Practical Workbook

CS-430 Microprocessor Programming & Interfacing (BM / EE / EL / TC)

Name : ______________________________

Year : __________ Batch: ____________

Roll No. : ______________________________

Group No. : ____________________________

Department : ___________________________

Dept. of Computer & Information Systems Engineering

NED University of Engineering & Technology

INTRODUCTION

Microprocessors play a vital role in the design of digital systems. They are found in a wide range of application such as process control, communication systems, digital instruments and consumer products. Before embedding microprocessor in any system, profound knowledge and full understanding of the architecture and the instruction set of that microprocessor is imperative.

This workbook is divided into three sections. The first section deals with understanding the architecture of a basic machine having a von Neumann instruction set architecture. In these labs, students will use PRIMA VIRTUAL MACHINE, which allows them to visualize the flow of instructions as it happens in a real computing environment.

The second part deals with Assembly Language programming of 8088 microprocessor, which helps the students to have a good knowledge of programming of a specific architecture, as well as working in the environments like DEBUG and MASM (Microsoft Macro Assembler).

The last part deals with hardware implementation and interfacing of the 8088 microprocessor with various I/O devices. These labs will assist the students to design and implement a basic microprocessor based system.

Practical Workbook
Microprocessor Programming & Interfacing

CONTENTS

Lab Session 01

OBJECTIVE

Understanding & simulating von Neumann IAS Computer using PRIMA VIRTUAL MACHINE

THEORY

THE VON NEUMANN ARCHITECTURE

In 1947 von Neumann designed the first *stored program* computer at the Institute of Advanced Studies (IAS), Princeton. This machine is therefore regarded as IAS computer. Prior to the notion of stored program computing, machines were programmed by re-arranging physical wiring and every time a new computation was required of a computer, it had to be re-wired again. A stored program computer stores instructions in memory in the same fashion as it stores data for processing. The idea of stored program computing revolutionized the whole computing paradigm and computer architecture proposed by von Neumann became the foundation of every computing machinery to come with no exception of the today's state-of-the-art modern computers.

The IAS computer consists of three major subsystems: instruction processing unit, arithmetic unit, and memory. Instructions and data share the same address space and hence instructions can be processed as data. The word read from the memory is routed to either Instruction Processing Unit or the Arithmetic Unit, depending upon whether an instruction or a datum is being fetched.

THE VON NEUMANN INSTRUCTION SET ARCHITECTURE (ISA)

The von Neumann ISA is quite simple, having only 21 instructions. In fact, this ISA could be called an early reduced instruction set computer (RISC) processor. As with any ISA, there are three components: addresses, data types, and operations.

Addresses

The addresses of an ISA establish the architectural style - the organization of memory and how operands are referenced and results are stored. Being a simple ISA, there are only two memories addressed: the main memory and the accumulator. The main memory of the von Neumann ISA is random access and is equivalent to the dynamic random-access memory (DRAM) found in today's computers. The technology of the 1940s restricted random-access memory (RAM) to very small sizes; thus the memory is addressed by a 12-bit direct address allocated to the 20-bit instructions. Local storage in the processor is a single accumulator. An accumulator register receives results from the ALU that has two inputs, a datum from memory, and the datum held in the accumulator. Thus only a memory address is needed in the instruction as the accumulator is implicitly addressed.

Data Types

The von Neumann ISA has two data types: fractions and instructions. Instructions are considered to be a data type since the instructions can be operated on as data, a feature called self-modifying code. Today, the use of self-modifying code is considered a bad programming practice and thus many operating systems even don't allow execution of such code.

Fractions

The 40-bit word is typed as a 2's complement fraction; the range is $-1 \leq f < +1$:

Instructions

Two 20-bit instructions are allocated to the 40-bit memory word. An 8-bit operation code, or op-code, and a 12-bit address are allocated to each of the instructions. Note that, with only 21 instructions, fewer op-code bits and more address bits could have been allocated. The direct memory address is allocated to the 12 most significant bits (MSBs) of each instruction. The address and the op-code pairs are referred to in terms of left and right:

39 28	27 20	19 8	7 0
Left Address	Left Op-code	Right Address	Right Op-code

Figure 1.1

Registers

A block diagram of the IAS computer is shown in the following figure. (I/O connections are not shown).

The processor has seven registers that support the interpretation of the instructions fetched from memory. Note that two of the registers are explicitly addressed by the instructions and defined in the ISA (called architected registers) while the other six are not defined.

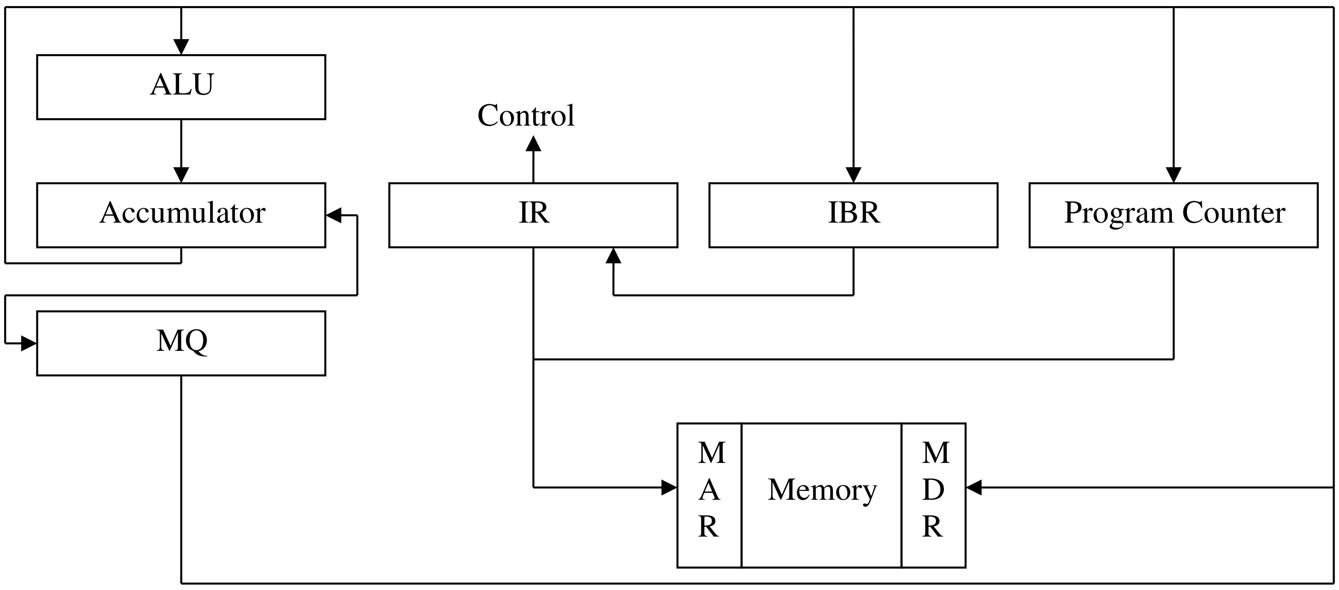


Figure 1.2

MQ: Multiplier Quotient
IR: Instruction Register
IBR: Instruction Buffer Register
MAR: Memory Address Register
MDR: Memory Data Register

The function of each of these registers is described in the following table:

Name	Function
Architected (Programmer-Visible) Registers	
Accumulator, AC, 40 bits	This register holds one of the following: 1. the output of the ALU after an arithmetic operation 2. a datum loaded from memory 3. the most-significant digits of a product and 4. the divisor for division.
Multiplier Quotient Register, MQ, 40 bits	This register holds one of the following: 1. a temporary data value such as the multiplier 2. the least-significant bits of the product as multiplication proceeds and 3. the quotient from division.
Implemented (Programmer-Transparent) Registers	
Program Counter, PC, 12 bits	Holds the pointer to memory. The PC contains the Address of the instruction pair to be fetched next.
Instruction Buffer Register, IBR, 40 bits	Holds the instruction pair when fetched from the Memory.
Instruction Register, IR, 20 bits	Holds the active instruction while it is decoded in the Control unit.
Memory Address Register, MAR, 12 bits	Holds the memory address while the memory is being read or written. The MAR receives input from the program counter for an instruction fetch and from the address field of an instruction for a datum read or write.
Memory Data Register, MDR, 40 bits	Holds the datum (instruction or data) for a memory read or write cycle.

Operations

The operations of the von Neumann ISA are of three types:

- data transfer between the accumulator, multiplier quotient register, and memory
- ALU operations such as add, subtract, multiply, and divide
- Unconditional and conditional branch instructions that redirect program flow.

INTRODUCTION TO PRIMA- SIMULATOR FOR VON-NEUMANN COMPUTER

The screen shown in figure 1.3 appears when PRIMA simulator is run.

- **The speed controller:** You can control the speed of the animation with this scrollbar.
- **The "clock" button:** This button simulates a clock signal to the PRIMA. While the animation is running, the applet will not react to user input.
- **The "start" button:** This button runs the program that is in the RAM. You first have to load a program into the RAM to use this function. (See "the edit button"). Unlike the clock button, this function has its own *thread* and user input will be processed while the program runs.

- **The "reset" button:** Reset the PRIMA. The RAM will not be erased by this function, but the PRIMA returns to the state it was in before the program was started.
- **The "edit" button:** You can invoke the "Edit RAM" window with this button. This window is used for loading program examples into the RAM and viewing the source code. You can also write your own programs in this window. See "Edit RAM" for details on using the Edit window.
- **The "command" button:** This button invokes the "Command" window. This window shows the current command (instruction) that is being executed by the PRIMA and explains what it does. For details, see "The Command window".
- **The radix menu:** You can switch between number representations with radix 10(decimal) or radix16 (Hexadecimal) with this menu.

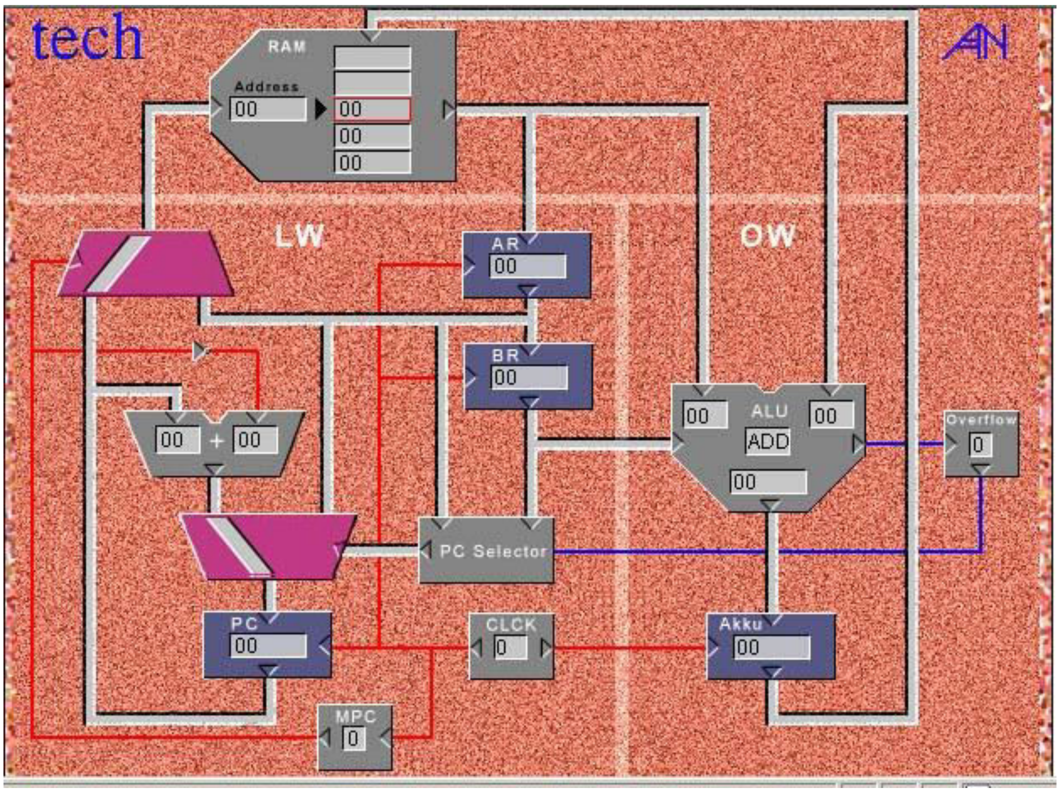


Figure 1.3

THE PRIMA WINDOWS

The main window: In this window, you see the building blocks of the PRIMA and the control panel at the bottom of the window. You see an image of the PRIMA, explaining the different elements:

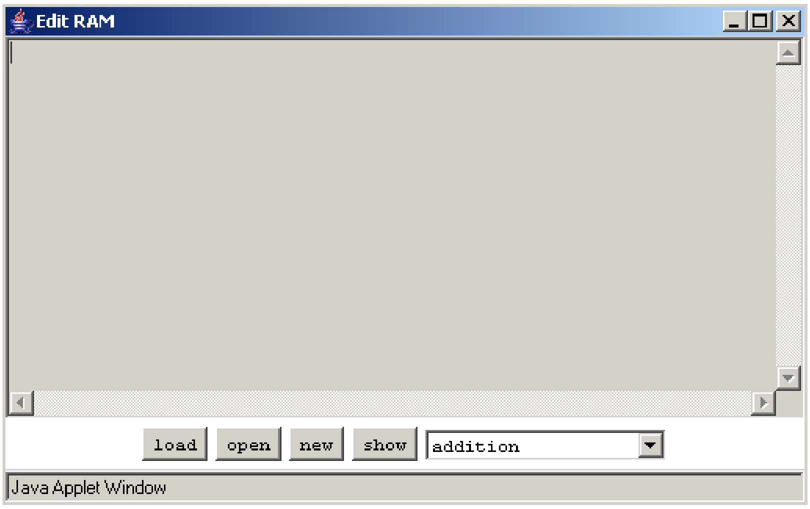
Building Blocks of PRIMA:

1. **RAM Block:** The left field contains the current address. The field with the red border contains the *Value i.e. contents* at the current address.
2. **Upper MULTIPLEXER Block:** The Multiplexer switches between the PC and Address Register.
3. **ADDER Block:** The adder adds two numbers as its inputs. Here it is used to increment the PC.

4. **Lower MULTIPLEXER Block:** The Multiplexer switches between the adder and Address Register.
5. **PROGRAM COUNTER Block**: The Program Counter stores the address of next instruction to be fetched.
6. **MPC Block:** The MPC stores the mode of PRIMA which are: 1= apply 0= load
7. **CLOCK Block**: The Clock generates the clock signals for PRIMA.
8. **PC SELECT Block:** The PC Select block contains the logic to handle branches.
9. **BR Block:** It is the command register which contains the command that is applied in the *apply mode*.
10. **AR Block:** It is the Address Register which contains the address that is applied in the *apply mode*.
11. **ALU Block:** The ALU does the arithmetic and logical operations. The field at the centre specifies the current operation being performed in ALU.
12. **OVERFLOW Block:** It stores the Overflow flag.
13. **AKKU Block:** It is the accumulator which stores the output of ALU for further use.

EDIT RAM

Here is a screenshot of the "Edit RAM" window:



Following are the main components:

The control panel

- **The "load" button:** You can load a program into the RAM with this button. To do this you have to choose a program from the choice or type in your own program.
- **The "open" button:** You can view the program of your choice with this button. First you have to choose a program from the menu. This function does not load the program into the RAM. You still have to use the load button to do this.
- **The "new" button:** If you want to write your own program, use this button. The text editor will be cleared and you can type in your program. The commands are given in decimal numbers, followed by the address on which they should act. Command list will be studied in the next lab. The commands and addresses have to be separated by at least one space. After typing in the program, you can load it into the RAM with the load button. To

verify your code, you can use the show button. This button will show the code in the RAM and comment it.

- **The "show" button:** This button shows the current program in the RAM. If the programs save their results in variables, you can use this function to see the values of the variables after program execution.
- **The program examples menu:** You can choose one of the program examples here. To view or load the program, you have to use the open or the load button.
- **The "Command" window:** This window shows the current command that is being executed by the PRIMA, and explains what it does.

PROCEDURE

1. Open HotJava Browser application first, and then load INDEX.html file present in the PRIMA folder. The complete simulator is available locally and no internet connection is required.
2. By default, addition operation is loaded into the simulator. So just run the simulator from the main screen by pressing START.
3. Before starting, note down that in the list box, DEC is loaded which implies that all the register contents, addresses and memory contents will be displayed in decimal.
4. Carefully note down the first simulation and fill in the observation chart # 1 after stopping simulation.
5. Press RESET button. Then Change the DEC to HEX in list box, which implies that all the register contents, addresses and memory contents will now be displayed in hexadecimal.
6. Now carefully note down the second simulation and fill in observation chart # 2 after stopping simulation

OBSERVATIONS

(1) SIMULATION # 1 (In Decimal): (for first 6 clock cycles)

PRIMA BUILDING BLOCKS	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
1.RAM Contents						
2.Address Register (AR)						
3. Command Register(BR)						
4.Program Counter(PC)						
5.Clock(CLK)						
6.ALU operation						
7.Overflow flag						
8.Accumulator(AKKU)						
9.MPC(Mode of PRIMA)						

(1) SIMULATION # 1 (In Hexadecimal): (for first 6 clock cycles)

PRIMA BUILDING BLOCKS	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
1.RAM Contents						
2.Address Register (AR)						
3. Command Register(BR)						
4.Program Counter(PC)						
5.Clock(CLK)						
6.ALU operation						
7.Overflow flag						
8.Accumulator(AKKU)						
9.MPC(Mode of PRIMA)						

EXERCISES

1. From the GUI (Graphical User Interface) of PRIMA, Identify the three major subsystems of Von Neumann Computer.

____
____
____
____
____

2. One of the unique points of the von Neumann architecture is that both instructions and data share the same address space. Justify this statement from PRIMA simulation.

____
____
____
____
____

3. Name the seven registers of von Neumann processor that support the interpretation of the instructions fetched from memory.

a. ______________________
b. ______________________
c. ______________________
d. ______________________
e. ______________________
f. ______________________
g. ______________________

4. Which of the IAS registers contains the output of ALU after completion of an arithmetic operation? What is the *width* of this register in bits? Verify from PRIMA simulator interface.

5. Specify whether the instructions of von Neumann computer are processed as data. Justify from PRIMA simulation and output of loaded program.

6. Which register holds the address of memory to be read or written? What is the *width* of this register in bits?

7. Which register holds the instruction pair when fetched from a 40-bit word sized memory? Load the Program Addition and verify your answer from the PRIMA simulator GUI based Animation.

8. Which number format is used to represent negative numbers in von Neumann machine?

Lab Session 02

OBJECTIVE

Simulating Instruction Set of von Neumann Machine using PRIMA VIRTUAL MACHINE (PVM)

THEORY

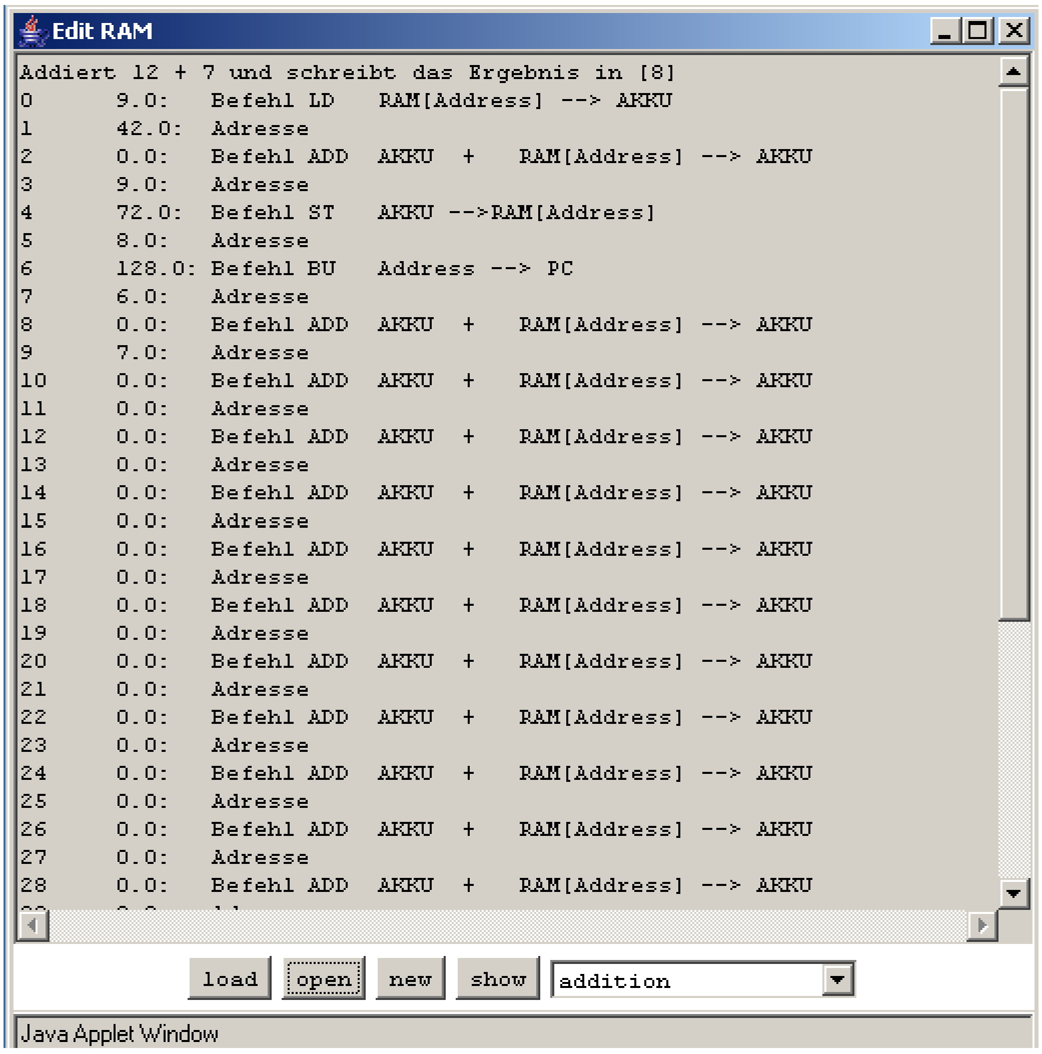
All instructions are constructed from two sequential bytes. The first byte is the opcode (i.e. the operation to be performed), and the second is the address of the operand upon which the operation is to be performed.
The overflow flag OV can be reset before applying the instruction by adding a "*" to mnemonic of the instruction.

Mnemonic	Decimal	Binary	Action
ADD	0	00000000	Accu + RAM[Address] $\rightarrow$ Accu
ADD*	32	00100000	
SUB	1	00000001	Accu - RAM[Address] $\rightarrow$ Accu
SUB*	33	00100001	
AD1	10	00001010	Accu + 1 $\rightarrow$ Accu
AD1*	42	00101010	
SB1	12	00001100	Accu - 1 $\rightarrow$ Accu
SB1*	44	00101100	
OR	2	00000010	Accu OR RAM[Address] $\rightarrow$Accu
OR*	34	00100010	
AND	3	00000011	Accu AND RAM[Address] $\rightarrow$Accu
AND*	35	00100011	
XOR	4	00000100	Accu XOR RAM[Address] $\rightarrow$Accu
XOR*	36	00100100	
NOP	8	00001000	Accu $\rightarrow$ Accu (Does nothing)
NOP*	40	00101000	
LD	9	00001001	RAM[Address] $\rightarrow$ Accu
LD*	41	00101001	
LDI	11	00001011	RAM[Address] + 1 $\rightarrow$ Accu
LDI*	43	00101011	
LD0	14	00001110	0 $\rightarrow$ Accu
LD0*	46	00101110	
LD1	15	00001111	1 $\rightarrow$ Accu
LD1*	47	00101111	

ST	72	01001000	Accu → RAM[Address]
ST*	104	01101000	
SL	5	00000101	Accu[i] → Accu[i+1], (0 <= i < 8); 0 → Accu[0] "LEFT SHIFT"
SL*	37	00100101	
SR	6	00000110	Accu[i] → Accu[i-1], (0 <= i < 8); 0 → Accu[8] "RIGHT SHIFT"
SR*	38	00100110	
RR	7	00000111	Accu[i] → Accu[i-1], (0 < i <= 8); Accu[0] → Accu[8] "ROATE RIGHT"
RR*	39	00100111	
BU	128	1X0XXXX0	Address → PC
BU*	160	1X1XXXX0	
BZ	131	10000011	IF Accu=0 THEN Address → PC
BZ*	163	10100011	
BCY	133	10000101	IF Accu[8]=1 THEN Address → PC
BCY*	165	10100101	
BEV	193	11000001	IF Accu[0]=1 THEN Address → PC
BEV*	225	11100001	
BLS	137	10001001	IF Accu[7]=1 THEN Address → PC
BLS*	169	10101001	
BOV			Exists only as BOV*
BOV*	161	10100001	IF OV=1 THEN Address → PC
BSW	145	10010001	IF SW=1 THEN Address → PC
BSW*	177	10110001	

PROCEDURE

1. Open HotJava Browser application first, and then load INDEX.html file present in the PRIMA folder.
2. After the applet starts, you will see the block diagram of the PVM. There are 7 UI elements at the bottom of the screen.
3. Press the edit button. A window will pop up called "Edit RAM”.
4. Take a look at the example program "addition". This is the default choice.
5. Press the open button to look at the program. After a short delay, the program code and its explanation in the window will appear.
6. Load the program into the RAM by pressing the “load “button.
7. Press the start button in the main window. The program will start and you will see an animation of the data flow.
8. You can control the speed of the animation with the “speed control”.
9. Pressing the command button will pop up a window that shows the currently executing instruction.



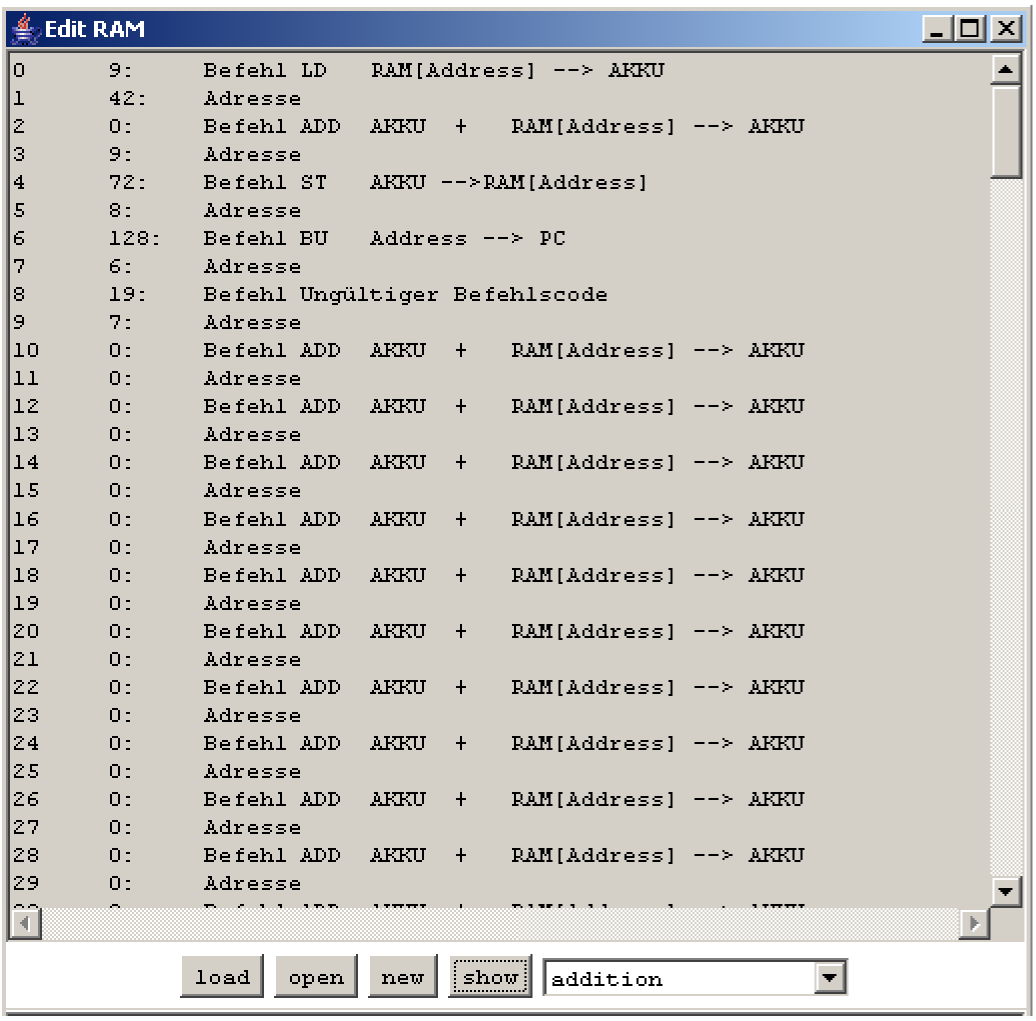
```
Addiert 12 + 7 und schreibt das Ergebnis in [8]
0       9.0:    Befehl LD   RAM[Address] --> AKKU
1       42.0:   Adresse
2       0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
3       9.0:    Adresse
4       72.0:   Befehl ST   AKKU -->RAM[Address]
5       8.0:    Adresse
6       128.0:  Befehl BU   Address --> PC
7       6.0:    Adresse
8       0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
9       7.0:    Adresse
10      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
11      0.0:    Adresse
12      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
13      0.0:    Adresse
14      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
15      0.0:    Adresse
16      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
17      0.0:    Adresse
18      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
19      0.0:    Adresse
20      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
21      0.0:    Adresse
22      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
23      0.0:    Adresse
24      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
25      0.0:    Adresse
26      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
27      0.0:    Adresse
28      0.0:    Befehl ADD  AKKU  +   RAM[Address] --> AKKU
```

Fig. 2.1

OBSERVATIONS

(For program given in figure 2.1)

1. The program runs in an infinite loop.
2. You can recognize this by the repeating values and addresses in the RAM. Now we can stop the program and view the results in the "Edit RAM" window. We expect to see the result of the addition 12 + 7 at the RAM address 8.
3. To show the contents of the RAM, press the show button.
4. Check the value at address 8. As expected, the value is 19.
5. To write your own program, press the "new" button. You can type in your program now. The instructions must be decimal numbers with a valid “Opcode”. The addresses must be smaller than 256.



```
0       9:     Befehl LD    RAM[Address] --> AKKU
1       42:    Adresse
2       0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
3       9:     Adresse
4       72:    Befehl ST    AKKU -->RAM[Address]
5       8:     Adresse
6       128:   Befehl BU    Address --> PC
7       6:     Adresse
8       19:    Befehl Ungültiger Befehlscode
9       7:     Adresse
10      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
11      0:     Adresse
12      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
13      0:     Adresse
14      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
15      0:     Adresse
16      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
17      0:     Adresse
18      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
19      0:     Adresse
20      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
21      0:     Adresse
22      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
23      0:     Adresse
24      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
25      0:     Adresse
26      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
27      0:     Adresse
28      0:     Befehl ADD   AKKU  +   RAM[Address] --> AKKU
29      0:     Adresse
```

EXERCISES

1. Write a program to add two numbers. Please type in the following program:

9 8 0 9 72 10 128 6 14 2

Description of the program logic is as follows:

a. Here 9 is opcode of LD instruction.
b. Number 14 is stored in accumulator at $8. ($8=14)
c. 0 is the opcode of ADD instruction.
d. Number 2 is stored in accumulator at $9. ($9=2)
e. It then saves (72), the value of the accumulator at address $10.

Program goes in an infinite loop .We can load the program into the RAM by pressing the load button. Look at the contents of the RAM by pressing the show button. We execute the program by pressing the start button. Once the program goes into an infinite loop, we can stop the program and view the contents of the RAM. As expected, the address 10 contains the value 16.

Write down the source code and corresponding flow chart of the program here:

Program **Flow Chart**

Lab Session 03

OBJECTIVE

Implementing Conditional Branch Instructions of von Neumann Machine using PRIMA

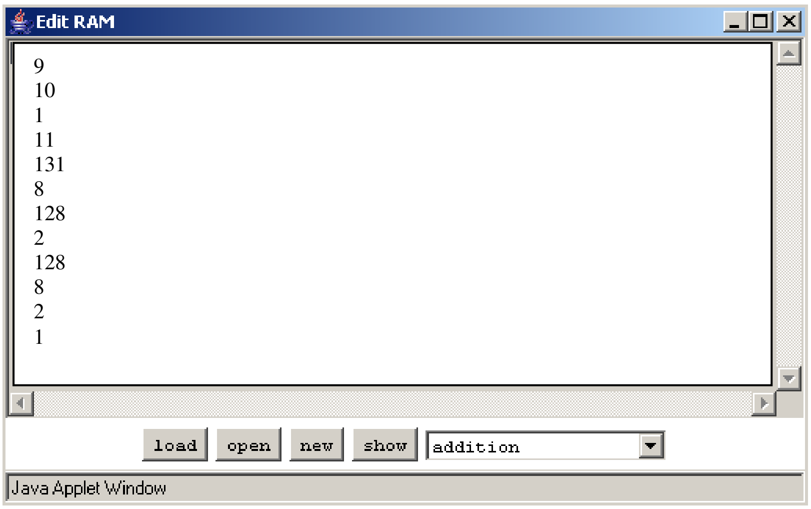
THEORY

Following sequence of instructions illustrates use of *conditional branch* in von-Neumann machine:

OPCODE/ADDRESS		**ACTION PERFORMED**	
9:	opcode	LD	RAM [Address] → AKKU
10:	Address		
1:	opcode	SUB	AKKU - RAM [Address] → AKKU
11:	Address		
131:	opcode	BZ	IF AKKU = 0 THEN Address → PC
8:	Address		
128:	opcode	BU	Address → PC
2:	Address		
128:	opcode	BU	Address → PC
8:	Address		
2:	opcode	Operand	
1:	Address	Operand	

PROCEDURE

1. Execute PVM as described in the previous lab sessions.
2. Press **edit** button to "Edit RAM" window.
3. Press the **new** button and start writing your program in this new window. (See figure 3.1)



```
9
10
1
11
131
8
128
2
128
8
2
1
```

Figure 3.1

4. Load the program into RAM by pressing the **load** button.
5. Press the start button in the main window. The program will start and you will see an animation of the data flow.
6. You can control the speed of the animation with the “speed control”.
7. Pressing the command button will pop up a window that shows the currently executing instruction.
8. The program runs in an infinite loop.
9. You can recognize this by the repeating values and addresses in the RAM. Now we can stop the program and view the results in the "Edit RAM" window.
10. To show the contents of the RAM, press the show button.
11. After stopping the program, go to the main PRIMA applet screen.
12. Note down the simulation output on PRIMA screen in the given observation table.

OBSERVATIONS

PRIMA BUILDING BLOCKS	**READINGS**											
1.RAM Contents												
2.Address Register (AR)												
3. Command Register(BR)												
4.Program Counter(PC)												
5.Clock(CLK)												
6.ALU operation												
7.Overflow flag												
8.Accumulator(AKKU)												
9.MPC(Mode of PRIMA)												

EXERCISES

1. Which specific instruction enables you to perform Indexed addressing in von Neumann IAS Computer? What is the opcode of that instruction?

2. Do we need a RESET in von Neumann IAS computer, when we execute programs? Verify from PVM simulator and give reason for your answer.

3. Which Conditional Branch Instruction of IAS instruction set was used in the given program? Write down its opcode in binary.

____

4. Suppose that during execution of conditional branch instruction, condition is found to be TRUE also known as a *taken branch*. Which register will hold the branch address when branch or transfer of control occurs? Verify your answer from PVM Simulator graphical user interface.

Register: ____________________________
Contents of Register (Before Branch): ______________________________
Contents of Register (After Branch): ______________________________

5. At which memory address, the control will be transferred, if the condition is found to be TRUE, during execution of Conditional Branch Instruction in the given program?

____

6. At which memory address, the control will be transferred, if the condition is found to be FALSE, also known as *not taken branch* during execution of Conditional Branch instruction in the given program?

____

7. Which specific register of von Neumann IAS computer contains the address of the next Instruction in a given program sequence? What is the size of this register in bits?

____
____
____
____

8. Write down the opcode of the Unconditional Branch instruction of von Neumann IAS Computer “BU”. How BU command works?

____
____
____
____

Lab Session 04

OBJECTIVE

Executing Self-Modifying Instructions on von Neumann Machine using PRIMA

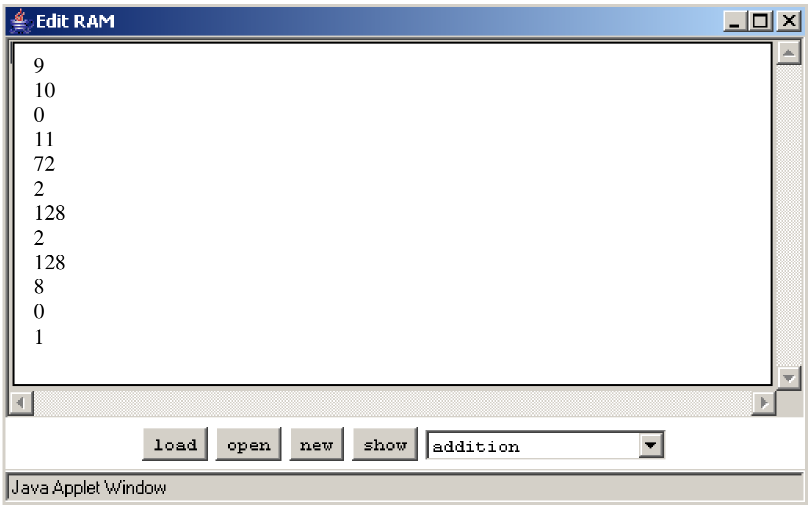
THEORY

Following sequence of instructions illustrates use of self-modifying instructions in von-Neumann machine:

OPCODE/ADDRESS		ACTION PERFORMED	
9:	opcode	LD	RAM [Address] → AKKU
10:	Address		
0:	opcode	ADD	AKKU + RAM [Address] → AKKU
11:	Address		
72:	opcode	ST	AKKU → RAM [Address]
2:	Address		
128:	opcode	BU	Address → PC
2:	Address		
128:	opcode	BU	Address → PC
8:	Address		
0:	opcode	Operand	
1:	Address	Operand	

PROCEDURE

1. Execute PVM as described in the previous lab sessions.
2. Press **edit** button to "Edit RAM" window.
3. Press the **new** button and start writing your program in this new window. (See figure 4.1)



```
9
10
0
11
72
2
128
2
128
8
0
1
```

Figure 4.1

4. Load the program into RAM by pressing the **load** button.
5. Press the start button in the main window. The program will start and you will see an animation of the data flow.
6. You can control the speed of the animation with the "speed control".
7. Pressing the command button will pop up a window that shows the currently executing instruction.
8. The program runs in an infinite loop.
9. You can recognize this by the repeating values and addresses in the RAM. Now we can stop the program and view the results in the "Edit RAM" window.
10. To show the contents of the RAM, press the show button.
11. After stopping the program, go to the main PRIMA applet screen.
12. Note down the simulation output on PRIMA screen in the given observation table.

OBSERVATIONS

PRIMA BUILDING BLOCKS	**READINGS**											
1.RAM Contents												
2.Address Register (AR)												
3. Command Register(BR)												
4.Program Counter(PC)												
5.Clock(CLK)												
6.ALU operation												
7.Overflow flag												
8.Accumulator(AKKU)												
9.MPC(Mode of PRIMA)												

EXERCISES

1. Which specific instruction enables you to perform Indexed addressing in von Neumann IAS Computer? What is the opcode of that instruction?

____

____

____

____

____

2. Which Unconditional Branch Instruction of IAS instruction set was used in the given program? Write down its opcode in binary.

____

Lab Session 05

OBJECTIVE

Exploring the Instruction Set Architecture (ISA) of 8088 microprocessor

THEORY

INSTRUCTION SET ARCHITECTURE

The collection of all the operations possible in a machine's language is its *instruction set*. The programmer's view is composed of the machine / assembly language instruction set of the machine, along with the machine resources that can be managed with those instructions. This collection of instructions and resources is sometimes referred to as *instruction set architecture* (ISA) of the machine. The ISA includes the instruction set, the machine's memory, and all the programmer-accessible registers in the CPU and elsewhere in the machine.

ISA of 8088 microprocessor-based computer

- **The Processor**

  The processor is partitioned into two *logical* units: an **Execution Unit (EU)** and a **Bus Interface Unit (BIU)**. The role of the EU is to execute instructions, whereas the BIU delivers instructions and data to the EU. The EU contains ALU, a control unit and a number of registers. These features provide for execution of instructions and arithmetic and logic operations. The BIU controls the buses that transfer data to the EU, to memory, and to I/O devices. It also manages *segment registers* and *instruction queue*. Segment registers control memory addressing and will be described shortly.

  Instructions fetched from main memory by BIU are placed in an instruction queue, which varies in size depending on the processor. This feature enables fetching of instructions in parallel with instruction execution and hence results in speeding up execution.

- **Memory Addressing**

  Depending on the machine, a processor can access one or more bytes from memory at a time. The number of bytes accessed simultaneously from main memory is called *word length* of machine.

  Generally, all machines are *byte-addressable* i.e.; every byte stored in memory has a unique address. However, word length of a machine is typically some integral multiple of a byte. Therefore, the address of a word must be the address of one of its constituting bytes. In this regard, one of the following methods of addressing may be used.

  ‣ **Big Endian**: The word address is taken as the address of the most significant byte in the word. MIPS, Apple Macintosh are some of the machines in this class.

  ‣ **Little Endian**: Here the word address is taken as the address of the least significant byte in the word. Intel's machines are of this type. Consider for example, storing hex number 245A in main memory. The least significant byte 5A will be stored in low memory address and most significant byte 24 will be stored in high memory address.

Address	Contents
1002	5A
1003	24

- **Segments**

A segment is an area *defined* in a program that begins on a paragraph boundary, that is, an address evenly divisible by 16, or hex 10. To address an item in a segment, the starting address of the segment is provided by a special purpose *segment register*. The three major segments in a program are the following.

‣ **Code Segment**: It contains the machine instructions that are to execute.
‣ **Data Segment:** This contains a program's defined data and constants.
‣ **Stack Segment:** This contains any data and addresses that you need to save temporarily or for use by your own "called" subroutines.

- **Segment Boundaries**

A segment register contains the starting address of a segment. Segment registers associated with code, data and stack segments are respectively CS, DS and SS registers. Other segment registers are the ES (extra segment) and, on the 80386 and later processors, the FS and GS registers, which have specialized uses.

A segment begins on a paragraph boundary, which is an address evenly divisible by hex 10. Consider a code segment that begins at an address 038E0H. Because in this and all other cases the rightmost hex digit is zero, the computer designers decided that it would be unnecessary to store the zero digit in the segment register Thus 038E0H is stored as 038E, with the rightmost zero understood.

- **Segment Offsets**

Within a program, all items (instructions and data) are addressed relative to the starting address of the segment in which the items appear. The distance in bytes from the segment's starting address to another location within the segment is expressed as an *offset* (or displacement). A 2-byte (16-bit) offset can range from 0000H through FFFFH, or zero through 65, 535. To reference any memory location in a segment, the processor combines the segment address in a segment register with an offset value.

Consider for example that a data segment begins at location 038E0H. The DS register contains the segment address of the data segment, 038EH, and an instruction references a location with an offset of 0032H bytes within the data segment. The actual memory address of the byte referenced by the instruction is therefore:

```
DS Segment Address:          038E0H
Offset:                      +0032H
Actual Address:              03912H
```

A program contains one or more segments, which may begin almost anywhere in memory, may vary in size, and may be in any sequence.

- **Registers**

The processor's registers are used to control instruction execution, to handle addressing of memory, and to provide arithmetic capability. The registers are addressable by name, such as CS, DS, and SS. Bits in a register are conventionally numbered from right to left, beginning with 0, as

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Some of the registers of interest are as follows.

▸ **Pointer Registers**

✓ ***Instruction Pointer (IP) Register:*** The 16-bit IP register contains the offset address of the next instruction that is to execute. The IP is associated with the CS register in that the IP indicates the current instruction within the currently executing code segment.

✓ ***Stack Pointer (SP) Register:*** The 16-bit SP register provides an offset, which when associated with the SS register, refers to the current word being processed in the stack. The 80386 and later processors have an extended 32-bit stack pointer, the ESP register.

✓ ***Base Pointer (BP) Register:*** The 16-bit BP facilitates referencing parameters, which are data and addresses that a program passes via the stack. The processor combines the address in SS register with the offset in the BP. The 80386 and later processors have an extended 32-bit BP called EBP register.

▸ **General-Purpose Registers**

✓ ***AX Register:*** The AX register, the accumulator, is used for operations involving I/O and most arithmetic. Some instructions generate more efficient code if they reference the AX register rather than other registers.

Like other general-purpose registers, it is a 16-bit register that can be accessed byte-wise. The high byte is AH and low byte is AL.

AX

AH	AL
8 bits	8 bits

The 80386 and later processors support all the general-purpose registers, plus 32-bit extended versions of them: EAX, EBX, ECX, and EDX.

✓ ***BX Register:*** The BX is known as base register since it is the only general-purpose register that can be used as an index to extended addressing.

✓ ***CX Register:*** The CX is known as count register. It may contain a value to control the number of times a loop is repeated or a value to shift bits left or right.

- ***DX Register:*** The DX is also known as data register. Some I/O operations require its use, and multiply and divide operations that involve large values assume the use of the DX and AX together as pair.

Index Registers

- ***SI Register:*** The 16-bit source index register is required for some string (character) operations. In this context, the SI is associated with the DS register. The 80386 and later processors support a 32-bit extended register, the ESI.

- ***DI Register:*** The 16-bit destination index register is also required for some string (character) operations. In this context, the DI is associated with the ES register. The 80386 and later processors support a 32-bit extended register, the EDI.

Flag Register

Nine of the 16 bits of the flag register are common to all 8086- family processors to indicate the current status of the processor and the results of processing. Many instructions involving comparisons and arithmetic change the status of the flags, which some instructions may test to determine subsequent action.

The following briefly describes the common flag bits:

- ***OF*** (overflow): Indicates overflow resulting from some arithmetic operation.
- ***DF*** (direction): Determines left or right direction for moving or comparing string (character) data.
- ***IF*** (interrupt): Indicates that all external interrupts, such as keyboard entry, are to be processed or ignored.
- ***TF*** (trap): Permits operation of the processor in single-step mode. Debugger programs such as DEBUG set the trap flag so that you can step through execution in a single-instruction at a time to examine the effects on registers and memory.
- ***SF*** (sign): Contains the resulting sign of an arithmetic operation (0 = positive and 1 = negative).
- ***ZF*** (zero): Indicates the result of an arithmetic or comparison operation (0 = nonzero and 1 = zero result)
- ***AF*** (auxiliary carry):Contains a carry out of bit 3 on 8–bit data, for specialized arithmetic.
- ***PF*** (parity): Indicates even or odd parity of a low-order (rightmost) 8-bit data operation.
- ***CF*** (carry): Contains carry from a high-order (leftmost) bit following an arithmetic operation; also, contains the contents of the last bit of a shift or rotate operation.

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The 80286 and later processors have a 32-bit *extended* flags register known as *E*flags.

EXERCISES

a) Show how the following values are stored in memory beginning at address 7123.

i) 1234H

Address	Contents
7123	
7124	

ii) 01DB5CH

Address	Contents
7123	
7124	
7125	

b) What are (i) the three kinds of segments, (ii) their maximum size, and (iii) the address boundary on which they begin?

c) Explain which registers are used for the following purposes: (i) addressing segments; (ii) offset address of an instruction that is to execute; (iii) addition and subtraction; (iv) multiplication and division; (v) counting for looping; (vi) indication of a zero result.

d) Show the EDX register and the size and position of the DH, DL, and DX within it.

e) During execution of a program, the CS contains 6C3AH, the SS contains 6C62H, the IP contains 42H, and the SP contains 36H. Calculate the addresses of (i) the instruction to execute and (ii) the top (current location) of the stack.

Lab Session 06

OBJECTIVE

Programming in Assembly Language Programming of 8088 microprocessor

THEORY

ASSEMBLY LANGUAGE SYNTAX

name operation operand (s) comment

Assembly language statement is classified in two types

Instruction

Assembler translates into machine code.

Example:

```
        START:      MOV   CX, 5   ; initialize counter
```

Comparing with the syntax of the Assembly statement, name field consists of the label START:. The operation is MOV, operands are CX and 5 and the comment is ;initialize counter.

Assembler Directive

Instructs the assembler to perform some specific task, and are not converted into machine code.

Example:

```
        MAIN        PROC
```

MAIN is the name, and operation field contains PROC. This particular directive creates a procedure called MAIN.

Name field

Assembler translate name into memory addresses. It can be 31 characters long.

Operation field

It contains symbolic operation code (opcode). The assembler translates symbolic opcode into machine language opcode. In assembler directive, the operation field contains a pseudo-operation code (pseudo-op). Pseudo-op are not translated into machine code, rather they simply tell the assembler to do something.

Operand field

It specifies the data that are to be acted on by the operation. An instruction may have a zero, one or two operands.

Comment field

A semicolon marks the beginning of a comment. Good programming practice dictates comment on every line.

Examples:	MOVCX, 0	;move 0 to CX
	Do not say something obvious; so:	
	MOV CX, 0	;CX counts terms, initially 0

Put instruction in context of program
; initialize registers

DATA REPRESENTATION

Numbers

11011	decimal
11011B	binary
64223	decimal
-21843D	decimal
1,234	illegal, contains a non-digit character
1B4DH	hexadecimal number
1B4D	illegal hex number, does not end with
FFFFH	illegal hex number, does not begin with digit
OFFFFH	hexadecimal number

Signed numbers represented using 2's complement.

Characters

- Must be enclosed in single or double quotes, e.g. "Hello", 'Hello', "A", 'B'
- encoded by ASCII code
  - 'A' has ASCII code 41H
  - 'a' has ASCII code 61H
  - '0' has ASCII code 30H
  - Line feed has ASCII code OAH
  - Carriage Return has ASCII code
  - Back Space has ASCII code 08H
  - Horizontal tab has ASCII code 09H

VARIABLE DECLARATION

Each variable has a type and assigned a memory address.
Data-defining pseudo-ops

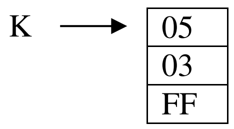
DB	define byte
DW	define word
DD	define double word (two consecutive words)
DQ	define quad word (four consecutive words)
DT	define ten bytes (five consecutive words)

Each pseudo-op can be used to define one or more data items of given type.

Byte Variables

Assembler directive format assigning a byte variable
```
Name    DB    initial value
```
A question mark (“?”) place in initial value leaves variable uninitialized

```
I        DB    4           ;define variable I with initial value 4
J        DB    ?           ;Define variable J with uninitialized value
Name     DB    "Course"    ;allocate 6 bytes for name
K        DB    5, 3,-1     ;allocate 3 bytes
```



Other data type variables have the same format for defining the variables.
Like:
```
Name          DW    initial value
```

NAMED CONSTANTS

- EQU pseudo-op used to assign a name to constant.
- Makes assembly language easier to understand.
- No memory allocated for EQU names.

```
LF      EQU        0AH
      o  MOV             DL, 0AH
      o  MOV             DL, LF

PROMPT    EQU       "Type your name"
      o  MSG       DB     “Type your name”
      o  MDC       DB     PROMPT
```

INPUT AND OUTPUT USING DOS ROUTINES

CPU communicates with peripherals through I/O registers called I/O ports. Two instructions access I/O ports directly: IN and OUT. These are used when fast I/O is essential, e.g. games.

Most programs do not use IN/OUT instructions. Since port addresses vary among computer models and it is much easier to program I/O with service routines provided by manufacturer.

Two categories of I/O service routines are Basic input & output system (BIOS) routines and Disk operating system (DOS) routines. Both DOS and BIOS routines are invoked by INT (interrupt) instruction.

Disk operating system (DOS) routines

INT 21 H is used to invoke a large number of DOS function. The type of called function is specified by pulling a number in AH register.

For example

AH=1	input with echo
AH=2	single-character output
AH=9	character string output
AH=8	single-key input without echo
AH=0Ah	character string input

Single-Key Input

Input: AH=1
Output: AL= ASCII code if character key is pressed, otherwise 0.

To input character with echo:

```
MOV     AH, 1
INT     21H             read character will be in AL register
```

To input a character without echo:

```
MOV     AH, 8
INT     21H             read character will be in AL register
```

Single-Character Output

```
Input:   AH=2,
         DL= ASCII code of character to be output
Output:  AL=ASCII code of character
```

To display a character

```
MOV     AH, 2
MOV     DL, '?'
INT     21H             displaying character'?'
```

Combining it together:

```
MOV     AH, 1
INT     21H
MOV     AH, 2
MOV     DL, AL
INT     21H             read a character and display it
```

To Display a String

Input: AH=9,
DX= offset address of a string.
String must end with a '$' character.

To display the message Hello!

```
MSG     DB      "Hello!"
MOV     AH, 9
MOV     DX, offset MSG
INT     2IH
```

OFFSET operator returns the address of a variable The instruction LEA (load effective address) loads destination with address of source
LEA DX, MSG

PROGRAM STRUCTURE

Machine language programs consist of code, data and stack. Each part occupies a memory segment. Each program segment is translated into a memory segment by the assembler.

Memory models

The size of code and data a program can have is determined by specifying a memory model using the .MODEL directive. The format is:

```
.MODEL   memory-model
```

Unless there is lot of code or data, the appropriate model is SMALL

memory-model	description
SMALL	One code-segment. One data-segment.
MEDIUM	More than one code-segment. One data-segment. Thus code may be greater than 64K
COMPACT	One code-segment. More than one data-segment.
LARGE	More than one code-segment. More than one data-segment. No array larger than 64K.
HUGE	More than one code-segment. More than one data-segment. Arrays may be larger than 64K.

Data segment
A program's DATA SEGMENT contains all the variable definitions.
To declare a data segment, we use the directive .DATA, followed by variable and constants declarations.

```
.DATA
WORD1           DW          2
MASK            EQU         10010010B
```

Stack segment
It sets aside a block of memory for storing the stack contents.

```
.STACK          100H        ;this reserves 256 bytes for the stack
```

If size is omitted then by-default size is 1KB.

Code segment
Contain program's instructions.

```
.CODE           name
```

Where name is the optional name of the segment
There is no need for a name in a SMALL program, because the assembler will generate an error). Inside a code segment, instructions are organised as procedures. The simplest procedure definition is

```
name      PROC
;body of message
name      ENDP
```

An example

```
MAIN PROC
;main procedure instructions
MAIN ENDP
;other procedures go here
```

Putting it together

```
.MODEL          SMALL
.STACK          100H
.DATA
;data definition go here
.CODE
MAIN      PROC
;instructions go here
MAIN      ENDP
```

```
;other procedures go here
END                MAIN
```

The last line in the program should be the END directive followed by name of the main procedure.

A Case Conversion Program

Prompt the user to enter a lowercase letter, and on next line displays another message with letter in uppercase, as:
Enter a lowercase letter: a
In upper case it is: A

```
TITLE PGM4_1:  CASE CONVERSION PROGRAM
.MODEL  SMALL
.STACK   100H
.DATA
        CR          EQU         0DH
        LF          EQU         0AH
        MSG1        DB          'ENTER A LOWER CASE LETTER:  $'
        MSG2        DB          CR, LF, 'IN UPPER CASE IT IS:  '
        CHAR        DB          ?,'$'
.CODE
MAIN PROC
;initialize DS
        MOV         AX,@DATA    ; get data segment
        MOV         DS,AX       ; initialize DS
;print user prompt
        LEA         DX,MSG1     ; get first message
        MOV         AH,9        ; display string function
        INT         21H         ; display first message
;input a character and convert to upper case
        MOV         AH,1        ; read character function
        INT         21H         ; read a small letter into AL
        SUB         AL,20H      ; convert it to upper case
        MOV         CHAR,AL     ; and store it
;display on the next line
        LEA         DX,MSG2     ; get second message
        MOV         AH,9        ; display string function
        INT         21H         ; display message and upper case letter in front
;DOS exit
        MOV         AH,4CH      ; DOS exit
        INT         21H
MAIN ENDP
        END         MAIN
```

Save your program with (.asm) extension.
If “**first**” is the name of program then save it as “**first.asm**”

Lab Session 07

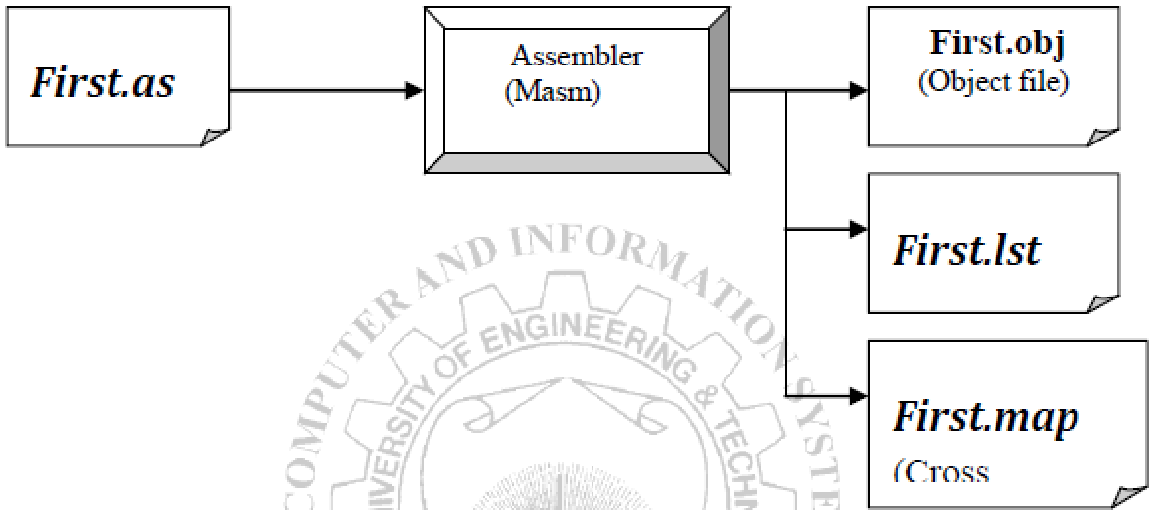
OBJECTIVE

Running an Assembly language program of 8088 microprocessor using the DEBUG tool

THEORY

ASSEMBLING THE PROGRAM

Assembling is the process of converting the assembly language source program into machine language object file. The program "ASSEMBLER" does this.



Assemble the program

```
C:\>masm first.asm
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.
Object filename [first.OBJ]: first
Source listing [NUL.LST]: first
Cross-reference [NUL.CRF]: first
47338 + 430081 Bytes symbol space free
0 Warning Errors
0 Severe Errors
```

After assembling the program as shown above you will find two additional files with the **object file**, automatically generated by the assembler, in your directory i.e. the **list file** and the **cross-reference file**. Name must be provided for .LST else NUL (nothing) will be generated.

1. OBJECT FILE

A non-executable file contains the machine code translation of assembly code, plus other information needed to produce the executable.

2. LIST FILE

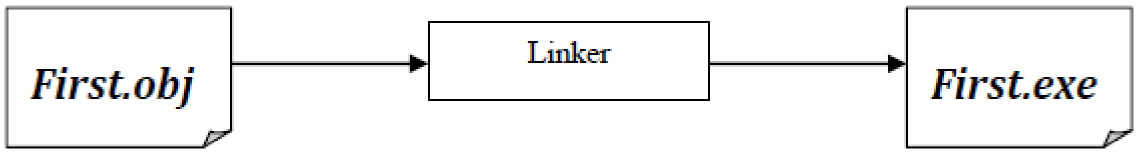
The list file is a text file that gives you assembly language code and the corresponding machine language code, a list of names used in the program, error messages and other statistics as shown below for the assembly file first.asm:

```
PGM4_1: CASE CONVERSION PROGRAM                    Page    1-1


   1                                 TITLE        PGM4_1: CASE CONVERSION PROGRAM
   2                                .MODEL        SMALL
   3                                .STACK        100H
   4                                .DATA
   5  = 000D                                 CR      EQU     0DH
   6  = 000A                                 LF      EQU     0AH
   7  0000    45 4E 54 45 52 20              MSG1    DB      'ENTER A LOWER CASE LETTER:  $'
   8           41 20 4C 4F 57 45
   9           52 20 43 41 53 45
  10          20 4C 45 54 54 45
  11          52 3A 20 20 24
  12 001D   0D 0A 49 4E 20 55              MSG2    DB      0DH, 0AH, 'IN UPPER CASE IT IS:   '
  13           50 50 45 52 20 43
  14           41 53 45 20 49 54
  15          20 49 53 3A 20 20
  16 0035    00 24                         CHAR    DB      ? ,'$'
  17                                 .CODE
  18 0000                              MAIN           PROC
  19                                   ; initialize DS
  20 0000   B8 ---- R                        MOV     AX, @DATA     ; get data segment
  21 0003   8E D8                            MOV     DS, AX        ; initialize DS
  22                                   ; print user prompt
  23 0005  8D 16 0000 R                      LEA     DX, MSG1      ; get first message
  24 0009  B4 09                             MOV      AH, 9        ; display string function
  25 000B  CD 21                             INT     21H           ; display first message

  26                                   ; input a character and convert to uppercase
  27 000D  B4 01                             MOV     AH, 1         ; read character function
  28 000F  CD 21                             INT     21H           ; read a small letter into AL
  29 0011  2C 20                             SUB     AL, 20H       ; convert it to upper case
  30 0013  A2 0035 R                         MOV     CHAR, AL      ; and store it
  31                                   ; display on the next line
  32 0016  8D 16 001D R                      LEA     DX, MSG2      ; get second message
  33 001A  B4 09                             MOV     AH, 9         ; display string function
  34 001C  CD 21                             INT     21H           ; display message & upper case letter in front
  35                                   ; DOS    exit

PGM4_1: CASE CONVERSION PROGRAM                    Page    1-2

  36 001E  B4 4C                             MOV     AH, 4CH       ; DOS exit
  37 0020  CD 21                             INT     21H

  38 0022                              MAIN           ENDP
  39                                   END     MAIN

PGM4_1: CASE CONVERSION PROGRAM                    Symbols-1

Segments and Groups:

          N a m e              Length        Align         Combine       Class

DGROUP . . . . . . . . . . . . .      GROUP
  _DATA  . . . . . . . . . . . . . 0037     WORD          PUBLIC        'DATA'
  STACK  . . . . . . . . . . . . . 0100     PARA          STACK         'STACK'
_TEXT  . . . . . . . . . . . . . . 0022     WORD          PUBLIC        'CODE'
```

```
Symbols:

            N a m e                   Type          Value          Attr

CHAR . . . . . . . . . . . . . . . .  L BYTE        0035           _DATA
CR . . . . . . . . . . . . . . . . .  NUMBER        000D

LF . . . . . . . . . . . . . . . . .  NUMBER        000A

MAIN . . . . . . . . . . . . . . . . N PROC      0000        _TEXT       Length = 0022
MSG1 . . . . . . . . . . . . . . . .  L BYTE        0000           _DATA
MSG2 . . . . . . . . . . . . . . . . L BYTE      001D        _DATA

@CODE . . . . . . . . . . . . . . .   TEXT      _TEXT
@CODESIZE . . . . . . . . . . .       TEXT       0
@CPU . . . . . . . . . . . . . . .   TEXT      0101h
@DATASIZE . . . . . . . . . . .       TEXT       0
@FILENAME . . . . . . . . . . .       TEXT       cc
@VERSION . . . . . . . . . . .       TEXT       510

   32 Source  Lines
   32 Total   Lines
   23 Symbols

 46146 + 447082 Bytes symbol space free

    0 Warning Errors
    0 Severe  Errors
```

3. CROSS-REFERENCE FILE
List names used in the program and the line number.

LINKING THE PROGRAM
Linking is the process of converting the one or more object files into a single executable file. The program “LINKER” does this.



```
C:\>link first.obj;
Microsoft (R) Overlay Linker Version 3.64
Copyright (C) Microsoft Corp 1983-1988. All rights reserved.
```

RUNNING THE PRORAM
On the command line type the name of the program to run.

```
C:\>first.exe
ENTER A LOWER CASE LETTER: a
IN UPPER CASE IT IS: A
```

DEBUGGING

DEBUG is a primitive but utilitarian program, supplied with MS-DOS, with a small easy to learn command set. After assembling and linking the program in previous practical, (**first.asm**) we take the **first.exe** into DEBUG.
On the MS-DOS prompt type the following command,

```
C:\>DEBUG first.exe
-
```

DEBUG comes back with its "-" command prompt.

Useful Commands

Commands	Description
R	to display registers
R IP	to display/change IP register
T	to execute single instruction
T 4	to execute 4 instructions
G	execute till completion
G 4	execute till address 0004
D	dump bytes in hex format
D 100	dump 128bytes starting from DS:100
D 100 104	dump from 100 to 104
E DS:0 A B C	enter Ah, Bh, Ch in bytes DS:0, DS:1, DS:2
E 25	Enter bytes interactively starting at DS: 25. Space bar moves to next byte
Q	quit from debug

To view registers and FLAGS, type "R"

```
C:\>debug first.exe
-R
AX=0000 BX=0000 CX=0030 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=1189 ES=1189 SS=119C CS=1199 IP=0000
NV UP EI PL NZ NA PO NC
1199:0000 B89A11 MOV AX,119A
-
```

As we know 8086/88 has 14 registers, all of these registers are shown by DEBUG with different values stored in these registers.

FLAG REGISTER

The letters pairs on the fourth line are the current setting of some of the status and control FLAGS. The FLAGS displayed and the symbols DEBUG uses are the following:

▨	▨	▨	▨	O	D	I	T	S	Z	▨	A	▨	P	▨	C

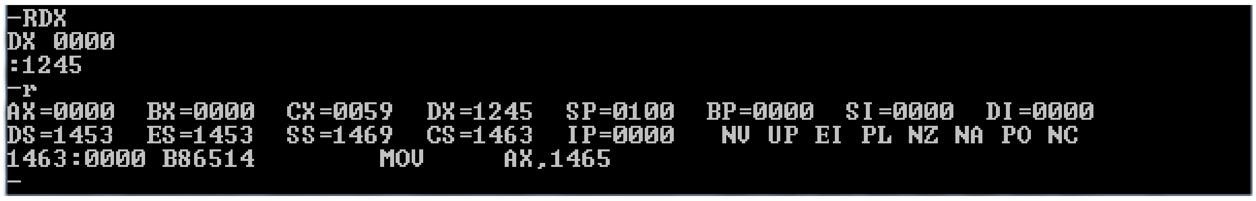
▨ Unused Flag Register Bits

SYMBOL	FLAGS	CLEAR (0)	SET (1)
O	Overflow Flag	NV	OV
D	Direction Flag	UP	DN
I	Interrupt Flag	DI	EI
S	Sign Flag	PL	NG
Z	Zero Flag	NZ	ZR
A	Auxiliary Flag	NA	AC
P	Parity Flag	PO	PE
C	Carry Flag	NC	CY

To change the contents of a register-for example, AX to 1245h

```
-RDX
DX 0000
:1245
```

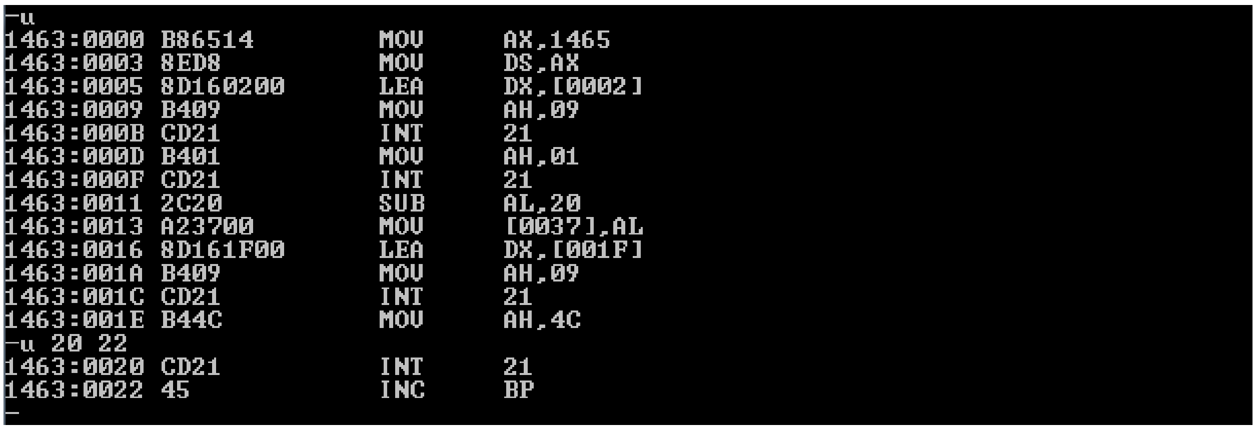
Note:- DEBUG assumes that all numbers are expressed in **hex**. Now let us verify the change, through "R" command.



```
-RDX
DX 0000
:1245
-r
AX=0000  BX=0000  CX=0059  DX=1245  SP=0100  BP=0000  SI=0000  DI=0000
DS=1453  ES=1453  SS=1469  CS=1463  IP=0000   NV UP EI PL NZ NA PO NC
1463:0000 B86514        MOV     AX,1465
-
```

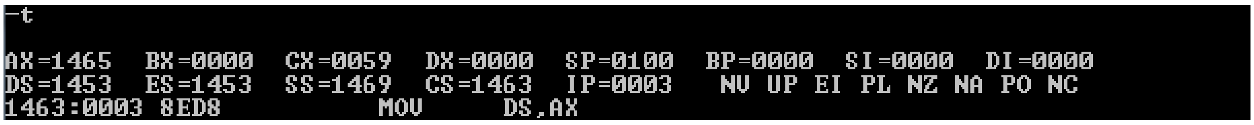
DX now contain 1245h.

The next instruction to be executed by the CPU is written on the last line with its address in the memory. Let us execute each instruction one by one using "T" trace command. But before that, just check whether the ".exe" file is representing the same assembly language program or not, using the U (unassembled) command.



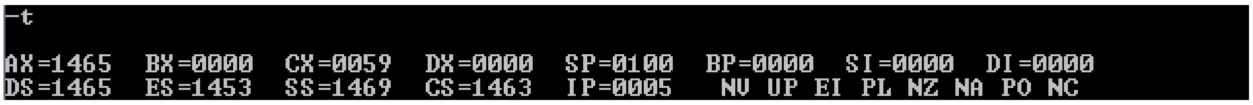
```
-u
1463:0000 B86514        MOV     AX,1465
1463:0003 8ED8          MOV     DS,AX
1463:0005 8D160200      LEA     DX,[0002]
1463:0009 B409          MOV     AH,09
1463:000B CD21          INT     21
1463:000D B401          MOV     AH,01
1463:000F CD21          INT     21
1463:0011 2C20          SUB     AL,20
1463:0013 A23700        MOV     [0037],AL
1463:0016 8D161F00      LEA     DX,[001F]
1463:001A B409          MOV     AH,09
1463:001C CD21          INT     21
1463:001E B44C          MOV     AH,4C
-u 20 22
1463:0020 CD21          INT     21
1463:0022 45            INC     BP
-
```

The U command by default shows 32 bytes of program coding. The last instruction shown above is not our last program's instruction. To see the remaining instructions, specify directly some address ranges ahead. Now execute instructions one be one using T command.



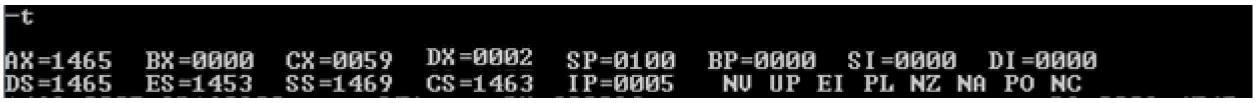
```
-t

AX=1465  BX=0000  CX=0059  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1453  ES=1453  SS=1469  CS=1463  IP=0003   NV UP EI PL NZ NA PO NC
1463:0003 8ED8          MOV     DS,AX
```

AX now have the segment number of the data segment. Again press T for one more time will execute the instruction MOV DS, AX as shown on the last line above. This will initialize the data segment register with the data segment address of the program.



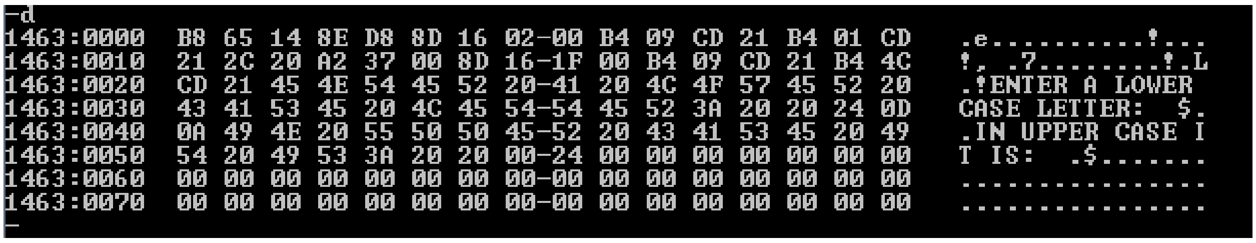
```
-t

AX=1465  BX=0000  CX=0059  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=0005   NV UP EI PL NZ NA PO NC
```

The next command LEA DX, [0002] will load the offset address of MSG1 in DX which is 0002.



```
-t

AX=1465  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=0005   NV UP EI PL NZ NA PO NC
```

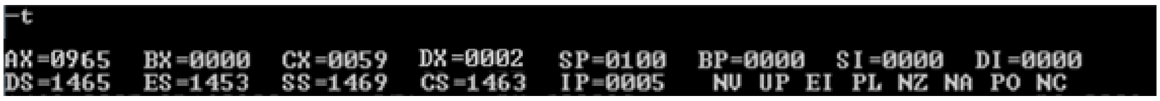
Check the contents of the data segment using the D command:



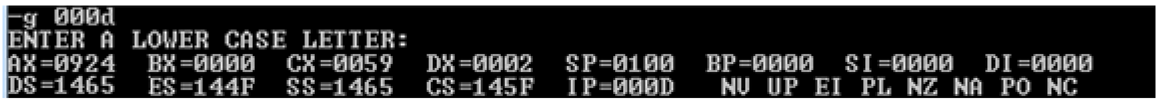
```
-d
1463:0000  B8 65 14 8E D8 8D 16 02-00 B4 09 CD 21 B4 01 CD   .e..........!...
1463:0010  21 2C 20 A2 37 00 8D 16-1F 00 B4 09 CD 21 B4 4C   !, .7........!.L
1463:0020  CD 21 45 4E 54 45 52 20-41 20 4C 4F 57 45 52 20   .!ENTER A LOWER
1463:0030  43 41 53 45 20 4C 45 54-54 45 52 3A 20 20 24 0D   CASE LETTER:  $.
1463:0040  0A 49 4E 20 55 50 50 45-52 20 43 41 53 45 20 49   .IN UPPER CASE I
1463:0050  54 20 49 53 3A 20 20 00-24 00 00 00 00 00 00 00   T IS:  .$.......
1463:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1463:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

We can see that the string variables initialized in the Data Segment has been successfully loaded into the memory locations as above.

Now through MOV AH, 09 and interrupt command -g 000d, MSG1will be displayed as shown below:

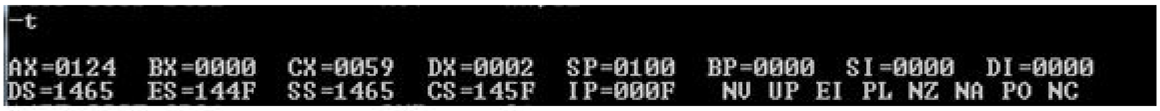


```
-t

AX=0965  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=0005   NV UP EI PL NZ NA PO NC
```



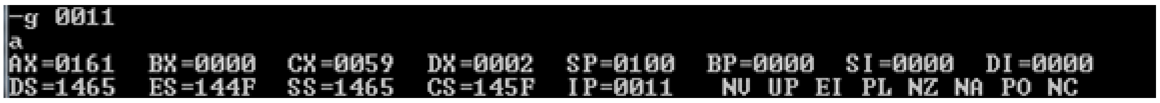
```
-g 000d
ENTER A LOWER CASE LETTER:
AX=0924  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=144F  SS=1465  CS=145F  IP=000D   NV UP EI PL NZ NA PO NC
```

Pressing T one more time will move 01 in AH so that we can take input.

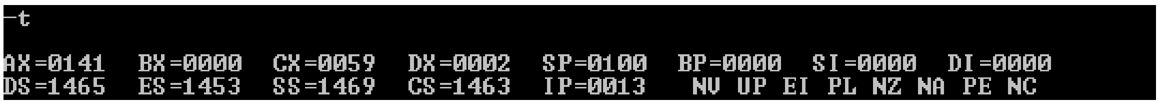


```
-t

AX=0124  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=144F  SS=1465  CS=145F  IP=000F   NV UP EI PL NZ NA PO NC
```

Now through interrupt command -g 0011, user will be prompted to enter a lower case letter As you can see, ‘a’ is entered as input, so AX will now contain 0161 where 61 is the ASCII code of ‘a’.

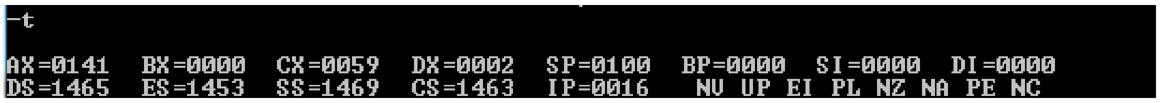

```
-g 0011
a
AX=0161  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=144F  SS=1465  CS=145F  IP=0011   NV UP EI PL NZ NA PO NC
```

Now the SUB command will subtract 20 out of the contents of AL to perform case conversion.

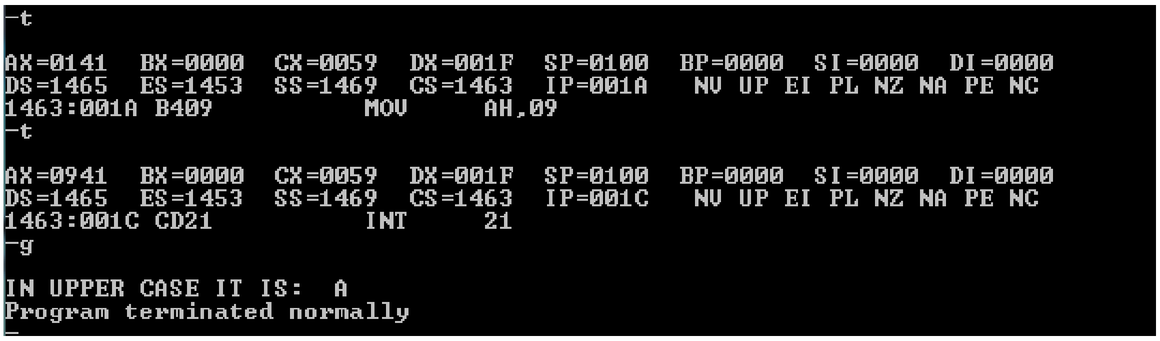

```
-t

AX=0141  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=0013   NV UP EI PL NZ NA PE NC
```

Again pressing ‘t’, it will store the case conversion output i.e. ‘A’ in memory.

Now to display MSG2, its offset address will be loaded in DX:

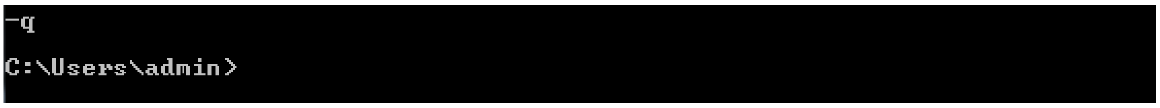

```
-t

AX=0141  BX=0000  CX=0059  DX=0002  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=0016   NV UP EI PL NZ NA PE NC
```

MOV AH, 09 and interrupt command are used to print the string on screen as done before. The result will be displayed as follows:


```
-t

AX=0141  BX=0000  CX=0059  DX=001F  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=001A   NV UP EI PL NZ NA PE NC
1463:001A B409          MOV     AH,09
-t

AX=0941  BX=0000  CX=0059  DX=001F  SP=0100  BP=0000  SI=0000  DI=0000
DS=1465  ES=1453  SS=1469  CS=1463  IP=001C   NV UP EI PL NZ NA PE NC
1463:001C CD21          INT     21
-g

IN UPPER CASE IT IS:  A
Program terminated normally
-
```

This message indicates that the program has run to completion. The program must be reloaded to execute again.

Now leave the DEBUG using “Q”


```
-q

C:\Users\admin>
```

Lab Session 08

OBJECTIVE

Calling a subroutine from another assembly file as a near procedure

THEORY

Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.

Procedure Declaration

- The syntax of procedure declaration is the following:

  name **PROC** ***NEAR***
  ; *body of procedure*
  ret
  name **ENDP**

The CALL Instruction

- CALL invokes a procedure as:

  call ***name***

  where ***name*** is the name of a procedure.

Executing a CALL

- The return address to the calling program (the current value of the IP) is saved on the stack
- IP get the offset address of the first instruction of the procedure (this transfers control to the procedure)

The RET instruction

- To return from a procedure, the executed instruction is:

  ret ***pop_value***

- The integer argument ***pop_value*** is optional.
- **ret** causes the stack to be popped into IP.

A Case Conversion Program
Prompt the user to enter a lowercase letter, and on next line displays another message with letter in uppercase, as:
Enter a lowercase letter: a
In upper case it is: A

We will create two different assembly files to implement case conversion. First file contains the code that will prompt user to enter a lower case letter. This file contains a call to a near procedure named

CONVERT, which is used to perform case conversion. The second file contains the code of the procedure CONVERT. So, when the procedure CONVERT is invoked, the given lower case letter will be converted to upper case. The control will then be returned back to the calling procedure in the first file which will display the output.

Assembly code for both of the files is given below:

```
TITLE           PGM4_2: CASE CONVERSION
EXTRN   CONVERT: NEAR
.MODEL          SMALL
.STACK  100H
.DATA
MSG     DB      'ENTER A LOWER CASE LETTER:  $'
.CODE
MAIN    PROC
        MOV     AX, @DATA       ; get data segment
        MOV     DS, AX          ; initialize DS
; print user prompt
        LEA     DX, MSG         ; get first message
        MOV     AH, 9           ; display string function
        INT     21H             ; display first message
; input a character and convert to upper case
        MOV     AH, 1           ; read character function
        INT     21H             ; read a small letter into AL
        CALL    CONVERT         ; convert to uppercase
        MOV     AH, 4CH
        INT     21H             ; DOS exit
MAIN    ENDP
        END     MAIN
```

Save your program with (.asm) extension. If “**first**” is the name of program then save it as “**first.asm**”.

```
TITLE           PGM4_2A : CASE CONVERSION
PUBLIC  CONVERT
.MODEL          SMALL
.DATA
MSG     DB      0DH, 0AH, 'IN UPPER CASE IT IS:  '
CHAR    DB      -20H,'$'
.CODE
CONVERT         PROC    NEAR
;converts char in AL to uppercase
        PUSH    BX
        PUSH    DX
        ADD     CHAR,AL
        MOV     AH,9
        LEA     DX,MSG
        INT     21H
        POP     DX
        POP     BX
        RET
CONVERT         ENDP
        END
```

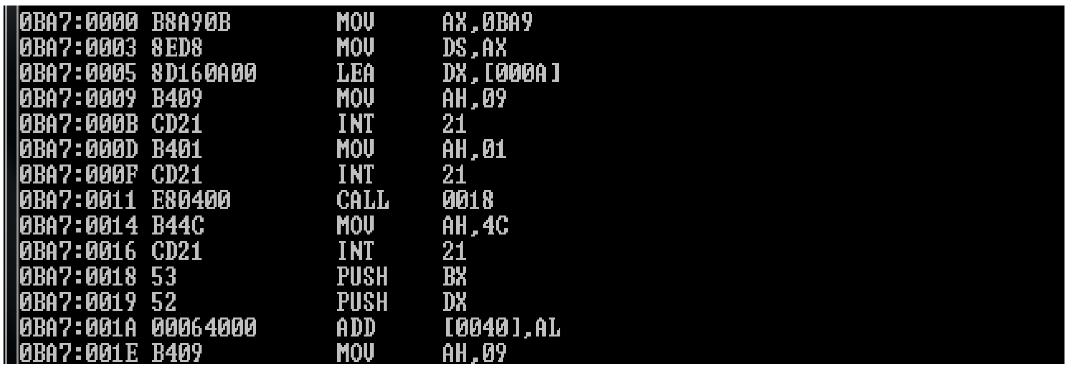
Save the previous program as well with (.asm) extension. If "**second**" is the name of program then save it as "**second.asm**".

Now follow the steps as mentioned in the previous lab session to assemble the two files. First perform all the steps to assemble and create .obj file for the first program, list file and cross reference file will also be generated automatically by the assembler for the first program. Now, do the same for the second program. Observe the list files for both the programs.

Now we have to link the two files. For this, write the following line on the command prompt:
>*link first + second*

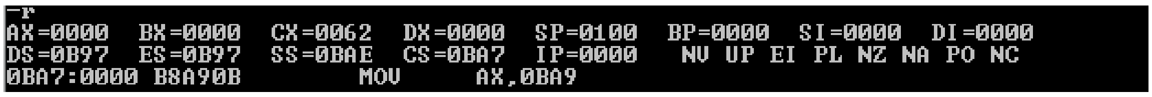
Then give any name to the resultant file (e.g.: first). Now we have a single .exe file to perform case conversion. Write following line on the command prompt:
>*debug first.exe*

Check whether the .exe file is representing the same assembly language program or not, using the U (unassembled) command.

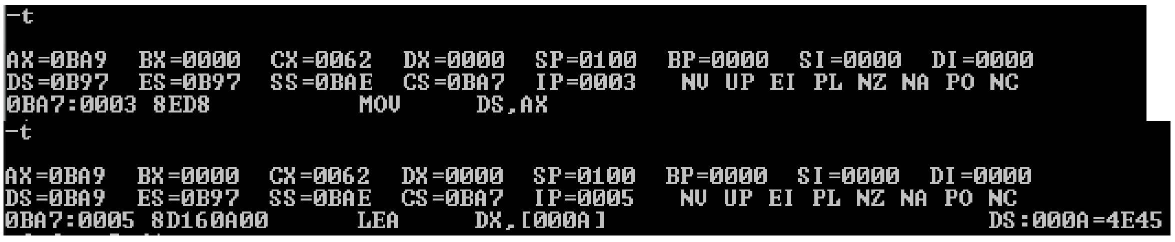

```
0BA7:0000 B8A90B        MOV     AX,0BA9
0BA7:0003 8ED8          MOV     DS,AX
0BA7:0005 8D160A00      LEA     DX,[000A]
0BA7:0009 B409          MOV     AH,09
0BA7:000B CD21          INT     21
0BA7:000D B401          MOV     AH,01
0BA7:000F CD21          INT     21
0BA7:0011 E80400        CALL    0018
0BA7:0014 B44C          MOV     AH,4C
0BA7:0016 CD21          INT     21
0BA7:0018 53            PUSH    BX
0BA7:0019 52            PUSH    DX
0BA7:001A 00064000      ADD     [0040],AL
0BA7:001E B409          MOV     AH,09
```

The U command by default shows 32 bytes of program coding. To see the remaining instructions, specify directly some address ranges ahead.

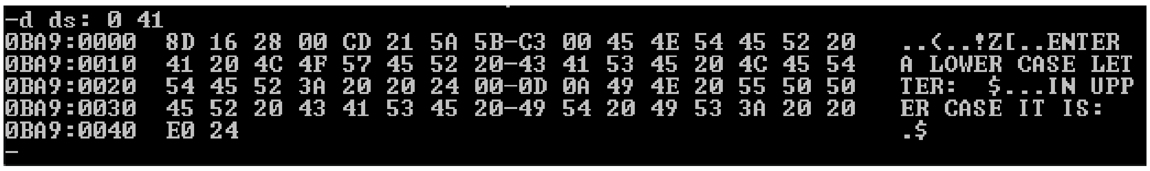
To see initial condition of registers, type R command.


```
-r
AX=0000  BX=0000  CX=0062  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=0B97  ES=0B97  SS=0BAE  CS=0BA7  IP=0000   NV UP EI PL NZ NA PO NC
0BA7:0000 B8A90B        MOV     AX,0BA9
```

Now execute instructions one be one using T command.


```
-t

AX=0BA9  BX=0000  CX=0062  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=0B97  ES=0B97  SS=0BAE  CS=0BA7  IP=0003   NV UP EI PL NZ NA PO NC
0BA7:0003 8ED8          MOV     DS,AX
-t

AX=0BA9  BX=0000  CX=0062  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=0BA9  ES=0B97  SS=0BAE  CS=0BA7  IP=0005   NV UP EI PL NZ NA PO NC
0BA7:0005 8D160A00      LEA     DX,[000A]                          DS:000A=4E45
```

Through above commands, we have initialized the data segment, verify by using D command.



```
-d ds: 0 41
0BA9:0000  8D 16 28 00 CD 21 5A 5B-C3 00 45 4E 54 45 52 20   ..(..!Z[..ENTER
0BA9:0010  41 20 4C 4F 57 45 52 20-43 41 53 45 20 4C 45 54   A LOWER CASE LET
0BA9:0020  54 45 52 3A 20 20 24 00-0D 0A 49 4E 20 55 50 50   TER:  $...IN UPP
0BA9:0030  45 52 20 43 41 53 45 20-49 54 20 49 53 3A 20 20   ER CASE IT IS:
0BA9:0040  E0 24                                             .$
-
```

You can see in the above figure that the data segment is initialized with the messages. Now execute the assembly and interrupt commands and note down the observations stepwise.

EXERCISE 1

Write a program that takes two numbers as input and performs their addition. The code for addition of the numbers should be present in another assembly file that should be called as a near procedure in the first file.

Lab Session 09

OBJECTIVE

Executing Data Transfer and Stack operation instructions

THEORY

Opcode of following MOV instructions: 100010dw oorrrmmm disp

MOV reg1 , reg2	; copy the contents of 8-bit register "reg2" in the 8-bit register "reg1".
MOV mem , reg	; copy the contents of 8-bit register "reg" in memory location "mem".
MOV reg , mem	; copy the contents of memory location "mem" into the register "reg".

Opcode of following MOV instruction: 100010dw oorrrmmm disp data

MOV mem , imm ; copy the immediate data "imm" into memory location "mem".

Opcode of following MOV instruction: 1011wrrr data

MOV reg , imm ; copy the immediate data "imm" into the register "reg".

Opcode of following MOV instructions: 101000dw disp

MOV mem , acc	; copy the contents of accumulator into memory location "mem".
MOV acc , mem	; copy the contents of memory location "mem" into accumulator.

Stack instructions:

Instruction	**opcode**	**Description**
PUSH reg	01010rrr	pushes the contents of register "reg" onto the stack.
PUSH mem	11111111 oo110mmm disp	pushes the contents of memory location "mem" onto the stack.
PUSH seg	00sss110	pushes the contents of segment register "seg" onto the stack.
PUSH imm	011010s0 data	pushes the immediate data "imm" onto the stack.
PUSHA/PUSHAD	01100000	pushes all the registers onto the stack
PUSHF/PUSHFD	10011100	pushes the flags onto the stack.
POP reg	01011rrr	pops the contents of register "reg" from top of the stack.
POP mem	10001111 oo000mmm disp	pops the contents of memory location "mem" from top of the stack.
POP seg	00sss111	pops the contents of segment register "seg" from top of the stack
POPA/POPAD	01100001	pops all the registers from the stack.
POPF/POPFD	10010000	pops the flags from the stack.

PUSHA and POPA instructions are not available in 8008 microprocessor.

ASSEMBLY PROGRAM

```
        .MODEL SMALL

        .STACK 100H

        .CODE
        MAIN PROC

1.          MOV AX , 0B386H
2.          MOV BX , 0200H
3.          MOV CX , 0A5CH
4.          MOV DX , 0D659H
5.          MOV   BP , 0300
6.          MOV   ES , CX
7.          MOV WORD PTR DS:[0200H], 95D8H
8.          ADD    AX , BX
9.          PUSH  AX
10.         PUSH WORD PTR[BX]
11.         PUSH  ES
12.         PUSHF
13.         PUSH  DX
14.         POP     CX
15.         POP     DI
16.         POP    DS
17.         POP WORD PTR[BP]
18.         POPF

19.         MOV AH, 4CH
20.         INT 21H

        MAIN ENDP
        END MAIN
```

OBSERVATIONS

By using single stepping observe the working of the program.

Inst#	AX	BX	CX	DX	Flag	BP	SP	ES	DS	DI	[0200]	[0300]
7th												
8th												
13th												
14th												
15th												
16th												
17th												
18th												

Note the contents of the SS: SP register after 13th instruction and **then** examine the contents of the corresponding memory locations pointed out to by SS:SP.

EXERCISE 1

Write a program, which
1. Loads AX, BX, CX and DX registers with A154, 7812, 9067, BFD3.
2. Exchange lower byte of CX and higher byte of DX registers by using memory location 0150 in between the transfer. Then stores CX and DX registers onto memory location 0170 onward.
3. Exchange higher byte of AX and higher byte of BX registers by using memory location 0160 in between the transfer. Then stores AX and BX registers onto memory location 0174 onward.
4. Also draw the flow chart of the program.

Program Flowchart

OBSERVATIONS 1

- Observe the contents of memory location from 0170 to 0177 and record them below in a table.
- Observe the contents of registers by using single stepping and record the final contents below.

Contents of memory location

Contents of Registers

AX	
BX	
CX	
DX	

EXERCISE 2

Write a program that produces certain delay and then increment the Accumulator register. When accumulator produces a carry then the buzzer should generate tone for a certain time. Implement this program using subroutine. The length of delay is passed to the delay subroutine as a parameter, using stack. Also draw the flowchart. You can also use any assembler for this exercise.

Program Flowchart

Lab Session 10

OBJECTIVE

Implementing Logic group of instructions

THEORY

Gate instructions:

Opcode	Inst.	Operand1, Operand2	Description
001000dw oorrrmmm disp	AND	reg/mem, reg/mem	Perform logical operation on register/memory with the memory or the second register. Both the two operands cannot be the memory location.
000010dw oorrrmmm disp	OR		
001100dw oorrrmmm disp	XOR		
100000sw oo100mmm disp data	AND	reg/mem/acc, imm	Perform logical operation on the "immediate value" with the contents of the register / memory location or specifically the accumulator.
100000sw oo001mmm disp data	OR		
100000sw oo100mmm disp data	XOR		

Shift and Rotate Instructions:

Description	Instruction Op-code			TTT value
	1101000w ooTTTmmm disp	1101001w ooTTTmmm disp	1101001w ooTTTmmm disp	
	Shift/Rotate one time	Shift/Rotate according to the contents of the CL register	Shift/Rotate according to the immediate memory location "mem"	
Rotate left without carry	**ROL reg/mem , 1**	**ROL reg/mem , CL**	**ROL reg/mem , imm**	000
Rotate right without carry	**ROR reg/mem , 1**	**ROR reg/mem , CL**	**ROR reg/mem , imm**	001
Rotate left with carry	**RCL reg/mem , 1**	**RCL reg/mem , CL**	**RCL reg/mem , imm**	010
Rotate right with carry	**RCR reg/mem , 1**	**RCR reg/mem , CL**	**RCR reg/mem , imm**	011
Shift logic left	**SAL reg/mem , 1**	**SAL reg/mem , CL**	**SAL reg/mem , imm**	100
Shift Arithmetic left	**SHL reg/mem , 1**	**SHL reg/mem , CL**	**SHL reg/mem , imm**	″
Shift logic right	**SHR reg/mem , 1**	**SHR reg/mem , CL**	**SHR reg/mem , imm**	101
Shift arithmetic right	**SAR reg/mem , 1**	**SAR reg/mem , CL**	**SAR reg/mem , imm**	111

ASSEMBLER PROGRAM – I (GATE INSTRUCTIONS)

```
        .MODEL SMALL

        .STACK 100H

        .CODE
        MAIN PROC
1.              MOV AX, 8A53H
2.              MOV BX, 0200H
3.              MOV CX, 692DH
4.              MOV DX, 0E6CBH
5.              MOV WORD PTR[BX], 7B8AH
6.              AND AX, BX
7.              AND CX, WORD PTR[BX]
8.              OR WORD PTR[BX], CX
9.              OR WORD PTR[BX], 6F0CH
10.             XOR AX, 94D7H
11.             XOR DX, 0C4D1H

12.             MOV AH, 4CH
13.             INT 21H

        MAIN ENDP

        END MAIN
```

OBSERVATIONS

By using single stepping record the contents of following registers:

Register	After 5th instruction	After 6th instruction	After 7th instruction	After 8th instruction	After 9th instruction	After 10th instruction	After 11th instruction
AX							
BX							
CX							
DX							
Flag							
Word[0200]							

ASSEMBLER PROGRAM –II (SHIFT AND ROTATE INSTRUCTIONS)

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV AX, 1111H
        MOV BX, 2222H
        MOV CX, 3303H
        MOV SI, 9254H
        MOV WORD PTR DS:[0100H], 6655H
        MOV BYTE PTR DS:[0123H], 77H
        MOV WORD PTR DS:[0126H], 9988H
        ROL AX, 1
        ROL BYTE PTR DS:[0100H], 1
        ROL AX, CL
        ROL BYTE PTR DS:[0100H], CL
        RCL BX, 1
        RCL WORD PTR DS:[0100H], 1
        RCL AX, CL
        RCL WORD PTR DS:[0100H], CL
        ROR AX, 1
        ROR AX, CL
        ROR BYTE PTR DS:[0126H], CL
        RCR BX, 1
        RCR BYTE PTR DS:[0127H], CL
        SHL BX, 1
        SHL BYTE PTR DS:[0126H], CL
        SAR SI, 1
        SAR SI, CL
        SHR BYTE PTR DS:[0123H], 1
        SHR BYTE PTR DS:[0123H], CL

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

By using single stepping observe the contents of the registers and memory locations that are used to store data in the program.

	Registers				Memory Locations				
	AX	BX	SI	CF	100	101	123	126	127
7.									
8.									
9.									
10.									
11.									
12.									
13.									
14.									
15.									
16.									
17.									
18.									
19.									
20.									
21.									
22.									
23.									
24.									
25.									
26.									

EXERCISE 1

Write a program which mask the bits of AX register, by setting left-most 4 bits ,resetting right most 4 bits and complement bit position number 9 and 10.(Hint: Use AND,OR and XOR instructions for masking).

Program Flowchart

EXERCISE 2

An ASCII coded number can be converted to BCD by masking. Write a program which converts ASCII 30H - 39H to BCD 0-9. Use any assembler for this exercise.

Program Flowchart

EXERCISE 3

Write a program, which multiply two 8-bit numbers using add and shift logic. Check the program by

(i) loads accumulator with 20H and then multiply it by 10H.

(ii) loads BL with 10H and multiply it by 12H.

Use any assembler of your choice for this purpose.

Also draw the flow chart of the program.

Program Flowchart

OBSERVATIONS FOR EXERCISE 3

Value of the Multiplicand = -----------------.

Value of the Multiplier = -----------------.

Result of Multiplication = ------------------.

Lab Session 11

OBJECTIVE

Simulating Transfer of control instructions

THEORY

Jump Instructions transfers the control of program to the location addressed by the specified location (as listed in description column)

Instruction	**Opcode**	**Description**
JMP label (short)	11101011 disp	IP+disp
JMP label (near)	11101001 disp	
JMP label (far)	11101010 IPnew CSnew	Label
JMP reg (near)	11111111 oo100mmm	contents of register “reg”
JMP mem (near)		memory location “mem”
JMP mem (far)	11111111 oo101mmm	
Jcnd label (8-bit disp)	0111cccc disp	IP+disp; when condition “cnd” becomes true
Jcnd label (16-bit disp)	00001111 1000cccc disp	

Condition Codes	***Mnemonic***	***Flag***	***Description***
0000	*JO*	*O = 1*	*Jump if overflow*
0001	*JNO*	*O = 0*	*Jump if no overflow*
0010	*JB/JNAE*	*C = 1*	*Jump if below*
0011	*JAE/JNB*	*C = 0*	*Jump if above or equal*
0100	*JE/JZ*	*Z = 1*	*Jump if equal/zero*
0101	*JNE/JNZ*	*Z = 0*	*Jump if not equal/zero*
0110	*JBE/JNA*	*C = 1 , Z = 1*	*Jump if below or equal*
0111	*JA/JNBE*	*O = 0 , Z = 0*	*Jump if above*
1000	*JS*	*S = 1*	*Jump if sign*
1001	*JNS*	*S = 0*	*Jump if no sign*
1010	*JP/JPE*	*P = 1*	*Jump if parity*
1011	*JNP/JPO*	*P = 0*	*Jump if no parity*
1100	*JL/JNGE*	*S = O*	*Jump if less than*
1101	*JGE/JNL*	*S = 0*	*Jump if greater than or equal*
1110	*JLE/JNG*	*Z = 1 , S = O*	*Jump if less than or equal*
1111	*JG/JNLE*	*Z = 0 , S = O*	*Jump if greater than*

ASSEMBLER PROGRAM – I (Unconditional Branch)

```
.MODEL SMALL

.STACK 100H

.DATA
        MSG1  DB    0Dh, 0AH, 'ENTER THE FIRST CHARACTER: $'
        MSG2  DB    0DH, 0AH, 'ENTER THE SECOND CHARACTER: $'

.CODE
MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

again:
        LEA DX, MSG1        ; PROMPTING USER TO ENTER THE FIRST CHARACTER
        MOV AH, 9
        INT 21H

        MOV AH, 1           ; TAKING INPUT FROM THE USER
        INT 21H

        LEA DX, MSG2        ; PROMPTING USER TO ENTER THE SECOND CHARACTER
        MOV AH,9
        INT 21H

        MOV AH, 1           ; TAKING INPUT FROM THE USER
        INT 21H

        JMP again           ; Jump to the first instruction

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

By using single stepping observe the working of the program. Record the content of the AX registers.

	Character	*AX*
1		
2		
3		
4		
5		

When does this program end? ______________________________

ASSEMBLER PROGRAM – II (Conditional Branch)

```
.MODEL SMALL

.STACK 100H

.DATA
        MSG1  DB    0Dh, 0AH, 'ENTER THE FIRST CHARACTER: $'
        MSG2  DB    0DH, 0AH, 'THE FIRST CHARACTER IS: '
        CH1   DB    ?, '$'
        MSG3  DB    0DH, 0AH, 'ENTER THE SECOND CHARACTER: $'
        MSG4  DB    0DH, 0AH, 'THE SECOND CHARACTER IS: '
        CH2   DB    ?, '$'

.CODE
MAIN PROC
        MOV AX, @DATA
        MOV DS, AX
again:
        LEA DX, MSG1        ; PROMPTING USER TO ENTER THE FIRST CHARACTER
        MOV AH, 9
        INT 21H

        MOV AH, 1           ; TAKING INPUT FROM THE USER
        INT 21H

        MOV CH1, AL         ; UPDATING RESULT IN THE VARIABLE

        MOV BX, 0000H
        MOV BL, AL

        LEA DX, MSG2        ; OUTPUTTING THE FIRST CHARACTER ON SCREEN WITH MESSAGE
        MOV AH, 9
        INT 21H

        LEA DX, MSG3        ; PROMPTING USER TO ENTER THE SECOND CHARACTER
        MOV AH, 9
        INT 21H

        MOV AH, 1           ; TAKING INPUT FROM THE USER
        INT 21H

        MOV CH2, AL         ; UPDATING RESULT IN THE VARIABLE

        LEA DX, MSG4        ; OUTPUTTING THE SECOND CHARACTER ON SCREEN WITH MESSAGE
        MOV AH, 9
        INT 21H

        MOV AH, 0
        MOV AL, CH2

        CMP AX, BX
        JNZ again

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

By using single stepping observe the contents of registers AX, BX after execution of each instruction.

	(Different Key input)		(Same Key Input)	
	AX	**BX**	**AX**	**BX**
After 1st instruction	___________	___________	___________	___________
After 2nd instruction	___________	___________	___________	___________
After 3rd instruction	___________	___________	___________	___________
After 1st instruction	___________	___________	___________	___________
After 1st instruction	___________	___________	___________	___________
After 4th instruction	___________	___________	___________	___________
After 5th instruction	___________	___________	___________	___________
After 6th instruction	___________	___________	___________	___________
After 7th instruction	___________	___________	___________	___________
After 8th instruction	___________	___________	___________	___________
After 9th instruction	___________	___________	___________	___________
Flag register after 8th instruction	___________	___________	___________	___________

EXERCISE

Write a program, which prints your name on the screen when 'space' key is pressed from the keyboard. Implement using conditional jump instruction. Also draw the flow chart of the program.

Program

____
____
____
____
____
____
____
____
____
____
____
____
____
____

Flowchart

Lab Session 12

OBJECTIVE

Implementing Arithmetic group of instructions

THEORY

Opcode	Inst.	Operand1, Operand2	Description
000000/000101dw oorrrmmm disp	ADD/SUB	reg1, reg2 OR mem, reg OR reg, mem	add/subtract (with carry/borrow) the contents of the register “reg” or “mem” with/from the register “reg” or “mem”
000100/000110dw oorrrmmm disp	ADC/SBB		
100000sw oo000/101mmm disp data	ADD/SUB	reg, imm OR mem, imm OR acc, imm	add/subtract (with carry/borrow) the immediate data “imm” with/from register/memory location or specifically the accumulator.
100000sw oo010/011mmm disp data	ADC/SBB		

<u>Opcode of following MUL instructions:</u> 1111011w oo100mmm disp

```
MUL   reg          ; multiply the contents of register “reg” with the accumulator register and
                   ; return the result in “AH and AL” or “DX and AX”.
MUL  mem           ; multiply the contents of memory “mem” with the accumulator register and
                   ; return the result in “AH and AL” or “DX and AX”.
```

<u>Opcode of following DIV instructions:</u> 1111011w oo110mmm disp

```
DIV  reg           ; divide the contents of the accumulator register by the contents of  register “reg”
                   ; and return the remainder in AH and the quotient in AL,
                   ; or the remainder in DX and the quotient in AX.
DIV  mem           ; divide the contents of the accumulator register by the contents of
                   ; memory location “mem” and return the remainder in AH and the
                   ; quotient in AL or the remainder in DX and the quotient in AX.
```

ASSEMBLER PROGRAM – I (Addition)

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV AX, 4000H
        MOV BX, 0006H
        MOV CX, 8
```

```
again:
        ADC AX, BX
        LOOP again

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

Using single stepping record the contents of AX register until CX becomes zero

CX	AX	CX	AX	CX	AX

ASSEMBLER PROGRAM – II (Subtraction)

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV AX, 4000H
        MOV BX, 0006H
        MOV CX, 8
again:
        SBB AX, BX
        LOOP again

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

Using single stepping record the contents of AX register until CX becomes zero

CX	AX	CX	AX	CX	AX

ASSEMBLER PROGRAM – III (Multiplication)

8-bit MULTIPLICATION

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV AX, 0FFH
        MOV CL, 0006H

        MUL CL

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

16-bit MULTIPLICATION

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV AX, 0FFH
        MOV CL, 0006H

        MUL CL

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

Record values of AX, BX, CX & DX before & after execution of MUL instruction.

For 8-bit Multiplication:

Before Execution of MUL:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

After Execution of MUL:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

For 16-bit Multiplication:

Before Execution of MUL:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

After Execution of MUL:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

ASSEMBLER PROGRAM – IV (Division)

8-bit DIVISION:

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV AX, 0400H
        MOV CL, 0006H

        DIV CL

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

16-bit DIVISION:

```
.MODEL SMALL

.STACK 100H

.CODE
MAIN PROC
        MOV DX, 0023H
        MOV AX, 0004H
        MOV CL, 0300H

        DIV CX

        MOV AH, 4CH
        INT 21H

MAIN ENDP

END MAIN
```

OBSERVATIONS

Record values of AX, BX, CX & DX before & after execution of DIV instruction.

For 8-bit Division:

Before Execution of DIV:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

After Execution of DIV:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

For 16-bit Division:

Before Execution of DIV:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

After Execution of DIV:

AX	:	__________	,	BX	:	__________
CX	:	__________	,	DX	:	__________

EXERCISE 1

Write a program, which will add the contents of two 32 bit numbers stored in DX – AX (DX contains the high order word) and memory location WORD PTR [0202] – WORD PTR [0200].

Program

EXERCISE 2

Write a program, which calculate the factorial of any given number (the number may be used as an immediate operand in the instruction). Use any assembler for this exercise.

Program Flowchart

Lab Session 13

OBJECTIVE

De-multiplexing of Address/Data bus of 8088 microprocessor

THEORY

There is 20-bit address bus and 8-bit data bus present on the chip of 8088 microprocessor. Lower 8 bits of address and data buses are time multiplexed with each other. For any machine cycle address comes out of the microprocessor and after some time the bus is used for data transfer between microprocessor and memory or I/O device. In this way the address is not present there for the whole machine cycle on the bus. For holding the address for the full machine cycle we have to design a circuit.

DESIGN OF CIRCUIT

These components will be required for design of the circuit.
1. 8088 microprocessor
2. 74LS373 latches
3. 74LS244 buffers
4. 74LS245 buffers

STEPS OF DESIGNING (Connection description)

1. Connect the lower 8 bits of the time multiplexed address/data (AD0-AD7) bus to the inputs of latch 74LS373. The only address will be available after passing through the latch.
2. The enable pin of the latch 74LS373 will be connected to the ALE pin of the 8088.
3. The only address will be available after passing through the latch.
4. Connect the lower 8 bits of the time multiplexed address/data (AD0-AD7) bus to the inputs of bi-directional buffer 74LS245.
5. The enable pin of the buffer 74LS245 will be connected to the DEN pin of the 8088.
6. The only data will be passed through the buffer in either direction.
7. The DT/R pin of the microprocessor will control the direction of data flow through the bi-directional buffer.
8. Connect the higher 8 bits of the address bus (A8-A15) to the inputs of buffer 74LS244.
9. Connect the next 4 bits (A16-A19) of address bus to the latch 74LS373.
10. Connect the same pins to the inputs of buffer 74LS244 to get the status signals S3, S4, S5 and S6 from 8088.

EXERCISE

Draw the complete de-multiplexed circuit of the given steps.

Lab Session 14

OBJECTIVE

Creating input/output device select pulses using 8088 microprocessor

THEORY

The Microprocessor 8088 has 16-bit register to address I/O devices. Here we have to create device select pulses to select input and output devices. We will use DIP switches as input device and LEDs as output device.

DESIGN OF CIRCUIT

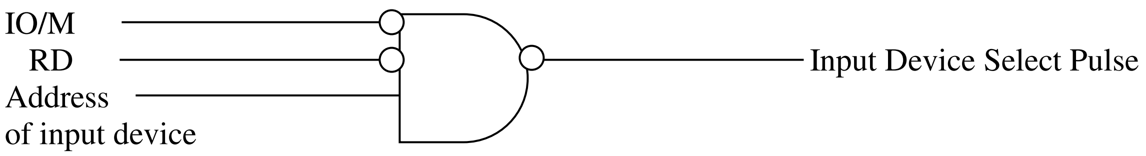
These components will be required for design of the circuit:
1. DIP switches.
2. LEDs.
3. 74LS08 AND gates.
4. 74LS04 hex inverter.
5. 74LS138 line decoder.

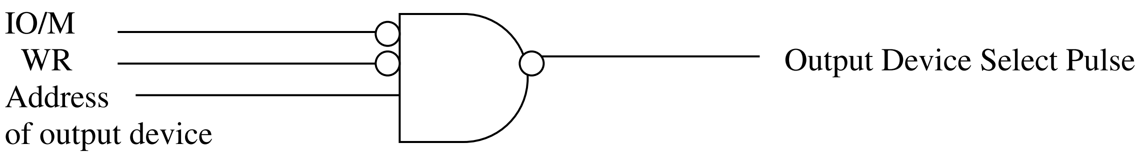
STEPS OF DESIGNING (Connection description)

- For input device selection we have to use IO/M and RD signals and address of the input device to be selected to generate the ***device select pulse***.

- For output device selection we have to use IO/M and WR signals and address of the output device to be selected to generate the ***device select pulse***.

- As IO/M, RD, WR are active low for I/O operations so we will generate the device select pulse in given below manner.





- By using these device select pulse we can select/enable the DIP switches or LEDs according to the need.

OR

- By using 74138 line-decoder we can generate the device select pulses for I/O devices.

EXERCISE

Implement the circuit to generate device select pulses using 74138 line-decoder.

Lab Session 15

OBJECTIVE

Interfacing 8255PPI to the 8088 Microprocessor

THEORY

There are three different ports (Port A, Port B and Port C) are available to interface I/O devices to 8088 microprocessor. There is an internal register, which stores Command Word so we can call it Command register. Command Word defines the modes of working of ports of the device. There are three different types of modes present in 8255 to interface I/O devices to 8088 microprocessor.

Mode 1 : Simple I/O.
Mode 2 : Strobed I/O.
Mode 3 : Handshake I/O.

There are two pins A_0 and A_1 present on the package of 8255PPI to select the ports.

A_1	A_0	Select
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Command Register

First of all the Command Register is selected and the Command Word is stored in the register. After that we can use the ports of 8255PPI according to the function that we have defined in the Command Word.

DESIGN OF CIRCUIT

These components will be required for design of the circuit.
1. 8088 microprocessor.
2. 8255 Programmable Peripheral Interface.
3. DIP switches.
4. LEDs.
5. 74LS373 latches.
6. 74LS244 buffers.
7. 74LS245 buffers.
8. 74LS04 hex inverter.
9. Small capacity RAM IC (e.g. 4016).
10. Small capacity EPROM IC (e.g. 2716).
11. 74LS138 line decoder.

STEPS OF DESIGNING (Connection description)

1. Connect the lower 8 bits of the time multiplexed address/data (AD0-AD7) bus to the inputs of latch 74LS373. The only address will be available after passing through the latch.
2. The enable pin of the latch 74LS373 will be connected to the ALE pin of the 8088.
3. The only address will be available after passing through the latch.
4. Connect the lower 8 bits of the time multiplexed address/data (AD0-AD7) bus to the inputs of bi-directional buffer 74LS245.
5. The enable pin of the buffer 74LS245 will be connected to the DEN pin of the 8088.
6. The only data will be pass through the buffer in either direction.
7. The DT/R pin of the microprocessor will control the direction of data flow through the bi-directional buffer.
8. Connect the higher 8 bits of the address bus (A8-A15) to the inputs of buffer 74LS244.
9. Connect the next 4 bits (A16-A19) of address bus to the latch 74LS373.
10. Connect the same pins to the inputs of buffer 74LS244 to get the status signals S3, S4, S5 and S6 from 8088.
11. Define the addresses for selecting 8255PPI, RAM and EPROM ICs.
12. Connect three address pins to the inputs (A, B and C) of 74138 decoder.
13. Connect the enable pins of the decoder 74138 to appropriate address lines.
14. Connect the data bus of microprocessor to the data bus of 8255PPI.
15. A_0 and A_1 pins of 8255PPI will be connected to A_0 and A_1 pins of 8088 microprocessor respectively.
16. CS (Chip Select) pin of 8255PPI will be connected to one of the outputs of 74138 decoder.
17. RESET of 8255PPI will be connected to RESET of 8088 microprocessor.
18. RD and WR pins of 8255PPI will be connected to the IORC and IOWC pins of 8088 microprocessor respectively.
19. Connect the address and data buses of EPROM and RAM to the address and data buses of 8088 microprocessor.
20. CE or CS pin of EPROM and RAM will be connected to one of the outputs of the 74138 decoder.
21. OE pin of the EPROM and RAM will be connected to the RD pin of the microprocessor.