5731×10^4.

Registers are present in between all these steps to store intermediate values from one step to another step and prevent overlapping of data. The following diagram illustrates this process of addition or subtraction of two floating point numbers.

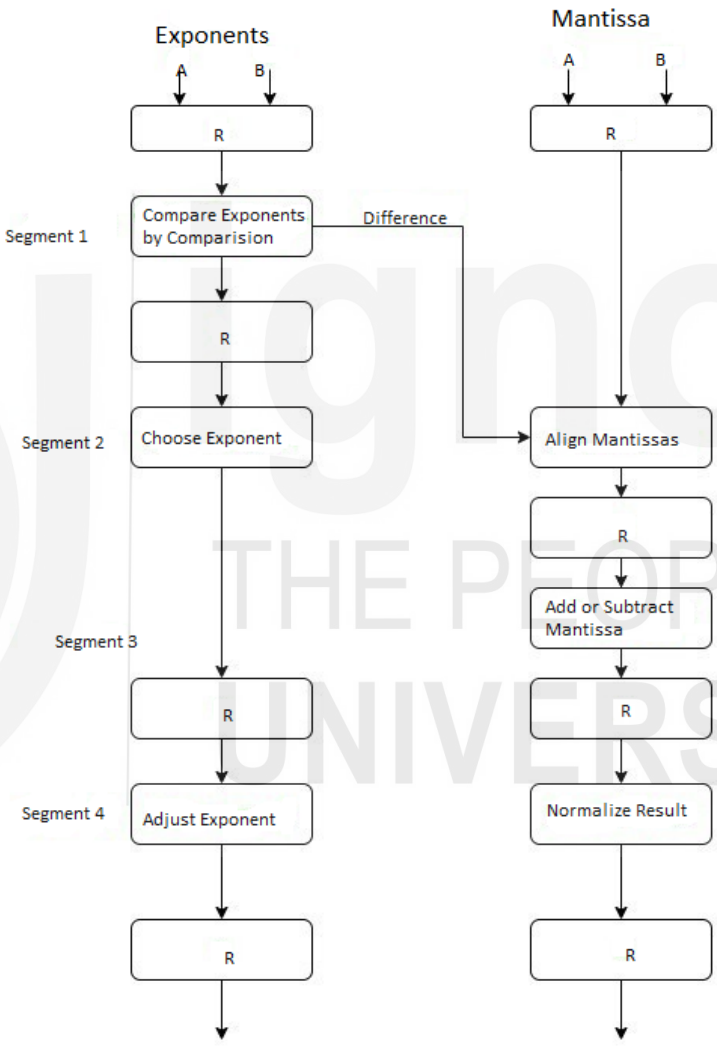


Figure 16.1: An arithmetic pipeline organization with intermediate registers for addition or subtraction of two floating point numbers.

16.3.2 Instruction Pipeline

In a processor when instructions are executed in various pipeline stages concurrently, then it is called instruction pipelining. The execution of an instruction is performed in asequence of stages. For example, in a 3-stage pipelined processor every instruction undergoes three stages namely Fetch, Decode, Execute.

Fetch: In the fetch stage, the instruction is fetched from instruction memory and stored in Instruction Register (IR).

Decode: In this stage the instruction is decoded for information like operation to be performed, source operand address, destination operand address etc., the source operand and destination operand may be stored in temporary registers for further processing.

Execute: In this stage the ALU performs the operation specified by the instruction on the operands in the temporary registers.

In the pipelined processor as stated above, when the first instruction is in decode stage the second instruction enters into fetch stage. When first instruction enters execute stage, second instruction enters decode stage and third instruction enters fetch stage. The process continues until all the instructions are out of the execution stage.

Assuming that one stage is performed in one clock cycle, then this processor will take**(k+n-1)** clock cycles to execute **n** instructions in a **k** stage pipeline. In the present case, k= 3, as we have used athree-stageinstruction pipeline. Thus, using this instruction pipeline **n** instructions, in an ideal condition, would be executed in **n+2** clock cycle.

If you are executing 100 instructions, then this processor will take 102 clock cycles to execute all of them. A Non-pipelined processor would have taken 300 clock cycles to complete the same task i.e., n×k.

We could see that there is a speedup ratio (S) $=\frac{(n\times k)t}{(k+n-1)t}$

Instruction 1	**FE**	**DE**	**EX**				
Instruction 2		**FE**	**DE**	**EX**			
Instruction 3			**FE**	**DE**	**EX**		
Instruction 4				**FE**	**DE**	**EX**	
Instruction 5					**FE**	**DE**	**EX**

Figure 16.2: An Instruction pipeline demonstrating execution of five instructions

The computation of speed up due to pipeline though looks very promising, however, the pipelined execution suffers from several problems. The first problem is due to limitation of resources of the processing unit. For example, the system bus is one of the resources required by several units.The second problem may be related to data dependencies among consecutive instructions, for example, an instruction may produce a result, which may be required by the immediate next instruction. This may sometimes may cause delay in execution of this immediate next instruction. Finally, in general, the decision to take a jump in conditional jump instructions is known only when the EX phase of that instruction is performed. This may result in emptying the pipeline. For example, consider instruction 1 is a conditional jump instruction, which causes jump to instruction 4 in case the condition is TRUE. The pipeline will execute as shown in the following diagram, assuming that the condition is evaluand to be TRUE.

Conditional Jump Instruction 1	**FE**	**DE**	**EX**				
Instruction 2		**FE**	**DE**	**-**			
Instruction 3			**FE**	**-**	**-**		
Instruction 4				**FE**	**DE**	**EX**	
Instruction 5					**FE**	**DE**	**EX**

Figure 16.3: An Instruction pipeline with conditional jump

Please note that the instruction 2 and instruction 3 are not to be executed, still they will be fetched. There a number of methods such as branch prediction, which can be used to handle such problem. A detailed discussion on these problems are beyond the scope of this Unit.

☞ Check Your Progress-1:

1) **State True or False**

i) A Non-pipelined processor is faster than a pipelined processor ☐

ii) Implementing parallelism in a system may lead to increased complexityof the system ☐

iii) Throughput is the measure of area of the chip ☐

iv) Modern day processors relay mostly on sequential data processing to improve their performance ☐

v) A Non-pipelined processor takes $n \times t_n$ clock cycles to complete n tasks with a clock period of t_n secs. ☐

2) Calculate the number of clocks cycles a processor takes to complete 180 instructions in a 7-stage pipeline.

……

……

……

……

3) Where can you use Arithmetic Pipelining?

……

……

……

……

16.4 PARALLELISM THROUGH HARDWARE AND SOFTWARE

As we know parallelism is the concept of processing multiple tasks concurrently. If this parallelism is achieved through hardware, then it is called Hardware parallelism. Similarly, if parallelism is achieved through software, then it is called Software parallelism.

Hardware Parallelism:

- Hardware parallelism is implemented in a system at the architectural level by implementing hardware multiplicity or by modifying machine architecture.
- While implementing hardware parallelism one should keep in mind that it is a tradeoff between cost and performance.
- For example, using multiple adders can speed up a system which is initially working with single adder. But there is an overhead of cost of extra adders implemented in the system. Depending on the application you should choose between cost and performance.

Software Parallelism:

- Software parallelism is achieved by modifying programs for data dependencies and control.
- Parallelism in software varies during the time of execution.

In the next section we discuss vector processing, which propose software instructions to exploit parallelism. This would be followed by a discussion of array processor, which is a hardware to exploit the parallelism.

16.4.1 Vector Processing

The capabilities of a conventional computer are limited to computations that are simple and not as complex as vector or matrix arithmetic. But some modern-day applications need more complex calculations to be performed on huge amount of data. In this scenario, performing computations on scalars one by one is not recommended. To overcome this limitation, computers with vector processing are preferred to do faster vector calculations. There are many scientific problems and real-time computations that could take many weeks on a conventional computer to complete. Vector processers do the same problem in a much faster way. But how do they do that? The following description and exampleattempt to answer this question.

Instruction sets in vector processors contain instructions that operate on vector data, such as one-dimensionalarray. In scalar processors the instructions operate on a single data element at a time.

A scalar subtraction operation can be represented asC= A-B, where the scalar value B is subtracted from scalar value A in a single step.

However, if both A and B are vector operands, you may perform the subtraction operation, as shown below:

$C_i = A_i - B_i$, where i=1,2,3, 4,…*n* and *n* is the number of elements in the vector

If you follow the above procedure, then you will subtract vector B from vector A in *n* steps.

In vector processing vectors are subtracted in a single step. This procedure saves (n-1) clock cycles, which makes a huge difference when processing large arrays of data.

Vector instructions and scalar instructions are specified in a different way. For example, a typical scalar instruction having three operands can be defined as follows:

Scalar instruction format:

Opcode	Address of source 1	Address of source 2	Address of destination

Whereas, a typical vector instruction would require to specify the base addresses of the operands and the length of the vector, which specifies the number of elements in those vectors. A typical vector instruction can be represented as follows:

Vector Instruction Format:

Vector Opcode	Base address of source operand1	Base address of source operand2	Base address of destination operand	Length of vector

Example: SUB X,Y,Z,10

Here opcode is SUB, this specifies that the operation to be performed is subtraction.Base address of source operands are X and Y respectively and the Base address of destination operand is Z. The length of vector is 10.

The scalar version of the program may be written as follows:

for i = 1 to 10 SUB X[i], Y[i], Z[i]

The advantage of the vector instruction over scalar instruction is that the scalar version of instruction would require repeated fetch and decode of the subtraction instruction, whereas, the vector instruction would be fetched and decoded only once. The efficiency of execution of vector instructions can be further enhanced by using an array processor, which is discussed next.

16.4.2 Array Processing

Array processing is a technique in which arithmetic operations are done on large arrays of data. Array processing also works on vectors of data, as explained in the previous section. In general, to support array processing array processors are used in a computer.Array processors perform operations on vectors, but they operate on large data sets at the same time.

Array processors are of two types:

1) Attached array processors

2) SIMD array processors

Let us discuss these two array processors in detail:

1) Attached Array Processors:

Attached array processors are used to aid host computer as an auxiliary processor or co-processor to perform array operations. Host computer can be a general-purpose computer that deals with simple computations and operations. When this general-purpose computer needs to operate on complex data like arrays, an array processor comes into play.

An array processor is interfaced with its host computer as follows:

1) I/O interface with host.

2) Interfaced to main memory of host.

Since array processor is attached to host computer through I/O interface, host computer treats it as a peripheral device.

2) SIMD array processors:

SIMD stands for Single Instruction Multiple Data,i.e. in a single instruction they process multiple data. This is achieved by having multiple processing elements working in parallel.All the processing elements are controlled using a single controller. Thus, a single instruction is sent to all the processing units, which are fed with different data by their local memory.

Every processing unit contains three elements:

a) Arithmetic and Logical Unit (ALU)

b) Floating point arithmetic unit

c) Working registers

Control unit stores the instructions while processing units store operands.

Example:A vector operation $C_i = A_i \times B_i$,where$i=1,2,3\ldots$is to be performed using SIMD Array processor.

The array processor will perform the following steps, for execution of the instruction as given above:

Step1:Control unit checks whether the data is scalar or vector.If data is scalar, then control unit performs the operations directly in the master processing unit.If data is vector it sends data to Processing Units (PUs) and the following steps are performed:

Step2: Control unit sends data to processing units such that PU_1 stores operands A_1 and B_1, PU_2 stores operands A_2 and B_2 and so on.

Step3: Finally, control unit send instruction to multiply the operands to all the PUs.Thus, all the data is processed concurrently in the array processors.

Depending on the length of the vector, the control unit decides the number of PUs that should be activated. For example, if the length of vector to be processed is 32, then only 32 PUs are activated, and all other PUs are masked. In case, the length of a vector is more than the number of PUs, then all the PUs are activated and control unit uses these units as per requirements.

☞Check Your Progress – 2

1) **State True or False**

	T/F
i) An Array processor performs operations on vector data	☐
ii) SIMDarray processors contains a single processing unit and multiple control	☐
iii) Masking is a technique of activating Processor Units (PU) in SMID Array Processor.	☐
iv) Throughput of Vector Processor is higher than Scalar Processor	☐

2) What are the components present in Processor Unit in an SIMD Array Processor?

……

……

……

……

3) Where can you use Vector Processing?

……

……

……

……

16.5 MULTI PROCESSORS

Multiprocessor systems refer to those systems that use multiple processors.These processors execute multiple tasks by sharing them on multiple processors simultaneously. You can think of multiprocessor as a system consisting of different processing units working together.

At the operating system level, multiprocessing can be referred as processing multiple processes simultaneously, with each process or task being executed by different cores. This is different from single processor executionflow.Major difference between multiprocessor and single processor flow comes from the fact that, in multiprocessor systems single task is executed by multiple cores, whereas, in single processor system multiple tasks are executed concurrently over a single core. This is called as multi-tasking.Sometimes multi-tasking is being confused with multi-processor, it may be noted that multi-tasking uses single processor but switches among the tasks on completion of a time slice allotted to a specific task. The Figure 16.4 describes the use of multi processors (CPU) while using the common shared memory.

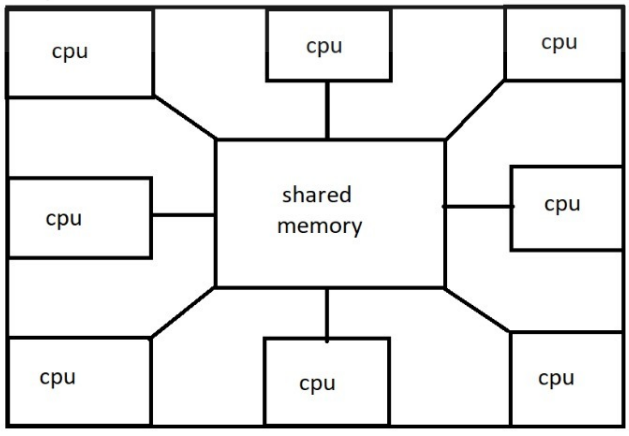


Figure 16.4: Multiprocessors with common shared memory

16.5.1 Characteristics of Multiprocessor

Multi-processor is a tightly coupled system with more than one CPU sharing same memory and input-output resources. Some of the major characteristics of multiprocessor are as discussed below:

- Processors used in multiprocessor system can be eitherCentralProcessing Units (CPUs) or Input-Output Processors (IOPs).
- They are categorized as Multiple Instruction Multiple Data(MIMD) systems.
- Major difference between a multiprocessor and a multicomputer system is, in multi computer system are connected with one another to form a network and they may or may not communicateinformation among them.
- A multiprocessor system may be designed to have a singlemaster Operating System. This operating system may be responsible for communication between the processors, other peripheralsand shared memory of the system.
- Multiprocessing can improve thereliability of the system such that when a failure happens, it affects only that processor, therefore, it has very less impact on rest of the processors.
- If one of the processors has failed, then some other processor can take up the duties of the faulty processor.
- Multiprocessing will enhance performance by breaking down a program into parallel executable tasks. Multiprocessors are flexible,i.e. the user can explicitly declare which processes are to be executed in parallel.
- Other effective way to improve performance of the system is to use a compiler which can identify parallelism in a process automatically.
- Compiler also tries to debug for data dependency in the program, which may cause issues while executing a task in parallel.
- Two threads of a task, which use different data can run concurrently.
- Multiprocessor systems can be implemented as ~~barely~~ loosely coupled systems as well.Each element of a processor in ~~barely~~ loosely coupled system has its own independent local memory.

16.5.2 Inter-Connection Structures

Every processor in a multi-processor system has a set of components, namely:

- CPU
- Shared Memory
- I/O

These components should communicate with each other for proper execution of programs. These components communicate among themselves through some interconnect paths;these paths which connects these modules or components are called as interconnection structures.

Let’s discuss various interconnection structures in the detail.

Multi-port memory Organization

The multiport memory organization is illustrated in Figure 16.5. The multi-core processors have been using the shared memory resource modules (MM1, MM2, MM3, MM4) with the bus interconnection, which is used as a path for communicating among them. In this interconnection structure every CPU can communicate with every memory module. The advantage of such kind of connection is that, in case every CPU is trying to access a different memory module, they will be allowed access in parallel. However, their can be conflict, if to processor are trying to access the same memory module at the same time. A detailed discussion on this topic is beyond the scope of this Unit.

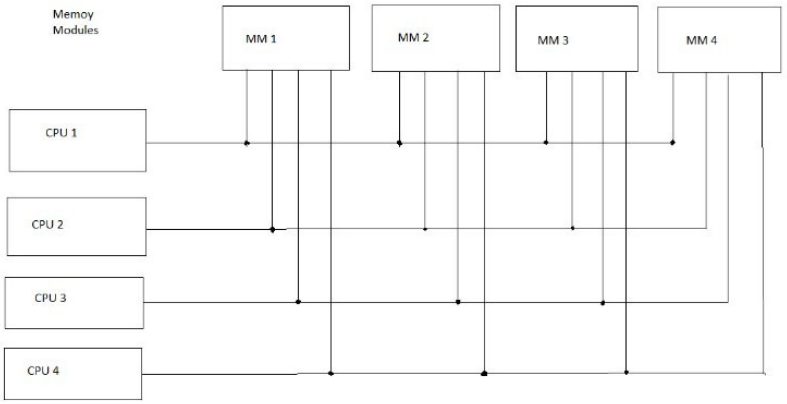


Figure 16.5: Multi-port memory Organization

Bus Interconnection Structure

A bus is used as a communication path for connecting two or more processors or devices. Bus is ashared transmission medium; this is a major advantage of bus interconnects. Bus has already been explained in the Block 2 of this course. In this section, let us discuss the serial arbitration procedure used in the buses, in the context of priority scheme for bus allocation. Figure 16.6 shows the serial arbitration process.

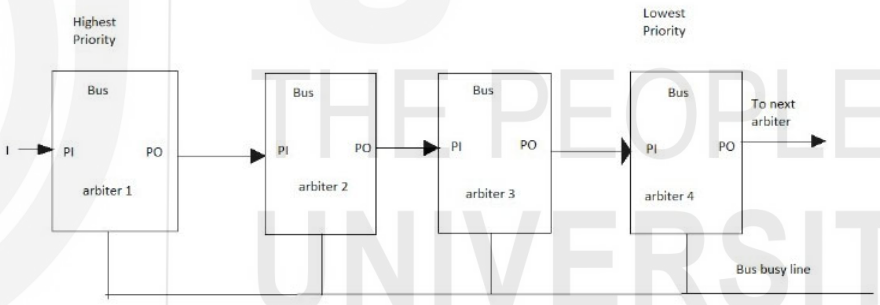


Figure 16.6: Serial Arbitration Procedure

In the above figure, Serial Arbitration procedure is illustrated. The arbiters are sharing a common bus busy line which is synchronized to maintain the synchronous communication. The input (l) is serially transmitted through the arbiters with the enabling of bus busy line. It might depend on the clock edge occurrence of bus busy line to transmit the input data through the arbiters.

16.6 INTER PROCESSOR COMMUNICATION AND SYNCHRONIZATION

In this section we will be discussing about Inter process communication facilities. Various applications of inter-process and inter-thread communication facilities, which uses data transfer are:

- Pipes (named, dynamic – shell or process generated)

- TCP/IP socket communication (named, dynamic - loop back interface or network interface)
- D-Bus is an IPC mechanism offering one to many broadcast and subscription facilities between processes.
- Shared memory
- Between Processes
- Between Threads (global memory)

Figure 16.7 shows and Figure 16.8 shows two simple mechanisms of inter-processor communication: message passing and shared memory.

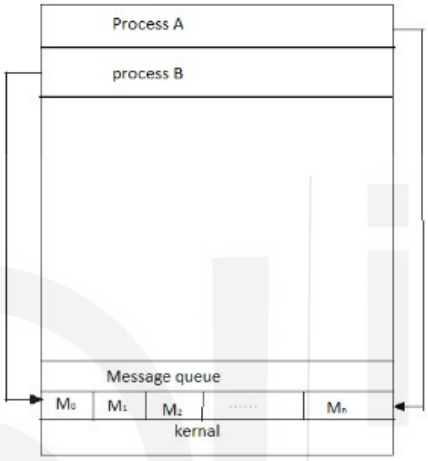


Figure 16.7: Message Passing

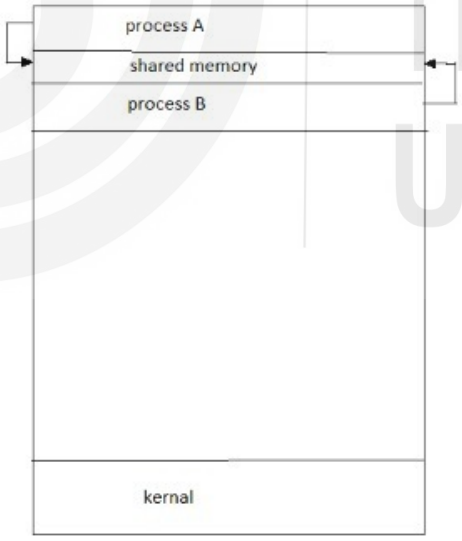


Figure 16.8: Shared Memory

The above figure illustrates the Inter-processor communication in two possible ways. The first one is with respect to two processes A and B trying to pass messages to the message queue. It depends on the algorithm used, which decides on which process gets to pass the message first.

The second figure indicates the two processes A and B trying to use the common memory and ultimately depends on the algorithm designed, which decides how the processes share the memory.

When two processes need multiple shared resources at the same time in order to proceed further a **Deadlock** condition occurs.

To understand this condition, let us consider a scenario in which Thread X is expecting data from Thread Y for further processing and Thread Y is expecting data from Thread X for further processing of data. This scenario is called deadlock and no further processing can be done between the two threads.

Operating System provides synchronization and communications between processes sharing resources and prevents them from facing potential Inter process communication problems.

16.7 CACHE COHERENCE

In multi-processor system, while processing the shared data, various processors can store this shared data in multiple local caches for faster local processing. Therefore, there is a chance of having caches with incoherent data. This can lead to non-uniformity in shared resources. This problem is predominant in the case of multi-processors that use multiple processors and have shared data.

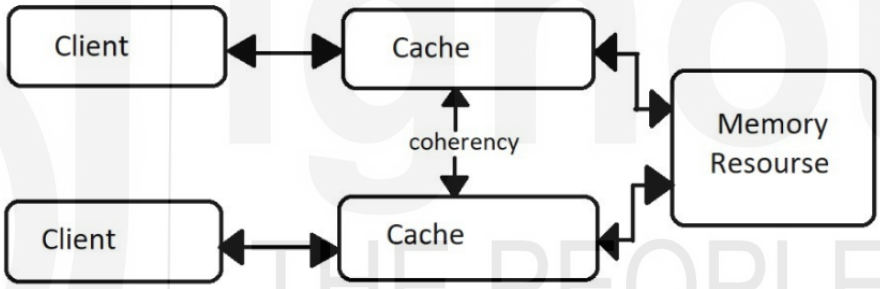


Figure 16.9: Cache Coherence

In the Figure 16.9, there are two client processors having their own local cache memory, which are initiallycoherent. Next, one of the client processors updatesits cache, which in turn changes the data in the shared memory. The other client processor should also update its cache memory to reflect the change made by the first client processor. This is how cache coherence works. However, what happens if the other processor is not able to update its cache. In this case, cache coherence is not present. Thus, the update of data made by one processor though is reflected in its cache and shared memory, but the other client processor has no clue about it. There will be data conflict between both these client processors. To overcome such conflicts and synchronize data between multiple caches, Cache Coherence protocols are used. The detailed discussion on these protocols is beyond the scope of this Unit.

16.8 MULTI-CORE PROCESSORS

Multi-core processor is adesign in which a number of cores are integrated on a single chip. These cores can be treated as processing units. They can process data and execute programs, similar to that of any other processor. It can be considered as a system having multiple processors. This system executes normal instructions such as add, subtract, and branch instructions. But these instructions can be executed on various cores at the same time. This results in drastic increase in speed by giving scope to multithreading and parallel computing.

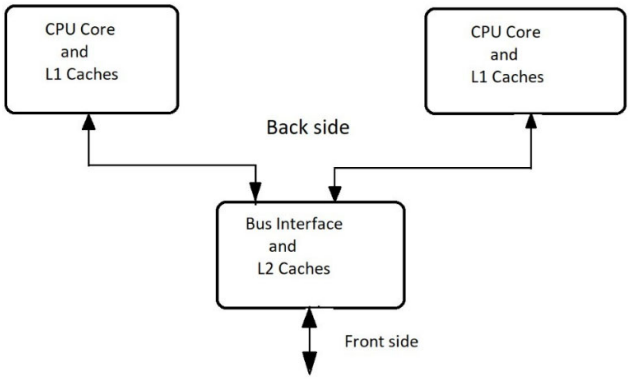


Figure 16.10: Multi-core Processor block diagram

Figure 16.10 shows the basic architecture of cores in a multi-core processor. Applications of Multi-core processors are in various domains such as embedded systems, networks, Digital Signal Processing (DSP) and other domains.

Performance of Multi-core processors depends mostly on algorithms and methodologies used to implement them. In general, a good multi-core processor requires a strong support from its operating system so that all the cores of the processor are efficiently used. This Software parallelization is one of the areas of the present research.

☞ Check Your Progress – 3

1. Explain Inter-processor Communication with various techniques.
   ……
   ……
   ……
   ……

2. Explain about Cache Coherence
   ……
   ……
   ……
   ……

3. Give some examples on Multi core processors
   ……
   ……
   ……
   ……

16.9 SUMMARY

This Unit provides a basic introduction pf the concepts of pipelining and multiprocessor. The two pipelining techniques discussed in this Unit are – arithmetic pipeline and instruction pipeline. Various Parallelism techniques such as Vector and array processing are also discussed in this unit, and further topics like Multiprocessors are demonstrated with diagrams and explained in brief.

The information given on various topics such as pipelining, both arithmetic and Instruction pipeline, is introductoryand can be supplemented with additional reading. In fact, a course in an area of computer must be supplemented by further reading to keep your knowledge up to date, as the computer world is changing with leaps and bounds. In addition to further readings the student is advised to study several Indian Journals on computers to enhance her/hisknowledge.

16.10 SOLUTIONS

Check your progress 1:

1.

i) False
ii) True
iii) False
iv) False
v) True

2. Number of stages in Pipeline is 'k' =7
Number of instructions is 'n' = 180
If each clock cycle is tp
Then the number of clock cycles taken by the processer to process 180 tasks in a 7-segment pipeline is = (k+n-1)tp
= (7+(180-1))tp
=186tp
Number of clock cycles for pipelined processor = 186

3.

- In high-speed Computers
- For doing floating point arithmetic
- For processing data of scientific problems

Check your progress – 2

1.

i) True
ii) False
iii) False
iv) True

2. Processing Unit contains the following components:

- ALU
- Floating Point Unit
- Registers

3. Vector processing is used in following fields:

- Weather forecasting
- Artificial Intelligence (AI)
- Image Processing
- Healthcare and diagnosis

Check your progress- 3

1. The inter processor communication can be achieved either through a shared memory area or through a queue. However, in both the cases software algorithms must control the sequence and synchronization of such communications.

2. In the case of multi-processors having shared memory and local caches, it is possible that a shared data item may be present in several local cache of the processors. In case, any of these processor changes the value of the shared data in it local cache, then it should result in update of this data in the shared memory and all other local caches. The coherence of cache is a required feature to keep the computation error free.

3. These days due to ULSI technologies most of the processors are multi-core processors. The Intel latest processors, AMD processor, processors of mobiles etc. all have multi-cores.

MCS-203 Operating Systems - Exam Preparation Roadmap

BLOCK 1: Introduction to OS & Process Management

- CPU Scheduling:
  - FCFS, SJF, Round Robin (with Quantum)
  - Preemptive vs Non-preemptive
  - Gantt charts, Waiting time, Turnaround time, Throughput
- Deadlocks:
  - Four necessary conditions
  - Resource Allocation Graph (RAG)
  - Deadlock Prevention, Avoidance (Banker's Algorithm), Detection
- Synchronization & Critical Section:
  - Critical Section Problem, Entry & Exit section
  - Dekker's, Bakery's, Peterson's Algorithm
  - Semaphores: Binary, Counting
  - Reader-Writer Problem
- Functions of OS:
  - Process Management, Memory, File, I/O, Protection, Networking, CLI
- Process Management:
  - Process states and transitions
  - Process Control Block (PCB), Threads
  - Types of schedulers: Short-term, Long-term, Medium-term

BLOCK 2: Memory, File & I/O Management

- Virtual Memory:
  - Logical vs Physical Address
  - Paging, Segmentation
  - Address Translation using TLB
- Demand Paging:
  - Page faults, Page Table, Valid/Invalid Bit
- Page Replacement Algorithms:
  - FIFO, LRU, Optimal - with numerical examples
- File Management:
  - File attributes, operations, directory structures (single, two-level, tree)
  - File Allocation: Contiguous, Linked, Indexed
- Disk Scheduling:
  - FCFS, SSTF, SCAN, LOOK
- Swapping vs Overlay:
  - Memory fragmentation (internal/external)

BLOCK 3: Advanced OS Topics

MCS-203 Operating Systems - Exam Preparation Roadmap

- Distributed Systems:
  - Models: Client/Server, Distributed Objects, Shared Memory
  - Transparency types: Location, Access, Replication, Concurrency
- Synchronization in Multiprocessors:
  - Test-and-Set, Compare-and-Swap, Spinlocks
- Remote Procedure Call (RPC):
  - Marshalling, Stubs, Communication between client and server

BLOCK 4: Case Studies - Android, iOS, Linux, Windows

- Android Architecture:
  - Layers: Linux Kernel, Libraries, Application Framework, Apps
  - Features: Security, File System, Memory Mgmt
- iOS Architecture:
  - Layers: Cocoa Touch, Media, Core Services, Core OS
  - Evolution, Features, Application Lifecycle
- Linux:
  - User management, Permissions, Encrypted Storage
  - Kernel components: Scheduler, Memory, File system
- Windows 10:
  - Process & Memory Management, Inter-process Communication
  - System calls, Scheduling

Additional High-Probability Topics (Expected in Upcoming Exams)

- System Calls:
  - Definition, Types: Process Control, File Management, Device Management
- Operating System Structures:
  - Monolithic, Layered, Microkernel, Modules, Hybrid
- User Mode vs Kernel Mode:
  - Privilege levels, Transitions, Use cases
- Protection & Security Models:
  - Access Matrix, Capability Lists, ACL, Authentication
- Booting Process:
  - BIOS, Bootloader, Kernel loading, Init/systemd
- File System Implementation:
  - Inodes, Superblocks, Mounting, Journaling
- Real-Time Operating Systems (RTOS):
  - Characteristics, Examples, Scheduling constraints
- Mobile OS Design Challenges:
  - Power Efficiency, UI Responsiveness, App Isolation
- Command Interpreters and Shells:

- Role of shells (Bash, C Shell), CLI vs GUI