Scheme Of Evaluation
Internal Assessment Test 2 – April.2019

Sub:	Microprocessor							**Code:**	17EC46
Date:	16/04/2019	Duration:	90mins	Max Marks:	50	**Sem:**	IV	**Branch:**	ECE(A,B,C,D)/ TCE

Note: Answer Any Five Questions

Description	**Marks Distribution**		**Max Marks**	
1	**Explain the following string instruction: (i) CMPSB, (ii) MOVSB, (iii) LODSB, (iv) STOSB, (v) SCASB.** • CMPSB • MOVSB • LODSB • STOSB • SCASB **Solution:** **CMPSB /CMPSW:** It is used to compare a byte(or word) in the data segment with a byte(or word) in the extra segment. The offset address of the source in the data segment should be in SI. The offset address of the destination in the extra segment should be in DI.SI and DI are incremented / decremented after each operation depending upon the direction flag DF in the flag register. Comparison is done by subtracting the byte (or word) in extra segment from the byte(word) in data segment. The flag bits are affected, but the result is not stored anywhere. **Example:** CMPSB ; compare DS:[SI] with ES:[DI] SI ← SI±1 …..depending upon DF DI ← DI±1 …..depending upon DF CMPSW ; compare {DS:[SI], DS:[SI+1]} with {ES:[DI], ES:[DI+1] SI ← SI±2 …..depending upon DF DI ← DI±2 …..depending upon DF IF DF=0, SI and DI are incremented, otherwise decremented. **MOVSB/MOVSW: Move String Byte or String Word** It is used to transfer a word/byte from data segment to extra segment. The offset address of the source in the data segment should be in SI. The offset address of the destination in the extra segment should be in DI. SI and DI are incremented / decremented depending upon the direction flag. Example: MOVSB ; ES:[DI] ← DS:[SI]	2 M 2 M 2 M 2 M 2 M	10 M	10 M

SI ← SI±1 …..depending upon DF

DI ← DI±1 …..depending upon DF

MOVSW ; {ES:[DI], ES:[DI+1] DS:[SI], DS:[SI+1]} ←

DI ← DI±2 …..depending upon DF

SI ← SI±2 …..depending upon DF

IF DF=0, SI and DI are incremented, otherwise decremented.

LODSB/LODSW: Load String Byte or String Word

The LODS instruction loads the AL/AX register by the content of a string pointed by SI in the data segment. SI - incremented / decremented after each operation depending upon the direction flag DF in the flag register.

Example:

LODSB ; AL ← DS:[SI]

LODSW; AL ← DS:[SI], AH DS:[SI+1],

STOSB/STOSW: Store String Byte or String Word

The STOS instruction stores the AL/AX register contents to a location of the string pointed by DI in the extra segment.DI incremented / decremented after each operation depending upon the direction flag DF in the flag register.

Example:

STOSB ; ES:[DI] ← AL

STOSW; ES:[DI] ← AL, ES:[DI+1] ← AH

SCASB/SCASW: Scan String Byte or String Word

This instruction scans string of bytes or words for an operand byte or word specified in the AL or AX register.The offset address of the string in extra segment should be in DI. DI is incremented /decremented after each operation depending upon the direction flag DF in the flag register. Comparison is done by subtracting the byte (or word) in extra segment from AL (AX). The flag bits are affected, but the result is not stored anywhere.

Example:

SCASB ; compare AL with ES:[DI]

DI ← DI±1 …..depending upon DF

SCASW; compare {AX} with {ES:[DI], ES:[DI+1]

DI ← DI±2 …..depending upon DF

If DF=0 SI and DI are incremented, otherwise decremented.

2	**What are assembler directives? Describe the following assembler directives with examples: (i) DB (ii) EQU (iii) DUP (iv) ASSUME (v) ENDS.** • Definition of assembler directive • Description with example • DB • EQU • DUP • ASSUME • ENDS An Assembly language program is a series of statements, or lines. Which contains either assembly language instructions or statements called directives. Assembler Directives (pseudo-instructions) give directions to the assembler about how it should translate the Assembly language instructions into machine code. Assembler directives are non processor executable program instructions which help the assembler to arrange and prepare the code better. i. **DB (Define Byte)**: this directive directs the assembler to reserve byte or bytes of memory locations. Ex: a. num db 25H This statement directs the assembler to reserve 1 byte memory location for a variable or label named num and initialize it with value 25H. b. rank db 01h,02h,03h,04h,05h This statement directs the assembler to reserve 5 byte memory locations for a list named rank and initialize them with values above specified 5 values25H. ii. EQU: (Equate) The directive EQU is used assign a label with a value or symbol. It is used to define a constant without occupying a memory location. Ex: count equ 5 Here equ is used to assign a label count with value 5. iii. DUP (duplicate): DUP will duplicate a given number of characters. Ex: list db 5 dup (25H) Dup directs the assembler to duplicate value 25H in 5 memory locations starting from label list. iv. ASSUME: Assume Logical Segment Name: It tells the assembler what address will be in the segment registers at execution time. Ex: Assume CS:code, DS:Data, ES:Extra v. ENDS:END of Segment Indicates the end of logical segment.	1 M 1 M 2 M 2 M 2 M 2 M	10 M	10 M

	Ex: Data Segment ; Indicates the ;beginning of logical segment named Data Num db 10H Data Ends; Indicates the end of logical segment named Data.			

3	**Explain the following commands: (i) AAM (ii) JNC LABEL (iii) CALL (iv) DAA (v) CLC and (vi) CMC.**
	• AAM
	• JNC LABEL
	• CALL
	• DAA
	• CLC
	• CMC
	AAM – ASCII adjust after Multiplication
	The AAM instruction used after MUL instruction that multiplies two unpacked BCD operands. After the execution of AAM instruction, the product available in AX will be converted into unpacked BCD format.
	Example:
	MOV AL,'9' ; AL=39H
	MOV BL,'8' ; BL=38H
	SUB AL, 30H ; AL=09H
	SUB BL, 30H ; BL=08H
	MUL BL ; AX=0048H
	AAM ;AX= 0702
	Note: AAM instruction does the conversion by dividing AX by 10 or 0AH
	AL= remainder and AH= quotient
	JNC Label:
	Conditional jump execution. If CY Flag=1, the sequence of execution transfers to the location identified by LABEL.
	E.g. JNC L2
	CALL:
	CALL is unconditional Control Transfer (Branch) instruction. This instruction is used to call subroutine (procedure) from a main program.
	CALL instruction transfers the execution control to a subroutine with the intention of coming back to the main program.
	Thus in CALL , 8086 saves the address of next instruction into to the stack before branching to subroutine. At the end of the subroutine, the control is transferred back to the main program using the return address from the stack.
	There are two types of CALL:
	a. Near CALL: The subroutine called must be in the same segment (hence intra - segment).
	; SP← SP-2; IP ← Offset address of subroutine BINtoHEX
	Far CALL: The subroutine called is in the another segment (hence inter - segment).
	Here CS and IP gets new values.
		2 M
		2 M
		2 M
		2 M
		1 M
		1M
			10 M	10 M

DAA -Decimal Adjust after Addition:

DAA works only on AL

- If after an ADD or ADC instruction the lower nibble of AL is greater than **9**, or if **AF = 1**, DAA instruction **adds 06 to the lower nibble of AL**.
- After adding 06, If the upper nibble of AL is greater than **9**, or if **CF = 1**, DAA instruction **adds 6 to the upper nibble of AL.**

For example, adding **29H** and **18H** will result in **41 H**, which is incorrect as far as BCD is concerned.

	Hex	BCD
	29	0010 1001
+	18	+0001 1000
	41	0100 0001
+	6	+ 0110
	47	0100 0111

AF = 1
because AF =1 DAA will add 6 to lower nibble
The final result is BCD.

MNEMONIC	MEANING	OPERATION	Flags Affected
CLC	Clear Carry Flag	(CF) ← 0	CF
CMC	Complement Carry Flag	(CF) ← $(CF)^{1}$	CF

4

Define interrupt and write the sequence of operations that are performed when an interrupt is recognized using neat diagram.

- Definition of Interrupt
- Sequecne of operation
- Diagram

1 M

7 M

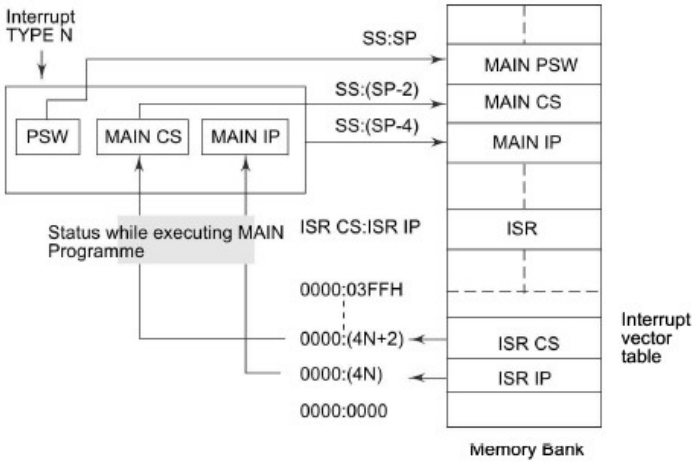
2 M

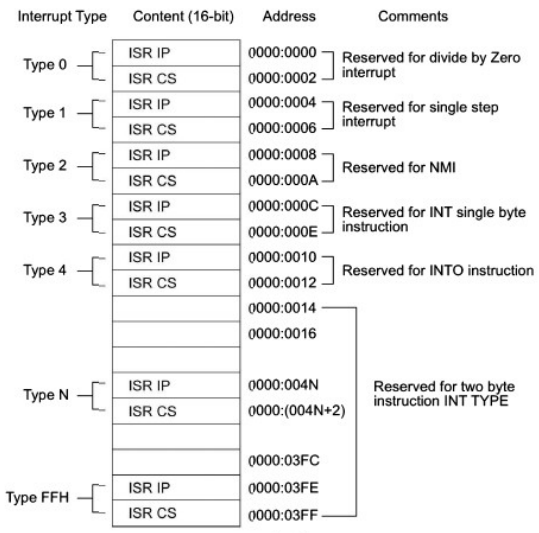
10 M

10 M

Interrupt breaks the normal sequence of execution, diverts execution to ISR. After execution of ISR control returns to the main program.

Suppose an external device interrupts the CPU at the interrupt pin, either NMI or INTR of the 8086, while the CPU is executing an instruction of a program. The CPU first completes the execution of the current instruction. The IP is then incremented to point to the next instruction. The CPU then acknowledges the requesting device on its $\overline{INTA}$ pin immediately if it is a NMI, TRAP or Divide by Zero interrupt. If it is an INT request, the CPU checks the IF flag. If the IF is set , the interrupt request is acknowledged using the $\overline{INTA}$ pin. If the IF is not set, the interrupt requests are ignored. Note that the responses to the NMI, TRAP and Divide by Zero interrupt requests are independent of the IF flag. After an interrupt is acknowledged , the CPU

	computes the vector address from the type of the interrupt that may be passed to the interrupt structure of the CPU internally (in case of software interrupts, NMI, TRAP and Divide by Zero interrupts) or externally, i.e. from an interrupt controller in case of external interrupts. (The contents of IP and CS are next pushed to the stack. The contents of IP and CS now point to the address of the next instruction of the main program from which the execution is to be continued after executing the ISR. The PSW is also pushed to the stack.) The Interrupt Flag (IF) is cleared. The TF is also cleared, after every response to the single step interrupt. The control is then transferred to the interrupt service routine for serving the interrupting device. The new address of ISR is found out from the interrupt vector table. The execution of the ISR starts. If further interrupts are to be responded to during the time the first interrupt is being serviced, the IF should again be set to 1 by the ISR of the first interrupt. If the interrupt flag is not set, the subsequent interrupt signals will not be acknowledged by the processor, till the current one is completed. The programmable interrupt controller is used for managing such multiple interrupts based on their priorities. At the end of ISR the last instruction should be IRET. When the CPU executes IRET , the contents of flags, IP and CS which were saved at the start by the CALL instruction are now retrieved to the respective registers. The execution continues onwards from this address, received by IP and CS.  			
5a)	**Draw the interrupt vector table. Mention dedicated interrupts with respect to 8086.** • Interrupt vector table diagram • Explanation of dedicated interrupts	3 M 3 M	6 M	10 M



ISR: Interrupt Service Routine

8086 supports a total of 256 types of the interrupts, i.e. from 00 to FFH. Each interrupt requires 4 bytes, i.e. two bytes each for IP and CS of its ISR. Thus a total of 1,024 bytes are required for 256 interrupt types, hence the interrupt vector table starts at location 0000:0000 and ends at 0000:03FFH. The interrupt vector table contains the IP and CS of all the interrupt types stored sequentially from address 0000:0000 to 0000:03FF H. The interrupt type N is multiplied by 4 and the hexadecimal multiplication obtained gives the offset address in the zeroeth code segment at which the IP and CS addresses of the interrupt service routine (ISR) are stored. The execution automatically starts from the new CS:IP.

Dedicated interrupts:

1.INT 0 (Divide by zero):

This interrupt occurs whenever there is division error. i.e. When the result of division is too large to be stored.

This condition normally occurs when the divisor is very small as compared to the dividend or the divisor is zero.

Its ISR address is stored at location 0 x4=00000in the IVT.

2. INT 1 (single step):

The microprocessor executes this interrupt after every instruction if TF is set.

It puts the microprocessor in single stepping mode i.e. µp pauses after executing every instruction.

This is very useful during debugging.

Its ISR generally displays contents of all registers.

Its ISR address is stored at location 1 x 4=00004 in the IVT.

3. INT 2: (Non-maskable interrupt)
The µp executes this ISR in response to an interrupt on NMI line.
Its ISR address is stored at location 2 x 4=00008H in the IVT .

4. INT 3: (Breakpoint interrupt):
It is used to create the breakpoints in the program.
It is used for debugging large programs where the single stepping is inefficient.
Its ISR address is stored at location 3 x 4=0000CH in the IVT .

5. INT 4: (Overflow interrupt):
This interrupt occurs if the overflow flag is set and the µp executes INTO instruction(Interrupt on overflow).
It is used to detect overflow error in signed arithmetic operations.
Its ISR address is stored at location 4 x 4=00010H in the IVT .

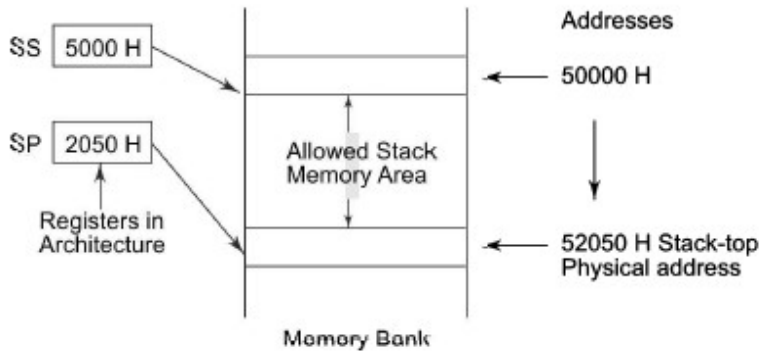
5(b)Explain stack structure of 8086 in detail.

- PUSH operation
- POP operations

PUSH Operation:

Let the content of SS be 5000 H and the content of the stack pointer register be 2050 H. To find c current stack-top address, the stack segment register content is shifted left by four bit positions (multipl 10 H) and the resulting 20-bit content is added with the 16-bit offset value, stored in the stack pointer re In the above case, the stack top address can be calculated as shown:

Thus the stack top address is 52050 H. Figure 4.1 makes the concept more clear.



2 M

2 M

4 M

If the stack top points to a memory location 52050 H, it means that the location 52050 H is alreac cupied, i.e. previously pushed data is available at 52050 H. The next 16-bit push operation will decr the stack pointer by two, so that it will point to the new stack-top 5204EH, and the decremented conte SP will be 204E H. This location will now be occupied by the recently pushed data. Thus, if a 16-bi is pushed onto the stack, the push operation will decrement the SP by two because two locations will quired for a 2-byte (16-bit) data. Thus it may be noted here that the stack grows down.

Thus for a selected value of SS, the maximum value of SP = FFFF H and the segment can have max of 64K locations. Thus after starting with an initial value of FFFFH, the Stack Pointer (SP) is decrem by two, whenever a 16-bit data is pushed onto the stack. After successive push operations, when the Pointer contains 0000 H, any attempt to further push the data to the stack will result in stack overflow.

	POP operation: Suppose, a main program is being executed by the processor. At some stage during the execution of the program, all the registers in the CPU may contain useful data. In case there is a subroutine CALL instruction at this stage, there is a possibility that all or some of the registers of the main program may be modified due to the execution of the subroutine. This may result in loss of useful data, which may be avoided by using the stack. At the start of the subroutine, all the registers' contents of the main program may be pushed onto the stack one by one. After each PUSH operation SP will be modified as already explained before. Thus all the registers can be copied to the stack. Now these registers may be used by the subroutine, since their original contents are saved onto the stack. At the end of the execution of the subroutine, all the registers can get back their original contents by popping the data from the stack. The sequence of popping is exactly the reverse of the pushing sequence. In other words, the register or memory location that is pushed into the stack at the end should be popped off first.			

6	**Write an ALP which replaces all occurrences of character '-' in a given string by '*'.**
	• Template
	• Algorithm
	.MODEL SMALL
	.STACK 64H
	.DATA
	STAR DB '*'
	DASH DB '-'
	BLOCK1 DB 'C-M-R-I-T$'
	COUNT EQU ($-BLOCK1)
	.CODE
	MOV AX,@DATA
	MOV DS,AX
	MOV DL,STAR
	MOV BL,DASH
	MOV CX, COUNT
	MOV SI, OFFSET BLOCK1
	L1: MOV AL,[SI]
	CMP AL,BL
	JZ L2
	INC SI
	JMP L3
	L2: MOV [SI],DL
	INC SI
	L3: LOOP L1
	
	MOV AH,4CH
	INT 21H
	END
		4 M
		6 M
			10 M	10 M

7	**Copy 100 bytes of data from LOC1 to LOC2 using MOVS instruction. Give the significance of SI, DI, CX and DF bit.**
	• Template
	• Algorithm
	• Significance of SI, DI, CX and DF
	.MODEL SMALL
	.STACK 64H
	.DATA
	LOC1 DB 100 DUP(0)
	LOC2 DB 100 DUP('?')
	COUNT EQU 100
	.CODE
	MOV AX,@DATA
	MOV DS,AX
	MOV ES,AX
	
	MOV CX, COUNT
	MOV SI, OFFSET LOC1
	MOV DI, OFFSET LOC2
	CLD
	REP MOVSB
	
	MOV AH,4CH
	INT 21H
	END
	As the direction flag is cleared in the program, SI and DI are incremented after every MOVSB instruction.
	CX register value decides how many times REP will execute. E.g. if initially CX is loaded with 5 REP MOVSB will repeat 5 times. Every REP instruction decrements CX register value by 1.
	DF=0, SI and DI are auto incremented on execution of string instruction, otherwise they are auto decremented,
		3 M
		4 M
		3 M
			10 M	10 M

8	**A two digit BCD number is typed using a keyboard. Write an ALP to read the value, save it as BCD number at LOC as packed BCD.**		10 M	10 M

	• Template
	• Algorithm
	.MODEL SMALL
	.STACK 64H
	.DATA
	LOC DB 1 DUP(0)
	.CODE
	MOV AX,@DATA
	MOV DS,AX
	
	MOV AH,01H
	INT 21H
	
	SUB AL,30H
	MOV BH,AL
	
	MOV AH,01H
	INT 21H
	
	SUB AL,30H
	MOV BL,AL
	
	MOV CL,04
	ROL BH,AL
	OR BL,BH
	
	MOV SI, OFFSET LOC
	MOV [SI],BL
	MOV AH,4CH
	INT 21H
	END
		4
		M
		6
		M		