

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING,  
FACULTY OF ECE,  
Rajshahi University of Engineering & Technology, Bangladesh

# EEE - 3210- Microprocessor, Interfacing & System Design Sessional Sessional Report

---

## Student Lab Report

### Submitted by

**Ashraf Al- Khalique**

Roll: 1801171

Session: 2018-2019

Dept. of Electrical & Electronic Engineering,  
Rajshahi University of Engineering and Technology.

## Table of Contents

Experiment no.	Experiment name	Remarks
1.	Introduction to Assembly Language Programming and Kit Mode of MDA 8086 Trainer Board	
2.	Arithmetic operations (addition, subtraction, division, and multiplication) using memory location	
3.	Logical operation and Loop using Assembly Language Programming	
4.	Experimental Study of Stack Operation and Introduction to Procedures	
5.	Experimental study of String operation using assembly language	
6.	Familiarization with serial monitor mode of MDA 8086 trainer kit	
7.	8255 PPI interfacing: 7 - segment display	
8.	8255 PPI interfacing: Dot matrix	
9.	8255 PPI interfacing: LED	
10.	LED ON/OFF using 8253, 8255 & 8259	

## Experiment No. 01

### **1.1 Experiment Name**

Introduction to Assembly Language Programming and Kit Mode of MDA 8086 Trainer Board

### **1.2 Objectives**

- To understand the structure of Assembly Language programming
- To learn about the microprocessor emulator "Emu 8086" and its operation
- To get acquainted with the "MDA 8086" Trainer Board and its operation
- To learn how to implement program in "MDA 8086" Trainer Board and interconnect it with "Emu 8086"

### **1.3 Theory**

The Assembly language is a low-level computer programming language that consists primarily of symbolic versions of a computer's machine language. Different manufacturers' computers use different machine languages and require different assemblers and assembly languages.

A microprocessor is a type of computer processor that contains the capabilities of a central processing unit on an integrated circuit (IC) or a few integrated circuits that comprises the arithmetic, logic, and control circuitry required to perform the central processing unit functions of a digital computer.

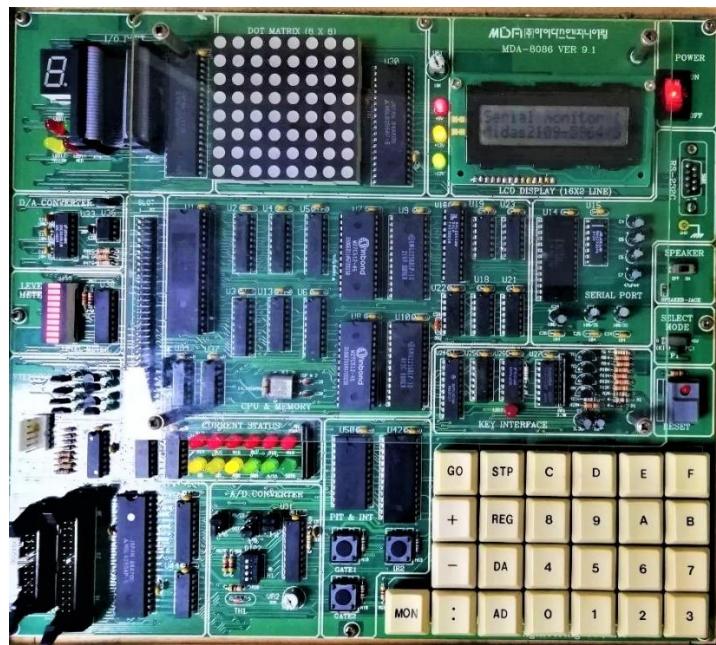


Fig 1.1: MDA 8086 Trainer Microprocessor Kit

The 8086 microprocessor has eight general-purpose registers. They are as follows: AX, BX, CX, DX, SP, BP, SI, and DI. The accumulator is AX. To perform 8-bit instructions, it is separated into two 8-bit registers, AH (upper byte) and AL (lower byte). The MOV operation moves the data item specified by its second operand (register, memory) to the location

specified by its first operand. MOV AX, 10D, for example. The source operand is 10D, while the destination operand is AX. The final result is really accumulated at the destination. In 1978, Intel debuted the 8086 microprocessors. This 16-bit microprocessor marked a significant advancement over the previous generation.

The MDA 8086 microprocessor has the following features:

- A 16-bit microprocessor
- A 20-bit address bus allows it to directly access 220 bits, or 1 MB, of memory.
- The 8086 has fourteen sixteen-bit registers.
- Clock frequency (5 to 10 MHz)

The MDA 8086 consists of a central processing unit (CPU), ROM, SRAM, display, keyboard, speaker, DOT matrix LED, A/D & D/A converter, stepping motor.

It also has function key which operate as follows

[AD]	Set memory address	[GO]	Go to user's final program instruction
[DA]	Update segment & offset, and input data to memory	[REG]	Register display
[STP]	Executing user's program in single steps	[MON]	Immediately break user's program and non-makeable interrupt
[:]	Offset data		

The working operation are as follows

- After imitating the code, it was saved in the software emu8086
- It executed the software step by step by clicking the single step
- The value 10D was first recorded in register AX
- By clicking the single step, the value 17H was recorded in register BX
- Finally, after clicking a single button, the addition of AX and BX, or 10D and 17H, was saved in register AX and displayed on the screen as 21H
- The same phenomenon happens in the MDA 8086 microprocessor kit, and it can be verified step by step.

#### 1.4 Apparatus

- Emu 8086 - Microprocessor Emulator
- MDA 8086 - Trainer Board

#### 1.5 Emulator Code

**MOV AX, 10D; MOVE 10D TO AX**

**MOV BX, 17H; MOVE 1122H TO BX**

**ADD AX, BX; ADD AX AND BX, AND THE ADDITION WILL BE STORED IN AX**

## 1.6 Output Emu 8086

The screenshot shows the Emu 8086 emulator interface. The assembly code window displays the following instructions:

```
01000: B8 184 3 NEWL
01001: 0A 010 NEWL
01002: 00 000 NULL
01003: BB 18? 11
01004: 17 023 ??
01005: 00 000 NULL
01006: 03 003 ?
01007: C3 195 -
01008: 90 144 E
01009: 90 144 E
0100A: 90 144 E
0100B: 90 144 E
0100C: 90 144 E
0100D: 90 144 E
0100E: 90 144 E
0100F: 90 144 E
01010: 90 144 E
01011: 90 144 E
01012: 90 144 E
01013: 90 144 E
01014: 90 144 E
01015: 90 144 E
```

The registers window shows the following values:

	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0000	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 1.1: Emulator output for Assembly Language Program (After Step 0)

The screenshot shows the Emu 8086 emulator interface. The assembly code window displays the following instructions:

```
01000: B8 184 3 NEWL
01001: 0A 010 NEWL
01002: 00 000 NULL
01003: BB 18? 11
01004: 17 023 ??
01005: 00 000 NULL
01006: 03 003 ?
01007: C3 195 -
01008: 90 144 E
01009: 90 144 E
0100A: 90 144 E
0100B: 90 144 E
0100C: 90 144 E
0100D: 90 144 E
0100E: 90 144 E
0100F: 90 144 E
01010: 90 144 E
01011: 90 144 E
01012: 90 144 E
01013: 90 144 E
01014: 90 144 E
01015: 90 144 E
```

The registers window shows the following values:

	H	L
AX	00	0A
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0003	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 1.2: Emulator output for Assembly Language Program (After Step 1)

The screenshot shows the emulator interface with the assembly code window open. The code window displays memory locations from 01000 to 01015. The instruction at address 01006 is highlighted in blue: ADD AX, BX. The registers window on the left shows the following values:

	H	L
AX	00	0A
BX	00	17
CX	00	00
DX	00	00
CS	0100	
IP	0006	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The status bar at the bottom shows: screen, source, reset, aux, vars, debug, stack, flags.

Fig 1.3: Emulator output for Assembly Language Program (After Step 2)

The screenshot shows the emulator interface with the assembly code window open. The code window displays memory locations from 01000 to 01015. The instruction at address 01008 is highlighted in blue: NOP. The registers window on the left shows the following values:

	H	L
AX	00	21
BX	00	17
CX	00	00
DX	00	00
CS	0100	
IP	0008	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The status bar at the bottom shows: screen, source, reset, aux, vars, debug, stack, flags.

Fig 1.4: Emulator output for Assembly Language Program (After Step-3)

## MDA 8086



Fig 1.4: Trainer kit output for Assembly Language Program

## **1.7 Discussion & Conclusion**

In this experiment, the MDA-8086 kit used includes a central processor unit and other components. During this experiment, we became acquainted with the MDA-8086 microprocessor kit and the EMU program.

The instructions are initially stored in memory in a sequential order. The microprocessor reads the instructions from memory, decodes them, and executes them until the STOP command is reached. It then transmits the binary result to the output port. The register stores temporary data between these steps, while the Assembly Language performs computing functions, in this case, we utilized the ADD instruction. In the end, output from emulator and trainer kit was compared and was similar.

## Experiment No. 02

### **2.1 Experiment Name**

Arithmetic operations (addition, subtraction, division, and multiplication) using memory location

### **2.2 Objectives**

- To understand the structure of Assembly Language programming
- To learn about the microprocessor emulator "Emu 8086" and its operation
- To know how to addition, subtraction, multiplication, division bit-size number using memory location at "Emu8086" using Assembly language
- To learn how to implement program in "MDA 8086" Trainer Board and interconnect it with "Emu 8086"

### **2.3 Theory**

Arithmetic operations are a branch of mathematics that deals with the study of numbers and the numerical operations that are used in all other branches of mathematics. Assembly-language programming is used to implement it. A low-level programming language for microprocessors and other programmable devices is assembly language. A microprocessor executes a set of instructions written in machine language, directing the processor what to do. A 16-bit microprocessor trainer board, the MDA-8086 Kit. It is made up of various units. Arithmetic instructions are classified into four types: addition, subtraction, multiplication, and division.

- The ADD instruction modifies the contents of another register (destination) or memory address by adding immediate data or the contents of a memory location specified in the instruction. The outcome is saved in the destination operand.
- The SUB instruction modifies the contents of another register (destination) or memory address by subtracting immediate data or the contents of a memory location specified in the instruction.
- The MUL operation is used to multiply two unsigned numbers, while the IMUL procedure is used to multiply two signed numbers. Based on the number of bits, multiplication is split into three types.
- The DIV operation is used for division of two unsigned numbers or operands.

### **2.4 Apparatus**

- Emu 8086 - Microprocessor Emulator

### **2.5 Emulator Code & Output**

- **Addition**

**DTA DW 5,2,1,9,8**

**MOV SI, OFFSET DTA**

**MOV AX, [SI]**

**MOV BX, [SI+6H]**

**ADD AX, BX**

**DAA**

The screenshot shows the emulator interface with the title bar "emulator: noname.bin\_". The menu bar includes "file", "math", "debug", "view", "external", "virtual devices", "virtual drive", and "help". Below the menu are several control buttons: "Load" (with a folder icon), "reload" (with a circular arrow icon), "step back" (with a left arrow icon), "single step" (with a green triangle icon), "run" (with a green triangle icon), and "step delay ms: 0".

The left panel displays the "registers" window, which lists the following register values:

	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0000	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The middle section shows two memory dump windows labeled "0100:0000". The left window shows the initial state of memory, and the right window shows the state after "Step 0". The right window highlights the first instruction: "ADD AX, 00200h".

At the bottom are buttons for "screen", "source", "reset", "aux", "vars", "debug", "stack", and "flags".

Fig 2.1: Emulator output for Assembly Language Program (After Step 0)

This screenshot shows the emulator after one step has been taken. The interface is identical to Fig 2.1, with the title bar "emulator: noname.bin\_" and the same menu and control buttons.

The "registers" window shows the following updated register values:

	H	L
AX	00	14
BX	00	09
CX	00	00
DX	00	00
CS	0100	
IP	0029	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The memory dump windows show the state after "Step 1". The right window highlights the instruction at IP 0029: "MOV SI, 00000h".

At the bottom are buttons for "screen", "source", "reset", "aux", "vars", "debug", "stack", and "flags".

Fig 2.2: Emulator output for Assembly Language Program (After Step 1)

The screenshot shows a debugger window titled "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu are buttons for Load, reload, step back, single step, run, and step delay ms: 0. The registers pane on the left shows the following values:

	H	L
AX	00	14
BX	00	09
CX	00	00
DX	00	00
CS	0100	
IP	0029	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code window shows memory starting at address 0100A:

```

0100A: BE 190 3
0100B: 00 000 NULL
0100C: 00 000 NULL
0100D: 8B 139 i
0100E: 04 004 ♦
0100F: 8B 139 i
01010: 5C 092 \_
01011: 06 006 ♣
01012: 03 003 ♥
01013: C3 195 J
01014: 27 039 ,
01015: 90 144 É
01016: 90 144 É
01017: 90 144 É
01018: 90 144 É
01019: 90 144 É
0101A: 90 144 É
0101B: 90 144 É
0101C: 90 144 É
0101D: 90 144 É
0101E: 90 144 É
0101F: 90 144 É

```

The instruction at 0100D is highlighted in yellow. The right pane shows the assembly code:

```

MOU AX, [SI]
MOU BX, [SI] + 06h
ADD AX, BX
DAA
NOP

```

Fig 2.3: Emulator output for Assembly Language Program (After Step 2)

This screenshot is identical to Fig 2.3, showing the same assembly code and register values. The instruction at 0100D is still highlighted in yellow.

Fig 2.4: Emulator output for Assembly Language Program (After Step-3)

The screenshot shows the emulator interface with the file "emulator: noname.bin\_" open. The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. The toolbar contains Load, reload, step back, single step, run, and step delay ms: 0. The registers window shows the following values:

	H	L
AX	00	14
BX	00	09
CX	00	00
DX	00	00
CS	0100	
IP	0029	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code window displays the following instructions at address 0100H:

```

0100A: BE 190 3    ; ADD AX, BX
0100B: 00 000 NULL
0100C: 00 000 NULL
0100D: 8B 139 i
0100E: 04 004 ♦
0100F: 8B 139 i
01010: 5C 092 \
01011: 06 006 ♪
01012: 03 003 ♥
01013: C3 195 †
01014: 27 039 ,
01015: 90 144 É
01016: 90 144 É
01017: 90 144 É
01018: 90 144 É
01019: 90 144 É
0101A: 90 144 É
0101B: 90 144 É
0101C: 90 144 É
0101D: 90 144 É
0101E: 90 144 É
0101F: 90 144 É
...

```

The instruction at address 01012 is highlighted in yellow. The right pane shows the assembly mnemonic "ADD AX, BX" followed by several NOP (No Operation) instructions.

Fig 2.5: Emulator output for Assembly Language Program (After Step-4)

The screenshot shows the emulator interface with the file "emulator: noname.bin\_" open. The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. The toolbar contains Load, reload, step back, single step, run, and step delay ms: 0. The registers window shows the same values as in Fig 2.5.

The assembly code window displays the following instructions at address 0100H:

```

0100A: BE 190 3    ; DAA
0100B: 00 000 NULL
0100C: 00 000 NULL
0100D: 8B 139 i
0100E: 04 004 ♦
0100F: 8B 139 i
01010: 5C 092 \
01011: 06 006 ♪
01012: 03 003 ♥
01013: C3 195 †
01014: 27 039 ,
01015: 90 144 É
01016: 90 144 É
01017: 90 144 É
01018: 90 144 É
01019: 90 144 É
0101A: 90 144 É
0101B: 90 144 É
0101C: 90 144 É
0101D: 90 144 É
0101E: 90 144 É
0101F: 90 144 É
...

```

The instruction at address 01014 is highlighted in yellow. The right pane shows the assembly mnemonic "DAA" followed by several NOP (No Operation) instructions.

Fig 2.6: Emulator output for Assembly Language Program (After Step-5)

- Subtraction

DTA DW 5,2,1,9,8

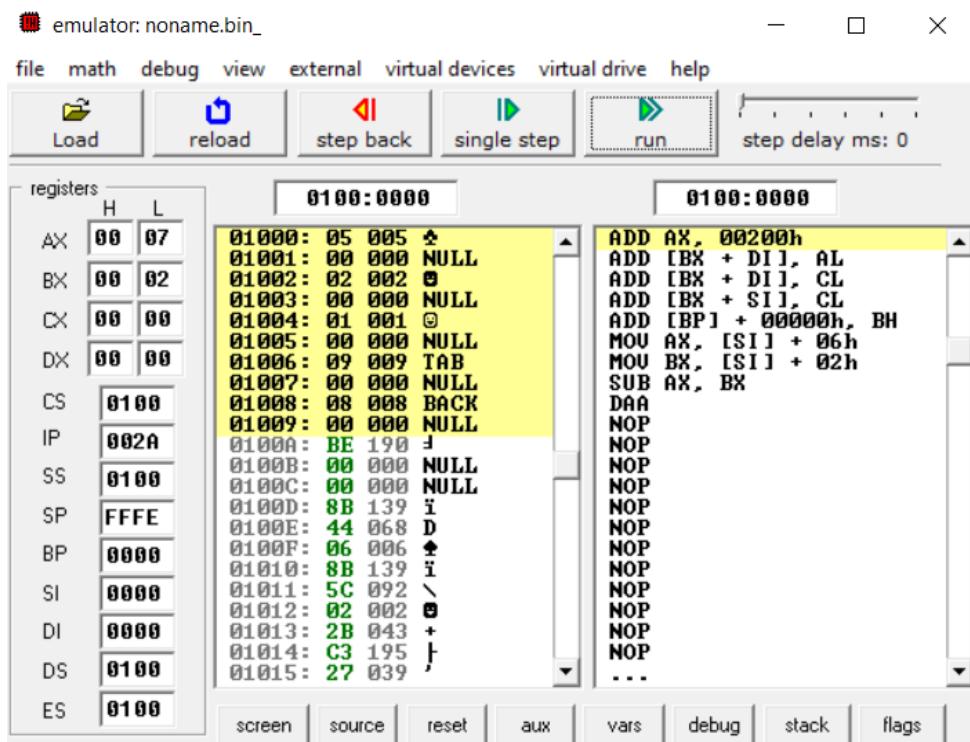
MOV SI, OFFSET DTA

MOV AX, [SI+6H]

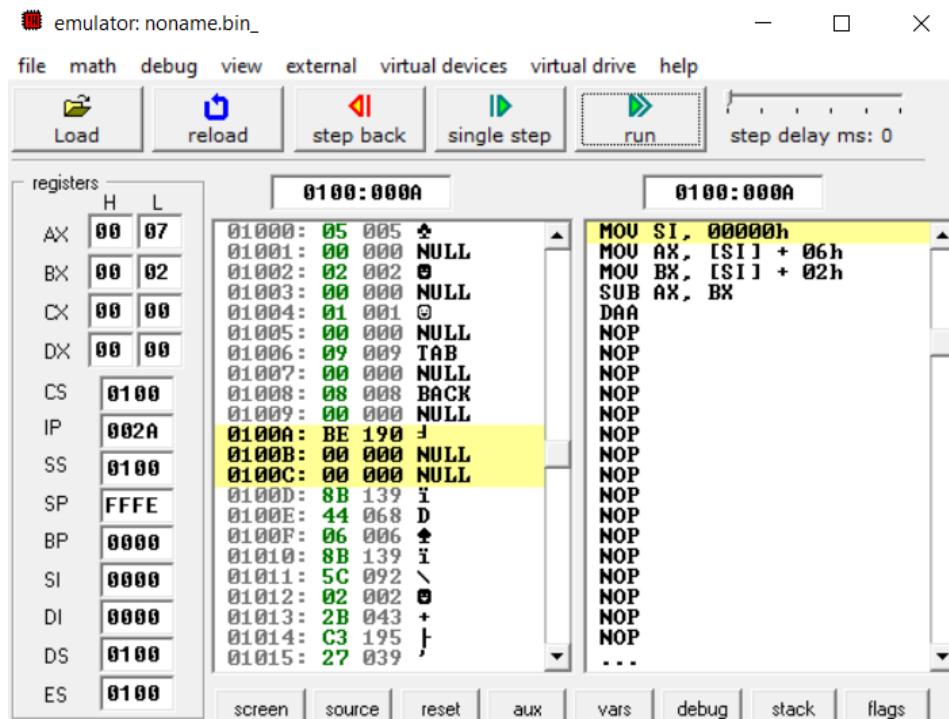
**MOV BX, [SI+2H]**

## **ADD AX, BX**

DAA



*Fig 2.7: Emulator output for Assembly Language Program (After Step 0)*



*Fig 2.8: Emulator output for Assembly Language Program (After Step 1)*

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		H	L
AX	00	07	
BX	00	02	
CX	00	00	
DX	00	00	
CS	0100		
IP	002A		
SS	0100		
SP	FFFE		
BP	0000		
SI	0000		
DI	0000		
DS	0100		
ES	0100		

0100:000D	0100:000D
01000: 05 005 ♣	MOU AX, [SI] + 06h
01001: 00 000 NULL	MOU BX, [SI] + 02h
01002: 02 002 ♦	SUB AX, BX
01003: 00 000 NULL	DAA
01004: 01 001 ♠	NOP
01005: 00 000 NULL	NOP
01006: 09 009 TAB	NOP
01007: 00 000 NULL	NOP
01008: 08 008 BACK	NOP
01009: 00 000 NULL	NOP
0100A: BE 190 ↴	NOP
0100B: 00 000 NULL	NOP
0100C: 00 000 NULL	NOP
0100D: 8B 139 ↪	NOP
0100E: 44 068 D	NOP
0100F: 06 006 ♦	NOP
01010: 8B 139 ↪	NOP
01011: 5C 092 ↘	NOP
01012: 02 002 ♦	NOP
01013: 2B 043 +	NOP
01014: C3 195 ↩	NOP
01015: 27 039 ↴	...

screen source reset aux vars debug stack flags

Fig 2.9: Emulator output for Assembly Language Program (After Step 2)

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		H	L
AX	00	07	
BX	00	02	
CX	00	00	
DX	00	00	
CS	0100		
IP	002A		
SS	0100		
SP	FFFE		
BP	0000		
SI	0000		
DI	0000		
DS	0100		
ES	0100		

0100:0010	0100:0010
01000: 05 005 ♣	MOU BX, [SI] + 02h
01001: 00 000 NULL	SUB AX, BX
01002: 02 002 ♦	DAA
01003: 00 000 NULL	NOP
01004: 01 001 ♠	NOP
01005: 00 000 NULL	NOP
01006: 09 009 TAB	NOP
01007: 00 000 NULL	NOP
01008: 08 008 BACK	NOP
01009: 00 000 NULL	NOP
0100A: BE 190 ↴	NOP
0100B: 00 000 NULL	NOP
0100C: 00 000 NULL	NOP
0100D: 8B 139 ↪	NOP
0100E: 44 068 D	NOP
0100F: 06 006 ♦	NOP
01010: 8B 139 ↪	NOP
01011: 5C 092 ↘	NOP
01012: 02 002 ♦	NOP
01013: 2B 043 +	NOP
01014: C3 195 ↩	NOP
01015: 27 039 ↴	...

screen source reset aux vars debug stack flags

Fig 2.10: Emulator output for Assembly Language Program (After Step 3)

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		0100:0013	0100:0013
H	L		
AX	00 07	01000: 05 005 ♣	SUB AX, BX
BX	00 02	01001: 00 000 NULL	DAA
CX	00 00	01002: 02 002 ♣	NOP
DX	00 00	01003: 00 000 NULL	NOP
CS	0100	01004: 01 001 ♣	NOP
IP	002A	01005: 00 000 NULL	NOP
SS	0100	01006: 09 009 TAB	NOP
SP	FFFE	01007: 00 000 NULL	NOP
BP	0000	01008: 08 008 BACK	NOP
SI	0000	01009: 00 000 NULL	NOP
DI	0000	0100A: BE 190 ↴	NOP
DS	0100	0100B: 00 000 NULL	NOP
ES	0100	0100C: 00 000 NULL	NOP

0100D: 8B 139 i  
0100E: 44 068 D  
0100F: 06 006 ♣  
01010: 8B 139 i  
01011: 5C 092 ↵  
01012: 02 002 ♣  
**01013: 2B 043 +**  
**01014: C3 195 ↴**  
01015: 27 039 ,

screen source reset aux vars debug stack flags

Fig 2.11: Emulator output for Assembly Language Program (After Step 4)

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		0100:0015	0100:0015
H	L		
AX	00 07	01000: 05 005 ♣	DAA
BX	00 02	01001: 00 000 NULL	NOP
CX	00 00	01002: 02 002 ♣	NOP
DX	00 00	01003: 00 000 NULL	NOP
CS	0100	01004: 01 001 ♣	NOP
IP	002A	01005: 00 000 NULL	NOP
SS	0100	01006: 09 009 TAB	NOP
SP	FFFE	01007: 00 000 NULL	NOP
BP	0000	01008: 08 008 BACK	NOP
SI	0000	01009: 00 000 NULL	NOP
DI	0000	0100A: BE 190 ↴	NOP
DS	0100	0100B: 00 000 NULL	NOP
ES	0100	0100C: 00 000 NULL	NOP

0100D: 8B 139 i  
0100E: 44 068 D  
0100F: 06 006 ♣  
01010: 8B 139 i  
01011: 5C 092 ↵  
01012: 02 002 ♣  
**01013: 2B 043 +**  
**01014: C3 195 ↴**  
**01015: 27 039 ,**

screen source reset aux vars debug stack flags

Fig 2.12: Emulator output for Assembly Language Program (After Step 5)

- Multiplication

**MOV AL, 4**  
**MOV SI, 0120H**  
**MOV [SI], 3**  
**MUL [SI]**

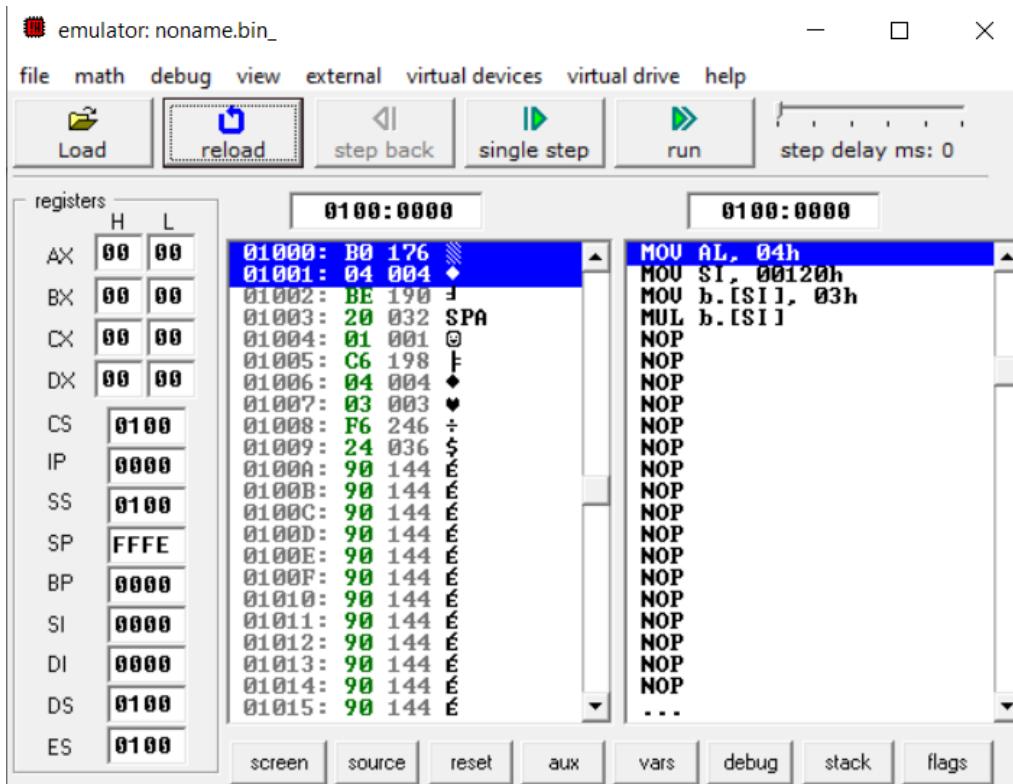
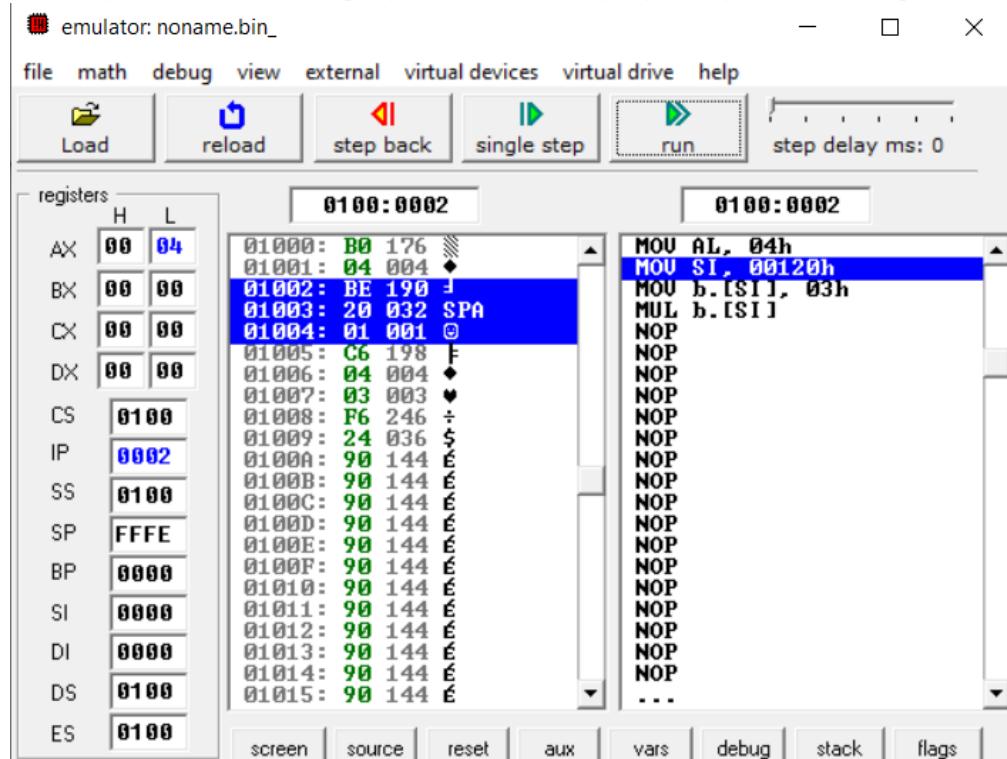


Fig 2.13: Emulator output for Assembly Language Program (After Step 0)



*Fig 2.14: Emulator output for Assembly Language Program (After Step 1)*

The screenshot shows the emulator interface with the title bar "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu are several control buttons: Load, reload, step back, single step, run, and step delay ms: 0. On the left, a "registers" window displays memory addresses from 01000 to 01015. The AX register has values 00 04. The IP register has value 0005. The CS register has value 0100. The SP register has value FFFE. The SI register has value 0120. The DS register has value 0100. The ES register has value 0100. The main windows show memory locations 01000 to 01015. Address 01000 contains B0 176. Address 01001 contains 04 004. Address 01002 contains BE 190. Address 01003 contains 20 032 SPA. Address 01004 contains 01 001. Address 01005 contains C6 198. Address 01006 contains 04 004. Address 01007 contains 03 003. Address 01008 contains F6 246. Address 01009 contains 24 036. Address 0100A contains 90 144. Address 0100B contains 90 144. Address 0100C contains 90 144. Address 0100D contains 90 144. Address 0100E contains 90 144. Address 0100F contains 90 144. Address 01010 contains 90 144. Address 01011 contains 90 144. Address 01012 contains 90 144. Address 01013 contains 90 144. Address 01014 contains 90 144. Address 01015 contains 90 144. The instruction at address 01005 is highlighted in blue.

Fig 2.15: Emulator output for Assembly Language Program (After Step 2)

The screenshot shows the emulator interface with the title bar "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu are several control buttons: Load, reload, step back, single step, run, and step delay ms: 0. On the left, a "registers" window displays memory addresses from 01000 to 01015. The AX register has values 00 04. The IP register has value 0008. The CS register has value 0100. The SP register has value FFFE. The SI register has value 0120. The DS register has value 0100. The ES register has value 0100. The main windows show memory locations 01000 to 01015. Address 01000 contains B0 176. Address 01001 contains 04 004. Address 01002 contains BE 190. Address 01003 contains 20 032 SPA. Address 01004 contains 01 001. Address 01005 contains C6 198. Address 01006 contains 04 004. Address 01007 contains 03 003. Address 01008 contains F6 246. Address 01009 contains 24 036. Address 0100A contains 90 144. Address 0100B contains 90 144. Address 0100C contains 90 144. Address 0100D contains 90 144. Address 0100E contains 90 144. Address 0100F contains 90 144. Address 01010 contains 90 144. Address 01011 contains 90 144. Address 01012 contains 90 144. Address 01013 contains 90 144. Address 01014 contains 90 144. Address 01015 contains 90 144. The instruction at address 01009 is highlighted in blue.

Fig 2.16: Emulator output for Assembly Language Program (After Step 3)

The screenshot shows a debugger window titled "emulator: noname.bin\_". The menu bar includes "file", "math", "debug", "view", "external", "virtual devices", "virtual drive", and "help". Below the menu are several control buttons: "Load", "reload", "step back", "single step", "run", and a slider for "step delay ms: 0".

The left side displays the "registers" pane, which lists the following register values:

	H	L
AX	00	0C
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	000A	
SS	0100	
SP	FFFE	
BP	0000	
SI	0120	
DI	0000	
DS	0100	
ES	0100	

The assembly code window shows two columns of memory dump. The left column is labeled "0100:000A" and the right column is labeled "0100:000B". The code consists of the following instructions:

```

01000: B0 176 ≡
01001: 04 004 ♦
01002: BE 190 ↴
01003: 20 032 SPA
01004: 01 001 ⊖
01005: C6 198 ↫
01006: 04 004 ♦
01007: 03 003 ♥
01008: F6 246 ÷
01009: 24 036 $̄
0100A: 90 144 Ē
0100B: 90 144 Ē
0100C: 90 144 Ē
0100D: 90 144 Ē
0100E: 90 144 Ē
0100F: 90 144 Ē
01010: 90 144 Ē
01011: 90 144 Ē
01012: 90 144 Ē
01013: 90 144 Ē
01014: 90 144 Ē
01015: 90 144 Ē
...

```

Below the code windows are tabs for "screen", "source", "reset", "aux", "vars", "debug", "stack", and "flags".

Fig 2.17: Emulator output for Assembly Language Program (After Step 4)

- Division

**MOV AL, 4**  
**MOV SI, 0120H**  
**MOV [SI], 3**  
**DIV [SI]**

This screenshot is identical to Fig 2.17, showing the same debugger interface and assembly code. The main difference is the step number, which is "0" instead of "4" in the title.

The assembly code window shows the initial state of the program at address 0100:

```

01000: B0 176 ≡
01001: 04 004 ♦
01002: BE 190 ↴
01003: 20 032 SPA
01004: 01 001 ⊖
01005: C6 198 ↫
01006: 04 004 ♦
01007: 03 003 ♥
01008: F6 246 ÷
01009: 34 052 4̄
0100A: 90 144 Ē
0100B: 90 144 Ē
0100C: 90 144 Ē
0100D: 90 144 Ē
0100E: 90 144 Ē
0100F: 90 144 Ē
01010: 90 144 Ē
01011: 90 144 Ē
01012: 90 144 Ē
01013: 90 144 Ē
01014: 90 144 Ē
01015: 90 144 Ē
...

```

Fig 2.18: Emulator output for Assembly Language Program (After Step 0)

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	H	L
AX	00	04
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0002	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

```
0100:0002
01000: B0 176
01001: 04 004 ◆
01002: BE 190 ↴
01003: 20 032 SPA
01004: 01 001 ⊗
01005: C6 198 F
01006: 04 004 ◆
01007: 03 003 ♥
01008: F6 246 ÷
01009: 34 052 4
0100A: 90 144 É
0100B: 90 144 É
0100C: 90 144 É
0100D: 90 144 É
0100E: 90 144 É
0100F: 90 144 É
01010: 90 144 É
01011: 90 144 É
01012: 90 144 É
01013: 90 144 É
01014: 90 144 É
01015: 90 144 É
...
0100:0002
MOU AL, 04h
MOU SI, 00120h
MOU b.[SI], 03h
DIV b.[SI]
NOP
...

```

screen source reset aux vars debug stack flags

Fig 2.19: Emulator output for Assembly Language Program (After Step 1)

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	H	L
AX	00	04
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0005	
SS	0100	
SP	FFFE	
BP	0000	
SI	0120	
DI	0000	
DS	0100	
ES	0100	

```
0100:0005
01000: B0 176
01001: 04 004 ◆
01002: BE 190 ↴
01003: 20 032 SPA
01004: 01 001 ⊗
01005: C6 198 F
01006: 04 004 ◆
01007: 03 003 ♥
01008: F6 246 ÷
01009: 34 052 4
0100A: 90 144 É
0100B: 90 144 É
0100C: 90 144 É
0100D: 90 144 É
0100E: 90 144 É
0100F: 90 144 É
01010: 90 144 É
01011: 90 144 É
01012: 90 144 É
01013: 90 144 É
01014: 90 144 É
01015: 90 144 É
...
0100:0005
MOU AL, 04h
MOU SI, 00120h
MOU b.[SI], 03h
DIV b.[SI]
NOP
...

```

screen source reset aux vars debug stack flags

Fig 2.20: Emulator output for Assembly Language Program (After Step 2)

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		0100:0008	0100:0008
H	L		
AX	00 04	01000: B0 176	MOU AL, 04h
BX	00 00	01001: 04 004	MOU SI, 00120h
CX	00 00	01002: BE 190	MOU b.[SI], 03h
DX	00 00	01003: 20 032 SPA	DIU b.[SI]
CS	0100	01004: 01 001	NOP
IP	0008	01005: C6 198	NOP
SS	0100	01006: 04 004	NOP
SP	FFFE	01007: 03 003	NOP
BP	0000	01008: F6 246	NOP
SI	0120	01009: 34 052 4	NOP
DI	0000	0100A: 90 144	NOP
DS	0100	0100B: 90 144	NOP
ES	0100	0100C: 90 144	NOP
		0100D: 90 144	NOP
		0100E: 90 144	NOP
		0100F: 90 144	NOP
		01010: 90 144	NOP
		01011: 90 144	NOP
		01012: 90 144	NOP
		01013: 90 144	NOP
		01014: 90 144	NOP
		01015: 90 144	NOP
		...	

screen source reset aux vars debug stack flags

Fig 2.21: Emulator output for Assembly Language Program (After Step 3)

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		0100:000A	0100:000A
H	L		
AX	01 01	01000: B0 176	MOU AL, 04h
BX	00 00	01001: 04 004	MOU SI, 00120h
CX	00 00	01002: BE 190	MOU b.[SI], 03h
DX	00 00	01003: 20 032 SPA	DIU b.[SI]
CS	0100	01004: 01 001	NOP
IP	000A	01005: C6 198	NOP
SS	0100	01006: 04 004	NOP
SP	FFFE	01007: 03 003	NOP
BP	0000	01008: F6 246	NOP
SI	0120	01009: 34 052 4	NOP
DI	0000	0100A: 90 144	NOP
DS	0100	0100B: 90 144	NOP
ES	0100	0100C: 90 144	NOP
		0100D: 90 144	NOP
		0100E: 90 144	NOP
		0100F: 90 144	NOP
		01010: 90 144	NOP
		01011: 90 144	NOP
		01012: 90 144	NOP
		01013: 90 144	NOP
		01014: 90 144	NOP
		01015: 90 144	NOP
		...	

screen source reset aux vars debug stack flags

Fig 2.22: Emulator output for Assembly Language Program (After Step 4)

## **2.6 Discussion & Conclusion**

In this experiment, the MDA-8086 kit used includes a central processor unit and other components. During this experiment, we became acquainted with the MDA-8086 microprocessor kit and the EMU program.

The assembly language is written on EMU 8086 to conduct arithmetic operations such as addition, subtraction, division, and multiplication, in various locations in our memory. This is due to the fact that we included the carry from the prior data transaction. MOV instructions were used to maneuver information from a memory location to a register. Then, by running the imitator step by step, the specified output was obtained. Thus, the offset was observed because of the RAM access. The experiment's aims are met, and the results are satisfactory. Finally, the experiment can be described as productive and successful.

## Experiment No. 03

### **3.1 Experiment Name**

Logical operation and Loop using Assembly Language Programming

### **3.2 Objectives**

- To understand the structure of Assembly Language programming
- To learn about the microprocessor emulator "Emu 8086" and its operation
- To get acquainted with the logical operation using ALU
- To learn how to implement program in "MDA 8086" Trainer Board and interconnect it with "Emu 8086"

### **3.3 Theory**

In assembly language programming, various logical operation is executed through exclusive instruction sets. In this experiment we will be using following operations and loops for study purpose.

**3.3.1 Shift Operation:** The shift operation simply changes the provided data bit in the register to the left or right. It comes in two varieties.

- **Shift Left (SHL):** The SHL instruction stands for "shift left". This instruction moves the given register bits one at a time to the left for a positive integer.
- **Shift Right (SHR):** SHR is an abbreviation for "shift right." This instruction moves the given register bits one at a time to the right.
- **Shift Arithmetic Left (SAL):** The SAL is frequently used when numeric multiplication is required. In other words, shift binary one bit to the left while leaving the MSB untouched.
- **Shift Arithmetic Right (SAR):** SAR is quite similar to SAL with a slight change. It shifts binary one bit to the right while leaving the MSB untouched.

**3.3.2 Rotate Operation:** This instruction rotates the bits. There are four rotation instructions:

- **Left Rotation (ROL):** The ROL instruction indicates to rotate the bits one by one to the left. The only distinction between ROL and SHL is, in SHL, the value of D0 is replaced by another value, whereas in ROL, the value of D0 is replaced by the bit from position D7.
- **Right Rotation (ROR):** The ROR instruction means to rotate the bits to the right one at a time. The sole difference between ROR and SHR is that in SHR, the bit of position D7 is replaced with another value.
- **Left Rotation with Carry (RCL):** The bits in the operand are rotated through the carry flag to the left using RCL. The most significant bit, together with the carry flag, is relocated to the least significant bit by the RCL instruction.
- **Right Rotation with Carry (RCR):** The bits in the operand are rotated through the carry flag using RCR. The least significant bit, together with the carry flag, is relocated to the most significant bit by the RCR instruction.

### 3.3.3 AND

Only if both of its inputs are true does the AND logic function return true. If one of the inputs is false, the output will be false as well.

### 3.3.4 OR

In OR operation, if one of the inputs is true, function return true. If one of the inputs is false, the output will be false as well.

### 3.3.5 X- OR

XOR compares two input bits and generates one output bit. The logic is simple. If the bits are the same, the result is true. If the bits are different, the result is false.

We will also implement looping operation in this experiment. Through this instruction, we will execute operation for the number of times defined before the loop

## 3.4 Apparatus

- Emu 8086 - Microprocessor Emulator

### 3.5 Emulator Code & output

- SHL

**MOV CL, 2; COUNTER**

**MOV AX, 0B3H; MOVE 0B3H TO AX**

**SHL AX, CL; SHIFT LEFT**

The screenshot shows the Emulator 8086 interface. The assembly code window displays the following sequence:

```
01000: B1 17? ; MOV CL, 2
01001: 02 002 ; COUNTER
01002: B8 184 ; MOV AX, 0B3H
01003: B3 179 ; MOVE 0B3H TO AX
01004: 00 000 NULL
01005: D3 211 ; SHL AX, CL
01006: E0 224 ; SHIFT LEFT
01007: 90 144 ; NOP
01008: 90 144 ; NOP
01009: 90 144 ; NOP
0100A: 90 144 ; NOP
0100B: 90 144 ; NOP
0100C: 90 144 ; NOP
0100D: 90 144 ; NOP
0100E: 90 144 ; NOP
0100F: 90 144 ; NOP
01010: 90 144 ; NOP
01011: 90 144 ; NOP
01012: 90 144 ; NOP
01013: 90 144 ; NOP
01014: 90 144 ; NOP
01015: 90 144 ; NOP
```

The register window on the left shows the following values:

	H	L
AX	00	00
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0002	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 3.1: Emulator output for SHL (After Step 1)

The screenshot shows the emulator interface with the title "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu are buttons for Load, reload, step back, single step, run, and step delay ms: 0. The registers window on the left shows the following values:

	H	L
AX	00	B3
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0005	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code windows show memory locations from 01000 to 01015. The first window is titled "0100:0005" and the second is "0100:0005". The instruction at IP (0005) is highlighted in blue: "SHL AX, CL".

Fig 3.2: Emulator output for SHL (After Step 2)

This screenshot shows the state of the emulator after step 3. The registers window now shows:

	H	L
AX	02	CC
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0007	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code windows show memory locations from 01000 to 01015. The first window is titled "0100:0007" and the second is "0100:0007". The instruction at IP (0007) is highlighted in blue: "90 144 E".

Fig 3.3: Emulator output for SHL (After Step 3)

- SHR
- MOV CL, 2; COUNTER**  
**MOV AX, 0B3H; MOVE 0B3H TO AX**  
**SHR AX, CL; SHIFT LEFT**

The screenshot shows the emulator interface with the assembly code window displaying the first 16 bytes of memory. The instruction at address 01000 is highlighted in blue: MOU CL, 02h. The registers window shows the AX register has a value of 00 00.

	H	L
AX	00	00
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0002	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 3.4: Emulator output for SHR (After Step 1)

The screenshot shows the emulator interface with the assembly code window displaying the first 16 bytes of memory. The instruction at address 01000 is highlighted in blue: MOU CL, 02h. The registers window shows the AX register has a value of 00 B3.

	H	L
AX	00	B3
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0005	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

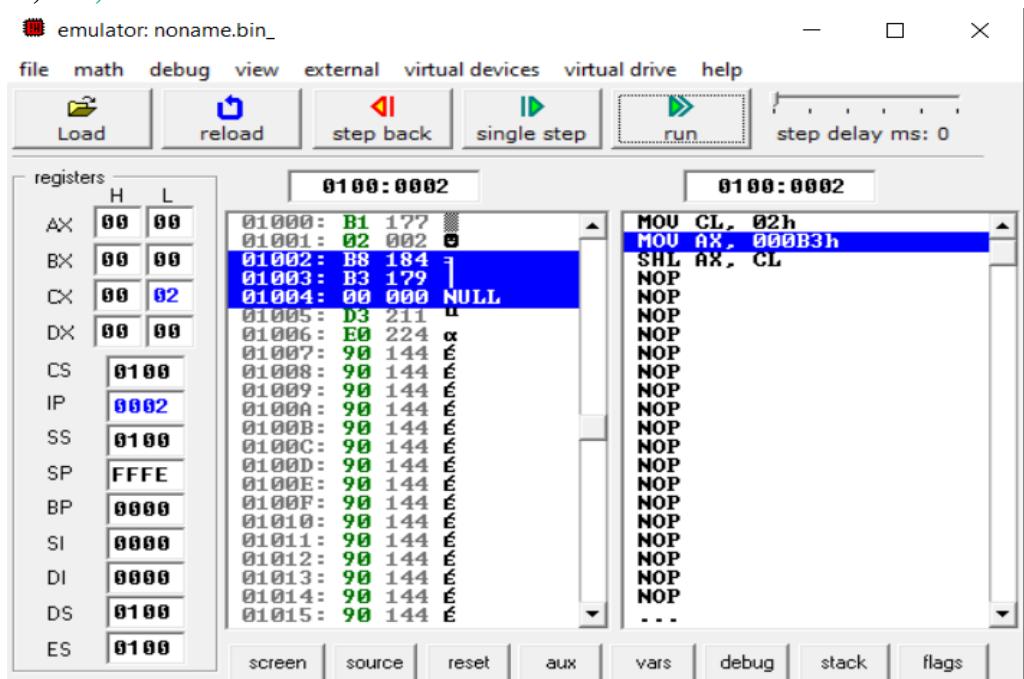
Fig 3.5: Emulator output for SHR (After Step 2)

The screenshot shows the emulator interface with the assembly code window displaying the first 16 bytes of memory. The instruction at address 01000 is highlighted in blue: MOU CL, 02h. The registers window shows the AX register has a value of 00 2C.

	H	L
AX	00	2C
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0007	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 3.6: Emulator output for SHR (After Step 3)

- SAL
- MOV CL, 2; COUNTER**  
**MOV AX, 0B3H; MOVE 0B3H TO AX**  
**SAL AX, CL; SHIFT ARITHAMTIC LEFT**



The screenshot shows the emulator interface with the following details:

- Registers:** AX = 00 00, BX = 00 00, CX = 00 02, DX = 00 00, CS = 0100, IP = 0002, SS = 0100, SP = FFFE, BP = 0000, SI = 0000, DI = 0000, DS = 0100, ES = 0100.
- Memory Dump (0100:0002):**

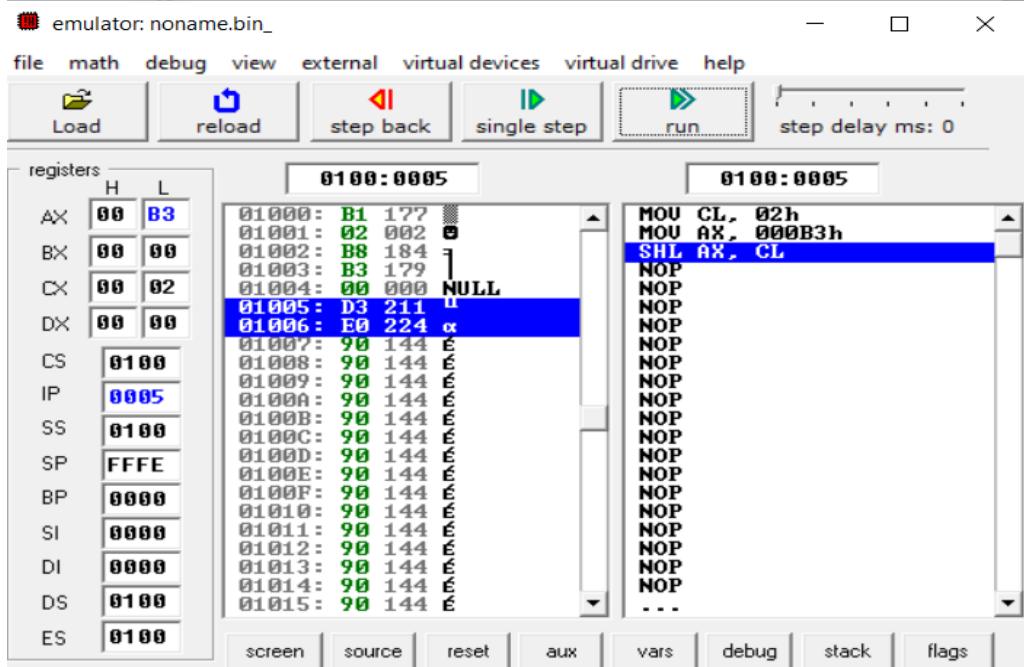
Address	Value	Content
01000	B1	177
01001	02	002
01002	B8	184
01003	B3	179
01004	00	000 NULL
01005	D3	211
01006	E0	224
01007	90	144
01008	90	144
01009	90	144
0100A	90	144
0100B	90	144
0100C	90	144
0100D	90	144
0100E	90	144
0100F	90	144
01010	90	144
01011	90	144
01012	90	144
01013	90	144
01014	90	144
01015	90	144
- Code View (0100:0002):**

```

MOU CL, 02h
MOU AX, 000B3h
SHL AX, CL
NOP
...

```
- Buttons:** Load, reload, step back, single step, run, step delay ms: 0.

Fig 3.7: Emulator output for SAL (After Step 1)



The screenshot shows the emulator interface with the following details:

- Registers:** AX = 00 B3, BX = 00 00, CX = 00 02, DX = 00 00, CS = 0100, IP = 0005, SS = 0100, SP = FFFE, BP = 0000, SI = 0000, DI = 0000, DS = 0100, ES = 0100.
- Memory Dump (0100:0005):**

Address	Value	Content
01000	B1	177
01001	02	002
01002	B8	184
01003	B3	179
01004	00	000 NULL
01005	D3	211
01006	E0	224
01007	90	144
01008	90	144
01009	90	144
0100A	90	144
0100B	90	144
0100C	90	144
0100D	90	144
0100E	90	144
0100F	90	144
01010	90	144
01011	90	144
01012	90	144
01013	90	144
01014	90	144
01015	90	144
- Code View (0100:0005):**

```

MOU CL, 02h
MOU AX, 000B3h
SHL AX, CL
NOP
...

```
- Buttons:** Load, reload, step back, single step, run, step delay ms: 0.

Fig 3.8: Emulator output for SAL (After Step 2)

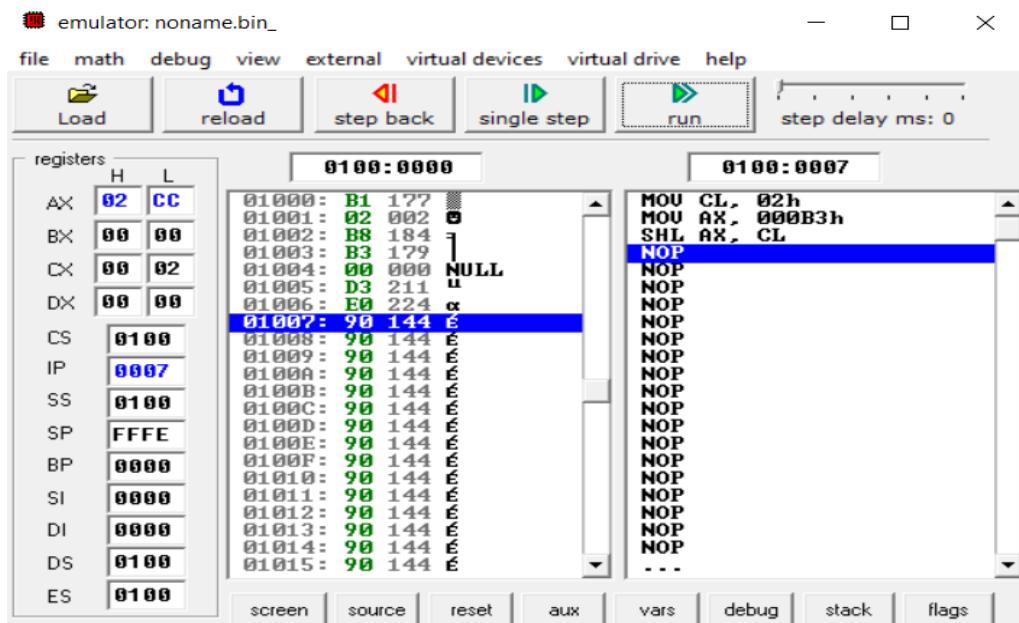


Fig 3.9: Emulator output for SAL (After Step 3)

- SAR

MOV CL, 2; COUNTER  
MOV AX, 0B3H; MOVE 0B3H TO AX  
SAR AX, CL; SHIFT ARITHAMTIC LEFT

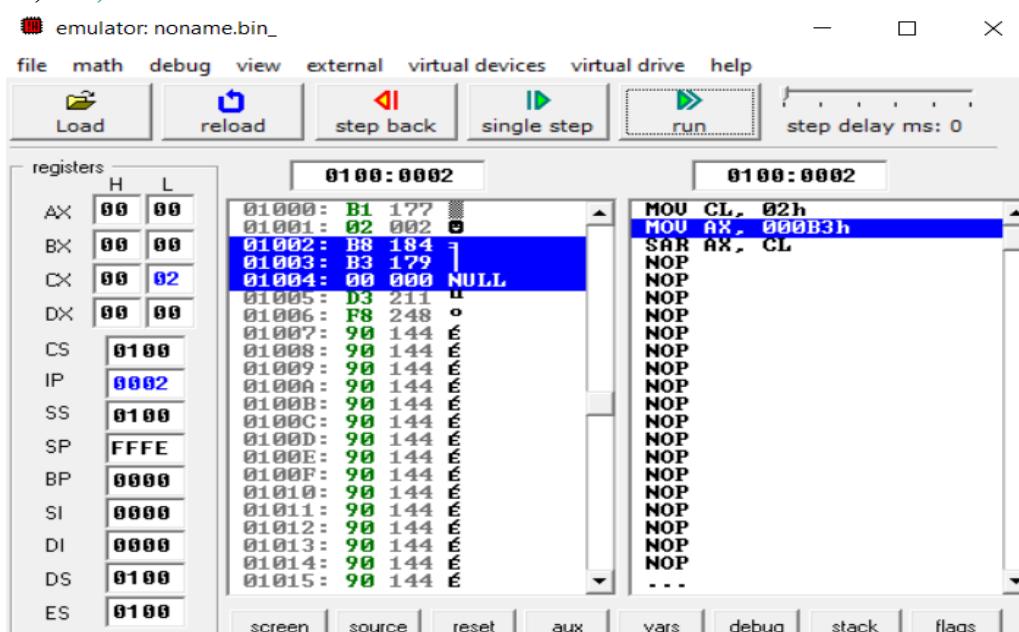


Fig 3.10: Emulator output for SAR (After Step 1)

The screenshot shows the emulator interface with the title "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu are buttons for Load, reload, step back, single step, run, and step delay ms: 0. The registers window shows the following values:

	H	L
AX	00	B3
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0005	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code window shows the following instructions at address 01000:

```

01000: B1 177
01001: 02 002
01002: B8 184
01003: B3 179
01004: 00 000 NULL
01005: D3 211
01006: F8 248
01007: 90 144
01008: 90 144
01009: 90 144
0100A: 90 144
0100B: 90 144
0100C: 90 144
0100D: 90 144
0100E: 90 144
0100F: 90 144
01010: 90 144
01011: 90 144
01012: 90 144
01013: 90 144
01014: 90 144
01015: 90 144

```

The right pane shows the stack memory starting at address 01000:

```

01000: MOU CL, 02h
01001: MOU AX, 000B3h
01002: SAR AX, CL
01003: NOP
01004: NOP
01005: NOP
01006: NOP
01007: NOP
01008: NOP
01009: NOP
0100A: NOP
0100B: NOP
0100C: NOP
0100D: NOP
0100E: NOP
0100F: NOP
01010: NOP
01011: NOP
01012: NOP
01013: NOP
01014: NOP
01015: NOP

```

Fig 3.11: Emulator output for SAR (After Step 2)

The screenshot shows the emulator interface with the title "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu are buttons for Load, reload, step back, single step, run, and step delay ms: 0. The registers window shows the following values:

	H	L
AX	00	2C
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0007	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code window shows the following instructions at address 01000:

```

01000: B1 177
01001: 02 002
01002: B8 184
01003: B3 179
01004: 00 000 NULL
01005: D3 211
01006: F8 248
01007: 90 144
01008: 90 144
01009: 90 144
0100A: 90 144
0100B: 90 144
0100C: 90 144
0100D: 90 144
0100E: 90 144
0100F: 90 144
01010: 90 144
01011: 90 144
01012: 90 144
01013: 90 144
01014: 90 144
01015: 90 144

```

The right pane shows the stack memory starting at address 01000:

```

01000: MOU CL, 02h
01001: MOU AX, 000B3h
01002: SAR AX, CL
01003: NOP
01004: NOP
01005: NOP
01006: NOP
01007: NOP
01008: NOP
01009: NOP
0100A: NOP
0100B: NOP
0100C: NOP
0100D: NOP
0100E: NOP
0100F: NOP
01010: NOP
01011: NOP
01012: NOP
01013: NOP
01014: NOP
01015: NOP

```

Fig 3.12: Emulator output for SAR (After Step 3)

- ROL

**MOV CL, 2; COUNTER**

**MOV AX, 0B3H; MOVE 0B3H TO AX**

**ROL AX, CL; ROTATE LEFT**

The screenshot shows the emulator interface with the assembly window displaying the following code:

```

01000: B1 177
01001: 02 002
01002: B8 184
01003: B3 179
01004: 00 000 NULL
01005: D3 211
01006: C0 192 L
01007: 90 144 E
01008: 90 144 E
01009: 90 144 E
0100A: 90 144 E
0100B: 90 144 E
0100C: 90 144 E
0100D: 90 144 E
0100E: 90 144 E
0100F: 90 144 E
01010: 90 144 E
01011: 90 144 E
01012: 90 144 E
01013: 90 144 E
01014: 90 144 E
01015: 90 144 E

```

The registers window shows the following values:

	H	L
AX	00	00
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0002	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 3.13: Emulator output for ROL (After Step 1)

The screenshot shows the emulator interface with the assembly window displaying the following code:

```

01000: B1 177
01001: 02 002
01002: B8 184
01003: B3 179
01004: 00 000 NULL
01005: D3 211
01006: C0 192 L
01007: 90 144 E
01008: 90 144 E
01009: 90 144 E
0100A: 90 144 E
0100B: 90 144 E
0100C: 90 144 E
0100D: 90 144 E
0100E: 90 144 E
0100F: 90 144 E
01010: 90 144 E
01011: 90 144 E
01012: 90 144 E
01013: 90 144 E
01014: 90 144 E
01015: 90 144 E

```

The registers window shows the following values:

	H	L
AX	00	B3
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0005	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 3.14: Emulator output for ROL (After Step 2)

The screenshot shows the emulator interface with the assembly window displaying the following code:

```

01000: B1 177
01001: 02 002
01002: B8 184
01003: B3 179
01004: 00 000 NULL
01005: D3 211
01006: C0 192 L
01007: 90 144 E
01008: 90 144 E
01009: 90 144 E
0100A: 90 144 E
0100B: 90 144 E
0100C: 90 144 E
0100D: 90 144 E
0100E: 90 144 E
0100F: 90 144 E
01010: 90 144 E
01011: 90 144 E
01012: 90 144 E
01013: 90 144 E
01014: 90 144 E
01015: 90 144 E

```

The registers window shows the following values:

	H	L
AX	02	CC
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0007	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

Fig 3.15: Emulator output for ROL (After Step 3)

- ROR

**MOV CL, 2; COUNTER**

**MOV AX, 0B3H; MOVE 0B3H TO AX**

**ROR AX, CL; ROTATE RIGHT**

The screenshot shows a debugger interface with two assembly panes and a register pane.

**Registers:**

	H	L
AX	00	00
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0002	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

**Assembly (0100:0002 to 0100:0002):**

```

01000: B1 177
01001: 02 002
01002: B8 184
01003: B3 179
01004: 00 000 NULL
01005: D3 211
01006: C8 200
01007: 90 144
01008: 90 144
01009: 90 144
0100A: 90 144
0100B: 90 144
0100C: 90 144
0100D: 90 144
0100E: 90 144
0100F: 90 144
01010: 90 144
01011: 90 144
01012: 90 144
01013: 90 144
01014: 90 144
01015: 90 144

```

**Assembly (0100:0005 to 0100:0005):**

```

MOV CL, 02h
MOV AX, 000B3h
ROR AX, CL
NOP

```

Fig 3.16: Emulator output for ROR (After Step 1)

The screenshot shows a debugger interface with two assembly panes and a register pane.

**Registers:**

	H	L
AX	00	B3
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0005	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

**Assembly (0100:0005 to 0100:0005):**

```

MOV CL, 02h
MOV AX, 000B3h
ROR AX, CL
NOP

```

Fig 3.17: Emulator output for ROR (After Step 2)

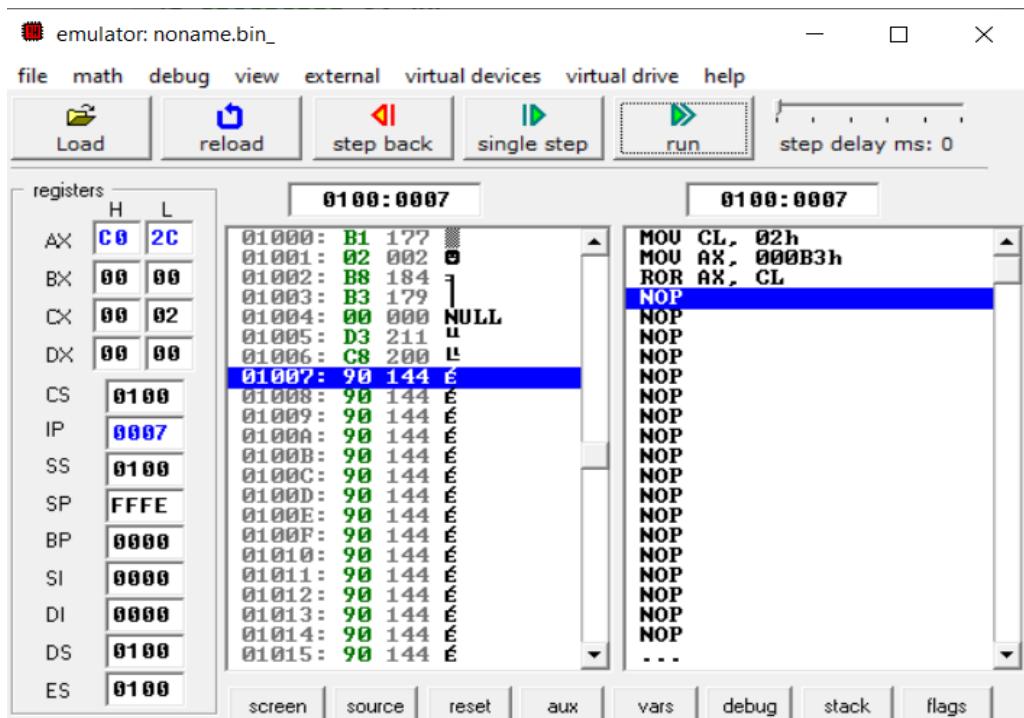


Fig 3.18: Emulator output for ROR (After Step 3)

- RCL

**MOV CL, 2; COUNTER**

**MOV AX, 0B3H; MOVE 0B3H TO AX**

**RCL AX, CL; ROTATE WITH CARRY TO LEFT**

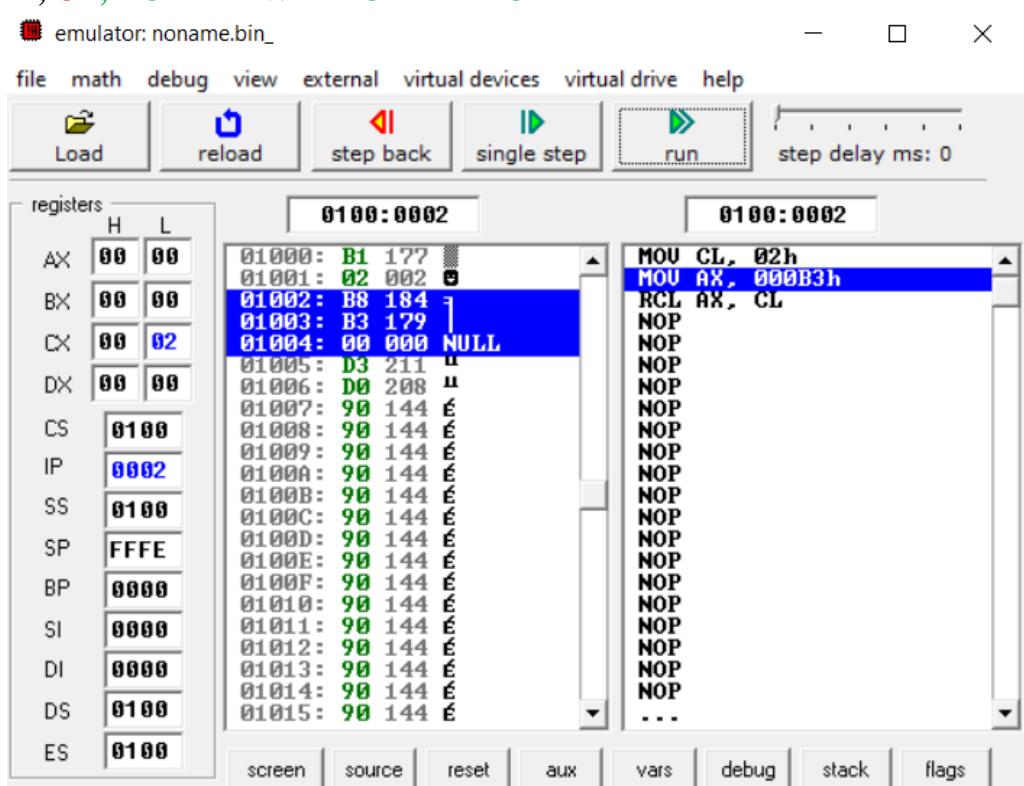


Fig 3.19: Emulator output for RCL (After Step 1)

Fig 3.20: Emulator output for RCL (After Step 2)

Fig 3.21: Emulator output for RCL (After Step 3)

- **RCR**

**MOV CL, 2; COUNTER**

**MOV AX, 0B3H; MOVE 0B3H TO AX**

**RCR AX, CL; ROTATE WITH CARRY TO RIGHT**

The screenshot shows the emulator interface with the title bar "emulator: noname.bin\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. The toolbar has buttons for Load, reload, step back, single step, run, and step delay ms: 0. The registers window shows the following values:

	H	L
AX	00	00
BX	00	00
CX	00	02
DX	00	00
CS	0100	
IP	0002	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

The assembly code window shows the instruction at address 0100:0002: MOU CL, 02h followed by RCR AX, CL. The instruction at address 0100:0004 is highlighted.

Fig 3.22: Emulator output for RCR (After Step 1)

This screenshot shows the state after the second step. The registers window shows CX updated to 00 03. The assembly code window shows the instruction at address 0100:0005: RCR AX, CL. The instruction at address 0100:0004 is highlighted.

Fig 3.23: Emulator output for RCR (After Step 2)

This screenshot shows the state after the third step. The registers window shows CX updated to 00 02. The assembly code window shows the instruction at address 0100:0007: NOP. The instruction at address 0100:0004 is highlighted.

Fig 3.24: Emulator output for RCR (After Step 3)

- AND

**MOV AX, 5**

**MOV BX, 5**

AND AX, BX

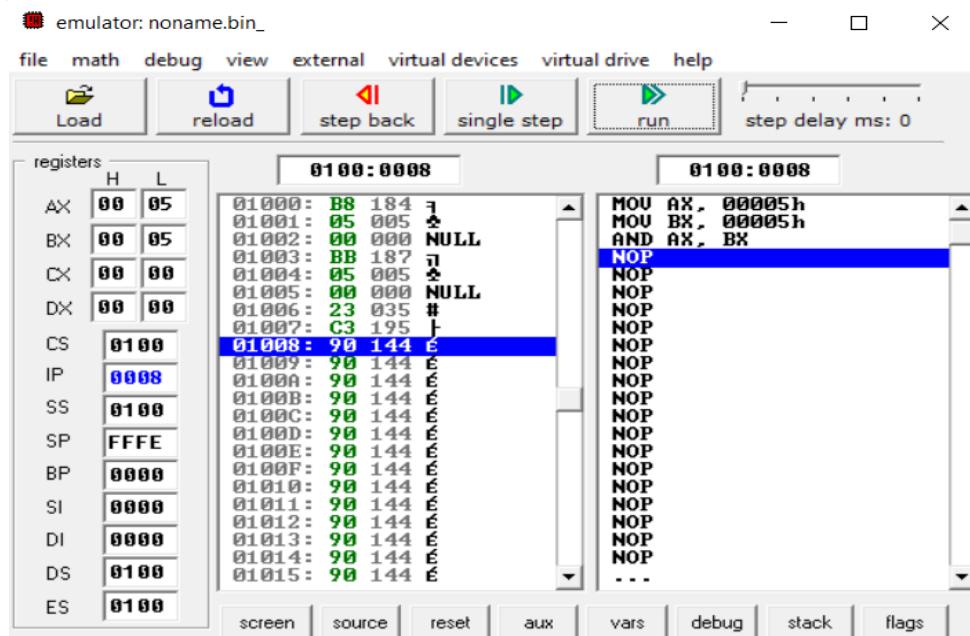


Fig 3.25: Emulator output for AND

- OR

**MOV AX, 5**

**MOV BX, 5**

**OR AX, BX**

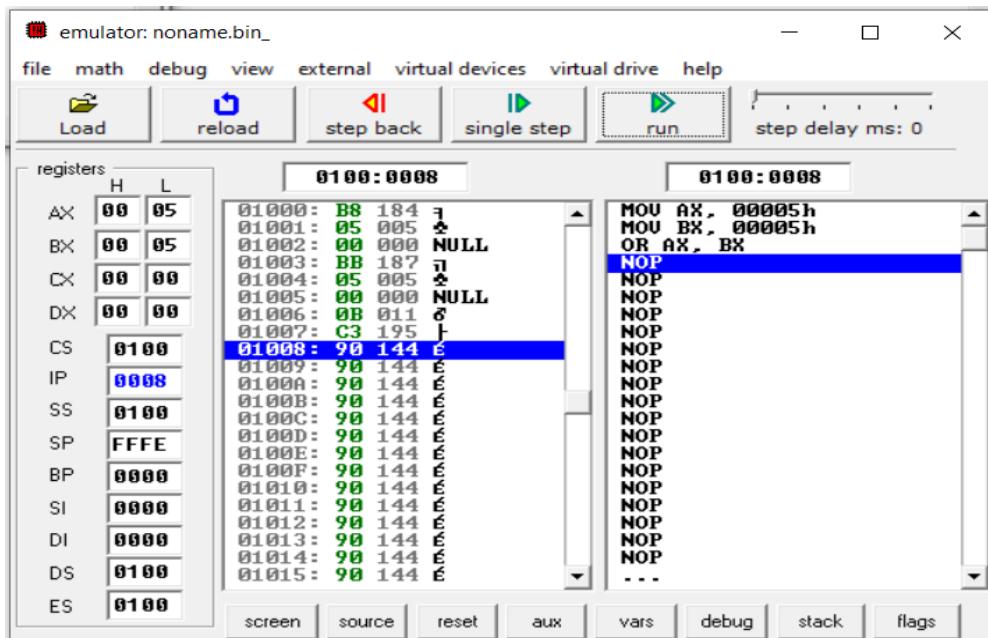


Fig 3.26: Emulator output for OR

- ### • XOR

**MOV AX, 5**

**MOV BX, 5**

### XOR AX, BX

Fig 3.27: Emulator output for XOR

- LOOP
- MOV CL, 5**  
**MOV AL, 10**
- TOP:**  
**DEC AL**  
**LOOP TOP**

Fig 3.27: Emulator output for LOOP

### 3.6 Discussion & Conclusion

In this lab, we practiced assembly language programming on the emu8086 for 1-byte or 2-byte data. We also studied about arithmetic and shift-shift programming languages.

The shift-left command moves the bit to the left, while the shift-right instruction moves it to the right. The only distinction between shift and rotate instructions is that rotate instructions

cycle the bits. While shift rotates the parts out, around goes out one side and comes in the other. either side, leaving the region where the rotated bits were either unmodified or zeroes.

We also observed the process of AND, OR, and X-OR operation in the program. Finally, the objective of the experiment was achieved.

## Experiment No. 04

### **4.1 Experiment Name**

Experimental Study of Stack Operation and Introduction to Procedures

### **4.2 Objectives**

- To understand the structure of Assembly Language programming
- To learn about the microprocessor emulator "Emu 8086" and its operation
- To get acquainted with the stack operation and its implementation

### **4.3 Theory**

The Assembly language is a low-level computer programming language that consists primarily of symbolic versions of a computer's machine language.

A stack is essentially a storage device that stores data in the LIFO (last in, first out) order. It is a memory unit with a Stack Pointer address register (SP). Stacking is broken down into two operations:

- **POP:** Deletion (decrements SP); removes the top word to the stack
- **PUSH:** Insertion (increments SP); adds a new top word to the stack

Addresses and data are saved in the stack when the CPU branches to a procedure. In RAM, a LIFO (last in, first out) data structure is implemented. This stack was then pushed to the top with the return address. In addition, the stack is used to swap the values of two registers and register pairs.

To write a program, a stack segment is required after ".MODEL" segment.

A ".STACK" segment is used to set a block of memory to store the stack.

After that a data segment ".DATA" is declared followed by a ".CODE" segment which by the way contains instructions of the program.

Then "MAIN PROC" segment acts as an entry point to the program. In some other cases, they may call other procedures, or themselves. These are mainly of two kinds: NEAR and FAR. NEAR segment is the same code segment as the calling program. But FAR is in different segment.

This segment is ended with "MAIN ENDP" where other procedures go. Both PROC and ENDP are pseudo-ops that delineate the procedure.

"END MAIN" segment ends the program and must be followed by the name of the main procedure.

### **4.4 Apparatus**

- Emu 8086 - Microprocessor Emulator

#### 4.5 Emulator Code

```
.MODEL SMALL
.STACK 100
.DATA
.VAR1 DB 12H
.CODE
MAIN PROC
MOV SP, 0101H
MOV AX, 1234H
PUSH AX
MOV BX, 4321H
PUSH AX
POP AX
POP BX
MAIN ENDP
END MAIN
```

#### 4.6 Output

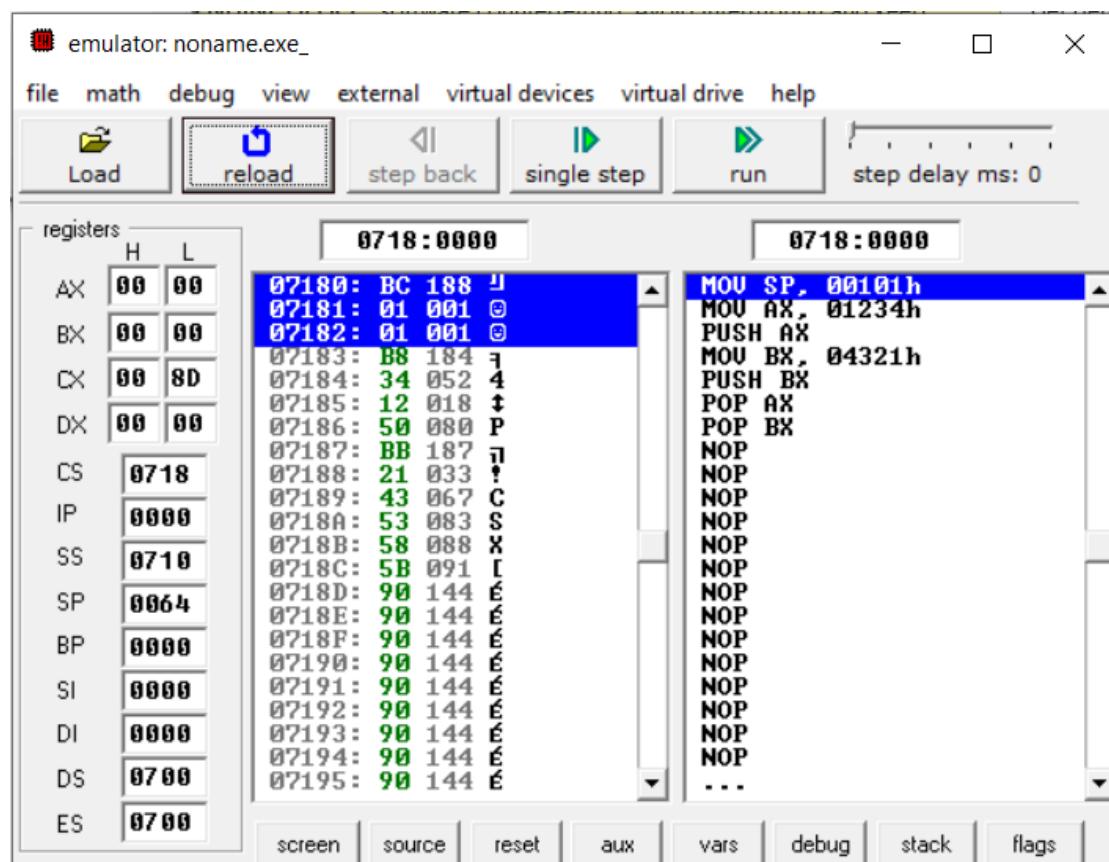


Fig 4.1: Emulator output for ALU for Stack operation (After Step 0)

The screenshot shows the emulator interface with the title bar "emulator: noname.exe\_". The menu bar includes file, math, debug, view, external, virtual devices, virtual drive, and help. Below the menu is a toolbar with buttons for Load, reload, step back, single step, run, and step delay ms: 0. On the left, a register window shows the following values:

	H	L
AX	00	00
BX	00	00
CX	00	8D
DX	00	00
CS	0718	
IP	0003	
SS	0710	
SP	0101	
BP	0000	
SI	0000	
DI	0000	
DS	0700	
ES	0700	

The assembly code window displays memory starting at address 07180. The instruction at 07183 is highlighted in blue: B8 184. The code window also shows the following assembly instructions:

```

07180: BC 188 J
07181: 01 001 @
07182: 01 001 @
07183: B8 184 J
07184: 34 052 4
07185: 12 018 +
07186: 50 080 P
07187: BB 187 I
07188: 21 033 !
07189: 43 067 C
0718A: 53 083 S
0718B: 58 088 X
0718C: 5B 091 L
0718D: 90 144 E
0718E: 90 144 E
0718F: 90 144 E
07190: 90 144 E
07191: 90 144 E
07192: 90 144 E
07193: 90 144 E
07194: 90 144 E
07195: 90 144 E
...

```

At the bottom of the interface are buttons for screen, source, reset, aux, vars, debug, stack, and flags.

Fig 4.2: Emulator output for ALU for Stack operation (After Step 1)

This screenshot shows the state of the emulator after Step 2. The assembly code window now starts at address 07180. The instruction at 07186 is highlighted in blue: 50 080 P. The code window shows the following assembly instructions:

```

07180: BC 188 J
07181: 01 001 @
07182: 01 001 @
07183: B8 184 J
07184: 34 052 4
07185: 12 018 +
07186: 50 080 P
07187: BB 187 I
07188: 21 033 !
07189: 43 067 C
0718A: 53 083 S
0718B: 58 088 X
0718C: 5B 091 L
0718D: 90 144 E
0718E: 90 144 E
0718F: 90 144 E
07190: 90 144 E
07191: 90 144 E
07192: 90 144 E
07193: 90 144 E
07194: 90 144 E
07195: 90 144 E
...

```

The registers window shows the following updated values:

	H	L
AX	12	34
BX	00	00
CX	00	8D
DX	00	00
CS	0718	
IP	0006	
SS	0710	
SP	0101	
BP	0000	
SI	0000	
DI	0000	
DS	0700	
ES	0700	

Fig 4.3: Emulator output for ALU for Stack operation (After Step 2)

The screenshot shows a debugger window titled "emulator: noname.exe\_". The menu bar includes "file", "math", "debug", "view", "external", "virtual devices", "virtual drive", and "help". Below the menu are several control buttons: "Load" (with a floppy disk icon), "reload" (with a circular arrow icon), "step back" (with a left arrow icon), "single step" (with a green right arrow icon), "run" (with a green right arrow icon), and "step delay ms: 0".

The left pane displays the "registers" window with the following values:

	H	L
AX	12	34
BX	00	00
CX	00	8D
DX	00	00
CS	0718	
IP	0007	
SS	0710	
SP	00FF	
BP	0000	
SI	0000	
DI	0000	
DS	0700	
ES	0700	

The middle pane shows two assembly windows side-by-side. The left window is labeled "0718:0007" and the right is "0718:0007". The assembly code is as follows:

```

07180: BC 188 J
07181: 01 001 @
07182: 01 001 @
07183: B8 184 I
07184: 34 052 4
07185: 12 018 D
07186: 50 080 P
07187: BB 187 I
07188: 21 033 !
07189: 43 067 C
0718A: 53 083 S
0718B: 58 088 X
0718C: 5B 091 L
0718D: 90 144 E
0718E: 90 144 E
0718F: 90 144 E
07190: 90 144 E
07191: 90 144 E
07192: 90 144 E
07193: 90 144 E
07194: 90 144 E
07195: 90 144 E

```

The instruction at address 0718A (opcode 53, operand 083) is highlighted in blue. The right window shows the continuation of the assembly code:

```

MOU SP, 00101h
MOU AX, 01234h
PUSH AX
MOU BX, 04321h
PUSH BX
POP AX
POP BX
NOP
...
```

Fig 4.4: Emulator output for ALU for Stack operation (After Step 3)

This screenshot is identical to Fig 4.4, showing the same debugger interface and assembly code. The only difference is the value in the BX register, which has changed from 00 00 to 43 21.

Fig 4.5: Emulator output for ALU for Stack operation (After Step 4)

The screenshot shows a debugger window titled "emulator: noname.exe\_". The menu bar includes "file", "math", "debug", "view", "external", "virtual devices", "virtual drive", and "help". Below the menu are several control buttons: "Load", "reload", "step back", "single step", "run", and "step delay ms: 0".

The left pane displays the register values:

	H	L
AX	12	34
BX	43	21
CX	00	8D
DX	00	00
CS	0718	
IP	000B	
SS	0710	
SP	00FD	
BP	0000	
SI	0000	
DI	0000	
DS	0700	
ES	0700	

The right pane shows the assembly code at address 0718:000B:

```

07180: BC 188 J
07181: 01 001 @
07182: 01 001 @
07183: B8 184 J
07184: 34 052 4
07185: 12 018 t
07186: 50 080 P
07187: BB 187 l
07188: 21 033 !
07189: 43 067 C
0718A: 53 083 S
0718B: 58 088 X
0718C: 5B 091 L
0718D: 90 144 E
0718E: 90 144 E
0718F: 90 144 E
07190: 90 144 E
07191: 90 144 E
07192: 90 144 E
07193: 90 144 E
07194: 90 144 E
07195: 90 144 E

```

The instruction at 0718B:088 is highlighted in blue.

Fig 4.6: Emulator output for ALU for Stack operation (After Step 5)

The screenshot shows the same debugger interface as Fig 4.6, but after step 6. The assembly code at address 0718:000C has changed:

```

07180: BC 188 J
07181: 01 001 @
07182: 01 001 @
07183: B8 184 J
07184: 34 052 4
07185: 12 018 t
07186: 50 080 P
07187: BB 187 l
07188: 21 033 !
07189: 43 067 C
0718A: 53 083 S
0718B: 58 088 X
0718C: 5B 091 L
0718D: 90 144 E
0718E: 90 144 E
0718F: 90 144 E
07190: 90 144 E
07191: 90 144 E
07192: 90 144 E
07193: 90 144 E
07194: 90 144 E
07195: 90 144 E

```

The instruction at 0718C:091 is highlighted in blue.

Fig 4.7: Emulator output for ALU for Stack operation (After Step 6)

The screenshot shows a debugger window titled "emulator: noname.exe\_". The menu bar includes "file", "math", "debug", "view", "external", "virtual devices", "virtual drive", and "help". Below the menu are several control buttons: "Load", "reload", "step back", "single step", "run", and "step delay ms: 0".

The left side displays the "registers" pane, which lists CPU registers AX, BX, CX, DX, CS, IP, SS, SP, BP, SI, DI, DS, and ES. The values for AX, BX, CX, and DX are shown in a 2x2 grid. The current instruction pointer (IP) is at address 0718:000D.

The main area shows two panes of assembly code. The left pane is labeled "0718:000D" and the right pane is labeled "0718:000D". The assembly code consists of the following instructions:

```

07180: BC 188 J
07181: 01 001 @
07182: 01 001 @
07183: B8 184 1
07184: 34 052 4
07185: 12 018 2
07186: 50 080 P
07187: BB 187 11
07188: 21 033 !
07189: 43 067 C
0718A: 53 083 S
0718B: 58 088 X
0718C: 5B 091 I
0718D: 90 144 E
0718E: 90 144 E
0718F: 90 144 E
07190: 90 144 E
07191: 90 144 E
07192: 90 144 E
07193: 90 144 E
07194: 90 144 E
07195: 90 144 E

```

The instruction at address 0718D is highlighted with a blue selection bar. The right pane shows the stack operations:

```

MOU SP, 00101h
MOV AX, 01234h
PUSH AX
MOU BX, 04321h
PUSH BX
POP AX
POP BX
NOP

```

At the bottom of the interface are buttons for "screen", "source", "reset", "aux", "vars", "debug", "stack", and "flags".

Fig 4.8: Emulator output for ALU for Stack operation (After Step 7)

#### 4.7 Discussion & Conclusion

In this experiment, we learned about the techniques for building an assembly language program. We understand the procedure code and how the stack operation works.

Inserting an element into the stack is known as a "push" & "pop" operation and the new element is positioned at the top of the stack since there is only one other location where it can be added: the very top of the stack.

We also became acquainted with the term 'pseudo-ops' and their operations. Finally, we utilized all of these practically. Thus, the objective of the experiment was achieved.

## Experiment No. 05

### **5.1 Experiment Name**

Experimental study of String operation using assembly language

### **5.2 Objectives**

- To understand the structure of Assembly Language programming
- To learn about the microprocessor emulator "Emu 8086" and its operation
- To get acquainted with the string operation and its implementation

### **5.3 Theory**

A memory string or string is essentially a byte or word array in the 8086-assembly language. String instructions are thus intended for array processing. The string instructions' tasks can be completed out using the register indirect addressing mode. Each string instruction may require a source, a destination, or both operands.

In string operations, the direction of action is chosen by the direction flag (DF). The direction flag determines the action direction in string operations (DF). These operations are carried out by SI and DI in this location.

If DF = 0, the instruction "CLD" is executed, and SI and DI proceed in the direction of increasing memory addresses (from left to right across the string), If DF = 1, the instruction STD is issued, and SI and DI proceed in the direction of decreasing memory addresses (from right to left),

MOVSB, MOVSW, REP, STOSB, LODSB, STASB, and CMPSB are some more string commands.

- **MOVSB:** This MOV instruction is used to copy the contents of one string into another string. The MOVSB instruction copies the contents of the byte addressed by DS:SI, to the byte addressed by ES:DI.
- **MOVSW:** MOVSW is a function that moves a word from the source string to the destination string. It expects DS:DI to point to a source string word and ES:DS to point to a destination string, just like MOVSB. SI and DI are both increased by 2 if DF is 0 and decreased by 2 if DF is 1.
- **REP:** MOVSB transfers only one byte from the source string to the destination string. Before executing the REP prefix, we must first initialize CX to the number n of bytes in the source string, which causes MOVSB to be executed n times in order to relocate the entire string. After each MOVSB, CX is reduced until it reaches zero.
- **CLD:** For a clear direction flag, set DF to 0, which causes the operation to run from left to right.
- **STD:** For set direction flagging, DF = 1, causing the procedure to proceed from right to left.
- **STOSB:** Stores string bytes and transfers the contents of the AL register to the byte specified by ES:DI. DI is calculated as increment DF = 0 DF=0 or decrement DF = 1 DF=1.

- **STOSW:** Saves a string word and copies the contents of the AL register to the byte specified by ES:DI. DI is calculated as increment DF = 0 DF=0 or decrement DF = 1 DF=1.
- **LODSB:** This instruction is used to transfer the byte addressed by DS:SI into AL. Then, depending on whether DF is zero or one, SI is increased or decreased. The word form converts the word addressed by DS, SI, into AX, with SI increasing by 2 if DF is zero and decreasing by 2 if DF is one. LODSB may analyze the characters in a string. loading a byte string
- **LODSW:** Load string word; the other operation is well known to LODSB.
- **CMPSW:** String words are compared.
- **CMPSB:** string byte comparison

## 5.4 Apparatus

- Emu 8086 - Microprocessor Emulator

## 5.5 Emulator Code & Output

### 5.5.1 Duplicate a string and print it

```
.MODEL SMALL
.STACK 100
.DATA
STRING1 DB 'ASHRAF'
STRING2 DB 6 DUP (?)
.CODE
MAIN PROC NEAR
    MOV AX, @DATA
    MOV DS, AX
    MOV ES, AX
    LEA SI, STRING1
    LEA DI, STRING2
    CLD
    MOV CL, 6
    REP MOVSB
MAIN ENDP
END MAIN
```

## Output

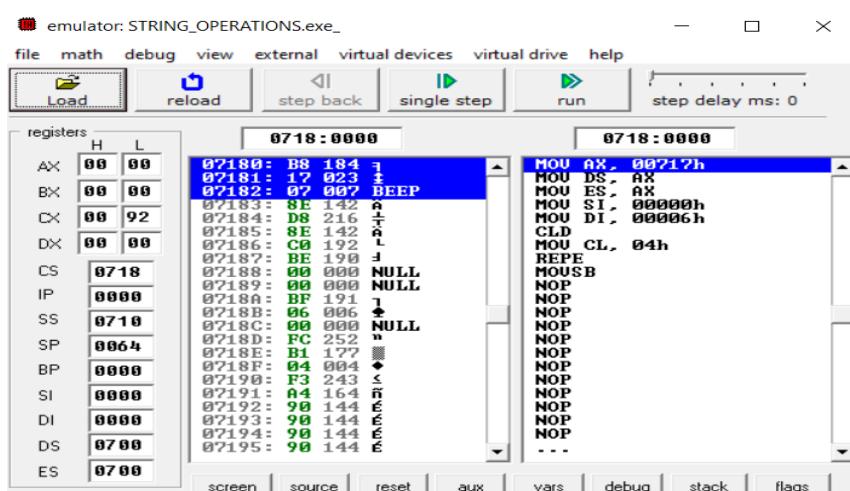


Fig 5.1.1: Emulator output for ALU for string operation (After Step 0)

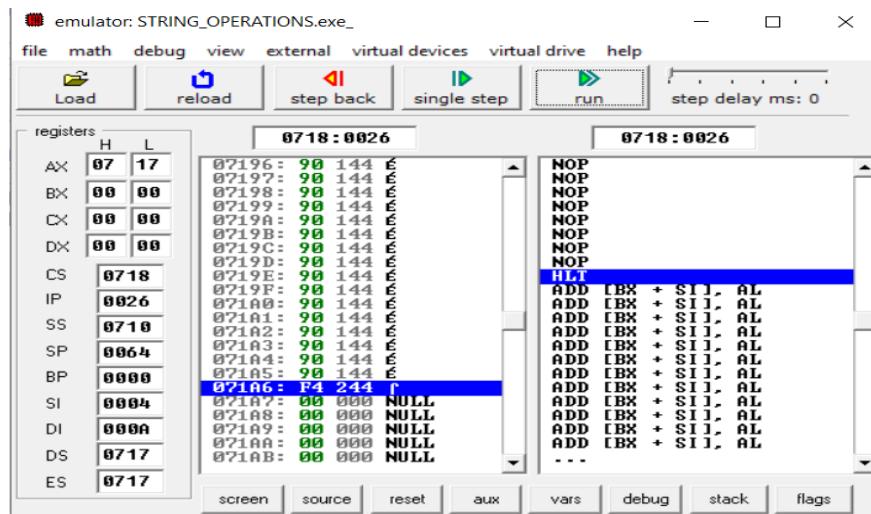


Fig 5.1.6: Emulator output for ALU for string operation (After Step 5)

Random Access Memory		
0717:0000	update	table
0717:0000: 41 065 A		
0717:0001: 53 083 S		
0717:0002: 48 072 H		
0717:0003: 52 082 R		
0717:0004: 41 065 A		
0717:0005: 46 070 F		
0717:0006: 41 065 A		
0717:0007: 53 083 S		
0717:0008: 48 072 H		
0717:0009: 52 082 R		
0717:000A: 41 065 A		
0717:000B: 46 070 F		
0717:000C: 00 000 NULL		
0717:000D: 00 000 NULL		
0717:000E: 00 000 NULL		
0717:000F: 00 000 NULL		
0717:0010: B8 184 1		
0717:0011: 17 023 2		
0717:0012: 07 007 BEEP		
0717:0013: 8E 142 Ä		
0717:0014: D8 216 T		
0717:0015: 8E 142 Ä		
0717:0016: C0 192 L		

Fig 5.1.3: Memory output for ALU for string operation (final)

**5.5.2 The same as Task Just this time, the copied phrase will begin with 0000H and the term that will be duplicated will begin after the copied term.**

```
.MODEL SMALL
.STACK 100
.DATA
STRING2 DB 8 DUP(?)
STRING1 DB 'EEE_RUET'
.CODE
MAIN PROC NEAR
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
LEA SI, STRING1
LEA DI, STRING2
CLD
MOV CL, 8
```

```
REP  
MOVSB  
MAIN ENDP  
END MAIN
```

## Output

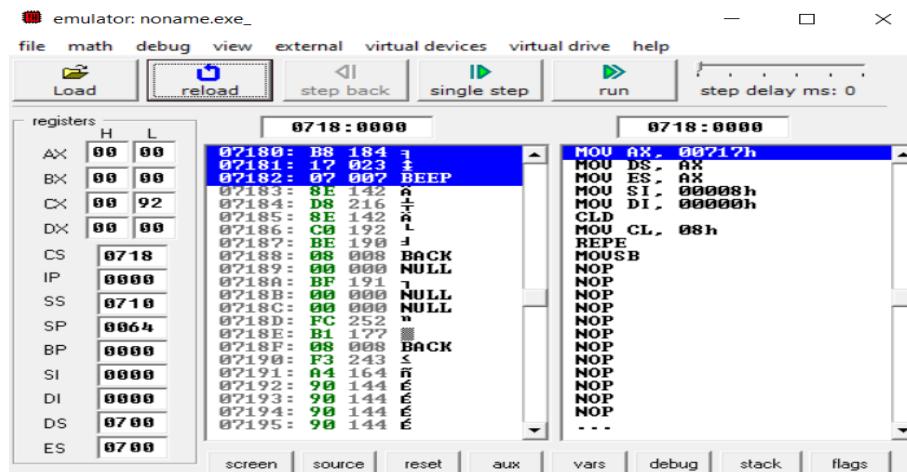


Fig 5.2.1: Emulator output for ALU for string operation (After Step 0)

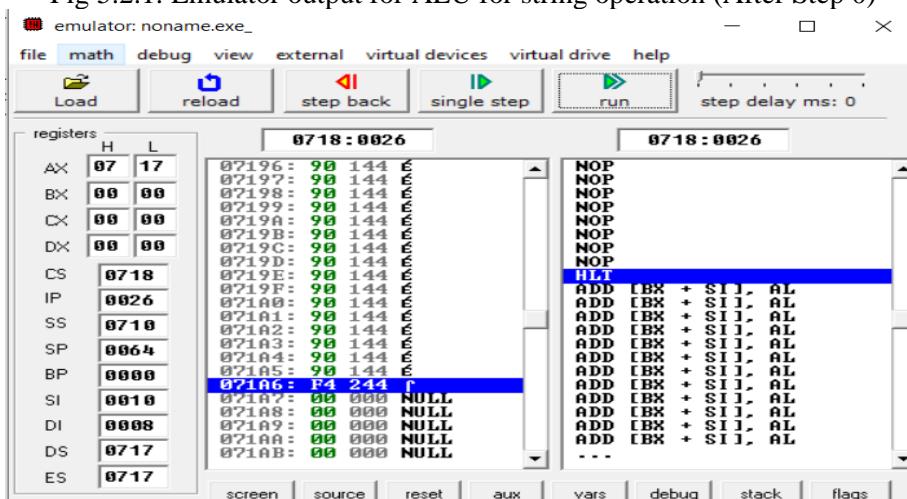


Fig 5.2.2: Emulator output for ALU for string operation (After final step)

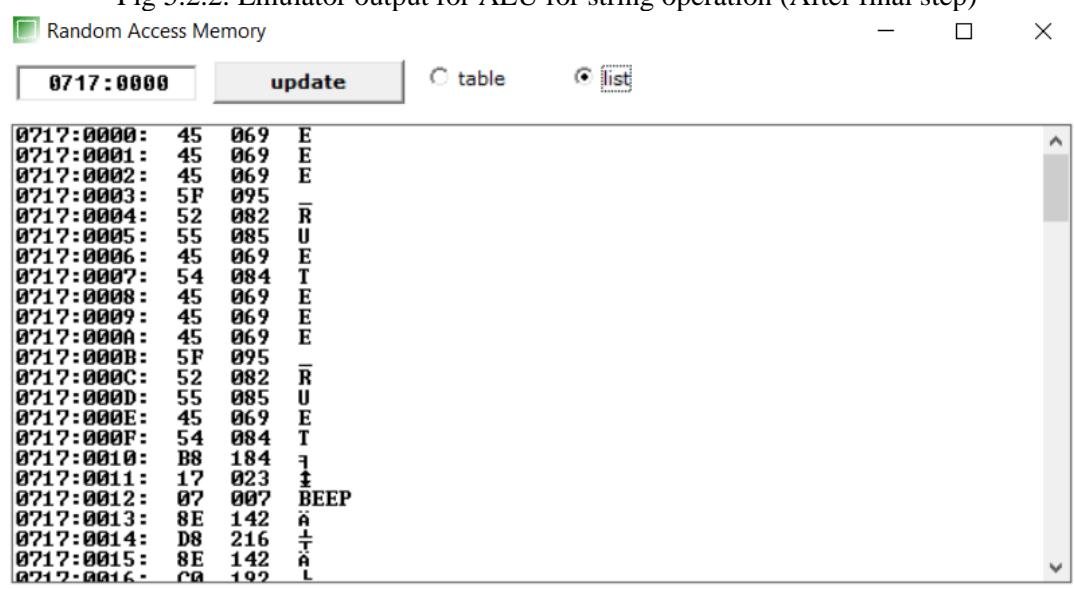


Fig 5.2.3: Memory output for ALU for string operation (final)

### 5.5.3 String correction of 'HLLO' to 'HELLO'

```
.MODEL SMALL
.STACK 100
.DATA
STRING1 DB 'HLLO'
.CODE
MAIN PROC NEAR
    MOV AX, @DATA
    MOV DS, AX
    MOV ES, AX
    LEA SI, STRING1+3
    LEA DI, STRING2+4
    STD
    MOV CL, 3
    REP MOVSB
    MOV AL, 'E'
    STOSB
MAIN ENDP
END MAIN
```

#### Output

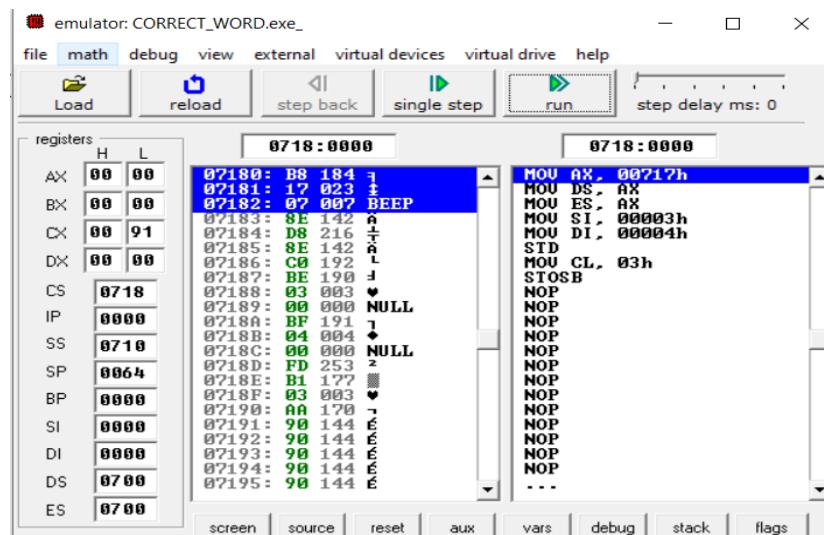


Fig 5.3.1: Emulator output for ALU for string operation (After Step 0)

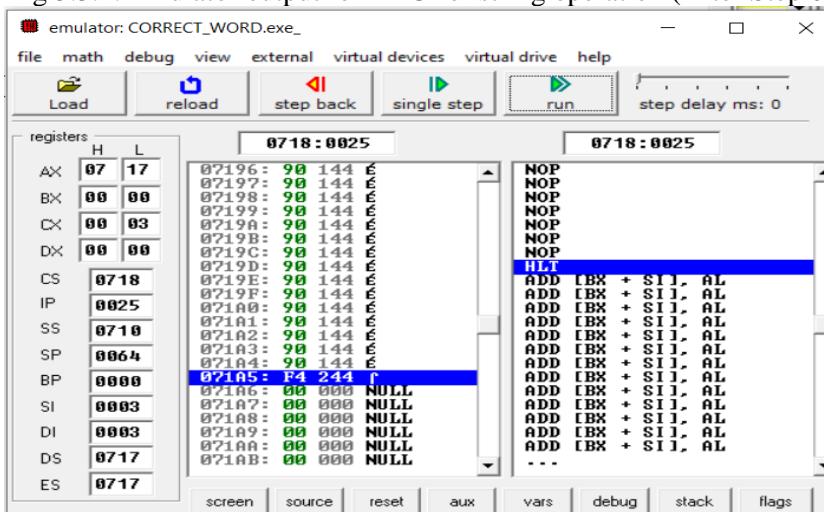


Fig 5.3.2: Emulator output for ALU for string operation (After final step)

0717:0000		update		<input type="radio"/> table	<input checked="" type="radio"/> list
0717:0000:	48	072	H		
0717:0001:	45	069	E		
0717:0002:	4C	076	L		
0717:0003:	4C	076	L		
0717:0004:	4F	079	O		
0717:0005:	00	000	NULL		
0717:0006:	00	000	NULL		
0717:0007:	00	000	NULL		
0717:0008:	00	000	NULL		
0717:0009:	00	000	NULL		
0717:000A:	00	000	NULL		
0717:000B:	00	000	NULL		
0717:000C:	00	000	NULL		
0717:000D:	00	000	NULL		
0717:000E:	00	000	NULL		
0717:000F:	00	000	NULL		
0717:0010:	B8	184	I		
0717:0011:	17	023	F		
0717:0012:	07	007	BEEP		
0717:0013:	8E	142	Ä		
0717:0014:	C0	192	L		
0717:0015:	8E	142	Ä		
0717:0016:	D9	216	I		

Fig 5.3.3: Memory output for ALU for string operation (final)

## 5.6 Discussion & Conclusion

In this experiment, we learned about the techniques for building an assembly language program. We understand the procedure code and how the string operation and some of its instructions work. An experimental test of the string's functionality was performed using the well-known application emu8086.

Finally, we utilized all of these practically. The results of the experiment and our manual computation were comparable. Thus, the objective of the experiment was achieved.

## Experiment No. 06

### **6.1 Experiment Name**

Familiarization with serial monitor mode of MDA 8086 trainer kit

### **6.2 Objectives**

- To get acquainted with the "MDA 8086" Trainer Board and its operation
- To learn how to implement program in "MDA 8086" Trainer Board and interconnect it with "Emu 8086"

### **6.3 Theory**

The MDA 8086 consists of a central processing unit (CPU), ROM, SRAM, display, keyboard, speaker, DOT matrix LED, A/D & D/A converter, stepping motor. It also has a 16-bit microprocessor, a 20-bit address bus allows it to directly access 220 bits, or 1 MB, of memory. The 8086 has fourteen sixteen-bit registers and clock frequency (5 to 10 MHz)

An 8086 MDA kit is available in two configurations-

**1. Kit mode:** Commands are entered using the kit's keyboard, and the results are displayed on the kit's monitor. Machine codes are captured using the emu8866 software.

**2. PC mode or serial monitor mode:** A serial monitor is the basic monitor application for MDA-8086 and computer data transfer.

In serial monitor mode, the monitor starts working as soon as the power is turned on. It features a memory-checking method in addition to all of the key functions. The MDA-8086 kit was supplied the machine codes (produced by Emu 8086 software) to execute the operations in kit machine mode.

The program code should be written in Notepad and saved as an ASM file. Microsoft Macro Assembler (MASM) software is used to transform an ASM file. The LOD18C software is used. ASM files can be created from OBJ files. Using the 'Comm' software, the code is then loaded into the MDA 8086 trainer kit.

### **6.4 Apparatus**

- MDA 8086 - Trainer Board

### **6.5 Working procedures**

**Step 1.** We connected MDA 8086 to the designated PC and open ASM8086. This will pop up a file directory screen on the monitor.

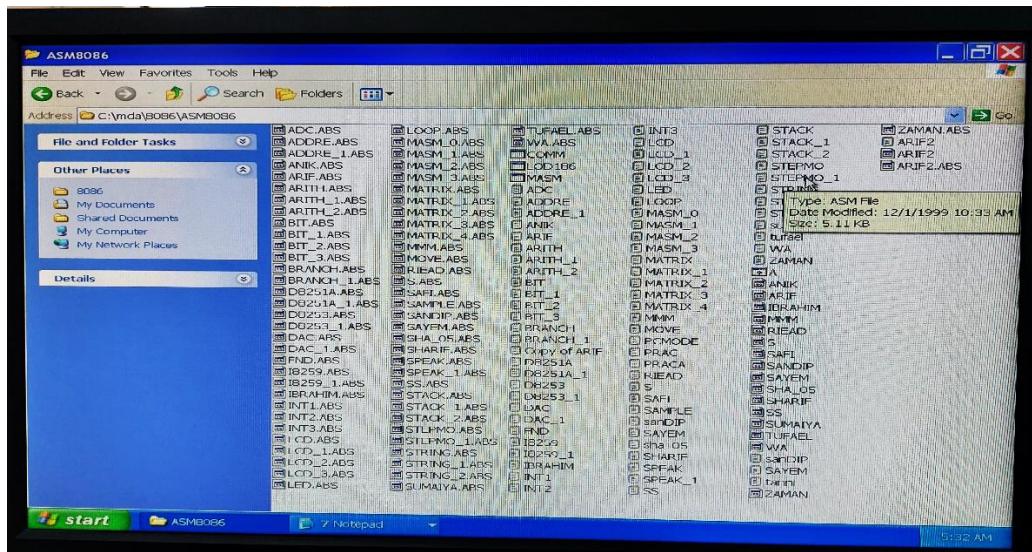


Fig. 6.1: Opening file directory

**Step 2.** In Notepad's extended mode, we wrote a program and saved it as "filename.asm" format.

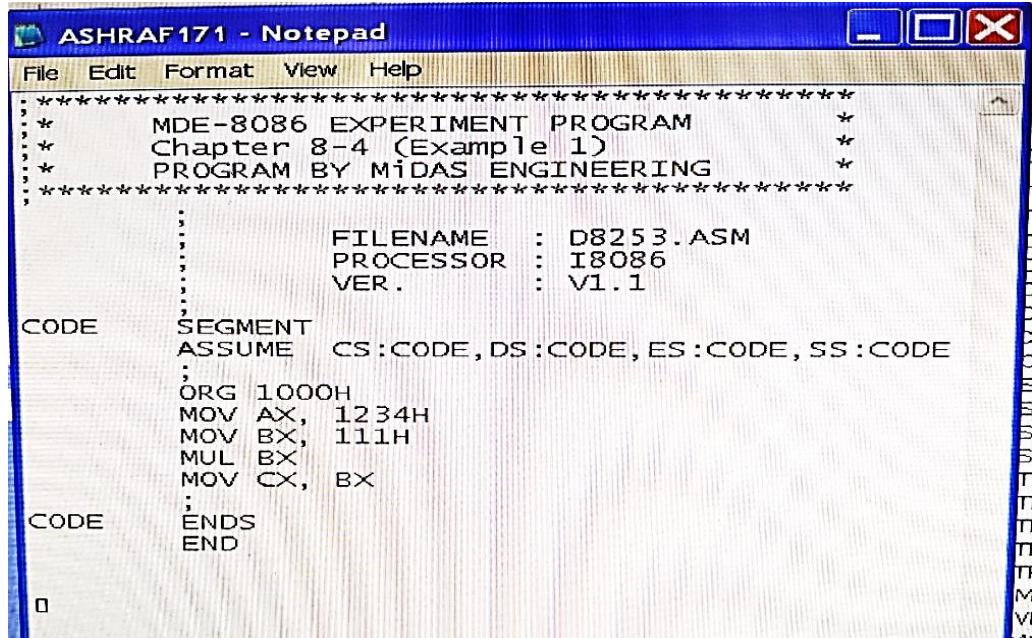


Fig. 6.2: Writing program on notepad

**Step 3.** Next, we launched the ‘MASM.exe’ software and load the ‘.asm’ file and convert it to ‘.obj’ file format.



Fig. 6.3: Load the ‘.asm’ file and convert it to ‘.obj’ file format.

**Step 4.** Then we loaded the ".obj" file into the "LOD186.exe" software, and the software will execute the ".abs" file, which turns the machine code into executable form.

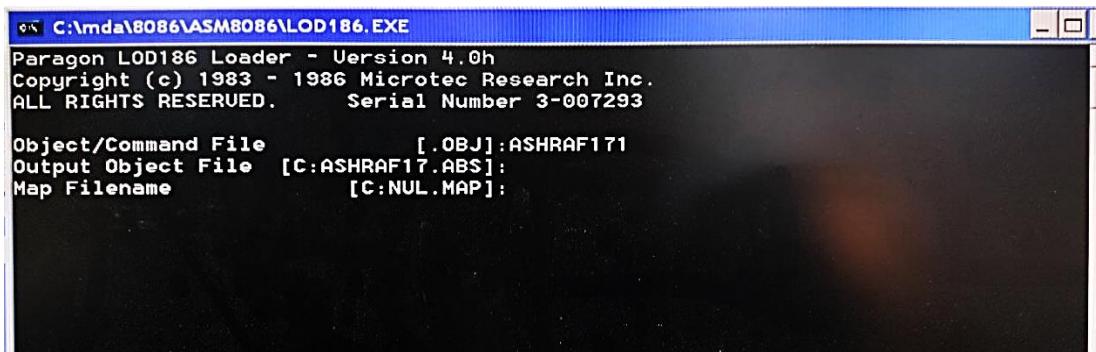


Fig. 6.4: Turn ‘.obj’ file to ‘.abs’ file

**Step 5.** Then, we selected the serial port from the ‘COMM.exe’ file.

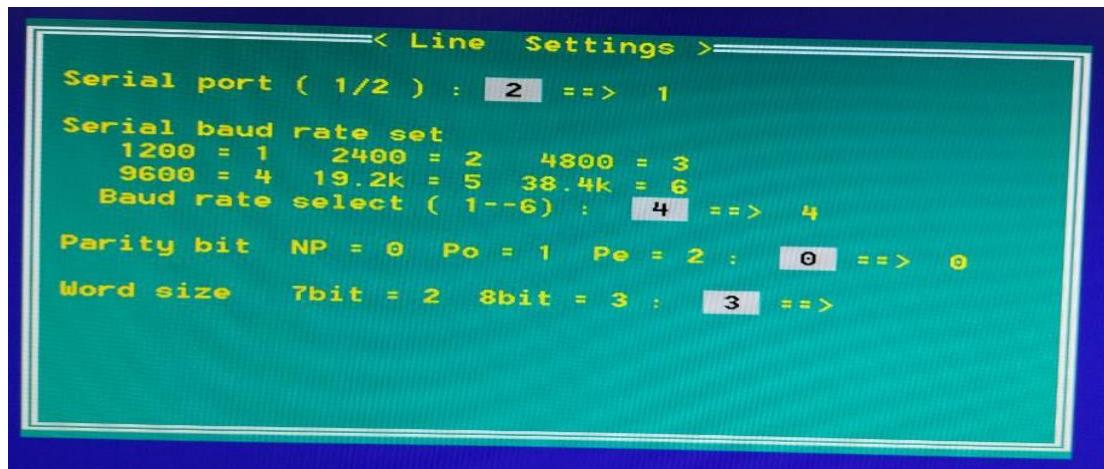


Fig. 6.5: Select serial port

**Step 6.** The 8086 MDA kit's reset button was pressed.

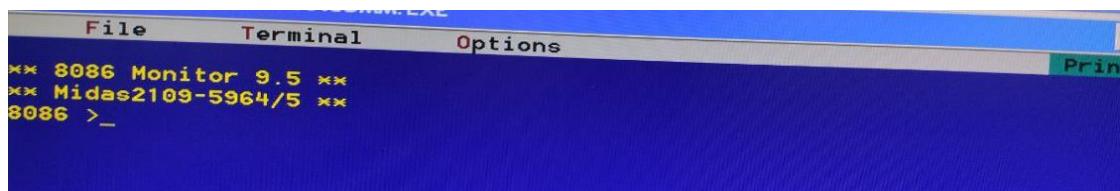


Fig. 6.6: Reset the kit

**Step 7.** After that, load the ‘.abs’ file and press T for single step execution until final output is obtained.

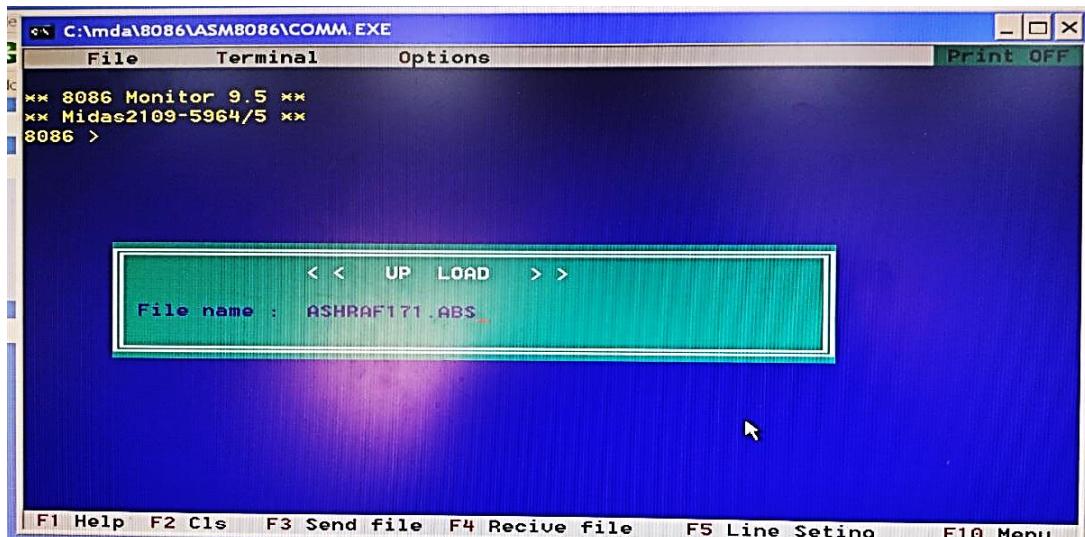


Fig. 6.7: Load the file

```
:0A100000B83412BB33542BD88BCB4D:00
OK Completed !!

8086 >T
AX=1234 BX=0000 CX=0000 DX=0000
SP=0540 BP=0000 SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000
IP=1003 FL=0100 = . . . t . . . . .

8086 >T
AX=1234 BX=5433 CX=0000 DX=0000
SP=0540 BP=0000 SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000
IP=1006 FL=0100 = . . . t . . . . .

8086 >T
AX=1234 BX=41FF CX=0000 DX=0000
SP=0540 BP=0000 SI=0000 DI=0000
DS=0000 ES=0000 SS=0000 CS=0000
IP=1008 FL=0114 = . . . t . . . p .
```

Fig. 6.8: After single step executing several times

## 6.6 Discussion & Conclusion

The procedure of working in serial monitor mode or PC mode with the 8086 MDA kit was introduced in this experiment. In this case, we used the MDA 8086 kit's direct monitor mode. We used code to perform the multiplication of two numbers.

A program was developed in notepad and saved as ‘.asm’ file, which was then translated to ‘.obj’ using Microsoft Macro Assembler. The ‘.obj’ file was then translated into a ‘.abs’ file using the LOD186 software. The ‘.abs’ file was then executed by following the instructions, and the output was observed using both single-step execution and direct execution.

## **Experiment No. 07**

## 7.1 Experiment Name

## 8255 PPI interfacing: 7 - segment display

## 7.2 Objectives

- To get acquainted with the "MDA 8086" Trainer Board and its operation
  - To learn how to implement program in “MDA 8086” Trainer Board and interconnect it with “Emu 8086”

### 7.3 Theory

A serial monitor mode is one of the configurations of 8086 MDA kit. Here, the monitor starts working as soon as the power is turned on. The program code should be written in Notepad and saved as an ASM file, which is transformed to OBJ files. Using the 'Comm' software, the code is then loaded into the MDA 8086 trainer kit.

For this experiment, we applied 8255 interfacing in 7 – segment decoder. Here, the LED, has seven segments, is used as a highly frequent output device. Furthermore, we have eight segments in an LED display, which contains '.', which is character 8 with a decimal point, dp directly next to it. The segments are denoted as 'a, b, c, d, e, f, g, and dp.' Furthermore, these are LEDs, or a series of Light Emitting Diodes.

## 7.4 Apparatus

- MDA 8086 - Trainer Board

## **7.5 Experimental problem no. 01**

## **CODE SEGMENT**

```
ASSUME CS: CODE, DS: CODE, SS: CODE, ES: CODE
ORG 1000H
PORTA EQU 19H
CONTRL EQU 1FH
MOV AL, 100000000B
OUT CONTRL, AL
MOV AL, 100000000B
OUT PORTA, AL
CODE ENDS
```

END

## 7.6 Output procedures

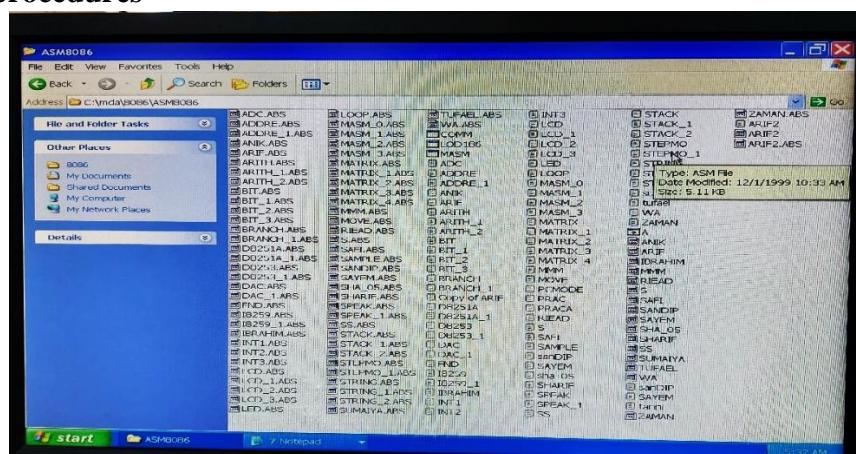


Fig. 7.1: Opening file directory

```

GROUP02 - Notepad
File Edit Format View Help
*****
* MDE-8086 EXPERIMENT PROGRAM
* Chapter 7-5 (Example 4)
* PROGRAM BY MIDAS ENGINEERING
*****
;
; FILENAME : MATRIX_3.ASM
; PROCESSOR : I8086
; VER.      : V1.1
;
CODE SEGMENT
ASSUME CS:CODE, DS:CODE, ES:CODE, SS:CODE;
PORTA ORG 1000H
CNTRL EQU 19H
EQU 1FH
MOV AL, 10000000B
OUT CNTRL, AL
MOV AL, 10000000B
OUT PORTA, AL
CODE ENDS
END

```

Fig. 7.2: Writing program on notepad

```

C:\mda\8086\ASM8086\MASM.EXE
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981-1988. All rights reserved.

Source filename [.ASM]: GROUP02
Object filename [GROUP02.OBJ]:
Source listing [NUL.LST]:
Cross-reference [NUL.CRF]:

```

Fig. 7.3: Load the ‘.asm’ file and convert it to ‘.obj’ file format.

```

C:\mda\8086\ASM8086\LOD186.EXE
Paragon LOD186 Loader - Version 4.0h
Copyright (c) 1983 - 1986 Microtec Research Inc.
ALL RIGHTS RESERVED. Serial Number 3-007293

Object/Command File [.OBJ]: GROUP02
Output Object File [C:GROUP02.ABS]:
Map Filename [C:NUL.MAP]:

```

Fig. 7.4: Turn ‘.obj’ file to ‘.abs’ file

```

< Line Settings >
Serial port ( 1/2 ) : 2 ==> 1
Serial baud rate set
  1200 = 1  2400 = 2  4800 = 3
  9600 = 4  19.2k = 5  38.4k = 6
Baud rate select ( 1--6 ) : 4 ==> 4
Parity bit  NP = 0  Po = 1  Pe = 2 : 0 ==> 0
Word size    7bit = 2  8bit = 3 : 3 ==>

```

Fig. 7.5: Select serial port

```

File Terminal Options
** 8086 Monitor 9.5 **
** Midas2109-5964/5 **
8086 >_

```

Fig. 7.6: Reset the kit

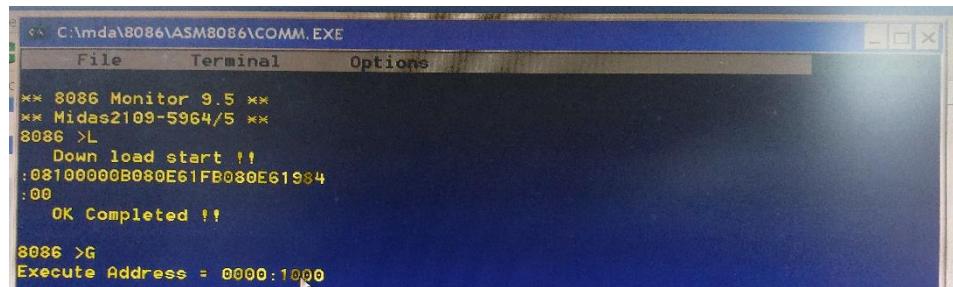


Fig. 7.8: Load the file

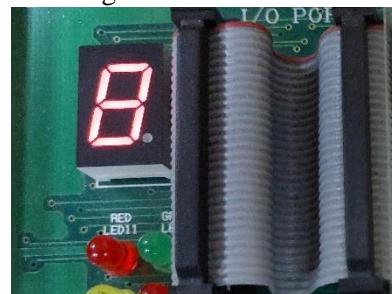


Fig. 7.9: Output

### 7.7 Experimental problem no. 02

**CODE SEGMENT**

```

ASSUME CS: CODE, DS: CODE, SS: CODE, ES: CODE
ORG 1000H
PORTA EQU 19H
CTRL EQU 1FH
MOV AL, 10000000B
OUT CTRL, AL
MOV AL, 11111001B
OUT PORTA, AL
CALL DELAY
MOV AL, 10000000B
OUT PORTA, AL
CALL DELAY
MOV AL, 11000000B
OUT PORTA, AL
CALL DELAY
MOV AL, 11000000B
OUT PORTA, AL
CALL DELAY
MOV AL, 11111001B
OUT PORTA, AL
CALL DELAY
MOV AL, 11111000B
OUT PORTA, AL
CALL DELAY
MOV AL, 11111001B
OUT PORTA, AL
CALL DELAY
DELAY PROC
MOV CX, 0FFFFH
LOOP1;
NOP
NOP
NOP
LOOP LOOP1

```

```

RET
DELAY ENDP
CODE ENDS
END

```

## 7.8 Output procedures

```

*****8086 EXPERIMENT PROGRAM*****
Chapter 7-5 (Example 4)
PROGRAM BY MIDAS ENGINEERING
*****
***** FILENAME : MATRIX_3.ASM
***** PROCESSOR : I8086
***** VER. : V1.1
*****
CODE SEGMENT
ASSUME CS:CODE,DS:CODE,ES:CODE,SS:CODE;
PORTA ORG 1000H
EQU 19H
EQU 1FH
MOV OUT AL,10000000B
CNTRL,AL
MOV OUT AL,11111001B
PORTA,AL
CALL DELAY
MOV OUT AL,10000000B
PORTA,AL
CALL DELAY
MOV OUT AL,11000000B
PORTA,AL
CALL DELAY
MOV OUT AL,11111001B
PORTA,AL
CALL DELAY
MOV OUT AL,111111001B
PORTA,AL
CALL DELAY
MOV OUT AL,10000010B
PORTA,AL
CALL DELAY
MOV OUT AL,10100100B
PORTA,AL
CALL DELAY
DELAY PROC
MOV CX,0FFFFH
LOOP1:
NOP
NOP
NOP
LOOP LOOP1
RET
DELAY ENDP

CODE ENDS
END

```

Fig. 7.10: Writing program on notepad



Fig. 7.10: 1801171 as output

## 7.9 Discussion & Conclusion

The procedure of working in serial monitor mode or PC mode with the 8086 MDA kit was introduced in this experiment. In this case, we used the MDA 8086 kit's direct monitor mode. We used code to perform the multiplication of two numbers.

A program was developed in notepad and saved as ‘.asm’ file, which was then translated to ‘.obj’ using Microsoft Macro Assembler. The ‘.obj’ file was then translated into a ‘.abs’ file using the LOD186 software. The ‘.abs’ file was then executed by following the instructions, and the output was observed using both single-step execution and direct execution.

## Experiment No. 08

### **8.1 Experiment Name**

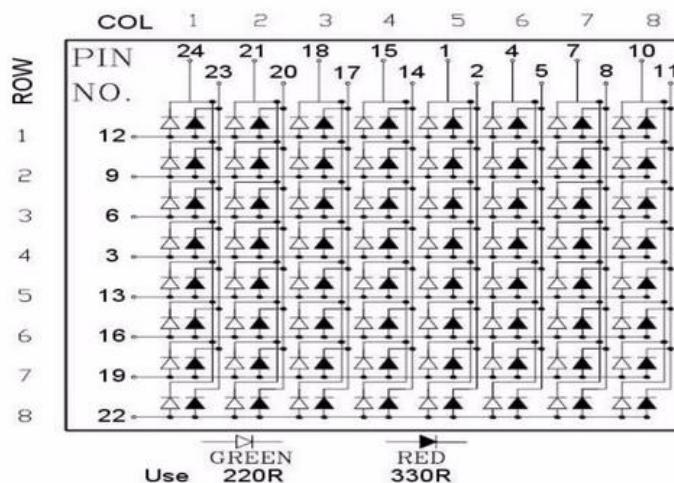
8255 PPI interfacing: Dot matrix

### **8.2 Objectives**

- To get acquainted with the "MDA 8086" Trainer Board and its operation
- To understand the operation of Dot matrix in 8255 PPI interfacing
- To learn how to implement program in “MDA 8086” Trainer Board and interconnect it with “Emu 8086”

### **8.3 Theory**

A dot Matrix consists of an array of LED's which are inter connected such that the positive terminal (anode) of each LED in the same column are connected together and the negative terminal (cathode) of each LED in the same row are connected together.



Here, the 8255 port address in accordance with the dot matrix are as follows,

Port name	Port address
Port A	18H
Port B	1AH
Port C	1CH
Control register	1EH

### **8.4 Apparatus**

- MDA 8086 - Trainer Board

### **8.5 Code for column wise operation**

CODE SEGMENT

```
ASSUME CS: CODE, DS: CODE, SS: CODE, ES: CODE
```

```
CNTRL EQU 1EH
```

```
PORTC EQU 1CH
```

```
PORTB EQU 1AH
```

```
PORTA EQU 18H
```

```
ORG 1000H
```

```
MOV AL, 100000000B
```

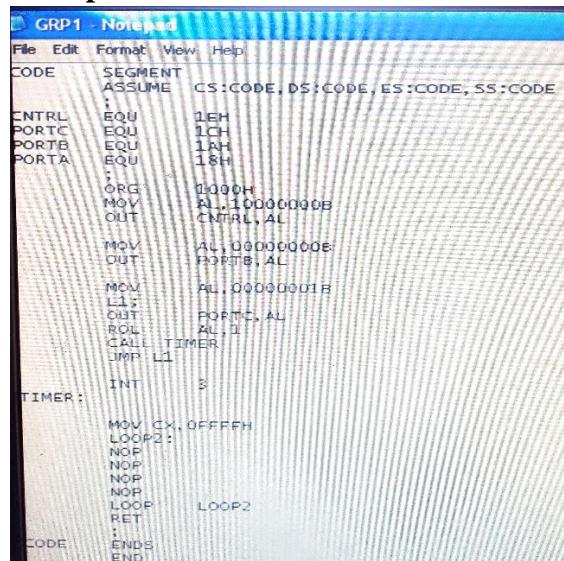
```
OUT CNTRL, AL
```

```

MOV AL, 100000000B
OUT PORTB, AL
MOV AL, 000000001B
L1:
OUT PORTC, AL
ROL AL, 1
CALL TIMER
JMP L1
INT
TIMER:
MOV CX, 0FFFFH
LOOP2:
NOP
NOP
NOP
NOP
LOOP LOOP2
RET
CODE ENDS
END

```

## 8.6 Output for column wise operation



The screenshot shows a Microsoft Notepad window titled "GRP1 - Notepad". The code is written in assembly language:

```

CODE SEGMENT
ASSUME CS:CODE, DS:CODE, ES:CODE, SS:CODE
;
CNTRL EQU 1EH
PORTC EQU 1CH
PORTB EQU 1AH
PORTA EQU 18H
;
ORG 1000H
MOV AL, 100000000B
OUT CNTRL, AL
;
MOV AL, 000000001B
OUT PORTB, AL
;
MOV AL, 00000001B
L1:
OUT PORTC, AL
ROL AL, 1
CALL TIMER
JMP L1
;
INT 3
;
TIMER:
MOV CX, 0FFFFH
LOOP2:
NOP
NOP
NOP
NOP
LOOP LOOP2
RET
CODE ENDS
END

```

Fig. 8.1: Writing program on notepad

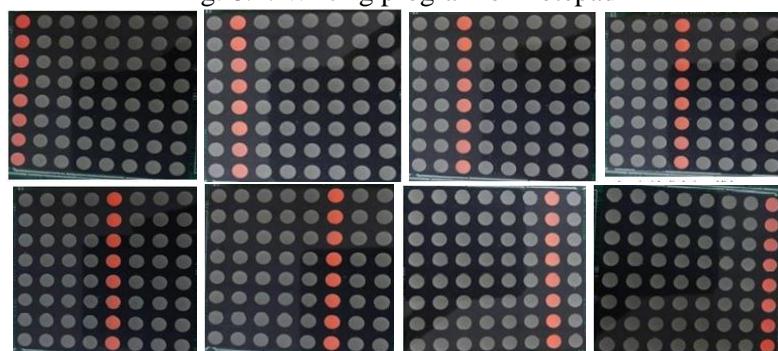


Fig. 8.2: Output

## 8.7 Code for row wise operation

```

CODE SEGMENT
ASSUME CS: CODE, DS: CODE, SS: CODE, ES: CODE

```

```

CNTRL EQU 1EH
PORTC EQU 1CH
PORTB EQU 1AH
PORTA EQU 18H
ORG 1000H
MOV AL, 100000000B
OUT CNTRL, AL
MOV AL, 11111111B
OUT PORTC, AL
MOV CL, 8
MOV AL, 01111111B
L1;
OUT PORTB, AL
ROL AL, 1
CALL TIMER
JMP L1
INT
TIMER:
    MOV CX, 0FFFFH
    LOOP2:
    NOP
    NOP
    NOP
    NOP
    LOOP LOOP2
    RET
CODE ENDS
END

```

### 8.8 Output for row wise operation

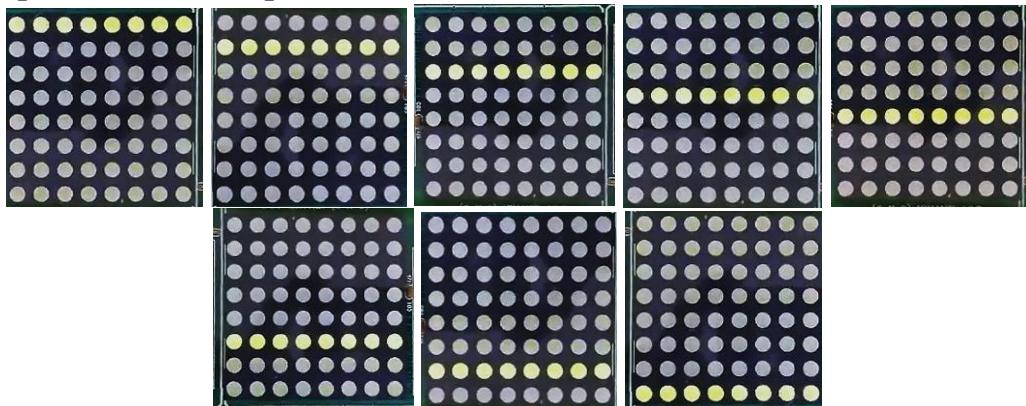


Fig. 8.3: Output

### 8.9 Discussion & Conclusion

The procedure of working in serial monitor mode or PC mode with the 8086 MDA kit was introduced in this experiment. We used code to perform the dot matrix. A program was developed in notepad and saved as ‘.asm’ file, which was then translated to ‘.obj’ which was then translated into a ‘.abs’ file. This was then executed by following the instructions, and the output was observed using both single-step execution and direct execution.

By using programming, the row-wise LEDs were turned on after the column-wise LEDs. Therefore, it may be concluded from all of the foregoing talks that the experiment was a success.

**Experiment No. 09****9.1 Experiment Name**

8255 PPI interfacing: LED

**9.2 Objectives**

- To get acquainted with the "MDA 8086" Trainer Board and its operation
- To understand working procedure of LED interfacing
- To learn how to implement program in "MDA 8086" Trainer Board and interconnect it with "Emu 8086"

**9.3 Theory**

For this experiment, we applied 8255 interfacing in LED. Four LEDs are connected to Port B's bottom four pins. Typically, the ground is connected to LED cathodes. The 8255 microcontrollers have four general-purpose I/O ports that can be configured as input or output. By designating the port pins as output, the state of the port pins can be managed and modified to high or low. When the port is configured as an input, reading the pins will show the voltage state of the pins. The port pins must be configured as outputs in order for the LED to flash.



Port name	Port address
Port A	18H
Port B	1AH
Port C	1CH
Control register	1EH

**9.4 Apparatus**

- MDA 8086 - Trainer Board

**9.5 Experimental problem no. 01****CODE SEGMENT**

```

ASSUME CS: CODE, DS: CODE, ES: CODE, SS: CODE
ORG 1000H
PORTA EQU 1BH
CNTRL EQU 1FH
MOV AL,10000000B
OUT CNTRL, AL
MOV CL,5
L1:
MOV AL,00000001B
OUT PROT B, AL
CALL DELAY
MOV AL,00000010B
OUT PROT B, AL
CALL DELAY
MOV AL,000001000B
OUT PROT B, AL
CALL DELAY

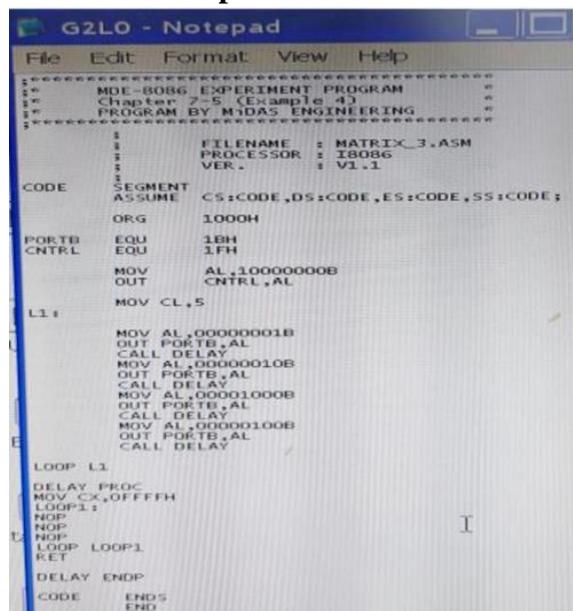
```

```

MOV AL,000000100B
OUT PROTB, AL
CALL DELAY
LOOP L1
DELAY PROC
MOV CX,0FFFFH
LOOP1:
NOP
NOP
NOP
NOP
LOOP LOOP1
RET
DELAY ENDP
CODE ENDS
END

```

## 9.6 Output as clockwise direction in Z pattern



```

G2LO - Notepad
File Edit Format View Help
***** MDE-8086 EXPERIMENT PROGRAM *****
***** Chapter 7-S (Example 4) *****
***** PROGRAM BY MIDAS ENGINEERING *****
*****
; FILENAME : MATRIX_3.ASM
; PROCESSOR : i8086
; VER. : V1.1
*****
CODE SEGMENT CS:CODE,DS:CODE,ES:CODE,SS:CODE;
ORG 1000H
PORTB EQU 1BH
CNTRL EQU 1FH
MOV AL,10000000B
OUT CNTRL,AL
MOV CL,5
L1:
MOV AL,00000001B
OUT PORTB,AL
CALL DELAY
MOV AL,000000010B
OUT PORTB,AL
CALL DELAY
MOV AL,00000100B
OUT PORTB,AL
CALL DELAY
MOV AL,00000100B
OUT PORTB,AL
CALL DELAY
LOOP L1
DELAY PROC
MOV CX,0FFFFH
LOOP1:
NOP
NOP
LOOP LOOP1
RET
DELAY ENDP
CODE ENDS
END

```

Fig. 9.1: Writing program on notepad

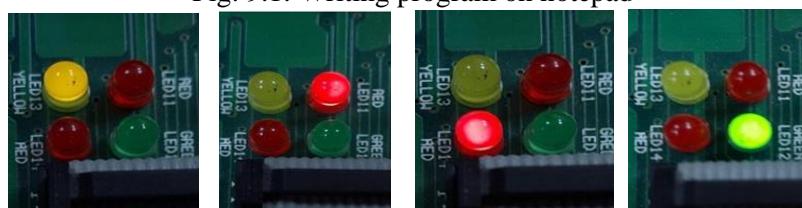


Fig. 9.2: Output

## 9.7 Discussion & Conclusion

In this experiment, we used code to perform the LED interfacing. A program was developed in notepad and saved as ‘.asm’ file, which was then translated to ‘.obj’ and later into a ‘.abs’ file. This was then executed and the output was observed using direct execution.

Here, there were two more types of patterns available on the LED interface of the 8086 kits. Here, using programming, we activated the LEDs in two different directions: clockwise & z-pattern. Therefore, it may be concluded from all of the foregoing talks that the experiment was a success.

## Experiment No. 10

### **10.1 Experiment Name**

LED ON/OFF using 8253, 8255 & 8259

### **10.2 Objectives**

- To get acquainted with the "MDA 8086" Trainer Board and its operation
- To understand working procedure of LED ON/OFF using 8253, 8255, and 8259
- To learn how to implement program in "MDA 8086" Trainer Board and interconnect it with "Emu 8086"

### **10.3 Theory**

The **8255** is a **Programmable Peripheral Interface**, is a general purpose programmable I/O device designed to interface the CPU with its outside world such as ADC, DAC, keyboard etc.

On the contrary, the **8253** is a **Programmable Timer Interval IC** which are designed for microprocessors to perform timing and counting functions using three 16-bit registers.

The **8259** is a **Priority/Programmable Interrupt Control IC** which combines the multi-interrupt input sources into a single interrupt output.

For this experiment, in order to turn on or off the LEDs concurrently with the specified time delay, the 8086 will control the 8255 PPI. The 8255 PPI IC's Port B is linked to the LEDs in the following way:

Port name	LED no.
PB <sub>0</sub>	11
PB <sub>1</sub>	12
PB <sub>2</sub>	13
PB <sub>3</sub>	14

The LEDs will be on and off in the following sequence by turning on each separately,

**LED-11(PB0) -- LED-12(PB1) -- LED-13(PB2) -- LED-14(PB3)**

The MDA 8086 kit contains I/O mapped memory. So, to communicate with the peripherals, the commands "in" and "out" are needed. The 8255 PPI-CS-2 is used to connect the LEDs. So, port addresses for 8255, 8253 PTIC, and 8259 PICIC respectively are,

8255		8253		8259	
Port name	Port address	Port name	Port address	Port name	Port address
Port A	18H	Counter - 0	09H	INTA (Command Register)	10H
Port B	1AH	Counter - 1	0BH		
Port C	1CH	Counter - 2	0DH	INTA <sub>2</sub> (Data Register)	12H
Control register	1EH	Control register	0FH		

### **10.4 Apparatus**

- MDA 8086 - Trainer Board

### **10.5 Code and Output as Z pattern**

D8253 - Notepad

File Edit Format View Help

```
*****  
* MDE-8086 EXPERIMENT PROGRAM  
* Chapter 8-4 (Example 1)  
* PROGRAM BY MIDAS ENGINEERING  
*****  
  
        FILENAME : D8253.ASM  
        PROCESSOR : I8086  
        VER. : V1.1  
  
CODE SEGMENT  
ASSUME CS:CODE, DS:CODE, ES:CODE, SS:CODE  
;  
PPIC_C EQU 1FH  
PPIC EQU 1DH  
PPIB EQU 1BH  
PPIA EQU 19H  
;  
CTC1 EQU 0BH  
CTCC EQU OFH  
;  
INTA EQU 10H  
INTA2 EQU INTA+2  
;  
INT_V EQU 40H*4  
;  
ORG 1000H  
;  
XOR BX, BX  
MOV ES, BX  
;  
MOV AX, OFFSET INT_SER  
MOV BX, INT_V  
MOV WORD PTR ES:[BX], AX  
;  
XOR AX, AX  
MOV WORD PTR ES:[BX+2], AX  
;  
CALL INIT  
CALL P_INIT  
;  
MOV AL, 10000000B  
OUT PPIC_C, AL  
;  
MOV AL, 11111111B  
OUT PPIA, AL  
;  
MOV AL, 00000000B  
OUT PPIB, AL
```

D8253 - Notepad

File Edit Format View Help

```
        OUT PPIC_C, AL  
;  
MOV AL, 11111111B  
OUT PPIA, AL  
;  
MOV AL, 00000000B  
OUT PPIB, AL  
;  
MOV AH, 11110001B  
MOV AL, AH  
OUT PPIB, AL  
;  
STI  
L2:  
NOP  
JMP L2  
;  
INT 3  
;  
INT_SER:  
SHL AH, 1  
TEST AH, 00010000B  
JNZ L1  
OR AH, 11110000B  
JMP L3  
;  
LED OUT  
L1:  
MOV AH, 11110001B  
MOV AL, AH  
OUT PPIB, AL  
;  
PUSH AX  
MOV AX, 0FFFFH  
OUT CTC1, AL  
MOV AL, AH  
OUT CTC1, AL  
POP AX  
;  
EOI Command  
MOV AL, 00100000B  
OUT INTA, AL  
;  
STI  
IRET  
;  
P_INIT PROC NEAR  
PUSH AX  
MOV AL, 01110000B  
OUT CTCC, AL  
;  
MOV AX, 0FFFFH  
OUT CTC1, AL
```

```

D8253 - Notepad
File Edit Format View Help
OUT    PPIB, AL
;
PUSH   AX
MOV    AX, OFFFFFH
OUT    CTC1, AL
MOV    AL, AH
OUT    CTC1, AL
POP    AX
; EOI  Command
MOV    AL, 00100000B
OUT    INTA, AL
STI
IRET
;
P_INIT PROC NEAR
PUSH   AX
MOV    AL, 01110000B
OUT    CTCC, AL
;
MOV    AX, OFFFFFH
OUT    CTC1, AL
MOV    AL, AH
OUT    CTC1, AL
POP    AX
RET
ENDP
;
INIT  PROC NEAR
; ICW1
MOV    AL, 00010011B
OUT    INTA, AL
; ICW2 interrupt vector
MOV    AL, 40H
OUT    INTA2, AL
; ICW4
MOV    AL, 00000001B
OUT    INTA2, AL
; interrupt mask
MOV    AL, 11111110B
OUT    INTA2, AL
RET
ENDP
;
CODE  ENDS
END

```

Fig. 9.1: Writing program on notepad

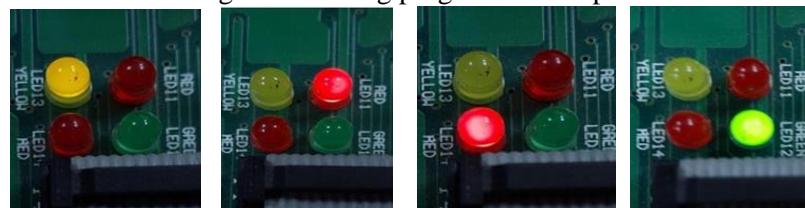


Fig. 9.2: Output

## 10.6 Discussion & Conclusion

In this experiment, we used code to perform the LED interfacing in 8255, 8253, and 8259. A program was developed in notepad and saved as ‘.asm’ file, which was then translated to ‘.obj’ and later into a ‘.abs’ file. This was then executed and the output was observed using direct execution.

To turn ON and OFF each LED sequentially and concurrently with a predetermined interval, the 8253 PTI IC provided the necessary time delay and pushed the pin IR0 of the 8259 PIC IC with an interrupt. The 8255 PPI IC, which is coupled to the LEDs, was managed by the 8086, which received this interrupt. The 8086, which was in charge of the associated 8255 LEDs, lighted them up one at a time while maintaining the required time delay.

Thus, the experiment was a success.