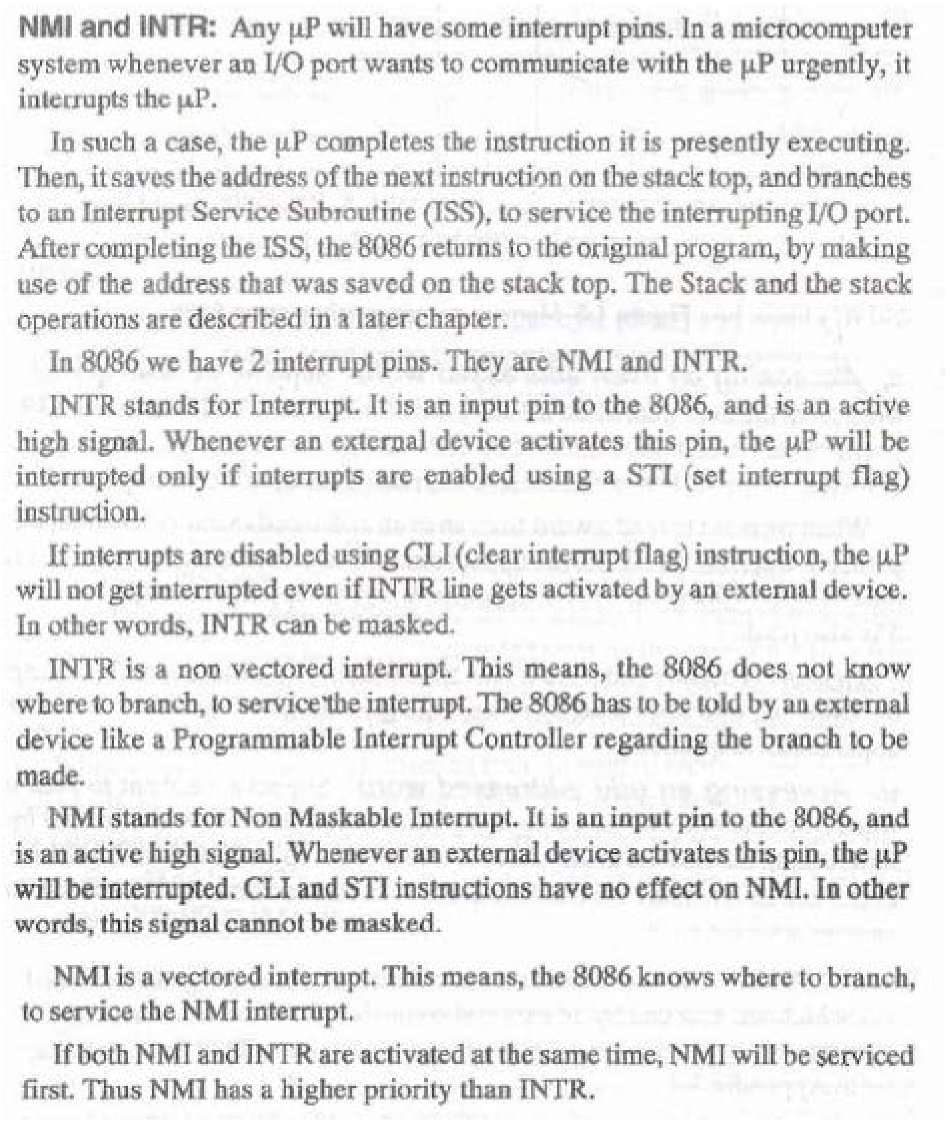
Q.2 a. Explain the action of 8086 when NMI and INTR pins are activated. (8)

Answer:



NMI and INTR: Any µP will have some interrupt pins. In a microcomputer system whenever an I/O port wants to communicate with the µP urgently, it interrupts the µP.

In such a case, the µP completes the instruction it is presently executing. Then, it saves the address of the next instruction on the stack top, and branches to an Interrupt Service Subroutine (ISS), to service the interrupting I/O port. After completing the ISS, the 8086 returns to the original program, by making use of the address that was saved on the stack top. The Stack and the stack operations are described in a later chapter.

In 8086 we have 2 interrupt pins. They are NMI and INTR.

INTR stands for Interrupt. It is an input pin to the 8086, and is an active high signal. Whenever an external device activates this pin, the µP will be interrupted only if interrupts are enabled using a STI (set interrupt flag) instruction.

If interrupts are disabled using CLI (clear interrupt flag) instruction, the µP will not get interrupted even if INTR line gets activated by an external device. In other words, INTR can be masked.

INTR is a non vectored interrupt. This means, the 8086 does not know where to branch, to service the interrupt. The 8086 has to be told by an external device like a Programmable Interrupt Controller regarding the branch to be made.

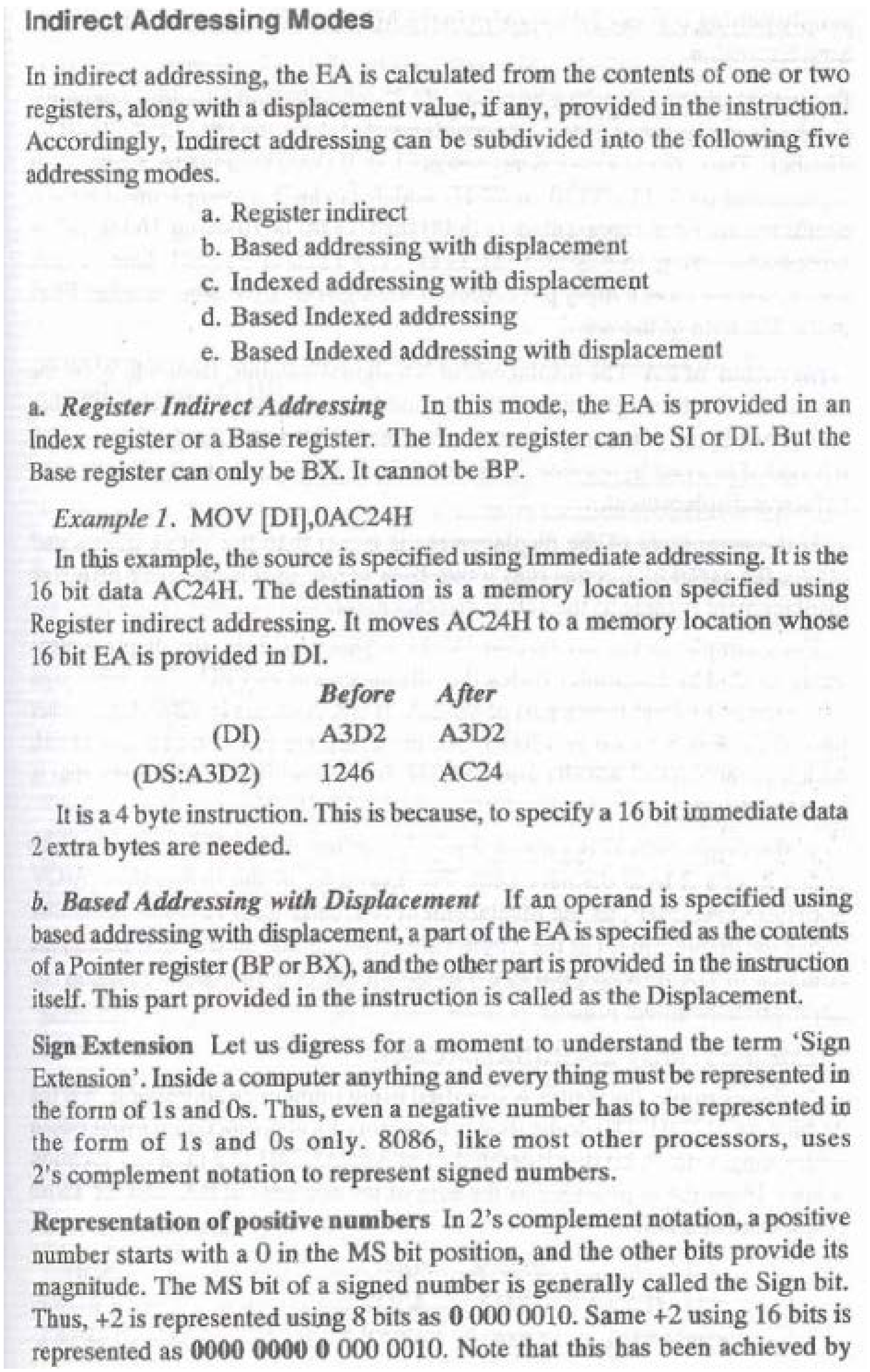
NMI stands for Non Maskable Interrupt. It is an input pin to the 8086, and is an active high signal. Whenever an external device activates this pin, the µP will be interrupted. CLI and STI instructions have no effect on NMI. In other words, this signal cannot be masked.

NMI is a vectored interrupt. This means, the 8086 knows where to branch, to service the NMI interrupt.

If both NMI and INTR are activated at the same time, NMI will be serviced first. Thus NMI has a higher priority than INTR.

b. Explain with examples indirect addressing modes available in microprocessor 8086. (8)

Answer:



Indirect Addressing Modes

In indirect addressing, the EA is calculated from the contents of one or two registers, along with a displacement value, if any, provided in the instruction. Accordingly, Indirect addressing can be subdivided into the following five addressing modes.

a. Register indirect
b. Based addressing with displacement
c. Indexed addressing with displacement
d. Based Indexed addressing
e. Based Indexed addressing with displacement

a. ***Register Indirect Addressing*** In this mode, the EA is provided in an Index register or a Base register. The Index register can be SI or DI. But the Base register can only be BX. It cannot be BP.

Example 1. MOV [DI],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Register indirect addressing. It moves AC24H to a memory location whose 16 bit EA is provided in DI.

	Before	*After*
(DI)	A3D2	A3D2
(DS:A3D2)	1246	AC24

It is a 4 byte instruction. This is because, to specify a 16 bit immediate data 2 extra bytes are needed.

b. ***Based Addressing with Displacement*** If an operand is specified using based addressing with displacement, a part of the EA is specified as the contents of a Pointer register (BP or BX), and the other part is provided in the instruction itself. This part provided in the instruction is called as the Displacement.

Sign Extension Let us digress for a moment to understand the term 'Sign Extension'. Inside a computer anything and every thing must be represented in the form of 1s and 0s. Thus, even a negative number has to be represented in the form of 1s and 0s only. 8086, like most other processors, uses 2's complement notation to represent signed numbers.

Representation of positive numbers In 2's complement notation, a positive number starts with a 0 in the MS bit position, and the other bits provide its magnitude. The MS bit of a signed number is generally called the Sign bit. Thus, +2 is represented using 8 bits as **0** 000 0010. Same +2 using 16 bits is represented as **0000 0000 0** 000 0010. Note that this has been achieved by simply putting copies of the sign bit in the MS byte. This process is known as Sign Extension.

Representation of negative numbers In 2's complement notation, a negative number is represented as the 2's complement of the same magnitude positive number. Thus, because +2 is represented as 0 000 0010 using 8 bits, -2 is represented as 1 111 1110 or FEH, which is the 2's complement of +2. Similarly, as +2 is represented as 0000 0000 0000 0010 using 16 bits, -2 is represented using 16 bits as 1111 1111 1111 1110 or FFFEH. Once again, notice that we have simply put copies of the sign bit of the 8 bit number FEH in the MS byte of the word.

Generation of EA The displacement is a signed number. Depending on the magnitude of the displacement, the assembler codes the displacement using one or two bytes. If the displacement is a small value in the range -128 to 127, it is coded as a one byte value. This value sign extended to 16 bits is called the Effective displacement.

If the magnitude of the displacement is larger than the above mentioned range, the assembler codes it as a two byte value. In this case, the effective displacement is same as the actual displacement.

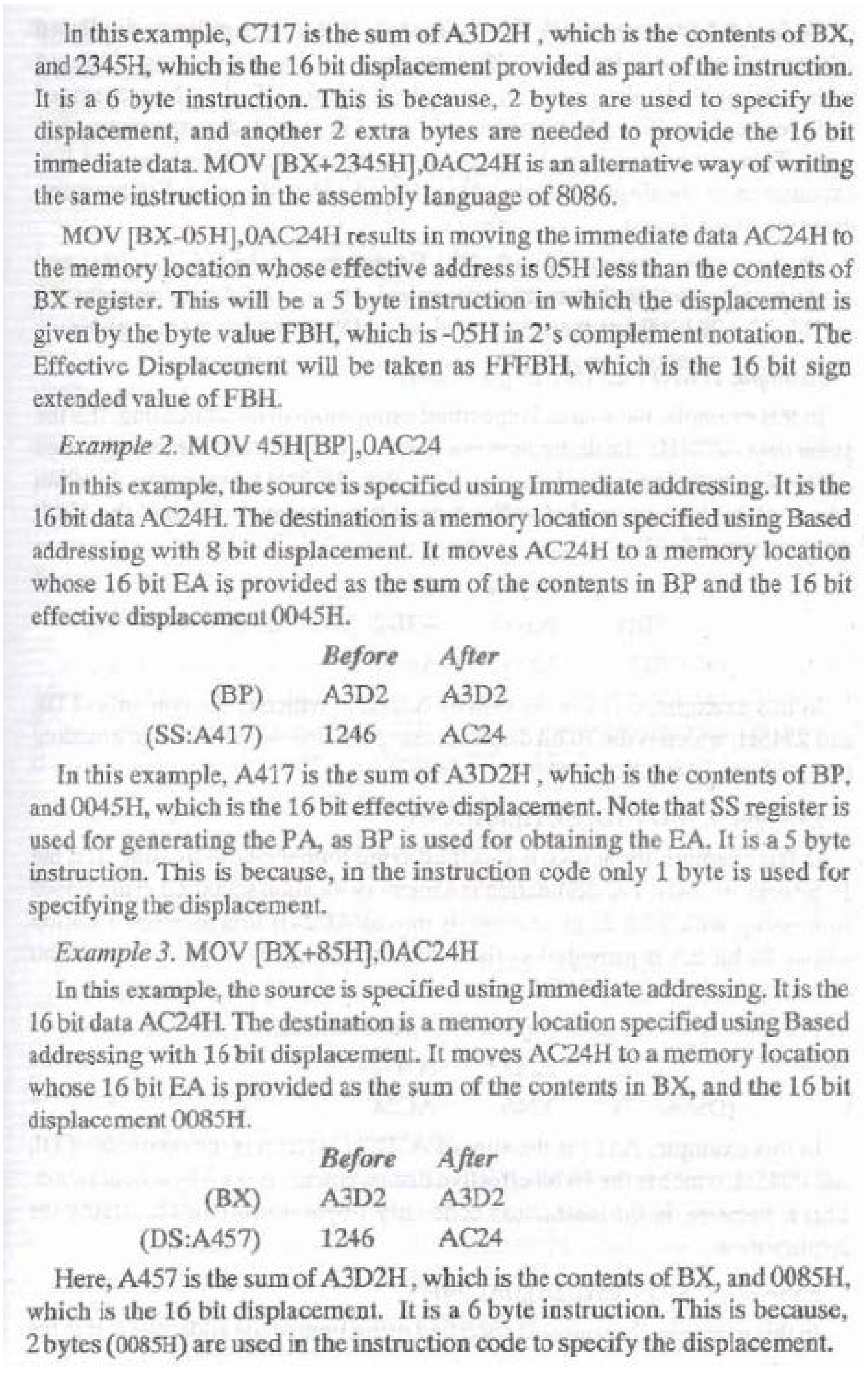
For example, in the instruction 'MOV -2[BX],AC24H' the displacement value is -2. The assembler codes this displacement as FEH. This FEH sign extended as FFFEH forms part of the EA. If BX contents is 4200H, the other part of the EA is taken as 4200H. So, the complete EA is taken as 41FEH, which is the sum of 4200H and FFFEH. In this addition, any carry that is generated is ignored.

If the displacement is greater than 127, or less than -128, the assembler codes it as a 2 byte displacement. For example, in the instruction 'MOV 300H[BX],AC24H', as the displacement is greater than 127, the assembler codes the displacement as the 2 byte value 0300H. The sum of 0300H and the contents of BX provides the EA. The 20 bit PA is then generated using the appropriate Segment register.

Example 1. MOV 2345H[BX],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based addressing with 16 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in BX, and the 16 bit displacement 2345H.

	Before	*After*
(BX)	A3D2	A3D2
(DS:C717)	1246	AC24



In this example, C717 is the sum of A3D2H , which is the contents of BX, and 2345H, which is the 16 bit displacement provided as part of the instruction. It is a 6 byte instruction. This is because, 2 bytes are used to specify the displacement, and another 2 extra bytes are needed to provide the 16 bit immediate data. MOV [BX+2345H],0AC24H is an alternative way of writing the same instruction in the assembly language of 8086.

MOV [BX-05H],0AC24H results in moving the immediate data AC24H to the memory location whose effective address is 05H less than the contents of BX register. This will be a 5 byte instruction in which the displacement is given by the byte value FBH, which is -05H in 2's complement notation. The Effective Displacement will be taken as FFFBH, which is the 16 bit sign extended value of FBH.

Example 2. MOV 45H[BP],0AC24

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based addressing with 8 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in BP and the 16 bit effective displacement 0045H.

	Before	***After***
(BP)	A3D2	A3D2
(SS:A417)	1246	AC24

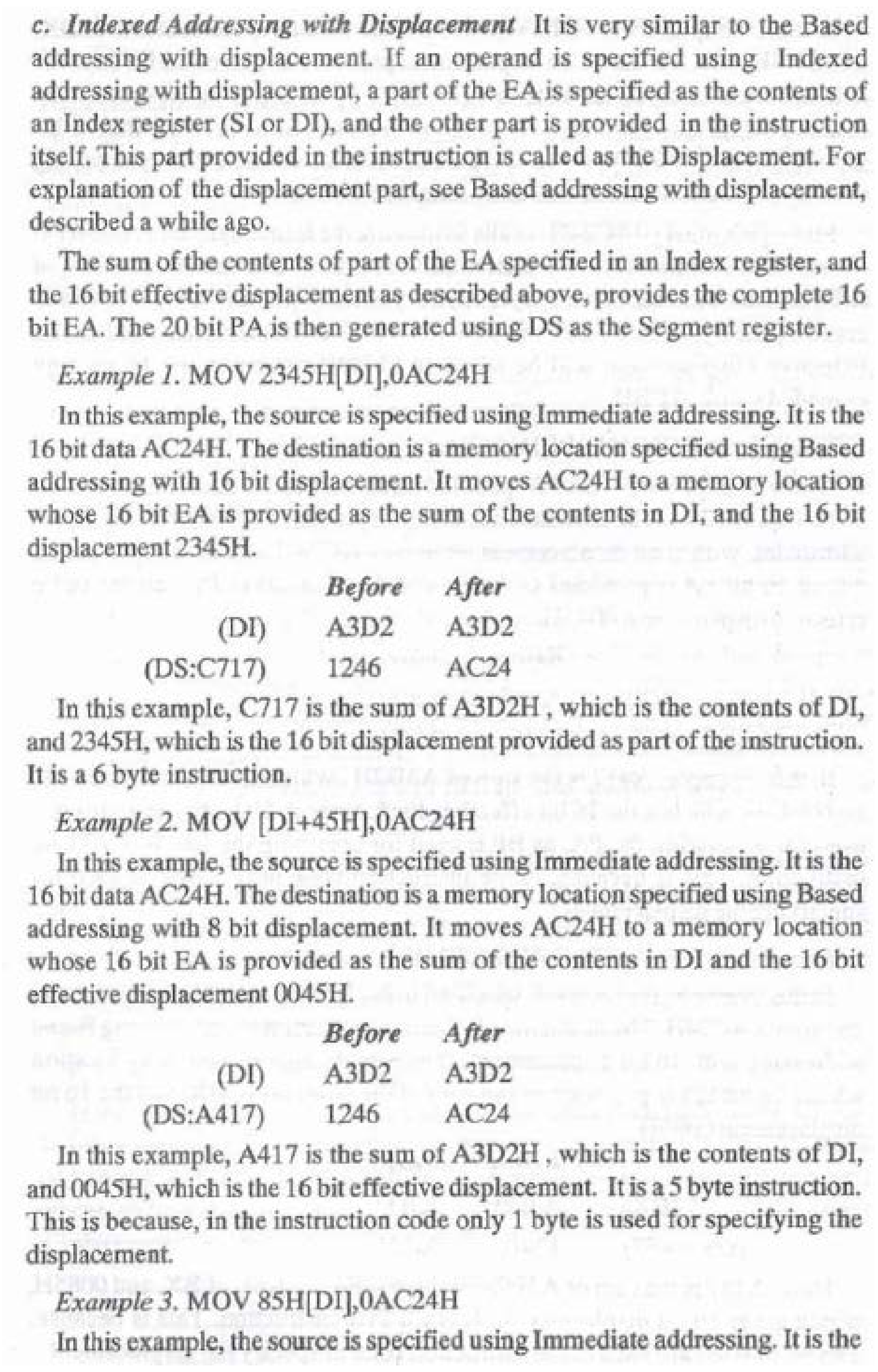
In this example, A417 is the sum of A3D2H , which is the contents of BP, and 0045H, which is the 16 bit effective displacement. Note that SS register is used for generating the PA, as BP is used for obtaining the EA. It is a 5 byte instruction. This is because, in the instruction code only 1 byte is used for specifying the displacement.

Example 3. MOV [BX+85H],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based addressing with 16 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in BX, and the 16 bit displacement 0085H.

	Before	***After***
(BX)	A3D2	A3D2
(DS:A457)	1246	AC24

Here, A457 is the sum of A3D2H , which is the contents of BX, and 0085H, which is the 16 bit displacement. It is a 6 byte instruction. This is because, 2 bytes (0085H) are used in the instruction code to specify the displacement.



c. Indexed Addressing with Displacement It is very similar to the Based addressing with displacement. If an operand is specified using Indexed addressing with displacement, a part of the EA is specified as the contents of an Index register (SI or DI), and the other part is provided in the instruction itself. This part provided in the instruction is called as the Displacement. For explanation of the displacement part, see Based addressing with displacement, described a while ago.

The sum of the contents of part of the EA specified in an Index register, and the 16 bit effective displacement as described above, provides the complete 16 bit EA. The 20 bit PA is then generated using DS as the Segment register.

Example 1. MOV 2345H[DI],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based addressing with 16 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI, and the 16 bit displacement 2345H.

	Before	*After*
(DI)	A3D2	A3D2
(DS:C717)	1246	AC24

In this example, C717 is the sum of A3D2H , which is the contents of DI, and 2345H, which is the 16 bit displacement provided as part of the instruction. It is a 6 byte instruction.

Example 2. MOV [DI+45H],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based addressing with 8 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI and the 16 bit effective displacement 0045H.

	Before	*After*
(DI)	A3D2	A3D2
(DS:A417)	1246	AC24

In this example, A417 is the sum of A3D2H , which is the contents of DI, and 0045H, which is the 16 bit effective displacement. It is a 5 byte instruction. This is because, in the instruction code only 1 byte is used for specifying the displacement.

Example 3. MOV 85H[DI],0AC24H

In this example, the source is specified using Immediate addressing. It is the

16 bit data AC24H. The destination is a memory location specified using Based addressing with 16 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI, and the 16 bit displacement 0085H.

	Before	*After*
(DI)	A3D2	A3D2
(DS:A457)	1246	AC24

In this example, A457 is the sum of A3D2H , which is the contents of DI, and 0085H, which is the 16 bit displacement. It is a 6 byte instruction. This is because, 2 bytes (0085H) are used in the instruction code to specify the displacement.

d. Based Indexed Addressing In this addressing mode, the EA is provided partly in a Base register (BX or BP), and partly in an Index register (SI or DI). Their sum provides the EA. In this mode, displacement is not provided as a part of the instruction. The final 20 bit PA is generated using the appropriate Segment Register.

Example 1. MOV [DI][BX],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based Indexed addressing. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI and BX.

	Before	*After*
(DI)	A3D2	A3D2
(BX)	7234	7234
(DS:1606)	1246	AC24

In this example, 1606 is the sum of A3D2H , which is the contents of DI, and 7234H which is the contents of BX. The carry generated in this addition is ignored. It is a 4 byte instruction. This is because, there is no displacement in the instruction. MOV [BX+DI],0AC24H is an alternative way of writing the same instruction.

Example 2. MOV [DI+BP],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based Indexed addressing. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI and BP.

If DI contents is A3D2H and BP contents is 1234H, the EA is their sum, which is B606H. This is also a 4 byte instruction. Note that SS is used for generating the PA.

	Before	*After*
(DI)	A3D2	A3D2
(BP)	1234	1234
(SS:B606)	1246	AC24

e. Based Indexed Addressing with Displacement It is same as Based Indexed addressing, with displacement of 8 bits or 16 bits, also provided as part of the instruction.

*Example 1.*MOV 37H[DI][BX],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based Indexed addressing with 8 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI, BX, and the 16 bit effective offset of 0037H.

	Before	*After*
(DI)	A3D2	A3D2
(BX)	7234	7234
(DS:163D)	1246	AC24

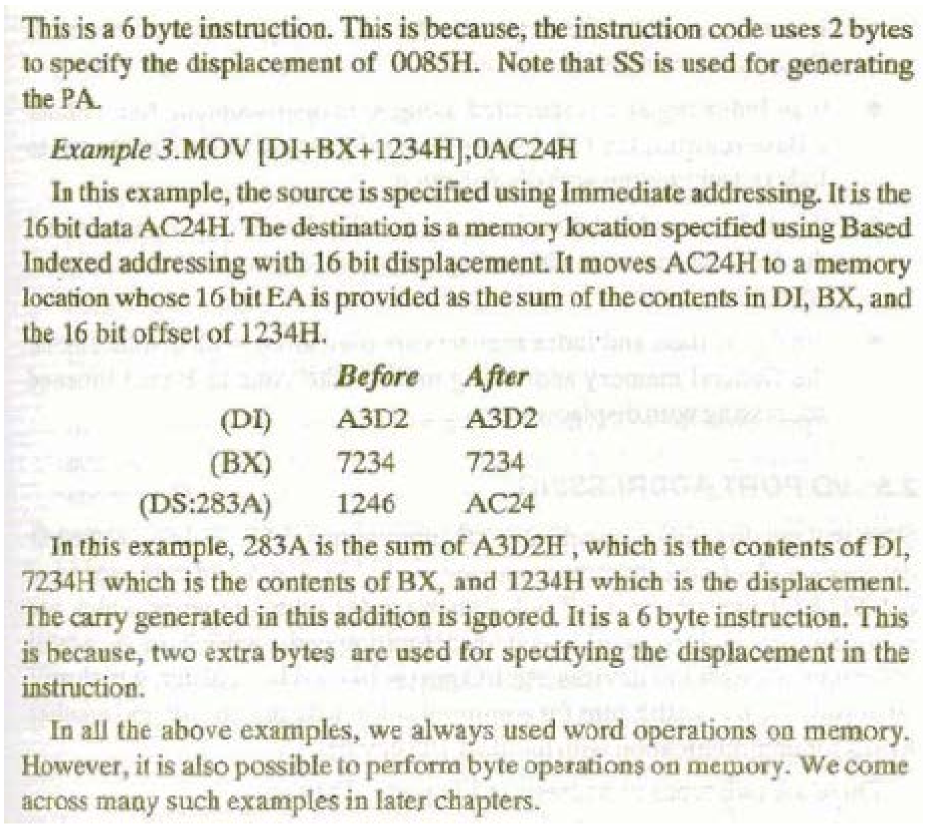
In this example, 163D is the sum of A3D2H , which is the contents of DI, 7234H which is the contents of BX, and 0037H which is the effective displacement. The carry generated in this addition is ignored. It is a 5 byte instruction. This is because, one extra byte is used for specifying the displacement in the instruction. MOV [BX+DI+37H],0AC24H is an alternative way of writing the same instruction.

Example 2. MOV [DI+BP+85H],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based Indexed addressing with 16 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI, BP, and the 16 bit displacement of 0085H.

	Before	*After*
(DI)	A3D2	A3D2
(BP)	1234	1234
(SS:B68B)	1246	AC24

In this example, B68B is the sum of A3D2H , which is the contents of DI, 1234H which is the contents of BP, and 0085H which is the 16 bit displacement.


This is a 6 byte instruction. This is because, the instruction code uses 2 bytes to specify the displacement of 0085H. Note that SS is used for generating the PA.

*Example 3.*MOV [DI+BX+1234H],0AC24H

In this example, the source is specified using Immediate addressing. It is the 16 bit data AC24H. The destination is a memory location specified using Based Indexed addressing with 16 bit displacement. It moves AC24H to a memory location whose 16 bit EA is provided as the sum of the contents in DI, BX, and the 16 bit offset of 1234H.

	Before	***After***
(DI)	A3D2	A3D2
(BX)	7234	7234
(DS:283A)	1246	AC24

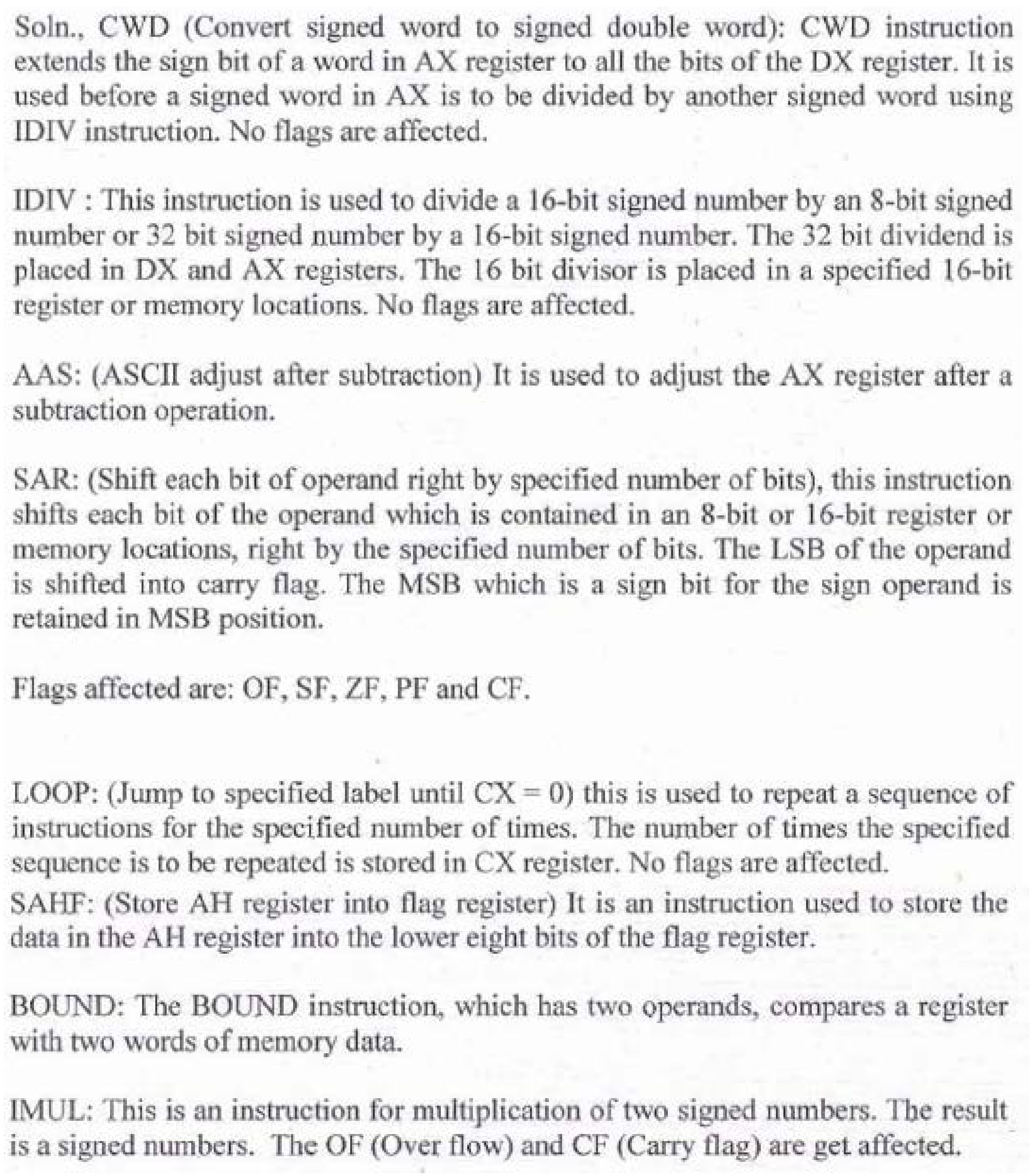
In this example, 283A is the sum of A3D2H , which is the contents of DI, 7234H which is the contents of BX, and 1234H which is the displacement. The carry generated in this addition is ignored. It is a 6 byte instruction. This is because, two extra bytes are used for specifying the displacement in the instruction.

In all the above examples, we always used word operations on memory. However, it is also possible to perform byte operations on memory. We come across many such examples in later chapters.

Q.3 a. Explain following instructions in 8086 family and their effect on flag.

(i) CWD
(ii) IDIV
(iii) AAS
(iv) SAR
(v) LOOP
(vi) SAHF
(vii) BOUND
(viii) IMUL **(12)**

Answer:


Soln., CWD (Convert signed word to signed double word): CWD instruction extends the sign bit of a word in AX register to all the bits of the DX register. It is used before a signed word in AX is to be divided by another signed word using IDIV instruction. No flags are affected.

IDIV : This instruction is used to divide a 16-bit signed number by an 8-bit signed number or 32 bit signed number by a 16-bit signed number. The 32 bit dividend is placed in DX and AX registers. The 16 bit divisor is placed in a specified 16-bit register or memory locations. No flags are affected.

AAS: (ASCII adjust after subtraction) It is used to adjust the AX register after a subtraction operation.

SAR: (Shift each bit of operand right by specified number of bits), this instruction shifts each bit of the operand which is contained in an 8-bit or 16-bit register or memory locations, right by the specified number of bits. The LSB of the operand is shifted into carry flag. The MSB which is a sign bit for the sign operand is retained in MSB position.

Flags affected are: OF, SF, ZF, PF and CF.

LOOP: (Jump to specified label until CX = 0) this is used to repeat a sequence of instructions for the specified number of times. The number of times the specified sequence is to be repeated is stored in CX register. No flags are affected.

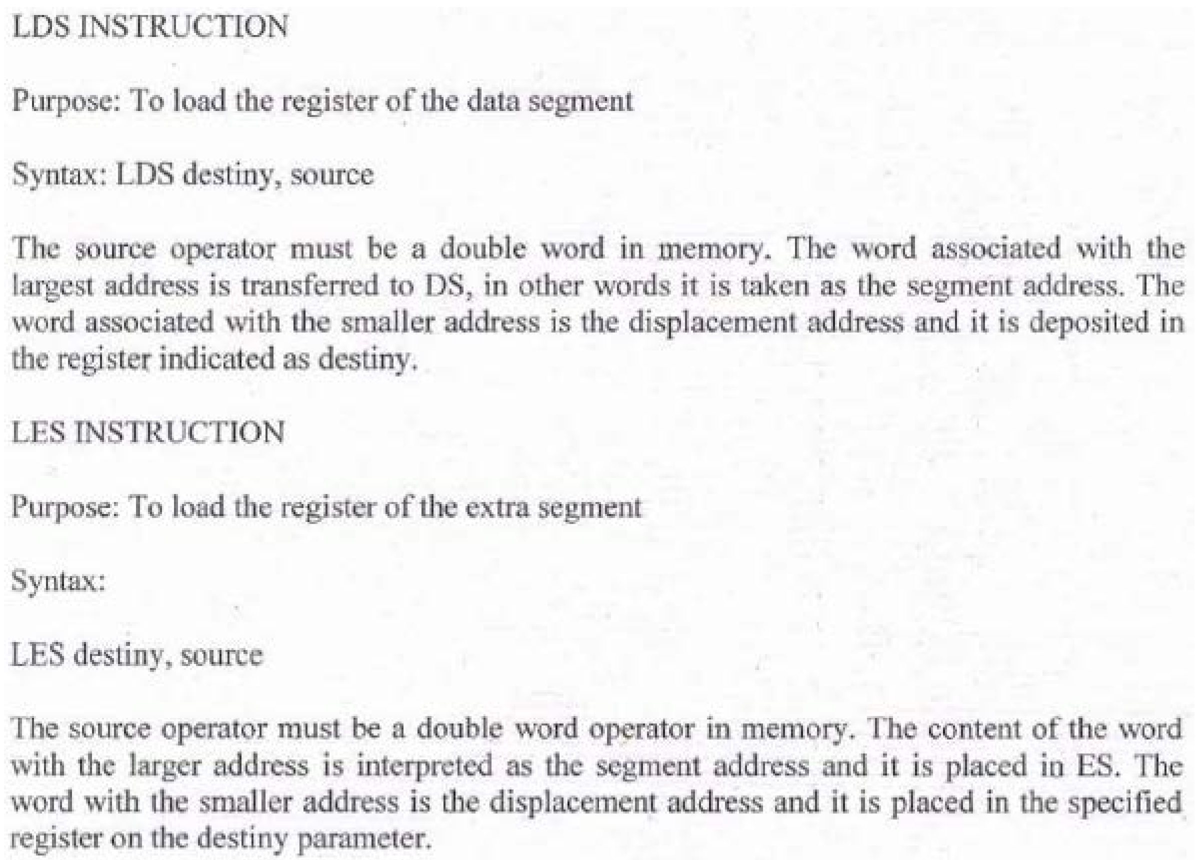
SAHF: (Store AH register into flag register) It is an instruction used to store the data in the AH register into the lower eight bits of the flag register.

BOUND: The BOUND instruction, which has two operands, compares a register with two words of memory data.

IMUL: This is an instruction for multiplication of two signed numbers. The result is a signed numbers. The OF (Over flow) and CF (Carry flag) are get affected.

b. Explain with examples LDS and LES instructions. (4)

Answer:



LDS INSTRUCTION

Purpose: To load the register of the data segment

Syntax: LDS destiny, source

The source operator must be a double word in memory. The word associated with the largest address is transferred to DS, in other words it is taken as the segment address. The word associated with the smaller address is the displacement address and it is deposited in the register indicated as destiny.

LES INSTRUCTION

Purpose: To load the register of the extra segment

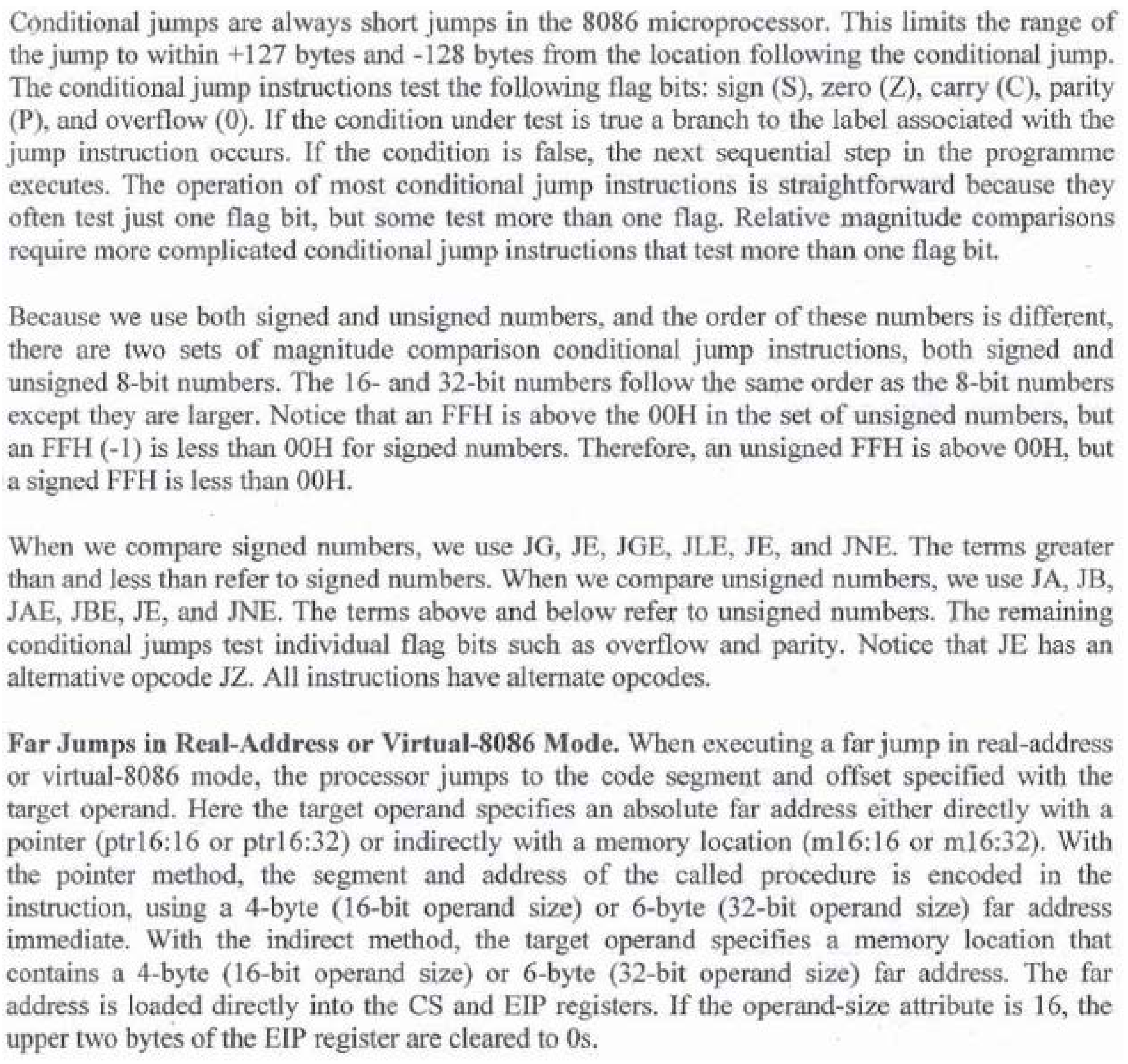
Syntax:

LES destiny, source

The source operator must be a double word operator in memory. The content of the word with the larger address is interpreted as the segment address and it is placed in ES. The word with the smaller address is the displacement address and it is placed in the specified register on the destiny parameter.

Q.4 a. Explain with examples conditional jump instructions which perform a jump based on the value of a single flag. What is the change needed in the code to branch anywhere in the segment based on a condition? (8)

Answer:



Conditional jumps are always short jumps in the 8086 microprocessor. This limits the range of the jump to within +127 bytes and -128 bytes from the location following the conditional jump. The conditional jump instructions test the following flag bits: sign (S), zero (Z), carry (C), parity (P), and overflow (0). If the condition under test is true a branch to the label associated with the jump instruction occurs. If the condition is false, the next sequential step in the programme executes. The operation of most conditional jump instructions is straightforward because they often test just one flag bit, but some test more than one flag. Relative magnitude comparisons require more complicated conditional jump instructions that test more than one flag bit.

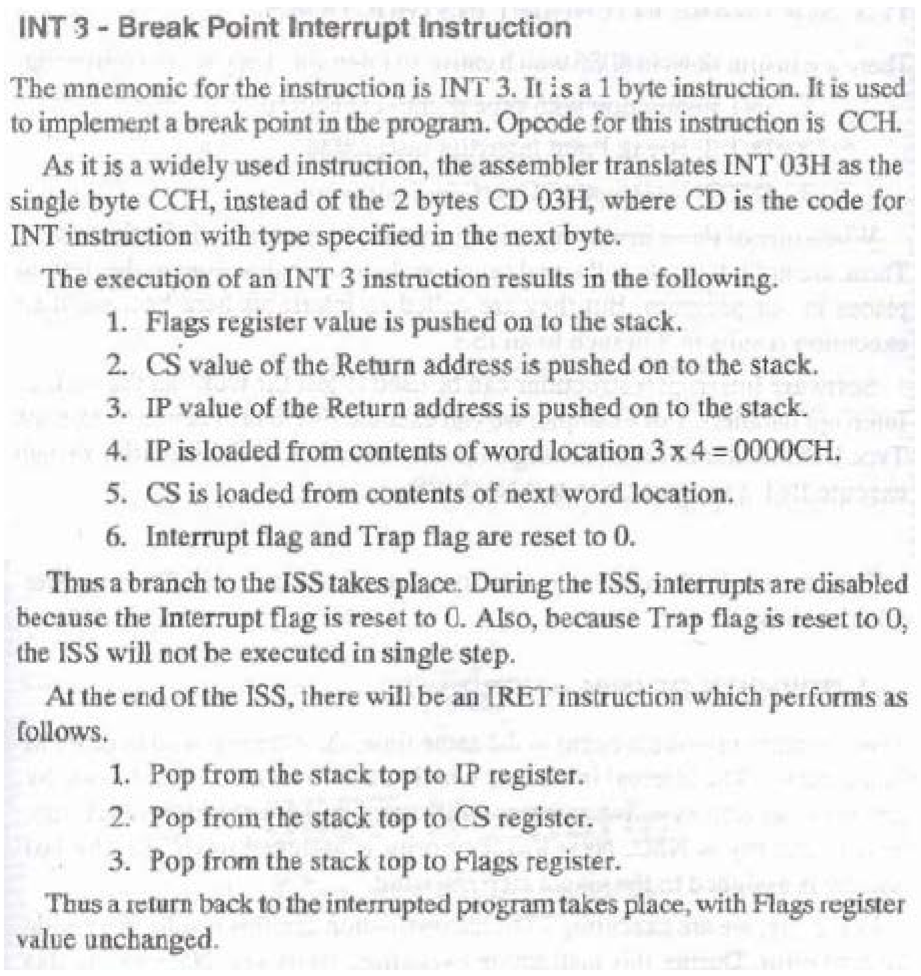
Because we use both signed and unsigned numbers, and the order of these numbers is different, there are two sets of magnitude comparison conditional jump instructions, both signed and unsigned 8-bit numbers. The 16- and 32-bit numbers follow the same order as the 8-bit numbers except they are larger. Notice that an FFH is above the 00H in the set of unsigned numbers, but an FFH (-1) is less than 00H for signed numbers. Therefore, an unsigned FFH is above 00H, but a signed FFH is less than 00H.

When we compare signed numbers, we use JG, JE, JGE, JLE, JE, and JNE. The terms greater than and less than refer to signed numbers. When we compare unsigned numbers, we use JA, JB, JAE, JBE, JE, and JNE. The terms above and below refer to unsigned numbers. The remaining conditional jumps test individual flag bits such as overflow and parity. Notice that JE has an alternative opcode JZ. All instructions have alternate opcodes.

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (ptr16:16 or ptr16:32) or indirectly with a memory location (m16:16 or m16:32). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

b. Enlists the different steps results in the execution of an INT3 instruction. (8)

Answer:



INT 3 - Break Point Interrupt Instruction

The mnemonic for the instruction is INT 3. It is a 1 byte instruction. It is used to implement a break point in the program. Opcode for this instruction is CCH.

As it is a widely used instruction, the assembler translates INT 03H as the single byte CCH, instead of the 2 bytes CD 03H, where CD is the code for INT instruction with type specified in the next byte.

The execution of an INT 3 instruction results in the following.

1. Flags register value is pushed on to the stack.
2. CS value of the Return address is pushed on to the stack.
3. IP value of the Return address is pushed on to the stack.
4. IP is loaded from contents of word location 3 x 4 = 0000CH.
5. CS is loaded from contents of next word location.
6. Interrupt flag and Trap flag are reset to 0.

Thus a branch to the ISS takes place. During the ISS, interrupts are disabled because the Interrupt flag is reset to 0. Also, because Trap flag is reset to 0, the ISS will not be executed in single step.

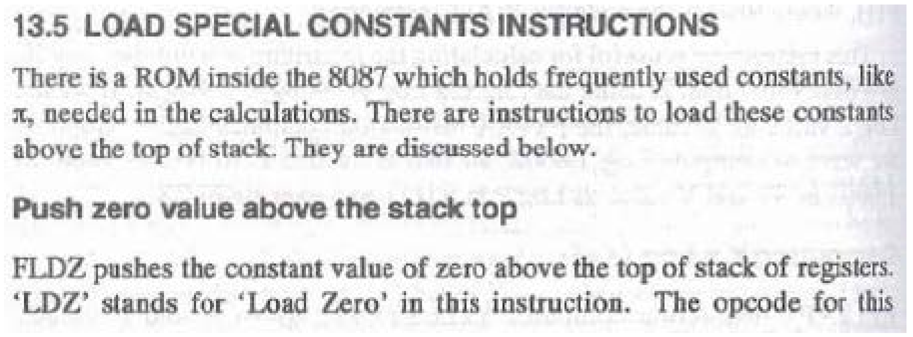
At the end of the ISS, there will be an IRET instruction which performs as follows.

1. Pop from the stack top to IP register.
2. Pop from the stack top to CS register.
3. Pop from the stack top to Flags register.

Thus a return back to the interrupted program takes place, with Flags register value unchanged.

Q.5 a. Explain the 8087 instructions to load special constants. (8)

Answer:

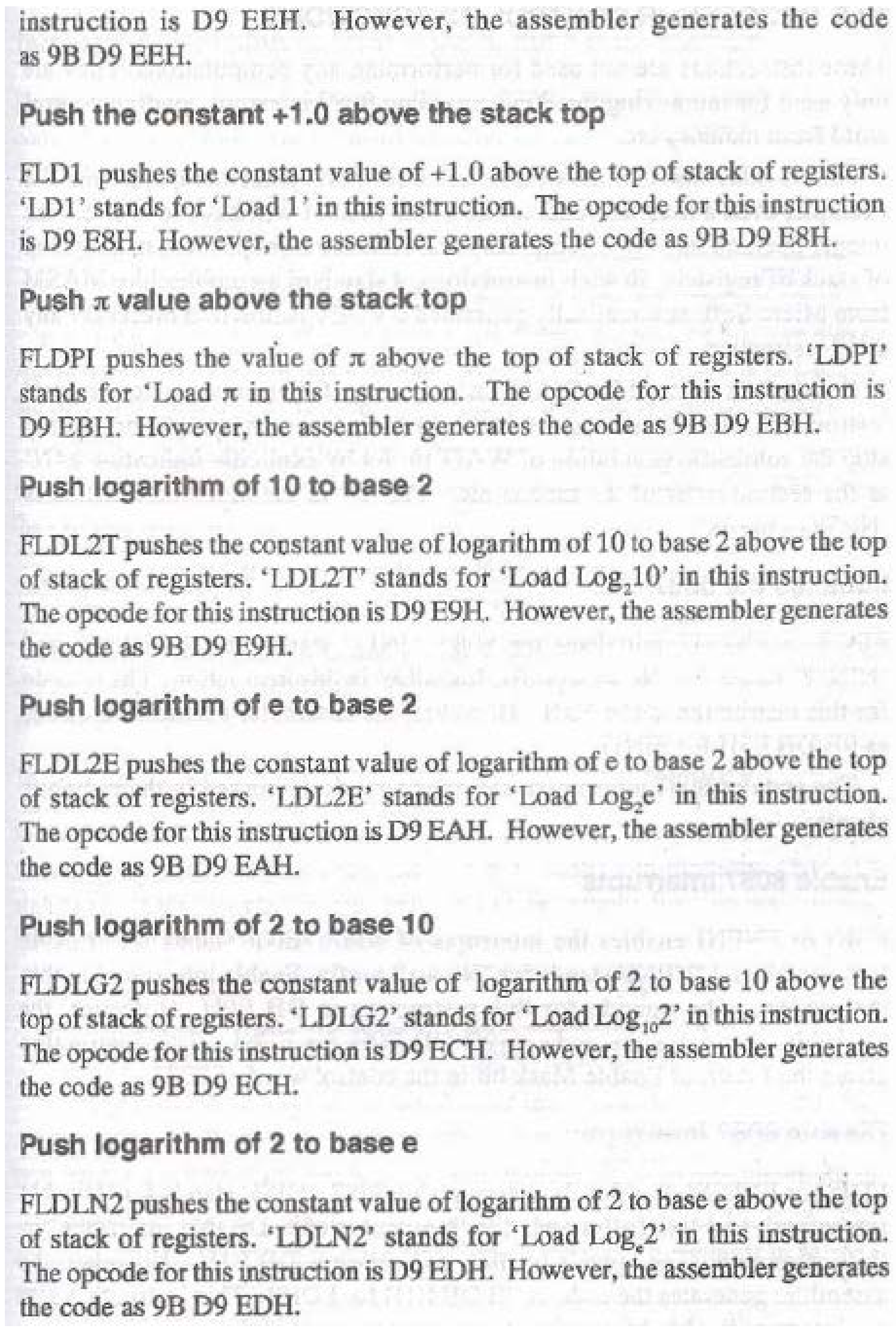


13.5 LOAD SPECIAL CONSTANTS INSTRUCTIONS

There is a ROM inside the 8087 which holds frequently used constants, like π, needed in the calculations. There are instructions to load these constants above the top of stack. They are discussed below.

Push zero value above the stack top

FLDZ pushes the constant value of zero above the top of stack of registers. 'LDZ' stands for 'Load Zero' in this instruction. The opcode for this



instruction is D9 EEH. However, the assembler generates the code as 9B D9 EEH.

Push the constant +1.0 above the stack top

FLD1 pushes the constant value of +1.0 above the top of stack of registers. 'LD1' stands for 'Load 1' in this instruction. The opcode for this instruction is D9 E8H. However, the assembler generates the code as 9B D9 E8H.

Push π value above the stack top

FLDPI pushes the value of π above the top of stack of registers. 'LDPI' stands for 'Load π in this instruction. The opcode for this instruction is D9 EBH. However, the assembler generates the code as 9B D9 EBH.

Push logarithm of 10 to base 2

FLDL2T pushes the constant value of logarithm of 10 to base 2 above the top of stack of registers. 'LDL2T' stands for 'Load Log$_2$10' in this instruction. The opcode for this instruction is D9 E9H. However, the assembler generates the code as 9B D9 E9H.

Push logarithm of e to base 2

FLDL2E pushes the constant value of logarithm of e to base 2 above the top of stack of registers. 'LDL2E' stands for 'Load Log$_2$e' in this instruction. The opcode for this instruction is D9 EAH. However, the assembler generates the code as 9B D9 EAH.

Push logarithm of 2 to base 10

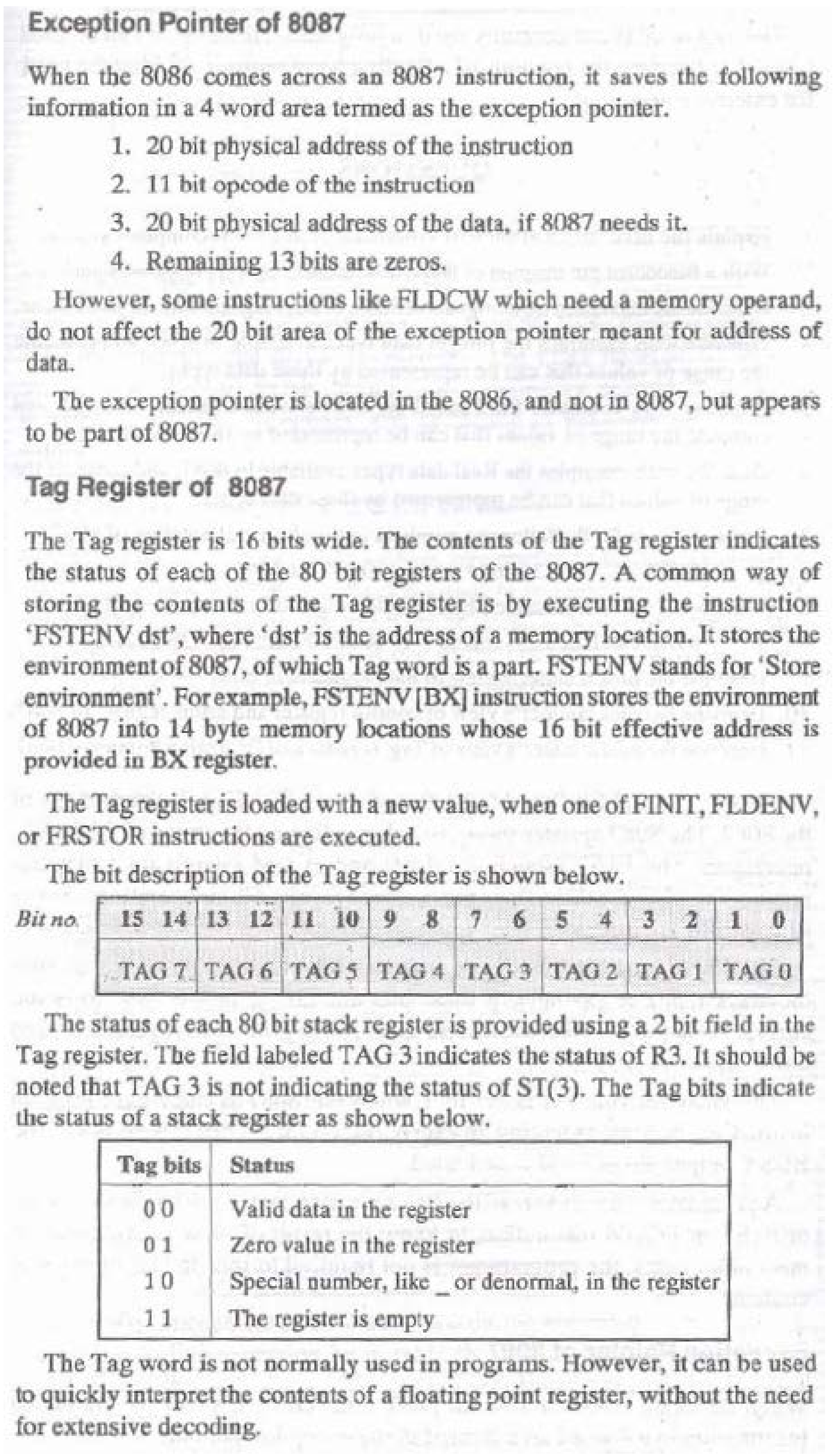
FLDLG2 pushes the constant value of logarithm of 2 to base 10 above the top of stack of registers. 'LDLG2' stands for 'Load Log$_{10}$2' in this instruction. The opcode for this instruction is D9 ECH. However, the assembler generates the code as 9B D9 ECH.

Push logarithm of 2 to base e

FLDLN2 pushes the constant value of logarithm of 2 to base e above the top of stack of registers. 'LDLN2' stands for 'Load Log$_e$2' in this instruction. The opcode for this instruction is D9 EDH. However, the assembler generates the code as 9B D9 EDH.

b. Describe the programmer's view of tag register and exception pointer of 8087.(8)

Answer:



Exception Pointer of 8087

When the 8086 comes across an 8087 instruction, it saves the following information in a 4 word area termed as the exception pointer.

1. 20 bit physical address of the instruction
2. 11 bit opcode of the instruction
3. 20 bit physical address of the data, if 8087 needs it.
4. Remaining 13 bits are zeros.

However, some instructions like FLDCW which need a memory operand, do not affect the 20 bit area of the exception pointer meant for address of data.

The exception pointer is located in the 8086, and not in 8087, but appears to be part of 8087.

Tag Register of 8087

The Tag register is 16 bits wide. The contents of the Tag register indicates the status of each of the 80 bit registers of the 8087. A common way of storing the contents of the Tag register is by executing the instruction 'FSTENV dst', where 'dst' is the address of a memory location. It stores the environment of 8087, of which Tag word is a part. FSTENV stands for 'Store environment'. For example, FSTENV [BX] instruction stores the environment of 8087 into 14 byte memory locations whose 16 bit effective address is provided in BX register.

The Tag register is loaded with a new value, when one of FINIT, FLDENV, or FRSTOR instructions are executed.

The bit description of the Tag register is shown below.

Bit no.	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
	TAG 7	TAG 6	TAG 5	TAG 4	TAG 3	TAG 2	TAG 1	TAG 0

The status of each 80 bit stack register is provided using a 2 bit field in the Tag register. The field labeled TAG 3 indicates the status of R3. It should be noted that TAG 3 is not indicating the status of ST(3). The Tag bits indicate the status of a stack register as shown below.

Tag bits	Status
0 0	Valid data in the register
0 1	Zero value in the register
1 0	Special number, like _ or denormal, in the register
1 1	The register is empty

The Tag word is not normally used in programs. However, it can be used to quickly interpret the contents of a floating point register, without the need for extensive decoding.

Q.6 a. Write a Program in assembly language to find the largest of n numbers stored in the memory. (8)

Answer:

Soln.,

```
MOV AX, 0000
MOV SI, 0200
MOV CX, [SI]
BACK : INC SI
INC SI
CMP AX, [SI]
JAE  GO
MOV AX, [SI]
GO: LOOP BACK
MOV [0251], AX
INT 3.
```

b. Discuss the following assembler directives with example (8)

(i) DWORD
(ii) OFFSET
(iii) SEGMENT
(iv) MACRO

Answer:

Soln.,

DWORD: It defines word type variable. The defined variable may have one or more initial values in the directive statement. If there is one value, two bytes of memory space are reserved. The general format is

Name of variable DW Initial value or values.

OFFSET: It is an operator to determine the offset (displacement) of a variable or procedure with respect to the base of the segment which contains the named variable or procedure. The operator can be used to laod a register with the offset of a variable.

The operator can be used as follows:

```
MOV SI, OFFSET ARRAY
```

SEGMENT: This directive defines to the assembler the start of a segment with name segement-name. The segment name should be unique and follows the rules of the assembler

The Syntax is as follows:

```
Segment Name   SEGMENT { Operand (Optional ) }        ; Comment
.
.
.

Segment Name          ENDS.
```

MACRO: A sequence of instructions to which a name is assigned is called macro. Macros and subroutines are similar. Macros are used for short sequence of instructions whereas subroutines for longer ones. Macros executes faster than subroutines.

The MACRO directive informs assembler the beginning of a macro This is used with ENDM directive to enclose a macro. The general format of the MACRO directive is:

```
        Macro Name   MACRO        ARG1, ARG2, ….., ARG N.
```

Q.7 a. Write an 8086 assembly language program to search for a given 8 bit value using linear search in an array of 8 bit numbers. (8)

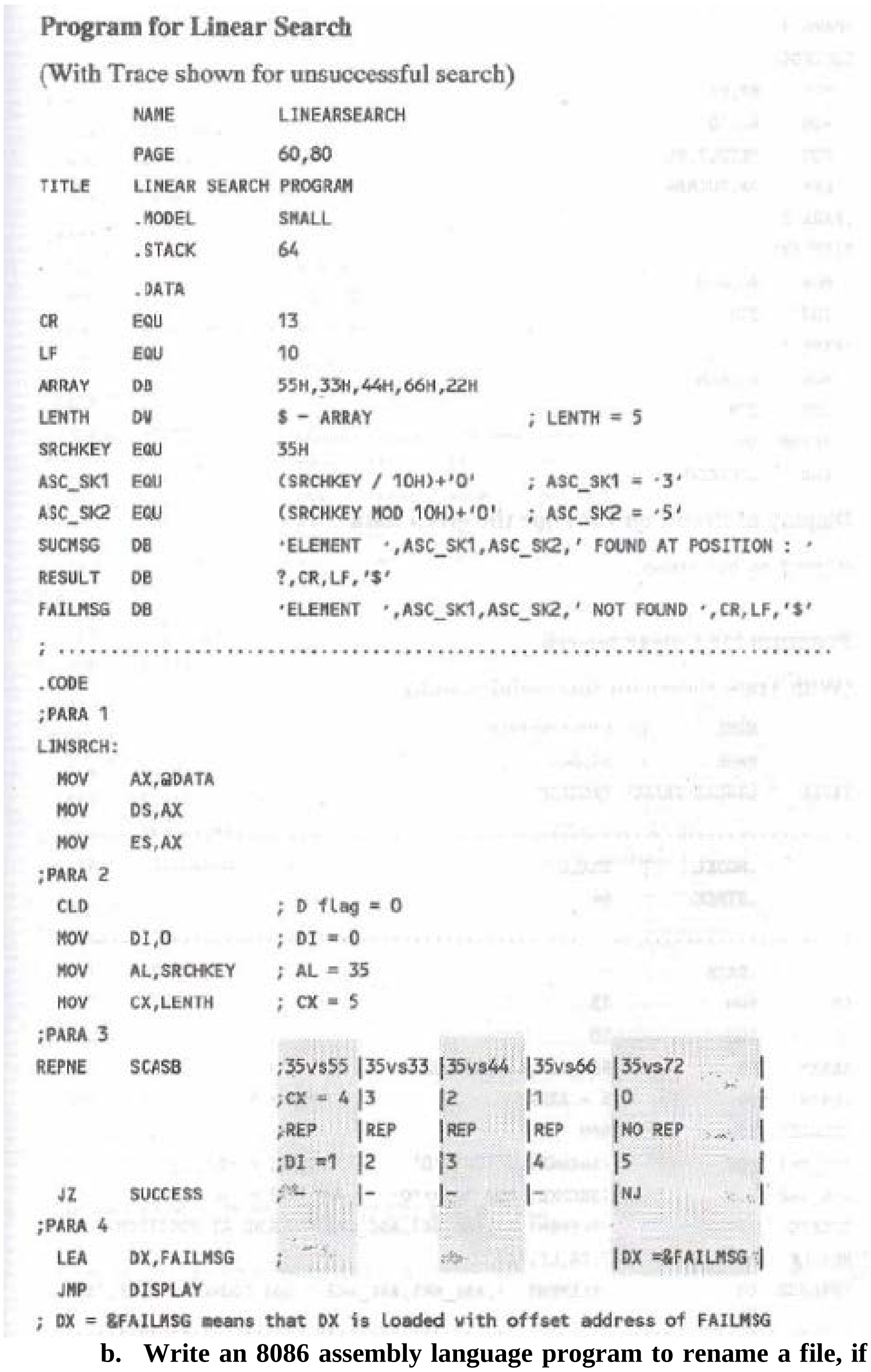
Answer:

Linear Search Program

Write an 8086 assembly language program to search for a given 8 bit value using linear search in an array of 8 bit numbers. Message should be displayed on CRT indicating whether the search was a failure or a success. If it is a success case, the position of the element in the array is to be displayed.

Approach Methodology Let us say, we want to search for the element 35 in the set of numbers 55,33,44,66,22 using linear search. We keep comparing the number 35 with each of the elements in the array, till we come across a match or till all the elements are compared.

In this example, because 35 is not found, we display the message "ELEMENT 35 NOT FOUND" on the CRT. Suppose we were searching for the number 66. Because it is found at position 4, the program should display the message "ELEMENT 66 FOUND AT POSITION : 4" on the CRT.



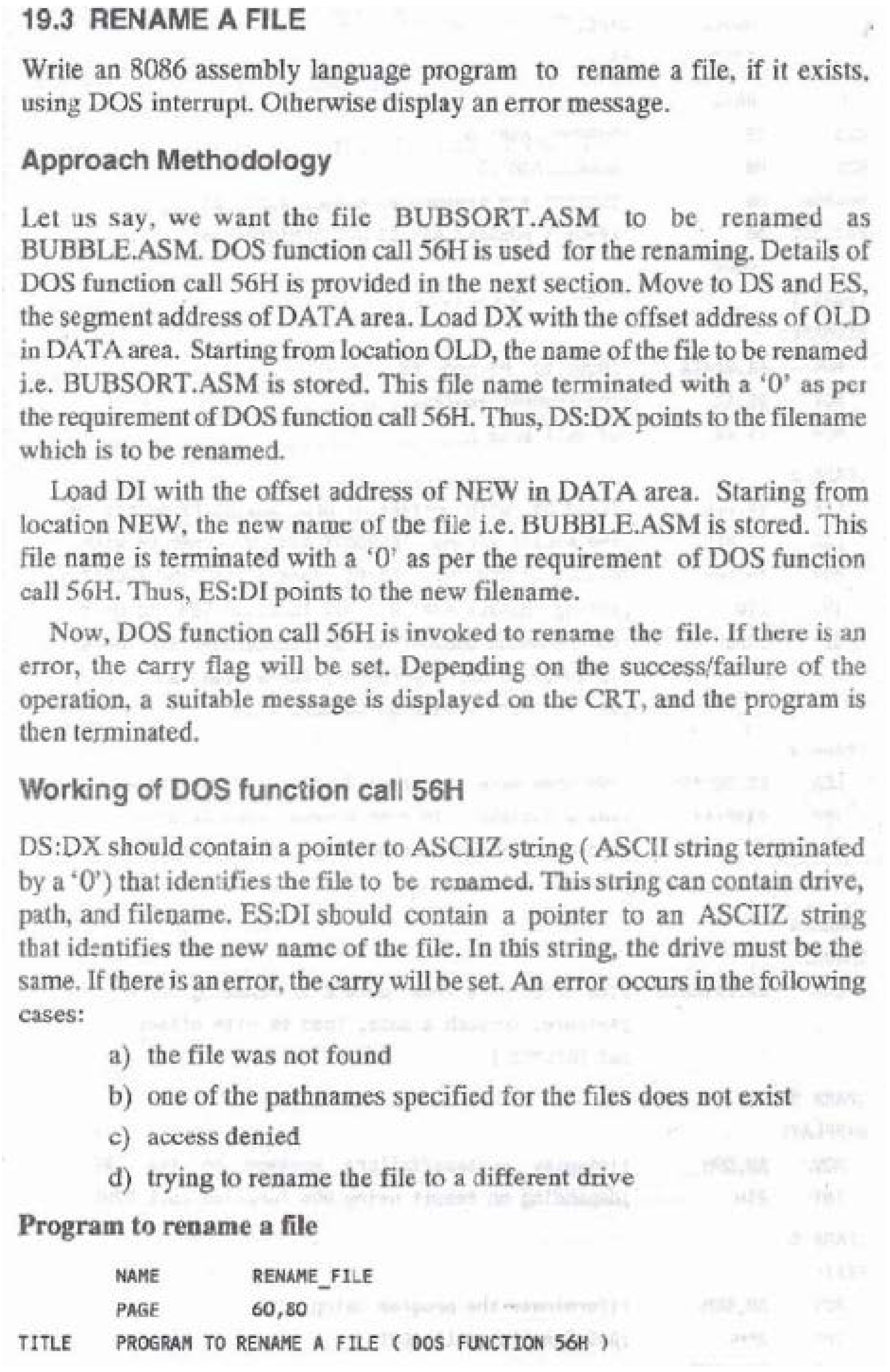
Program for Linear Search

(With Trace shown for unsuccessful search)

```
        NAME        LINEARSEARCH
        PAGE        60,80
TITLE   LINEAR SEARCH PROGRAM
        .MODEL      SMALL
        .STACK      64
        .DATA
CR      EQU         13
LF      EQU         10
ARRAY   DB          55H,33H,44H,66H,22H
LENTH   DW          $ - ARRAY              ; LENTH = 5
SRCHKEY EQU         35H
ASC_SK1 EQU         (SRCHKEY / 10H)+'0'    ; ASC_SK1 = '3'
ASC_SK2 EQU         (SRCHKEY MOD 10H)+'0'  ; ASC_SK2 = '5'
SUCMSG  DB          'ELEMENT  ',ASC_SK1,ASC_SK2,' FOUND AT POSITION : '
RESULT  DB          ?,CR,LF,'$'
FAILMSG DB          'ELEMENT  ',ASC_SK1,ASC_SK2,' NOT FOUND ',CR,LF,'$'
; ...........................................................................
.CODE
;PARA 1
LINSRCH:
  MOV   AX,@DATA
  MOV   DS,AX
  MOV   ES,AX
;PARA 2
  CLD               ; D flag = 0
  MOV   DI,0        ; DI = 0
  MOV   AL,SRCHKEY  ; AL = 35
  MOV   CX,LENTH    ; CX = 5
;PARA 3
REPNE   SCASB       ;35vs55 |35vs33 |35vs44 |35vs66 |35vs72       |
                    ;CX = 4 |3      |2      |1      |0            |
                    ;REP    |REP    |REP    |REP    |NO REP       |
                    ;DI =1  |2      |3      |4      |5            |
  JZ    SUCCESS     ;-      |-      |-      |-      |NJ           |
;PARA 4
  LEA   DX,FAILMSG  ;                               |DX =&FAILMSG |
  JMP   DISPLAY
; DX = &FAILMSG means that DX is loaded with offset address of FAILMSG
```

b. Write an 8086 assembly language program to rename a file, if it exists, using DOS interrupt. Otherwise display on error message. (8)

Answer:



19.3 RENAME A FILE

Write an 8086 assembly language program to rename a file, if it exists, using DOS interrupt. Otherwise display an error message.

Approach Methodology

Let us say, we want the file BUBSORT.ASM to be renamed as BUBBLE.ASM. DOS function call 56H is used for the renaming. Details of DOS function call 56H is provided in the next section. Move to DS and ES, the segment address of DATA area. Load DX with the offset address of OLD in DATA area. Starting from location OLD, the name of the file to be renamed i.e. BUBSORT.ASM is stored. This file name terminated with a '0' as per the requirement of DOS function call 56H. Thus, DS:DX points to the filename which is to be renamed.

Load DI with the offset address of NEW in DATA area. Starting from location NEW, the new name of the file i.e. BUBBLE.ASM is stored. This file name is terminated with a '0' as per the requirement of DOS function call 56H. Thus, ES:DI points to the new filename.

Now, DOS function call 56H is invoked to rename the file. If there is an error, the carry flag will be set. Depending on the success/failure of the operation, a suitable message is displayed on the CRT, and the program is then terminated.

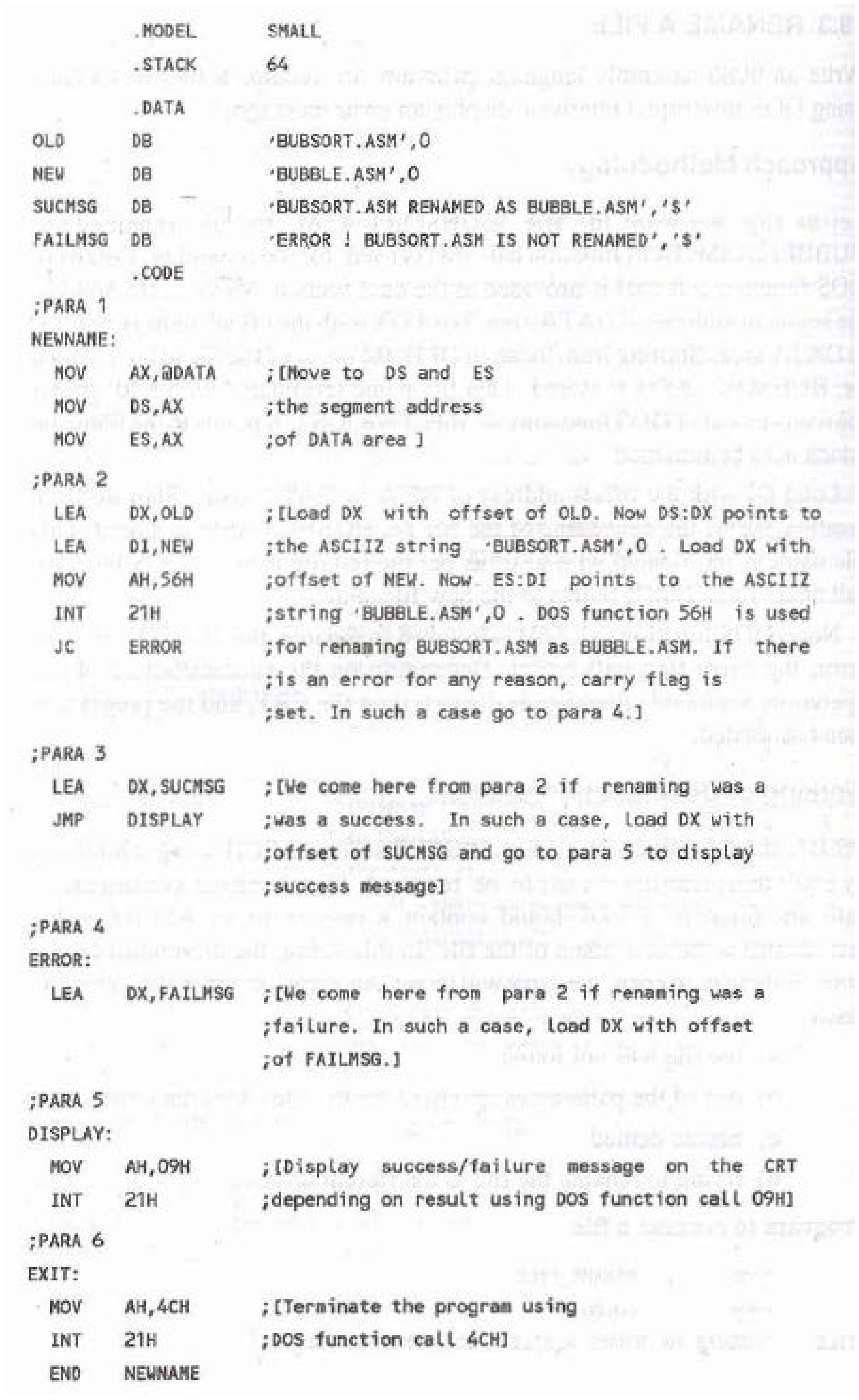
Working of DOS function call 56H

DS:DX should contain a pointer to ASCIIZ string (ASCII string terminated by a '0') that identifies the file to be renamed. This string can contain drive, path, and filename. ES:DI should contain a pointer to an ASCIIZ string that identifies the new name of the file. In this string, the drive must be the same. If there is an error, the carry will be set. An error occurs in the following cases:

a) the file was not found
b) one of the pathnames specified for the files does not exist
c) access denied
d) trying to rename the file to a different drive

Program to rename a file

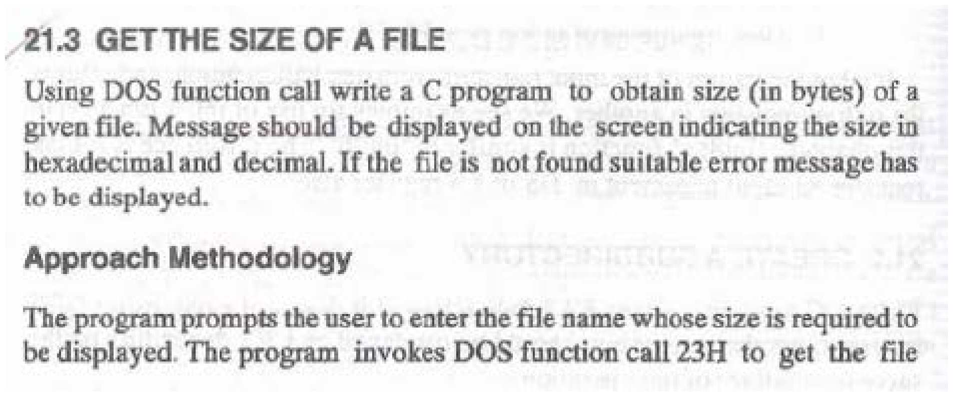
```
        NAME        RENAME_FILE
        PAGE        60,80
TITLE   PROGRAM TO RENAME A FILE ( DOS FUNCTION 56H )
```



```
          .MODEL      SMALL
          .STACK      64
          .DATA
OLD       DB          'BUBSORT.ASM',0
NEW       DB          'BUBBLE.ASM',0
SUCMSG    DB          'BUBSORT.ASM RENAMED AS BUBBLE.ASM','$'
FAILMSG   DB          'ERROR ! BUBSORT.ASM IS NOT RENAMED','$'
          .CODE
;PARA 1
NEWNAME:
  MOV     AX,@DATA    ;[Move to  DS and  ES
  MOV     DS,AX       ;the segment address
  MOV     ES,AX       ;of DATA area ]
;PARA 2
  LEA     DX,OLD      ;[Load DX  with  offset of OLD. Now DS:DX points to
  LEA     DI,NEW      ;the ASCIIZ string  'BUBSORT.ASM',0 . Load DX with
  MOV     AH,56H      ;offset of NEW. Now  ES:DI  points  to  the ASCIIZ
  INT     21H         ;string 'BUBBLE.ASM',0 . DOS function 56H  is used
  JC      ERROR       ;for renaming BUBSORT.ASM as BUBBLE.ASM. If  there
                      ;is an error for any reason, carry flag is
                      ;set. In such a case go to para 4.]
;PARA 3
  LEA     DX,SUCMSG   ;[We come here from para 2 if  renaming  was a
  JMP     DISPLAY     ;was a success.  In such a case, load DX with
                      ;offset of SUCMSG and go to para 5 to display
                      ;success message]
;PARA 4
ERROR:
  LEA     DX,FAILMSG  ;[We come  here from  para 2 if renaming was a
                      ;failure. In such a case, load DX with offset
                      ;of FAILMSG.]
;PARA 5
DISPLAY:
  MOV     AH,09H      ;[Display  success/failure  message  on  the  CRT
  INT     21H         ;depending on result using DOS function call 09H]
;PARA 6
EXIT:
  MOV     AH,4CH      ;[Terminate the program using
  INT     21H         ;DOS function call 4CH]
  END     NEWNAME
```

Q.8 a. Using DOS function call, write a C program to obtain the size of given file. Message should be displayed on the screen indicating the size in hexadecimal and decimal format. If the file is not found suitable error message should be displayed. (8)

Answer:

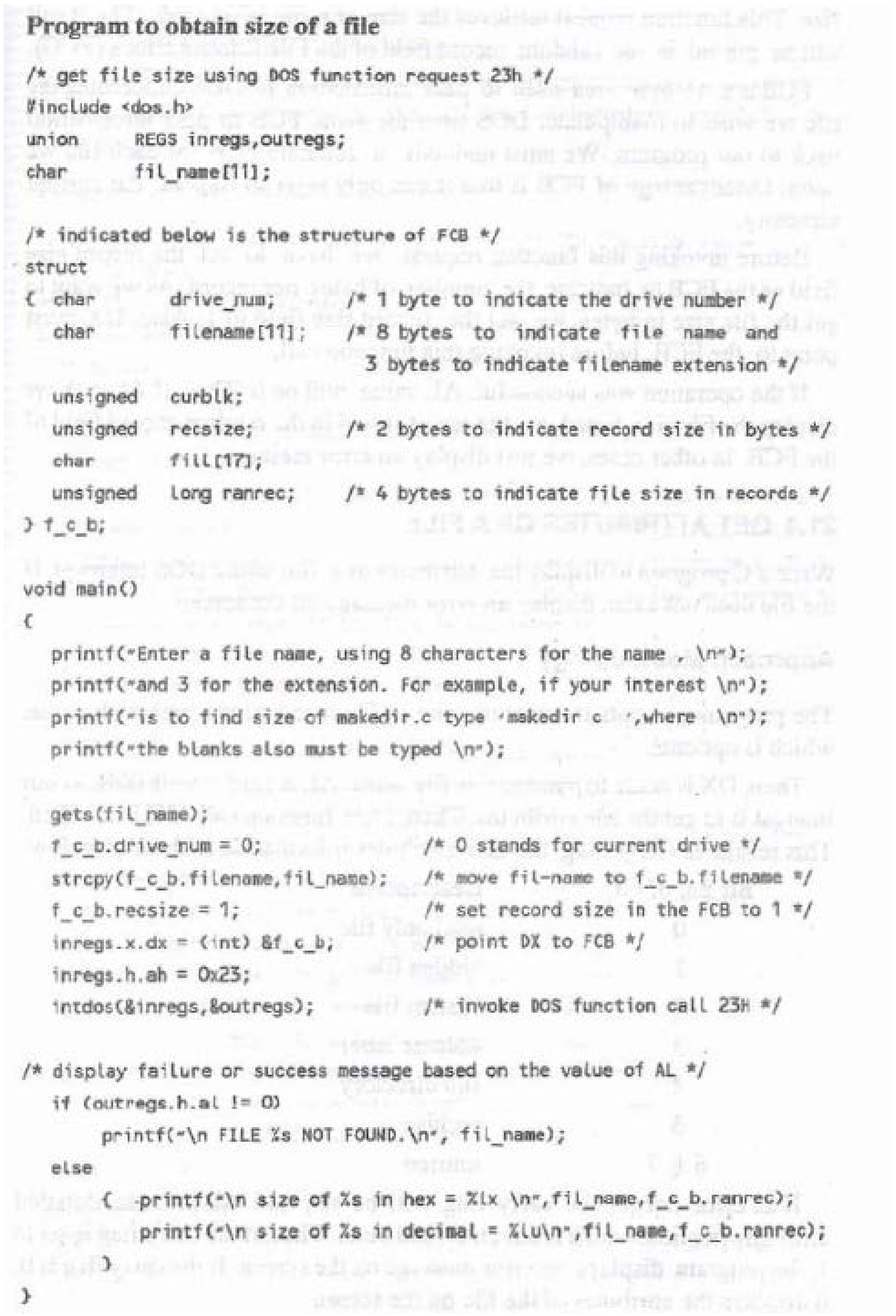


21.3 GET THE SIZE OF A FILE

Using DOS function call write a C program to obtain size (in bytes) of a given file. Message should be displayed on the screen indicating the size in hexadecimal and decimal. If the file is not found suitable error message has to be displayed.

Approach Methodology

The program prompts the user to enter the file name whose size is required to be displayed. The program invokes DOS function call 23H to get the file

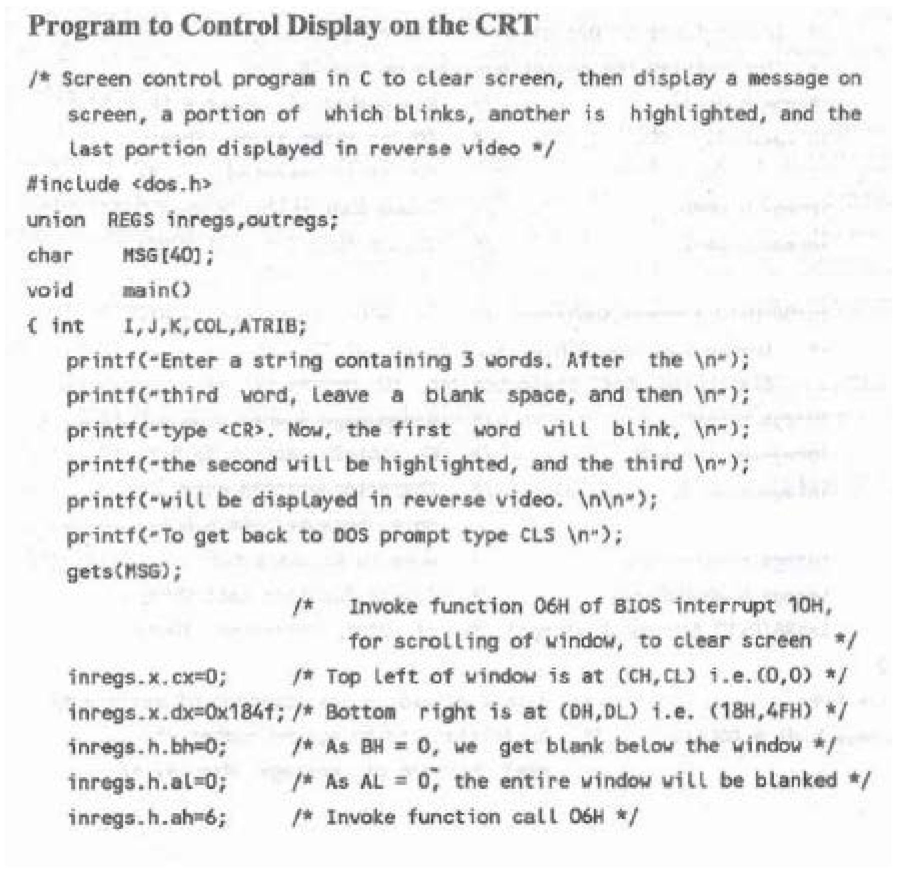


Program to obtain size of a file

```
/* get file size using DOS function request 23h */
#include <dos.h>
union     REGS inregs,outregs;
char      fil_name[11];

/* indicated below is the structure of FCB */
struct
{ char       drive_num;      /* 1 byte to indicate the drive number */
  char       filename[11];   /* 8 bytes  to  indicate  file  name  and
                                3 bytes to indicate filename extension */
  unsigned   curblk;
  unsigned   recsize;        /* 2 bytes to indicate record size in bytes */
  char       fill[17];
  unsigned   long ranrec;    /* 4 bytes to indicate file size in records */
} f_c_b;

void main()
{
  printf("Enter a file name, using 8 characters for the name , \n");
  printf("and 3 for the extension. For example, if your interest \n");
  printf("is to find size of makedir.c type 'makedir c   ',where  \n");
  printf("the blanks also must be typed \n");

  gets(fil_name);
  f_c_b.drive_num = 0;               /* 0 stands for current drive */
  strcpy(f_c_b.filename,fil_name);   /* move fil-name to f_c_b.filename */
  f_c_b.recsize = 1;                 /* set record size in the FCB to 1 */
  inregs.x.dx = (int) &f_c_b;        /* point DX to FCB */
  inregs.h.ah = 0x23;
  intdos(&inregs,&outregs);          /* invoke DOS function call 23H */

/* display failure or success message based on the value of AL */
  if (outregs.h.al != 0)
       printf("\n FILE %s NOT FOUND.\n", fil_name);
  else
       {  printf("\n size of %s in hex = %lx \n",fil_name,f_c_b.ranrec);
          printf("\n size of %s in decimal = %lu\n",fil_name,f_c_b.ranrec);
       }
}
```

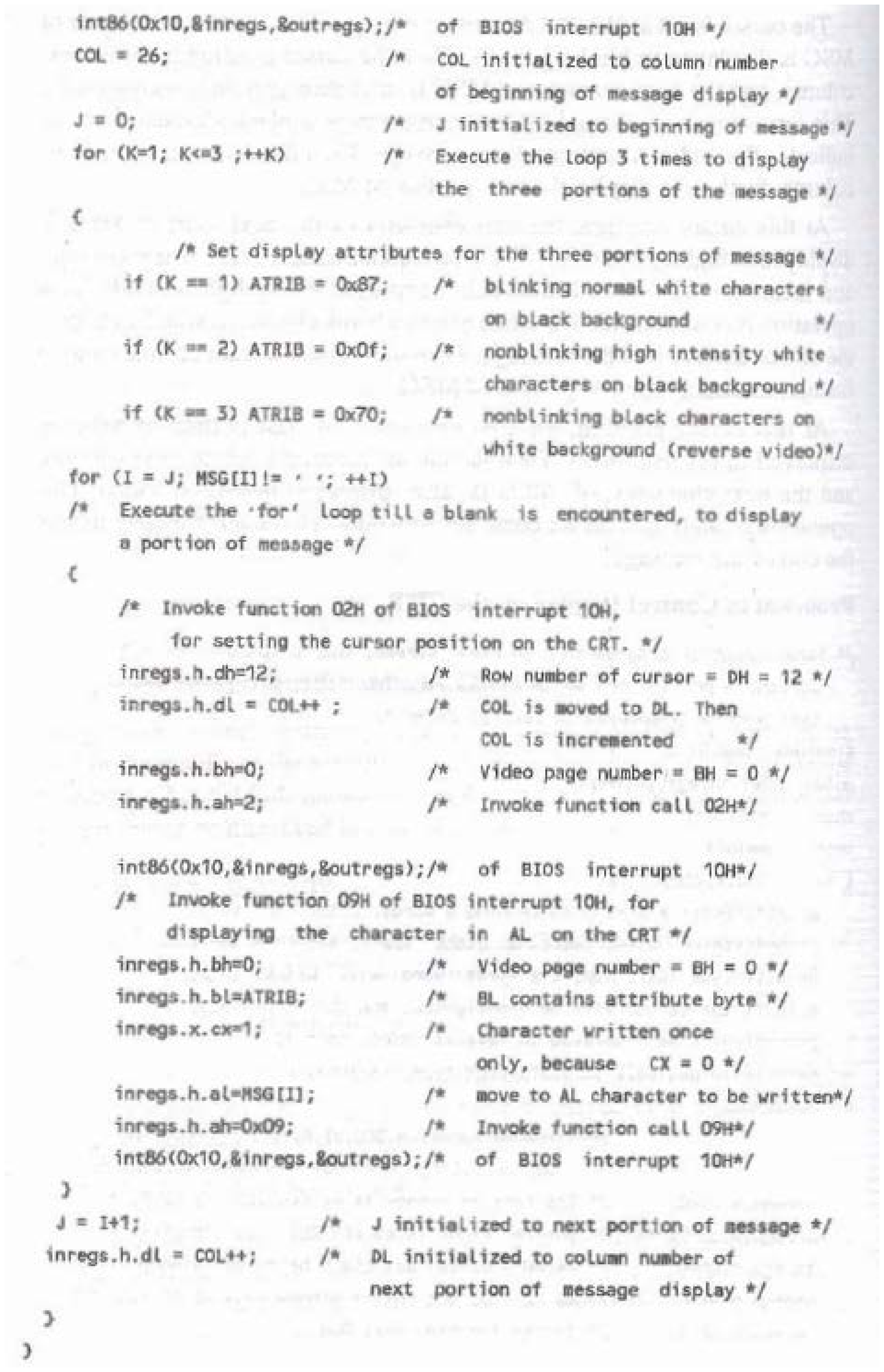
b. Using BIOS routines, write a C program to display a suitable message on CRT in the middle of the screen, after clearing the screen first. (8)

Answer:



Program to Control Display on the CRT

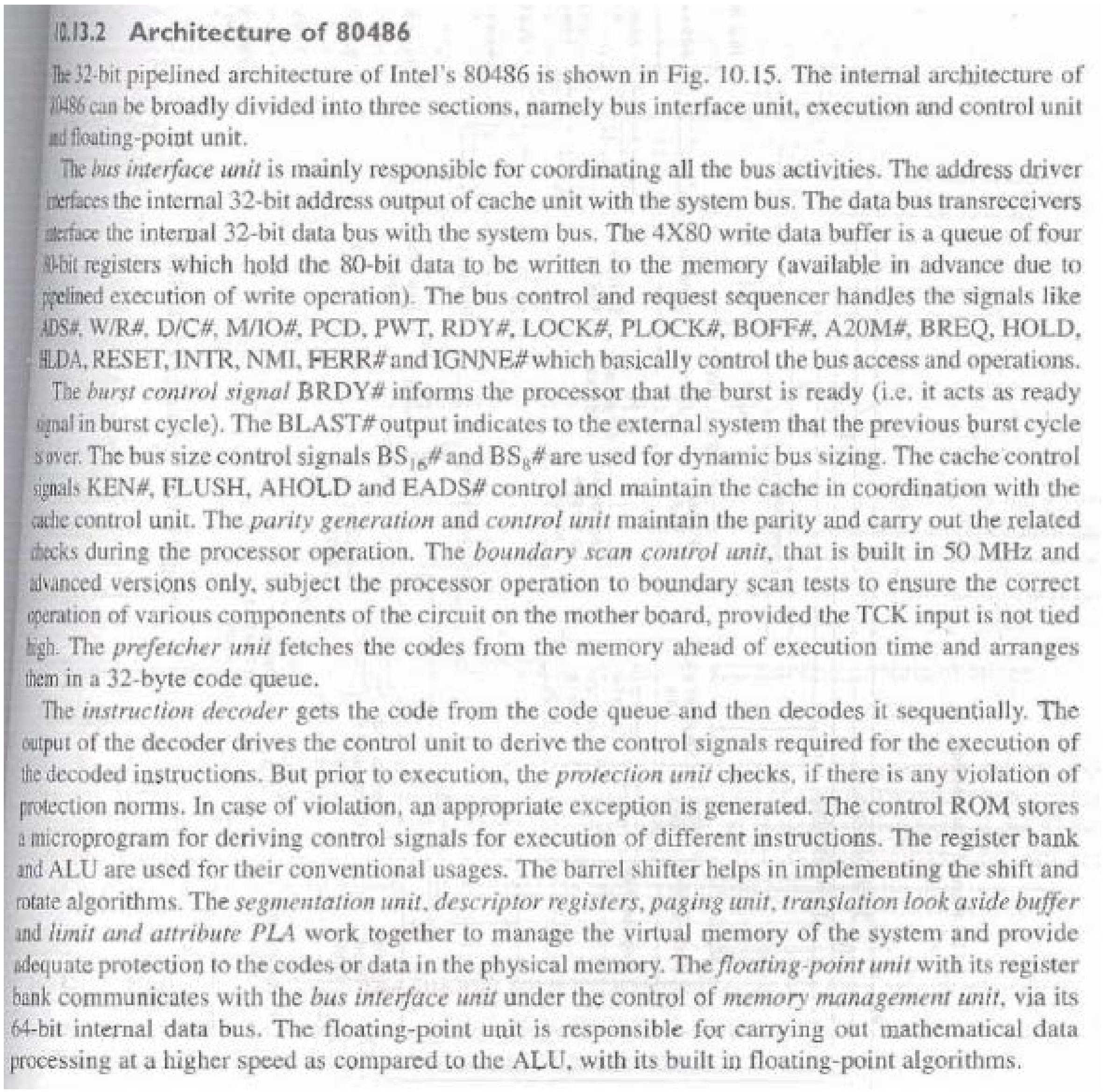
```
/* Screen control program in C to clear screen, then display a message on
   screen, a portion of  which blinks, another is  highlighted, and the
   last portion displayed in reverse video */
#include <dos.h>
union  REGS inregs,outregs;
char    MSG[40];
void    main()
{ int   I,J,K,COL,ATRIB;
   printf("Enter a string containing 3 words. After  the \n");
   printf("third  word, leave  a  blank  space, and then \n");
   printf("type <CR>. Now, the first  word  will  blink, \n");
   printf("the second will be highlighted, and the third \n");
   printf("will be displayed in reverse video. \n\n");
   printf("To get back to DOS prompt type CLS \n");
   gets(MSG);
                     /*   Invoke function 06H of BIOS interrupt 10H,
                          for scrolling of window, to clear screen  */
   inregs.x.cx=0;      /* Top left of window is at (CH,CL) i.e.(0,0) */
   inregs.x.dx=0x184f; /* Bottom  right is at (DH,DL) i.e. (18H,4FH) */
   inregs.h.bh=0;      /* As BH = 0, we  get blank below the window */
   inregs.h.al=0;      /* As AL = 0, the entire window will be blanked */
   inregs.h.ah=6;      /* Invoke function call 06H */
```



```
 int86(0x10,&inregs,&outregs);/*   of  BIOS  interrupt  10H */
 COL = 26;                    /*   COL initialized to column number
                                   of beginning of message display */
  J = 0;                      /*   J initialized to beginning of message */
 for (K=1; K<=3 ;++K)         /*   Execute the loop 3 times to display
                                   the  three  portions of the message */
 {
          /* Set display attributes for the three portions of message */
     if (K == 1) ATRIB = 0x87;     /*   blinking normal white characters
                                        on black background             */
     if (K == 2) ATRIB = 0x0f;     /*   nonblinking high intensity white
                                        characters on black background */
     if (K == 3) ATRIB = 0x70;     /*   nonblinking black characters on
                                        white background (reverse video)*/
 for (I = J; MSG[I]!= ' '; ++I)
 /*   Execute the 'for'  loop till a blank  is  encountered, to display
      a portion of message */
 {
      /*  Invoke function 02H of BIOS  interrupt 10H,
          for setting the cursor position on the CRT. */
      inregs.h.dh=12;             /*   Row number of cursor = DH = 12 */
      inregs.h.dl = COL++ ;       /*   COL is moved to DL. Then
                                       COL is incremented      */
      inregs.h.bh=0;              /*   Video page number = BH = 0 */
      inregs.h.ah=2;              /*   Invoke function call 02H*/

      int86(0x10,&inregs,&outregs);/*   of  BIOS  interrupt  10H*/
      /*   Invoke function 09H of BIOS interrupt 10H, for
           displaying  the  character  in  AL  on the CRT */
      inregs.h.bh=0;              /*   Video page number = BH = 0 */
      inregs.h.bl=ATRIB;          /*   BL contains attribute byte */
      inregs.x.cx=1;              /*   Character written once
                                       only, because   CX = 0 */
      inregs.h.al=MSG[I];         /*   move to AL character to be written*/
      inregs.h.ah=0x09;           /*   Invoke function call 09H*/
      int86(0x10,&inregs,&outregs);/*   of  BIOS  interrupt  10H*/
 }
  J = I+1;                /*   J initialized to next portion of message */
 inregs.h.dl = COL++;     /*   DL initialized to column number of
                               next  portion of  message  display */
 }
}
```

Q.9 a. Explain the Architecture of 80486. (8)

Answer:



10.13.2 Architecture of 80486

The 32-bit pipelined architecture of Intel's 80486 is shown in Fig. 10.15. The internal architecture of 80486 can be broadly divided into three sections, namely bus interface unit, execution and control unit and floating-point unit.

The *bus interface unit* is mainly responsible for coordinating all the bus activities. The address driver interfaces the internal 32-bit address output of cache unit with the system bus. The data bus transreceivers interface the internal 32-bit data bus with the system bus. The 4X80 write data buffer is a queue of four 80-bit registers which hold the 80-bit data to be written to the memory (available in advance due to pipelined execution of write operation). The bus control and request sequencer handles the signals like ADS#, W/R#, D/C#, M/IO#, PCD, PWT, RDY#, LOCK#, PLOCK#, BOFF#, A20M#, BREQ, HOLD, HLDA, RESET, INTR, NMI, FERR# and IGNNE# which basically control the bus access and operations. The *burst control signal* BRDY# informs the processor that the burst is ready (i.e. it acts as ready signal in burst cycle). The BLAST# output indicates to the external system that the previous burst cycle is over. The bus size control signals $BS_{16}\#$ and $BS_8\#$ are used for dynamic bus sizing. The cache control signals KEN#, FLUSH, AHOLD and EADS# control and maintain the cache in coordination with the cache control unit. The *parity generation* and *control unit* maintain the parity and carry out the related checks during the processor operation. The *boundary scan control unit*, that is built in 50 MHz and advanced versions only, subject the processor operation to boundary scan tests to ensure the correct operation of various components of the circuit on the mother board, provided the TCK input is not tied high. The *prefetcher unit* fetches the codes from the memory ahead of execution time and arranges them in a 32-byte code queue.

The *instruction decoder* gets the code from the code queue and then decodes it sequentially. The output of the decoder drives the control unit to derive the control signals required for the execution of the decoded instructions. But prior to execution, the *protection unit* checks, if there is any violation of protection norms. In case of violation, an appropriate exception is generated. The control ROM stores a microprogram for deriving control signals for execution of different instructions. The register bank and ALU are used for their conventional usages. The barrel shifter helps in implementing the shift and rotate algorithms. The *segmentation unit*, *descriptor registers*, *paging unit*, *translation look aside buffer* and *limit and attribute PLA* work together to manage the virtual memory of the system and provide adequate protection to the codes or data in the physical memory. The *floating-point unit* with its register bank communicates with the *bus interface unit* under the control of *memory management unit*, via its 64-bit internal data bus. The floating-point unit is responsible for carrying out mathematical data processing at a higher speed as compared to the ALU, with its built in floating-point algorithms.

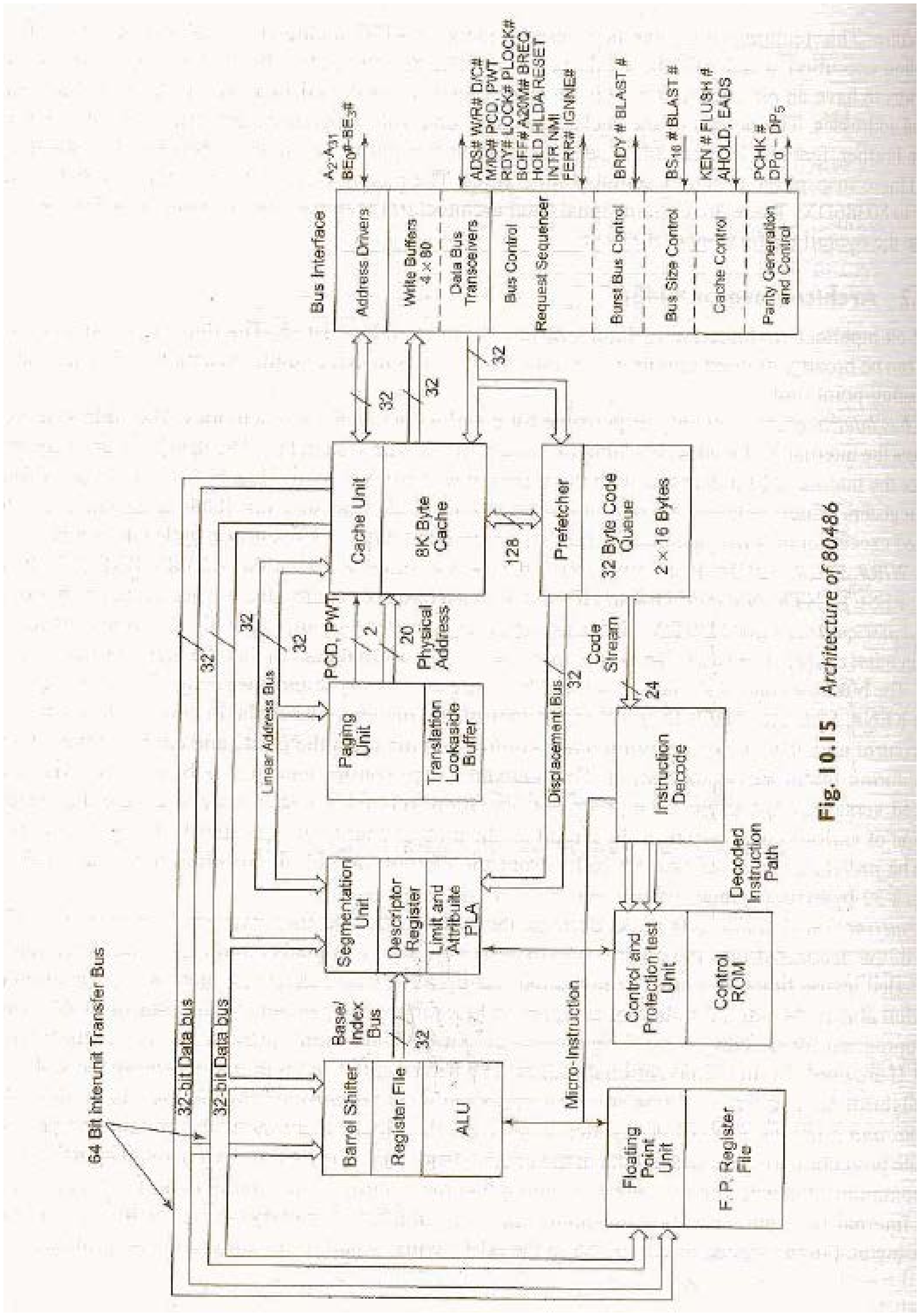


Fig. 10.15 *Architecture of 80486*

b. Explain the register organization of 80286. (8)

Answer:

9.2 INTERNAL ARCHITECTURE OF 80286

9.2.1 Register Organisation of 80286

The 80286 CPU contains almost the same set of registers, as in 8086, viz.

(a) Eight 16-bit general purpose registers
(b) Four 16-bit segment registers
(c) Status and control register
(d) Instruction pointer.

The register set of 80286 is shown in Fig. 9.1.

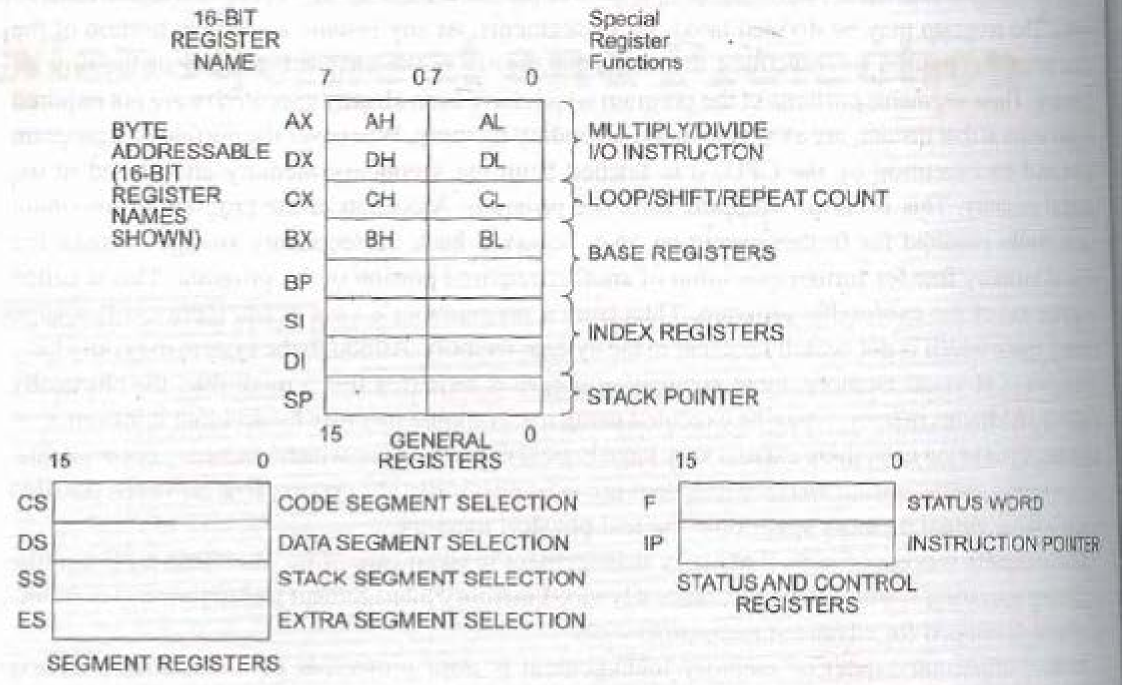
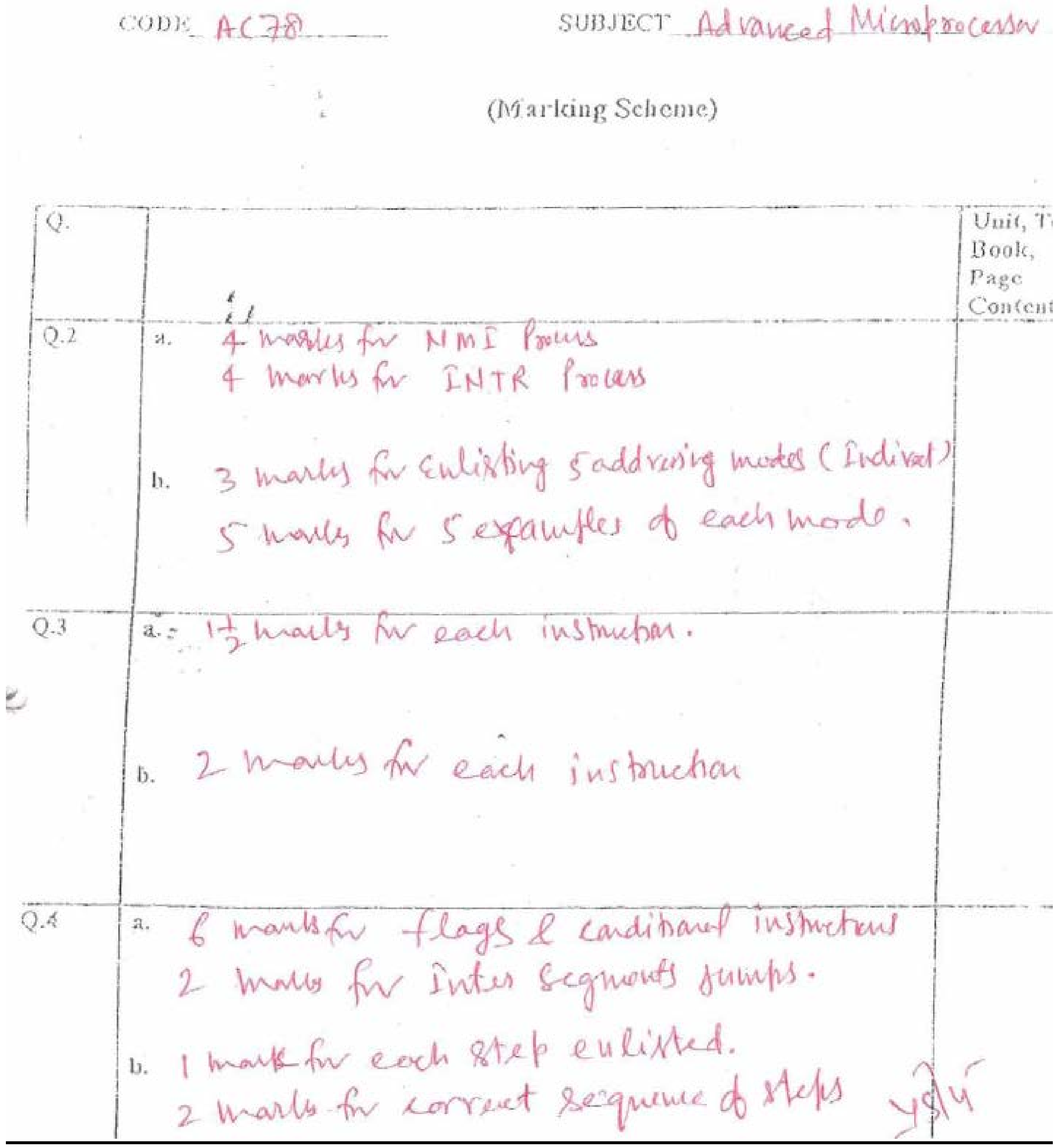


Fig. 9.1 *Register Set of 80286 (Intel Corp.)*

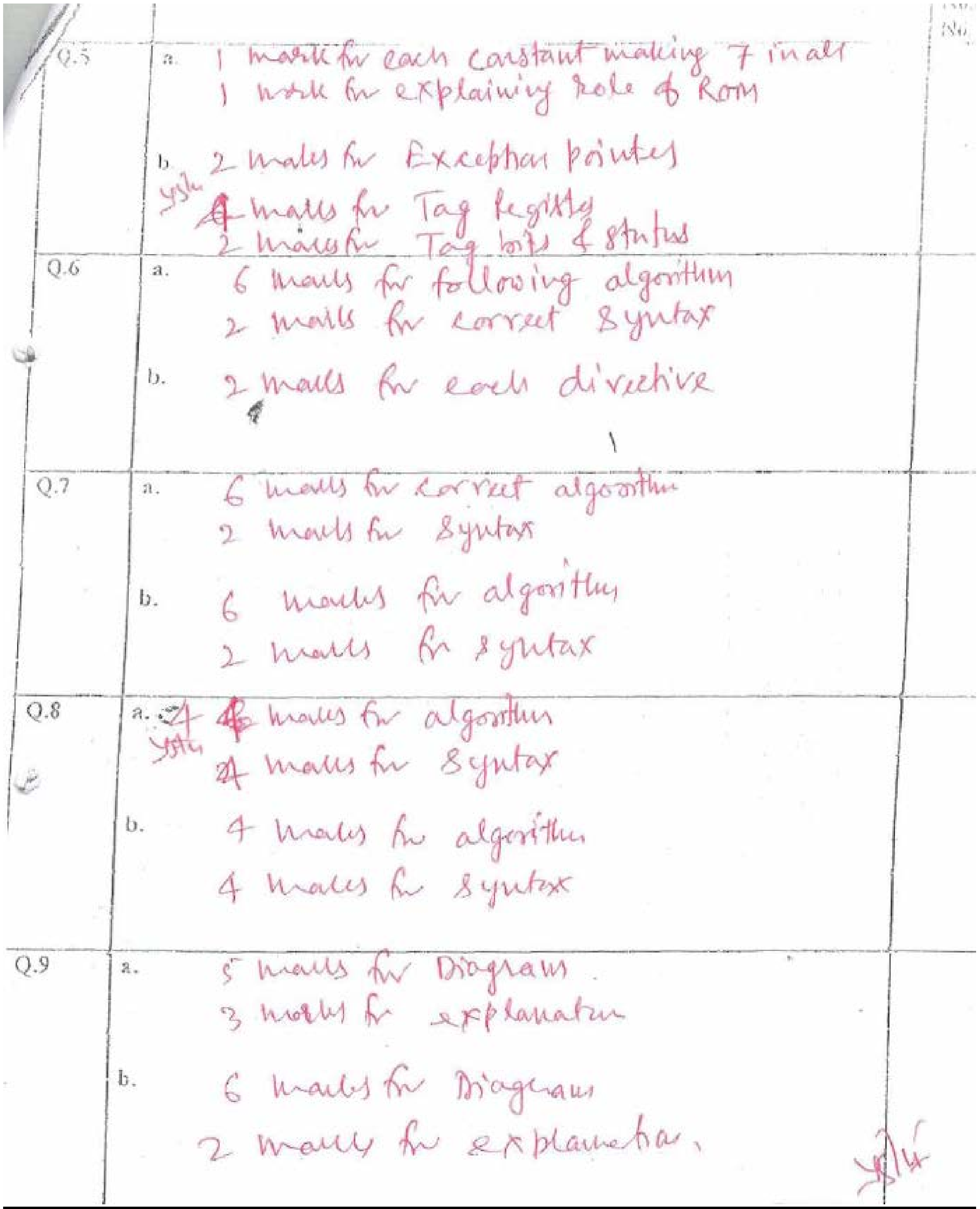
The flag register reflects the results of logical and arithmetic instructions. The flag register bits D_0, D_2, D_4, D_6, D_7 and D_{11} are modified according to the result of the execution of logical and arithmetic instructions. These are called as status flag bits. The bits D_8 and D_9 namely, Trap Flag (TF) and Interrupt Flag (IF) bits, are used for controlling machine operation and thus they are called control flags. All the above discussed flags are also available in 8086. Figure 9.2 shows the flag register of 80286 with the bit definitions, and the additional field definitions.



CODE AC78 SUBJECT Advanced Microprocessor

(Marking Scheme)

Q.		Unit, T Book, Page Content
Q.2	a. 4 marks for NMI Process 4 marks for INTR Process b. 3 marks for enlisting 5 addressing modes (Indival) 5 marks for 5 examples of each mode.	
Q.3	a. 1½ marks for each instruction. b. 2 marks for each instruction	
Q.4	a. 6 marks for flags & conditional instructions 2 marks for inter segments jumps. b. 1 mark for each step enlisted. 2 marks for correct sequence of steps [illegible]	



Q.5	a.	1 mark for each constant making 7 in all 1 mark for explaining role of ROM
	b.	2 marks for Exception pointers 4 marks for Tag Registry 2 marks for Tag bits & status
Q.6	a.	6 marks for following algorithm 2 marks for correct syntax
	b.	2 marks for each directive
Q.7	a.	6 marks for correct algorithm 2 marks for syntax
	b.	6 marks for algorithm 2 marks for syntax
Q.8	a.	4 marks for algorithm 4 marks for syntax
	b.	4 marks for algorithm 4 marks for syntax
Q.9	a.	5 marks for Diagram 3 marks for explanation
	b.	6 marks for Diagrams 2 marks for explanation.

TEXT BOOK

I. Advanced Microprocessors & IBM-PC Assembly Language Programming, K. Udaya Kumar and B.S. Umashankar, TMH, 1996

II. Advanced Microprocessors and Peripherals, A.K. Ray and K.M. Burchandi, TMH, 2000