



## **PRACTICAL FILE**

# **MICROPROCESSORS AND COMPUTER ARCHITECTURE**

### **SUBMITTED TO:**

Ms. Pradnya Amrish Gajbhiye  
Dept. Of ECE  
NSUT

### **SUBMITTED BY:**

Akshay Bhardwaj  
2020UEC2548  
ECE-1



# INDEX

1. Introduction to Microprocessor 8086 and its Architecture
2. Write a program to arrange the 10 numbers in SRAM memory in ascending order. Repeat your experiment to arrange the data in descending order.
3. Develop a subroutine for Multiply and divide operations.
4. Move a block of data from a source address location to target address location using assembly language programming in 8086.
5. Write a program to find whether a number is even or odd.
6. Write a program to find the smallest and largest numbers from a list.
7. Write a subroutine to convert ASCII to Binary, Binary to ASCII, BCD to Binary, and Binary to BCD.
8. Study the operation of 8255 Interface.
9. To study the operation of the 8259 interface.

# EXPERIMENT-1

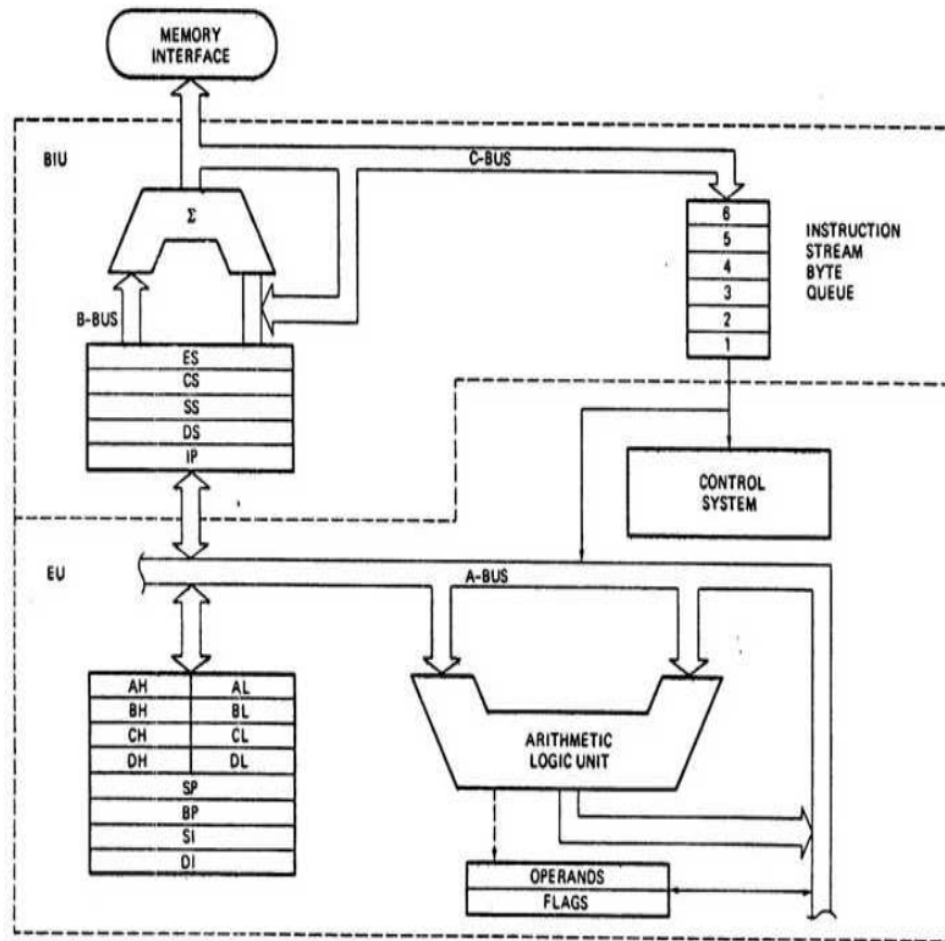
## Introduction To Microprocessor 8086 and its Architecture

8086 Microprocessor is an enhanced version of 8085 Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and 16 data lines that provide up to 1MB of storage. It consists of a powerful instruction set, which provides operations like multiplication and division easily. It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for systems having multiple processors and Minimum mode is suitable for systems having a single processor.

The most prominent features of an 8086 microprocessor are as follows –

- It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
- It was the first 16-bit processor to have 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
- It is available in 3 versions based on the frequency of operation –
  - 8086 → 5MHz
  - 8086-2 → 8MHz
  - 8086-1 → 10 MHz
- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.
- Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.
- Execute stage executes these instructions.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.

## Architecture of 8086



As 8086 does 2-stage pipelining, its architecture is divided into two units:

1. Bus Interface Unit (BIU)
2. Execution Unit (EU)

## **BUS INTERFACE UNIT (BIU)**

1. It provides the interface of 8086 to other devices.
2. It operates w.r.t. Bus cycles .

This means it performs various machine cycles such as Mem Read, IO Write etc to transfer data with Memory and I/O devices.

3. It performs the following functions:
  - a) It generates the 20-bit physical address for memory access.
  - b) Fetches Instruction from memory.
  - c) Transfers data to and from the memory and IO.
  - d) Supports Pipelining using the 6-byte instruction queue.

Its main components are:

- a) Segment Registers
- b) Address Generation Circuit
- c) 6-Byte Pre-Fetch Queue

## **Execution Unit (EU)**

1. It fetches instructions from the Queue in BIU, decodes and executes them.
2. It performs arithmetic, logic and internal data transfer operations.
3. It sends request signals to the BIU to access the external module.
4. It operates w.r.t. T-States (clock cycles)

Its main components are:

- a) General Purpose Registers
- b) Arithmetic and logical operations Unit
- c) Operand Register
- d) Instruction Register and Instruction Decoder
- e) Flag Register

# Introduction to EMU 8086

EMU8086 (Microprocessor emulator) is a free emulator for multiple platforms. It provides its user with the ability to emulate old 8086 processors, which were used in Macintosh and Windows computers from the 1980s and early 1990s. It can emulate a large amount of software that was used on these microprocessors, but a savvy user can also program their own assembly code to run on it.

Working:

EMU8086 - MICROPROCESSOR EMULATOR primarily emulates the processor, not the other functions that a microcomputer running a 8086 processor would have. However, it still serves many of the same functions that an emulator for a more specific microcomputer might have, and more besides. For example, both the NEC-P9801 and early IBM-compatible computers used the 8086. Using EMU8086, one might be able to write assembly software that can run on either of those devices. On the flip side, EMU8086 can't access some of the more advanced hardware functionality that you might find in the monitors or other components of those devices.

Overall, EMU8086 - MICROPROCESSOR EMULATOR will be useful to computing enthusiasts and gearheads, and anyone who happens to work with this legacy processor even today: some computers, particularly in business and industrial applications, still use the 8086.

PROS:

- Complete emulation
- Useful for assembly programming

CONS:

- Limited hardware emulation
- Some setup required

## **EXPERIMENT-2**

### **Aim:**

Write a program to arrange the 10 numbers in SRAM memory in ascending order. Repeat your experiment to arrange the data in descending order.

**Software Used:** EMU8086

### **Theory:**

To arrange numbers in ascending and descending order we use the bubble sorting technique. In this sorting technique, there will be n passes for n different numbers. In ith pass, the ith largest element will be placed at the end or in the beginning depending on the sorting order. This is a comparison-based sort. We take two consecutive numbers, compare them, and then swap them if the numbers are not in correct order. Here we first initialise the data segment along with its offset pointer (SI) and store a list of numbers in the memory (starting at 42000H) then we initialise the count of the inner and outer loops as size-1. Finally, we do pairwise comparisons (Bubble Sort Algorithm) and sort the list.

**Code:**

MEMORY ADDRESS	MNEMONICS	COMMENT
400	MOV SI, 500	SI<-500
403	MOV CL, [SI]	CL<-[SI]
405	DEC CL	CL<-CL-1
407	MOV SI, 500	SI<-500
40A	MOV CH, [SI]	CH<-[SI]
40C	DEC CH	CH<-CH-1

40E	INC SI	SI<-SI+1
40F	MOV AL, [SI]	AL<-[SI]
411	INC SI	SI<-SI+1
412	CMP AL, [SI]	AL-[SI]
414	JC 41C	JUMP TO 41C IF CY=1
416	XCHG AL, [SI]	SWAP AL AND [SI]
418	DEC SI	SI<-SI-1
419	XCHG AL, [SI]	SWAP AL AND [SI]
41B	INC SI	SI<-SI+1

41C	DEC CH	CH<-CH-1
41E	JNZ 40F	JUMP TO 40F IF ZF=0
420	DEC CL	CL<-CL-1
422	JNZ 407	JUMP TO 407 IF ZF=0
424	HLT	END



**Input:**

Random Access Memory

4000:2000      update      ☒ table      ☐ list

4000:2000	05	04	03	02	01	00	00	00-00	00	00	00	00	00	00
4000:2010	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00
4000:2020	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00
4000:2030	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00
4000:2040	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00
4000:2050	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00
4000:2060	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00
4000:2070	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00

**Output:**

Random Access Memory

4000:2000

update

table

list

4000:2000	01	02	03	04	05	00	00	00-00	00	00	00	00	00	00	00
4000:2010	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2020	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2030	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2040	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2050	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2060	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2070	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00

## Conclusion:


Successfully arranged 5 numbers in SRAM memory in ascending order.

## Descending order:

Code:

MEMORY ADDRESS	MNEMONICS	COMMENT
400	MOV SI, 500	SI < -500
403	MOV CL, [SI]	CL < -[SI]
405	DEC CL	CL < -CL-1
407	MOV SI, 500	SI < -500
40A	MOV CH, [SI]	CH < -[SI]
40C	DEC CH	CH < -CH-1
40E	INC SI	SI < -SI+1
40F	MOV AL, [SI]	AL < -[SI]
411	INC SI	SI < -SI+1
412	CMP AL, [SI]	AL-[SI]
414	JNC 41C	JUMP TO 41C IF CY!=1
416	XCHG AL, [SI]	SWAP AL AND [SI]
418	DEC SI	SI < -SI-1
419	XCHG AL, [SI]	SWAP AL AND [SI]
41B	INC SI	SI < -SI+1
41C	DEC CH	CH < -CH-1
41E	JNZ 40F	JUMP TO 40F IF ZF=0
420	DEC CL	CL < -CL-1
422	JNZ 407	JUMP TO 407 IF ZF=0
424	HLT	END

## Input:

 Random Access Memory

4000:2000

update

☒ table

☐ list

4000:2000	01	09	10	69	96	00	00	00-00	00	00	00	00	00	00	00
4000:2010	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2020	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2030	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2040	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2050	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2060	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00
4000:2070	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00

## Output:

Random Access Memory

4000:2000

update

☒ table

☐ list

4000:2000	96	80	69	09	01	00	00	00	00-00	00	00	00	00	00	00	00	ûÇi.©.....
4000:2010	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2020	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2030	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2040	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2050	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2060	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2070	00	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....

## Conclusion:

Successfully arranged 5 numbers in SRAM memory in descending order.

# EXPERIMENT-3

## Aim:

Develop a subroutine for Multiply and divide operations.

**Software Used:** EMU8086

## Theory:

To create a subroutine we first initialize the data segment along with its offset pointer (SI) and assign data to the memory location 42000H. Subroutines can have 2 types of branches near and far, In the near branch, the information present in the subroutine is in the same segment that calls the subroutine (like in this case both are in the code segment). In the far branch, the information present in the subroutine is not in the same segment that calls the subroutine. Finally, we use the MUL and DIV commands to multiply or divide the numbers present in AL and BL registers.

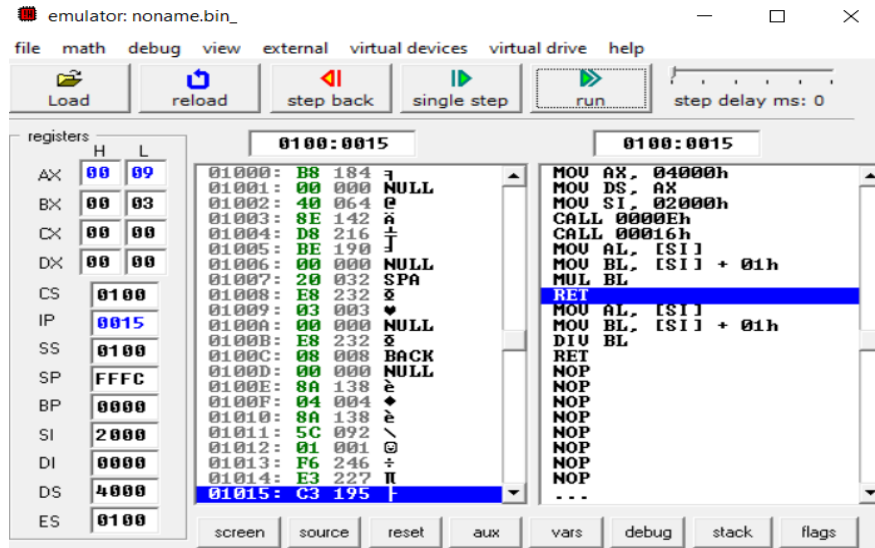
## Code:

```
01 MOV AX,5000H
02 MOV DS,AX
03 MOV AX,50H
04 CALL L1
05 CALL L2
06 MOV AH,4CH
07 INT 21H
08 L1: MOV AL,02H
09     MOV BL,02H
10     MUL BL
11     RET
12 L2: MOV AX,0080H
13     MOV BL,04H
14     DIV BL
15     RET
```

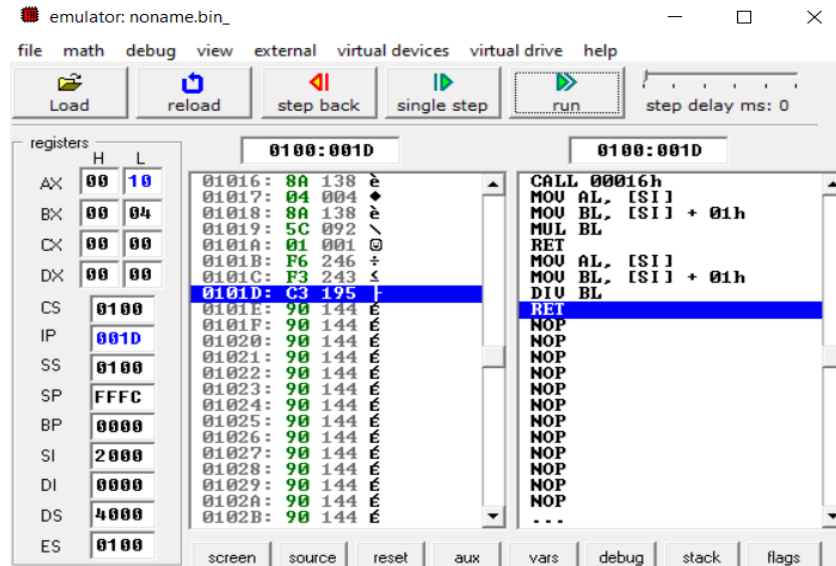


## Output:

a) For multiplication



b) For Division.



## Conclusion:

Successfully implemented multiplication and division operation on two numbers.

# EXPERIMENT-4

## Aim–

Move a block of data from a source address location to target address location using assembly language programming in 8086

## Theory –

To move a block of memory/ code from data segment to extra segment we use the MOVSB command. Firstly we initialize the data and extra segments along with their offset pointers( SI and DI respectively)and load the data at the memory locations after that we put the size of the block in the CX register. Then the direction of copying is specified (here the direction is from top to bottom as specified by CLD) Finally the MOVSB command is used repeatedly.

## Code:

Memory	Mnemonics	Operands	Comment
2000	MVI	C, 05	[C] <- 05
2002	LXI	H, 2500	[H-L] <- 2500
2005	LXI	D, 2600	[D-E] <- 2600
2008	MOV	A, M	[A] <- [[H-L]]
2009	STAX	D	[A] -> [[D-E]]
200A	INX	H	[H-L] <- [H-L] + 1
200B	INX	D	[D-E] <- [D-E] + 1
200C	DCR	C	[C] <- [C] - 1
200D	JNZ	2008	Jump if not zero to 2008
2010	HLT		Stop

## Input:

☒ Random Access Memory

4000:2000

update

☒ table ☐ list

42005

4000:2000	69	58	90	19	70	00	00	00-00	00	00	00	00	00	00	00	ix↓p.....
4000:2010	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2020	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2030	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2040	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2050	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2060	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....
4000:2070	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	.....

## Output:

☒ Random Access Memory

3000:6000

update

☒ table ☐ list

3000:6000	69	58	90	19	70	00	00	00-00	00	00	00	00	00	00	00	ix↓p.....
3000:6010	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..
3000:6020	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..
3000:6030	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..
3000:6040	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..
3000:6050	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..
3000:6060	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..
3000:6070	00	00	00	00	00	00	00	00-00	00	00	00	00	00	00	00	..

## Conclusion:

Successfully moved a block of data from a source address location to target address location using assembly language programming in 8086



# EXPERIMENT-5

## Aim:

Write a program to find whether a number is even or odd.

**Software Used:** EMU8086

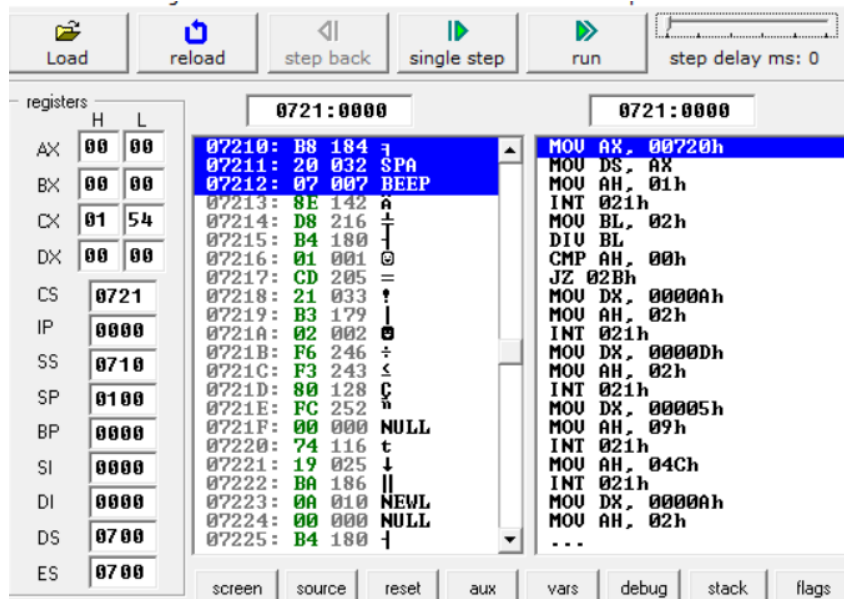
## Theory:

To find whether a number is odd or even we store that number in the memory (here at the location 42000H) then we initialize 2 registers BL and BH initially to zero and if the number is odd BL is set to 1, otherwise if the number is even BH is set to 1. To check for odd and even we take right to rotate the number and if the LSB of the number is 1 then the number is odd else if it is 0 then the number is even. Below is the code for the same.

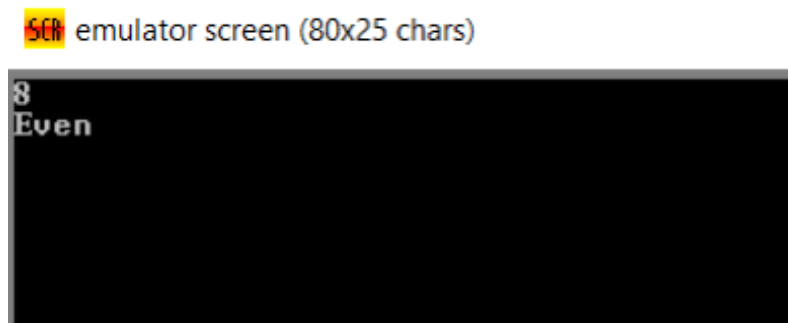
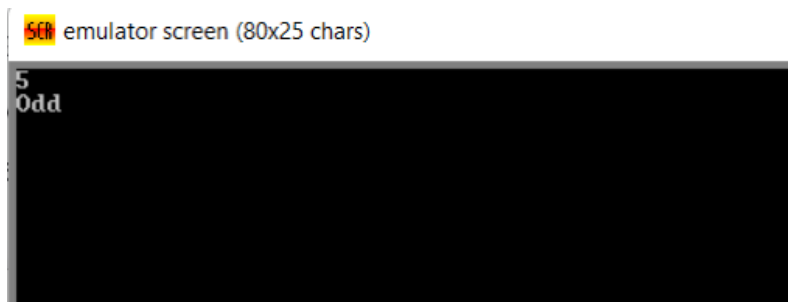
## Code:

```
01 .model small
02 .stack 100h
03 .data
04
05 ev db 'Even$' ; create a variable to ev to display even
06 od db 'Odd$' ; create a variable to od to display odd
07 .code
08 main proc
09 mov ax,@data ; move data segment address into ax
10 mov ds,ax ; move ax->ds
11 mov ah,1 ; to take input on display window
12 int 21h
13 mov bl,2 ; to display output on window
14 div bl ; divide content of bl
15 cmp ah,0 ; compare remainder
16 je IsEven ; if remainder == 0 ,jump to even
17 mov dx,10 ; move cursor to next line
18 mov ah,2
19 int 21h
20 mov dx,13
21 mov ah,2 ;else it os odd
22 int 21h
23 mov dx,offset od ; move address of od into ds
24 mov ah,9
25 int 21h
26 mov ah,4ch ;to out of register|
27 int 21h
28 IsEven:
29 mov dx,10
30 mov ah,2
31 int 21h
32 mov dx,13
33 mov ah,2
34 int 21h
35 mov dx,offset ev
36 mov ah,9
37 int 21h
38 mov ah,4ch
39 int 21h
40
41 main endp
42 end main
```

## Input and upcodes :



## Output:



## Conclusion:

Successfully detected whether a number is even or odd.

# EXPERIMENT-6

## Aim:

Write a program to find the smallest and largest numbers from a list.

## Theory:

To find the largest and smallest numbers we first store a list of numbers in the memory then we store the first number of the list in the memory and loop around the whole list by pairwise comparing with the previous element to find the largest or smallest number in the whole list.

## Code:

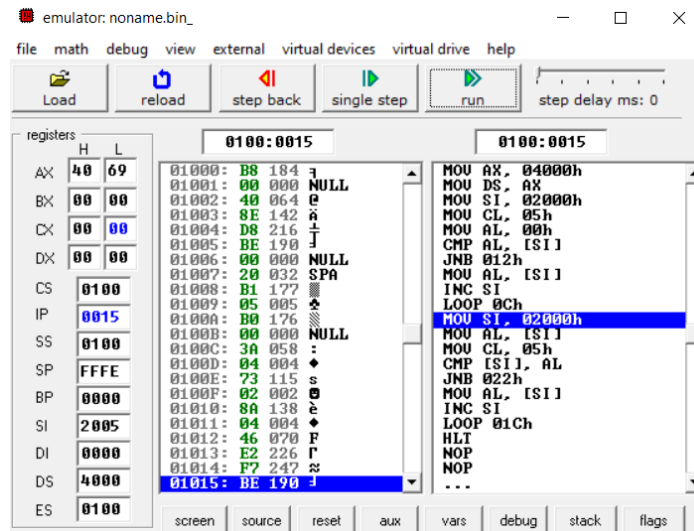
MEMORY ADDRESS	MNEMONICS	COMMENT
400	MOV SI, 500	SI←500
403	MOV CL, [SI]	CL←[SI]
405	MOV CH, 00	CH←00
407	INC SI	SI←SI+1
408	MOV AL, [SI]	AL←[SI]
40A	DEC CL	CL←CL-1
40C	INC SI	SI←SI+1
40D	CMP AL, [SI]	AL-[SI]
40F	JNC 413	JUMP TO 413 IF CY=0
411	MOV AL, [SI]	AL←[SI]
413	INC SI	SI←SI+1
414	LOOP 40D	CX←CX-1 & JUMP TO 40D IF CX NOT 0
416	MOV [600], AL	AL→[600]
41A	HLT	END

## Input:

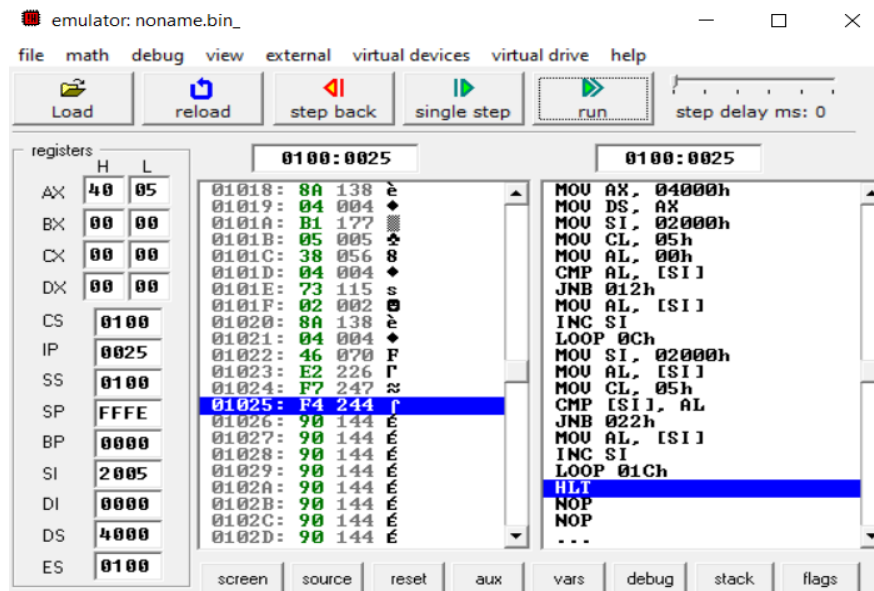
Random Access Memory		— □ ×	
4000:2000	update	table	list
4000:2000	56 60 69 05 08 00 00 00-00 00 00 00 00 00 00 00	U' i . . . . .	
4000:2010	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	. . . . .	
4000:2020	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	. . . . .	
4000:2030	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	. . . . .	
4000:2040	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	. . . . .	
4000:2050	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	. . . . .	

## Output:

### a) Greatest Number



### b) Smallest Number



## Conclusion:

Successfully detected the smallest and largest numbers from a list.

# EXPERIMENT-7

## Aim:

Write subroutines to convert ASCII to binary, binary to ASCII.

**Software Used :** EMU8086

## Theory:

We know that the ASCII of the number 00H is 30H (48D), and the ASCII of 09H is 39H (57D). So all other numbers are in the range 30H to 39H. The ASCII value of 0AH is 41H (65D) and the ASCII of 0FH is 46H (70D), so all other alphabets (B, C, D, E, F) are in the range 41H to 46H. We are checking whether the ASCII value is less than 58H (ASCII of 9+1). When the number is less than 58, then it is a numeric value. So we simply subtract 30H from the ASCII value, and when it is greater than 58H, then it is an alphabetical value. So, for that we are subtracting 37H.

## Code:

Memory Address	Mnemonics	Comments
400	MOV AL, [2000]	AL<-[2000]
404	MOV AH, AL	AH<-AL
406	AND AL, 0F	AL<- (AL AND 0F)
408	MOV CL, 04	CL<- 04
40A	SHR AH, CL	Shift AH content Right by 4 bits (value of CL)
40C	OR AX, 3030	AX<- (AX OR 3030)
40F	MOV [3000], AX	[3000]<-AX
413	HLT	Stop Execution

Memory	Mnemonics	Operands	Comment
2000	MOV	AL, [2050]	[AL] <- [2050]
2004	AND	AL, 0F	[AL] <- ([AL] AND 0F)
2006	MOV	[3050], AL	[3050] <- [AL]
200A	HLT		Stop

## Input:

extended value viewer

watch: MEM: 4200 : 0000

word byte

hex: 21

bin: 00100001

oct: 041

decimal 8 bit

unsigned: 33

signed: 33

ascii: ?

Random Access Memory

4000:2000 update table list 42002

4000:2000	21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	!
4000:2010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
4000:2020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
4000:2030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
4000:2040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
4000:2050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
4000:2060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
4000:2070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

## Output:

emulator: noname.bin\_

file math debug view external virtual devices virtual drive help

Load reload step back single step run step delay ms: 0

registers

AX	40 32	0101B: 88 136 €	MOV [SI], AL
BX	00 21	0101C: 04 004 ♦	HLT
CX	00 04	0101D: F4 244 ¶	AND AL, 0Fh
DX	00 00	0101E: 24 036 \$	CMP AL, 0Ah
		0101F: 0F 015 *	JB 026h
		01020: 3C 060 <	ADD AL, 07h
		01021: 0A 010 <	ADD AL, 030h
		01022: 72 114 r	RET

Random Access Memory

4000:2000 update table list

4000:2000	31 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4000:2070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

screen source reset aux vars debug stack flags

emulator: noname.bin\_

file math debug view external virtual devices virtual drive help

Load reload step back single step run step delay ms: 0

registers

AX	40 D2	0101B: 88 136 €	MOV [SI], AL
BX	00 21	0101C: 04 004 ♦	HLT
CX	00 04	0101D: F4 244 ¶	AND AL, 0Fh
DX	00 00	0101E: 24 036 \$	CMP AL, 0Ah
		0101F: 0F 015 *	JB 026h
		01020: 3C 060 <	SUB AL, 07h
		01021: 0A 010 <	SUB AL, 030h
		01022: 72 114 r	RET
		01023: 02 002 0	NOP
		01024: 2C 044 ,	NOP
		01025: 07 007 BEEP	NOP
		01026: 2C 044 ,	NOP
		01027: 30 048 0	NOP
		01028: C3 195 †	NOP
		01029: 90 144 €	NOP
		0102A: 90 144 €	NOP
		0102B: 90 144 €	NOP
		0102C: 90 144 €	NOP
		0102D: 90 144 €	NOP
		0102E: 90 144 €	NOP
		0102F: 90 144 €	NOP
		01030: 90 144 €	...

screen source reset aux vars debug stack flags

## Conclusion:

Successfully converted ASCII to binary, binary to ASCII.

# EXPERIMENT-8

## **Aim–**

Study the operation of the 8255 Interface.

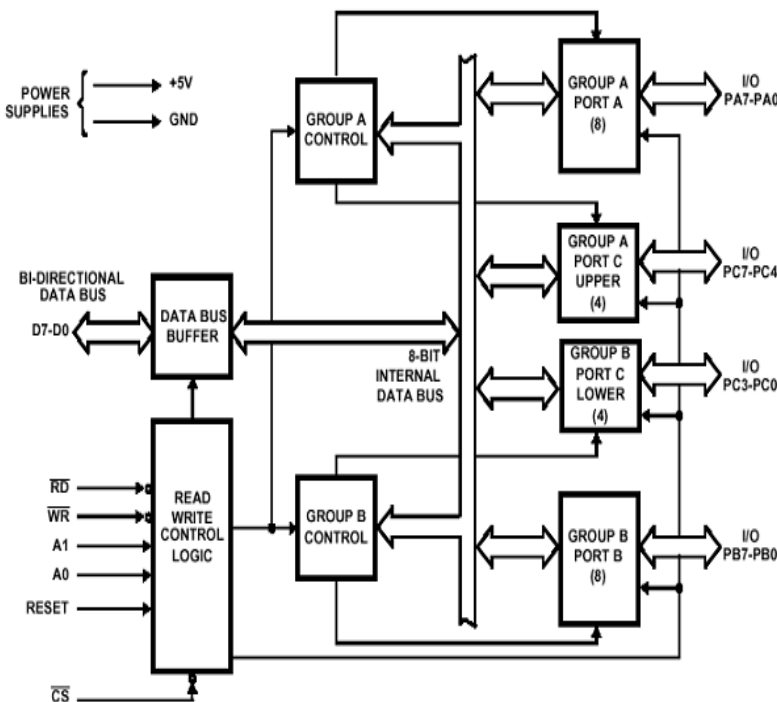
## **Theory –**

8255 Programmable Interface Card is a general-purpose programmable I/O device that can be used with many different microprocessors for the purpose of data transfer, in any 8/16-bit slot. It can be programmed by the system software. The 24 I/O pins can be programmed in 3 groups of 8 port A, port B and port C; thus, providing a 3-state bi-directional buffer that interfaces with the microprocessor's data bus.

## **Interfacing of 8255 with 8086**

- 1) 8255 is a programmable peripheral interface. It is used to interface microprocessors with I/O devices via three ports: PA, PB, PC. All ports are 8-bit and bidirectional.
- 2) 8255 transfers data with the microprocessor through its 8-bit data bus
- 3) The two address lines A1 and A0 are used to make an internal selection in 8255. They can have 4 options, selecting PA, PB, PC or the control word. The ports are selected to transfer data. The Control Word is selected to send commands.
- 4) Two commands can be sent to 8255, called the I/O command and the BSR command. I/O command is used to initialize the mode and direction of the ports. BSR command is used to set or reset a single line of Port C.
- 5) 8255 has three operational modes of data transfer.
- 6) Mode 0 is a simple data transfer mode. It does not perform handshaking but all three ports are available for data transfer.
- 7) Mode 1 performs unidirectional handshaking. That makes transfers more reliable. Port C lines are used by Port A and Port B to perform Handshaking.
- 8) Mode 2 performs bidirectional handshaking. Only Port A can operate in Mode 2. At that time Port B can operate in Mode 1 or Mode 0. Port C lines are again used up for performing Handshaking for Port A and Port B

## Architecture of 8255



The architecture of 8255 can be divided into the following parts:

- **Data Bus Buffer:** This is a 8-bit bi-directional buffer used to interface the internal data bus of 8255 with the external (system) data bus. The CPU transfers data to and from the 8255 through this buffer
- **Read/Write Control Logic** It accepts address and control signals from the  $\mu P$ : The Control signals determine whether it is a read or a write operation and also select or reset the 8255 chip. The address bits (A1, A0) are used to select the Ports or the Control Word Register as shown

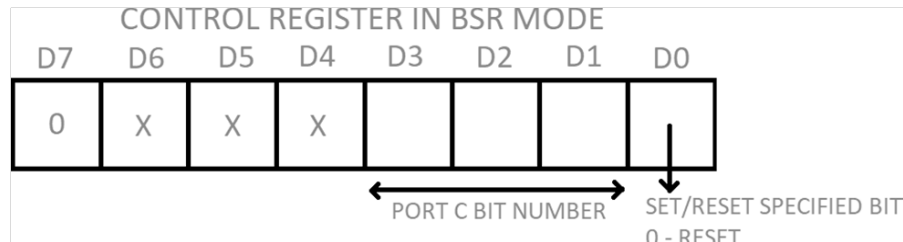
**The Ports are controlled by their respective Group Control Registers.**

- 3) **Group A Control** This Control block controls Port A and Port C-Upper i.e., PC7-PC4. It accepts Control signals from the Control Word and forwards them to the respective Ports.
- 4) **Group B Control** This Control block controls Port B and Port C-Lower i.e., PC3- PC0. It accepts Control signals from the Control Word and forwards them to the respective Ports.
- 5) **Port A, Port B, Port C** These are 8-bit Bi-directional Ports. They can be programmed to work in the various modes as follows:

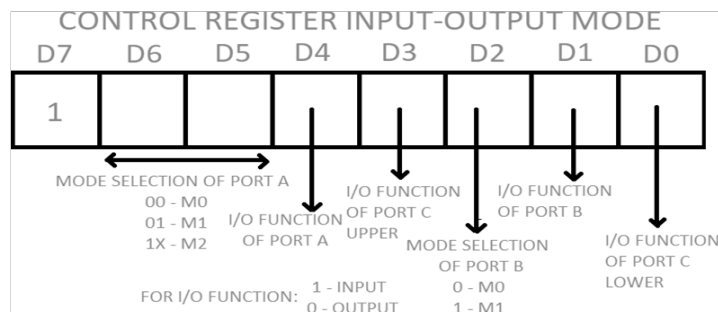


## Operating modes –

**Bit set-reset (BSR) mode** – If MSB of control word (D7) is 0, PPI works in BSR mode. In this mode only port C bits are used to a set or reset



**Input-Output mode** – If MSB of control word (D7) is 1, PPI works in input-output mode. This is further divided into three modes:



**Mode 0** – In this mode all the three ports (port A, B, C) can work as a simple input function or simple output function. In this mode there is no interrupt handling capacity.

**Mode 1** – Handshake I/O mode or strobed I/O mode. In this mode either port A or port B can work as a simple input port or simple output port, and port C bits are used for handshake signals before actual data transmission. It has interrupt handling capacity and input and output are latched. Example: A CPU wants to transfer data to a printer. In this case since speed of the processor is very fast as compared to relatively slow printer, so before actual data transfer it will send handshake signals to the printer for synchronization of the speed of the CPU and the peripherals

**Mode 2** – Bi-directional data bus mode. In this mode only port A works, and port B can work either in mode 0 or mode 1. 6 bits port C are used as handshake signals. It also has interrupt handling capacity.

## Conclusion:

Successfully studied the interface of 8255 with 8086.

# EXPERIMENT-9

## **Aim:**

To study the operation of 8259 interface.

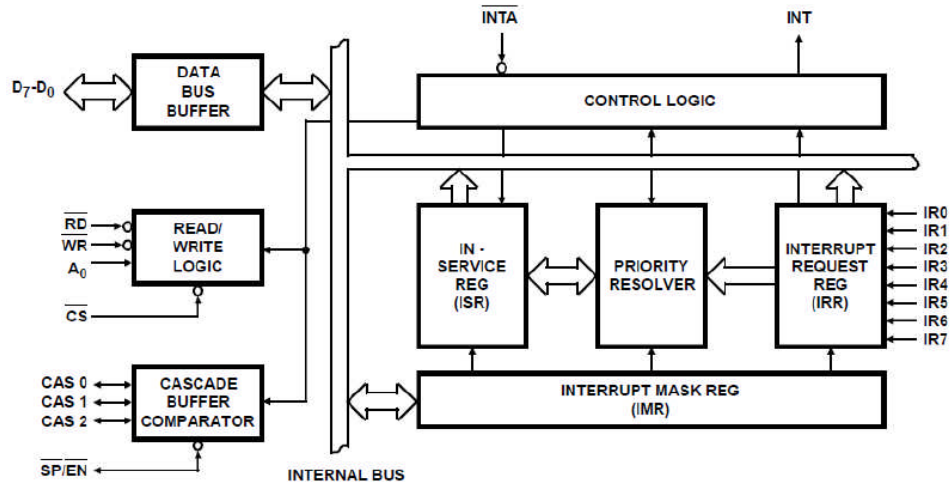
## **Theory:**

PIC 8259 is a Programmable Interrupt Controller that can work with 8085, 8086 etc. It is used to increase the number of interrupts. A single 8259 provides 8 interrupts while a cascaded configuration of 1 master 8259 and 8 slave can provide up to 64 interrupts. 8259 can handle edge as well as level triggered interrupts. It has a flexible priority structure. Interrupts can be masked individually. The Vector address of the interrupts is programmable. In a cascaded configuration, each 8259 has to be individually initialized, master as well as each slave.

## **Interfacing of 8259 with 8086**

- 1) The corresponding bit for an interrupt is set in IRR.
- 2) The Priority Resolver checks the 3 registers: IRR (for highest interrupt request) IMR (for the masking Status) InSR (for the current level serviced) and determines the highest priority interrupt. It sends the INT signal to the  $\mu$ P.
- 3) The  $\mu$ P finishes the current instruction and acknowledges the interrupt by sending the first INTA pulse.
- 4) On receiving the first INTA signal, the corresponding bit in the InSR is set (indicating that now this interrupt is in service) and the bit in the IRR is reset (to indicate that the request is accepted).
- 5) The  $\mu$ P sends the second INTA pulse to 8259.
- 6) In response to the 2nd INTA pulse, 8259 sends the one-byte Vector Number N to  $\mu$ P.
- 7) Now the  $\mu$ P multiplies  $N \times 4$ , to get the values of CS and IP from the IVT.
- 8) In the AEOI Mode the InSR bit is reset at this point, otherwise it remains set until an appropriate EOI command is given at the End of the ISR.
- 9) The  $\mu$ P pushes the contents of Flag Register, CS, IP, into the Stack, Clears IF and TF and transfers the program to the address of the ISR.

## Architecture of 8259



The Block Diagram consists of 8 blocks which are – Data Bus Buffer, Read/Write Logic, Cascade Buffer Comparator, Control Logic, Priority Resolver and 3 registers- ISR, IRR, IMR.

1. Data bus buffer – This Block is used as a mediator between 8259 and 8085/8086 microprocessor by acting as a buffer. It takes the control word from the 8085 (let say) microprocessor and transfer it to the control logic of 8259 microprocessor. Also, after selection of Interrupt by 8259 microprocessor, it transfers the opcode of the selected Interrupt and address of the Interrupt service sub routine to the other connected microprocessor. The data bus buffer consists of 8 bits represented as D0-D7 in the block diagram. Thus, shows that a maximum of 8 bits data can be transferred at a time.

2. Read/Write logic – This block works only when the value of pin CS is low (as this pin is active low). This block is responsible for the flow of data depending upon the inputs of RD and WR. These two pins are active low pins used for read and write operations.

3. Control logic – It is the centre of the microprocessor and controls the functioning of every block. It has pin INTR which is connected with other microprocessor for taking interrupt requests and pin INT for giving the output. If 8259 is enabled, and the other microprocessor Interrupt flag is high then this causes the value of the output INT pin high and in this way 8259 responds to the request made by another microprocessor.

4. Interrupt request register (IRR) – It stores all the interrupt level which are requesting for Interrupt services.

5. Interrupt service register (ISR) – It stores the interrupt level which are currently being executed.
6. Interrupt mask register (IMR) – It stores the interrupt level which have to be masked by storing the masking bits of the interrupt level.
7. Priority resolver – It examines all the three registers and set the priority of interrupts and according to the priority of the interrupts, interrupt with highest priority is set in ISR register. Also, it reset the interrupt level which is already been serviced in IRR.
8. Cascade buffer – To increase the Interrupt handling capability, we can further cascade a greater number of pins by using cascade buffer. So, during increment of interrupt capability, CSA lines are used to control multiple interrupt structure.

## PIN DIAGRAM

$\overline{CS}$	1	28	$V_{CC}$
$\overline{WR}$	2	27	A0
$\overline{RD}$	3	26	$\overline{INTA}$
D7	4	25	IR7
D6	5	24	IR6
D5	6	23	IR5
D4	7	22	IR4
D3	8	21	IR3
D2	9	20	IR2
D1	10	19	IR1
D0	11	18	IR0
CAS0	12	17	INT
CAS1	13	16	$\overline{SP/EN}$
Gnd	14	15	CAS2

## Conclusion:

Successfully studied the operation of 8259 interface..