Contents

Book	Chap	Title	Starting Page
Douglas Hall	1	Computer Number systems, codes and digital devices	
Douglas Hall	2	Computers, Micro-computers and microprocessor -introduction	
Marutt	1	Microcomputer system	
Marutt	2	Representation of Number and characters	
Marutt	3	Organization of IBM Personal Computers	
Marutt	4	Introduction to Assembly language	
Marutt	5	Processor status and flag registers	
Marutt	6	Flow control instructions	
Marutt	7	Logic, shift, rotate instruction	
Marutt	8	Stack and introduction to procedures	
Marutt	10	Arrays and addressing	
Marutt	11	String Instructions and Macros	
		Microprocessor Configuration	
		8085 microprocessor	
		8086 microprocessor timing diagram	
		Bus Timing Diagram	
		Tri-State Logic	
		8086 interrupt system	
		Interfacing	
		DMA with Diagram	
		Instruction Sets	
		Labs	
		Previous Year Questions-Answers	

Computer Number systems, codes and digital devices

Microcomputer System

The Components of a Microcomputer System =>
Introduction :

- Keyboard, display screen, and disk drives are examples of input/output (1/0) devices.These devices are also known as peripheral devices or peripherals.
- Integrated-circuit (IC) chips contain hundreds or thousands of transistors used in computer circuits.IC circuits operate on discrete voltage signal levels represented by high and low voltages, typically denoted by the symbols 1 and 0, respectively.These symbols are called binary digits or bits.
- All information processed by the computer is represented by bit strings consisting of 1s and 0s.
- Computer circuits are divided into three main parts: the central processing unit (CPU), memory circuits, and input/output (1/0) circuits.
- In a microcomputer, these three parts are typically integrated onto a single microchip called a microprocessor.The CPU is responsible for controlling all operations in the computer and acts as the "brain" of the system. The memory circuits are used to store information that the CPU needs to perform its operations. The 1/0 circuits are used to communicate with input/output devices such as keyboards, displays, and disk drives.

<u>The System Board :</u>

- The main circuit board inside the system unit is called the system board, or motherboard.
- The system board contains the microprocessor and memory circuits, as well as expansion slots.
- Expansion slots are connectors on the motherboard used for additional circuit boards called add-in boards or add-in cards.
- Input/output (1/0) circuits are often located on add-in cards, which can be added or removed to provide additional functionality to the computer.
- The system board and add-in cards work together to provide the necessary components and functionality for the computer system.

Memory =>
<u>Bytes and Words :</u>

- Information processed by the computer is stored in its memory, which is made up of memory circuit elements that can store one bit of data each.
- Memory circuits are typically organized into groups of eight bits, called bytes, and each byte is identified by a unique number called its address.
- The contents of a memory byte are referred to as its bytes or data values.

- The address of a memory byte is fixed and unique to that byte, while the contents are subject to change as they represent the data currently stored.
- The number of bits used in an address varies depending on the processor, with some using 20-bit addresses and others using 24-bit addresses.
- The number of bits used in the address determines the number of bytes that can be accessed by the processor. For example, the Intel 8086 microprocessor assigns a 20-bit address, and the Intel 80286 microprocessor uses a 24~bit address

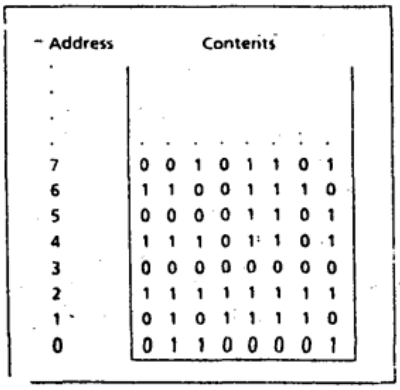


Figure 1.3 Memory Represented as Bytes

Example 1.1 Suppose a processor uses 20 bits for an address. How many memory bytes can be accessed?

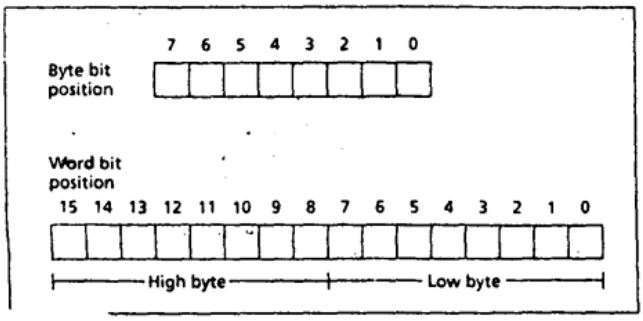
Solution: A bit can have two possible values, so in a 20-bit address there can be 2^{20} = 1,048,576 different values, with each value being the potential address of a memory byte. In computer terminology, the number 2^{20} is called *1* **mega**. Thus, a 20-bit address can be used to address **1 megabyte** or **1 MB**.

- In a typical microcomputer, two bytes form a word.The IBM PC allows any pair of successive memory bytes to be treated as a single unit called a memory word. The lower address of the two memory bytes is used as the address of the memory word.For example, the memory word with the address 2 is made up of the memory bytes with addresses 2 and 3. This allows for more efficient storage and retrieval of data in the computer's memory.

Bit position

- The bit positions are numbered from right to left, starting with 0.
- In a word, the bits 0 to 7 form the low byte and the bits 8 to 15 form the high byte.
- For a word stored in memory, its low byte comes from the memory byte with the lower address, and its high byte comes from the memory byte with the higher address.
- This organization of bits and bytes is important for manipulating data stored in memory, as well as for transferring data between different devices and systems.

Figure 1.4 Bit Positions in a Byte and a Word



Memory Position

- The processor can perform two operations on memory: read and write.
- In a read operation, the processor gets a copy of the data from a memory location without changing its original contents.
- In a write operation, the data written becomes the new contents of the memory location, replacing the original contents.

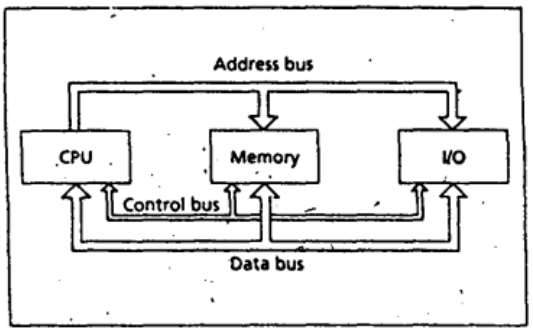
RAM and ROM

- There are two types of memory circuits: random access memory (RAM) and read-only memory (ROM).
- RAM locations can be both read and written, while ROM locations can only be read.
- The contents of RAM memory are lost when the power is turned off, so anything valuable in RAM must be saved on a disk or printed out beforehand.
- ROM circuits retain their values even when the power is off, and are used by computer manufacturers to store system programs, known as firmware.
- Firmware is responsible for loading the operating system from disk as well as for self-testing the computer when it is turned on.

Buses

- A processor communicates with memory and I/O circuits using signals that travel along a set of wires or connections called buses.
- There are three types of signals: address, data, and control.
- There are three types of buses: address bus, data bus, and control bus.
- The processor places the address of the memory location on the address bus to read its contents and receives the data on the data bus.
- A control signal is required to inform the memory to perform a read operation, and it is sent by the CPU on the control bus.

Figure 1.5 Bus Connections of a Microcomputer



The CPU =>

Introduction

- Each institution that the CPU executes Is a bit string (for the Intel 8086, instructions are from one to six bytes long). This language of O's and 1 's is called machine language.
- The instructions executed by a CPU are called its instruction set.
- Machine language instructions are designed to be simple to keep the cost of computers down.
- The Intel 8086 microprocessor will be used as an example of a CPU.
- The Intel 8086 microprocessor has two main components: the execution unit and the bus interface unit.

Execution Unit :

- The Execution Unit (EU) is a component of the CPU that executes instructions.
- It contains the Arithmetic and Logic Unit (ALU) that can perform arithmetic (+, -, x, /) and logic (AND, OR, NOT) operations.
- The EU has eight registers for storing data: AX, BX, CX, DX, SI, DI, BP, and SP.
- Registers are similar to memory locations, but they are referred to by name rather than number.
- The EU also has temporary registers for holding operands for the ALU.
- The FLAGS register contains individual bits that reflect the result of a computation.

Bus Interface Unit

- The Bus Interface Unit (BIU) facilitates communication between the EU and memory or I/O circuits.
- The BIU transmits addresses, data, and control signals on the buses.
- The BIU registers are named CS, DS, ES, SS, and IP, which hold addresses of memory locations.
- The IP (Instruction Pointer) contains the address of the next instruction to be executed by the EU.
- The BIU and the EU are connected by an internal bus and work together.

- The BIU fetches up to six bytes of the next instruction and places them in the instruction queue while the EU is executing an instruction. This operation is called instruction prefetch.
- The BIU suspends instruction prefetch and performs the necessary operations if the EU needs to communicate with memory or peripherals.

I/O Ports :

- 1/0 devices are connected to the computer through 1/0 circuits.
- Each 1/0 circuit contains several registers called 1/0 ports.
- Some 1/0 ports are used for data while others are used for control commands.
- I/O ports have addresses and are connected to the bus system.
- These addresses are known as I/O addresses and can only be used in input or output instructions.
- 1/0 ports function as transfer points between the CPU and 1/0 devices.
- Data to be input from a 1/0 device are sent to a port where they can be read by the CPU.
- On output, the CPU writes data to an 1/0 port and the 1/0 circuit transmits the data to the 1/0 device.

Serial and Parallel Ports :

- Data transfer between I/O port and device can be serial or parallel
- Serial transfer happens 1 bit at a time, while parallel transfer happens 8 or 16 bits at a time
- Parallel ports require more wiring connections, while serial ports tend to be slower
- Slow devices like keyboards usually connect to serial ports, and fast devices like disk drives usually connect to parallel ports
- Some devices like printers can connect to either serial or parallel ports.

Instruction Execution :

A machine instruction has two parts: an opcode and operands.
The opcode specifies the type of operation, and the operands are often given as memory addresses to the data to be operated on.
The CPU goes through the following steps to execute a machine instruction (the fetch-execute cycle):
Fetch

1. Fetch an instruction from memory.
2. Decode the Instruction to determine the operation.
3. · Fetch data from memory if necessary

Execute

4. Perform the operation on the data.
5.. Store the result in memory if needed.

Explanation of each steps :

To see what this entails, let's trace through the execution of a typical machine language instruction for the 8086. Suppose we look at the instruction that adds the contents of register AX to the contents of the memory word at address 0. The CPU actually adds the two numbers in the ALU and then stores the result back to memory word O. The machine code is 00000001 00000110 00000000 00000000 Before execution, we assume that the first byte of the Instruction is stored at the location indicated by the IP.

1. Fetch: The instruction is fetched from memory by the Bus Interface Unit (BIU), which places a memory read request on the control bus and the address of the instruction on the address bus. The memory responds by sending the instruction code over the data bus. The CPU accepts the data and updates the instruction pointer (IP) to point to the next instruction.
2. Decode: A decoder circuit in the Execution Unit (EU) decodes the instruction and determines the operation to be performed.
3. Fetch data from memory: If necessary, the EU requests data from memory by informing the BIU to get the contents of the specified memory address. The BIU sends the address over the address bus and a memory read request over the control bus. The data is sent back over the data bus and placed in a holding register.
4. Execute/Perform Operation: The contents of the holding register and any necessary CPU registers are sent to the Arithmetic Logic Unit (ALU) circuit, which performs the required operation and holds the result.
5. Store the result: If necessary, the EU directs the BIU to store the result in memory. The BIU sends a memory write request over the control bus, the address to be written over the address bus, and the result over the data bus. The previous contents of the memory word at that address are overwritten by the result.

The cycle is repeated for the next instruction, whose address is contained in the IP.

Timing :

1. The execution of machine instructions is complex and requires order to be carried out properly.
2. A clock circuit controls the processor by generating clock pulses at a regular interval, known as the clock period.
3. The clock rate or clock speed is the number of clock pulses per second, measured in megahertz (MHz).
4. Each computer circuit is activated by the clock pulses, and operations are performed only when a clock pulse is present.
5. Each step in the instruction fetch and execution cycle requires one or more clock periods.

6. The clock speed can be increased to make a processor operate faster, but each processor has a rated maximum clock speed beyond which it may not function properly.

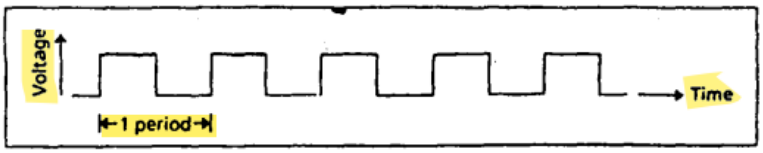


Figure 1.7 Train of Clock Pulses

I/O Devices =>

The primary 1/0 devices are magnetic disks, the keyboard, the display monitor, and the printer

Magnetic Disk

- Magnetic disks are used for permanent storage of programs and data.
- There are two types of magnetic disks: floppy disks and hard disks.
- Floppy disks come in two sizes: 5.25-inch and 3.5-inch. They are portable and can store from 360 kilobytes to 1.44 megabytes of data.
- Hard disks are enclosed in a container that is not removable from the computer and can store much more data than floppy disks, typically from 20 gigabytes to over 100 gigabytes.
- Programs can access information on a hard disk much faster than on a floppy disk.

Keyboard :

- The keyboard is used to input information into the computer.
- It has keys that are similar to those found on a typewriter, as well as control and function keys.
- The keyboard has its own microprocessor that sends a coded signal to the computer when a key is pressed or released.
- When a key is pressed, the corresponding key character is displayed on the screen, but there is no direct connection between the keyboard and the screen.
- The data from the keyboard is received by the current running program, which must send the data to the screen before a character is displayed.

Monitor :

- Display monitor is the standard output device of the computer.
- The information displayed on the screen is generated by a circuit in the computer called a video adapter.
- Most adapters can generate both text characters and graphics images.
- Some monitors are capable of displaying in color.

Printers

- Printers provide more permanent output compared to display monitors
- Printer outputs are known as hardcopies

- Three common kinds of printers are daisy wheel, dot matrix, and inkjet printers
- Daisy wheel printers produce output similar to a typewriter
- Dot matrix printers print characters composed of dots and can generate near-letter-quality printing
- Dot matrix printers can print characters with different fonts and graphics
- Laser printers also print characters composed of dots but have high resolution (300 dots per inch) resulting in typewriter quality printing
- Laser printers are expensive but indispensable in the field of desktop publishing
- Laser printers are quieter compared to other printers.

Programming Languages =>

Machine Language

A CPU can only execute machine language instructions. As we've seen, they are bit strings. The following is a short machine language program for the IBM PC:

Machine instruction	*Operation*
10100001 00000000 00000000	Fetch the contents of memory word 0 and put it in register AX.
00000101 00000100 00000000	Add 4 to AX.
10100011 00000000 00000000	Store the contents of AX in memory word 0.

As you can well imagine, writing programs in machine language is tedious and subject to error!

Assembly Language :

Assembly language instruction	*Comment*
MOV AX,A	;fetch the contents of ;location A and ;put it in register AX
ADD AX,4	;add 4 to AX
MOV A,AX	;move the contents of AX ;into location A

A program written in assembly language must be converted to machine language before the CPU can execute it. A program called the **assembler** translates each assembly language statement into a single machine language instruction.

High Level Languages

- Assembly language is easier than machine language, but still difficult due to the primitive instruction set.
- High-level programming languages, such as FORTRAN and C, were developed to make programming easier and more readable for humans.
- Different high-level languages are designed for different applications and have syntax that resembles natural language text.
- A compiler is needed to translate a high-level language program into machine code.

- Compilation is more complex than assembly because it involves translating complex mathematical expressions and natural language commands into simple machine operations.
- A single high-level language statement typically translates into multiple machine language instructions.

Advantages Of High Level Languages

- High-level languages are closer to natural languages, making it easier to convert a natural language algorithm to a high-level language program.
- High-level language programs are easier to read and understand than assembly language programs.
- Assembly language programs generally contain more statements than equivalent high-level language programs, requiring more time to code.
- Assembly language programs are limited to one machine due to each computer having its own unique assembly language, while high-level language programs can be executed on any machine with a compiler for that language.

Disadvantages Of High Level Languages

1. Slower execution time: High-level languages are generally slower than assembly language because they require more time to translate into machine code.
2. Larger memory footprint: High-level languages often require more memory than assembly language because of their syntax and built-in libraries.
3. Limited low-level control: High-level languages abstract away many of the low-level details of the computer, which can make it difficult to write code that interacts directly with hardware or memory.
4. Learning curve: High-level languages can have a steeper learning curve than assembly language
5. Compatibility issues: High-level languages may not be compatible with all systems or architectures, which can limit their usefulness in certain environments. For example, some high-level languages may not be compatible with embedded systems or real-time applications.

Advantages Of Assembly Language

- Assembly language programs are written for efficiency, as they produce faster and shorter machine language programs.
- Some operations can be easily done in assembly language but may be impossible at a higher level.
- High-level languages can accept subprograms written in assembly language, allowing crucial parts of a program to be written in assembly language.
- Learning assembly language can help understand how the computer works and why certain things happen the way they do.
- High-level languages tend to obscure the details of the compiled machine language program that the computer actually executes.

- Disadvantages Of Assembly Language

1. Assembly language programs are typically longer and more complex than equivalent programs written in high-level languages.
2. Assembly language is machine-dependent, which means that programs written in assembly language are specific to the particular hardware architecture for which they were written.
3. Assembly language is less expressive than high-level languages. Writing complex algorithms in assembly language can be a difficult and time-consuming process.
4. Assembly language programming requires a deep understanding of computer architecture and low-level programming concepts.
5. Assembly language programs are less portable and less modular than high-level language programs.
6. Assembly language programs are less readable and maintainable than programs written in high-level languages.

Assembly Vs High Level Language

Aspects	Assembly Language	High-Level Language
Syntax	Syntax is low-level, consisting of basic commands and memory locations	Syntax is high-level, resembling natural language and complex expressions
Abstraction	Provides a low-level of abstraction from hardware	Provides a higher level of abstraction from hardware
Complexity	Writing an assembly language program is complex and requires more effort	Writing a high-level language program is comparatively easier and requires less effort
Readability	Assembly language programs are harder to read and understand	High-level language programs are easier to read and understand
Portability	Assembly language programs are machine-specific	High-level language programs are generally portable and can be executed on different platforms with little to no modification
Efficiency	Assembly language programs are more efficient and faster in execution	High-level language programs are slower in execution and less efficient

Microprocessor vs Microcontroller

Parameter	Microprocessor	Microcontroller
Definition	Microprocessors can be understood as the heart of a computer system.	Microcontrollers can be understood as the heart of an embedded system.
What is it?	A microprocessor is a processor where the memory and I/O component are connected externally.	A microcontroller is a controlling device wherein the memory and I/O output component are present internally.
Circuit complexity	The circuit is complex due to external connection.	Microcontrollers are present on chip memory. The circuit is less complex.
Memory and I/O components	The memory and I/O components are to be connected externally.	The memory and I/O components are available.
Compact system compatibility	Microprocessors can't be used in compact system.	Microcontrollers can be used with a compact system.
Efficiency	Microprocessors are not efficient.	Microcontrollers are efficient.
Zero status flag	Microprocessors have a zero status flag.	Microcontroller doesn't have a zero status flag.
Number of registers	Microprocessors have less number of registers.	Microcontrollers have more number of registers.
Applications	Microprocessors are generally used in personal computers.	Microcontrollers are generally used in washing machines, and air conditioners.

Microprocessor vs Microcontroller: Key Difference

- Microprocessor consists of only a Central Processing Unit, whereas Micro Controller contains a CPU, Memory, I/O all integrated into one chip.
- Microprocessor is used in Personal Computers whereas Micro Controller is used in an embedded system.
- Microprocessor uses an external bus to interface to RAM, ROM, and other peripherals, on the other hand, Microcontroller uses an internal controlling bus.
- Microprocessors are based on Von Neumann model Micro controllers are based on Harvard architecture
- Microprocessor is complicated and expensive, with a large number of instructions to process but Microcontroller is inexpensive and straightforward with fewer instructions to process.

Which is Better Microcontroller or Microprocessor?

Both of these processes are good. However, which one you should use depends upon your requirements. Microcontrollers are mainly used for small applications like washing machines, Cameras, Security alarms, Keyboard controllers, etc., Whereas Microprocessor is used in Personal Computers, Complex industrial controllers, Traffic light, Defense systems, etc.

Which is Faster Microprocessor or Microcontroller?

Microprocessors are much faster than microcontrollers. The clock speed of a microprocessor is above 1 GHz. While in the case of the Microcontroller, the clock speed is 200MHz or more, depending on the architecture.

Assembly Language Code Inspection :

- Assembly language programs consist of statements, which are either instructions or directives for the assembler.For example, .MODEL SMALL is an assembler directive that specifies the size of the progr.IWi. MOV AX,A is an Instruction. Anything that follows a semicolon Is a comment, and Is ignored by the assembler.
- Comments in assembly language programs are ignored by the assembler and are used to provide information to the programmer.
- Programs are divided into three segments: the stack segment, the data segment, and the code segment.
- The stack segment is used for temporary storage, and it must be declared even if the program doesn't use a stack.
- Variables are declared and initialized in the data segment using directives like DW (define word).
- Instructions are placed in the code segment and are organized into procedures.
- Procedures are defined using the PROC and ENDP directives.
- The main procedure is a special procedure that initializes the DS register and returns to the DOS operating system.

=> Define bus, address bus, control and address bus in 8086

1. Bus: A bus is a set of physical lines (wires) used for communication between different components of a computer system. In the 8086 microprocessor, three types of buses are used: the data bus, the address bus, and the control bus.
2. Address Bus: The address bus is a set of physical lines used to specify the memory location or input/output port that the microprocessor wants to read from or write to. In the 8086 microprocessor, the address bus is 20 bits wide, allowing the processor to address up to 2^20 (1,048,576) memory locations or ports.
3. Data Bus: The data bus is a set of physical lines used to transfer data between the microprocessor and memory or input/output devices. In the 8086 microprocessor, the data bus is 16 bits wide, allowing the processor to transfer 2 bytes (16 bits) of data at a time.
4. Control Bus: The control bus is a set of physical lines used to control the flow of data and signals between the microprocessor and other components of the computer system. In the 8086 microprocessor, the control bus includes several signals such as the read/write signal, the memory enable signal, the input/output enable signal, and others. These signals control the flow of data between the microprocessor and memory or input/output devices.

instruction pointer, IP	**A CPU register that contains the address of the next instruction**
instruction set	**The instructions the CPU is capable of performing**

Exercise

1. Suppose memory bytes 0–4 have the following contents:

Address	*Contents*
0	01101010
1	11011101
2	00010001
3	11111111
4	01010101

a. Assuming that a word is 2 bytes, what are the contents of

- the memory word at address 2? • 1111111100010001b
- the memory word at address 3? • 0101010111111111b
- the memory word whose high byte is the byte at address 2? • 0001000111011101b

b. What is

- bit 7 of byte 2? • 0
- bit 0 of word 3? • 1
- bit 4 of byte 2? • 1
- bit 11 of word 2? • 1

2. A **nibble** is four bits. Each byte is composed of a high nibble and a low nibble, similar to the high and low bytes of a word. Using the data in exercise 1, give the contents of

   a. the low nibble of byte 1. 1101
   b. the high nibble of byte 4. 0101

a) 1101
b) 0101

3. The two kinds of memory are RAM and ROM. Which kind of memory
   a. holds a user's program? RAM
   b. holds the program used to start the machine? RAM
   c. can be changed by the user?
   d. retains its contents, even when the power is turned off? ROM

a) RAM
b) ROM
c) RAM
d) ROM

4. What is the function of
   a. the microprocessor? integration between CPU, memory and I/O
   b. the buses? channel of wires to send data between components

a) *the microprocessor* :
brain of the computer , it controls the computer by executing programs stored in memory.

b) *the buses*:
connect the different component to make processor communicate with memory and I/O circuits.

5. The two parts of the microprocessor are the EU and the BIU.
   a. What is the function of the EU?
   b. What is the function of the BIU?

a) *EU (Execution Unit)* :
execute instructions – because it contains the ALU.

b) *BIU (Bus Interface Unit)* :
facilitates communication between the EU and the memory or I/O circuits , and useful for speed-up the processor by instruction prefetch .

6. In the microprocessor, what is the function of
   a. the IP? the next instruction to be executed
   b. the ALU?

a) *IP* : contains the address of the next instruction to be executed by EU .

b) *ALU* : perform arithmetic & logic operation .

7. a. What are the I/O ports used for?
   b. How are they different from memory locations?

a) *I/O ports* : transfer points between the CPU & I/O devices

b) because they have addresses connected to the bus system known as I/O addresses and they can only be used in input or output instruction , this allows the CPU to distinguish between them .

8. What is the maximum length (in bytes) of an instruction for the 8086-based IBM PC? 4-Bytes

9. Consider a machine language instruction that moves a copy of the contents of register AX in the CPU to a memory word. What happens during
   a. the fetch cycle?
   b. the execute cycle?

a)
- Fetch an instruction from memory.
- Decode the instruction to determine the operation.

b)
- Perform the operation if needed .
- Store the results in memory.

10. Give
    a. three advantages of high-level language programming.
    b. the primary advantage of assembly language programming.

a) *Advantages of H.L.L:*
1. closer to natural language , so it's easier to read and understand the program .
2. assembly contains more statement than an equivalent HLL ,so It's need more time to code the assembly language program.

3. each computer has its own unique assembly language , so It's limited to one machine ,but HLL can be executed on any machine has the compiler of that language .

b) *primary advantage of assembly language*:

It's so close to machine language ,so it's produce a faster shorter machine language program .
And some operation (read /write) can be done on specific Memory location and I/O ports, and that may be impossible In HLL

Representation of Number and characters

1 Kilobyte (1 KB) = 1024 = 400h
64 Kilobytes (64 KB) = 65536 = 10000h
1 Megabyte (1 MB) = 1,048,576 = 100000h

#What's a positional number system ?
A positional number system is a mathematical system for representing numbers using a fixed set of symbols, where the value of each symbol depends on its position within the number.

Table 2.2 Decimal, Binary, and Hexadecimal Numbers

Decimal	*Binary*	*Hexadecimal*
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Special Control Characters

Dec	Hex	Char	Meaning
7	07	BEL	bell
8	08	BS	backspace
9	09	HT	horizontal tab
10	0A	LF	line feed
12	0C	FF	form feed
13	0D	CR	carriage return

#How Integers Are Represented in the Computer ?

The hardware of a computer necessarily restricts th" 51.ZC' of numbers that can be stored in a register or memory location. The most significant bit, or msb, is the leftmost bit. In a word, the msb is bit 15; in a byte, it is bit 7. Similarly, the least significant bit, or lsb, is the rightmost bit; that is, Oit 0 .

- An unsigned integer is an integer that represents a magnitude, so it is never negative. Use of it are : such as addresses of memory locations, counters, and ASCII character codes
- In signed integers, The most significant bit is reserved for the sign: 1 means negative and 0 means positive,significant bit is reserved for the sign: 1 means negative and 0 means positive. Negative integers are stored in the computer in a special way known as two's complement
- The advantage ol two's complement representation of negative integers·rs in the computer " that subtraction can be done by bit complement and addition And the circuit that can add and complement bits is easy to design.

#Decimal interpretation :

- *Unsigned decimal interpretation:* Just do a binary-to-decimal conversion. It's usually easier to convert binary to hex first, and then convert hex to decimal.
- *Signed decimal interpretation:* If the most significant bit is 0, the number is positive, and the signed decimal is the same as the unsigned decimal. If the msb is 1, the number is negative, so call it $-N$. To find N, just take the twos' complement and then convert to decimal as before.

#ASCII Code :

- ASCII (American Standard Code for Information Interchange) is the most popular encoding scheme for characters.

- Originally used in communications by teletype, ASCII code is now used by all personal computers today.
- ASCII code uses 7 bits to code each character, resulting in a total of 128 ASCII codes.
- Of these, only 95 codes (from 32 to 126) are considered to be printable characters, while the codes 0 to 31 and 127 are used for communication control purposes and do not produce printable characters.
- Most microcomputers use only the printable characters and a few control characters such as LF, CR, BS, and Bell.
- Each ASCII character is coded by only seven bits, so the code of a single character fits into a byte, with the most significant bit set to zero.
- The printable characters can be displayed on the video monitor or printed by the printer, while the control characters are used to control the operations of these devices.
- To display a character on the screen, a program sends the corresponding ASCII code to the screen, and to control the operations of the device, control characters such as CR (carriage return) and LF (line feed) are sent.
-

Keyboard :

- Early microcomputers used ASCII keyboards where a key press was identified by its corresponding ASCII code.
- Modern keyboards have many control and function keys in addition to ASCII character keys, so other encoding schemes are used.
- For the IBM PC, each key is assigned a unique number called a scan code.
- When a key is pressed on an IBM PC keyboard, the keyboard sends the key's scan code to the computer.
- The computer's keyboard controller translates the scan code into a code called a virtual key code.
- The virtual key code is then translated into a character code or a control code, depending on the context in which the key was pressed.
- The character or control code is then passed to the application that has the input focus.

#Extra :

- If A and B are stored in integers,-the processors computes A - B by adding the two's complement of B to A

least significant bit, lsb	The rightmost bit in a word or byte; that is, bit 0
most significant bit, msb	The leftmost bit in a word or byte; that is, bit 15 in a word or bit 7 in a byte

- The range of unsigned Integers that can be stored in a byte Is 0-255; in a 16-bit word, if is 0-65535.
    - For signed numbers, the most significant bit is the sign bit; 0 means positive and 1 means negative. The range of signed numbers that can be stored in a byte is –128 to 127; in a word, it is –32768 to 32767.
- The unsigned decimal interpretation of a word is outaim:<l by converting the binary value to decimal. If the sign bit is 0, this is also the signed decimal interpretation. If the sign bit Is l, the signed decimal interpretation may be obtained by subtracting 65536 from the unsigned decimal Interpretation.

#Conversions :

Converting Binary and Hex to Decimal

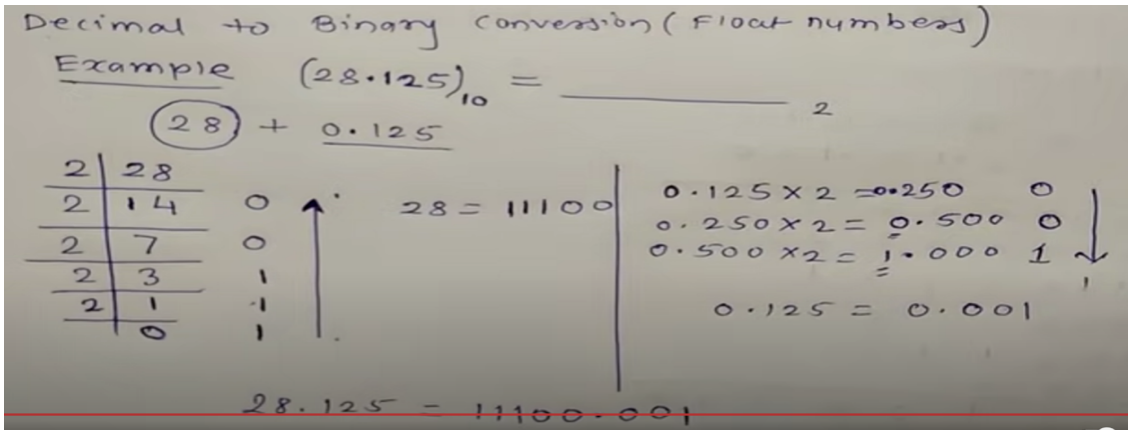
Consider the hex number 82AD. It can be written as

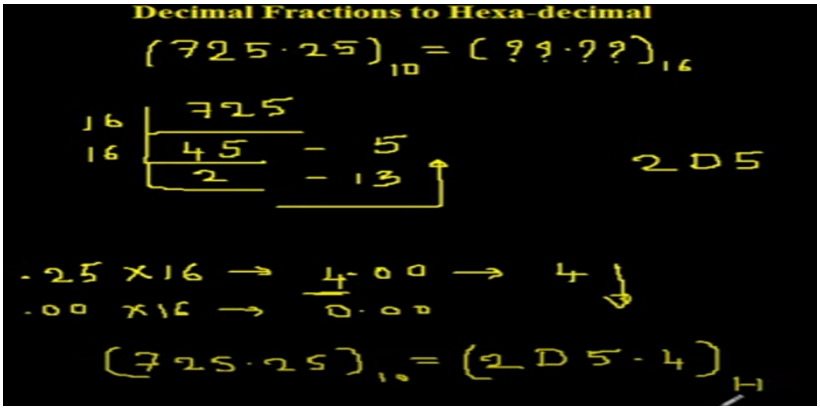
$$8A2Dh = 8 \times 16^3 + A \times 16^2 + 2 \times 16^1 + D \times 16^0$$
$$= 8 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 13 \times 16^0 = 35373d$$

Similarly, the binary number 11101 may be written as

$$11101b = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 29d$$

Converting Decimal to Binary and Hex





Conversion between Hex and Binary

Example 2.4 Convert 2B3Ch to binary.

Solution:

2 B 3 C
= 0010 1011 0011 1100
= 0010101100111100

To go from binary to hex, just reverse this process; that is, group the binary digits in fours starting from the right. Then convert each group to a hex digit.

Example 2.5 Convert 1110101010 to hex.

Solution: 1110101010 = 11 1010 1010 = 3AAh

#Complement

Example 2.6 Find the one's complement of 5 = 0000000000000101.

Solution:

5 = 0000000000000101
One's complement of 5 = 1111111111111010

Example 2.7 Find the two's complement of 5.

Solution: From above,

one's complement of 5 = 1111111111111010
+ 1
two's complement of 5 = 1111111111111011 = FFFBh

Subtraction in Two'.s Complement Addition

Example 2.10 Suppose AX contains 5ABCh and BX contains 21FCh. Find the difference of AX minus BX by using complementation and addition.

Solution:

AX contains 5ABCh = 0101 1010 1011 1100
BX contains 21FCh = 0010 0001 1111 1100
5ABCh = 0101 1010 1011 1100
+ one's complement of 21FCh = 1101 1110 0000 0011
+ 1
Difference = 1 0011 1000 1100 0000 = 38C0h

A one is carried out of the most significant bit and is lost. The answer stored, 38C0h, is correct, as may be verified by hex subtraction.

#Exercise :

2. Convert the following binary and hex numbers to decimal:
   a. 1110
   b. 100101011101
   c. 46Ah
   d. FAE2Ch

a) 14d
b) 2397d
c) 1130d
d) 1027628d

3. Convert the following decimal numbers:
   a. 97 to binary
   b. 627 to binary
   c. 921 to hex
   d. 6120 to hex

a) 1100001b
b) 1001110011b
c) 399h
d) 17E8h

4. Convert the following numbers:

a. 1001011 to hex
b. 1001010110101110 to hex
c. A2Ch to binary
d. B34Dh to binary

a) 4Bh
b) 95AEh
c) 101000101100b
d) 1011001101001101b

5. Perform the following additions:
   a. 100101b + 10111b
   b. 1001111101b + 100011111001b
   c. B23CDh + 17912h
   d. FEFFEh + FBCADh

a) 111100b
b) 10110110110b
c) C9CDFh
d) 1FACABh

6. Perform the following subtractions:
   a. 11011b – 10110b
   b. 10000101b – 111011b
   c. 5FC12h – 3ABD1h
   d. F001Eh – 1FF3Fh

a) 101b
b) 1001010b
c) 25041h
d) D00DFh

7. Give the 16-bit representation of each of the following decimal integers. Write the answer in hex.
   a. 234
   b. -16
   c. 31634
   d. -32216

a) 234d = 0000000011101010b = 00Eah

b) 16d = 00000000010000b
-16d = 2's comp. (16)
= 1111111111110000b = FFF0h

c) 31634d = 0111101110010010b = 7B92h

d) 32216d = 0111110111011000b
-32216d = 2's comp.(32216)
= 1000001000101000b = 8228h
Note: 2's comp. =1's comp +1
1's comp. =NOT

8. Do the following binary and hex subtractions by two's complement addition.

a. 10110100 - 10010111

b. 10001011 - 11110111

c. FE0Fh - 12ABh

d. 1ABCh - B3EAh

Take 2's comp. for negative number ,then Add the two numbers.,if no.1> no.2 : positive result stay as it.
If no.1< no.2 :negative take 2's comp.

a) 2's comp.(-10010111b) = 11111111101101001b
10110100b+11111111101101001b = 00000000000011101b
Psitive no. =11101b

b) 2's comp.(-11110111b) = 1111111100001001b
10001011b+1111111100001001b =1111111110010100b
Negative no. =2's comp.(1111111110010100)
= -1101100b

c) 2's comp.(-12ABh) = ED55h
FE0Fh+ ED55h = EB64h
Positive no. = EB64h

d) 2's comp.(-B3EAh) = 4C16h
1ABCh+ 4C16h = 66D2h
Negative no. = 2's comp.(66D2h)
= - 992Eh

Note : we can juddge on no. from subtraction equation (e.g: 1ABCh – B3EAh = (negative number) : because 1ABC < B3EAh ,so we must take 2's comp.

9. Give the unsigned and signed decimal interpretations of each of the following 16-bit or 8-bit numbers.

a. 7FFEh

b. 8543h

c. FEh

d. 7Fh

Unsigned :convert the number as it is.
Signed :if msb=0 convert the number as it is.
If msb=1 take 2's comp. then convert.

	Unsigned	signed
a) 7FFEh	32766d	32766d
b) 8543h	34115d	-31421d
c) FEh	254d	-2d
d) 7Fh	127d	127d

Note :msb (most significant bit) last bit on the left.
i.e :bit-15 in word OR bit-7 in byte.

10. Show how the decimal integer -120 would be represented

a. in 16 bits.

b. in 8 bits.

120d = 01111000b 8-bit
0000000001111000b 16-bit
To make it negative take 2's comp.
-120d = 10001000b 8-bit
1111111110001000b 16-bit

a) 10001000b
b) 1111111110001000b

11. For each of the following decimal numbers, tell whether it could be stored (a) as a 16-bit number (b) as an 8-bit number.

a. 32767

b. -40000

c. 65536

d. 257

e. -128

	16-bit	8-bit
a) 32767d	111111111111111b	X :too big
b) -40000d	X :too big	X :too big
c) 65536d	X :too big	X :too big
d) 257d	0000000100000001b	X :too big
e) -128	0000000010000000b	10000000b

Note : Range of unsigned integer :
0 – 255 for byte 8-bit
0 – 65535 for word 16-bit
Range of signed numbers :
-128 – 127 for byte 8-bit
-32768 – 32767 for word 16-bit
(msb represente the sign)

12. For each of the following 16-bit signed numbers, tell whether it is positive or negative.

a. 1010010010010100b

b. 78E3h

c. CB33h

d. 807Fh

e. 9AC4h

If msb =0 positive else negative
a) 1010010010010100b Negative
b) 78E3h = 0111100011010011b Positive
c) CB33h = 1100101100110011b Negative
d) 807Fh = 1000000001111111b Negative
e) 9AC4h = 1001101011000100b Negative

13. If the character string "$12.75" is being stored in memory starting at address 0, give the hex contents of bytes 0–5.

Address	Content(hex)	Data
0	24	$
1	31	1
2	32	2
3	2E	.
4	37	7
5	35	5

14. Translate the following secret message, which has been encoded in ASCII as 41 74 74 61 63 6B 20 61 74 20 44 61 77 6E.

41	74	74	61	63	6B	20	61	74	20	44	61	77	6E
A	t	t	a	c	k		a	t		D	a	w	n

"Attack at Dawn"

15. Suppose that a byte contains the ASCII code of an uppercase letter. What hex number should be added to it to convert it to lower case?

A = 41h a = 61h
The difference is 20h
Add 20h to convert from upper to lower case.

16. Suppose that a byte contains the ASCII code of a decimal digit; that is, "0". . . "9." What hex number should be subtracted from the byte to convert it to the numerical form of the characters?

"0" = 30h numerical value = 0h
"1" = 31h numerical value = 1h
The difference is 30h
SUB 30h to get the numerical value.

Just a technique:
0 1 2 3 4 5 6 7 8 9 A B C D E F .....->
5(c) B23CDh
+ 1791 2h
D+2= go from D to the right 2 places = F
C+1= go from C to the right 1 place = D
3+9= go from 3 to the right 9 places = C
2+7= go from 2 to the right 7 places = 9
B+1= go from B to the right 1 place = C

= C9CDFh

17. It is not really necessary to refer to the hex addition table to do addition and subtraction of hex digits. To compute Eh + Ah, for example, first copy the hex digits:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Now starting at Eh, move to the right Ah = 10 places. When you go off the right end of the line, continue on from the left end and attach a 1 to each number you pass:

10 11 12 13 14 15 16 17 18 9 A B C D E F
STOP ^ START ^

You get Eh + Ah = 18h. Subtraction can be done similarly. For example, to compute 15h – Ch, start at 15h and move left Ch = 12 places. When you go off the left end, continue on at the right:

10 11 12 13 14 15 6 7 8 9 A B C D E F
^ START ^ STOP

You get 15h – Ch = 9h.
Rework exercises 5(c) and 6(c) by this method.

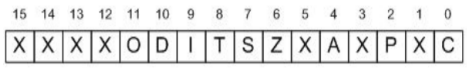
Table of Contents

The Processor Status and the FLAGS Register..... 1
Status Flag..... 2
Overflow Flag and Carry Flag..... 2
Decimal Range..... 3
Unsigned Overflow CF..... 3
Signed Overflow OF..... 3
Types of Overflow..... 3
Examples of Overflow..... 3
Some "How" Q/As..... 4
Examples..... 5
Important Questions (Quiz, Exams, Assignments)..... 8
Exercises From Books (Marut)..... 9

The Processor Status and the FLAGS Register

For the 8086 processor, the processor state is implemented as nine individual bits called flags. The flags are placed in the FLAGS register and they arc classified as either status flags or control flags

The FLAGS Register



An x bit means an unidentified value

Table 5.1 Flag Names and Symbols

Status Flags		
Bit	***Name***	***Symbol***
0	Carry flag	CF
2	Parity flag	PF
4	Auxiliary carry flag	AF
6	Zero flag	ZF
7	Sign flag	SF
11	Overflow flag	OF
Control Flags		
Bit	***Name***	***Symbol***
8	Trap flag	TF
9	Interrupt flag	IF
10	Direction flag	DF

Status Flag

As stated earlier, the processor uses the status flags to reflect the result of an operation. For example, If SUB AX,AX is executed, the zero flag becomes l, thereby Indicating that a zero result was produced. Now let's get to know the status flags.

The Carry Flag (C): This flag is set when the result of an unsigned arithmetic operation is too large to fit in the destination register. CF=1 if a carry from most significant bit (msb) on addition, or a borrow into msb on subtraction; otherwise CF=0.

The Overflow Flag (O): This flag is set when the result of a signed arithmetic operation is too large to fit in the destination register (i.e. when an overflow occurs). Overflow can occur when adding two numbers with the same sign (i.e. both positive or both negative). A value of 1 = overflow and 0 = no overflow.

The Sign Flag (S): This flag is set when the result of an arithmetic or logic operation is negative. This flag is a copy of the MSB of the result (i.e. the sign bit). A value of 1 means negative and 0 = positive.

The Zero Flag (Z): This flag is set when the result of an arithmetic or logic operation is equal to zero. A value of 1 means the result is zero and a value of 0 means the result is not zero.

The Auxiliary Carry Flag (A): This flag is set when an operation causes a carry from bit 3 to bit 4 (or a borrow from bit 4 to bit 3) of an operand. A value of 1 = carry and 0 = no carry.

The Parity Flag (P): This flags reflects the number of 1s in the low byte of a result of an operation. If the number of 1s is even its value = 1 and if the number of 1s is odd then its value = 0.

Overflow Flag and Carry Flag

- Both are indicators of out-of-range condition.
- Overflow flag is used to evaluate an out-of-range condition of signed number operations .
- Carry flag is used in unsigned number operations.
- Since a binary number can represent an unsigned number or signed number, the processor computes both flags and the user checks the appropriate flag to check if the result is out of range or not.

The (decimal) range of signed numbers that can be represented by a 16 bit word is -32768 to 32767; for an 8 bit byte the range is -128 to 127. For unsigned numbers, the range for a word is 0 to 65535; for a byte, it is 0 to 255. If the result of an operation falls outside these ranges, overflow occurs and the truncated result that is saved will be incorrect.

Decimal Range

UNSIGN	SIGN
BYTE [0, 255]	[-128, 127]
WORD [0, 65535]	[-32768, 32767]

Unsigned Overflow CF

Carry Flag CF is set to 1 if there is an end carry in an addition operation or there an end borrow in a subtraction operation. A value of 1 = carry and 0 = no carry.

Signed Overflow OF

- Overflow Flag is set to 1 when adding two numbers the same sign and the result has a different sign. Otherwise, OF is reset to 0.
- Subtraction operation A - B can be performed as operation A + (-B). OF=1 if the result has a different sign than expected
- OF=1 if the carries into and out of the msb don't match

Types of Overflow

Signed and unsigned overflows are independent phenomena. When we perform an arithmetic operation such as addition, there are four possible outcomes:

(1) no overflow,
(2) signed overflow only,
(3) unsigned overflow only, and
(4) both signed and unsigned overflows.

Examples of Overflow

As an example of unsigned overflow but not signed overflow, suppose AX contains FFFFh, BX contains 0001h, and ADD AX,BX is executed. The binary result is-

1111 1111 1111 1111
+ 0000 0000 0000 0001

---

1 0000 0000 0000 0000

If we are giving an unsigned interpretation, the correct ans is 10000h = 65536, but this is out of range for a word operation. A 1 is carried out of the msb and the answer stored in AX, 0000h, is wrong, so unsigned overflow occurred. But the stored answer is correct as a signed number, for FFFFh = -1, 0001h = 1, and FFFFh + 0001h = -1 + 1 = 0, so signed overflow did not occur.

As an example of signed but not unsigned overflow, suppose AX and BX both contain 7FFFh, and we execute ADD AX,BX. The binary result is

0111 1111 1111 1111
+ 0111 1111 1111 1111

1111 1111 1111 1110 = FFFEh

The signed and unsigned decimal interpretation of 7HFh IS 32767. Thus for both signed and unsigned addition, 7FFFh + 7FFFh = 32767 + 32767 = 65534. This is out of range for signed numbers; the signed interpretation of the stored answer FFFEh is -2. so signed overflow occurred. However, the unsigned interpretation of FFFEh is 65534, which is the right answer, so there is no unsigned overflow.

Some "How" Q/As

How the Processor Indicates Overflow ?

The processor sets OF= 1 for signed overflow and CF= 1 for unsigned overflow. It is then up to the program to take appropriate action, and If nothing is done immediately the result of a subsequent instruction may cause the overflow flag to be turned off. ·

How the Processor Determines that Overflow Occurred ?

Many instructions can cause overflow; for simplicity, we'll limit the discussion to addition and subtraction.

How the Processor Determines that Unsigned Overflow Occurred?

- CF = 1
- Addition
  - ▪ There is a carry out of the msb.
  - ▪ The correct answer is larger than the biggest unsigned number (FFFFh and FFh).
- Subtraction
  - ▪ There is a borrow into the msb.
  - ▪ The correct answer is smaller than 0.

How the Processor Determines that Signed Overflow Occurred?

· OF = 1

▪ There is a carry into the msb but no carry out.

▪ There is a carry out but no carry in.

· Addition
▪ The sum has a different sign.
· Subtraction
▪ The result has a different sign than expected.
▪ A – (–B) = A + B
▪ –A – (+B) = –A + –B
· Addition of Numbers with Different Signs
▪ Overflow is impossible.
▪ A + (– B) = A – B

How Instructions Affect the Flags ?

Instruction	Affects Flags
MOV/XCHG	none
ADD/SUB	all
INC/DEC	all except CF
NEG	all (CF = 1 unless result is 0,OF = 1 if word operand is 8000h, or byte operand is 80h)

Examples

NEG

NEG AX where AX contains 8000h.

8000h = 1000 0000 0000 0000
one's complement = 0111 1111 1111 1111
+1
= 1000 0000 0000 0000 = 8000h

SF = 1, PF = 1, ZF = 0.
CF = 1 because for NEG CF is always 1 unless the result is 0.
OF = 1 because the result is 8000h; when a number is negated, we would expect a sign change, but because 8000h is its own two's complement, there is no sign change.

MOV

MOV AX, -5

AX = FFFBh

None of the flags are affected by MOV.

INC

INC AL where AL contains FFh.

	FFh	1111 1111	
+	1h	+ 0000 0001	
1	00h	1 0000 0000	AL = 00h

SF = 0, PF = 1, ZF = 1.

CF is unaffected by INC.

If CF = 0 before the execution of the instruction, CF will still be 0 afterward.

OF = 0 because numbers of unlike sign are being added (there is a carry into the msb and also a carry out).

ADD

ADD AX, BX where AX contains FFFFh and BX contains FFFFh.

	FFFFh	1111 1111 1111 1111	
+	FFFFh	+ 1111 1111 1111 1111	
1	FFFEh	1 1111 1111 1111 1110	AX = FFFEh

SF = 1 because the msb is 1.

PF = 0 because there are 7 (odd number) of 1 bits in the low byte of the result.

ZF = 0 because the result is nonzero.

CF = 1 because there is a carry out of the msb on addition.

OF = 0 because the sign of the stored result is the same as that of the numbers being added (as a binary addition, there is a carry into the msb and also a carry out).

SUB

SUB AX, BX where AX contains 8000h and BX contains 0001h.

could be a good question in exam

	8000h	1000 0000 0000 0000	
−	0001h	− 0000 0000 0000 0001	
	7FFFh	0111 1111 1111 1111	AX = 7FFFh

SF = 0 because the msb is 0.

PF = 1 because there are 8 (even number) one bits in the low byte of the result.

ZF = 0 because the result is nonzero.

CF = 0 because a smaller unsigned number is being subtracted from a larger one.

OF = 1 because in a signed sense we are subtracting a positive number from a negative one, which is like adding two negatives but the result is positive (the wrong sign).

Example

ADD AL,BL

AL = 80h
BL = + 80h
1 00h

CF=1, OF=1, PF=1, ZF=1, SF=0

Example

ADD AX,BX

AX = FFFFh
BX = + FFFFh
1 FFFEh= 1111 1111 1111 1110

CF=1, OF=0, PF=0, ZF=0, SF=1

Example

ADD AX,BX

AX = 7FFFh 0111111111111111
BX = 7FFFh 0111111111111111
AX=AX+BX=FFFEh 0 1111111111111110

CF=0, OF=1, The result and operands have different sign bits. PF=0, ZF=0, SF=1, AF=1

As a signed operation, 7FFFh+ 7FFFh = =32767 + 32767 = 65534 = FFFEh (out-of-range).

As an unsigned operation, 7FFFh+7FFFh =32767 + 32767 = 65534 (ok).

Example

ADD AX,BX

AX = FFFFh 1111111111111111
BX = 0001h 0000000000000001
AX=AX+BX=0000h 1 0000000000000000

CF=1, OF=0,Operands have different sign bits. PF=1, ZF=1, SF=0, AF=1

As a signed operation, FFFFh+ 0001h = 10000h (out-of-range).

As an unsigned operation, FFFFh+0001h=-1+1=0 (ok).

Example

ADD AL,BL

AL = F8h 11111000
BL = 81h 10000001
AL=AL+BL=79h 101111001

CF=1, OF=1, Operands are negatives, the result is positive. PF=0, ZF=0, SF=0, AF=0

As a signed operation, -8 + -127 = -135 < -128 (out-of-range).

As an unsigned operation, 248 + 129 = 377 > 255 (out-of-range).

Example

ADD AL,BL

AL = 4Fh 01001111
BL = 40h 01000000
AL=AL+BL=8Fh 010001111

CF=0, OF=1, The result and operands have different sign bits. PF=0, ZF=0, SF=1, AF=0

As a signed operation, 79 + 64 = 143 > 127 (out-of-range).

As an unsigned operation, 143 < 255 (ok).

Example

ADD AL,BL

AL = 0Fh 00001111
BL = F8h 11111000
AL=AL+BL= 07h 100000111

CF=1, OF=0,Operands have different sign bits. PF=0, ZF=0, SF=0, AF=1

As a signed operation, 15 + (-8) = 7 (ok).

As an unsigned operation, 15 + 248 = 263 > 255 (out-of-range).

Important Questions (Quiz, Exams, Assignments)

1.Perform the indicated operation on the following number

Ans:

0111 1001 BCD
+ 1001 0101 BCD
_____
0001 0110 0001 0100 BCD

2.How do we detect when a signed overflow does occur? Discuss three effective methods in brief.

Ans:

- OF = 1
  - There is a carry into the msb but no carry out.
  - There is a carry out but no carry in.
- Addition
  - The sum has a different sign.
- Subtraction
  - The result has a different sign than expected.
  - A – (–B) = A + B
  - –A – (+B) = –A + –B
- Addition of Numbers with Different Signs
  - Overflow is impossible.
  - A + (– B) = A – B

3.Suppose ADD AX, BX is executed. In each of the following parts, the first number being added is the contents of AX, and the second number is the contents of BX. Give the resulting value of the AX and tell whether signed or unsigned overflow occurred.

a) 512Ch + 4185h
b) 7132h + 7000h

Ans: a)

```
  512Ch
+ 4185h
-----------
  92B1h
```

The resulting value of AX is 92B1h.

To determine whether signed or unsigned overflow occurred, we need to look at the most significant bit (MSB) of the result (bit 15 in this case). Since the MSB is 1, we can tell that unsigned overflow occurred.

b) 7132h
+ 7000h

E132h

The resulting value of AX is E132h.
To determine whether signed or unsigned overflow occurred, we need to look at the MSB of the result (bit 15 in this case). Since the MSB is 1, we can tell that signed overflow occurred.

Exercises From Books (Marut)

1. For each of the following instructions, give the new destination contents and the new settings of CF, SF, TF, PF, and OF. Suppose that the flags are initially 0 in each part of this question.

Ans:

a. ADD AX, BX where AX contains 7FFFh and BX contains 1000h
Ans: New destination contents: AX = 8FFFh
New settings of CF, SF, TF, PF, and OF: CF = 0, SF = 1, TF = 0, PF = 0, OF = 0

b. SUB AL, BL where AL contains 01h and BL contains FFh
Ans: New destination contents: AL = 02h
New settings of CF, SF, TF, PF, and OF: CF = 0, SF = 0, TF = 0, PF= 0, OF = 0

c. DEC AL where AL contains 00h
Ans: New destination contents: AL = FFh
New settings of CF, SF, TF, PF, and OF: CF = 1, SF = 1, TF = 0, PF = 1, OF = 0

d. NEG AL where AL contains 7Fh
Ans: New destination contents: AL = 81h
New settings of CF, SF, TF, PF, and OF: CF = 1, SF = 1, TF = 0, PF = 0, OF = 1

e. XCHG AX, BX where AX contains 1ABCh and BX contains 712Ah
Ans: New destination contents: AX = 712Ah, BX = 1ABCh
New settings of CF, SF, TF, PF, and OF: CF = 0, SF = 0, TF = 0, PF = 0, OF = 0

f. ADD AL, BL where AL contains 80h and BL contains FFh
Ans: New destination contents: AL = 7Fh
New settings of CF, SF, TF, PF, and OF: CF = 1, SF = 1, TF = 0, PF = 0, OF = 0

g. SUB AX, BX where AX contains 0000h and BX contains 8000h
Ans: New destination contents: AX = 8000h
New settings of CF, SF, TF, PF, and OF: CF = 1, SF = 1, TF = 0, PF = 1, OF = 0

h. NEG AX where AX contains 0001h
Ans: New destination contents: AX = FFFFh
New settings of CF, SF, TF, PF, and OF: CF = 1, SF = 1, TF = 0, PF = 1, OF = 1

Or

(1)

Assuming that the flags are initially = 0 *

Content of	CF	SF	ZF	PF	OF
a) AX= 8000h	0	1	0	1	1
b) AL= 02h	1	0	0	0	0
c) AL= FFh	0	1	0	1	0
d) AL= 81h	1	1	0	1	0
e) AX= 712Ah BX= 1ABCh	0	0	0	0	0
f) AL= 7Fh	1	0	0	0	1
g) AX= 8000h	1	1	0	1	1
h) AX= FFFFh	1	1	0	1	0

2. a. Suppose that AX and BX both contain positive numbers and ADD AX,BX is executed. Show that there is a carry into the msb but no carry out of the msb if, and only if, signed overflow occurs.

b. Suppose AX and BX both contain negative numbers, and ADD AX,BX is executed. Show that there is a carry out of the msb but no carry into the msb if, and only if, signed overflow occurs.

Ans:

a. If AX and BX both contain positive numbers, then the msb of both AX and BX is 0 (since both numbers are positive). When these numbers are added, a carry into the msb occurs if and only if the sum of the msbs of AX and BX is 1. This is because the addition of two bits results in a carry if and only if both bits are 1.

If there is a carry into the msb, then the resulting msb of the sum will be 0 (since the sum of two 1s and a carry is 0 with a carry out). If there is no carry into the msb, then the resulting msb of the sum will be 1 (since the sum of two 0s and a carry is 1 with no carry out).

Therefore, signed overflow occurs if and only if there is a carry into the msb but no carry out of the msb. This is because a carry into the msb indicates that the result is too large to be represented as a signed number (i.e., it is positive, but the msb is 1), while the lack of a carry out of the msb indicates that the result is too small to be represented as a signed number (i.e., it is negative, but the msb is 0).

b. If AX and BX both contain negative numbers, then the msb of both AX and BX is 1 (since both numbers are negative). When these numbers are added, a carry out of the msb occurs if and only if the sum of the msbs of AX and BX is 0.

If there is a carry out of the msb, then the resulting msb of the sum will be 0 (since the sum of two 0s and a carry is 0 with a carry out). If there is no carry out of the

msb, then the resulting msb of the sum will be 1 (since the sum of two 1s and a carry is 1 with no carry out).
Therefore, signed overflow occurs if and only if there is a carry out of the msb but no carry into the msb. This is because a carry out of the msb indicates that the result is too large to be represented as a negative signed number (i.e., it is positive, but the msb is 0), while the lack of a carry into the msb indicates that the result is too small to be represented as a negative signed number (i.e., it is negative, but the msb is 1).

Or

(2) cy_{out} 0 1 cy_{into}

a)

AX= 01xx xxxx xxxx xxxx
BX= 01xx xxxx xxxx xxxx +
10xx xxxx xxxx xxxx

x: could be 0 or 1 (i.e any number)

01

ex: AX= 7FA0h = 0111 1111 1010 0000
BX= 600Dh = 0110 0000 0000 1101 +
DFADh =1101 1111 1010 1101

Cy_{into} =1 ,cy_{out} =0
Cy_{into} XOR cy_{out} = 1 XOR 0 = 1 = OF (signed overflow)

b) cy_{out} 1 0 cy_{into}

AX= 10xx xxxx xxxx xxxx
BX= 10xx xxxx xxxx xxxx +
100xx xxxx xxxx xxxx

Ex: cy_{out} 1 0 cy_{into}

AX=9DE4h =1000 1101 1110 0100
BX= B216h =1011 0010 0001 0 110 +
=10011 1111 1111 1010

Cy_{into} =0 ,cy_{out} =1
Cy_{into} XOR cy_{out} = 0 XOR 1 = 1 = OF (signed overflow)

3. Suppose ADD AX,BX is executed. In each of the following parts, the first number being added is the contents of AX, and the second number is the contents of BX. Give the resulting value of AX and tell whether signed or unsigned overflow occurred.
a. 512Ch + 4185h
b. FE12h + lACBh

c. EIE.4h + DAB3h
d. 7132h + 7000h
e. 6389h + l l 76h
Ans:

(3)

We'll show solution of one and the rest are the same
a.

```
                0 1
512Ch  = 0101 0001 0010 1100
4185h  = 0100 0001 1000 0101  +
92B1h  = 1001 0010 1011 0001
```

Method 1:
By taking XOR between cy_{into} and cy_{out}
$Cy_{into} = 1$, $cy_{out} = 0$ see Q2.a
1 XOR 0 = 1 OF = 1 (signed)
$Cy_{out} = CF = 0$ (unsigned)

Method 2 :
First number is positive
Second number is positive
After addition the result expected is positive , but the result is negative ,so there are error occurred
The error indicate by OF = 1

	Content of AX	signed OF	unsigned CF
a)	92B1h	1	0
b)	18DDh	0	1
c)	BC97h	0	1
d)	E132h	1	0
e)	74FFh	0	0

4. Suppose SUB AX, BX is executed. In each of the following parts, the first number is the initial contents of AX and the second number is the contents of BX. Give the resulting value of AX and tell whether signed or unsigned overflow occurred.
a. 2143h - J986h
b. 8JFEh - 1986h
c. 19BCh - 81fEh
d. 0002h - FEOFh
e. 8BCDh - 71ABh
Ans:

(4)

We'll show solution of one and the rest are the same

c. borrow 1 0 borrow
into msb ↘ ↘ into bit-14

19BCh = 0001 1001 1011 1100
81FEh = 1000 0001 1111 1110 -
97BEh= 10001 0111 1011 1110

Method 1:
By taking XOR between borrow$_{\text{into msb}}$ and borrow$_{\text{into bit-14}}$
borrow$_{\text{into msb}}$ XOR borrow$_{\text{into bit-14}}$
1 XOR 0 = 1 OF = 1 (signed)

borrow$_{\text{into msb}}$ = CF = 1 (unsigned)

Method 2:
First number is positive
Second number is negative
But second number is greater than first number.
And $1^{st} - (-2^{nd}) = 1^{st} + 2^{nd}$ = positive.
The result is negative so an error occurred OF = 1

	Content of AX	signed OF	unsigned CF
a)	07BDh	0	0
b)	6878h	1	0
c)	97BEh	1	1
d)	01F3h	0	1
e)	1A22h	1	0

Flow control Instruction

Table of Contents

Flow control Instruction **1**
Overview **1**
Range of the conditional jump: 1
How the CPU implements the conditional Jump: 1
How CMP works: 1
Signed Jump vs Unsigned Jump: 3
Working with Characters: 3
PROBLEM 1 3
JMP instruction: 3
Case: 4
And using Jump: 5
OR using Jump: 5
LOOP: 6
While LOOP ➰ 6

Overview

1. **Why do we need flow control?**

There must be a way to make decisions and repeat sections of code to do useful tasks. In order to do that we need to control the flow of the program.

- Jump and loop are used to control the flow of the program
- There are 256 characters in the IBM character set
- 32 -127 are standard ASCII characters
- 0 - 31 and 128 - 255 are the graphics characters.

Range of the conditional jump:

The destination label must precede the jump instruction by no more than 126 bytes and follow no more than 127 bytes.

How the CPU implements the conditional Jump:

To implement a conditional jump, the CPU looks at the FLAGS register. You already know It reflects the result of the last thing the processor did. If the condition for the jump is true then the CPU adjusts the IP to point to the destination label. so that the instruction on this label will be done next. If the jump condition is false, then IP is not altered; this means that the next instruction in line will be done.

Categories of the conditional Jumps:

1. Signed Jump
2. Unsigned Jump
3. Single flag jump

How CMP works:

Syntax: CMP destination, source;

This compares the destination and the source by substituting the contents of the source with the contents of the destination. The result of this computation is not stored but it affects the flags. The CMP is just like the SUB only it does not change the contents of the destination. The destination can be a memory location or a constant.

The conditional Jumps:

Table 6.1 Conditional Jumps

Signed Jumps

Symbol	*Description*	*Condition for Jumps*
JG/JNLE	jump if greater than jump if not less than or equal to	ZF = 0 and SF = OF
JGE/JNL	jump if greater than or equal to jump if not less than or equal to	SF = OF
JL/JNGE	jump if less than jump if not greater than or equal	SF <> OF
JLE/JNG	jump if less than or equal jump if not greater than	ZF = 1 or SF <> OF

Unsigned Conditional Jumps

Symbol	*Description*	*Condition for Jumps*
JA/JNBE	jump if above jump if not below or equal	CF = 0 and ZF = 0
JAE/JNB	jump if above or equal jump if not below	CF = 0
JB/JNAE	jump if below jump if not above or equal	CF = 1
JBE/JNA	jump if equal jump if not above	CF = 1 or ZF = 1

Single-Flag Jumps

Symbol	*Description*	*Condition for Jumps*
JE/JZ	jump if equal jump if equal to zero	ZF = 1
JNE/JNZ	jump if not equal jump if not zero	ZF = 0
JC	jump if carry	CF = 1
JNC	jump if no carry	CF = 0
JO	jump if overflow	OF = 1
JNO	jump if no overflow	OF = 0
JS	jump if sign negative	SF = 1
JNS	jump if nonnegative sign	SF = 0
JP/JPE	jump if parity even	PF = 1
JNP/JPO	jump if parity odd	PF = 0

- Generally, all these jumps are used with the CMP
  CMP AX, BX
  JA There
- But it is not compulsory to use the CMP it can be used with

Signed Jump vs Unsigned Jump:

For every signed jump there is an analogous unsigned jump. JG has JA, JL has JB.
Flags used by Signed flags: ZF , SF, OF
Flags used by unsigned flags: CF , ZF

For example, suppose we're giving a signed interpretation. If AX = 7FFFh, BX = 8000h, and we execute

```
CMP   AX,BX
JA    BELOW
```

then even though 7FFFh > 8000h in a signed sense, the program does not jump to BELOW. The reason is that 7FFFh < 8000h in an unsigned sense, and we are using the unsigned jump JA.

Working with Characters:

In working with the standard ASCII character set. either signed or unsigned jumps may be used because the sign bit of a byte containing a character code is always zero. However, unsigned jumps should be used when comparing EXTENDED ASCII char. (code 80 TO FFh)

PROBLEM 1

Example 6.1 Suppose AX and BX contain signed numbers. Write some code to put the biggest one in CX.

Solution:

```
        MOV   CX,AX          ;put AX in CX
        CMP   BX,CX          ;is BX bigger?
        JLE   NEXT           ;no, go on
        MOV   CX,BX          ;yes, put BX in CX
NEXT:
```

JMP instruction:

The jMP (jump) instruction causes an unconditional transfer of control (unconditional jump): The syntax is. JMP Destination;
The destination is usually a label in the same segment. JMP can be used to get around the range restriction.
If then using Jump:

Example 6.2 Replace the number in AX by its absolute value.

Solution: A pseudocode algorithm is

```
IF AX < 0
 THEN
  replace AX by -AX
END_IF
```

It can be coded as follows:

```
;if AX < 0
                CMP   AX,0        ;AX < 0 ?
                JNL   END_IF      ;no, exit
;then
                NEG   AX          ;yes, change sign
END_IF:
```

The condition AX < 0 is expressed by CMP AX,0. If AX is not less than 0, there is nothing to do, so we use a JNL (jump if not less) to jump around the NEG AX. If condition AX < 0 is true, the program goes on to execute NEG AX.

If then else:

Example 6.3 Suppose AL and BL contain extended ASCII characters. Display the one that comes first in the character sequence.

Solution:

```
IF AL <= BL
 THEN
  display the character in AL
 ELSE
  display the character in BL
END_IF
```

It can be coded like this:

```
                MOV  AH,2       ;prepare to display
;if AL <= BL
                CMP  AL,BL      ;AL <= BL?
                JNBE ELSE_      ;no, display char in BL
;then                           ;AL <= BL
                MOV DL,AL       ;move char to be displayed
                JMP DISPLAY     ;go to display
ELSE_:                          ;BL < AL
                MOV DL,BL
```

Case:

Example 6.4: If AX contains a negative number, put -1 In BX; if AX contains 0, put O In BX; if AX contains a positive number, put 1 In BX.

Solution:

```
CASE AX
   <0: put -1 in BX
   =0: put 0 in BX
   >0: put 1 in BX
END_CASE
```

It can be coded as follows:

```
;case AX
                CMP AX,0        ;test ax
                JL  NEGATIVE    ;AX < 0
                JE  ZERO        ;AX = 0
                JG  POSITIVE    ;AX > 0
NEGATIVE:
                MOV  BX,-1      ;put -1 in BX
                JMP  END_CASE   ;and exit
ZERO:
                MOV  BX,0       ;put 0 in BX
                JMP  END_CASE   ;and exit
POSITIVE:
                MOV  BX,1       ;put 1 in BX
END_CASE:
```

Note: only one CMP is needed, because jump instructions do not affect the

And using Jump:

Example 6.6 Read a character, and if it's an uppercase letter, display it.

Solution:

```
Read a character (into AL)
IF ('A' <= character) and (character <= 'Z')
 THEN
  display character
END_IF
```

To code this, we first see if the character in AL follows "A" (or is "A") in the character sequence. If not, we can exit. If so, we still must see if the character precedes "Z" (or is "Z") before displaying it. Here is the code:

```
;read a character
            MOV  AH,1       ;prepare to read
            INT  21H        ;char in AL
;if ('A' <= char) and (char <= 'Z')
            CMP  AL,'A'     ;char >= 'A'?
            JNGE END_IF     ;no, exit
            CMP  AL,'Z'     ;char <= 'Z'?
            JNLE END_IF     ;no, exit
;then display char
            MOV  DL,AL      ;get char
            MOV  AH,2       ;prepare to display
            INT  21H        ;display char
END_IF:
```

OR using Jump:

Example 6.1 Read a character. If it's "y" or "Y", display it; otherwise, terminate the program.

```
;read a character
            MOV  AH,1       ;prepare to read
            INT  21H        ;char in AL
;if (character = 'y') or (character = 'Y')
            CMP  AL,'y'     ;char = 'y'?
            JE   THEN       ;yes, go to display it
            CMP  AL,'Y'     ;char = 'Y'?
            JE   THEN       ;yes, go to display it
            JMP  ELSE_      ;no, terminate
THEN:
            MOV  AH,2       ;prepare to display
            MOV  DL,AL      ;get char
            INT  21H        ;display it
            JMP  END_IF     ;and exit
ELSE_:
            MOV  AH,4CH
            INT  21H        ;DOS exit
END_IF:
```

LOOP:

The counter for the loop is the register CX which is initialized to loop_Count. Execution of the LOOP Instruction causes CX to be decremented automatically. and if CX Is not 0, the control transfers to destination_label. If CX "'0, the next instruction after LOOP is done. Destinatlon_label must precede the LOOP instruction by no more than 126 bytes.
The critical case for loop: Typical for loop that is implemented using loop instruction will be executed at least once. So before entering the loop if CX holds 0 so the loop instruction will decrement the CX to FFFFh and the loop will be executed 65535 times so JCXZ is used before the loop starts to check this corner case.

While LOOP ➰

Example 6.9 Write some code to count the number of characters In the input line.

The code is

```
        MOV   DX,0        ;DX counts characters
        MOV   AH,1        ;prepare to read
        INT   21H         ;character in AL
WHILE_:
        CMP   AL,0DH      ;CR?
        JE    END_WHILE   ;yes, exit
        INC   DX          ;not CR, increment count
        INT   21H         ;read a character
        JMP   WHILE_      ;loop back
END_WHILE:
```

While vs Repeat:
In many situations where a conditional loop Is needed, the use of a WHILE loop or a REPEAT loop Is a matter of personal preference. The advantage of a WHILE is that the loop can be bypassed if the terminating, condition is initially false, whereas the statements in a REPEAT must be done at least once. However, the code for a REPEAT loop Is likely to be a little shorter because there is only a conditional jump at the end, but a WHILE loop has two jumps: a conditional jump at the top and a JMP at the bottom

Problem:
Prompt the user to enter a line of text. On the next line, display the capital Jetter entered that comes first alphabetically and the one that comes last. If no capital letters are entered, display "No capital letters". The execution should look like this:

Program Listing PGM6_2.ASM

```
TITLE PGM6_2: FIRST AND LAST CAPITALS
.MODEL SMALL
.STACK 100H
.DATA
PROMPT DB     'Type a line of text',0DH,0AH,'$'
NOCAP_MSG  DB 0DH,0AH,'No capitals $'
CAP_MSG       DB      0DH,0AH,'First capital = '
FIRST  DB     ']'
       DB     ' Last capital = '
LAST   DB     '@ $'

.CODE
MAIN   PROC
;initialize DS
       MOV   AX,@DATA
       MOV   DS,AX
;display opening message
       MOV   AH,9              ;display string function
       LEA   DX,PROMPT         ;get opening message
       INT   21H               ;display it
;read and process a line of text
       MOV   AH,1              ;read char function
       INT   21H               ;char in AL
WHILE_:
;while character is not a carriage return do
       CMP   AL,0DH            ;CR?
       JE    END_WHILE         ;yes, exit
;if character is a capital letter
       CMP   AL,'A'            ;char >= 'A'?
       JNGE  END_IF            ;not a capital letter
       CMP   AL,'Z'            ;char <= 'Z'?
       JNLE  END_IF            ;not a capital letter
;then
;if character precedes first capital
       CMP   AL,FIRST          ;char < first capital?
       JNL   CHECK_LAST        ;no, >=
;  then first capital = character
       MOV   FIRST,AL          ;FIRST = char
;end_if
CHECK_LAST:
;if character follows last capital
       CMP   AL,LAST           ;char > last capital?
       JNG   END_IF            ;no, <=
;then last capital = character
       MOV   LAST,AL           ;LAST = char
;end_if
END_IF:
;read a character
       INT   21H               ;char in AL
       JMP   WHILE_            ;repeat loop
END_WHILE:
;display results


       MOV   AH,9              ;display string function
;if no capitals were typed
       CMP   FIRST,']'         ;first = ']'
       JNE   CAPS              ;no, display results
;then
       LEA   DX,NOCAP_MSG      ;no capitals
       JMP   DISPLAY
CAPS:
       LEA   DX,CAP_MSG        ;capitals
DISPLAY:
       INT   21H               ;display message
;end_if
;dos exit
       MOV   AH,4CH
       INT   21H
MAIN   ENDP
       END   MAIN
```

Book Exercises:

Exercises

1. Write assembly code for each of the following decision structures.

a.
```
IF AX < 0
  THEN
   PUT -1 IN BX
 END_IF
```

b.
```
IF AL < 0
  THEN
   put FFh in AH
  ELSE
   put 0 in AH
 END_IF
```

c. Suppose DL contains the ASCII code of a character.
```
(IF DL >= "A") AND (DL <= '2')
 THEN
  display DL
 END_IF
```

e.
```
IF (AX < BX) OR (BX < CX)
   THEN
    put 0 in DX
   ELSE
    put 1 in DX
  END_IF
```

f.
```
IF AX < BX
   THEN
    put 0 in AX
   ELSE
    IF BX < CX
     THEN
      put 0 in BX
     ELSE
      put 0 in CX
    END_IF
  END_IF
```

(1)

a)
```
        CMP     AX,0
        JGE     END_IF
        MOV     BX,-1
END_IF:
```

---

b)
```
        CMP     AL,0
        JNL     ELSE_
        MOV     AH,0FFh
        JMP     END_IF
ELSE_:  MOV     AH,0
END_IF:
```

OR

```
        CMP     AL,0
        JL      THEN_
        MOV     AH,0
        JMP     END_IF
THEN_:  MOV     AH,0FFh
END_IF:
```

---

c)
```
        CMP     DL,'A'
        JL      END_IF
        CMP     DL,'Z'
        JG      END_IF
        MOV     AH,2            ; display DL
        INT     21H
END_IF:
```

e)
```
        CMP     AX,BX
        JL      THEN_
        CMP     BX,CX
        JL      THEN_
        MOV     DX,1
        JMP     END_IF

THEN_:  MOV     DX,0
END_IF:
```

---

f)
```
        CMP     AX,BX
        JNL     ELSE_
        MOV     AX,0
        JMP     END_IF

ELSE_:  CMP     BX,CX
        JNL     ELSE_2

        MOV     BX,0
        JMP     END_IF

ELSE_2: MOV     CX,0
END_IF:
```

2. Use a CASE structure to code the following:

Read a character.
If it's "A", then execute carriage return.
If it's "B", then execute line feed.
If it's any other character, then return to DOS.

Write a sequence of instructions to do each of the following:

a. Put the sum 1 + 4 + 7 + . . . + 148 in AX.
b. Put the sum 100 + 95 + 90 + . . . + 5 in AX.

Employ LOOP instructions to do the following:

a. put the sum of the first 50 terms of the arithmetic sequence 1, 5, 9, 13, . . . in DX.
b. Read a character and display it 80 times on the next line.
c. Read a five-character password and overprint it by executing a carriage return and displaying five X's. You need not store the input characters anywhere.

The following algorithm may be used to carry out division of two nonnegative numbers by repeated subtraction:

```
initialize quotient to 0
WHILE dividend >= divisor DO
increment quotient
subtract divisor from dividend
END_WHILE
```

Write a sequence of instructions to divide AX by BX, and put the quotient in CX.

(2)

```
        MOV   AH,1          ;read a character
        INT   21H

        CMP   AL,'A'        ;AL has the letter in.
        JE    EXE_CR
        CMP   AL,'B'
        JE    EXE_LF
        ;any letter else
        MOV   AH,4Ch        ;return to DOS
        INT   21H
        JMP   END_CASE

EXE_CR: MOV   AH,2
        MOV   DL,0Dh
        INT   21H
        JMP   END_CASE

EXE_LF: MOV   AH,2
        MOV   DL,0Ah
        INT   21H

END_CASE:
```

(3)

a) first we find how many loops needed:
(last term – first term)/ difference
(148 – 1)/3 = 49 loops

```
        MOV   CX,49
        MOV   AX,1          ;first term
        MOV   BX,1

L1:     ADD   BX,3  ;add the diff. between each term
        ADD   AX,BX

        LOOP L1
```

b) (first term – last term)/ difference
(100 – 5)/5 = 19 loops

```
        MOV     CX,19
        MOV     AX,100
        MOV     BX,100

L1:     SUB     BX,5
        ADD     AX,BX

        LOOP L1
```

---

(4)

```
a) MOV      CX,50
   MOV      DX,1
   MOV      AX,1
L1:ADD      AX,4

  LOOP      L1
```

---

4. b)

```
        MOV     AH,1
        INT     21H
        MOV     AH,2
        MOV     DL,0AH      ;et
        INT     21H
        MOV     DL,0DH
        INT     21H
        MOV     DL,AL
        MOV     CX,80

DISPLAY: INT    21H
        LOOP    DISPLAY
```

---

4 c)

```
        MOV     CX,5
        MOV     AH,7    ;*
L1:     INT  21H
        LOOP    L1
;-----------------------
        MOV     DL,'X'
        MOV     CX,5
        MOV     AH,2
L2:     INT  21H
        LOOP    L2
```

(5)

```
MOV AX,0
;CX divident    BX  divisor      AX quotient
while_:
        CMP     CX,BX
        JL   END_WHILE
        INC     AX
        SUB     CX,BX
        JMP     WHILE_
END_WHILE:
```

8. **Write a program to display a "?", read two capital letters, and display them on the next line in alphabetical order.**
9. **Write a program to display the extended ASCII characters (ASCII codes 80h to FFh). Display 10 characters per line, separated by blanks. Stop after the extended characters have been displayed once.**

(8)

```
.MODEL SMALL

.CODE
MAIN PROC
                MOV AH,2
                MOV DL,'?'
                INT  21H            ;display '?'

                MOV AH,1
                INT 21H   ; read 1st char. & put it in BL
                MOV BL,AL
                INT 21H  ;read 2nd char. in AL
;-------------------------
                CMP BL,AL
                JG SWITCH     ;if not orderd
                JMP DISPLAY

SWITCH:         XCHG AL,BL
DISPLAY:
                MOV AH,2
                MOV DL, 0AH       ;enter
                INT 21H
                MOV DL,BL
                INT 21H
                MOV DL,AL


                INT 21H


OUT_:           MOV AH,4CH
                INT 21H

MAIN ENDP
END MAIN
```

(9)

```
.MODEL SMALL
.CODE
MAIN PROC
                ;BH counter (10 char per line)
                MOV     BH,0
                 MOV    AH,2
                MOV     CX,7FH  ; 127 char.
                MOV     DL,80H  ;1ST letter
                MOV     BL,80H

CONTINUE:       MOV DL,BL
                INT 21H
                DEC CX
                INC DL
                MOV BL,DL
                MOV DL,20H      ;blank space
                INT 21H
                INC BH
                CMP BH,10
                JE COUNT
                CMP CX,0
                JNE CONTINUE    ;is letter finished?
                JMP out_

COUNT:          MOV BH,0        ;start again counter
                MOV DL,0AH      ;enter
                INT 21H
                MOV DL,0DH
                INT 21H


                JMP CONTINUE
Out_:           MOV AH,4CH
                INT 21H


MAIN ENDP
                END MAIN
```

Stack and Introduction to Procedures

The Stack

- Stack is considered as the **temporary storage** of data and addresses
- **One dimensional** data structure
- **LIFO** (Last In First Out) technique is used
- The **most recent addition** to the stack is called the **top of the stack**
- The statement **.STACK 100H** in your program sets aside a block of **256 bytes** of memory to hold the stack (Here 100h is equivalent to 256 in decimal)
- The **SS** (Stack Segment Register) contains the segment number of the stack segment
- The complete segment:offset address to access the stack is **SS:SP**
- Initially before any data or addresses have been placed on the stack, the SP contains the offset address of the memory location immediately following the stack segment

Push:

Push command is mainly used to insert a new **word** into the stack. Here the **word** takes up to 2 bytes. As a result-

1. When a word is entered into the stack the SP (stack Pointer) is **decreased by 2**.
2. A **copy** of the source is moved to the stack to the address specified by the stack SS:SP

It is to be noted that-

a. Any data cannot be directly pushed into the stack
b. Any command like PUSH DL i.e. a 8-bit register cannot be pushed into the stack.

Command for PUSH -

PUSH source

Here source is the 16-bit register. For example -

PUSH DX

Illegal -

PUSH DL

PUSHF:

The instruction PUSHF, which has no operands, pushes the contents of the FLAGS register onto the stack.

PUSH and PUSHF don't affect the flag registers.

Figure 8.1A Empty Stack

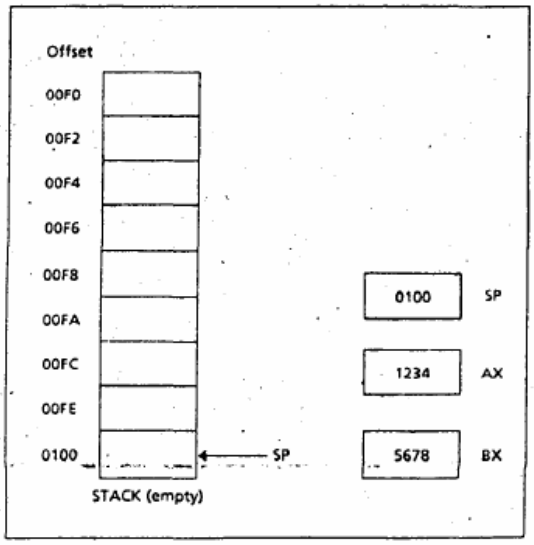


Figure 8.1B After PUSH AX

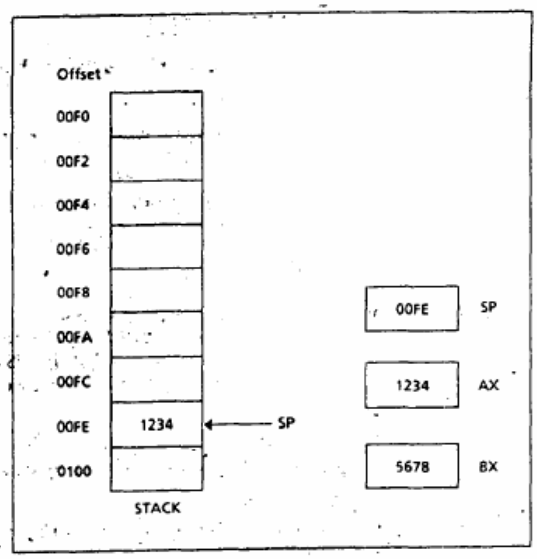


Figure 8.1A Empty Stack

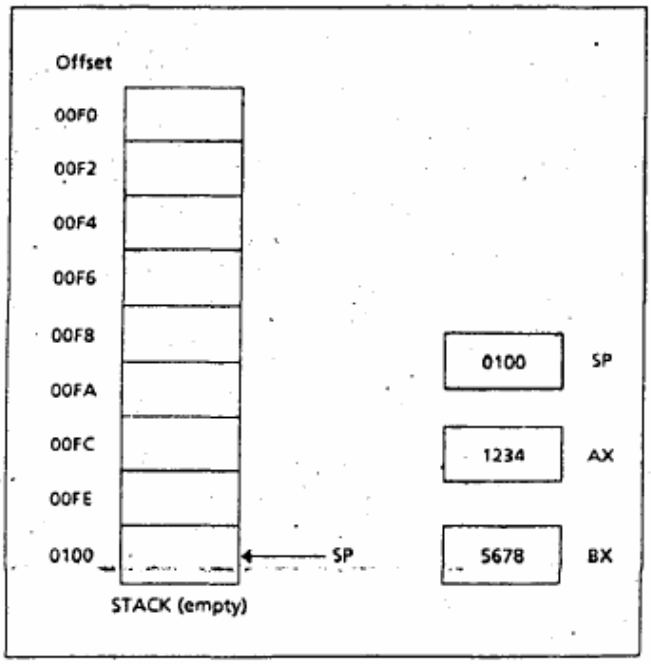
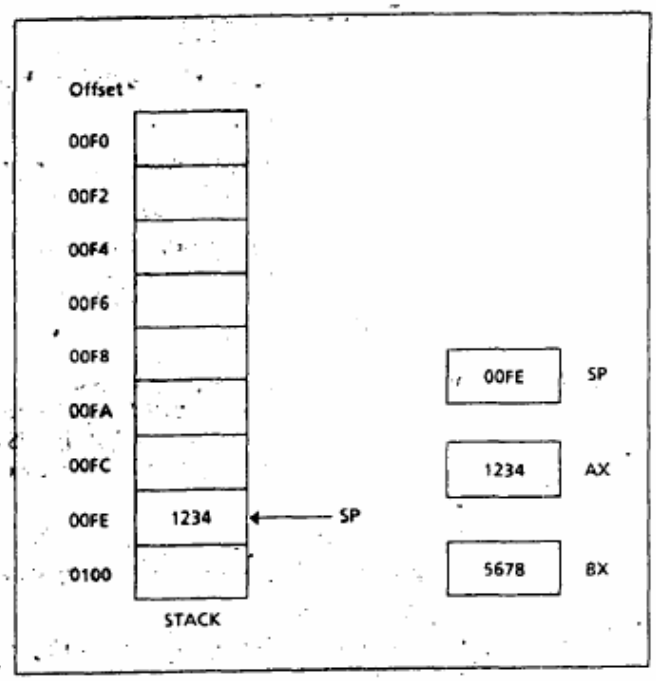


Figure 8.1B After PUSH AX



POP:

To remove the top item from the stack, we POP it. The syntax Is

POP destination

Here source is the 16-bit register. For example -

POP DX

Illegal -

POP DL

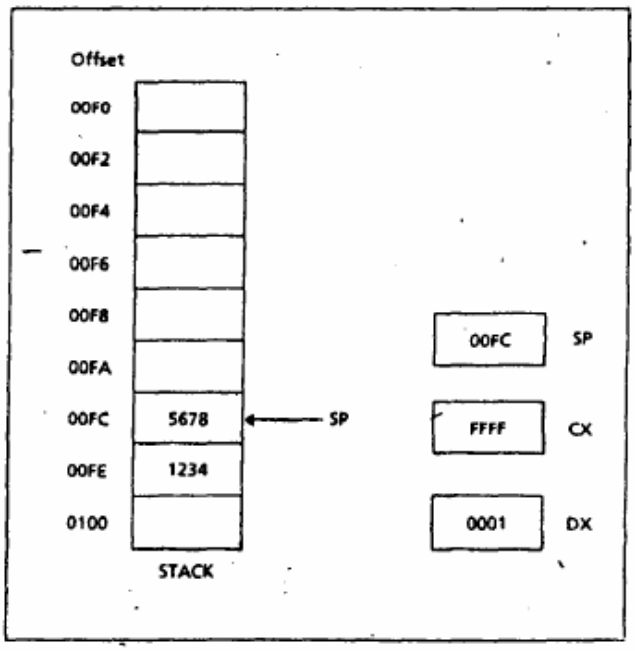
a. The opposite of push occurs here.
b. The SP is increased by 2.
c. The content of the top of the stack is moved to the source register.
d. It does not affect any flag register.

POPF:

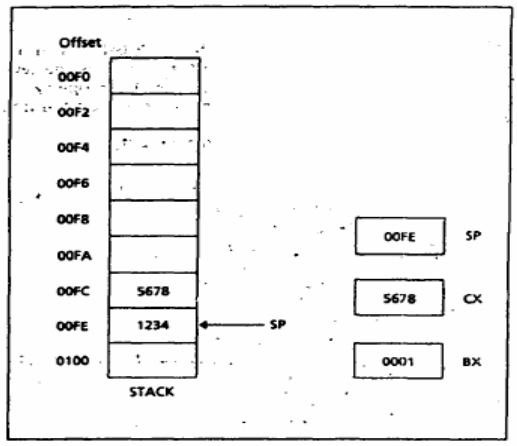
The Instruction POPF pops the top of the stack into the FLAGS register.

POPF could theoretically change all the flags because it resets the FLAGS REGISTER to some original value that you have previously saved with the PUSHF instruction.

8.2A Before POP

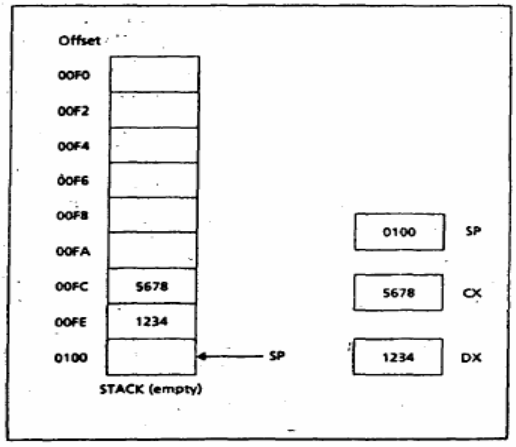


?B After POP CX



because any values DOS pushes onto the stack are popped off by DOS before it returns control to the user's program.

?C After POP DX



If the Stack is empty then the value of the SP is 100h otherwise it has a value between 0000h-00FEh.

Important Notes

- *Not only can the programmer use the stack but DOS can and also does use the stack*
- *In fact DOS uses the stack every time the user executes an INT 21h function*
- *Because of the "last-in first-out" nature of the stack, the order that items are removed from the stack is the reverse of the order in which they are placed on the stack*

Important Notes on Stack

- PUSHES AND POPS are often used to save data temporarily on the program stack. They are also used implicitly each time a CALL and a RETurn sequence is executed.
- Remember that the SP is decremented **BEFORE** placing a word on the stack at PUSH time but it is incremented **AFTER** removing a word from the stack at POP time.
- If, for some reason, you want a copy of the FLAGS register in BX, you can accomplish this by:

  PUSHF
  POP BX
- Stack allows you to save the contents of a register, use the register for something else temporarily, and the restore the register to its original value.

Notes on Stack (cont'd)

- Pushing and popping the contents of registers is preferable to storing their contents as variables in the DATA segment
- Reasons:
  - Using the stack is more economical. Instead of allocating data space, by pushing and popping data into the stack, you use space as you need it and release it when you no longer need it.
  - Since the 8088/8086 allows recursion, if a routine called itself and saved the contents of a register to a data location each time it was invoked, it would be overwriting the previous contents of that location with each recursion!
  - Using a stack instead of a data location makes code more portable. Once you have written a good routine, you may choose to incorporate that routine into several different programs. Or if you are working with a programming team piecing smaller subroutines into a one larger main routine, subroutines that do their work without referencing particular data locations are more easily patched into main programs than subroutines that do reference particular data locations. Therefore, should not refer to ANY variable data names in ANY procedure that you write!!!

Notes on Stack (cont'd)

- Care must be taken not to corrupt the STACK because not only does it save values for the programmer but it also saves values for the CPU. These values get interwoven on the stack. If the SP becomes confused, the CPU could get lost throwing your computer into a system error!!
- Always check to see that the PUSHES and POPS in a program are paired --- or --- at least that each of them is balanced by program code that restores the stack pointer to its proper value.
- If you find yourself in the middle of a system error, more than likely you look for a problem in the way you implemented the stack.

Example Program

- The following code allows a user to input a string consisting of 10 characters and then displays the 10 characters in reverse order on the screen

```
TITLE   DISPLAY THE 10 CHARACTERS IN REVERSE ORDER
.MODEL   SMALL
.STACK   100H
.DATA
 CR                  EQU     0DH
 LF            EQU    0AH
MESSAGE     DB      CR,LF,'PLEASE TYPE ANY 10 '
              DB      ' CHARACTERS',CR,LF,'$'
REVERSE     DB      CR,LF,'THE CHARACTERS IN REVERSE'
              DB      ' ARE:',CR,LF,'$'
```

Example Program (cont'd)

```
.CODE
MAIN    PROC
; ----------------------------INITIALIZE DATA SEGMENT REGISTER
      MOV    AX,@DATA
      MOV    DS,AX
;---------------------- SOUND BELL AND PRINT A MESSAGE FOR
   INPUT
      MOV    AH,2
      MOV    DL,07H
      INT    21H
      MOV    AH,9
      LEA    DX,MESSAGE
      INT    21H
;----------------------------ACCEPT CHARACTERS
      MOV    CX,10
      MOV    AH.1
```

Example Program (cont'd)

```
    READ:
          INT    21H
          PUSH   AX        ;CAN'T PUSH AL SO PUSH AX!
          LOOP   READ
    ;----------------------------PRINT REVERSE MESSAGE
          MOV    AH,9
          LEA    DX,REVERSE
          INT    21H
    ;----------------------------PREPARE TO PRINT IN REVERSE
          MOV    CX,10
          MOV    AH,2
```

Example Program (cont'd)

```
READ:
      INT    21H
      PUSH   AX       ;CAN'T PUSH AL SO PUSH AX!
      LOOP   READ
;----------------------------PRINT REVERSE MESSAGE
      MOV    AH,9
      LEA    DX,REVERSE
      INT    21H
;----------------------------PREPARE TO PRINT IN REVERSE
      MOV    CX,10
      MOV    AH,2
```

Algorithm to Reverse Input

```
Display a '?'
Initialize count to 0
Read a character
WHILE character is not a carriage return DO
   push character onto the stack
   increment count
   read a character
END_WHILE;
Go to a new line
FOR count times DO
   pop a character from the stack;
   display it;
END_FOR
```

Here is the program:

Program Listing PGM8_1.ASM

```
1:  TITLE PGM8_1:REVERSE INPUT
2:  .MODEL    SMALL
3:  .STACK    100H
4:  .CODE
5:  MAIN      PROC
6:  ;display user prompt
7:          MOV    AH,2          ;prepare to display
8:          MOV    DL,'?'        ;char to display
9:          INT    21H           ;display '?'
10: ;initialize character count
11:         XOR    CX,CX         ;count = 0
12: ;read a character
13:         MOV    AH,1          ;prepare to read
14:         INT    21H           ;read a char
15: ;while character is not a carriage return do
16: WHILE_:
17:         CMP    AL,0DH        ;CR?
18:         JE     END_WHILE     ;yes, exit loop
19:         ;save character on the stack and increment cou
20:         PUSH   AX            ;push it on stack
21:         INC    CX            ;count = count + 1
22: ;read a character
23:         INT    21H           ;read a char
24:         JMP    WHILE_        ;loop back
25: END_WHILE:
26: ;go to a new line
27:         MOV    AH,2          ;display char fcn
28:         MOV    DL,0DH        ;CR
29:         INT    21H           ;execute
30:         MOV    DL,0AH        ;LF
```

```
31:           INT    21H          ;execute
32:           JCXZ   EXIT         ;exit if no characters read
33: ;for count times do
34: TOP:
35: ;pop a character from the stack
36:           POP    DX           ;get a char from stack
37: ;display it
38:           INT    21H          ;display it
39:           LOOP   TOP
40: ;end_for
41: EXIT:
42:           MOV    AH,4CH
43:           INT    21H
44: MAIN      ENDP
45:           END    MAIN
```

The Procedure

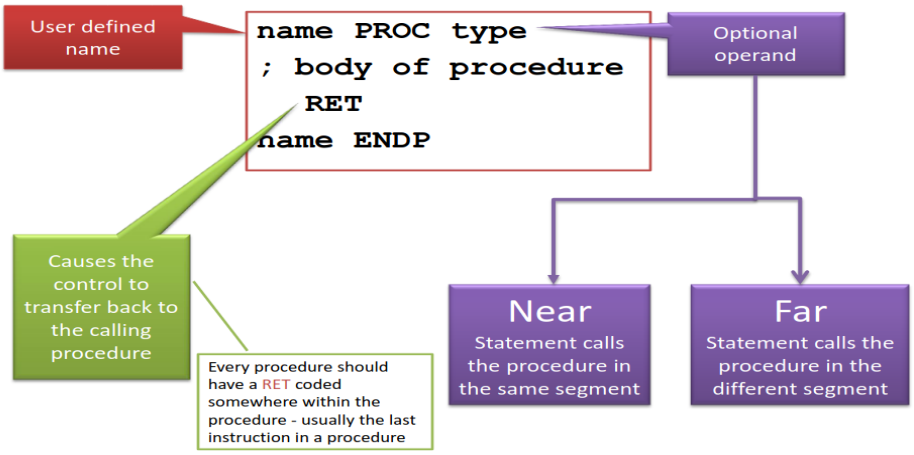
Terminology of Procedures

- Top-down program design
  - Decompose the original problem into a series of subproblems that are easier to solve than the original problem
- Subproblems in assembler language can be structured as a collection of procedures
- Main procedure contains the entry point to the program and can call one of the other procedures using a CALL statement
- It is possible for a called sub-procedure to call other procedures
- In AL, it is also possible for a called sub-procedure to call itself (recursion)!

Terminology of Procedures (cont'd)

- When a procedure calls another procedure, control transfers to the called procedure
- When the instructions in a called procedure have been executed, the called procedure usually returns control to the calling procedure at the next sequential instruction after the CALL statement
- In high level languages, mechanism for call and return are hidden from programmer

Procedure Declaration

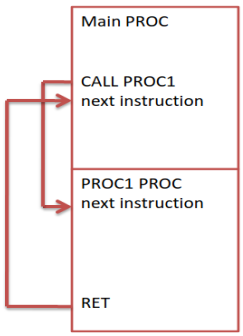


PROC Instruction

- PROC instruction establishes a procedure
- Procedure declaration syntax:

```
name        PROC
; body of the procedure
            RET
name    ENDP
```

- name is a user-defined variable.
- RET instruction causes control to transfer back to the calling Procedure.
- Every procedure should have a RET coded somewhere within the procedure - usually the last instruction in a procedure



COMMUNICATION BETWEEN PROCEDURES

- Programmers must devise a way to communicate between procedures –
- there are no parameter lists !!!
- Typically in assembler language, procedures often pass data to each other through registers

Procedures Documentation

- Procedures should be well-documented
  - Describe what the procedure does
  - Indicate how it receives its input from the calling program
  - Indicate how it delivers the results to the calling program
  - Indicate the names of any other procedures that this procedure calls

```
; Describe what the procedure does
; input:  Indicate how it receives its input
          from the calling program

; output: Indicate how it delivers the results
          to the calling program

; uses:   Indicate the names of any other
          procedures that this procedure calls
```

Procedures (cont'd)

- A procedure usually begins by PUSHing (saving) the current contents of all of the registers on the stack.
- A procedure usually ends by POPing the stack contents back into the registers before returning to the CALLing procedure
- When writing a procedure, do NOT PUSH or POP any registers in which you intend to return output!!

CALL Instruction

- A CALL instruction invokes a procedure
- SYNTAX: `CALL name` (direct CALL)
  where name is the name of a procedure.

- Executing a CALL instruction causes the following to happen:
  - The return address of the CALLing program which is in the IP register is pushed (saved) on the STACK. This saved address is the offset of the next sequential instruction after the CALL statement (CS:IP)
  - The IP then gets the offset address of the first instruction in the procedure

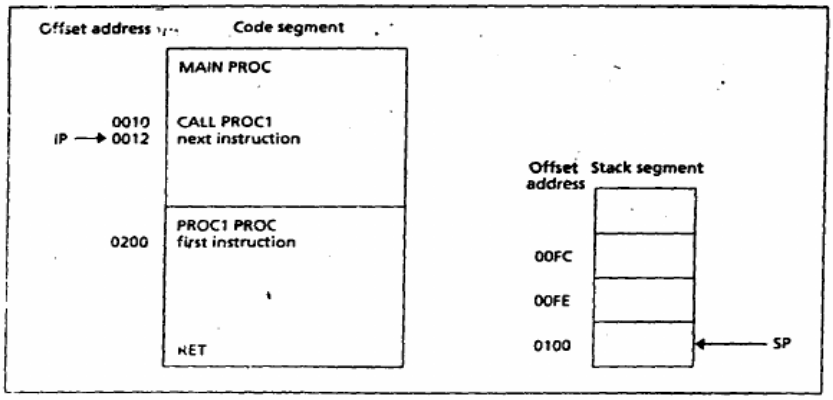


Figure 8.4A Before CALL

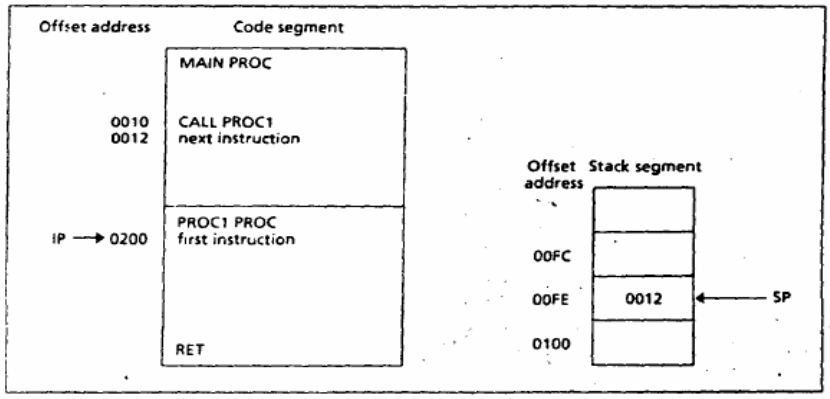


Figure 8.4B After CALL

RET Instruction

- RET statement cause the stack to be popped into IP. Procedures typically end with a RET statement.
- Syntax: RET
- Once the RET is executed, CS:IP now contains the segment offset of the return address and control returns to the calling program
- In order for the return address to be accessible, each procedure must ensure that the return address is at the top of the stack when the RET instruction is executed.

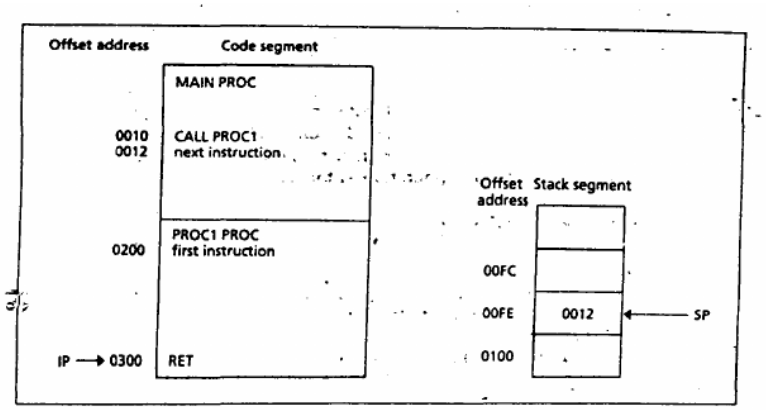


Figure 8.5A Before RET

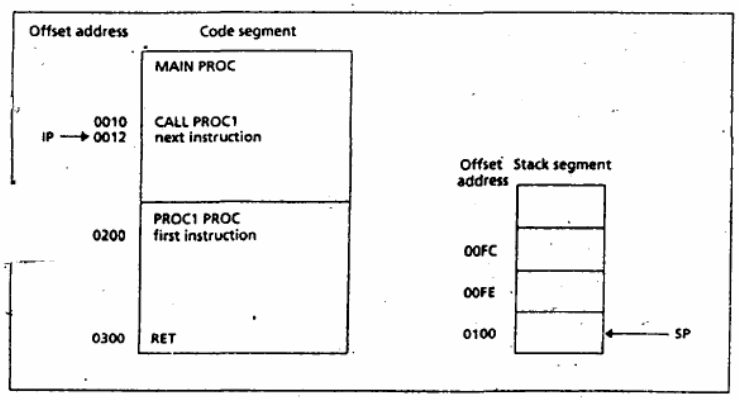


Figure 8.5B After RET

Example Program

- Now let's study the Multiplication Procedure

```
Product  = 0
Repeat
  IF lsb of B is 1
  then
     product  = product + A
  end_if
  shift left A
  shift right B
Until B = 0
```

Example Program (cont'd)

```
TITLE   MULTIPLICATION BY ADDING AND SHIFTING (8 BITS BY 8 BITS)
.MODEL   SMALL
.STACK   100H
.CODE
MAIN     PROC
;-------------------------------> INITIALIZE AX AND BX
       MOV     AX,13          ;SOME ARBITRARY VALUE
       MOV     BX,10          ;SOME ARBITRARY VALUE
;-------------------------------> INVOKE PROCEDURE
        CALL              MULTIPLY
;-------------------------------> DX NOW CONTAINS PRODUCT
; RETURN TO DOS
       MOV     AH,4CH
       INT     21H
MAIN     ENDP
```

Example (cont'd)

```
MULTIPLY PROC
;----------------------------> THIS PROCEDURE MULTIPLIES THE
;----------------------------> VALUE IN AX BY THE VALUE IN BX
;----------------------------> RETURNING THE PRODUCT IN DX.
;----------------------------> VALUES IN AX AND BX ARE LIMITED ;----------------------------> TO 00 - FFh.
;----------------------------> IT USES SHIFTING AND ADDITION
;----------------------------> TO ACCOMPLISH THE MULTIPLICATION
        PUSH   AX          ; DON'T DESTROY AX
         PUSH   BX          ; DON'T DESTROY BX
         XOR   DX,DX        ; CLEAR DX WHERE PRODUCT WILL BE
REPEAT:
         TEST   BX,1        ; IS LSB = 0?
         JZ     END_IF
         ADD    DX,AX       ; PRODUCT = PRODUCT + A
END_IF:
         SHL    AX,1
         SHR    BX,1
        JNZ    REPEAT
         POP    BX
         POP    AX
        RET
MULTIPLY ENDP
      END     MAIN
```

Example Procedure on Safe Use of Stack

- A procedure to display a carriage return and a line feed:

```
CRLF    PROC
        PUSH        AX          ; Save AX
        PUSH        DX          ; Save DX
        MOV         AH,2        ; Display a Carriage Return
        MOV         DL,0Dh
        INT         21h
        MOV         DL,0Ah      ; Display a Line Feed
        INT         21h
        POP         DX          ; Restore DX
        POP         AX          ; Restore AX
        RET
CRLF    ENDP
```

Example (cont'd)

- This procedure can be called by the programmer at any time regardless of what is in his/her AX or DX registers. As far as the programmer is concerned, all you know is that this procedure issues the CR/LF sequence to the console and all of your registers will be unchanged when the procedure has finished executing!

Multiplication algorithm:

```
Product = 0
REPEAT
   IF lsb of B is 1   (Recall lsb = least
              significant bit)
   THEN
      Product = Product + A
   END_IF
   Shift left A
   Shift right B
UNTIL B = 0
```

For example, if A = 111b = 7 and B = 1101b = 13

```
Product = 0
Since lsb of B is 1, Product = 0 + 111b = 111b
Shift left A: A = 1110b
Shift right B: B = 110b

Since lsb of B is 0,
Shift left A: A = 11100b
Shift right B: B = 11b

Since lsb of B is 1
Product = 111b + 11100b = 100011b
Shift left A: A = 111000b
Shift right B: B = 1

Since lsb of B is 1
Product = 100011b + 111000b = 1011011b
Shift left A: A = 1110000b
Shift right B: B = 0

Since lsb of B = 0
Return Product = 1011011b = 91d
```

Note that we get the same answer by performing the usual decimal multiplication process on the binary numbers:

```
    111b
  x 1101b
  -------
    111
   000
  111
 111
-------
1011011b
```

Program Listing PGM8_2.ASM

```
1:  TITLE   PGM8_2: MULTIPLICATION BY ADD AND SHIFT
2:  .MODEL  SMALL
3:  .STACK  100H
4:  .CODE
5:  MAIN    PROC
6:  ;execute in DEBUG. Place A in AX and B in BX
7:          CALL   MULTIPLY
8:  ;DX will contain the product
9:          MOV    AH,4CH
10:         INT    21H
11: MAIN    ENDP
12: MULTIPLY PROC
13: ;multiplies two nos. A and B by shifting and addition
14: ;input:  AX = A, BX = B. Nos. in range 0 - FFh
15: ;output: DX = product
16:         PUSH   AX
17:         PUSH   BX
18:         XOR    DX,DX        ;product = 0
19: REPEAT:
20: ;if B is odd
21:         TEST   BX,1         ;is B odd?
22:         JZ     END_IF       ;no, even
23: ;then
24:         ADD    DX,AX        ;prod = prod + A
25: END_IF:
26:         SHL    AX,1         ;shift left A
27:         SHR    BX,1         ;Shift right B
28: ;until
29:         JNZ    REPEAT
30:         POP    BX
31:         POP    AX
32:         RET
33: MULTIPLY ENDP
34:         END  MAIN
```

EXERCISE

(MARUTT) ------>

1. Suppose the stack segment is declared as follows:

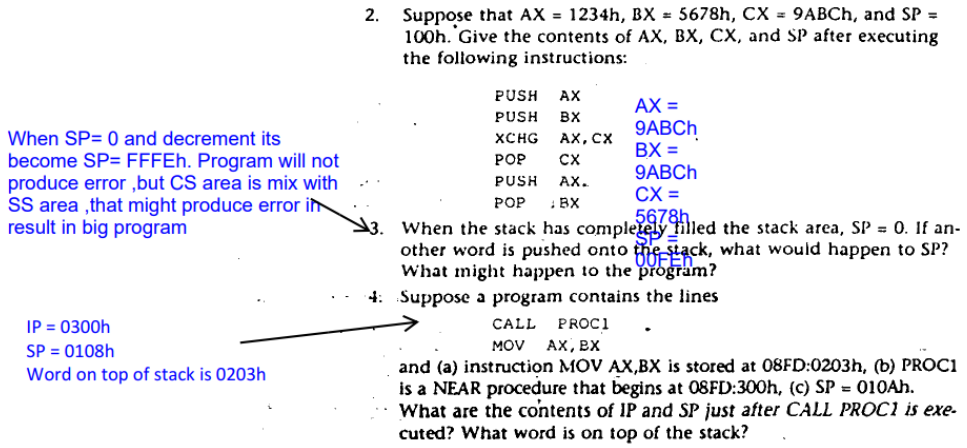
   ```
   .STACK  100h
   ```

   a. What is the hex contents of SP when the program begins? 0100h

   b. What is the maximum hex number of words that the stack may contain? 80h word

a) SP = 0100h

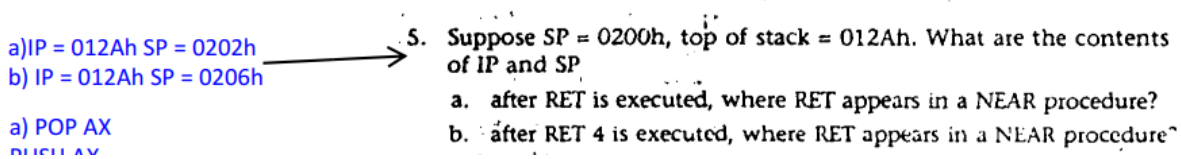
b) 100h byte for stack mean 100h/2 word = 80h word



2) AX = 9ABCh
BX = 9ABCh
CX = 5678h
SP = 00FEh

3) When SP= 0 and decrement it becomes SP= FFFEh. Program will not produce error ,but CS area is mixed with SS area ,that might produce error in result in big program.

4) IP = 0300h
SP = 0108h
Word on top of stack is 0203h



5) a) IP = 012Ah SP = 0202h
b) IP = 012Ah SP = 0206h
Explanation : RET POP_VALUE Mean the stack get more size Original value of SP =202h + POP_VALUE(4) = 206h

6. Write some code to

   a. place the top of the stack into AX, without changing the stack contents.

   b. place the word that is below the stack top into CX, without changing the stack contents. You may use AX.

   c. exchange the top two words on the stack. You may use AX and BX.

a)
```
POP AX
PUSH AX
```
b)
```
POP AX
POP CX
PUSH CX
PUSH AX
```
c)
```
POP AX
POP BX
PUSH AX
PUSH BX
```

7. Procedures are supposed to return the stack to the calling program in the same condition that they received it. However, it may be useful to have procedures that alter the stack. For example, suppose we would like to write a NEAR procedure SAVE_REGS that saves BX,CX,DX,SI,DI,BP,DS, and ES on the stack. After pushing these registers, the stack would look like this:

```
ES content
.
.
.
DX content
CX content
BX content
return_address (offset)
```

Now, unfortunately, SAVE_REGS can't return to the calling program, because the return address is not at the top of the stack.

   a. Devise a way to implement a procedure SAVE_REGS that gets around this problem (you may use AX to do this).

   b. Write a procedure RESTORE_REGS that restores the registers that SAVE_REGS has saved.

a)
```
SAVE_REGS PROC
 POP AX ;save return address
 PUSH BX ;save regs in the stack
 PUSH CX
 PUSH DX
 PUSH SI
 PUSH DI
 PUSH BP
```

```
PUSH DS
PUSH AX ;top of stack is return address
RET
SAVE_REGS ENDP
```

b)
```
RESTORE_REGS PROC
POP AX ; save return address
POP DS ; restore regs from the stack
POP BP
POP DI
POP SI
POP DX
POP CX
POP BX
PUSH AX ; top of stack is return address
RET
RESTORE_REGS ENDP
```

8. Write; a program that lets the user type some text, consisting of words separated by blanks, ending with a carriage return, and displays the text in the same word order as entered, but with the let- __, ters ·in each word reversed. For example, "this is a test" becomes "siht si a tset". Hint: modify program PGM8_2.ASM in section 8.3.

9.; A problem in elementary algebra is to decide if an expression containing several kinds of brackets, such as, [,],{,),(,), is correctly bracketed. This is the case if (a) there are the same number of left and right brackets of each kind, and (b) when a right bracket appears, the most recent preceding unmatched left bracket should be of the same type.

For example,· ·(a~. I?- fc x (d - ~) I J + f} is correctly brackett!d, but (a+ (b -.lex (d - e)) I + f J is not Correct bracketing can be decided by using a stack. The expres- '· The slon is scanned left to right. When a left bracket is encountered, it is pushed onto -the stack. When a right bracket is encountered, the stack is popped (if the stack is empty, there are too many right brackets) and the brackets are compared. If they are of the same type, the scanning continues. If there is a mismatch, the ex·pression is incorrectly bracketed. At the end of the expression, if the stack is empty the expression is correctly bracketed. If the stack Is not empty, there are too many left brackets.

Write a program that lets the user type in an algebraic expression, ending with a carriage return, that contains round (parentheses), square, and curly brackets. As the expression is being typed in, the program evaluates each character. If at any point the expression is incorrectly bracketed (too many right brackets or a mismatch between left and right brackets), the program tells the user to start over. After the· carriage return is typed,

If the expression is correct, the program displays "expression Is correct." If not, the program displays "too many left brackets". In both cases, the program asks the user if he or she wants to continue. If the user types 'Y', the program runs again. Your program docs not need to store the input string, only check it for correctness.

Sample execution:

```
ENTER AN ALGEBRAIC EXPRESSION:
(a + b)]TOO MANY RIGHT BRACKETS. BEGIN AGAIN!
ENTER AN ALGEBRAIC EXPRESSION
(a + [b - c] x d)
EXPRESSION IS CORRECT
TYPE Y IF YOU WANT TO CONTINUE:Y
ENTER AN ALGEBRAIC EXPRESSION:
[a + b x (c - d) - e}BRACKET MISMATCH. BEGIN AGAIN!
ENTER AN ALGEBRAIC EXPRESSION:
((a + [b - {c x (d - e) } ] + f)
TOO MANY LEFT BRACKETS. BEGIN AGAIN!
ENTER AN ALGEBRAIC EXPRESSION:
I'VE HAD ENOUGH
EXPRESSION IS CORRECT
TYPE Y IF YOU WANT TO CONTINUE:N
```

This is the case if

(a) There are the same number of left and right brackets of each kinds,and

(b) When a right bracket appears,the most recent preceding unmatched left bracket should be of the same type.

This can be decided by using stack. The expression is scanned left to right. When a left bracket is encountered,it is pushed onto the stack. When a right bracket is encountered, the stack is poped (if the stack is empty ,there are too many right brackets) and the brackets are compared. If they are of the same type,the scanning continues.If there is a mismatch,the expression is incorrectly bracketed. At the end of the expression ,if the stack is empty ,the expression is correctly bracketed otherwise there are too many left brackets.

Code :

```
.model small
.stack 100h
.data
```

```
  cr equ 0DH  ; cr represents carriage return
  lf equ 0AH  ; lf represents line feed

  msg DB cr,lf,'ENTER AN ALGEBRAIC EXPRESSION: ',cr,lf,'$'
  msg_correct DB  cr,lf,'EXPRESSION IS CORRECT.$'
  msg_left_bracket DB  cr,lf,'TOO MANY LEFT BRACKETS. BEGIN AGAIN!',cr,lf,'$'
  msg_right_bracket DB  cr,lf,'TOO MANY RIGHT BRACKETS. BEGIN AGAIN!',cr,lf,'$'
  msg_mismatch DB  cr,lf,'BRACKET MISMATCH. BEGIN AGAIN!',cr,lf,'$'
  msg_continue DB  cr,lf,'Type Y if you want to Continue: ',cr,lf,'$'

.code

main proc

  mov ax, @data  ;get data segment
  mov ds, ax     ;initialising

start:
  lea dx, msg   ;user prompt
  mov ah, 9
  int 21h

  xor cx, cx      ;initializing cx
  mov ah, 1

input:        ;this label for taking input

   int 21h

   cmp al, 0Dh       ;checking if the enter is pressed or not
        JE end_input

   ;if left bracket, then push on stack
   cmp al, '['
        JE push_data
   cmp al, '{'
        JE push_data
   cmp al, '('
        JE push_data

   ;if right bracket, then pop stack

   cmp al, ')'
        JE parentheses
```

```
    cmp al, '}'
        JE curly_braces
    cmp al, ']'
        JE line_bracket
    jmp input

push_data:
    push ax
    inc cx
    jmp input

parentheses:
        dec cx
        cmp cx, 0
        JL many_right

        pop dx
        cmp dl, '('
        JNE mismatch
        JMP input

curly_braces:
        dec cx
        cmp cx, 0
        JL many_right
        pop dx
        cmp dl, '{'
        JNE mismatch
        JMP input

line_bracket:
        dec cx
        cmp cx, 0
        JL many_right
        pop dx
        cmp dl, '['
        JNE mismatch
        JMP input

end_input:
    cmp cx, 0
    JNE many_left

    mov ah, 9
    lea dx, msg_correct
```

```
    int 21h

    lea dx, msg_continue
    int 21h

    mov ah, 1
    int 21h

    cmp al, 'Y'
    JNE exit
    JMP start

mismatch:
    lea dx, msg_mismatch
    mov ah, 9
    int 21h
    JMP start

many_left:
    lea dx, msg_left_bracket
    mov ah, 9
    int 21h
    JMP start

many_right:
    lea dx, msg_right_bracket
    mov ah, 9
    int 21h
    JMP start

exit:
    mov ah, 4ch
    int 21h

main endp
end main
```

10. The following method can be used to generate random numbers in the range 1 to 32767. Start with any number in this range. Shift left once. Replace bit 0 by the XOR of bits 14 and 15. Clear bit 15.
Write the following procedures:

Write the following procedures:

a. A procedure READ that lets the user enter a binary number and stores it in AX. You may use the code for binary input given in section 7.4.
b. A procedure RANDOM that receives a number in AX and returns a random number in AX.
c. A procedure WRITE that displays AX in binary. You may use the algorithm given in section 7.4.

Write a program that displays a '?', calls READ to read a binary number, and calls RANDOM and WRITE to compute and display 100 random numbers. The numbers should be displayed four per line, with four blanks separating the numbers.

a) Explanation : to make the range between 1 to 32767 we just ignore msb (sign bit /bit-15) by enter 15 bit.

```
INBINARY PROC
 XOR BX,BX
 Mov CX,15
 MOV AH,1
 INT 21H
WHILE_: CMP AL,0DH
 JE END_WHILE
 AND AL,01H
 SHL BX,1
 OR BL,AL
 INT 21H
 LOOP WHILE_
END_WHILE:
RET
INBINARY ENDP
```

b)

by following the algorithm:

```
RANDOM PROC
;input in AX : start with any number in the range
;output in AX (Random number)
 PUSH BX ;save regs
 PUSH DX
 SHL AX,1 ; shift left once
 MOV DX,AX ;get copy
 SHL DX,1 ;make bit-14 be bit-15
```

```
XOR AX,DX ;xor bit-14 with bit-15
TEST DX,8000h ;result of XOR bit-14 with bit-15
JZ NOT_ONE
OR AX,1 ;replace bit-0 with result (1)
JMP CL_B15 ;clear bit-15
NOT_ONE: AND AX,0FFFEH ;replace bit-0 with (0)
CL_B15: AND AX,7FFFH
POP DX ;restore regs
POP BX
RET
RANDOM ENDP
```

c)

```
OUTBIN PROC
;input AX
;output on screen
;we copy number in BX to make sure not changing the
;number because of output function that depend on AX
 MOV BX,AX ;get copy
 MOV CX,16
start: ROL BX,1
 JC ONE
 MOV AH,2
 MOV DL,'0'
 INT 21H
 JMP L1
ONE: MOV AH,2
 MOV DL,'1'
 INT 21H
L1: LOOP start
;put the random number in AX to be new number to
generate another random number
 MOV AX,BX
RET
OUTBIN ENDP
```

The full program looks like :

```
.MODEL SMALL
.stack 64
.DATA
Counter DB 100 ;number to be generated
Counter2 DB 4 ;4 number on same line
```

```
.CODE
Main PROC
 MOV AX,@DATA
 MOV DS,AX
;-------------------
 CALL INBINARY
 MOV AX,BX
Begin: MOV D,4
L10: POP AX
 CALL RANDOM
 CALL OUTBIN
 PUSH AX
;-------------------
 MOV AH,2
 MOV DL,20H ;blank space
 INT 21H
DEC counter2
 CMP counter2,0 ;is there 4-number on line
 JE NEW_LINE
 DEC Counter
 CMP Counter,0
 JNE L10
JMP OUT_
NEW_LINE: MOV AH,2
 MOV DL,0AH ;enter LF
 INT 21H
 JMP Begin
OUT_:
MOV AH,4CH
INT 21H
;------------------------
Main ENDP
;see Appendix C for include pseudo-ops
;Test is folder’s name in c:\ contain procedures
INCLUDE C:\TEST\RANDOM.ASM
INCLUDE C:\TEST\INBINARY.ASM
INCLUDE C:\TEST\OUTBIN.ASM
END Main
```

Arrays and Addressing

Notes

Overview:

Why do we need arrays?

In order to treat a collection of values as a group.

Advantages of arrays

A single name can be given to the whole structure
Any element can be accessed using indexing

What is a two-dimensional Array?

A two-dimensional array is a one-dimensional array whose elements are also one-dimensional arrays.
Most easily manipulated by the based index addressing mode.

XLAT instruction -> This instruction is useful when we want to do data conversion and we use it for encoding and decoding a secret message

One Dimensional Array

Definition: A one-dimensional array is an ordered list of elements of the same type. Ordered does not mean sorted it means there is a first, second element, and so on.

The Pseudo ops used to declare Arrays are DB for byte, DW for word arrays.

```
MSG    DB  'abcde'
W      DW 10,20,30,40,50,60
```

If the offset address assigned to W is 0200h, the array looks like this in memory:

Offset address	*Symbolic address*	*Decimal content*
0200h	W	10
0202h	W+2h	20
0204h	W+4h	30
0206h	W+6h	40
0208h	W+8h	50
020Ah	W+Ah	60

What is the base address of an array?

The address of the array variable is known as the **base address** of the array.

DUP operator: It is used for initialing duplicate values.

Syntax: repeat_count DUP(value)

W DW 100 DUP(0) → this will initiate the W array with 100 elements all will have the value 0. Instead of DUP(0) if we write **DUP(?)** it will create an array of 100 uninitialized words.

Nested DUP : Line DB 5,4, 3 DUP(2 , 3 DUP(0) , 1)

=> 5, 4, 2, 0, 0, 0,1,2, 0,0, 0, 1, 2, 0, 0, 0, 1

Location of Array Elements:

How to access an element in the array?

The address of an array element may be specified by adding a constant to the base address.

Formula to access an element: address of Nth element **= A + (N - 1) ✖ S** [A = base address of the array , N = number of the element , S = 1 or 2 (1 = byte and 2 = word)]

Example : W[10] = W + (10 - 1) * 2 => W + 9 * 2 => W + 18

Addressing Modes:

Definition: The way an operant is specified is known as its addressing mode.

Address modes:

Register mode: Operand is a register
Immediate mode: Operand is a constant;
Direct Mode: operand is a variable

Register addressing mode:

With the Register Addressing mode, the operand to be accessed is specified as residing in an internal register of the 8086.

Example: MOV AX, BX

This code moves the contents of BX (the source operand) to AX (the destination operand).Both the source and the destination operands have been specified as the contents of internal registers of the 8086.

Immediate Addressing

The 8-bit or 16-bit data is provided as a part of the instruction in the immediate addressing mode. The data occupies the immediate next byte to the instruction code in memory.

Example: MOV AL , 055H

In this instruction the operand 055H is an example of a byte-wide immediate source operand. The destination operand, which consists of the contents of AL, uses register addressing. Thus this instruction employs both immediate and register addressing modes.

Direct Addressing (according to prev chapters)

The 16-bit offset address of a memory location is directly provided as a part of the instruction in direct memory addressing mode. Direct address references are usually relative to DS.

Example: MOV CX, [1234H]

This addressing moves the contents of the memory location, which is offset by 1234H from the current value in DS (DS:1234H) into internal register CX. If DS contains 2000H, and the memory location 2000H:1234H and 2000H:1235H contains CDH and AB respectively the execution will load ABCDH into CX. This is because the destination set is 16-bit, so the source has to be 16 bit and so the 2 bytes are combined to form the 16-bit data. The location larger is the higher byte and the smaller location is the lower byte of the 16-bit data.

To change the reference register to any other use a segment override:

mov ch, [es:OverByte]

mov dh, [cs:CodeByte]

mov dh, [ss:StackByte]

mov dh, [ds:DataByte]

An override occupies a byte of machine code which is inserted just before the affected instruction

Four additional Addressing modes in the 8086:

Register Indirect

In this mode, the offset address of the operand is contained in a register.
The register acts as a pointer to the memory location.
Format : [register]
MOV AX , [SI] → the address stores in the SI is used as offset with the DS register . so the address will be DS:0100h [suppose SI has the 0100h]
MOV AX,SI moves 0100h in AX but AX,[SI] moves the value in the DS:0100h value in the AX.
Source register can be **BX,SI or DI** for register indirect addressing (is DS) . BP can be used for SS.

Example 10.3 Write some code to sum in AX the elements of the 10-element array W defined by "W DW 10, 20, 30, 40, 50, 60, 70, 80, 90, 100"

```
        XOR   AX,AX          ;AX holds sum
        LEA   SI,W           ;SI points to array W
        MOV   CX,10          ;CX has number of elements
ADDNOS:
        ADD   AX,[SI]        ;sum = sum + element
        ADD   SI,2           ;move pointer to the next
                             ;element
        LOOP  ADDNOS         ;loop until done
```

Example 10.4 Reverse an array
Solution : The idea ts to exchange the 1st and Nth words, the 2nd and (N-l)st words, and so on. The number of exchanges will be N/2 (rounded down to the nearest Integer If N Is odd). Recall the Nth element ln a word array A has address A+ 2 x (N - 1).

```
REVERSE PROC
;reverses a word array
;input: SI = offset of array
;       BX = number of elements
;output: reversed array
```

```
        PUSH   AX            ;save registers
        PUSH   BX
        PUSH   CX
        PUSH   SI
        PUSH   DI
;make  DI point to nth word
        MOV    DI,SI         ;DI pts to 1st word
        MOV    CX,BX         ;CX = n
        DEC    BX            ;BX = n-1
        SHL    BX,1          ;BX = 2 x (n-1)
        ADD    DI,BX         ;DX pts to nth word
        SHR    CX,1          ;CX = n/2 = no. of swaps to do
;swap  elements
XCHG_LOOP:
        MOV    AX,[SI]       ;get an elt in lower half of array
        XCHG   AX,[DI]       ;insert in upper half
        MOV    [SI],AX       ;complete exchange
        ADD    SI,2          ;move ptr
        SUB    DI,2          ;move ptr
        LOOP   XCHG_LOOP     ;loop until done
        POP    DI            ;restore registers
        POP    SI
        POP    CX
        POP    BX
        POP    AX
        RET
REVERSE      ENDP
```

Based and 3. Indexed

Definition: In these modes, the operand's offset address Is obtained by adding a number called a displacement to the contents of a register. Displacement can be :

the offset address or a variable
a constant (positive or negative)
the offset address or a variable plus or minus a constant

When it is called indexed and when it is called based?

=> The addressing mode is called based if BX (base register) or BP(base pointer) is used. The addressing mode is called indexed if SI (source index) or DI (destination index) is used

Syntax:
[register + displacement]
[displacement + register]
[register] + displacement
displacement + [register]
displacement [register]
Example :

Ex. Suppose W is a word array, and BX contains 4

MOV AX, W[BX]

• The displacement is the offset address of variable W.

• The instruction moves the element at address W + 4 to AX. (this is the third element in the array)

• The instuction could also have been written in any of these forms:

MOV AX, [W+BX]

MOV AX, [BX+W]

MOV AX, W+[BX]

MOV AX, [BX]+W

Array sum with base mode :

```
        XOR   AX,AX          ;AX holds sum
        XOR   BX,BX          ;clear base register
        MOV   CX,10          ;CX has number of elements
ADDNOS:
        ADD   AX,W[BX]       ;sum = sum + element
        ADD   BX,2           ;index next element
        LOOP  ADDNOS         ;loop until done
```

Example 10.7 Replace each lowercase letter in the following string by its upper case equivalent. Use index addressing mode.

```
MSG DB          'this is a message'
```

Solution:

```
        MOV   CX,17          ;no. of chars in string
        XOR   SI,SI          ;SI indexes a char
TOP:
        CMP   MSG[SI],' '    ;blank?
        JE    NEXT           ;yes, skip over
        AND   MSG[SI],0DFh   ;no, convert to upper case
NEXT:
        INC   SI             ;index next byte
        LOOP  TOP            ;loop until done
```

Consider the following instructions:

mov [BX], 100

add [SI], 20

inc [DI]

Where BX, SI, and DI contain memory addresses The size of the memory operand is not clear to the assembler BX, SI, and DI can be pointers to BYTE, WORD

Solution:

use PTR operator to clarify the operand size

mov BYTE PTR [BX], 100 ; BYTE operand in memory

add WORD PTR [SI], 20 ; WORD operand in memory

Label Pseudo-op:

```
Money   LABEL  word
Dollars   DB      1Ah
Cents     DB      52h
```

This declaration types Money as a word variable,and the components DOL· LARS and CENTS as byte variables, with MONEY and DOLLARS being assigned the same address by the assembler. The instruction

MOV AX,MONEY ;AL = dollars , AH = cents

Example 10.9 Suppose the following data are declared:

```
.DATA
A DW  1234h
B LABEL BYTE
  DW 5678h
C LABEL WORD
C1 DB 9Ah
C2 DB 0BCh
```

Tell whether the following instructions are legal, and if so, give the numb moved.

Instruction

a. `MOV AX,B`
b. `MOV AH,B`
c. `MOV CX,C`
d. `MOV BX, WORD PTR B`
e. `MOV DL, WORD PTR C`
f. `MOV AX, WORD PTR C1`

Solution:

a. illegal—type conflict
b. legal, 78h
c. legal, 0BC9Ah
d. legal, 5678h
e. legal, 9Ah
f. legal, 0BC9Ah

Segment override :

By default the DI , SI and BX specifies offset the the address of the DS. But we can explicitly specify the segment to override it.

MOV AX,ES: [SI] → it will add the offset to the ES.

Sort Array:

Selectsort algorithm

```
i = N
FOR N-1 times DO
   Find the position k of the largest element
      among A[1]..A[i]
(*) Swap A[k] and A[i]
   i = i-1
END_FOR
```

Step (*) will be handled by a procedure SWAP. The code for the procedures is the following (we'll suppose the array to be sorted is a byte array):

Program Listing PGM10_2.ASM

```
1:   SELECT   PROC
2:   ;sorts a byte array by the selectsort method
3:   ;input:   SI = array offset address
4:   ;            BX = number of elements
5:   ;output: SI = offset of sorted array
6:   ;uses: SWAP
7:             PUSH   BX
8:             PUSH   CX
9:             PUSH   DX
10:            PUSH   SI
11:            DEC    BX              ;N = N-1
12:            JE     END_SORT        ;exit if 1-elt array
13:            MOV    DX,SI           ;save array offset
14:  ;for N-1 times do
15:  SORT_LOOP:
16:            MOV    SI,DX           ;SI pts to array
17:            MOV    CX,BX           ;no. of comparisons to make
18:            MOV    DI,SI           ;DI pts to largest element
19:            MOV    AL,[DI]         ;AL has largest element
20:  ;locate biggest of remaining elts
21:  FIND_BIG:
22:            INC    SI              ;SI pts to next element
23:            CMP    [SI],AL         ;is new element > largest?
24:            JNC    NEXT            ;no, go on
25:            MOV    DI,SI           ;yes, move DI
26:            MOV    AL,[DI]         ;AL has largest element
27:  NEXT:
28:            LOOP   FIND_BIG        ;loop until done
29:  ;swap biggest elt with last elt
30:            CALL   SWAP            ;Swap with last elt
31:            DEC    BX              ;N = N-1
32:            JNE    SORT_LOOP       ;repeat if N <> 0
33:  END_SORT:
34:            POP    SI
35:            POP    DX
36:            POP    CX
37:            POP    BX
38:            RET
39:  SELECT   ENDP
40:  SWAP     PROC
41:  ;swaps two array elements
42:  ;input:  SI = one element
43:  ;          DI = other element
44:  ;output:  exchange elements
45:            PUSH   AX              ;save AX
46:            MOV    AL,[SI]         ;get A[i]
47:            XCHG   AL,[DI]         ;place in A[k]
48:            MOV    [SI],AL         ;put A[k] in A[i]
49:            POP    AX              ;restore AX
50:            RET
51:  SWAP      ENDP
```

Two-dimensional Array:

Row major order: the row 1 elements are stored, followed by the row 2 elements, then the row 3 elements, and so on.

Column major order: column after column is stored.

Locating an Element in a Two-Dimensional Array:

Row Major:

To find the location of A[i, j],

1. Find where row i begins.

2. Find the location of the jth element in that row.

Step 1 : Row 1 begins at A, has N elements and each element has size S. so the 2nd row begins at A+ N*S And row 3 begins at A + 2 * N * S . So we can say that the i row begins at **A + (i - 1) * N * S.**

Step 2: Now the second step j th element is in (j - 1) * S position

So if we combine the both step 1 and step 2 we get = A + ((i-1)*N + (j -1)) * S

Column major:

If A is an M x N array, with element size S, stored in column-major, then :

A[i,j] has address A + ((i - 1) + (j - 1) x M) x S

Example 10.12 Suppose A is an $M \times N$ word array stored in row-major order.

1. Where does row i begin?
2. Where does column j begin?
3. How many bytes are there between elements in a column?

Solution:

1. Row i begins at A[i, 1]; by formula (1) its address is $A + (i - 1) \times N \times 2$.
2. Column j begins at A[1, j]; by formula (1) the address is $A + (j - 1) \times 2$.
3. Because there are N columns, there are $2 \times N$ bytes between elements in any given column.

Based-Indexed Addressing mode:

This mode is used for 2-dimensional arrays.

In this mode, the offset address of the operand is the sum of

the contents of a base register (BX or Bl')
the contents of an index register (SI or DI)
optionally, a variable's offset address
optionally, a constant (positive or negative)

1. variable[base_register] [index_register]
2. [base_Resister + index_register + constant + variable]
3. Variable[base_register + index_register + constant]

4. Constant[base_register + index_register + variable]

Example 10.13 Suppose A is a 5- × 7-word array stored in row-major order. Write some code to (1) clear row 3, (2) clear column 4. Use based indexed mode.

Solution:

1. From example 10.12, we know that in an *M*- × *N*-word array A, row i begins at A + (i – 1) × *N* × 2. Thus in a 5 × 7 array, row 3 begins at A + (3 – 1) × 7 × 2 = A + 28. So we can clear row 3 as follows:

```
        MOV X,28;           BX indexes row 3
        XOR SI,SI           ;SI will index columns
        MOV CX,7            ;number of elements in a row
CLEAR:
        MOV A[BX][SI],0     ;clear A[3,j]
        ADD SI,2            ;go to next column
        LOOP CLEAR          ;loop until done
```

2. Again from example 10.12, column j begins at A + (j – 1) × 2 in an *M*- × *N*-word array. Thus column 4 begins at A + (4 – 1) × 2 = A + 6. Since A is a seven-column word array stored in row-major order, to get to the next element in column 4 we need to add 7 × 2 = 14. We can clear column 4 as follows:

```
        MOV SI,6            ;SI will index column 4
        XOR BX,BX           ;BX will index rows
        MOV CX,5            ;number of elements in a column
CLEAR:
        MOV A[BX][SI],0     ;clear A[i,4]
        ADD BX,1            ;go to next row
        LOOP CLEAR          ;loop until done
```

Example: Average subject score

Suppose a class of five students is given four exams. The results are recorded as follows:

	Test 1	*Test 2*	*Test 3*	*Test 4*
MARY ALLEN	67	45	98	33
SCOTT BAYLIS	70	56	87	44
GEORGE FRANK	82	72	89	40
BETH HARRIS	80	67	95	50
SAM WONG	78	76	92	60

We will write a program to find the class average on each exam. To do this, we sum the entries in each column and divide by 5.

Algorithm

```
1. j = 4
2. REPEAT
3.    sum the scores in column j
4.    divide sum by 5 to get the average in column j
4.    j = j-1
5. UNTIL j = 0
```

We choose to start summing in column 4 because it makes the code a little shorter. Step 3 may be broken down further as follows:

```
sum[j] = 0
i = 1
FOR 5 times DO
   sum[j] = sum[j] + score[i,j]
   i = 1+1
END_FOR
```

Program Listing PGM10_4.ASM

```
0:   TITLE PGM10_4: CLASS AVERAGE
1:   .MODEL   SMALL
2:   .STACK   100H
3:   .DATA
4:   FIVE     DW      5
5:   SCORES   DW      67,45,98,33   ;Mary Allen
6:            DW      70,56,87,44   ;Scott Baylis
7:            DW      82,72,89,40   ;George Frank
8:            DW      80,67,95,50   ;Beth Harris
9:            DW      78,76,92,60   ;Sam Wong
10:  AVG      DW      5 DUP (0)
11:  .CODE
12:  MAIN     PROC
13:           MOV     AX,@DATA
14:           MOV     DS,AX         ;initialize DS
15:  ;j=4
16:           MOV     SI,6          ;col index, initially col
17:  REPEAT:
18:           MOV     CX,5          ;no. of rows
19:           XOR     BX,BX         ;row index, initially 1
20:           XOR     AX,AX         ;col_sum, initially 0
21:  ;sum scores in column j
22:  FOR:
23:           ADD     AX,SCORES[BX+SI];col_sum=col_sum + score
24:           ADD     BX,8          ;index next row
25:           LOOP    FOR           ;keep adding scores
26:  ;endfor
27:  ;compute average in column j
28:           XOR     DX,DX         ;clear high part of divnd
29:           DIV     FIVE          ;AX = average
30:           MOV     AVG[SI],AX    ;store in array
31:           SUB     SI,2          ;go to next column
32:           ;until j=0
33:           JNL     REPEAT        ;unless SI < 0
34:  ;dos exit
35:           MOV     AH,4CH
36:           INT     21H
37:  MAIN     ENDP
38:           END   MAIN
```

The test scores are stored in a two-dimensional array (lines 5–9). In lines 22–25, a column is summed and the total placed in the array AVG. In lines 28–30, this total is divided by 5 to compute the column average.

Multiprocessor Configuration Overview

Multiprocessor means a multiple set of processors that executes instructions simultaneously. There are three basic multiprocessor configurations.

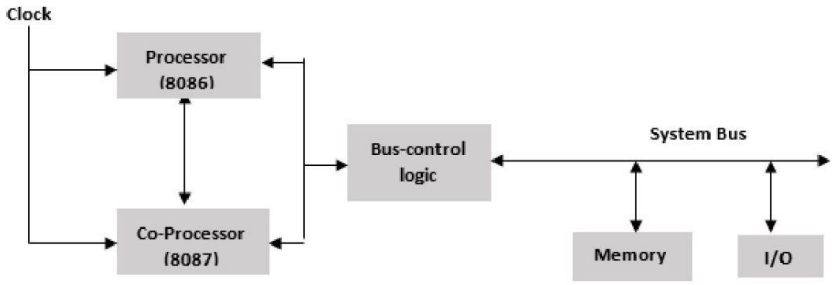
- Coprocessor configuration
- Closely coupled configuration
- Loosely coupled configuration

Coprocessor Configuration

A Coprocessor is a specially designed circuit on microprocessor chip which can perform the same task very quickly, which the microprocessor performs. It reduces the work load of the main processor. The coprocessor shares the same memory, IO system, bus, control logic and clock generator. The coprocessor handles specialized tasks like mathematical calculations, graphical display on screen, etc.

The 8086 and 8088 can perform most of the operations but their instruction set is not able to perform complex mathematical operations, so in these cases the microprocessor requires the math coprocessor like Intel 8087 math coprocessor, which can easily perform these operations very quickly.

Block Diagram of Coprocessor Configuration



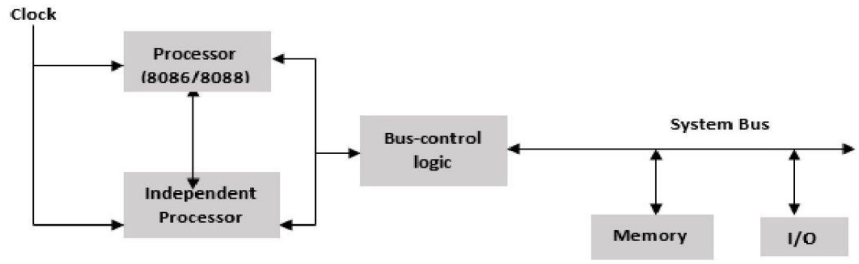
How is the coprocessor and the processor connected?

- The coprocessor and the processor is connected via TEST, RQ-/GT- and QS_0 & QS_1 signals.
- The TEST signal is connected to BUSY pin of coprocessor and the remaining 3 pins are connected to the coprocessor's 3 pins of the same name.
- TEST signal takes care of the coprocessor's activity, i.e. the coprocessor is busy or idle.
- The RT-/GT-is used for bus arbitration.
- The coprocessor uses QS_0 & QS_1 to track the status of the queue of the host processor.

Closely Coupled Configuration

Closely coupled configuration is similar to the coprocessor configuration, i.e. both share the same memory, I/O system bus, control logic, and control generator with the host processor. However, the coprocessor and the host processor fetches and executes their own instructions. The system bus is controlled by the coprocessor and the host processor independently.

Block Diagram of Closely Coupled Configuration



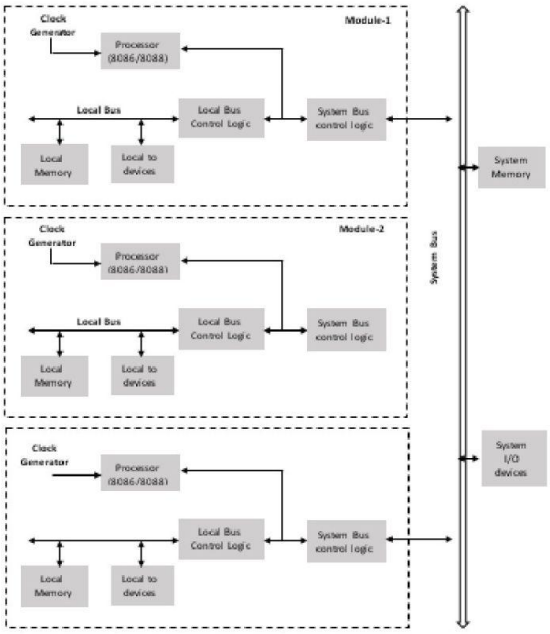
How is the processor and the independent processor connected?

- Communication between the host and the independent processor is done through memory space.
- None of the instructions are used for communication, like WAIT, ESC, etc.
- The host processor manages the memory and wakes up the independent processor by sending commands to one of its ports.
- Then the independent processor accesses the memory to execute the task.
- After completion of the task, it sends an acknowledgement to the host processor by using the status signal or an interrupt request.

Loosely Coupled Configuration

Loosely coupled configuration consists of the number of modules of the microprocessor based systems, which are connected through a common system bus. Each module consists of their own clock generator, memory, I/O devices and are connected through a local bus.

Block Diagram of Loosely Coupled Configuration



Advantages

- Having more than one processor results in increased efficiency.
- Each of the processors have their own local bus to access the local memory/I/O devices. This makes it easy to achieve parallel processing.
- The system structure is flexible, i.e. the failure of one module doesn't affect the whole system failure; faulty module can be replaced later.

- **Configurations of each categories :**

1. Coprocessor Configuration: In this configuration, a specialized coprocessor is added to a main processor to offload specific tasks, such as mathematical calculations or graphics rendering. The coprocessor works in tandem with the main processor, with each processor handling different types of tasks. This configuration can result in significant performance gains for certain applications, but can also add complexity and cost to the system.
2. Closely Coupled Configuration: In this configuration, multiple processors are closely integrated and share resources such as memory and I/O. The processors work in close collaboration with each other and communicate with low latency, resulting in high performance. This configuration is suitable for systems with high performance and low latency requirements, but can be complex and difficult to manage.
3. Loosely Coupled Configuration: In this configuration, multiple processors operate independently and communicate with each other through a shared memory or a

communication network. Each processor works on different tasks, and the system can be easily scaled by adding or removing processors. This configuration is suitable for systems that need to be easily scalable and flexible, but may have higher communication overhead and lower performance compared to closely coupled configurations.

- Advantages and disadvantages of these categories ?

Advantages of Coprocessor Configuration:

1. Improved performance: Coprocessors can offload specific tasks from the main processor, which can result in significant performance gains for certain applications.
2. Specialized functionality: Coprocessors are designed to handle specific types of tasks, such as mathematical calculations or graphics rendering, which can improve performance in those areas.
3. Scalability: Coprocessors can be added or removed from a system as needed, allowing for scalability and flexibility in the system's capabilities.

Disadvantages of Coprocessor Configuration:

1. Complexity: Coprocessors can add complexity to a system, requiring additional configuration and management.
2. Cost: Coprocessors can be expensive, especially for high-end models with specialized functionality.

Advantages of Closely Coupled Configuration:

1. High performance: Closely coupled configurations allow for close integration between the processors, resulting in high performance and low latency.
2. Efficient use of resources: Closely coupled configurations can efficiently use shared resources such as memory and I/O.
3. Low communication overhead: With closely coupled configurations, communication between processors is fast and efficient, with low overhead.

Disadvantages of Closely Coupled Configuration:

1. Complexity: Closely coupled configurations can be complex and difficult to manage, requiring specialized knowledge and skills.
2. Limited scalability: Closely coupled configurations can be difficult to scale beyond a certain number of processors, due to issues with communication and resource sharing.

Advantages of Loosely Coupled Configuration:

1. Scalability: Loosely coupled configurations can be easily scaled to support large numbers of processors, as each processor operates independently.
2. Flexibility: Loosely coupled configurations allow for different types of processors with varying capabilities to be used together.

Disadvantages of Loosely Coupled Configuration:

1. Lower performance: Loosely coupled configurations can have higher communication overhead, resulting in lower performance and increased latency.

2. Resource contention: With loosely coupled configurations, processors may compete for resources such as memory and I/O, which can lead to contention and reduced performance.

- Which one is better ?
  The choice of which configuration is better depends on the specific use case and requirements of the system. Each configuration has its own advantages and disadvantages, and the optimal choice will depend on factors such as performance requirements, scalability needs, and budget constraints.
  For example, Coprocessor Configuration may be beneficial for applications that require specialized functionality, such as mathematical calculations or graphics rendering, and where performance gains are important. On the other hand, Closely Coupled Configuration may be more suitable for systems that require high performance and low latency, but are limited in scalability. Loosely Coupled Configuration, on the other hand, may be a good choice for systems that need to be easily scalable and flexible, but do not require high performance or low latency.
  In summary, the choice of configuration will depend on the specific needs and requirements of the system, and there is no one-size-fits-all answer to which configuration is better.

- Advantages and disadvantages of multiprocessors ?

Advantages:

1. Improved performance: Multiprocessor configurations can improve system performance by allowing multiple processors to work on different tasks simultaneously.
2. Increased scalability: Multiprocessor configurations can be easily scaled up to support larger workloads and higher performance requirements.
3. Redundancy and fault tolerance: Multiprocessor configurations can provide redundancy and fault tolerance, as a system can continue to operate even if one processor fails.
4. Cost savings: In some cases, a multiprocessor configuration can be more cost-effective than using a single high-performance processor, as it allows for more efficient use of resources and better scalability.
5. Better utilization of resources: A multiprocessor configuration allows for better utilization of system resources, as multiple processors can work on different tasks simultaneously.

Disadvantages:

1. Increased complexity: Multiprocessor configurations can be more complex and difficult to manage than single-processor systems, requiring specialized knowledge and skills.
2. Increased power consumption: Multiprocessor configurations can consume more power than single-processor systems, leading to higher energy costs.
3. Synchronization issues: In a multiprocessor configuration, synchronization issues can arise when multiple processors need to access shared resources, leading to potential performance issues and delays.

4. Communication overhead: Multiprocessor configurations can have higher communication overhead, as processors need to communicate with each other and synchronize their work, leading to increased latency and reduced performance.
5. Scalability limitations: Some multiprocessor configurations may have scalability limitations, as adding additional processors may not result in linear improvements in performance, due to communication and synchronization issues.

Tri-State Logic

THEORY :

How does the concept of Tri-state arise ?

- what happens when multiple devices have their outputs connected to it. For example two or more logic gates' output is connected together. One of the devices might output a logic '1' and simultaneously the other might output a '0'. Leading to short circuit currents and pulling the potential of the line to some indeterminate state.

What states does a tri-state have ?

- Tri-state, or three-state, is a type of digital logic circuit that allows a signal to be in one of three states: logic high (1), logic low (0), or high-impedance (or "floating"). The high-impedance state essentially means that the signal is disconnected from the circuit, allowing other signals to pass through without interference.

Explain one of the main factors of using Tri-State ?

- When there are two or more devices in a circuit and they are using shared channel of data transfer i.e. Data Bus, and one of the devices produces output - 0 and the other produces 1 , now producing two different results that causes short-circuit in that single shared channel. That's why we use Tri-state Logic for creating high impedance.

What is the result of using Tri-state ?

- All the devices will have tri-state drives at their output. Now when enabled, the outputs can either be '1' or '0' but when disabled they go into a third state •i.e. tri-state, They can neither source or sink. Such outputs can be connected in parallel and they will not affect the existing condition of the bus

- When we connect multiple memory devices together on the data bus then the only device which is being addressed will enable the output buffers out of tri-state in response to the read signal.
- The main advantage of using Tri-state is that it allows multiple devices to share a common data bus without causing conflicts. By using Tri-state drivers, a device can place its signal on the data bus when it needs to transmit data, and then release the bus by putting its driver in high-impedance state when it's not transmitting.

Mention some advantages and disadvantages of using Tri-state ?

- Advantages of using Tri-state:
    1. Efficient use of shared buses: Tri-state logic is commonly used to share a bus between multiple devices, as it allows each device to place its signal on the bus when it needs to transmit data, and then release the bus when it's not transmitting.
    2. Reduced power consumption: When a Tri-state output is in high-impedance state, it consumes very little power, which can help to reduce overall power consumption in a digital system.
    3. Flexibility: The Tri-state output provides flexibility in connecting multiple devices to the same bus and allows multiple devices to share the same data line.
- Disadvantages of using Tri-state:
    1. Complexity: Using Tri-state requires additional circuitry to enable the high-impedance state, which can add complexity and cost to a digital system.
    2. Delay: When a Tri-state output is switched from high-impedance state to an active state, there may be a delay before the output signal stabilizes. This delay can cause issues with timing and can potentially result in errors.
    3. Signal Integrity: If multiple devices connected to a shared bus are driving signals simultaneously, it can lead to signal integrity issues such as crosstalk, noise, and other signal distortions.

Mention some Pins in 8085 that have Tri-state Type ?

Table 2.11 8085 Signal Description Summary

Pin name	Description	Type
AD0-AD7	Address/data bus	Bidirectional, tristate
A8-A15	Address bus	Output, tristate
ALE	Address latch enable	Output, tristate
$\overline{RD}$	Read control	Output, tristate
$\overline{WR}$	Write control	Output, tristate
IO / $\overline{M}$	I/O or memory indicator	Output, tristate

Mention some Pins in 8086 that have Tri-state Type ?

1. AD0-AD15 (Address/Data bus): These are the pins used for bi-directional communication between the microprocessor and memory or I/O devices.
2. DT/R (Data Transmit/Receive): This pin is used for transmitting or receiving data between the microprocessor and external devices.
3. DEN (Data Enable): This pin is used to enable or disable the Tri-state output of the DT/R pin.
4. HOLD (Hold Request): This pin is used for interrupting the microprocessor by external devices.
5. HLDA (Hold Acknowledge): This pin is used to acknowledge the HOLD request by external devices.

What does it mean that a pin in a microprocessor is a tri-state type ?

- When a pin in a microprocessor is a Tri-state type, it means that the pin can have three different states: logic high (1), logic low (0), or high-impedance (Z or floating).
- The pin can be placed in a high-impedance state, which means that it is effectively disconnected from the circuit and can float to any voltage level.
- It allows multiple devices to share a common bus without interfering with each other's signals.
- Tri-state type pins are commonly used in microprocessors to enable efficient communication between devices and to prevent conflicts between multiple signals that may try to drive the same output.

Suppose, there are multiple devices in a shared channel and produce output at the same time. Will there be any problem ?

- Yes

Suppose, there is a device in a shared channel and produces output and that received by multiple devices. Will there be any problem ?

- No

What's a Tri-state buffer ?

- A Tri-state buffer is a digital logic circuit that acts as a buffer between an input signal and an output signal. It allows the input signal to pass through to the output when the enable input is asserted (logic high), and it disconnects the output from the input when

the enable input is de-asserted (logic low). In other words, the Tri-state buffer can drive the output signal onto a shared bus or other connected devices when it is enabled, and it can release the bus by entering high-impedance state when it is disabled.

- The Tri-state buffer is commonly used to enable or disable multiple devices connected to a shared bus without causing conflicts or interference. By using Tri-state buffers, each device can place its signal on the bus when it needs to transmit data and then release the bus when it's not transmitting, allowing other devices to take their turn. Tri-state buffers are also useful for preventing contention between multiple signals that may try to drive the same output.

What's Tri-state enable buffer/ Tri-state enable database ?

- When a diode/output pin has the ability to generate three state: 1,0 and Disconnected state, then we call it Tri-state enable buffer/ Tri-state enable Databus. When a pin is disconnected , it means the device is disconnected from circuit perspective.

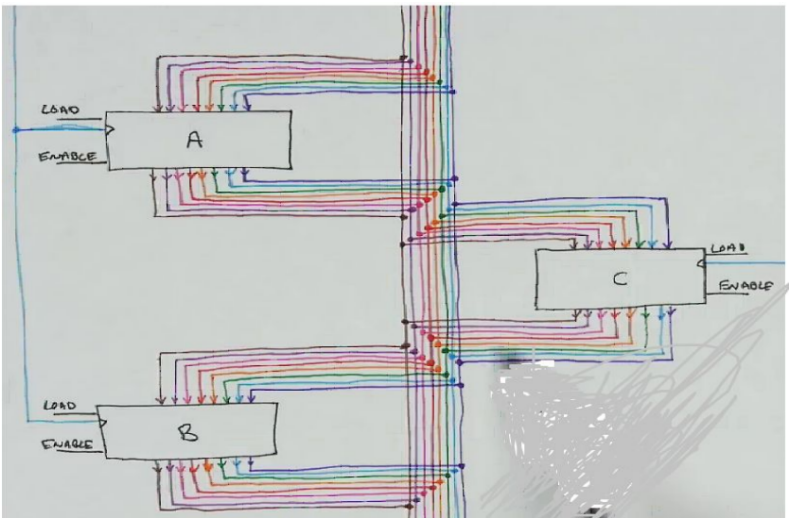
What does tri state mean in an 8085 microprocessor?

- The 8085 microprocessor is a dual-state logic device, which means its output pins can be in one of two states: high or low. However, some pins of the 8085 microprocessor also have tri-state capabilities.

  In the case of the 8085 microprocessor, the tristate means that the output pin of the device can be put in a third high-impedance state or "disconnected" state. In this state, the output pin is effectively disconnected from the bus and does not drive any sink or source currents into the data bus.
  The tristate capability of the 8085 microprocessor is useful in situations where multiple devices share the same communication channel or bus. By using tri-state outputs, multiple devices can share the same bus without interference, allowing for efficient communication between the devices. The tristate feature is typically used in conjunction with external buffers to interface the microprocessor with other devices or systems.

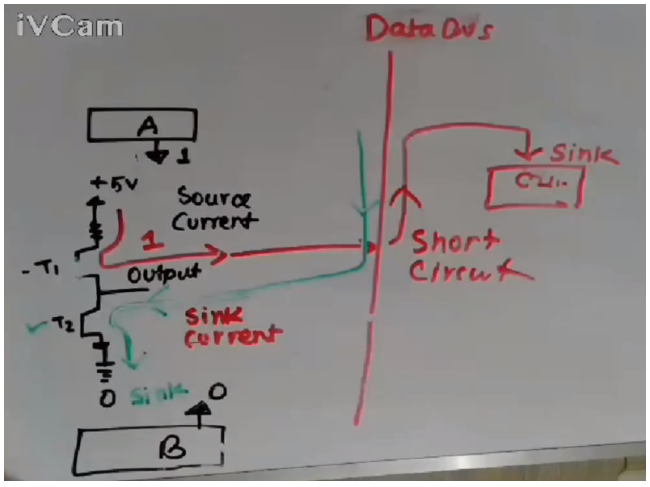
DIAGRAM :

Show a circuit diagram where we can say there is a possibility of conflict among devices and that conflict can be solved using Tri-state Logic ?



Explain using a circuit diagram the short-circuit and using dual state it is not possible to remove / prevent conflict ?

- In a shared channel or bus, such as a data bus, multiple devices can connect and share the same communication path. When one device transmits a signal (logic 1 or 0), it acts as a current source, and when another device receives the signal, it acts as a current sink.



- In the scenario, device A is acting as a current source and device C is acting as a current sink. The output pins of the devices are typically connected to transistors. When a logical 1 is transmitted, the transistor is turned ON and allows current to flow from the device, acting as a current source.

- And when B is transmitting 0, some amount of voltage current enters the device B via the shared channel (data bus). So here we can see a short circuit is generated as in the shared channel because there is forward current as well as the reverse current.
- So regarding device B, it should be in a high-impedance state (Z-state) when it is not transmitting any signals. This means that the output pin of device B is effectively disconnected from the bus and does not drive any sink or source currents into the data bus.

In Tri-state logic, the third state is known as the high-impedance state or the "disconnected" state. When a device's output pin is in this state, it is effectively disconnected from the bus and does not drive any sink or source currents into the data bus. This allows multiple devices to share the same bus without interference and enables efficient communication between the devices.

Draw a circuit diagram or a diode or Tri-state buffer gate that can generate tri-state.And explain this Tri-state buffer how it works ?

- We have to keep a third input in the diode which is known as the control signal C
- Whatever the value is in the input and the value of the voltage +5V or 0V, if the enable = 0 , the circuit is open, and no current will be pass to outside
- It doesn't enable the circuit and pass the output to the outside of the circuit

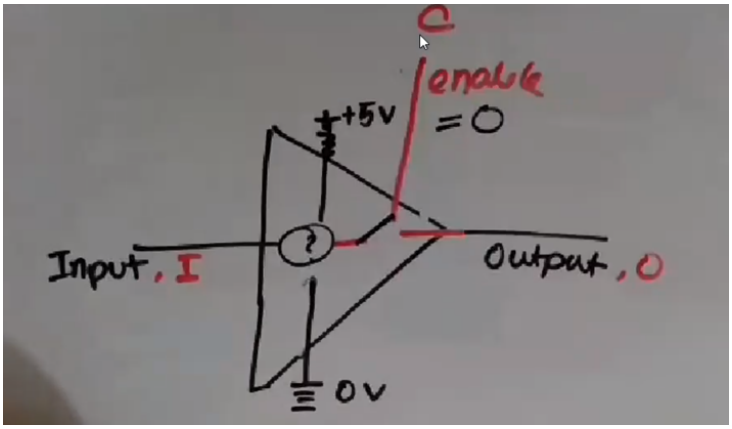


Fig : Tri-state buffer gate when enable = 0

- And in this case i.e. enable = 0 ; the output will be D (disconnected)

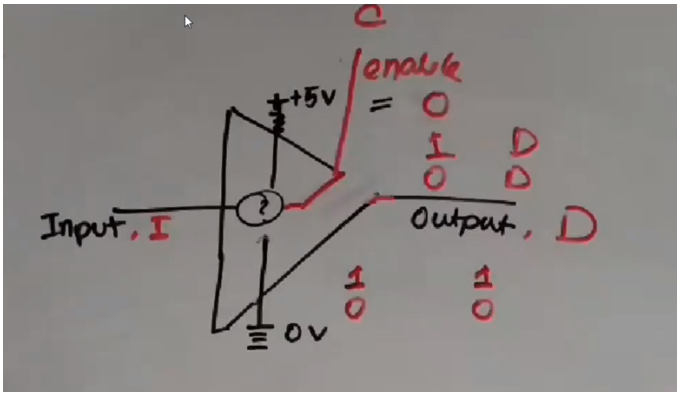
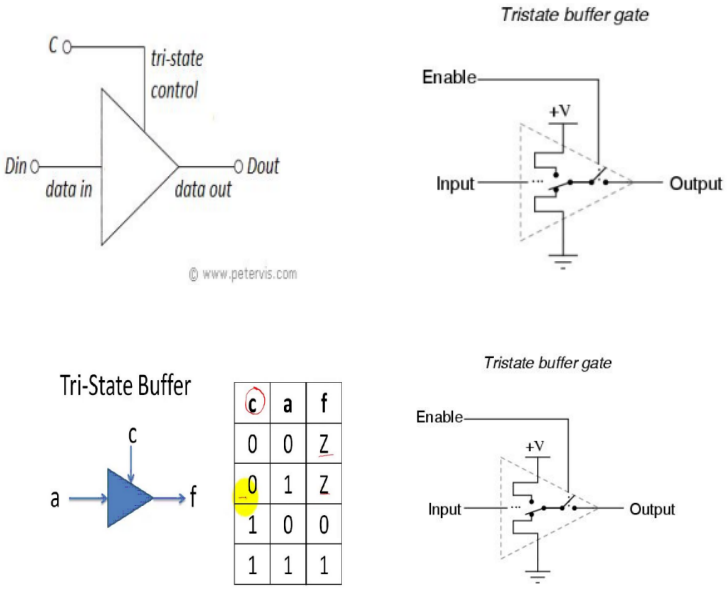


Fig : Tri-state buffer gate

- But when the enable = 1 , it means the circuit is closed and what the input is it will reflect to the output.
- If the input is I = 1, the output will be 1 and when its I = 0, the output will be 0



c	a	f
0	0	Z
0	1	Z
1	0	0
1	1	1

Fig : Tri-state buffer gate and truth table with the enable

- When a(input) = 1 and enable = 1, f (output) = 1 and a(input) = 0 and enable = 1, f (output) = 0
- When a(input) = 1 and enable = 0, f (output) = Z(high impedance) and a(input) = 0 and enable = 0, f (output) = Z(high impedance)

Important Questions

1. What is the purpose of a Tri State Buffer in microprocessor circuits? How does it work?

Ans: A Tri-State Buffer is a digital logic circuit that is commonly used in microprocessor circuits. Its purpose is to control the flow of data on a bus or a data line.

The term "tri-state" refers to the three possible output states that the buffer can have: high, low, or high impedance. When the buffer is in the high or low state, it behaves like a regular buffer, passing the input signal to the output. However, when it is in the high impedance state, the buffer effectively disconnects the output from the bus, allowing other devices to drive the bus without interference.

This feature is particularly useful in multi-master systems, where multiple devices share a common bus. In such systems, a device can place its output in high impedance mode when it is not actively driving the bus, allowing other devices to use the bus without interference. This can help avoid data collisions and improve system reliability.

In summary, the Tri-State Buffer is used to control the flow of data on a bus or data line in a microprocessor circuit. It can selectively allow data to pass through or isolate the output from the bus, depending on the state of its control input signal.

DMA with Diagram

THEORY :

WHAT IS DMA IN MICROPROCESSOR?

- Direct Memory Access (DMA) transfers the block of data between the memory and peripheral devices of the system, without the participation of the processor.
- The unit that controls the activity of accessing memory directly is called a DMA controller .The processor relinquishes the system bus for a few clock cycles. So, the DMA controller can accomplish the task of data transfer via the system bus.
- For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. Video cards that support DMA can also access the system memory and process graphics without needing the CPU. Ultra DMA hard drives use DMA to transfer data faster than previous hard drives that required the data to first be run through the CPU. In order for devices to use direct memory access, they must be assigned to a DMA channel. Each type of port on a computer has a set of DMA channels that can be assigned to each connected device. For example, a PCI controller and a hard drive controller each have their own set of DMA channels.

What is DMA and Why is it used?

- Direct memory access (DMA) is a mode of data transfer between the memory and I/O devices. This happens without the involvement of the processor. We have two other methods of data transfer, programmed I/O and Interrupt driven I/O. Let’s revise each and get acknowledge with their drawbacks. In programmed I/O, the processor keeps on scanning whether any device is ready for data transfer. If an I/O device is ready, the processor fully

dedicates itself in transferring the data between I/O and memory. It transfers data at a high rate, but it can't get involved in any other activity during data transfer. This is the major drawback of programmed I/O. In Interrupt driven I/O, whenever the device is ready for data transfer, then it raises an interrupt to processor. Processor completes executing its ongoing instruction and saves its current state. It then switches to data transfer which causes a delay. Here, the processor doesn't keep scanning for peripherals ready for data transfer. But, it is fully involved in the data transfer process. So, it is also not an effective way of data transfer. The above two modes of data transfer are not useful for transferring a large block of data. But, the DMA controller completes this task at a faster rate and is also effective for transfer of large data block.

- The DMA controller transfers the data in three modes:

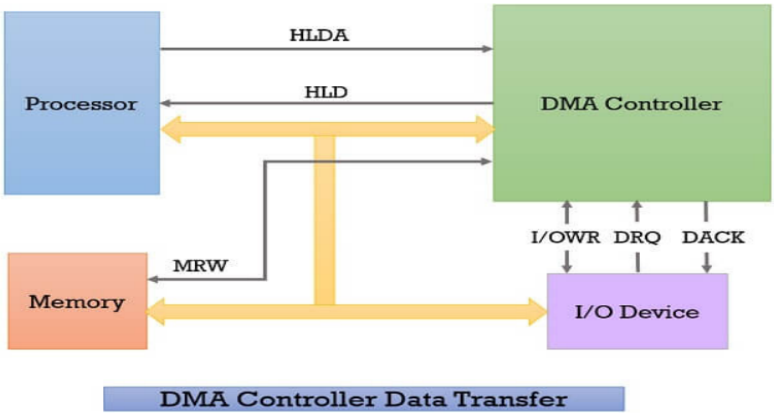
1. Burst Mode: Here, once the DMA controller gains the charge of the system bus, then it releases the system bus only after completion of data transfer. Till then the CPU has to wait for the system buses.
2. Cycle Stealing Mode: In this mode, the DMA controller forces the CPU to stop its operation and relinquish control over the bus for a short term to the DMA controller. After the transfer of every byte, the DMA controller releases the bus and then again requests for the system bus. In this way, the DMA controller steals the clock cycle for transferring every byte.
3. Transparent Mode: Here, the DMA controller takes the charge of system bus only if the processor does not require the system bus

- Direct Memory Access Advantages and Disadvantages
  - Advantages:
    - 1. Transferring the data without the involvement of the processor will speed up the read-write task.
    - 2. DMA reduces the clock cycle required to read or write a block of data.
    - 3. Implementing DMA also reduces the overhead of the processor.
  - Disadvantages
    - 1. As it is a hardware unit, it would cost to implement a DMA controller in the system.
    - 2. Cache coherence problems can occur while using a DMA controller.
- Key Takeaways :
  - DMA is an abbreviation of direct memory access.
  - DMA is a method of data transfer between main memory and peripheral devices. The hardware unit that controls the DMA transfer is a DMA controller.
  - DMA controller transfers the data to and from memory without the participation of the processor.
  - The processor provides the start address and the word count of the data block which is transferred to or from memory to the DMA controller and frees the bus for DMA controller to transfer the block of data.DMA controller transfers the data block at the faster rate as data is directly accessed by I/O devices and is not required to pass through the processor which save the clock cycles.
  - The DMA controller transfers the block of data to and from memory in three modes: burst mode, cycle steal mode and transparent mode.
  - DMA can be configured in various ways; it can be a part of individual I/O devices, or all the peripherals attached to the system may share the same DMA controller.

Thus the DMA controller is a convenient mode of data transfer. It is preferred over the programmed I/O and Interrupt-driven I/O mode of data transfer.

How does DMA work ?

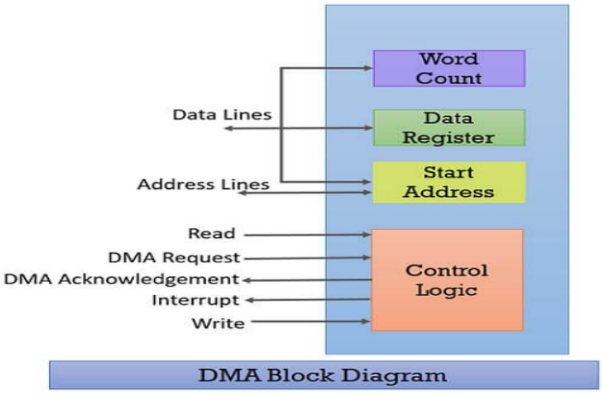
- The diagram of DMA controller that explains its working:



DMA Controller Data Transfer

Whenever an I/O device wants to transfer the data to or from memory, it sends the DMA request (DRQ) to the DMA controller. The DMA controller accepts this DRQ and asks the CPU to hold for a few clock cycles by sending it the Hold request (HLD). 2. CPU receives the Hold request (HLD) from DMA controller and relinquishes the bus and sends the Hold acknowledgement (HLDA) to DMA controller. 3. After receiving the Hold acknowledgement (HLDA), DMA controller acknowledges I/O device (DACK) that the data transfer can be performed and DMA controller takes the charge of the system bus and transfers the data to or from memory. 4. When the data transfer is accomplished, the DMA raises an interrupt to let the processor know that the task of data transfer is finished and the processor can take control over the bus again and start processing where it has left. Now the DMA controller can be a separate unit that is shared by various I/O devices, or it can also be a part of the I/O device interface.

Explain DMA block diagram ?

- 

DMA Block Diagram

Whenever a processor is requested to read or write a block of data, i.e. transfer a block of data, it instructs the DMA controller by sending the following information.

1. The first information is whether the data has to be read from memory or the data has to be written to the memory. It passes this information via read or write control lines that is between the processor and DMA controllers control logic unit.
2. The processor also provides the starting address of/ for the data block in the memory, from where the data block in memory has to be read or where the data block has to be written in memory. DMA controller stores this in its address register. It is also called the starting address register.
3. The processor also sends the word count, i.e. how many words are to be read or written. It stores this information in the data count or the word count register. 4. The most important is the address of the I/O device that wants to read or write data. This information is stored in the data register.

WHAT'S DMA CONTROLLER AND ITS USE?

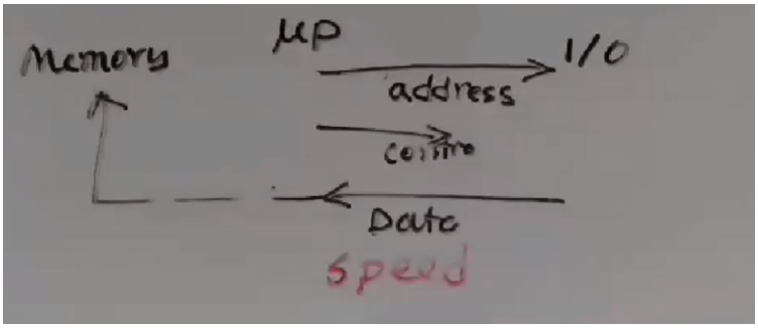
- A DMA controller is a specialized hardware device that manages data transfer between I/O devices and memory in a computer system. It contains several channels that can manage multiple DMA operations simultaneously, with each channel assigned to a specific I/O device. The DMA controller works by controlling the flow of data between the device and memory, freeing up the CPU to perform other tasks. It can handle interrupts and signal the CPU when a transfer is complete. DMA controllers are used in applications such as disk I/O, network communications, and multimedia processing to improve the performance of data transfer operations and achieve high data transfer rates.

WHATS USE OF DATA BUFFER IN DMAC?

- A data buffer is a temporary storage area used in a DMA controller (DMAC) to optimize data transfer between I/O devices and memory. The use of data buffers ensures efficient use of system resources, improves system performance, reduces the risk of data loss, and is compatible with different I/O devices. Data buffers allow the DMAC to transfer data in small blocks, which reduces the load on the system and ensures efficient data transfer. They also enable parallel data transfer operations, freeing up the CPU to perform other tasks. Overall, data buffers are essential for optimizing data transfer operations and improving system performance in modern computer systems.
-

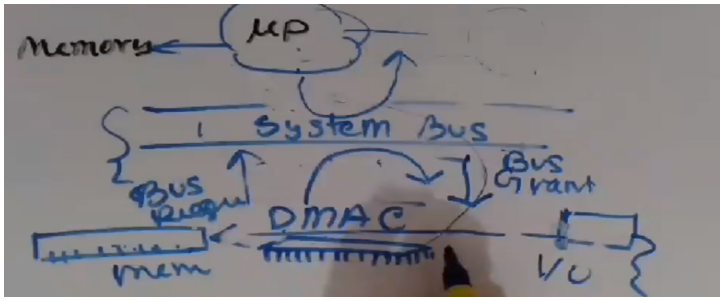
DIAGRAM :
WITHOUT DMA -

- Without DMA it’s a issue that I/O device and memory speed in case of communication like - read, write operations are slow down
- But with the help of DMA the I/O device can communicate with memory in fast mode without microprocessor’s involvement



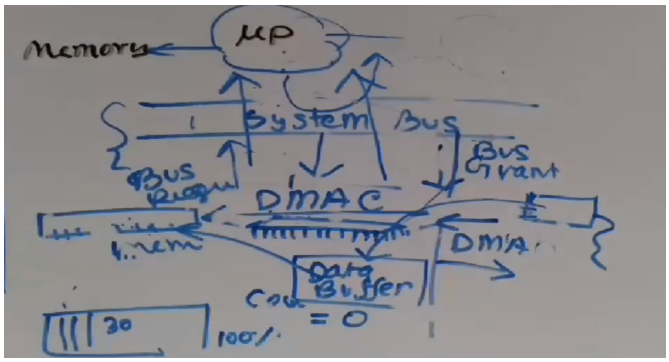
WITH DMAC (DMA CONTROLLER) :

- DMAC requests a microprocessor for the system bus(address, controller, data bus) so that it can make the communication of memory with the peripheral or I/O devices by transferring data via system bus.
- But it still remains slow in operation as the system bus may remain busy while serving the microprocessor, so DMAC has to wait for a long time for the available of the system bus.



WITH DMAC AND DATA BUFFER

- DATA BUFFER solves the above problem, while it stores the data to be read from the I/O device in its buffer and sets a count.
- When the system bus becomes available, the data buffer releases its data and the system bus passes this data to memory. It continues its job until the count of data stored in DATA BUFFER becomes zero.
- After that, the system bus again became available for the microprocessor.



Instruction Sets (NEXT PAGE)

Microprocessor - 8086 Instruction Sets

The 8086 microprocessor supports 8 types of instructions −

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Program Execution Transfer Instructions (Branch & Loop Instructions)
- Processor Control Instructions
- Iteration Control Instructions
- Interrupt Instructions

Let us now discuss these instruction sets in detail.

Data Transfer Instructions

These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group −

Instruction to transfer a word

- **MOV** − Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** − Used to put a word at the top of the stack.
- **POP** − Used to get a word from the top of the stack to the provided location.
- **PUSHA** − Used to put all the registers into the stack.
- **POPA** − Used to get words from the stack to all registers.
- **XCHG** − Used to exchange the data from two locations.
- **XLAT** − Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- **IN** − Used to read a byte or word from the provided port to the accumulator.
- **OUT** − Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- **LEA** − Used to load the address of operand into the provided register.
- **LDS** − Used to load DS register and other provided register from the memory
- **LES** − Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- **LAHF** − Used to load AH with the low byte of the flag register.
- **SAHF** − Used to store AH register to low byte of the flag register.
- **PUSHF** − Used to copy the flag register at the top of the stack.
- **POPF** − Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Following is the list of instructions under this group −

Instructions to perform addition

- **ADD** − Used to add the provided byte to byte/word to word.
- **ADC** − Used to add with carry.
- **INC** − Used to increment the provided byte/word by 1.
- **AAA** − Used to adjust ASCII after addition.
- **DAA** − Used to adjust the decimal after the addition/subtraction operation.

Instructions to perform subtraction

- **SUB** − Used to subtract the byte from byte/word from word.
- **SBB** − Used to perform subtraction with borrow.
- **DEC** − Used to decrement the provided byte/word by 1.
- **NPG** − Used to negate each bit of the provided byte/word and add 1/2’s complement.
- **CMP** − Used to compare 2 provided byte/word.
- **AAS** − Used to adjust ASCII codes after subtraction.
- **DAS** − Used to adjust decimal after subtraction.

Instruction to perform multiplication

- **MUL** − Used to multiply unsigned byte by byte/word by word.
- **IMUL** − Used to multiply signed byte by byte/word by word.
- **AAM** − Used to adjust ASCII codes after multiplication.

Instructions to perform division

- **DIV** − Used to divide the unsigned word by byte or unsigned double word by word.
- **IDIV** − Used to divide the signed word by byte or signed double word by word.
- **AAD** − Used to adjust ASCII codes after division.
- **CBW** − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- **CWD** − Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group −

Instructions to perform logical operation

- **NOT** − Used to invert each bit of a byte or word.
- **AND** − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **OR** − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- **TEST** − Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

- **SHL/SAL** − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

Instructions to perform rotate operations

- **ROL** − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- **ROR** − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- **RCR** − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.
- **RCL** − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

String Instructions

String is a group of bytes/words and their memory is always allocated in a sequential order.

Following is the list of instructions under this group −

- **REP** − Used to repeat the given instruction till CX ≠ 0.
- **REPE/REPZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **REPNE/REPNZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.
- **MOVS/MOVSB/MOVSW** − Used to move the byte/word from one string to another.
- **COMS/COMPSB/COMPSW** − Used to compare two string bytes/words.
- **INS/INSB/INSW** − Used as an input string/byte/word from the I/O port to the provided memory location.

- **OUTS/OUTSB/OUTSW** − Used as an output string/byte/word from the provided memory location to the I/O port.
- **SCAS/SCASB/SCASW** − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.
- **LODS/LODSB/LODSW** − Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −

Instructions to transfer the instruction during an execution without any condition −

- **CALL** − Used to call a procedure and save their return address to the stack.
- **RET** − Used to return from the procedure to the main program.
- **JMP** − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions −

- **JA/JNBE** − Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** − Used to jump if above/not below instruction satisfies.
- **JBE/JNA** − Used to jump if below/equal/ not above instruction satisfies.
- **JC** − Used to jump if carry flag CF = 1
- **JE/JZ** − Used to jump if equal/zero flag ZF = 1
- **JG/JNLE** − Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** − Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** − Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** − Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNC** − Used to jump if no carry flag (CF = 0)
- **JNE/JNZ** − Used to jump if not equal/zero flag ZF = 0
- **JNO** − Used to jump if no overflow flag OF = 0
- **JNP/JPO** − Used to jump if not parity/parity odd PF = 0
- **JNS** − Used to jump if not sign SF = 0
- **JO** − Used to jump if overflow flag OF = 1
- **JP/JPE** − Used to jump if parity/parity even PF = 1
- **JS** − Used to jump if sign flag SF = 1

Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

Following are the instructions under this group −

- **STC** − Used to set carry flag CF to 1
- **CLC** − Used to clear/reset carry flag CF to 0

- **CMC** − Used to put complement at the state of carry flag CF.
- **STD** − Used to set the direction flag DF to 1
- **CLD** − Used to clear/reset the direction flag DF to 0
- **STI** − Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- **CLI** − Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

- **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0
- **JCXZ** − Used to jump to the provided address if CX = 0

Interrupt Instructions

These instructions are used to call the interrupt during program execution.

- **INT** − Used to interrupt the program during execution and calling service specified.
- **INTO** − Used to interrupt the program during execution if OF = 1
- **IRET** − Used to return from interrupt service to the main program

8086 INSTRUCTION SET

DATA TRANSFER INSTRUCTIONS

MOV – MOV Destination, Source

The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location or an immediate number. The source and destination cannot both be memory locations. They must both be of the same type (bytes or words). MOV instruction does not affect any flag.

- MOV CX, 037AH — Put immediate number 037AH to CX
- MOV BL, [437AH] — Copy byte in DS at offset 437AH to BL
- MOV AX, BX — Copy content of register BX to AX
- MOV DL, [BX] — Copy byte from memory at [BX] to DL
- MOV DS, BX — Copy word from BX to DS register
- MOV RESULT [BP], AX — Copy AX to two memory locations;
  AL to the first location, AH to the second;
  EA of the first memory location is sum of the displacement represented by RESULTS and content of BP.
  Physical address = EA + SS.
- MOV ES: RESULTS [BP], AX — Same as the above instruction, but physical address = EA + ES, because of the segment override prefix ES

XCHG – XCHG Destination, Source

The XCHG instruction exchanges the content of a register with the content of another register or with the content of memory location(s). It cannot directly exchange the content of two memory locations. The source and destination must both be of the same type (bytes or words). The segment registers cannot be used in this instruction. This instruction does not affect any flag.

- XCHG AX, DX — Exchange word in AX with word in DX
- XCHG BL, CH — Exchange byte in BL with byte in CH
- XCHG AL, PRICES [BX] — Exchange byte in AL with byte in memory at EA = PRICE [BX] in DS.

LEA – LEA Register, Source

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

- LEA BX, PRICES — Load BX with offset of PRICE in DS
- LEA BP, SS: STACK_TOP — Load BP with offset of STACK_TOP in SS
- LEA CX, [BX][DI] — Load CX with EA = [BX] + [DI]

LDS – LDS Register, Memory address of the first word

This instruction loads new values into the specified register and into the DS register from four successive memory locations. The word from two memory locations is copied into the specified register and the word from the next two memory locations is copied into the DS registers. LDS does not affect any flag.

- LDS BX, [4326] — Copy content of memory at displacement 4326H in DS to BL, content of 4327H to BH. Copy content at displacement of 4328H and 4329H in DS to DS register.
- LDS SI, SPTR — Copy content of memory at displacement SPTR and SPTR + 1

in DS to SI register. Copy content of memory at displacements SPTR + 2 and SPTR + 3 in DS to DS register. DS: SI now points at start of the desired string.

LES – LES Register, Memory address of the first word

This instruction loads new values into the specified register and into the ES register from four successive memory locations. The word from the first two memory locations is copied into the specified register, and the word from the next two memory locations is copied into the ES register. LES does not affect any flag.

- LES BX, [789AH] — Copy content of memory at displacement 789AH in DS to BL, content of 789BH to BH, content of memory at displacement 789CH and 789DH in DS is copied to ES register.
- LES DI, [BX] — Copy content of memory at offset [BX] and offset [BX] + 1 in DS to DI register. Copy content of memory at offset [BX] + 2 and [BX] + 3 to ES register.

ARITHMETIC INSTRUCTIONS

ADD – ADD Destination, Source
ADC – ADC Destination, Source

These instructions add a number from some *source* to a number in some *destination* and put the result in the specified destination. The ADC also adds the status of the carry flag to the result. The source may be an immediate number, a register, or a memory location. The destination may be a register or a memory location. The source and the destination in an instruction cannot both be memory locations. The source and the destination must be of the same type (bytes or words). If you want to add a byte to a word, you must copy the byte to a word location and fill the upper byte of the word with 0's before adding. Flags affected: AF, CF, OF, SF, ZF.

- ADD AL, 74H — Add immediate number 74H to content of AL. Result in AL
- ADC CL, BL — Add content of BL plus carry status to content of CL
- ADD DX, BX — Add content of BX to content of DX
- ADD DX, [SI] — Add word from memory at offset [SI] in DS to content of DX
- ADC AL, PRICES [BX] — Add byte from effective address PRICES [BX] plus carry status to content of AL
- ADD AL, PRICES [BX] — Add content of memory at effective address PRICES [BX] to AL

SUB – SUB Destination, Source
SBB – SBB Destination, Source

These instructions subtract the number in some *source* from the number in some *destination* and put the result in the destination. The SBB instruction also subtracts the content of carry flag from the destination. The source may be an immediate number, a register or memory location. The destination can also be a register or a memory location. However, the source and the destination cannot both be memory location. The source and the destination must both be of the same type (bytes or words). If you want to subtract a byte from a word, you must first move the byte to a word location such as a 16-bit register and fill the upper byte of the word with 0's. Flags affected: AF, CF, OF, PF, SF, ZF.

- SUB CX, BX — CX – BX; Result in CX
- SBB CH, AL — Subtract content of AL and content of CF from content of CH. Result in CH
- SUB AX, 3427H — Subtract immediate number 3427H from AX
- SBB BX, [3427H] — Subtract word at displacement 3427H in DS and content of CF

	from BX
➢ SUB PRICES [BX], 04H	Subtract 04 from byte at effective address PRICES [BX], if PRICES is declared with DB; Subtract 04 from word at effective address PRICES [BX], if it is declared with DW.
➢ SBB CX, TABLE [BX]	Subtract word from effective address TABLE [BX] and status of CF from CX.
➢ SBB TABLE [BX], CX	Subtract CX and status of CF from word in memory at effective address TABLE[BX].

MUL – MUL Source

This instruction multiplies an *unsigned* byte in some *source* with an *unsigned* byte in AL register or an unsigned word in some *source* with an unsigned word in AX register. The source can be a register or a memory location. When a byte is multiplied by the content of AL, the result (product) is put in AX. When a word is multiplied by the content of AX, the result is put in DX and AX registers. If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0's. AF, PF, SF and ZF are undefined after a MUL instruction.

If you want to multiply a byte with a word, you must first move the byte to a word location such as an extended register and fill the upper byte of the word with all 0's. You cannot use the CBW instruction for this, because the CBW instruction fills the upper byte with copies of the most significant bit of the lower byte.

➢ MUL BH	Multiply AL with BH; result in AX
➢ MUL CX	Multiply AX with CX; result high word in DX, low word in AX
➢ MUL BYTE PTR [BX]	Multiply AL with byte in DS pointed to by [BX]
➢ MUL FACTOR [BX]	Multiply AL with byte at effective address FACTOR [BX], if it is declared as type byte with DB. Multiply AX with word at effective address FACTOR [BX], if it is declared as type word with DW.
➢ MOV AX, MCAND_16	Load 16-bit multiplicand into AX
MOV CL, MPLIER_8	Load 8-bit multiplier into CL
MOV CH, 00H	Set upper byte of CX to all 0's
MUL CX	AX times CX; 32-bit result in DX and AX

IMUL – IMUL Source

This instruction multiplies a *signed* byte from *source* with a *signed* byte in AL or a *signed* word from some *source* with a *signed* word in AX. The source can be a register or a memory location. When a byte from source is multiplied with content of AL, the signed result (product) will be put in AX. When a word from source is multiplied by AX, the result is put in DX and AX. If the magnitude of the product does not require all the bits of the destination, the unused byte / word will be filled with copies of the sign bit. If the upper byte of a 16-bit result or the upper word of a 32-bit result contains only copies of the sign bit (all 0's or all 1's), then CF and the OF will both be 0; If it contains a part of the product, CF and OF will both be 1. AF, PF, SF and ZF are undefined after IMUL.

If you want to multiply a signed byte with a signed word, you must first move the byte into a word location and fill the upper byte of the word with copies of the sign bit. If you move the byte into AL, you can use the CBW instruction to do this.

➢ IMUL BH	Multiply signed byte in AL with signed byte in BH; result in AX.
➢ IMUL AX	Multiply AX times AX; result in DX and AX
➢ MOV CX, MULTIPLIER	Load signed word in CX
MOV AL, MULTIPLICAND	Load signed byte in AL
CBW	Extend sign of AL into AH
IMUL CX	Multiply CX with AX; Result in DX and AX

DIV – DIV Source

This instruction is used to divide an *unsigned* word by a byte or to divide an *unsigned* double word (32 bits) by a word. When a word is divided by a byte, the word must be in the AX register. The divisor can be in a register or a memory location. After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder. When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder. If an attempt is made to divide by 0 or if the quotient is too large to fit in the destination (greater than FFH / FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after a DIV instruction.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all 0's.

➢ DIV BL	Divide word in AX by byte in BL; Quotient in AL, remainder in AH
➢ DIV CX	Divide down word in DX and AX by word in CX; Quotient in AX, and remainder in DX.
➢ DIV SCALE [BX]	AX / (byte at effective address SCALE [BX]) if SCALE [BX] is of type byte; or (DX and AX) / (word at effective address SCALE[BX] if SCALE[BX] is of type word

IDIV – IDIV Source

This instruction is used to divide a *signed* word by a *signed* byte, or to divide a *signed* double word by a *signed* word.

When dividing a signed word by a signed byte, the word must be in the AX register. The divisor can be in an 8-bit register or a memory location. After the division, AL will contain the signed quotient, and AH will contain the signed remainder. The sign of the remainder will be the same as the sign of the dividend. If an attempt is made to divide by 0, the quotient is greater than 127 (7FH) or less than –127 (81H), the 8086 will automatically generate a type 0 interrupt.

When dividing a signed double word by a signed word, the most significant word of the dividend (numerator) must be in the DX register, and the least significant word of the dividend must be in the AX register. The divisor can be in any other 16-bit register or memory location. After the division, AX will contain a signed 16-bit quotient, and DX will contain a signed 16-bit remainder. The sign of the remainder will be the same as the sign of the dividend. Again, if an attempt is made to divide by 0, the quotient is greater than +32,767 (7FFFH) or less than –32,767 (8001H), the 8086 will automatically generate a type 0 interrupt.

All flags are undefined after an IDIV.

If you want to divide a signed byte by a signed byte, you must first put the dividend byte in AL and sign-extend AL into AH. The CBW instruction can be used for this purpose. Likewise, if you want to divide a signed word by a signed word, you must put the dividend word in AX and extend the sign of AX to all the bits of DX. The CWD instruction can be used for this purpose.

➢ IDIV BL	Signed word in AX/signed byte in BL
➢ IDIV BP	Signed double word in DX and AX/signed word in BP
➢ IDIV BYTE PTR [BX]	AX / byte at offset [BX] in DS

INC – INC Destination

The INC instruction adds 1 to a specified register or to a memory location. AF, OF, PF, SF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result will be all 0's with no carry.

➢ INC BL	Add 1 to contains of BL register
➢ INC CX	Add 1 to contains of CX register
➢ INC BYTE PTR [BX]	Increment byte in data segment at offset contained in BX.
➢ INC WORD PTR [BX]	Increment the word at offset of [BX] and [BX + 1] in the data segment.
➢ INC TEMP	Increment byte or word named TEMP in the data segment. Increment byte if MAX_TEMP declared with DB. Increment word if MAX_TEMP is declared with DW.
➢ INC PRICES [BX]	Increment element pointed to by [BX] in array PRICES. Increment a word if PRICES is declared as an array of words; Increment a byte if PRICES is declared as an array of bytes.

DEC – DEC Destination

This instruction subtracts 1 from the destination word or byte. The destination can be a register or a memory location. AF, OF, SF, PF, and ZF are updated, but CF is not affected. This means that if an 8-bit destination containing 00H or a 16-bit destination containing 0000H is decremented, the result will be FFH or FFFFH with no carry (borrow).

➢ DEC CL	Subtract 1 from content of CL register
➢ DEC BP	Subtract 1 from content of BP register
➢ DEC BYTE PTR [BX]	Subtract 1 from byte at offset [BX] in DS.
➢ DEC WORD PTR [BP]	Subtract 1 from a word at offset [BP] in SS.
➢ DEC COUNT	Subtract 1 from byte or word named COUNT in DS. Decrement a byte if COUNT is declared with a DB; Decrement a word if COUNT is declared with a DW.

DAA (DECIMAL ADJUST AFTER BCD ADDITION)

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA to work correctly. If the lower nibble in AL after an addition is greater than 9 or AF was set by the addition, then the DAA instruction will add 6 to the lower nibble in AL. If the result in the upper nibble of AL in now greater than 9 or if the carry flag was set by the addition or correction, then the DAA instruction will add 60H to AL.

➢ Let AL = 59 BCD, and BL = 35 BCD	
ADD AL, BL	AL = 8EH; lower nibble > 9, add 06H to AL
DAA	AL = 94 BCD, CF = 0
➢ Let AL = 88 BCD, and BL = 49 BCD	
ADD AL, BL	AL = D1H; AF = 1, add 06H to AL
DAA	AL = D7H; upper nibble > 9, add 60H to AL AL = 37 BCD, CF = 1

The DAA instruction updates AF, CF, SF, PF, and ZF; but OF is undefined.

DAS (DECIMAL ADJUST AFTER BCD SUBTRACTION)

This instruction is used after subtracting one packed BCD number from another packed BCD number, to make sure the result is correct packed BCD. The result of the subtraction must be in AL for DAS to work correctly. If the lower nibble in AL after a subtraction is greater than 9 or the AF was set by the subtraction, then the DAS instruction will subtract 6 from the lower nibble AL. If the result in the upper nibble is now greater than 9 or if the carry flag was set, the DAS instruction will subtract 60 from AL.

➢ Let AL = 86 BCD, and BH = 57 BCD	
SUB AL, BH	AL = 2FH; lower nibble > 9, subtract 06H from AL AL = 29 BCD, CF = 0

- Let AL = 49 BCD, and BH = 72 BCD

SUB AL, BH	AL = D7H; upper nibble > 9, subtract 60H from AL
DAS	AL = 77 BCD, CF = 1 (borrow is needed)

The DAS instruction updates AF, CF, SF, PF, and ZF; but OF is undefined.

CBW (CONVERT SIGNED BYTE TO SIGNED WORD)

This instruction copies the sign bit of the byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. CBW does not affect any flag.

- Let AX = 00000000 10011011 (–155 decimal)

CBW	Convert signed byte in AL to signed word in AX
	AX = 11111111 10011011 (–155 decimal)

CWD (CONVERT SIGNED WORD TO SIGNED DOUBLE WORD)

This instruction copies the sign bit of a word in AX to all the bits of the DX register. In other words, it extends the sign of AX into all of DX. CWD affects no flags.

- Let DX = 00000000 00000000, and AX = 11110000 11000111 (–3897 decimal)

CWD	Convert signed word in AX to signed double word in DX:AX
	DX = 11111111 11111111
	AX = 11110000 11000111 (–3897 decimal)

AAA (ASCII ADJUST FOR ADDITION)

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD.

- Let AL = 0011 0101 (ASCII 5), and BL = 0011 1001 (ASCII 9)

ADD AL, BL	AL = 0110 1110 (6EH, which is incorrect BCD)
AAA	AL = 0000 0100 (unpacked BCD 4)
	CF = 1 indicates answer is 14 decimal.

The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.

AAS (ASCII ADJUST FOR SUBTRACTION)

Numerical data coming into a computer from a terminal is usually in an ASCII code. In this code the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to subtract the ASCII codes for two decimal digits without masking the "3" in the upper nibble of each. The AAS instruction is then used to make sure the result is the correct unpacked BCD.

- Let AL = 00111001 (39H or ASCII 9), and BL = 00110101 (35H or ASCII 5)

SUB AL, BL	AL = 00000100 (BCD 04), and CF = 0
AAS	AL = 00000100 (BCD 04), and CF = 0 (no borrow required)

- Let AL = 00110101 (35H or ASCII 5), and BL = 00111001 (39H or ASCII 9)

SUB AL, BL	AL = 11111100 (– 4 in 2's complement form), and CF = 1
AAS	AL = 00000100 (BCD 06), and CF = 1 (borrow required)

The AAS instruction works only on the AL register. It updates ZF and CF; but OF, PF, SF, AF are left undefined.

AAM (BCD ADJUST AFTER MULTIPLY)

Before you can multiply two ASCII digits, you must first mask the upper 4 bit of each. This leaves unpacked BCD (one BCD digit per byte) in each byte. After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product to two unpacked BCD digits in AX. AAM works only after the multiplication of two unpacked BCD bytes, and it works only the operand in AL. AAM updates PF, SF and ZF but AF; CF and OF are left undefined.

- Let AL = 00000101 (unpacked BCD 5), and BH = 00001001 (unpacked BCD 9)
  MUL BH AL x BH: AX = 00000000 00101101 = 002DH
  AAM AX = 00000100 00000101 = 0405H (unpacked BCD for 45)

AAD (BCD-TO-BINARY CONVERT BEFORE DIVISION)

AAD converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. After the BCD division, AL will contain the unpacked BCD quotient and AH will contain the unpacked BCD remainder. AAD updates PF, SF and ZF; AF, CF and OF are left undefined.

- Let AX = 0607 (unpacked BCD for 67 decimal), and CH = 09H
  AAD AX = 0043 (43H = 67 decimal)
  DIV CH AL = 07; AH = 04; Flags undefined after DIV

If an attempt is made to divide by 0, the 8086 will generate a type 0 interrupt.

LOGICAL INSTRUCTIONS

AND – AND Destination, Source

This instruction ANDs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0 after AND. PF, SF, and ZF are updated by the AND instruction. AF is undefined. PF has meaning only for an 8-bit operand.

- AND CX, [SI] AND word in DS at offset [SI] with word in CX register; Result in CX register
- AND BH, CL AND byte in CL with byte in BH; Result in BH
- AND BX, 00FFH 00FFH Masks upper byte, leaves lower byte unchanged.

OR – OR Destination, Source

This instruction ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after OR. PF, SF, and ZF are updated by the OR instruction. AF is undefined. PF has meaning only for an 8-bit operand.

- OR AH, CL CL ORed with AH, result in AH, CL not changed

- OR BP, SI — SI ORed with BP, result in BP, SI not changed
- OR SI, BP — BP ORed with SI, result in SI, BP not changed
- OR BL, 80H — BL ORed with immediate number 80H; sets MSB of BL to 1
- OR CX, TABLE [SI] — CX ORed with word from effective address TABLE [SI]; Content of memory is not changed.

XOR – XOR Destination, Source

This instruction Exclusive-ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after XOR. PF, SF, and ZF are updated. PF has meaning only for an 8-bit operand. AF is undefined.

- XOR CL, BH — Byte in BH exclusive-ORed with byte in CL. Result in CL. BH not changed.
- XOR BP, DI — Word in DI exclusive-ORed with word in BP. Result in BP. DI not changed.
- XOR WORD PTR [BX], 00FFH — Exclusive-OR immediate number 00FFH with word at offset [BX] in the data segment. Result in memory location [BX]

NOT – NOT Destination

The NOT instruction inverts each bit (forms the 1's complement) of a byte or word in the specified destination. The destination can be a register or a memory location. This instruction does not affect any flag.

- NOT BX — Complement content or BX register
- NOT BYTE PTR [BX] — Complement memory byte at offset [BX] in data segment.

NEG – NEG Destination

This instruction replaces the number in a destination with its 2's complement. The destination can be a register or a memory location. It gives the same result as the *invert each bit and add one* algorithm. The NEG instruction updates AF, AF, PF, ZF, and OF.

- NEG AL — Replace number in AL with its 2's complement
- NEG BX — Replace number in BX with its 2's complement
- NEG BYTE PTR [BX] — Replace byte at offset BX in DX with its 2's complement
- NEG WORD PTR [BP] — Replace word at offset BP in SS with its 2's complement

CMP – CMP Destination, Source

This instruction compares a byte / word in the specified source with a byte / word in the specified destination. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and the destination cannot both be memory locations. The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction. For the instruction CMP CX, BX, the values of CF, ZF, and SF will be as follows:

	CF	ZF	SF	
CX = BX	0	1	0	Result of subtraction is 0
CX > BX	0	0	0	No borrow required, so CF = 0
CX < BX	1	0	1	Subtraction requires borrow, so CF = 1

- CMP AL, 01H — Compare immediate number 01H with byte in AL
- CMP BH, CL — Compare byte in CL with byte in BH
- CMP CX, TEMP — Compare word in DS at displacement TEMP with word at CX
- CMP PRICES [BX], 49H — Compare immediate number 49H with byte at offset [BX] in array PRICES

TEST – TEST Destination, Source

This instruction ANDs the byte / word in the specified source with the byte / word in the specified destination. Flags are updated, but neither operand is changed. The test instruction is often used to set flags before a Conditional jump instruction.

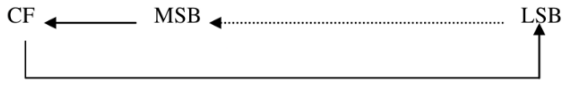
The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0's after TEST. PF, SF and ZF will be updated to show the results of the destination. AF is be undefined.

- TEST AL, BH — AND BH with AL. No result stored; Update PF, SF, ZF.
- TEST CX, 0001H — AND CX with immediate number 0001H; No result stored; Update PF, SF, ZF
- TEST BP, [BX][DI] — AND word are offset [BX][DI] in DS with word in BP. No result stored. Update PF, SF, and ZF

ROTATE AND SHIFT INSTRUCTIONS

RCL – RCL Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the left. The operation circular because the MSB of the operand is rotated into the carry flag and the bit in the carry flag is rotated around into LSB of the operand.



For multi-bit rotates, CF will contain the bit most recently rotated out of the MSB.

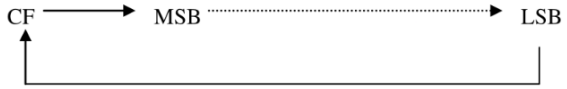
The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

RCL affects only CF and OF. OF will be a 1 after a single bit RCL if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

- RCL DX, 1 — Word in DX 1 bit left, MSB to CF, CF to LSB
- MOV CL, 4 — Load the number of bit positions to rotate into CL
  RCL SUM [BX], CL — Rotate byte or word at effective address SUM [BX] 4 bits left Original bit 4 now in CF, original CF now in bit 3.

RCR – RCR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotate around into MSB of the operand.



For multi-bit rotate, CF will contain the bit most recently rotated out of the LSB.

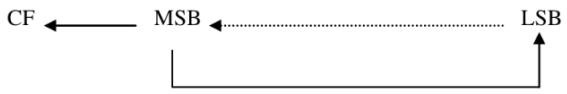
The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

RCR affects only CF and OF. OF will be a 1 after a single bit RCR if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

➢ RCR BX, 1	Word in BX right 1 bit, CF to MSB, LSB to CF
➢ MOV CL, 4	Load CL for rotating 4 bit position
RCR BYTE PTR [BX], 4	Rotate the byte at offset [BX] in DS 4 bit positions right CF = original bit 3, Bit 4 – original CF.

ROL – ROL Destination, Count

This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF. In the case of multiple-bit rotate, CF will contain a copy of the bit most recently moved out of the MSB.



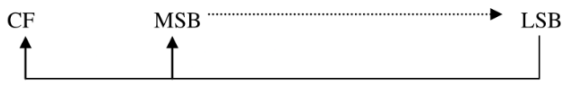
The destination can be a register or a memory location. If you to want rotate the operand by one bit position, you can specify this by putting 1 in the count position in the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

ROL affects only CF and OF. OF will be a 1 after a single bit ROL if the MSB was changed by the rotate.

➢ ROL AX, 1	Rotate the word in AX 1 bit position left, MSB to LSB and CF
➢ MOV CL, 04H	Load number of bits to rotate in CL
ROL BL, CL	Rotate BL 4 bit positions
➢ ROL FACTOR [BX], 1	Rotate the word or byte in DS at EA = FACTOR [BX] by 1 bit position left into CF

ROR – ROR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to right. The operation is desired as a rotate rather than shift, because the bit moved out of the LSB is rotated around into the MSB. The data bit moved out of the LSB is also copied into CF. In the case of multiple bit rotates, CF will contain a copy of the bit most recently moved out of the LSB.



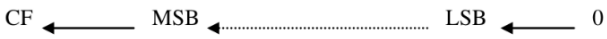
The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting 1 in the count position in the instruction. To rotate by more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

ROR affects only CF and OF. OF will be a 1 after a single bit ROR if the MSB was changed by the rotate.

➢ ROR BL, 1	Rotate all bits in BL right 1 bit position LSB to MSB and to CF
➢ MOV CL, 08H	Load CL with number of bit positions to be rotated
ROR WORD PTR [BX], CL	Rotate word in DS at offset [BX] 8 bit position right

SAL – SAL Destination, Count
SHL – SHL Destination, Count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB operation, a 0 is put in the LSB position. The MSB will be shifted into CF. In the case of multi-bit shift, CF will contain the bit most recently shifted out from the MSB. Bits shifted into CF previously will be lost.



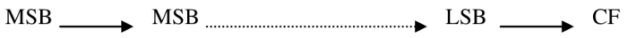
The destination operand can be a byte or a word. It can be in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected as follow: CF contains the bit most recently shifted out from MSB. For a count of one, OF will be 1 if CF and the current MSB are not the same. For multiple-bit shifts, OF is undefined. SF and ZF will be updated to reflect the condition of the destination. PF will have meaning only for an operand in AL. AF is undefined.

➢ SAL BX, 1	Shift word in BX 1 bit position left, 0 in LSB
➢ MOV CL, 02h	Load desired number of shifts in CL
SAL BP, CL	Shift word in BP left CL bit positions, 0 in LSBs
➢ SAL BYTE PTR [BX], 1	Shift byte in DX at offset [BX] 1 bit position left, 0 in LSB

SAR – SAR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. In other words, the sign bit is copied into the MSB. The LSB will be shifted into CF. In the case of multiple-bit shift, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.



The destination operand can be a byte or a word. It can be in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected as follow: CF contains the bit most recently shifted in from LSB. For a count of one, OF will be 1 if the two MSBs are not the same. After a multi-bit SAR, OF will be 0. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8- bit destination. AF will be undefined after SAR.

➢ SAR DX, 1	Shift word in DI one bit position right, new MSB = old MSB
➢ MOV CL, 02H	Load desired number of shifts in CL

SAR WORD PTR [BP], CL	Shift word at offset [BP] in stack segment right by two bit positions, the two MSBs are now copies of original LSB

SHR – SHR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is put in its place. The bit shifted out of the LSB position goes to CF. In the case of multi-bit shifts, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.

The destination operand can be a byte or a word in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put "CL" in the count position of the instruction.

The flags are affected by SHR as follow: CF contains the bit most recently shifted out from LSB. For a count of one, OF will be 1 if the two MSBs are not both 0's. For multiple-bit shifts, OF will be meaningless. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8-bit destination. AF is undefined.

➢ SHR BP, 1	Shift word in BP one bit position right, 0 in MSB
➢ MOV CL, 03H	Load desired number of shifts into CL
SHR BYTE PTR [BX]	Shift byte in DS at offset [BX] 3 bits right; 0's in 3 MSBs

TRANSFER-OF-CONTROL INSTRUCTIONS

Note: The following rules apply to the discussions presented in this section.

- The terms *above* and *below* are used when referring to the magnitude of unsigned numbers. For example, the number 00000111 (7) is above the number 00000010 (2), whereas the number 00000100 (4) is below the number 00001110 (14).
- The terms *greater* and *less* are used to refer to the relationship of two signed numbers. Greater means more positive. The number 00000111 (+7) is greater than the number 11111110 (-2), whereas the number 11111100 (-4) is less than the number 11110100 (-6).
- In the case of Conditional jump instructions, the destination address must be in the range of –128 bytes to +127 bytes from the address of the next instruction
- These instructions do not affect any flags.

JMP (UNCONDITIONAL JUMP TO SPECIFIED DESTINATION)

This instruction will fetch the next instruction from the location specified in the instruction rather than from the next location after the JMP instruction. If the destination is in the same code segment as the JMP instruction, then only the instruction pointer will be changed to get the destination location. This is referred to as a *near jump.* If the destination for the jump instruction is in a segment with a name different from that of the segment containing the JMP instruction, then both the instruction pointer and the code segment register content will be changed to get the destination location. This referred to as a *far jump*. The JMP instruction does not affect any flag.

➢ JMP CONTINUE

This instruction fetches the next instruction from address at label CONTINUE. If the label is in the same segment, an offset coded as part of the instruction will be added to the instruction pointer to produce the new fetch address. If the label is another segment, then IP and CS will be replaced with value coded in

part of the instruction. This type of jump is referred to as *direct* because the displacement of the destination or the destination itself is specified directly in the instruction.

- JMP BX

This instruction replaces the content of IP with the content of BX. BX must first be loaded with the offset of the destination instruction in CS. This is a near jump. It is also referred to as an *indirect* jump because the new value of IP comes from a register rather than from the instruction itself, as in a direct jump.

- JMP WORD PTR [BX]

This instruction replaces IP with word from a memory location pointed to by BX in DX. This is an indirect near jump.

- JMP DWORD PTR [SI]

This instruction replaces IP with word pointed to by SI in DS. It replaces CS with a word pointed by SI + 2 in DS. This is an indirect far jump.

JA / JNBE (JUMP IF ABOVE / JUMP IF NOT BELOW OR EQUAL)

If, after a compare or some other instructions which affect flags, the zero flag and the carry flag both are 0, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are not both 0, the instruction will have no effect on program execution.

➢ CMP AX, 4371H	Compare by subtracting 4371H from AX
JA NEXT	Jump to label NEXT if AX above 4371H
➢ CMP AX, 4371H	Compare (AX – 4371H)
JNBE NEXT	Jump to label NEXT if AX not below or equal to 4371H

JAE / JNB / JNC
(JUMP IF ABOVE OR EQUAL / JUMP IF NOT BELOW / JUMP IF NO CARRY)

If, after a compare or some other instructions which affect flags, the carry flag is 0, this instruction will cause execution to jump to a label given in the instruction. If CF is 1, the instruction will have no effect on program execution.

➢ CMP AX, 4371H	Compare (AX – 4371H)
JAE NEXT	Jump to label NEXT if AX above 4371H
➢ CMP AX, 4371H	Compare (AX – 4371H)
JNB NEXT	Jump to label NEXT if AX not below 4371H
➢ ADD AL, BL	Add two bytes
JNC NEXT	If the result with in acceptable range, continue

JB / JC / JNAE (JUMP IF BELOW / JUMP IF CARRY / JUMP IF NOT ABOVE OR EQUAL)

If, after a compare or some other instructions which affect flags, the carry flag is a 1, this instruction will cause execution to jump to a label given in the instruction. If CF is 0, the instruction will have no effect on program execution.

➢ CMP AX, 4371H	Compare (AX – 4371H)
JB NEXT	Jump to label NEXT if AX below 4371H
➢ ADD BX, CX	Add two words
JC NEXT	Jump to label NEXT if CF = 1
➢ CMP AX, 4371H	Compare (AX – 4371H)
JNAE NEXT	Jump to label NEXT if AX not above or equal to 4371H

JBE / JNA (JUMP IF BELOW OR EQUAL / JUMP IF NOT ABOVE)

If, after a compare or some other instructions which affect flags, either the zero flag or the carry flag is 1, this instruction will cause execution to jump to a label given in the instruction. If CF and ZF are both 0, the instruction will have no effect on program execution.

➢ CMP AX, 4371H	Compare (AX – 4371H)
JBE NEXT	Jump to label NEXT if AX is below or equal to 4371H
➢ CMP AX, 4371H	Compare (AX – 4371H)
JNA NEXT	Jump to label NEXT if AX not above 4371H

JG / JNLE (JUMP IF GREATER / JUMP IF NOT LESS THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the zero flag is 0 and the carry flag is the same as the overflow flag.

➢ CMP BL, 39H	Compare by subtracting 39H from BL
JG NEXT	Jump to label NEXT if BL more positive than 39H
➢ CMP BL, 39H	Compare by subtracting 39H from BL
JNLE NEXT	Jump to label NEXT if BL is not less than or equal to 39H

JGE / JNL (JUMP IF GREATER THAN OR EQUAL / JUMP IF NOT LESS THAN)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction, if the sign flag is equal to the overflow flag.

➢ CMP BL, 39H	Compare by subtracting 39H from BL
JGE NEXT	Jump to label NEXT if BL more positive than or equal to 39H
➢ CMP BL, 39H	Compare by subtracting 39H from BL
JNL NEXT	Jump to label NEXT if BL not less than 39H

JL / JNGE (JUMP IF LESS THAN / JUMP IF NOT GREATER THAN OR EQUAL)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the sign flag is not equal to the overflow flag.

➢ CMP BL, 39H	Compare by subtracting 39H from BL
JL AGAIN	Jump to label AGAIN if BL more negative than 39H
➢ CMP BL, 39H	Compare by subtracting 39H from BL
JNGE AGAIN	Jump to label AGAIN if BL not more positive than or equal to 39H

JLE / JNG (JUMP IF LESS THAN OR EQUAL / JUMP IF NOT GREATER)

This instruction is usually used after a Compare instruction. The instruction will cause a jump to the label given in the instruction if the zero flag is set, or if the sign flag not equal to the overflow flag.

➢ CMP BL, 39H	Compare by subtracting 39H from BL
JLE NEXT	Jump to label NEXT if BL more negative than or equal to 39H
➢ CMP BL, 39H	Compare by subtracting 39H from BL
JNG NEXT	Jump to label NEXT if BL not more positive than 39H

JE / JZ (JUMP IF EQUAL / JUMP IF ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is set, then this instruction will cause a jump to the label given in the instruction.

➢ CMP BX, DX	Compare (BX-DX)
JE DONE	Jump to DONE if BX = DX
➢ IN AL, 30H	Read data from port 8FH
SUB AL, 30H	Subtract the minimum value.
JZ START	Jump to label START if the result of subtraction is 0

JNE / JNZ (JUMP NOT EQUAL / JUMP IF NOT ZERO)

This instruction is usually used after a Compare instruction. If the zero flag is 0, then this instruction will cause a jump to the label given in the instruction.

➢ IN AL, 0F8H	Read data value from port
CMP AL, 72	Compare (AL –72)
JNE NEXT	Jump to label NEXT if AL ≠ 72
➢ ADD AX, 0002H	Add count factor 0002H to AX
DEC BX	Decrement BX
JNZ NEXT	Jump to label NEXT if BX ≠ 0

JS (JUMP IF SIGNED / JUMP IF NEGATIVE)

This instruction will cause a jump to the specified destination address if the sign flag is set. Since a 1 in the sign flag indicates a negative signed number, you can think of this instruction as saying “jump if negative”.

➢ ADD BL, DH	Add signed byte in DH to signed byte in DL
JS NEXT	Jump to label NEXT if result of addition is negative number

JNS (JUMP IF NOT SIGNED / JUMP IF POSITIVE)

This instruction will cause a jump to the specified destination address if the sign flag is 0. Since a 0 in the sign flag indicate a positive signed number, you can think to this instruction as saying “jump if positive”.

➢ DEC AL	Decrement AL
JNS NEXT	Jump to label NEXT if AL has not decremented to FFH

JP / JPE (JUMP IF PARITY / JUMP IF PARITY EVEN)

If the number of 1’s left in the lower 8 bits of a data word after an instruction which affects the parity flag is even, then the parity flag will be set. If the parity flag is set, the JP / JPE instruction will cause a jump to the specified destination address.

➢ IN AL, 0F8H	Read ASCII character from Port F8H
OR AL, AL	Set flags
JPE ERROR	Odd parity expected, send error message if parity found even

JNP / JPO (JUMP IF NO PARITY / JUMP IF PARITY ODD)

If the number of 1's left in the lower 8 bits of a data word after an instruction which affects the parity flag is odd, then the parity flag is 0. The JNP / JPO instruction will cause a jump to the specified destination address, if the parity flag is 0.

➢ IN AL, 0F8H	Read ASCII character from Port F8H
OR AL, AL	Set flags
JPO ERROR	Even parity expected, send error message if parity found odd

JO (JUMP IF OVERFLOW)

The overflow flag will be set if the magnitude of the result produced by some signed arithmetic operation is too large to fit in the destination register or memory location. The JO instruction will cause a jump to the destination given in the instruction, if the overflow flag is set.

➢ ADD AL, BL	Add signed bytes in AL and BL
JO ERROR	Jump to label ERROR if overflow from add

JNO (JUMP IF NO OVERFLOW)

The overflow flag will be set if some signed arithmetic operation is too large to fit in the destination register or memory location. The JNO instruction will cause a jump to the destination given in the instruction, if the overflow flag is not set.

➢ ADD AL, BL	Add signed byte in AL and BL
JNO DONE	Process DONE if no overflow

JCXZ (JUMP IF THE CX REGISTER IS ZERO)

This instruction will cause a jump to the label to a given in the instruction, if the CX register contains all 0's. The instruction does not look at the zero flag when it decides whether to jump or not.

➢ JCXZ SKIP	If CX = 0, skip the process
SUB [BX], 07H	Subtract 7 from data value
SKIP: ADD C	Next instruction

LOOP (JUMP TO SPECIFIED LABEL IF CX ≠ 0 AFTER AUTO DECREMENT)

This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOP instruction. This instruction does not affect any flag.

➢ MOV BX, OFFSET PRICES	Point BX at first element in array
MOV CX, 40	Load CX with number of elements in array
NEXT: MOV AL, [BX]	Get element from array
INC AL	Increment the content of AL
MOV [BX], AL	Put result back in array
INC BX	Increment BX to point to next location
LOOP NEXT	Repeat until all elements adjusted

LOOPE / LOOPZ (LOOP WHILE CX ≠ 0 AND ZF = 1)

This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes 0. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX ≠ 0 and ZF = 1, execution will jump to a destination specified by a label in the instruction. If CX = 0, execution simply go on the next instruction after LOOPE / LOOPZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 0. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOPE / LOOPZ instruction. This instruction does not affect any flag.

```
➢ MOV BX, OFFSET ARRAY    Point BX to address of ARRAY before start of array
  DEC BX                  Decrement BX
  MOV CX, 100             Put number of array elements in CX
  NEXT: INC BX            Point to next element in array
  CMP [BX], OFFH          Compare array element with FFH
  LOOPE NEXT
```

LOOPNE / LOOPNZ (LOOP WHILE CX ≠ 0 AND ZF = 0)

This instruction is used to repeat a group of instructions some number of times, or until the zero flag becomes a 1. The number of times the instruction sequence is to be repeated is loaded into the count register CX. Each time the LOOPNE / LOOPNZ instruction executes, CX is automatically decremented by 1. If CX ≠ 0 and ZF = 0, execution will jump to a destination specified by a label in the instruction. If CX = 0, after the auto decrement or if ZF = 1, execution simply go on the next instruction after LOOPNE / LOOPNZ. In other words, the two ways to exit the loop are CX = 0 or ZF = 1. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOPNE / LOOPZ instruction. This instruction does not affect any flags.

```
➢ MOV BX, OFFSET ARRAY    Point BX to adjust before start of array
  DEC BX                  Decrement BX
  MOV CX, 100             Put number of array in CX
  NEXT: INC BX            Point to next element in array
  CMP [BX], ODH           Compare array element with 0DH
  LOOPNZ NEXT
```

CALL (CALL A PROCEDURE)

The CALL instruction is used to transfer execution to a subprogram or a procedure. There two basic type of calls *near* and *far*.

1. *A near* call is a call to a procedure, which is in the same code segment as the CALL instruction. When the 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the offset of the next instruction after the CALL into the stack. This offset saved in the stack is referred to as the return address, because this is the address that execution will return to after the procedure is executed. A near CALL instruction will also load the instruction pointer with the offset of the first instruction in the procedure. A RET instruction at the end of the procedure will return execution to the offset saved on the stack which is copied back to IP.
2. A *far* call is a call to a procedure, which is in a different segment from the one that contains the CALL instruction. When the 8086 executes a far call, it decrements the stack pointer by 2 and copies the content of the CS register to the stack. It then decrements the stack pointer by 2 again and copies the offset of the instruction after the CALL instruction to the stack. Finally, it loads CS with the segment base of the segment that contains the procedure, and loads IP with the offset of the first instruction of the procedure in that segment. A RET instruction at the end of the procedure will return execution to the next instruction after the CALL by restoring the saved values of CS and IP from the stack.

- CALL MULT

This is a direct within segment (near or intra segment) call. MULT is the name of the procedure. The assembler determines the displacement of MULT from the instruction after the CALL and codes this displacement in as part of the instruction.

- CALL BX

This is an indirect within-segment (near or intra-segment) call. BX contains the offset of the first instruction of the procedure. It replaces content of IP with content of register BX.

- CALL WORD PTR [BX]

This is an indirect within-segment (near or intra-segment) call. Offset of the first instruction of the procedure is in two memory addresses in DS. Replaces content of IP with content of word memory location in DS pointed to by BX.

- CALL DIVIDE

This is a direct call to another segment (far or inter-segment call). DIVIDE is the name of the procedure. The procedure must be declared far with DIVIDE PROC FAR at its start. The assembler will determine the code segment base for the segment that contains the procedure and the offset of the start of the procedure. It will put these values in as part of the instruction code.

- CALL DWORD PTR [BX]

This is an indirect call to another segment (far or inter-segment call). New values for CS and IP are fetched from four-memory location in DS. The new value for CS is fetched from [BX] and [BX + 1]; the new IP is fetched from [BX + 2] and [BX +3].

RET (RETURN EXECUTION FROM PROCEDURE TO CALLING PROGRAM)

The RET instruction will return execution from a procedure to the next instruction after the CALL instruction which was used to call the procedure. If the procedure is near procedure (in the same code segment as the CALL instruction), then the return will be done by replacing the IP with a word from the top of the stack. The word from the top of the stack is the offset of the next instruction after the CALL. This offset was pushed into the stack as part of the operation of the CALL instruction. The stack pointer will be incremented by 2 after the return address is popped off the stack.

If the procedure is a far procedure (in a code segment other than the one from which it is called), then the instruction pointer will be replaced by the word at the top of the stack. This word is the offset part of the return address put there by the CALL instruction. The stack pointer will then be incremented by 2. The CS register is then replaced with a word from the new top of the stack. This word is the segment base part of the return address that was pushed onto the stack by a far call operation. After this, the stack pointer is again incremented by 2.

A RET instruction can be followed by a number, for example, RET 6. In this case, the stack pointer will be incremented by an additional six addresses after the IP when the IP and CS are popped off the stack. This form is used to increment the stack pointer over parameters passed to the procedure on the stack.

The RET instruction does not affect any flag.

STRING MANIPULATION INSTRUCTIONS

MOVS – MOVS Destination String Name, Source String Name
MOVSB – MOVSB Destination String Name, Source String Name
MOVSW – MOVSW Destination String Name, Source String Name

This instruction copies a byte or a word from location in the data segment to a location in the extra segment. The offset of the source in the data segment must be in the SI register. The offset of the destination in the extra segment must be in the DI register. For multiple-byte or multiple-word moves, the

number of elements to be moved is put in the CX register so that it can function as a counter. After the byte or a word is moved, SI and DI are automatically adjusted to point to the next source element and the next destination element. If DF is 0, then SI and DI will incremented by 1 after a byte move and by 2 after a word move. If DF is 1, then SI and DI will be decremented by 1 after a byte move and by 2 after a word move. MOVS does not affect any flag.

When using the MOVS instruction, you must in some way tell the assembler whether you want to move a string as bytes or as word. There are two ways to do this. The first way is to indicate the name of the source and destination strings in the instruction, as, for example. MOVS DEST, SRC. The assembler will code the instruction for a byte / word move if they were declared with a DB / DW. The second way is to add a "B" or a "W" to the MOVS mnemonic. MOVSB says move a string as bytes; MOVSW says move a string as words.

➢ MOV SI, OFFSET SOURCE	Load offset of start of source string in DS into SI
MOV DI, OFFSET DESTINATION	Load offset of start of destination string in ES into DI
CLD	Clear DF to auto increment SI and DI after move
MOV CX, 04H	Load length of string into CX as counter
REP MOVSB	Move string byte until CX = 0

LODS / LODSB / LODSW (LOAD STRING BYTE INTO AL OR STRING WORD INTO AX)

This instruction copies a byte from a string location pointed to by SI to AL, or a word from a string location pointed to by SI to AX. If DF is 0, SI will be automatically incremented (by 1 for a byte string, and 2 for a word string) to point to the next element of the string. If DF is 1, SI will be automatically decremented (by 1 for a byte string, and 2 for a word string) to point to the previous element of the string. LODS does not affect any flag.

➢ CLD	Clear direction flag so that SI is auto-incremented
MOV SI, OFFSET SOURCE	Point SI to start of string
LODS SOURCE	Copy a byte or a word from string to AL or AX

Note: The assembler uses the name of the string to determine whether the string is of type bye or type word. Instead of using the string name to do this, you can use the mnemonic LODSB to tell the assembler that the string is type byte or the mnemonic LODSW to tell the assembler that the string is of type word.

STOS / STOSB / STOSW (STORE STRING BYTE OR STRING WORD)

This instruction copies a byte from AL or a word from AX to a memory location in the extra segment pointed to by DI. In effect, it replaces a string element with a byte from AL or a word from AX. After the copy, DI is automatically incremented or decremented to point to next or previous element of the string. If DF is cleared, then DI will automatically incremented by 1 for a byte string and by 2 for a word string. If DI is set, DI will be automatically decremented by 1 for a byte string and by 2 for a word string. STOS does not affect any flag.

➢ MOV DI, OFFSET TARGET
STOS TARGET

Note: The assembler uses the string name to determine whether the string is of type byte or type word. If it is a byte string, then string byte is replaced with content of AL. If it is a word string, then string word is replaced with content of AX.

➢ MOV DI, OFFSET TARGET
STOSB

"B" added to STOSB mnemonic tells assembler to replace byte in string with byte from AL. STOSW would tell assembler directly to replace a word in the string with a word from AX.

CMPS / CMPSB / CMPSW (COMPARE STRING BYTES OR STRING WORDS)

This instruction can be used to compare a byte / word in one string with a byte / word in another string. SI is used to hold the offset of the byte or word in the source string, and DI is used to hold the offset of the byte or word in the destination string.

The AF, CF, OF, PF, SF, and ZF flags are affected by the comparison, but the two operands are not affected. After the comparison, SI and DI will automatically be incremented or decremented to point to the next or previous element in the two strings. If DF is set, then SI and DI will automatically be decremented by 1 for a byte string and by 2 for a word string. If DF is reset, then SI and DI will automatically be incremented by 1 for byte strings and by 2 for word strings. The string pointed to by SI must be in the data segment. The string pointed to by DI must be in the extra segment.

The CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.

```
➢ MOV SI, OFFSET FIRST      Point SI to source string
  MOV DI, OFFSET SECOND     Point DI to destination string
  CLD                       DF cleared, SI and DI will auto-increment after compare
  MOV CX, 100               Put number of string elements in CX
  REPE CMPSB                Repeat the comparison of string bytes until end of string
                            or until compared bytes are not equal
```

CX functions as a counter, which the REPE prefix will cause CX to be decremented after each compare. The B attached to CMPS tells the assembler that the strings are of type byte. If you want to tell the assembler that strings are of type word, write the instruction as CMPSW. The REPE CMPSW instruction will cause the pointers in SI and DI to be incremented by 2 after each compare, if the direction flag is set.

SCAS / SCASB / SCASW (SCAN A STRING BYTE OR A STRING WORD)

SCAS compares a byte in AL or a word in AX with a byte or a word in ES pointed to by DI. Therefore, the string to be scanned must be in the extra segment, and DI must contain the offset of the byte or the word to be compared. If DF is cleared, then DI will be incremented by 1 for byte strings and by 2 for word strings. If DF is set, then DI will be decremented by 1 for byte strings and by 2 for word strings. SCAS affects AF, CF, OF, PF, SF, and ZF, but it does not change either the operand in AL (AX) or the operand in the string.

The following program segment scans a text string of 80 characters for a carriage return, 0DH, and puts the offset of string into DI:

```
➢ MOV DI, OFFSET STRING
  MOV AL, 0DH               Byte to be scanned for into AL
  MOV CX, 80                CX used as element counter
  CLD                       Clear DF, so that DI auto increments
  REPNE SCAS STRING         Compare byte in string with byte in AL
```

REP / REPE / REPZ / REPNE / REPNZ (PREFIX)
(REPEAT STRING INSTRUCTION UNTIL SPECIFIED CONDITIONS EXIST)

REP is a prefix, which is written before one of the string instructions. It will cause the CX register to be decremented and the string instruction to be repeated until CX = 0. The instruction REP MOVSB, for example, will continue to copy string bytes until the number of bytes loaded into CX has been copied.

REPE and REPZ are two mnemonics for the same prefix. They stand for *repeat if equal* and *repeat if zero*, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are equal (ZF = 1) and CX is not yet counted down to zero. In other words, there are two conditions that will stop the repetition: CX = 0 or string bytes or words not equal.

- REPE CMPSB — Compare string bytes until end of string or until string bytes not equal.

REPNE and REPNZ are also two mnemonics for the same prefix. They stand for *repeat if not equal* and *repeat if not zero*, respectively. They are often used with the Compare String instruction or with the Scan String instruction. They will cause the string instruction to be repeated as long as the compared bytes or words are not equal (ZF = 0) and CX is not yet counted down to zero.

- REPNE SCASW — Scan a string of word until a word in the string matches the word in AX or until all of the string has been scanned.

The string instruction used with the prefix determines which flags are affected.

FLAG MANIPULATION INSTRUCTIONS

STC (SET CARRY FLAG)

This instruction sets the carry flag to 1. It does not affect any other flag.

CLC (CLEAR CARRY FLAG)

This instruction resets the carry flag to 0. It does not affect any other flag.

CMC (COMPLEMENT CARRY FLAG)

This instruction complements the carry flag. It does not affect any other flag.

STD (SET DIRECTION FLAG)

This instruction sets the direction flag to 1. It does not affect any other flag.

CLD (CLEAR DIRECTION FLAG)

This instruction resets the direction flag to 0. It does not affect any other flag.

STI (SET INTERRUPT FLAG)

Setting the interrupt flag to a 1 enables the INTR interrupt input of the 8086. The instruction will not take affect until the next instruction after STI. When the INTR input is enabled, an interrupt signal on this input will then cause the 8086 to interrupt program execution, push the return address and flags on the stack, and execute an interrupt service procedure. An IRET instruction at the end of the interrupt service procedure will restore the return address and flags that were pushed onto the stack and return execution to the interrupted program. STI does not affect any other flag.

CLI (CLEAR INTERRUPT FLAG)

This instruction resets the interrupt flag to 0. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. The CLI instructions, however, has no effect on the non-maskable interrupt input, NMI. It does not affect any other flag.

LAHF (COPY LOW BYTE OF FLAG REGISTER TO AH REGISTER)

The LAHF instruction copies the low-byte of the 8086 flag register to AH register. It can then be pushed onto the stack along with AL by a PUSH AX instruction. LAHF does not affect any flag.

SAHF (COPY AH REGISTER TO LOW BYTE OF FLAG REGISTER)

The SAHF instruction replaces the low-byte of the 8086 flag register with a byte from the AH register. SAHF changes the flags in lower byte of the flag register.

STACK RELATED INSTRUCTIONS

PUSH – PUSH Source

The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment to which the stack pointer points. The source of the word can be general-purpose register, segment register, or memory. The stack segment register and the stack pointer must be initialized before this instruction can be used. PUSH can be used to save data on the stack so that it will not destroyed by a procedure. This instruction does not affect any flag.

➢ PUSH BX	Decrement SP by 2, copy BX to stack.
➢ PUSH DS	Decrement SP by 2, copy DS to stack.
➢ PUSH BL	Illegal; must push a word
➢ PUSH TABLE [BX]	Decrement SP by 2, and copy word from memory in DS at EA = TABLE + [BX] to stack

POP – POP Destination

The POP instruction copies a word from the stack location pointed to by the stack pointer to a destination specified in the instruction. The destination can be a general-purpose register, a segment register or a memory location. The data in the stack is not changed. After the word is copied to the specified destination, the stack pointer is automatically incremented by 2 to point to the next word on the stack. The POP instruction does not affect any flag.

➢ POP DX	Copy a word from top of stack to DX; increment SP by 2
➢ POP DS	Copy a word from top of stack to DS; increment SP by 2
➢ POP TABLE [DX]	Copy a word from top of stack to memory in DS with EA = TABLE + [BX]; increment SP by 2.

PUSHF (PUSH FLAG REGISTER TO STACK)

The PUSHF instruction decrements the stack pointer by 2 and copies a word in the flag register to two memory locations in stack pointed to by the stack pointer. The stack segment register is not affected. This instruction does to affect any flag.

POPF (POP WORD FROM TOP OF STACK TO FLAG REGISTER)

The POPF instruction copies a word from two memory locations at the top of the stack to the flag register and increments the stack pointer by 2. The stack segment register and word on the stack are not affected. This instruction does to affect any flag.

INPUT-OUTPUT INSTRUCTIONS

IN – IN Accumulator, Port

The IN instruction copies data from a port to the AL or AX register. If an 8-bit port is read, the data will go to AL. If a 16-bit port is read, the data will go to AX.

The IN instruction has two possible formats, fixed port and variable port. For fixed port type, the 8-bit address of a port is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

- IN AL, OC8H — Input a byte from port OC8H to AL
- IN AX, 34H — Input a word from port 34H to AX

For the variable-port form of the IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16-bit register, the port address can be any number between 0000H and FFFFH. Therefore, up to 65,536 ports are addressable in this mode.

- MOV DX, 0FF78H — Initialize DX to point to port
  IN AL, DX — Input a byte from 8-bit port 0FF78H to AL
  IN AX, DX — Input a word from 16-bit port 0FF78H to AX

The variable-port IN instruction has advantage that the port address can be computed or dynamically determined in the program. Suppose, for example, that an 8086-based computer needs to input data from 10 terminals, each having its own port address. Instead of having a separate procedure to input data from each port, you can write one generalized input procedure and simply pass the address of the desired port to the procedure in DX.

The IN instruction does not change any flag.

OUT – OUT Port, Accumulator

The OUT instruction copies a byte from AL or a word from AX to the specified port. The OUT instruction has two possible forms, fixed port and variable port.

For the fixed port form, the 8-bit port address is specified directly in the instruction. With this form, any one of 256 possible ports can be addressed.

- OUT 3BH, AL — Copy the content of AL to port 3BH
- OUT 2CH, AX — Copy the content of AX to port 2CH

For variable port form of the OUT instruction, the content of AL or AX will be copied to the port at an address contained in DX. Therefore, the DX register must be loaded with the desired port address before this form of the OUT instruction is used.

- MOV DX, 0FFF8H — Load desired port address in DX
  OUT DX, AL — Copy content of AL to port FFF8H
  OUT DX, AX — Copy content of AX to port FFF8H

The OUT instruction does not affect any flag.

MISCELLANEOUS INSTRUCTIONS

HLT (HALT PROCESSING)

The HLT instruction causes the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The different ways to get the processor out of the halt state are with an interrupt signal on the INTR pin, an interrupt signal on the NMI pin, or a reset signal on the RESET input.

NOP (PERFORM NO OPERATION)

This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction. The NOP instruction can be used to increase the delay of a delay loop. When hand coding, a NOP can also be used to hold a place in a program for an instruction that will be added later. NOP does not affect any flag.

ESC (ESCAPE)

This instruction is used to pass instructions to a coprocessor, such as the 8087 Math coprocessor, which shares the address and data bus with 8086. Instructions for the coprocessor are represented by a 6-bit code embedded in the ESC instruction. As the 8086 fetches instruction bytes, the coprocessor also fetches these bytes from the data bus and puts them in its queue. However, the coprocessor treats all the normal 8086 instructions as NOPs. When 8086 fetches an ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6-bit code specified in the instruction. In most cases, the 8086 treats the ESC instruction as a NOP. In some cases, the 8086 will access a data item in memory for the coprocessor.

INT – INT TYPE

The term *type* in the instruction format refers to a number between 0 and 255, which identify the interrupt. When an 8086 executes an INT instruction, it will

1. Decrement the stack pointer by 2 and push the flags on to the stack.
2. Decrement the stack pointer by 2 and push the content of CS onto the stack.
3. Decrement the stack pointer by 2 and push the offset of the next instruction after the INT number instruction on the stack.
4. Get a new value for IP from an absolute memory address of 4 times the type specified in the instruction. For an INT 8 instruction, for example, the new IP will be read from address 00020H.
5. Get a new for value for CS from an absolute memory address of 4 times the type specified in the instruction plus 2, for an INT 8 instruction, for example, the new value of CS will be read from address 00022H.
6. Reset both IF and TF. Other flags are not affected.

- INT 35 — New IP from 0008CH, new CS from 0008Eh
- INT 3 — This is a special form, which has the single-byte code of CCH; Many systems use this as a break point instruction (Get new IP from 0000CH new CS from 0000EH).

INTO (INTERRUPT ON OVERFLOW)

If the overflow flag (OF) is set, this instruction causes the 8086 to do an indirect far call to a procedure you write to handle the overflow condition. Before doing the call, the 8086 will

1. Decrement the stack pointer by 2 and push the flags on to the stack.
2. Decrement the stack pointer by 2 and push CS on to the stack.
3. Decrement the stack pointer by 2 and push the offset of the next instruction after INTO instruction onto the stack.

4. Reset TF and IF. Other flags are not affected. To do the call, the 8086 will read a new value for IP from address 00010H and a new value of CS from address 00012H.

IRET (INTERRUPT RETURN)

When the 8086 responds to an interrupt signal or to an interrupt instruction, it pushes the flags, the current value of CS, and the current value of IP onto the stack. It then loads CS and IP with the starting address of the procedure, which you write for the response to that interrupt. The IRET instruction is used at the end of the interrupt service procedure to return execution to the interrupted program. To do this return, the 8086 copies the saved value of IP from the stack to IP, the stored value of CS from the stack to CS, and the stored value of the flags back to the flag register. Flags will have the values they had before the interrupt, so any flag settings from the procedure will be lost unless they are specifically saved in some way.

LOCK – ASSERT BUS LOCK SIGNAL

Many microcomputer systems contain several microprocessors. Each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a system bus so that each can access system resources such as disk drive or memory. Each microprocessor takes control of the system bus only when it needs to access some system resources. The LOCK prefix allows a microprocessor to make sure that another processor does not take control of the system bus while it is in the middle of a critical instruction, which uses the system bus. The LOCK prefix is put in front of the critical instruction. When an instruction with a LOCK prefix executes, the 8086 will assert its external bus controller device, which then prevents any other processor from taking over the system bus. LOCK instruction does not affect any flag.

- LOCK XCHG SAMAPHORE, AL

The XCHG instruction requires two bus accesses. The LOCK prefix prevents another processor from taking control of the system bus between the two accesses.

WAIT – WAIT FOR SIGNAL OR INTERRUPT SIGNAL

When this instruction is executed, the 8086 enters an idle condition in which it is doing no processing. The 8086 will stay in this idle state until the 8086 test input pin is made low or until an interrupt signal is received on the INTR or the NMI interrupt input pins. If a valid interrupt occurs while the 8086 is in this idle state, the 8086 will return to the idle state after the interrupt service procedure executes. It returns to the idle state because the address of the WAIT instruction is the address pushed on the stack when the 8086 responds to the interrupt request. WAIT does not affect any flag. The WAIT instruction is used to synchronize the 8086 with external hardware such as the 8087 Math coprocessor.

XLAT / XLATB – TRANSLATE A BYTE IN AL

The XLATB instruction is used to translate a byte from one code (8 bits or less) to another code (8 bits or less). The instruction replaces a byte in AL register with a byte pointed to by BX in a lookup table in the memory. Before the XLATB instruction can be executed, the lookup table containing the values for a new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX. The code byte to be translated is put in AL. The XLATB instruction adds the byte in AL to the offset of the start of the table in BX. It then copies the byte from the address pointed to by (BX + AL) back into AL. XLATB instruction does not affect any flag.

8086 routine to convert ASCII code byte to EBCDIC equivalent: ASCII code byte is in AL at the start, EBCDIC code in AL after conversion.

```
MOV BX, OFFSET EBCDIC    Point BX to the start of EBCDIC table in DS
XLATB                    Replace ASCII in AL with EBCDIC from table.
```

8086 ASSEMBLER DIRECTIVES

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to "bracket" a logical segment containing code of data.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement CODE SEGMENT WORD tells the assembler that we want the content of this segment located on the next available word (even address) when segments ate combined and given absolute addresses. Without this WORD addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the assembler that the segment may be put together with other segments named CODE from other assembly modules when the modules are linked together.

ENDS (END SEGMENT)

This directive is used with the name of a segment to indicate the end of that logical segment.

➢ CODE SEGMENT	Start of logical segment containing code instruction statements
CODE ENDS	End of segment named CODE

END (END PROCEDURE)

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

ASSUME

The ASSUME directive is used tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

➢ PRICES DB 49H, 98H, 29H	Declare array of 3 bytes named PRICE and initialize them with specified values.
➢ NAMES DB "THOMAS"	Declare array of 6 bytes and initialize with ASCII codes for the letters in THOMAS.
➢ TEMP DB 100 DUP (?)	Set aside 100 bytes of storage in memory and give it the name TEMP. But leave the 100 bytes un-initialized.
➢ PRESSURE DB 20H DUP (0)	Set aside 20H bytes of storage in memory, give it the name PRESSURE and put 0 in all 20H locations.

DD (DEFINE DOUBLE WORD)

The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word. The statement ARRAY DD 25629261H, for example, will define a double word named ARRAY and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word.

DQ (DEFINE QUADWORD)

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory. The statement BIG_NUMBER DQ 243598740192A92BH, for example, will declare a variable named BIG_NUMBER and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run.

DT (DEFINE TEN BYTES)

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory. The statement PACKED_BCD DT 11223344556677889900 will declare an array named PACKED_BCD, which is 10 bytes in length. It will initialize the 10 bytes with the values 11, 22, 33, 44, 55, 66, 77, 88, 99, and 00 when the program is loaded into memory to be run. The statement RESULT DT 20H DUP (0) will declare an array of 20H blocks of 10 bytes each and initialize all 320 bytes to 00 when the program is loaded into memory to be run.

DW (DEFINE WORD)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER, and initialized with the value 437AH when the program is loaded into memory to be run.

➢ WORDS DW 1234H, 3456H	Declare an array of 2 words and initialize them with the specified values.
➢ STORAGE DW 100 DUP (0)	Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named as STORAGE.
➢ STORAGE DW 100 DUP (?)	Reserve 100 word of storage in memory and give it the name STORAGE, but leave the words un-initialized.

EQU (EQUATE)

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name. Suppose, for example, you write the statement FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement ADD AL, FACTOR. When the assembler codes this instruction statement, it will code it as if you had written the instruction ADD AL, 03H.

➢ CONTROL EQU 11000110 B	Replacement
MOV AL, CONTROL	Assignment
➢ DECIMAL_ADJUST EQU DAA	Create clearer mnemonic for DAA
ADD AL, BL	Add BCD numbers
DECIMAL_ADJUST	Keep result in BCD format

LENGTH

LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array. When the assembler reads the statement MOV CX, LENGTH STRING1, for example, will determine the number of elements in STRING1 and load it into CX. If the string was declared as a string of bytes, LENGTH will produce the number of bytes in the string. If the string was declared as a word string, LENGTH will produce the number of words in the string.

OFFSET

OFFSET is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it. When the assembler reads the statement MOV BX, OFFSET PRICES, for example, it will determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and will load this value into BX.

PTR (POINTER)

The PTR operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction INC [BX], for example, it will not know whether to increment the byte pointed to by BX. We use the PTR operator to clarify how we want the assembler to code the instruction. The statement INC BYTE PTR [BX] tells the assembler that we want to increment the byte pointed to by BX. The statement INC WORD PTR [BX] tells the assembler that we want to increment the word pointed to by BX. The PTR operator assigns the type specified before PTR to the variable specified after PTR.

We can also use the PTR operator to clarify our intentions when we use indirect Jump instructions. The statement JMP [BX], for example, does not tell the assembler whether to code the instruction for a near jump. If we want to do a near jump, we write the instruction as JMP WORD PTR [BX]. If we want to do a far jump, we write the instruction as JMP DWORD PTR [BX].

EVEN (ALIGN ON EVEN MEMORY ADDRESS)

As an assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The EVEN directive tells the assembler to increment the location counter to the next even address, if it is not already at an even address. A NOP instruction will be inserted in the location incremented over.

- DATA SEGMENT

```
SALES DB 9 DUP (?)          Location counter will point to 0009 after this instruction.
EVEN                        Increment location counter to 000AH
INVENTORY DW 100 DUP (0)    Array of 100 words starting on even address for quicker read
DATA ENDS
```

PROC (PROCEDURE)

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term *near* or the term *far* is used to specify the type of the procedure. The statement DIVIDE PROC FAR, for example, identifies the start of a procedure named DIVIDE and tells the assembler that the procedure is far (in a segment with different name from the one that contains the instructions which calls the procedure). The PROC directive is used with the ENDP directive to “bracket” a procedure.

ENDP (END PROCEDURE)

The directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The directive, together with the procedure directive, PROC, is used to "bracket" a procedure.

```
➢ SQUARE_ROOT PROC      Start of procedure.
  SQUARE_ROOT ENDP      End of procedure.
```

ORG (ORIGIN)

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The location counter is automatically set to 0000 when assembler starts reading a segment. The ORG directive allows you to set the location counter to a desired value at any point in the program. The statement ORG 2000H tells the assembler to set the location counter to 2000H, for example.

A "$" it often used to symbolically represent the current value of the location counter, the $ actually represents the next available byte location where the assembler can put a data or code byte. The $ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value. The statement ORG $ + 100 tells the assembler increment the value of the location counter by 100 from its current value.

NAME

The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written.

LABEL

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to be keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term that specifics the type you want to associate with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as type *near* or type *far*. If the label is going to be used to reference a data item, then the label must be specified as type byte, type word, or type double word. Here's how we use the LABEL directive for a jump address.

```
➢ ENTRY_POINT LABEL FAR     Can jump to here from another segment
  NEXT: MOV AL, BL          Can not do a far jump directly to a label with a colon
```

The following example shows how we use the label directive for a data reference.

```
➢ STACK_SEG SEGMENT STACK
  DW 100 DUP (0)            Set aside 100 words for stack
  STACK_TOP LABEL WORD      Give name to next location after last word in stack
  STACK_SEG ENDS
```

To initialize stack pointer, use MOV SP, OFFSET STACK_TOP.

EXTRN

The EXTRN directive is used to tell the assembler that the name or labels following the directive are in some other assembly module. For example, if you want to call a procedure, which in a program module assembled at a different time from that which contains the CALL instruction, you must tell the assembler

that the procedure is external. The assembler will then put this information in the object code file so that the linker can connect the two modules together. For a reference to externally named variable, you must specify the type of the variable, as in the statement EXTRN DIVISOR: WORD. The statement EXTRN DIVIDE: FAR tells the assembler that DIVIDE is a label of type FAR in another assembler module. Name or labels referred to as external in one module must be declared public with the PUBLIC directive in the module in which they are defined.

- PROCEDURE SEGMENT
  EXTRN DIVIDE: FAR Found in segment PROCEDURES
  PROCEDURE ENDS

PUBLIC

Large program are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared PUBLIC in the module in which it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules. An example is the statement PUBLIC DIVISOR, DIVIDEND, which makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

SHORT

The SHORT operator is used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction in the program. The destination must in the range of –128 bytes to +127 bytes from the address of the instruction after the jump. The statement JMP SHORT NEARBY_LABEL is an example of the use of SHORT.

TYPE

The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually determines the number of bytes in the type of the variable. For a byte-type variable, the assembler will give a value of 1, for a word-type variable, the assembler will give a value of 2, and for a double word-type variable, it will give a value of 4. It can be used in instruction such as ADD BX, TYPE-WORD-ARRAY, where we want to increment BX to point to the next word in an array of words.

GLOBAL (DECLARE SYMBOLS AS PUBLIC OR EXTRN)

The GLOBAL directive can be used in place of a PUBLIC directive or in place of an EXTRN directive. For a name or symbol defined in the current assembly module, the GLOBAL directive is used to make the symbol available to other modules. The statement GLOBAL DIVISOR, for example, makes the variable DIVISOR public so that it can be accessed from other assembly modules.

INCLUDE (INCLUDE SOURCE CODE FROM FILE)

This directive is used to tell the assembler to insert a block of source code from the named file into the current source module.

Complete 8086 instruction set

Quick reference:

	CMPSB				MOV		
AAA	CMPSW	JAE	JNBE	JPO	MOVSB	RCR	SCASB
AAD	CWD	JB	JNC	JS	MOVSW	REP	SCASW
AAM	DAA	JBE	JNE	JZ	MUL	REPE	SHL
AAS	DAS	JC	JNG	LAHF	NEG	REPNE	SHR
ADC	DEC	JCXZ	JNGE	LDS	NOP	REPNZ	STC
ADD	DIV	JE	JNL	LEA	NOT	REPZ	STD
AND	HLT	JG	JNLE	LES	OR	RET	STI
CALL	IDIV	JGE	JNO	LODSB	OUT	RETF	STOSB
CBW	IMUL	JL	JNP	LODSW	POP	ROL	STOSW
CLC	IN	JLE	JNS	LOOP	POPA	ROR	SUB
CLD	INC	JMP	JNZ	LOOPE	POPF	SAHF	TEST
CLI	INT	JNA	JO	LOOPNE	PUSH	SAL	XCHG
CMC	INTO	JNAE	JP	LOOPNZ	PUSHA	SAR	XLATB
CMP	IRET	JNB	JPE	LOOPZ	PUSHF	SBB	XOR
	JA				RCL		

Operand types:

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

SREG: DS, ES, SS, and only as second operand: CS.

memory: [BX], [BX+SI+7], variable, etc...(see **Memory Access**).

immediate: 5, -24, 3Fh, 10001101b, etc...

Notes:

- When two operands are required for an instruction they are separated by comma. For example:

```
REG, memory
```

- When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:

```
AL, DL
DX, AX
m1 DB ?
AL, m1
m2 DW ?
AX, m2
```

- Some instructions allow several operand combinations. For example:

```
memory, immediate
```

```
REG, immediate

memory, REG
REG, SREG
```

- Some examples contain macros, so it is advisable to use **Shift + F8** hot key to *Step Over* (to make macro code execute at maximum speed set **step delay** to zero), otherwise emulator will step through each instruction of a macro. Here is an example that uses PRINTN macro:

```
include 'emu8086.inc'
ORG 100h
MOV AL, 1
MOV BL, 2
PRINTN 'Hello World!'   ; macro.
MOV CL, 3
PRINTN 'Welcome!'       ; macro.
RET
```

These marks are used to show the state of the flags:

1 - instruction sets this flag to **1**.
0 - instruction sets this flag to **0**.
r - flag value depends on result of the instruction.
? - flag value is undefined (maybe **1** or **0**).

Some instructions generate exactly the same machine code, so disassembler may have a problem decoding to your original code. This is especially important for Conditional Jump instructions (see "Program Flow Control" in Tutorials for more information).

Instructions in alphabetical order:

Instruction	Operands	Description
		ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values. It works according to the following Algorithm: `if low nibble of AL > 9 or AF = 1 then:`

<table>
<tr>
<td>AAA</td>
<td>No operands</td>
<td>

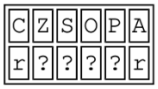
AL = AL + 6
AH = AH + 1
AF = 1
CF = 1

else

AF = 0
CF = 0

in both cases:
clear the high nibble of AL.

Example:

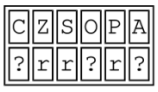
<pre>MOV AX, 15   ; AH = 00, AL = 0Fh
AAA   ; AH = 01, AL = 05
RET</pre>


 
</td>
</tr>
<tr>
<td>AAD</td>
<td>No operands</td>
<td>ASCII Adjust before Division.
Prepares two BCD values for division.

Algorithm:

AL = (AH * 10) + AL
AH = 0

Example:

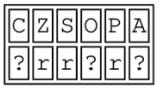
<pre>MOV AX, 0105h   ; AH = 01, AL = 05
AAD   ; AH = 00, AL = 0Fh (15)
RET</pre>


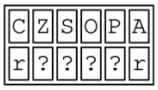
 
</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASCII Adjust after Multiplication.
Corrects the result of multiplication of two BCD values.

Algorithm:

AH = AL / 10
AL = remainder

</td>
</tr>
</table>

AAM	No operands	Example:
		`MOV AL, 15 ; AL = 0Fh`
		`AAM ; AH = 01, AL = 05`
		`RET`
		 

AAS	No operands	ASCII Adjust after Subtraction.
		Corrects result in AH and AL after subtraction when working with BCD values.
		Algorithm:
		`if low nibble of AL > 9 or AF = 1 then:`
		• AL = AL - 6
		• AH = AH - 1
		• AF = 1
		• CF = 1
		`else`
		• AF = 0
		• CF = 0
		`in both cases:`
		`clear the high nibble of AL.`
		Example:
		`MOV AX, 02FFh ; AH = 02, AL = 0FFh`
		`AAS ; AH = 01, AL = 09`
		`RET`
		 

ADC	REG, memory memory, REG REG, REG	Add with Carry. Algorithm: `operand1 = operand1 + operand2 + CF` Example:

	memory, immediate REG, immediate	STC ; set CF = 1 MOV AL, 5 ; AL = 5 ADC AL, 1 ; AL = 7 RET C Z S O P A r r r r r r

ADD	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		Add.
		
		Algorithm:
		
		operand1 = operand1 + operand2
		
		Example:
		
		MOV AL, 5 ; AL = 5
		ADD AL, -3 ; AL = 2
		RET
		
		C Z S O P A
		r r r r r r

AND	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical AND between all bits of two operands. Result is stored in operand1. These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Example: MOV AL, 'a' ; AL = 01100001b AND AL, 11011111b ; AL = 01000001b ('A') RET C Z S O P 0 r r 0 r
		Transfers control to procedure, return address is (IP) is pushed to stack. *4-byte address* may be entered in this form: 1234h:5678h, first value is a

CALL	procedure name
	label
	4-byte address
		segment second value is an offset (this is a far call, so CS is also pushed to stack).
		
		Example:
		
		`ORG 100h  ; for COM file.`
		
		`CALL p1`
		
		`ADD AX, 1`
		
		`RET  ; return to OS.`
		
		`p1 PROC  ; procedure declaration.`
		`  MOV AX, 1234h`
		`  RET  ; return to caller.`
		`p1 ENDP`
		
		C Z S O P A
		unchanged
		
		 

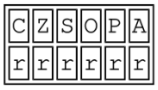
CBW	No operands	Convert byte into word.
		
		Algorithm:
		
		`if high bit of AL = 1 then:`
		
		• `AH = 255 (0FFh)`
		
		`else`
		
		• `AH = 0`
		
		Example:
		
		`MOV AX, 0  ; AH = 0, AL = 0`
		`MOV AL, -5  ; AX = 00FBh (251)`
		`CBW  ; AX = 0FFFBh (-5)`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

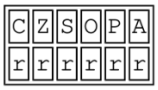
		Clear Carry flag. Algorithm: `CF = 0`

CLC	No operands	
CLD	No operands	Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: `DF = 0` 
CLI	No operands	Clear Interrupt enable flag. This disables hardware interrupts. Algorithm: `IF = 0` 

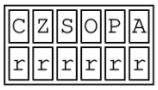
CMC	No operands	Complement Carry flag. Inverts value of CF.
		
		Algorithm:
		
		`if CF = 1 then CF = 0`
		`if CF = 0 then CF = 1`
		
		 

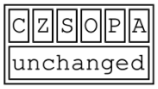
		Compare. Algorithm: `operand1 - operand2`

CMP	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result.
		
		Example:
		
		`MOV AL, 5`
		`MOV BL, 5`
		`CMP AL, BL ; AL = 5, ZF = 1 (so equal!)`
		`RET`
		
		 
		
		 

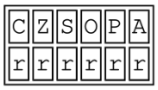
CMPSB	No operands	Compare bytes: ES:[DI] from DS:[SI].
		
		Algorithm:
		
		• DS:[SI] - ES:[DI]
		• set flags according to result: OF, SF, ZF, AF, PF, CF
		• if DF = 0 then
		  ○ SI = SI + 1
		  ○ DI = DI + 1
		  else
		  ○ SI = SI - 1
		  ○ DI = DI - 1
		
		Example:
		open **cmpsb.asm** from c:\emu8086\examples
		
		 
		
		 

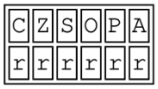
CMPSW	No operands	Compare words: ES:[DI] from DS:[SI].
		
		Algorithm:
		
		• DS:[SI] - ES:[DI]
		• set flags according to result: OF, SF, ZF, AF, PF, CF
		• if DF = 0 then
		  ○ SI = SI + 2
		  ○ DI = DI + 2
		  else
		  ○ SI = SI - 2
		  ○ DI = DI - 2

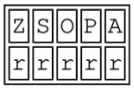
		example: open **cmpsw.asm** from c:\emu8086\examples  

CWD	No operands	Convert Word to Double word.
		Algorithm:
		`if high bit of AX = 1 then:`
		• `DX = 65535 (0FFFFh)`
		`else`
		• `DX = 0`
		Example:
		`MOV DX, 0 ; DX = 0`
		`MOV AX, 0 ; AX = 0`
		`MOV AX, -5 ; DX AX = 00000h:0FFFBh`
		`CWD ; DX AX = 0FFFFh:0FFFBh`
		`RET`
		 
		
		 

DAA	No operands	Decimal adjust After Addition.
		Corrects the result of addition of two packed BCD values.
		Algorithm:
		`if low nibble of AL > 9 or AF = 1 then:`
		• `AL = AL + 6`
		• `AF = 1`
		`if AL > 9Fh or CF = 1 then:`
		• `AL = AL + 60h`
		• `CF = 1`
		Example:

		MOV AL, 0Fh ; AL = 0Fh (15) DAA ; AL = 15h RET  

DAS	No operands	Decimal adjust After Subtraction.
		Corrects the result of subtraction of two packed BCD values.
		
		Algorithm:
		
		if low nibble of AL > 9 or AF = 1 then:
		
		• AL = AL - 6
		• AF = 1
		
		if AL > 9Fh or CF = 1 then:
		
		• AL = AL - 60h
		• CF = 1
		
		Example:
		
		MOV AL, 0FFh ; AL = 0FFh (-1)
		DAS ; AL = 99h, CF = 1
		RET
		
		 
		
		 

DEC	REG memory	Decrement. Algorithm: operand = operand - 1 Example: MOV AL, 255 ; AL = 0FFh (255 or -1) DEC AL ; AL = 0FEh (254 or -2) RET  CF - unchanged! 

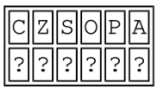
<table>
<tr>
<td>DIV</td>
<td>REG
memory</td>
<td>Unsigned divide.

Algorithm:

when operand is a byte:
AL = AX / operand
AH = remainder (modulus)

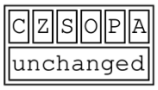
when operand is a word:
AX = (DX AX) / operand
DX = remainder (modulus)

Example:

<pre>MOV AX, 203   ; AX = 00CBh
MOV BL, 4
DIV BL   ; AL = 50 (32h), AH = 3
RET</pre> 

 </td>
</tr>
<tr>
<td>HLT</td>
<td>No operands</td>
<td>Halt the System.

Example:

<pre>MOV AX, 5
HLT</pre> 

 </td>
</tr>
<tr>
<td>IDIV</td>
<td>REG
memory</td>
<td>Signed divide.

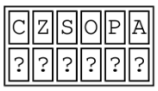
Algorithm:

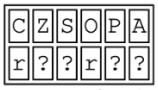
when operand is a byte:
AL = AX / operand
AH = remainder (modulus)

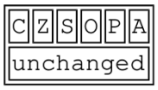
when operand is a word:
AX = (DX AX) / operand
DX = remainder (modulus)

Example:

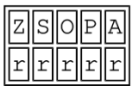
<pre>MOV AX, -203 ; AX = 0FF35h
MOV BL, 4
IDIV BL   ; AL = -50 (0CEh), AH = -3 (0FDh)
RET</pre></td>
</tr>
</table>

IMUL	REG
	memory
		Signed multiply.
		
		Algorithm:
		
		when operand is a **byte**:
		`AX = AL * operand.`
		
		when operand is a **word**:
		`(DX AX) = AX * operand.`
		
		Example:
		
		`MOV AL, -2`
		`MOV BL, -4`
		`IMUL BL ; AX = 8`
		`RET`
		
		 
		
		`CF=OF=0 when result fits into operand of IMUL.`

IN	AL, im.byte
	AL, DX
	AX, im.byte
	AX, DX
		Input from port into **AL** or **AX**.
		Second operand is a port number. If required to access port number over 255 - **DX** register should be used.
		Example:
		
		`IN AX, 4 ; get status of traffic lights.`
		`IN AL, 7 ; get status of stepper-motor.`
		
		 

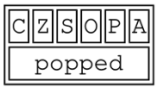
INC	REG
	memory
		Increment.
		
		Algorithm:
		
		`operand = operand + 1`
		
		Example:
		
		`MOV AL, 4`
		`INC AL ; AL = 5`

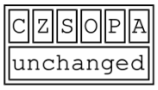
		RET  CF - unchanged! 

INT	immediate byte	Interrupt numbered by immediate byte (0..255).
		Algorithm:
		Push to stack:
		○ flags register
		○ CS
		○ IP
		• IF = 0
		• Transfer control to interrupt procedure
		Example:
		MOV AH, 0Eh ; teletype.
		MOV AL, 'A'
		INT 10h ; BIOS interrupt.
		RET
		 
		
		 

INTO	No operands	Interrupt 4 if Overflow flag is 1.
		Algorithm:
		if OF = 1 then INT 4
		Example:
		; -5 - 127 = -132 (not in -128..127)
		; the result of SUB is wrong (124),
		; so OF = 1 is set:
		MOV AL, -5
		SUB AL, 127 ; AL = 7Ch (124)
		INTO ; process error.
		RET
		 

		Interrupt Return.

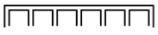
IRET	No operands	Algorithm: Pop from stack: ○ IP ○ CS ○ flags register  

JA	label	Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned.
		Algorithm:
		if (CF = 0) and (ZF = 0) then jump
		Example:
		include 'emu8086.inc'
		ORG 100h
		MOV AL, 250
		CMP AL, 5
		JA label1
		PRINT 'AL is not above 5'
		JMP exit
		label1:
		PRINT 'AL is above 5'
		exit:
		RET
		 
		
		 

JAE	label	Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned.
		Algorithm:
		if CF = 0 then jump
		Example:
		include 'emu8086.inc'
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JAE label1
		PRINT 'AL is not above or equal to 5'
		JMP exit
		label1:

		PRINT 'AL is above or equal to 5' exit: RET C Z S O P A unchanged 

JB	label	Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 then jump
		
		Example:
		
		include 'emu8086.inc'
		ORG 100h
		MOV AL, 1
		CMP AL, 5
		JB label1
		PRINT 'AL is not below 5'
		JMP exit
		label1:
		PRINT 'AL is below 5'
		exit:
		RET
		
		C Z S O P A
		unchanged
		
		 

JBE	label	Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 or ZF = 1 then jump
		
		Example:
		
		include 'emu8086.inc'
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JBE label1
		PRINT 'AL is not below or equal to 5'
		JMP exit
		label1:
		PRINT 'AL is below or equal to 5'
		exit:
		RET
		
		 

		C Z S O P A unchanged 

JC	label	Short Jump if Carry flag is set to 1.
		Algorithm:
		`if CF = 1 then jump`
		Example:
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV AL, 255`
		`ADD AL, 1`
		`JC label1`
		`PRINT 'no carry.'`
		`JMP exit`
		`label1:`
		`PRINT 'has carry.'`
		`exit:`
		`RET`
		C Z S O P A
		unchanged
		 

JCXZ	label	Short Jump if CX register is 0.
		Algorithm:
		`if CX = 0 then jump`
		Example:
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV CX, 0`
		`JCXZ label1`
		`PRINT 'CX is not zero.'`
		`JMP exit`
		`label1:`
		`PRINT 'CX is zero.'`
		`exit:`
		`RET`
		C Z S O P A
		unchanged
		 

		Short Jump if first operand is Equal to second operand (as set by CMP instruction).

JE	label	Signed/Unsigned.
		
		Algorithm:
		
		`if ZF = 1 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV AL, 5`
		`CMP AL, 5`
		`JE  label1`
		`PRINT 'AL is not equal to 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL is equal to 5.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

JG	label	Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		`if (ZF = 0) and (SF = OF) then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV AL, 5`
		`CMP AL, -5`
		`JG  label1`
		`PRINT 'AL is not greater -5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL is greater -5.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

		Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed. Algorithm:

JGE	label	if SF = OF then jump
		
		Example:
		
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, -5`
		`JGE label1`
		`PRINT 'AL < -5'`
		`JMP exit`
		`label1:`
		`PRINT 'AL >= -5'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

JL	label	Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if SF <> OF then jump
		
		Example:
		
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV AL, -2`
		`CMP AL, 5`
		`JL label1`
		`PRINT 'AL >= 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL < 5.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

		Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed. Algorithm: if SF <> OF or ZF = 1 then jump Example:

JLE	label	include 'emu8086.inc'
		ORG 100h
		MOV AL, -2
		CMP AL, 5
		JLE label1
		PRINT 'AL > 5.'
		JMP exit
		label1:
		PRINT 'AL <= 5.'
		exit:
		RET
		
		C Z S O P A
		unchanged
		
		 

JMP	label
	4-byte address
		Unconditional Jump. Transfers control to another part of the program. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.
		
		Algorithm:
		
		always jump
		
		Example:
		
		include 'emu8086.inc'
		ORG 100h
		MOV AL, 5
		JMP label1 ; jump over 2 lines!
		PRINT 'Not Jumped!'
		MOV AL, 0
		label1:
		PRINT 'Got Here!'
		RET
		
		C Z S O P A
		unchanged
		
		 

JNA	label	Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 or ZF = 1 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 2

```
   CMP AL, 5
   JNA label1
   PRINT 'AL is above 5.'
   JMP exit
label1:
   PRINT 'AL is not above 5.'
exit:
   RET
```

C	Z	S	O	P	A
unchanged					

JNAE	label	Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		`if CF = 1 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, 5`
		`JNAE label1`
		`PRINT 'AL >= 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL < 5.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

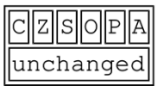
JNB	label	Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		`if CF = 0 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 7`
		`CMP AL, 5`
		`JNB label1`
		`PRINT 'AL < 5.'`

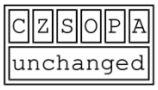
```
   JMP exit
label1:
   PRINT 'AL >= 5.'
exit:
   RET
```

C	Z	S	O	P	A
unchanged					

JNBE	label	Short Jump if first operand is Not Below and Not Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		`if (CF = 0) and (ZF = 0) then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 7`
		`CMP AL, 5`
		`JNBE label1`
		`PRINT 'AL <= 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL > 5.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

JNC	label	Short Jump if Carry flag is set to 0.
		
		Algorithm:
		
		`if CF = 0 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 2`
		`ADD AL, 3`
		`JNC label1`
		`PRINT 'has carry.'`
		`JMP exit`
		`label1:`
		`PRINT 'no carry.'`
		`exit:`

		RET  

JNE	label	Short Jump if first operand is Not Equal to second operand (as set by CMP instruction).
		Signed/Unsigned.
		
		Algorithm:
		
		if ZF = 0 then jump
		
		Example:
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 3
		JNE label1
		PRINT 'AL = 3.'
		JMP exit
		label1:
		PRINT 'Al <> 3.'
		exit:
		RET
		
		 
		
		 

JNG	label	Short Jump if first operand is Not Greater then second operand (as set by CMP instruction).
		Signed.
		
		Algorithm:
		
		if (ZF = 1) and (SF <> OF) then jump
		
		Example:
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 3
		JNG label1
		PRINT 'AL > 3.'
		JMP exit
		label1:
		PRINT 'Al <= 3.'
		exit:
		RET

		C Z S O P A unchanged 

JNGE	label	Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		`if SF <> OF then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, 3`
		`JNGE label1`
		`PRINT 'AL >= 3.'`
		`JMP exit`
		`label1:`
		`PRINT 'Al < 3.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		 

JNL	label	Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		`if SF = OF then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, -3`
		`JNL label1`
		`PRINT 'AL < -3.'`
		`JMP exit`
		`label1:`
		`PRINT 'Al >= -3.'`
		`exit:`
		`RET`
		
		C Z S O P A

		unchanged 

JNLE	label	Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		`if (SF = OF) and (ZF = 0) then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, -3`
		`JNLE label1`
		`PRINT 'AL <= -3.'`
		`JMP exit`
		`label1:`
		`PRINT 'Al > -3.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged 

JNO	label	Short Jump if Not Overflow.
		
		Algorithm:
		
		`if OF = 0 then jump`
		
		Example:
		
		`; -5 - 2 = -7 (inside -128..127)`
		`; the result of SUB is correct,`
		`; so OF = 0:`
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, -5`
		`SUB AL, 2 ; AL = 0F9h (-7)`
		`JNO label1`
		`PRINT 'overflow!'`
		`JMP exit`
		`label1:`
		`PRINT 'no overflow.'`
		`exit:`
		`RET`
		
		C Z S O P A

		unchanged 

JNP	label	Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		`if PF = 0 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 00000111b ; AL = 7`
		`OR AL, 0 ; just set flags.`
		`JNP label1`
		`PRINT 'parity even.'`
		`JMP exit`
		`label1:`
		`PRINT 'parity odd.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged 

JNS	label	Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		`if SF = 0 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 00000111b ; AL = 7`
		`OR AL, 0 ; just set flags.`
		`JNS label1`
		`PRINT 'signed.'`
		`JMP exit`
		`label1:`
		`PRINT 'not signed.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged

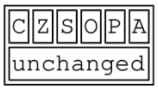
JNZ	label	Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		`if ZF = 0 then jump`
		
		Example:
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV AL, 00000111b ; AL = 7`
		`OR AL, 0 ; just set flags.`
		`JNZ label1`
		`PRINT 'zero.'`
		`JMP exit`
		`label1:`
		`PRINT 'not zero.'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

JO	label	Short Jump if Overflow.
		
		Algorithm:
		
		`if OF = 1 then jump`
		
		Example:
		
		`; -5 - 127 = -132 (not in -128..127)`
		`; the result of SUB is wrong (124),`
		`; so OF = 1 is set:`
		
		`include 'emu8086.inc'`
		
		`org 100h`
		`MOV AL, -5`
		`SUB AL, 127 ; AL = 7Ch (124)`
		`JO label1`
		`PRINT 'no overflow.'`
		`JMP exit`
		`label1:`
		`PRINT 'overflow!'`
		`exit:`
		`RET`
		
		C Z S O P A
		unchanged
		
		 

<table>
<tr>
<td>JP</td>
<td>label</td>
<td>Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

Algorithm:

<pre>   if PF = 1 then jump</pre>
Example:
<pre>   include 'emu8086.inc'

   ORG 100h
   MOV AL, 00000101b   ; AL = 5
   OR AL, 0   ; just set flags.
   JP label1
   PRINT 'parity odd.'
   JMP exit
label1:
   PRINT 'parity even.'
exit:
   RET</pre>


 </td>
</tr>
<tr>
<td>JPE</td>
<td>label</td>
<td>Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

Algorithm:

<pre>   if PF = 1 then jump</pre>
Example:
<pre>   include 'emu8086.inc'

   ORG 100h
   MOV AL, 00000101b   ; AL = 5
   OR AL, 0   ; just set flags.
   JPE label1
   PRINT 'parity odd.'
   JMP exit
label1:
   PRINT 'parity even.'
exit:
   RET</pre>

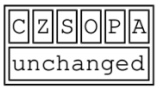

 </td>
</tr>
<tr>
<td></td>
<td></td>
<td>Short Jump if Parity Odd. Only 8 low bits of result</td>
</tr>
</table>

<table>
<tr>
<td>JPO</td>
<td>label</td>
<td>
are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

Algorithm:

<pre>   if PF = 0 then jump</pre>
Example:

<pre>   include 'emu8086.inc'

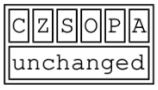
   ORG 100h
   MOV AL, 00000111b   ; AL = 7
   OR AL, 0   ; just set flags.
   JPO label1
   PRINT 'parity even.'
   JMP exit
label1:
   PRINT 'parity odd.'
exit:
   RET</pre>



</td>
</tr>
<tr>
<td>JS</td>
<td>label</td>
<td>
Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

Algorithm:

<pre>   if SF = 1 then jump</pre>
Example:

<pre>   include 'emu8086.inc'

   ORG 100h
   MOV AL, 10000000b   ; AL = -128
   OR AL, 0   ; just set flags.
   JS label1
   PRINT 'not signed.'
   JMP exit
label1:
   PRINT 'signed.'
exit:
   RET</pre>



</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Short Jump if Zero (equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</td>
</tr>
</table>

JZ	label	Algorithm:
		`if ZF = 1 then jump`
		Example:
		`include 'emu8086.inc'`
		`ORG 100h`
		`MOV AL, 5`
		`CMP AL, 5`
		`JZ label1`
		`PRINT 'AL is not equal to 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL is equal to 5.'`
		`exit:`
		`RET`
		C Z S O P A
		unchanged

LAHF	No operands	Load AH from 8 low bits of Flags register.
		Algorithm:
		`AH = flags register`
		`AH bit: 7 6 5 4 3 2 1 0`
		`[SF] [ZF] [0] [AF] [0] [PF] [1] [CF]`
		`bits 1, 3, 5 are reserved.`
		C Z S O P A
		unchanged

		Load memory double word into word register and DS. Algorithm: • `REG = first word` • `DS = second word` Example: `ORG 100h`

<table>
<tr>
<td>LDS</td>
<td>REG, memory</td>
<td>

```
LDS AX, m

RET

m  DW  1234h
   DW  5678h

END
```

AX is set to 1234h, DS is set to 5678h.

C	Z	S	O	P	A
unchanged					


</td>
</tr>
<tr>
<td>LEA</td>
<td>REG, memory</td>
<td>

Load Effective Address.

Algorithm:

- REG = address of memory (offset)

Example:

```
MOV BX, 35h
MOV DI, 12h
LEA SI, [BX+DI]     ; SI = 35h + 12h = 47h
```

Note: The integrated 8086 assembler automatically replaces **LEA** with a more efficient **MOV** where possible. For example:

```
org 100h
LEA AX, m        ; AX = offset of m
RET
m  dw  1234h
END
```

C	Z	S	O	P	A
unchanged					


</td>
</tr>
<tr>
<td></td>
<td></td>
<td>

Load memory double word into word register and ES.

Algorithm:
</td>
</tr>
</table>

<table>
<tr>
<td>LES</td>
<td>REG, memory</td>
<td>

REG = first word
ES = second word

Example:
<pre>
ORG 100h

LES AX, m

RET

m DW 1234h
   DW 5678h

END
</pre>
AX is set to 1234h, ES is set to 5678h.
<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>

</td>
</tr>
<tr>
<td>LODSB</td>
<td>No operands</td>
<td>
Load byte at DS:[SI] into AL. Update SI.
Algorithm:

AL = DS:[SI]
if DF = 0 then

SI = SI + 1

else

SI = SI - 1

Example:
<pre>
ORG 100h

LEA SI, a1
MOV CX, 5
MOV AH, 0Eh

m: LODSB
INT 10h
LOOP m

RET

a1 DB 'H', 'e', 'l', 'l', 'o'
</pre>
<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
</table>

LODSW

No operands

Load word at DS:[SI] into AX. Update SI.

Algorithm:

- AX = DS:[SI]
- if DF = 0 then
  - SI = SI + 2

  else
  - SI = SI - 2

Example:

```
ORG 100h

LEA SI, a1
MOV CX, 5

REP LODSW   ; finally there will be 555h in AX.

RET

a1 dw 111h, 222h, 333h, 444h, 555h
```

C	Z	S	O	P	A
unchanged					

LOOP

label

Decrease CX, jump to label if CX not zero.

Algorithm:

- CX = CX - 1
- if CX <> 0 then
  - jump

  else
  - no jump, continue

Example:

```
   include 'emu8086.inc'

   ORG 100h
   MOV CX, 5
label1:
   PRINTN 'loop!'
   LOOP label1
   RET
```

C	Z	S	O	P	A
unchanged					

LOOPE label

Decrease CX, jump to label if CX not zero and Equal (ZF = 1).

Algorithm:

- CX = CX - 1
- if (CX <> 0) and (ZF = 1) then
  - jump
  else
  - no jump, continue

Example:

```
; Loop until result fits into AL alone,
; or 5 times. The result will be over 255
; on third loop (100+100+100),
; so loop will exit.

   include 'emu8086.inc'

   ORG 100h
   MOV AX, 0
   MOV CX, 5
label1:
   PUTC '*'
   ADD AX, 100
   CMP AH, 0
   LOOPE label1
   RET
```

C	Z	S	O	P	A
unchanged					

LOOPNE label

Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0).

Algorithm:

- CX = CX - 1
- if (CX <> 0) and (ZF = 0) then
  - jump
  else
  - no jump, continue

Example:

```
; Loop until '7' is found,
; or 5 times.

   include 'emu8086.inc'

   ORG 100h
```

```
   MOV SI, 0
   MOV CX, 5
label1:
   PUTC '*'
   MOV AL, v1[SI]
   INC SI          ; next byte (SI=SI+1).
   CMP AL, 7
   LOOPNE label1
   RET
   v1 db 9, 8, 7, 6, 5
```

C	Z	S	O	P	A
unchanged					

LOOPNZ	label	Decrease CX, jump to label if CX not zero and ZF = 0.
		
		Algorithm:
		
		• CX = CX - 1
		• if (CX <> 0) and (ZF = 0) then
		  ○ jump
		  else
		  ○ no jump, continue
		
		Example:
		
		`; Loop until '7' is found,`
		`; or 5 times.`
		
		`include 'emu8086.inc'`
		
		`ORG 100h`
		`MOV SI, 0`
		`MOV CX, 5`
		`label1:`
		`PUTC '*'`
		`MOV AL, v1[SI]`
		`INC SI ; next byte (SI=SI+1).`
		`CMP AL, 7`
		`LOOPNZ label1`
		`RET`
		`v1 db 9, 8, 7, 6, 5`
		
		C Z S O P A
		unchanged
		
		 

		Decrease CX, jump to label if CX not zero and ZF = 1. Algorithm: • CX = CX - 1

<table>
<tr>
<td>LOOPZ</td>
<td>label</td>
<td>

if (CX <> 0) and (ZF = 1) then

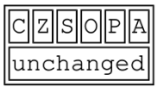
jump

else

no jump, continue

Example:
<pre>
; Loop until result fits into AL alone,
; or 5 times. The result will be over 255
; on third loop (100+100+100),
; so loop will exit.

   include 'emu8086.inc'

   ORG 100h
   MOV AX, 0
   MOV CX, 5
label1:
   PUTC '*'
   ADD AX, 100
   CMP AH, 0
   LOOPZ label1
   RET
</pre>



</td>
</tr>
<tr>
<td>MOV</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

SREG, memory
memory, SREG
REG, SREG
SREG, REG</td>
<td>
Copy operand2 to operand1.

The MOV instruction cannot:

set the value of the CS and IP registers.
copy value of one segment register to another segment register (should copy to general register first).
copy immediate value to segment register (should copy to general register first).

Algorithm:
<pre>
    operand1 = operand2
</pre>
Example:
<pre>
ORG 100h
MOV AX, 0B800h   ; set AX = B800h (VGA memory).
MOV DS, AX   ; copy value of AX to DS.
MOV CL, 'A'   ; CL = 41h (ASCII code).
MOV CH, 01011111b ; CL = color attribute.
MOV BX, 15Eh   ; BX = position on screen.
MOV [BX], CX   ; w.[0B800h:015Eh] = CX.
RET   ; returns to operating system.
</pre>

</td>
</tr>
</table>

		C Z S O P A unchanged 

MOVSB	No operands	Copy byte at DS:[SI] to ES:[DI]. Update SI and DI.
		
		Algorithm:
		
		• ES:[DI] = DS:[SI]
		• if DF = 0 then
		  ○ SI = SI + 1
		  ○ DI = DI + 1
		else
		  ○ SI = SI - 1
		  ○ DI = DI - 1
		
		Example:
		
		`ORG 100h`
		
		`CLD`
		`LEA SI, a1`
		`LEA DI, a2`
		`MOV CX, 5`
		`REP MOVSB`
		
		`RET`
		
		`a1 DB 1,2,3,4,5`
		`a2 DB 5 DUP(0)`
		
		C Z S O P A
		unchanged
		 

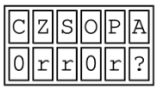
		Copy **word** at DS:[SI] to ES:[DI]. Update SI and DI.
		
		Algorithm:
		
		• ES:[DI] = DS:[SI]
		• if DF = 0 then
		  ○ SI = SI + 2
		  ○ DI = DI + 2
		else
		  ○ SI = SI - 2
		  ○ DI = DI - 2
		
		Example:

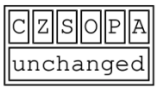
MOVSW	No operands	ORG 100h CLD LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSW RET a1 DW 1,2,3,4,5 a2 DW 5 DUP(0) C Z S O P A unchanged 

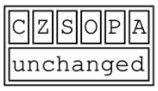
MUL	REG
	memory
		Unsigned multiply.
		
		Algorithm:
		
		when operand is a **byte**:
		AX = AL * operand.
		
		when operand is a **word**:
		(DX AX) = AX * operand.
		
		Example:
		
		MOV AL, 200 ; AL = 0C8h
		MOV BL, 4
		MUL BL ; AX = 0320h (800)
		RET
		
		C Z S O P A
		r ? ? r ? ?
		CF=OF=0 when high section of the result is zero.
		
		 

NEG	REG memory	Negate. Makes operand negative (two's complement). Algorithm: • Invert all bits of the operand • Add 1 to inverted operand Example: MOV AL, 5 ; AL = 05h NEG AL ; AL = 0FBh (-5) NEG AL ; AL = 05h (5)

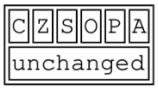
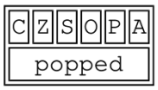
		RET C Z S O P A r r r r r r 
NOP	No operands	No Operation. Algorithm: • Do nothing Example: ; do nothing, 3 times: NOP NOP NOP RET C Z S O P A unchanged 
NOT	REG memory	Invert each bit of the operand. Algorithm: • if bit is 1 turn it to 0. • if bit is 0 turn it to 1. Example: MOV AL, 00011011b NOT AL ; AL = 11100100b RET C Z S O P A unchanged 
	REG, memory	Logical OR between all bits of two operands. Result is stored in first operand. These rules apply: 1 OR 1 = 1 1 OR 0 = 1 0 OR 1 = 1 0 OR 0 = 0

OR	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		Example:
		`MOV AL, 'A'  ; AL = 01000001b`
		`OR AL, 00100000b  ; AL = 01100001b  ('a')`
		`RET`
		 
		
		 

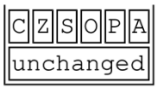
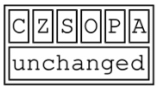
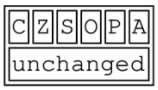
OUT	im.byte, AL
	im.byte, AX
	DX, AL
	DX, AX
		Output from **AL** or **AX** to port.
		First operand is a port number. If required to access port number over 255 - **DX** register should be used.
		Example:
		`MOV AX, 0FFFh ; Turn on all`
		`OUT 4, AX  ; traffic lights.`
		`MOV AL, 100b  ; Turn on the third`
		`OUT 7, AL  ; magnet of the stepper-motor.`
		 
		
		 

POP	REG
	SREG
	memory
		Get 16 bit value from the stack.
		Algorithm:
		• operand = SS:[SP] (top of the stack)
		• SP = SP + 2
		Example:
		`MOV AX, 1234h`
		`PUSH AX`
		`POP  DX  ; DX = 1234h`
		`RET`
		 
		
		 

		Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack.

POPA	No operands	SP value is ignored, it is Popped but not set to SP register). Note: this instruction works only on **80186** CPU and later! Algorithm: • POP DI • POP SI • POP BP • POP xx (SP value ignored) • POP BX • POP DX • POP CX • POP AX  
POPF	No operands	Get flags register from the stack. Algorithm: • flags = SS:[SP] (top of the stack) • SP = SP + 2  

PUSH	REG
	SREG
	memory
	immediate
		Store 16 bit value in the stack.
		
		Note: **PUSH immediate** works only on 80186 CPU and later!
		
		Algorithm:
		
		• SP = SP - 2
		• SS:[SP] (top of the stack) = operand
		
		Example:
		
		`MOV AX, 1234h`
		`PUSH AX`
		`POP DX ; DX = 1234h`
		`RET`

		
PUSHA	No operands	Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used. Note: this instruction works only on **80186** CPU and later! Algorithm: • PUSH AX • PUSH CX • PUSH DX • PUSH BX • PUSH SP • PUSH BP • PUSH SI • PUSH DI 
PUSHF	No operands	Store flags register in the stack. Algorithm: • SP = SP - 2 • SS:[SP] (top of the stack) = flags 
		Rotate operand1 left through Carry Flag. The number of rotates is set by operand2. When **immediate** is greater then 1, assembler generates several **RCL xx, 1** instructions because 8086 has machine code only for this instruction (the same principle works for all other shift/rotate instructions). Algorithm:

RCL	memory, immediate
	REG, immediate
	
	memory, CL
	REG, CL
		shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position.
		
		Example:
		
		`STC ; set carry (CF=1).`
		`MOV AL, 1Ch ; AL = 00011100b`
		`RCL AL, 1 ; AL = 00111001b, CF=0.`
		`RET`
		
		 
		
		OF=0 if first operand keeps original sign.
		 

RCR	memory, immediate
	REG, immediate
	
	memory, CL
	REG, CL
		Rotate operand1 right through Carry Flag. The number of rotates is set by operand2.
		
		Algorithm:
		
		shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position.
		
		Example:
		
		`STC ; set carry (CF=1).`
		`MOV AL, 1Ch ; AL = 00011100b`
		`RCR AL, 1 ; AL = 10001110b, CF=0.`
		`RET`
		
		 
		
		OF=0 if first operand keeps original sign.
		 

		Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.
		
		Algorithm:
		
		`check_cx:`
		
		`if CX <> 0 then`
		
		• `do following chain instruction`

REP	chain instruction	• CX = CX - 1 • go back to check_cx else • exit from REP cycle 

REPE	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times.
		
		Algorithm:
		
		check_cx:
		
		if CX <> 0 then
		
		• do following chain instruction
		• CX = CX - 1
		• if ZF = 1 then:
		  ○ go back to check_cx
		else
		  ○ exit from REPE cycle
		
		else
		
		• exit from REPE cycle
		
		example:
		open **cmpsb.asm** from c:\emu8086\examples
		
		 

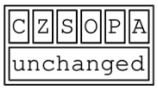
		Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction

REPNE	chain instruction	• CX = CX - 1
		• if ZF = 0 then:
		  ○ go back to check_cx
		else
		  ○ exit from REPNE cycle
		
		else
		
		• exit from REPNE cycle
		
		 

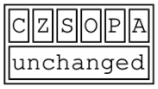
REPNZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times.
		
		Algorithm:
		
		check_cx:
		
		if CX <> 0 then
		
		• do following chain instruction
		• CX = CX - 1
		• if ZF = 0 then:
		  ○ go back to check_cx
		else
		  ○ exit from REPNZ cycle
		
		else
		
		• exit from REPNZ cycle
		
		 

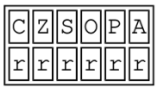
		Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Zero), maximum CX times. Algorithm: check_cx: if CX <> 0 then

REPZ	chain instruction	• do following chain instruction
		• CX = CX - 1
		• if ZF = 1 then:
		  ○ go back to check_cx
		else
		  ○ exit from REPZ cycle
		
		else
		
		• exit from REPZ cycle
		
		 
		
		 

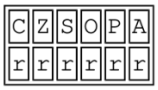
RET	No operands
	or even immediate
		Return from near procedure.
		
		Algorithm:
		
		• Pop from stack:
		  ○ IP
		• if immediate operand is present:
		  SP = SP + operand
		
		Example:
		
		`ORG 100h ; for COM file.`
		
		`CALL p1`
		
		`ADD AX, 1`
		
		`RET ; return to OS.`
		
		`p1 PROC ; procedure declaration.`
		`MOV AX, 1234h`
		`RET ; return to caller.`
		`p1 ENDP`
		
		 
		
		 

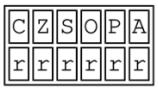
RETF	No operands
	or even immediate
		Return from Far procedure.
		
		Algorithm:
		
		• Pop from stack:
		  ○ IP
		  ○ CS
		• if immediate operand is present:

		SP = SP + operand  
ROL	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 left. The number of rotates is set by operand2. Algorithm: shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position. Example: MOV AL, 1Ch ; AL = 00011100b ROL AL, 1 ; AL = 00111000b, CF=0. RET  OF=0 if first operand keeps original sign. 
ROR	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 right. The number of rotates is set by operand2. Algorithm: shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position. Example: MOV AL, 1Ch ; AL = 00011100b ROR AL, 1 ; AL = 00001110b, CF=0. RET  OF=0 if first operand keeps original sign. 
		Store AH register into low 8 bits of Flags register. Algorithm:

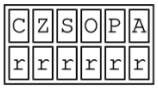
SAHF	No operands	flags register = AH AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved.  
SAL	memory, immediate REG, immediate memory, CL REG, CL	Shift Arithmetic operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 0E0h ; AL = 11100000b SAL AL, 1 ; AL = 11000000b, CF=1. RET  OF=0 if first operand keeps original sign. 

SAR	memory, immediate
	REG, immediate
	
	memory, CL
	REG, CL
		Shift Arithmetic operand1 Right. The number of shifts is set by operand2.
		
		Algorithm:
		
		• Shift all bits right, the bit that goes off is set to CF.
		• The sign bit that is inserted to the left-most position has the same value as before shift.
		
		Example:
		
		MOV AL, 0E0h ; AL = 11100000b
		SAR AL, 1 ; AL = 11110000b, CF=0.
		
		MOV BL, 4Ch ; BL = 01001100b
		SAR BL, 1 ; BL = 00100110b, CF=0.
		
		RET

		 OF=0 if first operand keeps original sign.
SBB	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Subtract with Borrow. Algorithm: operand1 = operand1 - operand2 - CF Example: STC MOV AL, 5 SBB AL, 3 ; AL = 5 - 3 - 1 = 1 RET 

SCASB	No operands	Compare bytes: AL from ES:[DI].
		Algorithm:
		• AL - ES:[DI]
		• set flags according to result: OF, SF, ZF, AF, PF, CF
		• if DF = 0 then
		  ○ DI = DI + 1
		else
		  ○ DI = DI - 1
		 

SCASW	No operands	Compare words: AX from ES:[DI]. Algorithm: • AX - ES:[DI] • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ DI = DI + 2

		else o DI = DI - 2  
SHL	memory, immediate REG, immediate memory, CL REG, CL	Shift operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 11100000b SHL AL, 1 ; AL = 11000000b, CF=1. RET  OF=0 if first operand keeps original sign. 
SHR	memory, immediate REG, immediate memory, CL REG, CL	Shift operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • Zero bit is inserted to the left-most position. Example: MOV AL, 00000111b SHR AL, 1 ; AL = 00000011b, CF=1. RET  OF=0 if first operand keeps original sign. 

STC	No operands	Set Carry flag. Algorithm: `CF = 1` 
STD	No operands	Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: `DF = 1` 
STI	No operands	Set Interrupt enable flag. This enables hardware interrupts. Algorithm: `IF = 1` 

STOSB	No operands	Store byte in AL into ES:[DI]. Update DI.
		
		Algorithm:
		
		• `ES:[DI] = AL`
		• `if DF = 0 then`
		  ○ `DI = DI + 1`
		`else`
		  ○ `DI = DI - 1`
		
		Example:
		
		`ORG 100h`
		
		`LEA DI, a1`

		MOV AL, 12h
		MOV CX, 5
		
		REP STOSB
		
		RET
		
		a1 DB 5 dup(0)
		
		C Z S O P A
		unchanged
		
		 

STOSW	No operands	Store word in AX into ES:[DI]. Update DI.
		
		Algorithm:
		
		• ES:[DI] = AX
		• if DF = 0 then
		  ○ DI = DI + 2
		else
		  ○ DI = DI - 2
		
		Example:
		
		ORG 100h
		
		LEA DI, a1
		MOV AX, 1234h
		MOV CX, 5
		
		REP STOSW
		
		RET
		
		a1 DW 5 dup(0)
		
		C Z S O P A
		unchanged
		
		 

SUB	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		Subtract.
		
		Algorithm:
		
		operand1 = operand1 - operand2
		
		Example:
		
		MOV AL, 5
		SUB AL, 1 ; AL = 4
		
		RET

		C Z S O P A r r r r r r 
TEST	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical AND between all bits of two operands for flags only. These flags are effected: **ZF, SF, PF.** Result is not stored anywhere. These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Example: MOV AL, 00000101b TEST AL, 1 ; ZF = 0. TEST AL, 10b ; ZF = 1. RET C Z S O P 0 r r 0 r 

XCHG	REG, memory
	memory, REG
	REG, REG
		Exchange values of two operands.
		
		Algorithm:
		
		operand1 < - > operand2
		
		Example:
		
		MOV AL, 5
		MOV AH, 2
		XCHG AL, AH ; AL = 2, AH = 5
		XCHG AL, AH ; AL = 5, AH = 2
		RET
		
		C Z S O P A
		unchanged
		 

		Translate byte from table. Copy value of memory byte at DS:[BX + unsigned AL] to AL register. Algorithm:

XLATB	No operands	AL = DS:[BX + unsigned AL] Example: ORG 100h LEA BX, dat MOV AL, 2 XLATB ; AL = 33h RET dat DB 11h, 22h, 33h, 44h, 55h C Z S O P A unchanged
XOR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand. These rules apply: 1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0 Example: MOV AL, 00000111b XOR AL, 00000010b ; AL = 00000101b RET C Z S O P A 0 r r 0 r ?

copyright © 2005 emu8086.com
all rights reserved.

Labs

Lab 2

1. Write an Assembly Language Program (ALP) to read two decimal digits both of which are less than 5 and display their summation in the next line.

CODE :

```
;SMALL MODEL HAS BEEN USED
.MODEL SMALL
;STACK HAS BEEN USED OF SIZE 100h
.STACK 100H


.DATA   ;ALL VARIABLE HAS BEEN DECLARED HERE

MESSAGE_1  DB  'ENTER THE FIRST DIGIT : $'
MESSAGE_2  DB  'ENTER THE SECOND DIGIT : $'
MESSAGE_3  DB  'SUMMATION RESULT : $'

.CODE                ;CODE SEGMENT SECTION

  MAIN PROC          ;MAIN SECTION LIKE INT MAIN() IN C

   ;STATEMENT SECTION
   ;INSTRUCTION DESTINATION, SOURCE <- COMMAND MECHANISM


    MOV AX, @DATA       ; IMPORT ALL DATA
    MOV DS, AX

    LEA DX, MESSAGE_1   ;DISPLAY THE MESSAGE_1
    MOV AH, 9           ;CALL THE FUNCTION
    INT 21H


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


    MOV BL, AL          ; SAVE FIRST DIGIT IN BL
    SUB BL, 30H
```

```
    ;NEW LINE
    MOV AH , 2
    MOV DL , 0AH
    INT 21H
    MOV DL , 0DH
    INT 21H

    LEA DX, MESSAGE_2   ;DISPLAY THE MESSAGE_2
    MOV AH, 9           ;CALL THE FUNCTION
    INT 21H


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


    MOV BH, AL          ; SAVE SECOND DIGIT IN BL
    SUB BH, 30H


    ;NEW LINE
    MOV AH , 2
    MOV DL , 0AH
    INT 21H
    MOV DL , 0DH
    INT 21H


    LEA DX, MESSAGE_3   ;DISPLAY THE MESSAGE_2
    MOV AH, 9           ;CALL THE FUNCTION
    INT 21H

    ; SUMMATION OF TWO INPUT DIGITS

    ADD BL, BH          ;ADDITION -> BL = BL + BH
    ADD BL, 30H

    ;DISPLAY THE RESULT

    MOV AH, 2
    MOV DL, BL
    INT 21H

    ; RETURN CONTROL TO OS / RETURNS ALL THE ALLOCATED SPACE TO THE SYSTEM
    MOV AH, 4CH
    INT 21H
  MAIN ENDP

;EXIT FROM THE PROGRAM
```

```
END MAIN
```

2. Write an Assembly Language Program (ALP) to translate the following high level language assignment statements into assembly language.
*A= B - 2*A +1*
Where the A, and B are byte variables. Your program does the following,
i. Keep the value of variable A as undefined and set the value of variable B to 10D
ii. Display '?'
iii. Read the value of variable A in new line
*iv. Display the updated value of A= B - 2*A -1 in new line*

CODE :

```
.MODEL SMALL ;SMALL MODEL HAS BEEN USED

.STACK 100h  ;STACK HAS BEEN USED OF SIZE 100h
.DATA        ;ALL VARIABLE HAS BEEN DECLARED HERE

B DB 10D     ;SET VALUE OF VARIABLE B TO 10D
A DB ?       ;SET VALUE OF VARIABLE A AS UNDEFINED
.CODE        ;CODE SEGMENT SECTION

MAIN PROC    ;MAIN SECTION

   ;STATEMENT SECTION
   ;INSTRUCTION DESTINATION, SOURCE <- COMMAND MECHANISM

   MOV AX, @DATA  ;IMPORT THE DATA
   MOV DS, AX

   MOV AH,2       ;DISPLAYED '?'
   MOV DL,'?'
   INT 21h



   ;NEW LINE
   MOV AH , 2
   MOV DL , 0AH
   INT 21H
   MOV DL , 0DH
   INT 21H


   MOV AH,1      ;READ THE VALUE OF A IN NEW LINE
```

```
    INT 21h
    MOV A, AL
    SUB A,30H


    MOV BL, 2     ;SAVED THE VALE 2 IN REGISTER BL



    MOV AL, BL    ;MOVED VALUE OF 2 IN REGISTER AL
    MUL A         ; MULTIPLICATION -> AL = BL * A
    ADD AL, 30H


    MOV BL, B     ;MOVED VALUE OF B IN REGISTER BL
    SUB BL, AL    ;SUBTRACTION -> BL = BL - AL
    ADD BL, 30H

    SUB BL, 1     ;SUBTRACTION = BL = BL - 1
    ADD BL, 30H

    ;NEW LINE
    MOV AH , 2
    MOV DL , 0AH
    INT 21H
    MOV DL , 0DH
    INT 21H



    MOV AH, 2      ;DISPLAYED THE OUTPUT
    MOV DL,BL
    INT 21H



    MOV AH , 4CH   ; RETURN CONTROL TO OS / RETURNS ALL THE ALLOCATED SPACE To SYSTEM
    INT 21H
    MAIN ENDP
END MAIN ; RETURN 0
```

3. Write an Assembly Language Program (ALP) to read one of the hex digits uppercase A-F and display it on the next line in decimal.

CODE :

```
.MODEL SMALL        ;SMALL MODEL HAS BEEN USED
.STACK 100H         ;STACK HAS BEEN USED OF SIZE 100H
```

```
.DATA             ;ALL VARIABLE HAS BEEN DECLARED HERE
MESSAGE_1 DB 'Enter a hex digit: $'
MESSAGE_2 DB 'In decimal it is: $'

.CODE             ;CODE SEGMENT SECTION
MAIN PROC         ;MAIN SECTION

   ;STATEMENT SECTION
   ;INSTRUCTION DESTINATION, SOURCE <- COMMAND MECHANISM

   MOV AX, @DATA   ;IMPORT ALL DATA
   MOV DS, AX
               ;DISPLAY THE MESSAGE_1
   MOV AH, 9
   LEA DX, MESSAGE_1
   INT 21H

   MOV AH, 1      ;HEX DIGIT INPUT
   INT 21H
   MOV BH, AL
   SUB BH, 17D    ;EQUIVALENT DECIMAL VALUE BY SUBTRACTING 17D.


   ;NEW LINE
   MOV AH , 2
   MOV DL , 10
   INT 21H
   MOV DL , 13
   INT 21H

               ;DISPLAY THE MESSAGE_2
   MOV AH, 9
   LEA DX, MESSAGE_2
   INT 21H

   MOV DL, 49D     ;DISPLAY ASCII REPRESENTATION OF THE DECIMAL DIGIT "1"
   MOV AH, 2
   INT 21H

   MOV DL, BH     ;DISPLAYED CONVERTED DECIMAL DIGIT
   MOV AH, 2
   INT 21H

   MOV AH , 4CH    ; RETURN CONTROL TO OS / RETURNS ALL THE ALLOCATED SPACE TO  SYSTEM
   INT 21H
   MAIN ENDP
END MAIN
```

4. Write an ALP to

v. Display '?'

vi. Read three initials

*vii. Display them in the middle of an 7*7 box of asterisk*

viii. Beep the computer

CODE :

```
.MODEL SMALL              ;SMALL MODEL HAS BEEN USED

.STACK 100H               ;STACK HAS BEEN USED OF SIZE 100H
                    ;ALL VARIABLE HAS BEEN DECLARED HERE
.DATA

A DB '*******',10,13,'$'   ;STRING OF 7 * ASTERIK SYMBOL IS DECLARED (WITH NEW LINE)
B DB '**$'                ;STRING OF 2 * ASTERIK SYMBOL IS DECLARED
.CODE                     ;CODE SEGMENT SECTION

MAIN PROC                 ;MAIN SECTION

   ;STATEMENT SECTION
   ;INSTRUCTION DESTINATION, SOURCE <- COMMAND MECHANISM

   MOV AX, @DATA          ; IMPORT ALL DATA
   MOV DS, AX

   MOV AH, 2              ;DISPLAYED SYMBOL '?'
   MOV DL, '?'
   INT 21H


   MOV AH, 1              ; 'a' INPUT TAKEN
   INT 21H
   MOV BL, AL             ;SAVED IN REGISTER BL


   MOV AH, 1              ; 'b' INPUT TAKEN
   INT 21H
   MOV BH, AL             ;SAVED IN REGISTER BH


   MOV AH, 1              ; 'c' INPUT TAKEN
   INT 21H
   MOV CL, AL             ;SAVED IN REGISTER CL
```

```
;NEW LINE
MOV AH, 2
MOV DL, 10
INT 21H
MOV DL, 13
INT 21H

LEA DX, A          ;DISPLAYED 7 * ASTERIK SYMBOL
MOV AH, 9
INT 21H            ;INTERUPT FOR 1ST ROW
INT 21H            ;INTERUPT FOR 2ND ROW
INT 21H            ;INTERUPT FOR 3RD ROW

MOV AH, 9          ;DISPLAYED 2 * ASTERIK SYMBOL FOR 4TH ROW
LEA DX, B
INT 21H

;DISPLAYED 'abc; IN THE MIDDLE SECTION OF 4TH ROW OF 7*7 ASTERIK BOX

MOV AH, 2          ;DISPLAYED 'a'
MOV DL, BL
INT 21H

MOV AH, 2          ;DISPLAYED 'b'
MOV DL, BH
INT 21H

MOV AH, 2          ;DISPLAYED 'c'
MOV DL, CL
INT 21H

MOV AH, 9          ;DISPLAYED LAST 2 * ASTERIK SYMBOL FOR 4TH ROW
LEA DX, B
INT 21H

;NEW LINE
MOV AH, 2
MOV DL, 10
INT 21H
MOV DL, 13
INT 21H


LEA DX, A          ;DISPLAYED 7 * ASTERIK SYMBOL
MOV AH, 9
INT 21H            ;INTERUPT FOR 5TH ROW
INT 21H            ;INTERUPT FOR 6TH ROW
INT 21H            ;INTERUPT FOR 7TH ROW
```

```
   MOV AH, 2
   MOV DL, 07H
   INT 21H              ;INTERUPT TO BEEP THE COMPUTER

   MOV AH , 4CH
   INT 21H
   MAIN ENDP
END MAIN ; RETURN 0
```

Lab 03

1. Write an assembly language program section which puts the product of the first 6 terms of the arithmetic sequence 1, 3, 5, 7 … in DX. Use a looping structure to produce the product.

CODE :

```
.MODEL SMALL
.STACK 100H
.DATA
.CODE
   MAIN PROC

   MOV CX,6    ;THE LOOP WILL ITERATE 6 TIMES
   MOV AX,1    ;ASSIGNED THE INITIAL VALUE OF AX TO 1
   MOV BX,1    ;ASSIGNED THE INITIAL VALUE OF BX TO1

   JCXZ SKIP   ;JUMP IF CX IS ZERO

   TOP:
   MUL BX      ;MULTIPLYING BX WITH AX
   ADD BX,2    ;INCREMENTING BX BY 2
   LOOP TOP    ;DECREMENT COUNTER AND AGAIN GO TO TOP TILL COUNTER BECOMES 0

   SKIP:
   MOV DX,AX   ;MOVING THE RESULT OF AX INTO DX

   MOV AH,4CH  ;EXIT FUNCTION
   INT 21H     ;TRANSFERING THE CONTROL TO OS

   MAIN ENDP   ;END MAIN PROCEDURE
END MAIN       ;RETURN 0
```

2. Write a. program that prompts the user to enter two unsigned hex numbers, 0 to FFFFh, and prints their sum in hex on the next line. Each input ends with a carriage return. Define two procedures called 'INHEX' and 'OUTHEX' respectively for handling hexadecimal input and hexadecimal output.
Sample execution:

Type a HEX Number, 0 - FFEF: 21AB
Type a HEX Number, 0 - FFFF: 5E03
The SUM is: 7FAE

CODE :

```
.MODEL SMALL
.STACK 100H
.DATA
   STR1 DB "Type a HEX Number, 0 - FFEF:$"
   STR2 DB "The SUM is:$"
   INP1 DW ?

.CODE

   MAIN PROC

      MOV AX,@DATA  ;MOVING THE DATA SEGMENT INTO AX
      MOV DS,AX     ;INITIALIZE DS

      CALL INHEX    ;TAKING THE FIRST INPUT

      MOV INP1,BX   ;STORING THE VALUE TO INP1 VARIABLE

      ;NEW LINE

      MOV AH,2      ;DISPLAY FUNCTION
      MOV DL,0AH    ;LINE FEED
      INT 21H       ;GO TO BEGINNING OF THE LINE
      MOV DL,0DH    ;CARRIAGE RETURN
      INT 21H       ;GO TO NEW LINE

      CALL INHEX    ;TAKING THE SECOND INPUT

      ;NEW LINE

      MOV AH,2      ;DISPLAY FUNCTION
      MOV DL,0AH    ;LINE FEED
      INT 21H       ;GO TO BEGINNING OF THE LINE
      MOV DL,0DH    ;CARRIAGE RETURN
      INT 21H       ;GO TO NEW LINE

      CALL OUTHEX   ;SHOWING THE RESULT OF THE ADDITION OF TWO INPUT

      MOV AH,4CH    ;EXIT FUNCTION
      INT 21H       ;TRANSFERING THE CONTROL TO OS

   MAIN ENDP
```

```
INHEX PROC

    MOV AH,9      ;DISPLAT STRING FUNCTION
    LEA DX,STR1   ;GETTING THE STR1
    INT 21H       ;DISPLAY MESSAGE

    MOV AH,1      ;INPUT FUNCTION
    MOV CX,4      ;SETTING THE CX VALUE TO 4
    XOR BX,BX     ;CLEARING THE BX


    INP:

    INT 21H       ;TAKE INPUT
    CMP AL,0DH    ;CHECKING IF ENTER IS PRESSED OR NOT  ( 0DH=NEW LINE )
    JE SKIP       ;IF ENTER IS PRESSED THEN JUMP TO SKIP

    CMP AL,41H    ;CHECKING IF THE GIVEN INPUT IS BETWEEN A TO F
    JGE HEXA      ;IF TRUE THEN JUMP TO HEXA

    SUB AL,30H    ;CONVERTING TO DIGIT
    JMP SHIFT     ;JUMP TO SHIFT

    HEXA:
    SUB AL,37H    ;CONVERTING TO HEXA

    SHIFT:
    SHL BX,4      ;SHIFTING THE VALUE IN THE BX REGISTER LEFT BY 4 BITS
    OR BL,AL      ;CONCATING THE DIGITS

    LOOP INP      ;DECREMENT COUNTER AND AGAIN GO TO INP TILL COUNTER BECOMES 0


    SKIP:
    RET  ;POPS THE RETURN ADDRESS OFF THE STACK AND RETURNS THE CONTROL in LOCATION

 INHEX ENDP


 OUTHEX PROC

    MOV AH,9      ;DISPLAT STRING FUNCTION
    LEA DX,STR2  ;GETTING THE STR1
    INT 21H       ;DISPLAY MESSAGE

    XOR DX,DX     ;CLEARING DX
    XOR CX,CX     ;CLEARING CX
```

```
    MOV CX,4      ;SET THE COUNTER TO 4

    ADD BX,INP1   ;ADDING TWO INPUTS

    JNC CARRY_ZERO;IF CARRY ZERO THEN JUMP TO THIS LABEL

    MOV AH,2      ;DISPLAY FUNCTION
    MOV DL,31H    ;DISPLAYING 1 IF CARRY EXISTS
    INT 21H       ;DISPLAY

    CARRY_ZERO:

       MOV AH,2  ;DISPLAY FUNCTION
       MOV DL,BH ;GETTING THE BH VALUE
       SHR DL,4  ;SHIFTING THE VALUE IN THE DL RIGHT BY 4 BITS
       SHL BX,4  ;SHIFTING THE VALUE IN THE BX LEFT BY 4 BITS

       CMP DL,9  ;CHECKING IF IT IS A LETTER OR NOT
       JG LETTER ;IF IT IS GREATER THAN 9 THE JUMP TO LETTER

       ADD DL,30H;IF IT IS NOT THEN ADD 30H TO GET THE ASCII VALUE OF THAT DIGIT ;
       JMP PRINT ;GO TO PRINT

       LETTER:

       ADD DL,37H;ADDING 37H TO GET THE ASCII VALUE OF A TO F CHARACTER

       PRINT:

       INT 21H   ;DISPLAY
       LOOP CARRY_ZERO
        ;DECREMENT COUNTER AND AGAIN GO TO CARRY_ZERO TILL COUNTER BECOMES 0

    OUTHEX ENDP   ;END PROCEDURE OUTHEX
END MAIN          ;RETURN 0
```

3. Write a program that prompts the user to enter two binary numbers of up to 8 digits each, and
prints their sum on the next line in binary. Each input ends with a carriage return. Define two procedures called 'INBIN' and 'OUTBIN' respectively for handling binary input and binary output.
Sample execution:
TYPE 'A BINARY NUMBER, UP TO 8 DIGITS: 11001010
TYPE 'A BINARY NUMBER, UP TO 8 DIGITS: 00011100
THE BINARY SUM IS: 11100110

CODE :

```
.MODEL SMALL
.STACK 100H

.DATA
  STR1 DB "TYPE 'A BINARY NUMBER, UP TO 8 DIGITS:$"
  STR2 DB "THE BINARY SUM IS:$"
  INP1 DB ?

.CODE
   MAIN PROC
      MOV AX,@DATA ;MOVING THE DATA SEGMENT INTO AX
      MOV DS,AX    ;INITIALIZE DS

      CALL INBIN   ;TAKING THE FIRST INPUT
      MOV INP1,BH  ;STORING IT INTO THE INP1 VARIABLE

      ;NEW LINE

      MOV AH,2     ;DISPLAY FUNCTION
      MOV DL,0AH   ;LINE FEED
      INT 21H      ;GO TO BEGINNING OF THE LINE
      MOV DL,0DH   ;CARRIAGE RETURN
      INT 21H      ;GO TO NEW LINE

      CALL INBIN   ;TAKING THE SECOND INPUT

      ;NEW LINE

      MOV AH,2     ;DISPLAY FUNCTION
      MOV DL,0AH   ;LINE FEED
      INT 21H      ;GO TO BEGINNING OF THE LINE
      MOV DL,0DH   ;CARRIAGE RETURN
```

```
    INT 21H      ;GO TO NEW LINE

    CALL OUTBIN  ;DISPLAYING THE OUTPUT OF THE ADDITION

    MOV AH,4CH   ;EXIT FUNCTION
    INT 21H      ;TRANSFERING THE CONTROL TO THE OS


  MAIN ENDP

  INBIN PROC
    MOV AH,9     ;DISPLAY STRING FUNCTION
    LEA DX,STR1  ;GETTING THE STR1
    INT 21H      ;DISPLAY

    XOR BX,BX    ;CLEARING BX
    XOR CX,CX    ;CLEARING CX
    MOV AH,1     ;INPUT FUNCTION
    MOV CX,8     ;SET THE COUNTER TO 8

    INP:

    INT 21H      ;TAKE INPUT
    SUB AL,30H   ;CONVERTING IT INTO BINARY

    SHL BH,1     ;SHIFTING THE VALUE IN THE BH REGISTER LEFT BY 1 BITS
    OR BH,AL     ;CONCATING THE DIGITS

    LOOP INP     ;DECREMENT COUNTER AND AGAIN GO TO INP TILL COUNTER BECOMES 0

    RET
;POPS THE RETURN ADDRESS OFF THE STACK AND RETURNS THE CONTROL TO THAT LOCATION

    INBIN ENDP   ;END INBIN PROCEDURE


    OUTBIN PROC

    MOV AH,9     ;DISPLAY STRING FUNCTION
    LEA DX,STR2  ;GET STR2
    INT 21H      ;DISPLAY

    ADD BH,INP1  ;ADD THE TWO INPUTS
    MOV AH,2     ;DISPLAY FUNCTION
    MOV CX,8     ;SET THE COUNTER

    OUTPUT:

    SHL BH,1     ;SHIFTING THE VALUE IN THE BH REGISTER LEFT BY 1 BITS
    JC CARRY     ;IF CARRY EXISTS THE GO TO CARRY
```

```
    MOV DL,'0'    ;IF NOT THEN GET '0'
    INT 21H       ;DISPLAY '0'
    JMP END       ;GO TO END

    CARRY:

    MOV DL,'1'    ;GET '1'
    INT 21H       ;DISPLAY '1'

    END:

    LOOP OUTPUT
;DECREMENT COUNTER AND AGAIN GO TO OUTPUT TILL COUNTER BECOMES 0

    OUTBIN ENDP   ;END OUTBIN PROC
END MAIN          ;RETURN 0
```

****

INDEC, INHEX, OUTDEC, OUTHEX procedural codes provided by sir pasted below :

INDEC :

```
INDEC PROC
;READ NUMBER IN RANGE -32768 TO 32767
;input :none
;output :AX =binary equivalent of number
   PUSH BX
   PUSH CX
   PUSH DX
;print prompt
@BEGIN:
   MOV AH,2
   MOV DL,'?'
   INT 21H ;print '?'
;total =0
   XOR BX,BX ;BX hold total
;negative =false
   XOR CX,CX ;CX hold sign
;read char.
   MOV AH,1
   INT 21H
;case char. of
   CMP AL,'-' ;minus sign
   JE @MINUS ;yes,set sign
   CMP AL,'+' ;plus sign
   JE @PLUS ;yes,get another char.
   JMP @REPEAT2 ;start processing char.
@MINUS:
   MOV CX,1
```

```
@PLUS:
    INT 21H
;end case
@REPEAT2:
;if char. is between '0' and '9'
    CMP AL,'0' ;char >='0'?
    JNGE @NOT_DIGIT ;illegal char.
    CMP AL,'9' ;char<='9' ?
    JNLE @NOT_DIGIT
;then convert char to digit
    AND AX,000FH
    PUSH AX   ;Save on stack
;total =total *10 +digit
    MOV AX,10
    MUL BX      ;AX= total * 10
    POP BX      ;retrive degit
    ADD BX,AX   ;total= total * 10 + degit
;Read a char
    MOV AH,1
    INT 21H
    CMP AL,0DH  ;carriage return?
    JNE @REPEAT2 ;no, keep going
;Untill CR
    MOV AX,BX   ;store number in AX
;if negetive
    OR CX,CX
    JE @EXIT ;NO, EXIT
;THEN
    NEG AX ; YES, NEG
;END_IF
@EXIT:
    POP DX ; RESTORE REGISTERS
    POP CX
    POP BX
    RET
;HERE IF ILLIGAL CHAR ENTERED
@NOT_DIGIT:
    ;MOVE CURSORE TO A NEW LINE
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV DL,0AH
    INT 21H
    JMP @BEGIN  ;GO TO BEGININIG
INDEC ENDP
```

INHEX :

```
INHEX PROC

    XOR BX,BX ; clear BX

    ;input character
    MOV CX,4
    MOV AH,1
    INT 21H

    LOOP1:
        ;check if inout is within 0 to 9
        CMP AL,0DH
        JE END1
        CMP AL,39H
        JG A_TO_F
        CMP AL, 30H
        JL A_TO_F

        AND AL,0FH
        JMP REPEAT1

    A_TO_F: ;check for input A to F
        SUB AL,37H

    REPEAT1:
        SHL BX,CL
        OR BL,AL
        INT 21H
        JMP LOOP1

    END1:
        RET

INHEX ENDP
```

OUTDEC :

```
OUTDEC PROC
;prints AX as a signed decimal integer
; input : AX
;output: none
    PUSH AX  ;save registers
    PUSH BX
    PUSH CX
    PUSH DX
```

```
    OR AX,AX    ;ax<0 ?
    JGE @END_IF1  ;no , >0
    PUSH AX       ;save numbers
    MOV DL,'-'    ;get '-'
    MOV AH,2
    INT 21H      ; PRINT '-'
    POP AX       ; GET AX BACK
    NEG AX       ;AX = -AX
@END_IF1:
;Get decimal degits
    XOR CX,CX    ;CX counts digits
    MOV BX,10D   ;BX has Divisor
@REPEAT1:
    XOR DX,DX    ;Prepare high word of divident
    DIV BX       ;AX= quitent DX= remainder
    PUSH DX      ;save remainder in stack
    INC CX       ; count++
;until
    OR AX,AX     ;quitent =0 ?
    JNZ @REPEAT1 ;no, keep going
;convert digits to caracter and print
    MOV AH,2
;for count times do
@PRINT_LOOP:
    POP DX       ;digit in DL
    OR DL,30H    ; convert to char
    INT 21H      ;print digit
    LOOP @PRINT_LOOP  ;loop until done
;end for
    POP DX      ;restore registers
    POP CX
    POP BX
    POP AX
    RET
OUTDEC ENDP
```

OUTHEX :

```
OUTHEX PROC

      MOV COUNT,0

      LOOP2:

        ;seperate digits
        MOV CX,0
        MOV DL,BH
        MOV CL,4
        SHR DL,CL
```

```
    ;display digits
    CMP DL,9
    JG SHOW_A_TO_F
    ADD DL,48D
    JMP OUTPUT

    SHOW_A_TO_F: ;display A to F
       ADD DL,55D
       JMP OUTPUT

    OUTPUT:
       MOV AH,2
       INT 21H

    INC COUNT
    CMP COUNT,4
    JGE END2
    MOV CL,4
    ROL BX,CL
    JMP LOOP2

  END2:
    RET

OUTHEX ENDP
```

Lab 04

*1. Write an assembly language program which converts an integer number i (0 – 255) from Fahrenheit degrees to Celsius degrees. Use C = (5/9) * (F - 32) as conversion formula. Where the i is entered by user.*
Sample Execution:
Enter Temperature in Fahrenheit (0 – 255):
212
Temperature in Celsius:
100

CODE :

```
INCLUDE "EMU8086.INC"                ; INCLUDE EMU8086 LIBRARY
.MODEL SMALL                         ; MODEL SMALL
.STACK 100H                          ; STACK SIZE 100H

.DATA                                ; DATA SECTION
   FAHRENHEIT DW ?
   CELSIUS DW ?

.CODE
```

```
                        ; MAIN PROGRAM
MAIN PROC

   MOV AX, @DATA                ; MOVE DATA SEGMENT ADDRESS TO AX
   MOV DS, AX                   ; SET DS TO DATA SEGMENT



   PRINTN "ENTER TEMPERATURE IN FAHRENHEIT (0-255):"

                        ; READ INPUT FAHRENHEIT TEMPERATURE
   CALL INDEC
   MOV FAHRENHEIT, AX

                        ;CALL OUTDEC

                        ;CONVERSION OF FAHRENHEIT TO CELSIUS
   MOV BX, FAHRENHEIT
   SUB BX, 32
   MOV AX, 5
   MUL BX
   MOV BX, 9
   DIV BX
   MOV CELSIUS, AX




   PRINTN                       ; NEW LINE

   PRINTN "TEMPERATURE IN CELSIUS:"

                        ; DISPLAY OUTPUT CELSIUS TEMPERATURE

   MOV AX, CELSIUS
   CALL OUTDEC

                        ; EXIT PROGRAM
   MOV AH, 4CH
   INT 21H
MAIN ENDP


INDEC PROC
                        ;READ INPUT

                        ;OUTPUT BINARY EQUIVALENT OF NUMBER
      PUSH BX
      PUSH CX
      PUSH DX
```

```
                    ;PRINT

@BEGIN:


   XOR BX,BX              ;REGISTER BX HOLDS TOTAL

   XOR CX,CX              ;CX HOLD NEGATIVE SIGN
                    ;CHARACTER READ
   MOV AH,1
   INT 21H

   CMP AL,'-'            ;MINUS SIGN IN CASE OF NEGATIVE
   JE @MINUS               ;WHEN SET SIGN
   CMP AL,'+'            ;WHEN PLUS SIGN
   JE @PLUS              ;GET ANOTHER CHARACTER
   JMP @REPEAT2             ;START PROCESSING CHARACTER
@MINUS:
   MOV CX,1
@PLUS:
   INT 21H
                    ;END CASE
@REPEAT2:
                    ;IF CHARACTER IS BETWEEN '0' AND '9'
   CMP AL,'0'            ;WHEN CHARACTER >='0'?
   JNGE @NOT_DIGIT            ;WHEN ILLEGAL CHARACTER
   CMP AL,'9'            ;WHEN CHARACTER <='9' ?
   JNLE @NOT_DIGIT
                    ;THEN CONVERT CHARACTER TO DIGIT
   AND AX,000FH
   PUSH AX               ;STORED ON THE STACK
                    ;TOTAL = TOTAL *10 + DIGIT
   MOV AX,10
   MUL BX                ;AX = TOTAL * 10
   POP BX                ;RETRIVE DEGIT
   ADD BX,AX             ;TOTAL= TOTAL * 10 + DIGIT
                    ;READ A CHARACTER
   MOV AH,1
   INT 21H
   CMP AL,0DH             ;WHEN CARRIAGE RETURN
   JNE @REPEAT2             ;IF NO MOVE ON
                    ;UNTILL CHARACTER
   MOV AX,BX             ;STORE NUMBER IN AX
                    ;IF NUMBER IS NEGETIVE
   OR CX,CX
   JE @EXIT              ;ELSE EXIT
                    ;THEN
   NEG AX                ;YES IF NEGATIVE
                    ;END_IF
@EXIT:
```

```
    POP DX                  ; RESTORE REGISTERS
    POP CX
    POP BX
    RET
                        ;HERE IF ILLIGAL CHARACTER ENTERED
  @NOT_DIGIT:
                        ;MOVE CURSOR TO A NEW LINE
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV DL,0AH
    INT 21H
    JMP @BEGIN              ;GO TO THE TOP
  INDEC ENDP


OUTDEC PROC
                        ;PRINTS AX AS A SIGNED DECIMAL INTEGER
                        ;INPUT IN AX
                        ;OUTPUT
    PUSH AX             ;SAVE REGISTERS
    PUSH BX
    PUSH CX
    PUSH DX

    OR AX,AX            ;IS AX < 0 ?
    JGE @END_IF1            ;NO IF > 0
    PUSH AX             ;SAVE NUMBERS
    MOV DL,'-'          ;GET '-' SIGN
    MOV AH,2
    INT 21H             ; PRINT '-'
    POP AX              ; GET AX BACK
    NEG AX              ;AX = -AX
  @END_IF1:
                        ;GET DECIMAL DEGITS
    XOR CX,CX           ;CX COUNTS DIGITS
    MOV BX,10D          ;BX HAS DIVISOR
  @REPEAT1:
    XOR DX,DX           ;PREPARE HIGH WORD OF DIVIDENT
    DIV BX              ;AX = QUITENT DX = REMAINDER
    PUSH DX             ;SAVE REMAINDER IN STACK
    INC CX              ;COUNT++ UNTIL QUITENT = 0 ?

    OR AX,AX
    JNZ @REPEAT1            ;ELSE MOVE ON
                        ;CONVERT DIGITS TO CARACTER AND PRINT
    MOV AH,2

  @PRINT_LOOP:
    POP DX              ;DIGIT IN DL
```

```
        OR DL,30H                ; CONVERT TO CHARACTER
        INT 21H                  ;PRINT DIGIT
        LOOP @PRINT_LOOP         ;LOOP UNTIL NOT FINISHED
                                 ;END FOR LOOP
        POP DX                   ;RESTORE REGISTERS
        POP CX
        POP BX
        POP AX
        RET
    OUTDEC ENDP
END
```

2 Modify the Task-1 to let the user to enter 6 integer numbers (0–255) as 6 Fahrenheit degrees separated by a carriage return. Then the program puts the summation of these numbers in Celsius degrees in DX register. Use a looping structure to display the summation. Use separate procedure named "Fahrenheit_to_Celsius" for conversion.
Sample Execution:
Enter 6 Temperature in Fahrenheit (0 – 255):
212
5
100
120
21
33
Temperature Summation in Celsius:
255

CODE :

```
INCLUDE "EMU8086.INC"              ; INCLUDE EMU8086 LIBRARY
.MODEL SMALL                       ; SMALL MODEL
.STACK 100H                        ; STACK SIZE 100H
                                   ; DATA SECTION
.DATA
   FAHRENHEIT DW 6 DUP (?)
   CELSIUS DW ?
   SUM DW 0


.CODE
                                   ; MAIN PROGRAM
   MAIN PROC                       ; MOVE DATA SEGMENT ADDRESS to AX
      MOV AX, @DATA                ; SET DS TO DATA SEGMENT
      MOV DS, AX

                                   ; READ INPUT FAHRENHEIT TEMPERATURE
```

```
  PRINTN "ENTER TEMPERATURE IN FAHRENHEIT (0-255):"

  MOV CX, 6
  MOV SI, 0
  LEA DX, FAHRENHEIT              ; LOAD OFFSET OF ARRAY INTO DX

  READ_INPUT:
     CALL INDEC
     MOV FAHRENHEIT, AX
    MOV BX, DX                    ; MOVE THE CONTENTS OF DX TO BX
     ADD BX, SI                   ; ADD THE CONTENTS OF SI TO BX
     MOV [BX], AX                  ; STORE AX INTO MEMORY AT ADDRESS BX
                                ; CALL OUTDEC
     ADD SUM, AX

     PRINTN                     ; NEW LINE

     MOV AX, [BX]
                                ; CALL OUTDEC
     LOOP READ_INPUT



 PRINTN                         ; NEW LINE

 MOV AX, SUM
 CALL OUTDEC

                                ;CONVERT FAHRENHEIT TO CELSIUS
  MOV BX, SUM
  CALL FAHRENHEIT_TO_CELSIUS
                                ;SUB BX, 32
                                ;MOV AX, 5
                                ;MUL BX
                                ;MOV BX, 9
                                ;DIV BX
  MOV CELSIUS, AX

  PRINTN

  PRINTN "Temperature Summation in Celsius "

  MOV AX, CELSIUS
  CALL OUTDEC

                                ; EXIT PROGRAM
  MOV AH, 4CH
  INT 21H
MAIN ENDP
```

```
FAHRENHEIT_TO_CELSIUS PROC
                        ; CONVERT FAHRENHEIT TO CELSIUS
                        ; INPUT: BX = FAHRENHEIT TEMPERATURE
                        ; OUTPUT: AX = CELSIUS TEMPERATURE
  SUB BX, 32                ; SUBTRACT 32 FROM FAHRENHEIT TEMPERATURE
  MOV AX, 5                 ; MULTIPLY BY 5
  MUL BX
  MOV BX, 9                 ; DIVIDE BY 9
  DIV BX
  RET
FAHRENHEIT_TO_CELSIUS ENDP


INDEC PROC
                        ;READ NUMBER IN RANGE -32768 TO 32767
                        ;INPUT :NONE
                        ;OUTPUT :AX =BINARY EQUIVALENT OF NUMBER
    PUSH BX
    PUSH CX
    PUSH DX
                        ;PRINT PROMPT TO THE USER
  @BEGIN:

                        ;TOTAL = 0
    XOR BX,BX               ;BX HOLD TOTAL
                        ;NEGATIVE =FALSE
    XOR CX,CX ;CX HOLD SIGN
                        ;READ CHAR.
    MOV AH,1
    INT 21H
                        ;CASE CHAR. OF
    CMP AL,'-'              ;MINUS SIGN
    JE @MINUS                 ;YES,SET SIGN
    CMP AL,'+'              ;PLUS SIGN
    JE @PLUS                ;ELSE GET ANOTHER CHAR.
    JMP @REPEAT2              ;START PROCESSING CHAR.
  @MINUS:
    MOV CX,1
  @PLUS:
    INT 21H
                        ;END CASE
  @REPEAT2:
                        ;IF CHARACTER IS BETWEEN '0' AND '9'
    CMP AL,'0'              ;CHAR >= '0'?
    JNGE @NOT_DIGIT             ;ILLEGAL CHARACTER
    CMP AL,'9'              ;CHARACTER <= '9' ?
    JNLE @NOT_DIGIT
                        ;THEN CONVERT CHAR TO DIGIT
```

```
    AND AX,000FH
    PUSH AX                 ;SAVE ON STACK
                            ;TOTAL =TOTAL *10 +DIGIT
    MOV AX,10
    MUL BX                  ;AX = TOTAL * 10
    POP BX                  ;RETRIVE DEGIT
    ADD BX,AX                ;TOTA = TOTAL * 10 + DIGIT
                            ;READ A CHAR
    MOV AH,1
    INT 21H
    CMP AL,0DH              ;CARRIAGE RETURN?
    JNE @REPEAT2              ;NO, KEEP GOING
                            ;UNTILL CR
    MOV AX,BX               ;STORE NUMBER IN AX
                            ;IF NEGETIVE
    OR CX,CX
    JE @EXIT                ;ELSE EXIT
                            ;THEN
    NEG AX                  ;IF NEGATIVE
                            ;END_IF
  @EXIT:
    POP DX                  ; RESTORE REGISTERS
    POP CX
    POP BX
    RET
                            ;HERE IF ILLIGAL CHAR ENTERED
  @NOT_DIGIT:
                            ;MOVE CURSOR TO A NEW LINE
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV DL,0AH
    INT 21H
    JMP @BEGIN              ;GO TO BEGININIG
  INDEC ENDP


OUTDEC PROC
                            ;PRINTS AX AS A SIGNED DECIMAL INTEGER
                            ;INPUT IN AX

    PUSH AX                 ;SAVE REGISTERS
    PUSH BX
    PUSH CX
    PUSH DX

    OR AX,AX                ;AX  <0 ?
    JGE @END_IF1            ;ELSE >0
    PUSH AX                 ;SAVE NUMBERS
    MOV DL,'-'              ;GET '-'
```

```
        MOV AH,2
        INT 21H                 ;PRINT '-' FOR NEGATIVE
        POP AX                  ;GET AX BACK
        NEG AX                  ;AX = -AX
    @END_IF1:
                            ;GET DECIMAL DEGITS
        XOR CX,CX               ;CX COUNTS DIGITS
        MOV BX,10D              ;BX HAS DIVISOR
    @REPEAT1:
        XOR DX,DX               ;PREPARE HIGH WORD OF DIVIDENT
        DIV BX                  ;AX STORES QUITENT, DX STORES REMAINDER
        PUSH DX                 ;SAVE REMAINDER IN STACK
        INC CX                  ; COUNT++ UNTILL QUITENT = 0

        OR AX,AX
        JNZ @REPEAT1            ;ELSE KEEP GOING
                            ;CONVERT DIGITS TO CARACTER AND PRINT
        MOV AH,2
                            ;FOR COUNT TIMES DO
    @PRINT_LOOP:
        POP DX                  ;DIGIT IN DL
        OR DL,30H               ;CONVERT TO CHARACTER
        INT 21H                 ;PRINT DIGIT
        LOOP @PRINT_LOOP        ;LOOP UNTIL DONE
                            ;END FOR LOOP
        POP DX                  ;RESTORE REGISTERS
        POP CX
        POP BX
        POP AX
        RET
    OUTDEC ENDP

END
OF DIVIDENT
        DIV BX                  ;AX STORES QUITENT DX STORES REMAINDER
        PUSH DX                 ;SAVE REMAINDER IN STACK
        INC CX                  ;COUNT++
                            ;UNTIL QUITENT =0 ?
        OR AX,AX
        JNZ @REPEAT1            ;ELSE KEEP GOING
                            ;CONVERT DIGITS TO CARACTER AND PRINT
        MOV AH,2
                            ;FOR COUNT TIMES DO
    @PRINT_LOOP:
        POP DX                  ;DIGIT IN DL
        OR DL,30H               ;CONVERT TO CHARACTER
        INT 21H                 ;PRINT DIGIT
        LOOP @PRINT_LOOP        ;LOOP UNTIL DONE
                            ;END FOR LOOP
        POP DX                  ;RESTORE REGISTERS
```

```
        POP CX
        POP BX
        POP AX
        RET
    OUTDEC ENDP
END
```

3. Modify the Task-2 to store this 6 integer numbers as 6 Fahrenheit degrees in an array and print
the array.
Sample Execution:
Enter 6 Temperature in Fahrenheit (0 – 255):
212
5
100
120
21
33
The Array:
212
5
100
120
21
33

CODE :

```
INCLUDE "EMU8086.INC"              ; INCLUDE EMU8086 LIBRARY
.MODEL SMALL                       ; SMALL MODEL
.STACK 100H                        ; STACK SIZE 100H
                                   ; DATA SECTION
.DATA
   FAHRENHEIT DW 6 DUP (?)
   CELSIUS DW ?
   SUM DW 0

.CODE
                                   ; MAIN PROGRAM
   MAIN PROC
      MOV AX, @DATA                ; MOVE DATA SEGMENT ADDRESS TO AX
      MOV DS, AX                   ; SET DS TO DATA SEGMENT
```

```
                        ; READ INPUT FAHRENHEIT TEMPERATURE
   PRINTN "ENTER TEMPERATURE IN FAHRENHEIT (0-255):"


   MOV CX, 6
   MOV SI, 0
   LEA DX, FAHRENHEIT            ; LOAD OFFSET OF ARRAY INTO DX

                        ; STORE THE VALUES 1 TO 10 IN THE ARRAY
  MOV CX, 6                  ; SET THE LOOP COUNTER TO 10
  MOV BX, 0                  ; SET THE BASE OFFSET OF THE ARRAY
  MOV AX, 1                  ; SET THE STARTING VALUE
READ_INPUT:
   CALL INDEC
   ADD SUM, AX
   MOV [FAHRENHEIT+BX], AX       ; STORE THE VALUE IN THE ARRAY
   ADD BX, 2                  ; INCREMENT THE OFFSET BY 2 (I.E., THE SIZE OF A WORD)
   ADD AX, 1                  ; INCREMENT THE VALUE BY 1

   PRINTN                     ; NEW LINE

   LOOP READ_INPUT               ; CONTINUE UNTIL THE LOOP COUNTER REACHES 0

                        ; DISPLAY THE CONTENTS OF THE ARRAY
   MOV CX, 6                  ; SET THE LOOP COUNTER TO 10
   MOV BX, 0                  ; RESET THE BASE OFFSET OF THE ARRAY

  PRINTN                      ; NEW LINE

PRINT_INPUT:
      MOV AX, [FAHRENHEIT+BX]     ; LOAD THE VALUE FROM THE ARRAY INTO AX
      CALL OUTDEC              ; DISPLAY THE VALUE IN AX (E.G., USING INT 21H)
      ADD BX, 2                ; INCREMENT THE OFFSET BY 2

      PRINTN                   ; NEW LINE

      LOOP PRINT_INPUT          ; CONTINUE UNTIL THE LOOP COUNTER REACHES 0


   PRINTN                      ; NEW LINE

   MOV AX, SUM
   CALL OUTDEC

                        ;CONVERT FAHRENHEIT TO CELSIUS
   MOV BX, SUM
   CALL FAHRENHEIT_TO_CELSIUS

   MOV CELSIUS, AX
```

```
                    ; DISPLAY OUTPUT CELSIUS TEMPERATURE
  PRINTN

  PRINTN "TEMPERATURE IN CELSIUS:"

   MOV AX, CELSIUS
   CALL OUTDEC

                    ; EXIT PROGRAM
   MOV AH, 4CH
   INT 21H

   MAIN ENDP



FAHRENHEIT_TO_CELSIUS PROC
                    ; CONVERT FAHRENHEIT TO CELSIUS
                    ; INPUT: BX = FAHRENHEIT TEMPERATURE
                    ; OUTPUT: AX = CELSIUS TEMPERATURE
   SUB BX, 32           ; SUBTRACT 32 FROM FAHRENHEIT TEMPERATURE
   MOV AX, 5            ; MULTIPLY BY 5
   MUL BX
   MOV BX, 9            ; DIVIDE BY 9
   DIV BX
   RET
FAHRENHEIT_TO_CELSIUS ENDP


INDEC PROC
                    ;READ NUMBER
                    ;INPUT :NONE
                    ;OUTPUT :AX =BINARY EQUIVALENT OF NUMBER
      PUSH BX
      PUSH CX
      PUSH DX
                    ;PRINT PROMPT TO THE USER
   @BEGIN:

                    ;TOTAL = 0
      XOR BX,BX         ;BX HOLD TOTAL
                    ;NEGATIVE =FALSE
      XOR CX,CX         ;CX HOLD SIGN
                    ;READ CHARACTER
      MOV AH,1
      INT 21H
                    ;CASE CHARACTER OF ;MINUS SIGN
      CMP AL,'-'
      JE @MINUS          ;YES,SET SIGN
```

```
   CMP AL,'+'              ;PLUS SIGN
   JE @PLUS                 ;YES,GET ANOTHER CHARACTER.
   JMP @REPEAT2              ;START PROCESSING CHARACTER
@MINUS:
   MOV CX,1
@PLUS:
   INT 21H
                     ;END CASE
@REPEAT2:
                     ;IF CHAR. IS BETWEEN '0' AND '9'
   CMP AL,'0'              ;CHAR >='0'?
   JNGE @NOT_DIGIT           ;ILLEGAL CHAR.
   CMP AL,'9'              ;CHAR<='9' ?
   JNLE @NOT_DIGIT
                     ;THEN CONVERT CHAR TO DIGIT
   AND AX,000FH
   PUSH AX                  ;SAVE ON STACK
                     ;TOTAL =TOTAL *10 +DIGIT
   MOV AX,10
   MUL BX                  ;AX= TOTAL * 10
   POP BX                  ;RETRIVE DEGIT
   ADD BX,AX                ;TOTAL= TOTAL * 10 + DEGIT
                     ;READ A CHAR
   MOV AH,1
   INT 21H
   CMP AL,0DH               ;CARRIAGE RETURN?
   JNE @REPEAT2              ;NO, KEEP GOING
                     ;UNTILL CR
   MOV AX,BX               ;STORE NUMBER IN AX
                     ;IF NEGETIVE
   OR CX,CX
   JE @EXIT                ;NO, EXIT
                     ;THEN
   NEG AX                  ; YES, NEG
                     ;END_IF
@EXIT:
   POP DX                  ; RESTORE REGISTERS
   POP CX
   POP BX
   RET
                     ;HERE IF ILLIGAL CHAR ENTERED
@NOT_DIGIT:
                     ;MOVE CURSORE TO A NEW LINE
   MOV AH,2
   MOV DL,0DH
   INT 21H
   MOV DL,0AH
   INT 21H
   JMP @BEGIN               ;GO TO BEGININIG
INDEC ENDP
```

```
OUTDEC PROC
                        ;PRINTS AX AS A SIGNED DECIMAL INTEGER
                        ; INPUT : AX
                        ;OUTPUT: NONE
    PUSH AX                 ;SAVE REGISTERS
    PUSH BX
    PUSH CX
    PUSH DX

    OR AX,AX                ;AX < 0 ?
    JGE @END_IF1                ;NO  > 0
    PUSH AX                 ;SAVE NUMBERS
    MOV DL,'-'              ;GET '-'
    MOV AH,2
    INT 21H                 ; PRINT '-'
    POP AX                  ; GET AX BACK
    NEG AX                  ;AX = -AX
  @END_IF1:
                        ;GET DECIMAL DEGITS
    XOR CX,CX               ;CX COUNTS DIGITS
    MOV BX,10D               ;BX HAS DIVISOR
  @REPEAT1:
    XOR DX,DX               ;PREPARE HIGH WORD OF DIVIDENT
    DIV BX                 ;AX = QUITENT DX= REMAINDER
    PUSH DX                 ;SAVE REMAINDER IN STACK
    INC CX                 ; COUNT++ UNTIL QUITENT = 0

    OR AX,AX
    JNZ @REPEAT1            ;ELSE KEEP GOING
                        ;CONVERT DIGITS TO CARACTER AND PRINT
    MOV AH,2

                        ;FOR COUNT
    PRINT "The Array Element:"

  @PRINT_LOOP:
    POP DX                 ;DIGIT STORED IN DL
    OR DL,30H               ; CONVERT TO CHARACTER
    INT 21H                ;PRINT DIGIT
    LOOP @PRINT_LOOP          ;LOOP UNTIL DONE
                        ;END FOR LOOP
    POP DX                  ;RESTORE REGISTERS
    POP CX
    POP BX
    POP AX
    RET
  OUTDEC ENDP
END
```

4. Modify the Task-3 to Find the Largest element in this array. Print the Largest element in Hexadecimal form if the element is an odd number. To check a number is even or odd, you must use a logic instruction which does bit wise operation.
Sample Execution:
Enter 6 Temperature in Fahrenheit (0 – 255):
212
5
100
120
21
33
The Largest element of this Array (in Hexadecimal):
D4

CODE :

```
INCLUDE "EMU8086.INC"                    ; INCLUDE EMU8086 LIBRARY

.MODEL SMALL                              ; SMALL MODEL
.STACK 100H                              ; STACK SIZE 100H
                                  ; DATA SEGMENT
.DATA
   FAHRENHEIT DW 6 DUP (?)
   CELSIUS DW ?
   SUM DW 0
   LARGEST DW ?
   COUNT DB ?


.CODE

   MAIN PROC                              ; MAIN PROGRAM
      MOV AX, @DATA                        ; MOVE DATA SEGMENT ADDRESS to AX
      MOV DS, AX                          ; SET DS TO DATA SEGMENT
```

```
                                ; READ INPUT FAHRENHEIT TEMPERATURE
    PRINTN "ENTER 6 TEMPERATURE IN FAHRENHEIT (0-255):"

    MOV CX, 6
    MOV SI, 0
    LEA DX, FAHRENHEIT                  ; LOAD OFFSET OF ARRAY INTO DX

                                ; STORE THE VALUES 1 TO 10 IN THE ARRAY
   MOV CX, 6                        ; SET THE LOOP COUNTER TO 10
   MOV BX, 0                        ; SET THE BASE OFFSET OF THE ARRAY
   MOV AX, 1                        ; SET THE STARTING VALUE
  READ_INPUT:
    CALL INDEC
    ADD SUM, AX
    MOV [FAHRENHEIT+BX], AX             ; STORE THE VALUE IN THE ARRAY

                                ; FIND THE LARGEST ELEMENT IN THE ARRAY
  CMP AX, LARGEST
  JLE NOT_LARGEST
  MOV LARGEST, AX
 NOT_LARGEST:


    ADD BX, 2                   ; INCREMENT THE OFFSET BY 2
    ADD AX, 1                   ; INCREMENT THE VALUE BY 1

    PRINTN                      ; NEW LINE

    LOOP READ_INPUT                 ; CONTINUE UNTIL THE LOOP COUNTER REACHES 0




    PRINTN                      ; NEW LINE
    PRINTN "The Largest element of this Array (in Hexadecimal):"

    MOV BX, LARGEST
    CALL OUTHEX

    PRINTN



                                ; DISPLAY THE CONTENTS OF THE ARRAY
    MOV CX, 6                       ; SET THE LOOP COUNTER TO 10
    MOV BX, 0                       ; RESET THE BASE OFFSET OF THE ARRAY

    PRINTN                      ; NEW LINE

  PRINT_INPUT:
```

```
    MOV AX, [FAHRENHEIT+BX]          ; LOAD THE VALUE FROM THE ARRAY INTO AX
    CALL OUTDEC                      ; DISPLAY THE VALUE IN AX
    ADD BX, 2                        ; INCREMENT THE OFFSET BY 2

    PRINTN                           ; NEW LINE
    LOOP PRINT_INPUT                 ; CONTINUE UNTIL THE LOOP COUNTER REACHES 0


  PRINTN                             ; NEW LINE

  MOV AX, SUM
  CALL OUTDEC

                                     ;CONVERT FAHRENHEIT TO CELSIUS
  MOV BX, SUM
  CALL FAHRENHEIT_TO_CELSIUS

  MOV CELSIUS, AX

                                     ; DISPLAY OUTPUT CELSIUS TEMPERATURE
 PRINTN
 PRINTN "TEMPERATURE IN CELSIUS:"

  MOV AX, CELSIUS
  CALL OUTDEC

                                     ; EXIT PROGRAM
  MOV AH, 4CH
  INT 21H

  MAIN ENDP



FAHRENHEIT_TO_CELSIUS PROC
                                     ; CONVERT FAHRENHEIT TO CELSIUS
                                     ; BX = FAHRENHEIT TEMPERATURE
                                     ; AX = CELSIUS TEMPERATURE
  SUB BX, 32                         ; SUBTRACT 32 FROM FAHRENHEIT TEMPERATURE
  MOV AX, 5                          ; MULTIPLY BY 5
  MUL BX
  MOV BX, 9                          ; DIVIDE BY 9
  DIV BX
  RET
FAHRENHEIT_TO_CELSIUS ENDP


INDEC PROC
                                     ;READ NUMBER INPUT
                                     ;INPUT WBEN NONE
```

```
                        ;OUTPUT IN AX = BINARY EQUIVALENT OF NUMBER
   PUSH BX
   PUSH CX
   PUSH DX
                        ;PRINT PROMPT TO THE USER
@BEGIN:

                        ;TOTAL = 0
   XOR BX,BX                 ;BX HOLD TOTAL
                        ;NEGATIVE =FALSE
   XOR CX,CX                 ;CX HOLD SIGN
                        ;READ CHAR.
   MOV AH,1
   INT 21H
                        ;CASE CHAR. OF
   CMP AL,'-'              ;MINUS SIGN
   JE @MINUS                  ;YES,SET SIGN
   CMP AL,'+'              ;PLUS SIGN
   JE @PLUS                ;YES,GET ANOTHER CHAR.
   JMP @REPEAT2               ;START PROCESSING CHAR.
@MINUS:
   MOV CX,1
@PLUS:
   INT 21H
                        ;END CASE
@REPEAT2:
                        ;IF CHAR. IS BETWEEN '0' AND '9'
   CMP AL,'0'              ;CHAR >='0'?
   JNGE @NOT_DIGIT            ;ILLEGAL CHAR.
   CMP AL,'9'              ;CHAR<='9' ?
   JNLE @NOT_DIGIT
                        ;THEN CONVERT CHAR TO DIGIT
   AND AX,000FH
   PUSH AX                 ;SAVE ON STACK
                        ;TOTAL =TOTAL *10 +DIGIT
   MOV AX,10
   MUL BX                  ;AX= TOTAL * 10
   POP BX                  ;RETRIVE DEGIT
   ADD BX,AX               ;TOTAL= TOTAL * 10 + DEGIT
                        ;READ A CHAR
   MOV AH,1
   INT 21H
   CMP AL,0DH              ;CARRIAGE RETURN?
   JNE @REPEAT2               ;NO, KEEP GOING
                        ;UNTILL CR
   MOV AX,BX               ;STORE NUMBER IN AX
                        ;IF NEGETIVE
   OR CX,CX
   JE @EXIT                ;NO, EXIT
                        ;THEN
```

```
    NEG AX                      ; YES, NEG
                               ;END_IF
  @EXIT:
    POP DX                      ; RESTORE REGISTERS
    POP CX
    POP BX
    RET
                               ;HERE IF ILLIGAL CHAR ENTERED
  @NOT_DIGIT:
                               ;MOVE CURSOR TO A NEW LINE

    PRINTN
    JMP @BEGIN                      ;GO TO BEGININIG
  INDEC ENDP


OUTDEC PROC
                               ;PRINTS AX AS A SIGNED DECIMAL INTEGER
                               ; INPUT : AX
                               ;OUTPUT: NONE
    PUSH AX                     ;SAVE REGISTERS
    PUSH BX
    PUSH CX
    PUSH DX

    OR AX,AX                    ;AX<0 ?
    JGE @END_IF1                    ;NO , >0
    PUSH AX                     ;SAVE NUMBERS
    MOV DL,'-'                  ;GET '-'
    MOV AH,2
    INT 21H                     ; PRINT '-' FOR NEGATIVE SIGN
    POP AX                      ; GET AX BACK
    NEG AX                      ;AX = -AX
  @END_IF1:
                               ;GET DECIMAL DEGITS
    XOR CX,CX                   ;CX COUNTS DIGITS
    MOV BX,10D                  ;BX HAS DIVISOR
  @REPEAT1:
    XOR DX,DX                   ;PREPARE HIGH WORD OF DIVIDENT
    DIV BX                     ;AX= QUITENT DX= REMAINDER
    PUSH DX                     ;SAVE REMAINDER IN STACK
    INC CX                     ; COUNT++
                               ;UNTIL
    OR AX,AX                    ;QUITENT =0 ?
    JNZ @REPEAT1                    ;NO, KEEP GOING
                               ;CONVERT DIGITS TO CARACTER AND PRINT
    MOV AH,2
                               ;FOR COUNT TIMES DO
  @PRINT_LOOP:
    POP DX                      ;DIGIT IN DL
```

```
        OR DL,30H                   ; CONVERT TO CHAR
        INT 21H                     ;PRINT DIGIT
        LOOP @PRINT_LOOP                ;LOOP UNTIL DONE
                                ;END FOR
        POP DX                      ;RESTORE REGISTERS
        POP CX
        POP BX
        POP AX
        RET
     OUTDEC ENDP


   OUTHEX PROC

     MOV COUNT,0

     LOOP2:


       MOV CX,0
       MOV DL,BH
       MOV CL,4
       SHR DL,CL

                                ;DISPLAY THE DIGITS
       CMP DL,9
       JG SHOW_A_TO_F
       ADD DL,48D
       JMP OUTPUT

       SHOW_A_TO_F:                 ;DISPLAY HEX CARACTERS - > A TO F
          ADD DL,55D
          JMP OUTPUT

       OUTPUT:
          MOV AH,2
          INT 21H

       INC COUNT
       CMP COUNT,4
       JGE END2
       MOV CL,4
       ROL BX,CL
       JMP LOOP2

     END2:
       RET

OUTHEX ENDP
END
```

5. An assembly language program is written in L4_T3.asm, which is a modification of program PGM8_1.ASM in section 8.2 of "Assembly Language Programming and Organization of the IBM PC", Author: Ythe Yu, Charles Marut.
This program lets the user type some text, consisting of words separated by blanks, ending with a carriage return, and displays the text in a specified order.
a) Run the program and analyze the output
b) Write the detailed algorithm of this program

****** No L4_T.asm file was given in the classroom, so worked with the PGM8_1.ASM file of book ******

a)

CODE :

```
.MODEL SMALL
.STACK 100h
.DATA

.CODE
MAIN PROC

;display user prompt
   MOV AH, 2
   MOV DL, '?'
   INT 21H

;initialize character count
   XOR CX, CX

; prepare to read a character

   MOV AH, 1
   INT 21H

;while character is not a carriage return do
WHILE_:
   CMP AL, 0DH
   JE END_WHILE

   ;save character on the stack and increment count
   PUSH AX
   INC CX

   ;read a character
```

```
    INT 21H
    JMP WHILE_

END_WHILE:
;go to a new line
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV DL,0AH
    ;CR?
    ;yes, exit loop
    INT 21H
    JCXZ EXIT

;for count times
TOP:
    ;pop a character
    POP DX
    ;display it
    INT 21H
    LOOP TOP

EXIT:
    MOV AH, 4CH
    INT 21H

ENDP
END MAIN
```

Explanation :

In this task, I was asked to Run the program and analyze the output :

- **A MODEL SMALL and .STACK 100h, is used here.**
- **No data is stored in .DATA section**
- **The .CODE section contains the actual program code.**
- **The MOV AH, 2 and MOV DL, '?' lines display a prompt to the user**
- **The XOR CX, CX instruction sets the value of the CX register to zero**
- **Character are read using the MOV AH, 1 and INT 21H instructions**
- **The WHILE_ label begins a loop that continues until the user enters a carriage return (0DH).**
- **The CMP AL, 0DH instruction compares the value in the AL register to the value of a carriage return. The program jumps to END_WHILE if they are equal,**
- **The PUSH AX instruction saves the input character on the stack while the CX increments the character count.**
- **The JMP WHILE_ instruction jumps back to the WHILE_ label to check if the input character is a carriage return.**
- **When the user enters a carriage return The END_WHILE label is reached**
- **The MOV DL,0AH instruction moves the cursor to the beginning of the line.**

• If character count is zero, the program jumps to EXIT.
• The TOP label begins a loop that continues for CX iterations. Each iteration, the program pops a character off the stack and displays it.
• The LOOP TOP instruction decrements the CX register and jumps back to the TOP label until the CX register is zero.
• END MAIN labels show the end of the main program.

b)

- Display a question mark to prompt the user.
- Initialize a count variable to zero.
- Read a character from the user.
- While the character is not a carriage return:
- Push the character onto a stack.
- Increment the count variable.
- Read another character from the user.
- Go to a new line.
- For each character in the stack (count number of times):
- Pop a character from the stack.
- Display the character.
- End the program

Lab 5

****** Lab taken by imtiaj sir in group B (interfacing lab)*******

Lab # 2

Interfacing with LEDs Lights through 8255A in MDA 8086 Kit.

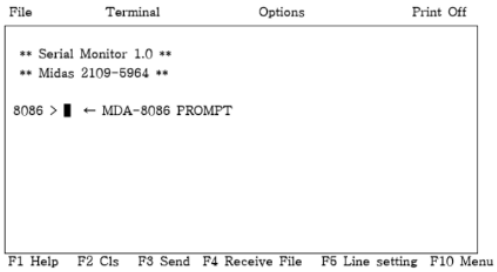
Objective:

To understand the basic interfacing concept through implementing a model for with LEDs through 8255A interfacing of MDA 8086 trainer Kit.

Theory:

- **Serial Monitor Mode**
  Serial monitor is the basic monitor program mode to have data communication between MDA-8086 and Computer (i.e., PC). To use serial monitor mode, move jumper P1 which located on the PCB like the following figure.

To connect the serial interface of the MDA-8086 with the Computer (PC) interface and run the WinComm program in PC to have the following screen shot after pressing RES key.



- **Operation Serial Monitor Command**
  User can only use command which stored at serial monitor using WinComm. Serial monitor can execute to command when user type the command and then press ENTER key. If there is no any command at serial monitor, error message will be displayed with bell sound and serial monitor prompt will be displayed again.
  8086 >?

HELP Command

E segment : offset.....: Enter Data To Memory

D segment : offset length.....: Dump Memory Contents

R [register name].....: Register Display & Change

M address1, length, address2.....: Move Memory From 1 to 2

F address, length, data.....: Fill Memory With Any Data

L Return key.....: Program Down Load

G segment : offset.....: Execute Program

T.....: Program 1 step execute

- **Basic Command Syntax:**

1. **Memory Modify Command**

Syntax: E segment: offset
Purpose: This command is used to enter data to memory.
Example:
8086 > E 0000:1000
0000:1000 FF ? 11
0000:1001 FF ? 22
0000:1002 FF ? 33
0000:1003 FF ? 44
0000:1004 FF ? 55
0000:1005 FF ? / (Offset decrement)
0000:1004 55 ?

2. **Memory Display Command**

Syntax: D segment: offset
Purpose: This command is used to display the data stored in memory.
Example:
8086> D 0000:1000
0000:1000 11 22 33 44 55 FF FF FF - FF FF FF FF FF FF FF FF
0000:1020 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF

2. Memory Display Command

Syntax: D segment: offset
Purpose: This command is used to display the data stored in memory.
Example:
8086> D 0000:1000
0000:1000 11 22 33 44 55 FF FF FF - FF FF FF FF FF FF FF FF
0000:1020 FF FF FF FF FF FF FF FF - FF FF FF FF FF FF FF FF

3. Display Register Command

Syntax: R

Purpose: The R command is used to display the 8086 processor registers

Example:

8086 > R

AX=0000 BX=0000 CX=0000 DX=0000

SP=0540 BP=0000 SI=0000 DI=0000

DS=0000 ES=0000 SS=0000 CS=0000

IP=1000 FL=0000 = ...

To change individual register:-

8086 > R AX

AX = 0000 1234

8086 > R BX

BX = 0000 4567

8086 > R CX

CX = 0000 7788

8086 > R DX

DX =0000 1111

4. Program Download and Execute Command

The L command moves object data in hexa format from an external devices to memory.

```
8086 >L↵
   Down load start !!    ← ( Note : See section 5. Serial monitor experiment)

:14100000B83412BB7856B90010BA00208BF08BFBBD0030BC08
:0910140000408EDA8ED18EC0CCB2
:00
   OK Completed !!
```

☞ Set IP

```
8086 >R IP↵
IP=1000↵
8086 >T↵
        AX=1234  BX=4567  CX=7788  DX=1111
        SP=0540  BP=0000  SI=0000  DI=0000
        DS=0000  ES=0000  SS=0000  CS=0000
        IP=1003   FL=0100  = . . . t . . . . .
         ↓
      Next address
```

☞ Execute program command

```
        Segment  Offset
           ↓      ↓
8086 >G 0000:1000↵
Execute Address = 0000:1000
```

- **LEDs in MDA 8086 Kit**

There are 4 LEDs namely RED (L1), GREEN (L2), YELLOW (L3) and RED (L4) inside the MDA–8086 trainer kit and can be modeled to design a simpler application. This requires 8255 PPI ports which are already connected to the 4 LEDs internally. Through a code we can access these ports and provide binary or hex values to switch on the required LED (on or off). In order to turn a particular LED ON, a logical '1' should be provided to a particular port. Note that only Port B of 8255A PPI is used in the following example code to control the LEDs.

RED **L1**	GREEN **L2**
YELLOW **L3**	RED **L4**

Different ports of Programmable Peripheral Interface (PPI) 8255A is used for switching on LEDs, the address of the ports are:

Port A: 19h	Port C: 1Dh
Port B: 1Bh	Control Register: 1FH

To control the LEDs, Port B will be used for the value to select the LEDs and Port A and C will be set with constant values. In-case of Port B, pass a value of '11110001' to select the L1 LED and pass a value of '11110010' to select the L2 LED and so on.

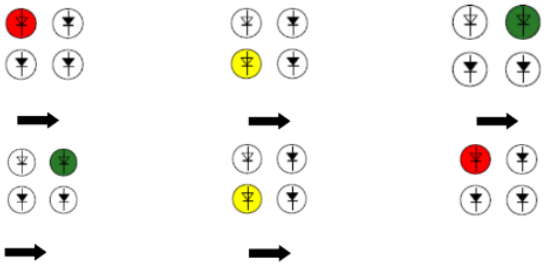
Setup jumper cap of P2 as following:

- **Example:** Example code to illuminate LEDs with a sequence of L1, L2, L3 and L4.

```
CODE    SEGMENT
        ASSUME      CS:CODE, DS:CODE, ES:CODE, SS:CODE
PPIC_C  EQU   1FH                ; Control register  address
PPIC    EQU   1DH                ; Port C address
PPIB    EQU   1BH                ; Port B address
PPIA    EQU   19H                ; Port A address

        ORG   1000H
        MOV   AL, 10000000B      ; Control register initialization with a 8255 Mode set
        OUT   PPIC_C, AL
        MOV   AL, 11111111B
        OUT   PPIA, AL
        MOV   AL, 00000000B
        OUT   PPIC, AL
L1:     MOV   AL, 11110001B
L2:     OUT   PPIB, AL           ; Select a L1 LED with AL value and make it ON
        CALL  TIMER
        SHL   AL, 1
        TEST  AL, 00010000B      ; Perform Logical AND and set Zero Flag (ZF), SF, PF
        JNZ   L1
        OR    AL, 11110000B      ; Perform Logical OR and store the result in AL
        JMP   L2
        INT   3                  ; Single-step interrupt
TIMER:  MOV   CX, 1
TIMER2:       PUSH  CX
              MOV   CX, 0
TIMER1:       NOP
              NOP
              NOP
              NOP
              LOOP  TIMER1
              POP   CX
              LOOP  TIMER2
              RET

CODE    ENDS
        END
```

Tasks to do:

1. Write a program to illuminate LEDs for the implementation of Simple Traffic Control Light Signaling model. Vehicles will be in STOP (with red color), WAIT (with yellow color) & GO (i.e., with green color) states for a moderately long time and vice-versa:



BOOK + CSE - EXERCISE (extras)

- *STACK AND PROCEDURE SECTION :*

```
.MODEL SMALL
.STACK 200H
.DATA
   Default_Stack_SP DW ?    ; Stack Pointer (or Top) Offset of Default Stack
   Stack2_SP DW 0100h  ; Stack Pointer (or Top) Offset of (Temporary)Stack_2

   InPrompt  DB "Input: $"
   OutPrompt  DB "Output: $"
   Str_len   DW ?      ;String length of input string
   Word_len   DW 0      ;WORD length

.CODE
MAIN PROC
   ;Load Data Segment
   MOV AX,@DATA
   MOV DS,AX
   ;display user prompt
   MOV AH,9
   LEA DX,InPrompt
   INT 21H

   ;initialize character count
   XOR CX,CX

   ;read a char
   MOV AH,1
   INT 21H

   ;input string untill carriage return
   WHILE:
      CMP AL,0DH
      JE END_WHILE

      PUSH AX
      INC CX

      INT 21H
   JMP WHILE

   END_WHILE:


      ;Push a Blank space(ASCII: 0020H) at top of default stack,
      ;it is used for word finding, while generating output
      MOV AX,0020H
```

```
    PUSH AX       ;push at top of the default stack

    ;String length = String length + 1 (for 1 additional space)
    INC CX
    MOV Str_len,CX
    JCXZ EXIT_      ; if string length is zero,
                ; then avoid infinite loop by LOOP operation

    ;print new line
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV DL,0AH
    INT 21H
    ;display Output prompt
    MOV AH,9
    LEA DX,InPrompt
    INT 21H

; Following loop exchanges the contents of default stack to stack2
; by popping from default stack and pushing it into stack2

Loop_DefaultStack_to_Stack2_TransferContent:
    ;pop from Default_Stack
    POP DX
    ;SAVE Stack Pointer (offset address) of Default_Stack
    MOV Default_Stack_SP, SP

    ;LOAD Stack Pointer (offset address) of Stack2
    MOV SP, Stack2_SP
    ;PUSH at Stack 2
    PUSH DX
    ;SAVE Stack Pointer (offset address) of Stack2
    MOV Stack2_SP, SP

    ;LOAD Stack Pointer (offset address) of Stack2
    MOV SP, Default_Stack_SP

LOOP Loop_DefaultStack_to_Stack2_TransferContent

    ;Set the string length for loop counter of following loop
    MOV CX,Str_len
    JCXZ EXIT_


Loop_Stack2_to_DefaultStack_TransferContent_and_Word_Parse:

    ;LOAD Stack Pointer (offset address) of Stack2
    MOV SP, Stack2_SP
    ;pop from Stack 2
```

```
  POP BX
  ;SAVE Stack Pointer (offset address) of Stack2
  MOV Stack2_SP, SP

  INC Word_len
  ;LOAD Stack Pointer (offset address) of Default_Stack
  MOV SP, Default_Stack_SP

  ;chck space
  CMP BL,20h
  JNE Continue_push
    ;if space found(word complete), then pop the all contents of
    ;default stack which hold the compete word.
    ;Poped chars are printed one by one.

    Complete_Word_Found:
      MOV AH,2        ;For INT 21H DOS Routine, AH is set to print char
                      ;inside the following loop

      ;deduct the space from word length
      ; since space would be printed manually
      DEC Word_len

      Loop_Print_word:
         ;POP
         POP DX
         INT 21h ;print the poped char
         ;word terminate or not?
         DEC Word_len
         CMP Word_len,0
         JE  Print_Space  ;if word printing terminates,
             ;then skip Pushing and Go for Popping from Stack2 again
      JMP Loop_Print_word
  Continue_push:
    ;PUSH at Default_Stack Again
    PUSH BX
    ;SAVE offset address of Default_Stack_SP
    MOV Default_Stack_SP, SP

    JMP Skip_Printing_Space

    Print_Space:
       CMP Word_len,0
       JNE Skip_Printing_Space
          ;Print_Space
          MOV DL,20h
          INT 21h
    Skip_Printing_Space:
       ;do nothing here, just go to next itaration of this loop
LOOP Loop_Stack2_to_DefaultStack_TransferContent_and_Word_Parse
```

```
    ;DOS Exit
    EXIT_:
        MOV AH,4CH
        INT 21H

MAIN ENDP

END MAIN
```

Analyze and explain this STACK and PROC based code statements :

Analyze :

The program takes a set of strings separated by space. In the output, each of those strings are reversed and displayed separated by spaces.

INPUT : CSE CIVIL
OUTPUT: ESC LIVIC

Explain :

1. Declare mode and stack

2. Declare variables in Data Segment:

    a. 1 for default stack offset

    b. 1 for temporary stack offset

    c. Message for input and output

    d. Variable to calculate word and string length

3. Load Data Segment in Code

4. Display User prompt using interrupt 21H

5. Clear CX using XOR

6. Read a character using interrupt 21H

7. Begin loop WHILE

    a. Push value of AX to Stack

    b. Increase value of CX (this will indicate the length of the string entered

    c. Repeat if not carriage return

8. If input is carriage return, go to END_WHILE

    a. Move a space to AX and push to stack

    b. Increase value of CX

    c. If CX is zero, go to

        i. Return control to DOS with interrupt 21H

9. Print a new line using 0DH (carriage return) and 0AH (line feed)

10. Display Output Prompt using interrupt 21H

11. Begin Loop_DefaultStack_to_Stack2_TransferContent: (this will continue until CX = 0)

    a. Pop Default Stack Value to DX

    b. Move value of Stack Pointer to Default_Stack_SP (Save SP value to Default Stack)

    c. Move the starting point of Stack2_SP to SP

    d. Push the Value of DX in Stack 2

    e. Move the Stack Pointer to Stack2_SP

    f. Move Default_Stack_SP back to SP

12. Move the value of Str_len to CX

13. Begin Loop Loop_Stack2_to_DefaultStack_TransferContent_and_Word_Parse: (continues until CX = 0)

    a. Move the starting address of Stack2_SP to SP (load stack2 in SP)

    b. Pop the current value of SP to BX

    c. Move the current value of SP to Stack2_SP (save the SP value in Stack2)

    d. Increase Word_len

    e. Move the start of Default_Stack_SP to SP (Load Default Stack in SP)

    f. Compare BL to 20H (Black Space)

    g. If BL != 20H, go to Continue_Push

        i. Push the current value of default stack in BX

        ii. Move SP value to Default_Stack_SP (Save SP value in Default Stack)

        iii. Jump to Skip_Printing_Space

1. Do nothing, just go to next iteration

iv. Go to Print_Space:

1. Compare Word_Len to 0

2. If word length is not zero, go to Skip_Printing_Space

3. Print space

h. If BL == 20H, go to Complete_Word_Found:

i. Move 2 o AH for character read

ii. Decrease Word_len

iii. Go to Loop_Print_Word:

1. Pop value to DX

2. Print character

3. Decrease Word_len

4. Compare Word_len to 0

5. If Word_len == 0, go to Print_Space

6. Jump to Loop_Print_Word

- *ARRAY SECTION*

1. Write a program to multiply two 16-bit decimal signed numbers where starting address is 2000 and the numbers are at 3000 and 3002 memory address and store result into 3004 and 3006 memory address

   CODE :

```
.MODEL SMALL
  .STACK 100H
  .DATA
    PROMPT1  DB  'First Number: $'
    PROMPT2  DB  'Second Number: $'
    NUM1 DW ?
    NUM2 DW ?
    PROMPT3  DB  'Product: $'
  .CODE
    MAIN PROC

      MOV AX,@DATA
      MOV DS,AX

      LEA DX,PROMPT1
      MOV AH,9
      INT 21H
      CALL INDEC
      MOV 3000H,AX
      CALL LINEBREAK

      LEA DX,PROMPT2
      MOV AH,9
      INT 21H
      CALL INDEC
      MOV 3002H,AX
      CALL LINEBREAK

      LEA DX,PROMPT3
      MOV AH,9
      INT 21H

      ; CALCULATION
      MOV AX, [3000H]
      MOV BX, [3002H]
      MUL BX
      MOV [3004H], AX
      MOV [3006H], AX
```

```
    ;output
    CALL OUTDEC

    MOV AH,4CH
    INT 21H

  MAIN ENDP

 ; FOLDER PATH
  INCLUDE C: \Codes\INDEC.ASM
  INCLUDE C:\Codes\OUTDEC.ASM
  INCLUDE C:\Codes\LINEBREAK.ASM

END MAIN
```

2. Write a program in 8086 microprocessor to find out the product of two arrays of 8-bit n numbers, where size “n” is stored at offset 500 and the numbers of first array are stored from offset 501 and the numbers of second array are stored from offset 601 and store the result numbers into first array i.e offset 501.

   CODE :

```
.MODEL SMALL
.STACK 100h
.DATA
   ARR DB 50 DUP(?) ; declare array with null value initially
   PROMPT1  DB   'ENTER SIZE OF ARRAY: $'
   PROMPT2  DB   'ENTER VALUES FOR ARRAY 1: $'
   PROMPT4  DB   'ENTER VALUES FOR ARRAY 2: $'
   PROMPT3  DB   'OUTPUT: $'
   COUNT DW ?

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

      LEA DX, PROMPT1
      MOV AH, 9
      INT 21H

      CALL INDEC
      MOV COUNT,AX
      MOV SI,501H
      CALL LINEBREAK
```

```
LEA DX, PROMPT2
MOV AH, 9
INT 21H
CALL LINEBREAK

MOV CX, COUNT
INPUT:
  CALL INDEC
  MOV [SI], AL
  INC SI
  CALL LINEBREAK
  LOOP INPUT

CALL LINEBREAK

MOV SI,601H
CALL LINEBREAK
LEA DX, PROMPT4
MOV AH, 9
INT 21H
CALL LINEBREAK

MOV CX, COUNT
INPUT2:
  CALL INDEC
  MOV [SI], AL
  INC SI
  CALL LINEBREAK
  LOOP INPUT2

CALL LINEBREAK

LEA DX, PROMPT3
MOV AH, 9
INT 21H

MOV CX, COUNT
MOV SI, 601H
MOV DI, 501H

MULDIV:
   MOV AL, [DI]
   MOV BL, [SI]
   MUL BL
   MOV [DI], AL
   INC SI
   INC DI
   LOOP MULDIV

MOV CX,COUNT
```

```
    MOV SI, 501H
    MOV AH,2
    OUTPUT:
      MOV AH, 0
      MOV AL,[SI]
      CALL OUTDEC
      INC SI
      CALL LINEBREAK
     LOOP OUTPUT

    MOV AH, 4CH
    INT 21H
  MAIN ENDP
    ; FOLDER PATH
    INCLUDE C: \Codes\INDEC.ASM
    INCLUDE C:\Codes\OUTDEC.ASM
    INCLUDE C:\Codes\LINEBREAK.ASM


END MAIN
```

3. Write a procedure FIND_IJ that returns the offset address of the element in row i and column j in a two-dimensional M x N word array A stored in row-major order. The procedure receives i in AX, j in BX, N in CX, and the offset of A in DX. It returns the offset address of the element in DX.

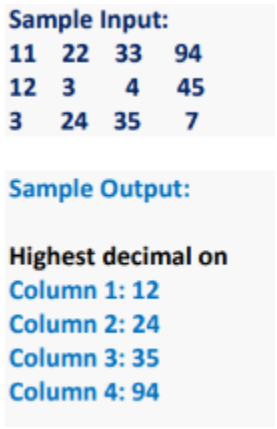
   CODE :

```
.MODEL SMALL

 .STACK 100H

 .DATA
   PROMPT1 DW 'ROW(M): $'
   ROW DW ?
   PROMPT2 DW 'COLUMN(N): $'
   COLUMN DW ?
   PROMPT3 DW 'I: $'
   ROWINDEX DW ?
   PROMPT4  DW  'J: $'
   COLINDEX DW ?
   SIZE DW 2
   PROMPT5  DW  'OFFSET ADDRESS : $'
   ARRAY   DW  50 DUP(?)

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

```
MOV DS, AX

LEA DX, PROMPT1
MOV AH, 9
INT 21H

CALL INDEC
MOV ROW, AX
CALL LINEBREAK

LEA DX, PROMPT2
MOV AH, 9
INT 21H

CALL INDEC
MOV COLUMN, AX
CALL LINEBREAK

LEA SI, ARRAY
MOV CX, ROW


@L_1:
  MOV BX, COLUMN

  @L_2:
    CALL INDEC
    MOV [SI], AX
    CALL LINEBREAK
    ADD SI, 2
    DEC BX
    JNZ @L_2

  CALL LINEBREAK
  LOOP @L_1


;TAKING I, J
LEA DX, PROMPT3
MOV AH, 9
INT 21H
CALL INDEC
MOV ROWINDEX, AX
CALL LINEBREAK

LEA DX, PROMPT4
MOV AH, 9
INT 21H
CALL INDEC
MOV COLINDEX, AX
```

```
CALL LINEBREAK

;FIND_IJ
MOV AX, ROWINDEX
MOV BX, COLINDEX
MOV CX, COLUMN
LEA SI, ARRAY
MOV DX, SI
CALL FIND_IJ

MOV SI,DX

LEA DX, PROMPT5
MOV AH, 9
INT 21H

MOV AX,SI
CALL OUTDEC

MOV AH, 4CH
INT 21H

MAIN ENDP

  ; FOLDER PATH
    INCLUDE C: \Codes\INDEC.ASM
    INCLUDE C:\Codes\OUTDEC.ASM
    INCLUDE C:\Codes\LINEBREAK.ASM
   INCLUDE C:\Codes\FIND_IJ.ASM

END MAIN
```

4. Write an assembly language program which load 12 decimal numbers in a 3 x 4 word array in row-major order from user input.
   a) Display the Matrix
   b) Display the highest decimal on each column




```
.MODEL SMALL

.STACK 100H

.DATA
   PROMPT1 DB  'Highest decimal on',0DH,0AH,'$'
   PROMPT2 DB  'Column $'
   COLNO DB 49
   PROMPT3 DB ' : $'
   TEMP    DW  0
   COUNT   DW  0
   SIZE1   DW  2
   SIZE2   DW  8
   MIN     DW  0
   ARRAY   DW  4 DUP(?)
           DW  4 DUP(?)
           DW  4 DUP(?)


.CODE
   MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    LEA SI, ARRAY
    MOV CX, 3

   ROW_INPUT:
      MOV BX, 4
```

```
    READ:
        CALL INDEC
        MOV [SI], AX
        CALL LINEBREAK
        ADD SI, 2
        DEC BX
        JNZ READ

    CALL LINEBREAK
    LOOP ROW_INPUT

; SHOW OUTPUT MATRIX
LEA SI, ARRAY
MOV CX, 3

ROW_OUTPUT:
    MOV BX, 4

    PRINT:
        MOV AX, [SI]
        CALL OUTDEC
        MOV DL, ' '
        MOV AH, 2
        INT 21H
        ADD SI, 2
        DEC BX

    JNZ PRINT
    CALL LINEBREAK
    LOOP ROW_OUTPUT

; LARGEST NUMBER
CALL LINEBREAK

LEA DX, PROMPT1
MOV AH, 9
INT 21H

LEA SI, ARRAY
MOV TEMP, SI
MOV CX, 4

COMPARE:
    MOV SI, TEMP
    MOV AX, CX
    NEG AX
    ADD AX, 4
    MUL SIZE1
    MOV COUNT, AX
```

```
    ADD SI, COUNT
    MOV BX, 3

    SET_MAX:
        MOV AX, [SI]
        CMP AX, MIN
        JG  IF
        JMP ELSE

        IF:
            MOV MIN, AX
            JMP ELSE

        ELSE:
            ADD SI, SIZE2
            DEC BX

    JNZ SET_MAX

LEA DX, PROMPT2
MOV AH, 9
INT 21H

MOV AH, 2
MOV DL, COLNO
INT 21H
INC COLNO

LEA DX, PROMPT3
MOV AH, 9
INT 21H
MOV AX, MIN

CALL OUTDEC
MOV MIN, 0

CALL LINEBREAK
LOOP COMPARE
MOV AH, 4CH
INT 21H


MAIN ENDP
        ; FOLDER PATH
            INCLUDE C: \Codes\INDEC.ASM
            INCLUDE C:\Codes\OUTDEC.ASM
           INCLUDE C:\Codes\LINEBREAK.ASM

END MAIN
```

- EXTRA CODES

  - ***2-D ARRAY_TRAVAERSE***

```
.MODEL SMALL
.STACK 100H
.DATA
   PROMPT_1  DB  'The contents of the array:',0DH,0AH,'$'

   ARRAY_DATA_SIZE EQU 2    ;1: for Byte Variable , 2: for Word Variable
   ;following ARRAY variable is a 3x4 array
   Row_Count EQU 3
   Column_Count EQU 4

   ARRAY   DW  1,2,3,4
           DW  5,6,7,8
           DW  9,10,11,12
   ;for calculating row and column memory displacement form array base address
   Row_Adjustment DW ?
   Column_Adjustment DW ?

   ;for Loop control
   R   DW 1    ;loop control variable for traversing each Row,initially 1
   C   DW 1    ;loop control variable for traversing each Column,initially 1

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

    LEA DX, PROMPT_1            ; load and display the string PROMPT_1
    MOV AH, 9
    INT 21H

    ; PRINT_ARRAY Procedure Requires no Input setting
    CALL PRINT_2D_ARRAY         ; call the procedure PRINT_ARRAY

    ; return control to DOS
    MOV AH, 4CH
    INT 21H
  MAIN ENDP



 PRINT_2D_ARRAY PROC
   ; this procedure will print the elements of a given array
   ; input : none
   ; output : none
```

```
; Preserve the Contents
PUSH AX                    ; push AX onto the STACK
PUSH CX                    ; push CX onto the STACK
PUSH DX                    ; push DX onto the STACK
PUSH SI                   ; push SI onto the STACK
PUSH BX                    ; push BX onto the STACK

MOV R,1
@PRINT_ROW:                ; loop label

 PUSH R     ;Preserve the Current Row index into a STACK,
         ;now R can be use locally here
 ;update Displacement for current Row
 DEC R          ;R=R-1
 MOV AX,Column_Count
 MUL R       ; AX=(R-1)*Column_Count
 MOV R,AX   ;Preserve the result AX=(R-1)*Column_Count in R
 MOV AX,ARRAY_DATA_SIZE
 MUL R      ;(R-1)*Column_Count*ARRAY_DATA_SIZE
 MOV Row_Adjustment,AX

 MOV SI,Row_Adjustment ;For Based and Indexed Addressing Mode
 ;Load the Preserved Row index (R) from STACK after local use
 POP R

 MOV C,1
 @PRINT_COLUMN:

   PUSH C     ;Preserve the Current Column index into a STACK,
           ;now C can be use locally here
   ;update Displacement for current Column
   DEC C          ;C=C-1
   MOV AX,ARRAY_DATA_SIZE
   MUL C      ;(C-1)*ARRAY_DATA_SIZE
   MOV Column_Adjustment,AX

   MOV BX,Column_Adjustment ;For Based and Indexed Addressing Mode
   ;Load the Preserved Row index (R) from STACK after local use
   POP C
   XOR AX,AX
   ;SI=Row_Adjustment
   ;BX=Column_Adjustment
   MOV AX, ARRAY[SI][BX]       ; Based and Indexed Addressing MODE
   CALL OUTDEC                ; call the procedure OUTDEC

   ; print a Space
   MOV AH, 2                  ; set output function
   MOV DL, 20H                  ; set DL=20H
   INT 21H                   ; print a character
```

```
    CMP C,Column_Count    ;@PRINT_COLUMN Loop Continues as Column Times
    JE Next_Row_
    INC C              ;Increment the Column index by 1
  JMP @PRINT_COLUMN
  Next_Row_:
    ;PRINT_NEW_LINE
    MOV AH, 2                ; set output function
    MOV DL, 0DH                ;
    INT 21H                  ; print a NewLine
    MOV AH, 2                ; set output function
    MOV DL, 0AH                ;
    INT 21H                  ; print a NewLine

    CMP R,Row_Count
    JE Procedure_Exit_
    INC R              ;Increment the Row index
 JMP @PRINT_ROW            ; jump to label @PRINT_ROW while CX!=0

 Procedure_Exit_:
 ; Load the Preserved Contents
 POP BX                   ; push BX onto the STACK
 POP SI                   ; push SI onto the STACK
 POP DX                   ; pop a value from STACK into DX
 POP CX                   ; pop a value from STACK into CX
 POP AX                   ; pop a value from STACK into AX
 ; return control to the calling procedure
 RET
PRINT_2D_ARRAY ENDP

INCLUDE C:\TEST\OUTDEC.ASM
INCLUDE C:\TEST\INDEC.ASM

END MAIN
```

- ***SEARCH 'N' IN STRING***

```
.MODEL SMALL
.STACK 100H
.DATA

STRING DW 'Hello$'
Char_      DW 'o'
Prompt_found DB 'Found at Position: String + $'
String_Len DW 5
STRING_Data_type DB 2 ; for Word array

.CODE
MAIN PROC
```

```
    MOV AX, @DATA
    MOV DS, AX

    ; check if the char is found in string
    ; or not
    CALL Search_Char

    ;interrupt to exit
    MOV AH, 4CH
    INT 21H
MAIN ENDP

Search_Char PROC

    PUSH AX
    PUSH BX
    PUSH SI
    ; load the starting address
    ; of the string
    ;MOV SI,OFFSET STRING
    LEA SI,STRING
    ; traverse to the end of the String;
    ; and find the Char_
    MOV BX,0 ;initally BX=0,if Found_ set BX=1
    MOV CX,0 ;loop index
    LOOP1:
       XOR AH,AH
            MOV AX, [SI]
            CMP AL, BYTE PTR Char_
            JNE Continue_
               ;Since Found_, so set BX=1
               MOV BX,1
               ;;Calculate the found_index = CX * STRING_Data_type
               MOV AL,STRING_Data_type
               MUL CL   ;Calculate found index = CX * STRING_Data_type
               PUSH AX ;Actually no need to push/preserve found index CX here
                     ;but done for extra safety
               JMP Exit_Loop1
            Continue_:
               INC SI    ; to point the next byte in string
               INC CX    ;loop index increment
               CMP CX,String_Len
               JE Exit_Procedure
          JMP LOOP1
    ;;
    Exit_Loop1:
    CMP BX,1
    JNE Exit_Procedure
      ;if Char found then
      POP CX
```

```
      ;print prompt
      MOV AH,9
      LEA DX,Prompt_found
      INT 21h
      ;print found index
      MOV AX,CX
      CALL OUTDEC
   Exit_Procedure:
   POP SI
   POP BX
   POP AX

         RET
Search_Char ENDP

INCLUDE C:\TEST\OUTDEC.ASM
;INCLUDE C:\TEST\INDEC.ASM

END MAIN
```

- ***SORT***

```
.MODEL SMALL
 .STACK 100H

 .DATA
   PROMPT_1  DB  'The contents of the array before sorting : $'
   PROMPT_2  DB  0DH,0AH,'The contents of the array after sorting : $'

   ARRAY   DB  5,3,9,0,2,6,1,7,8,4

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

    MOV BX, 10                ; set BX=10

    LEA DX, PROMPT_1           ; load and display the string PROMPT_1
    MOV AH, 9
    INT 21H

    LEA SI, ARRAY              ; set SI=offset address of ARRAY

    CALL PRINT_ARRAY           ; call the procedure PRINT_ARRAY

    LEA SI, ARRAY              ; set SI=offset address of the ARRAY

    CALL SELECT_SORT           ; call the procedure SELECT_SORT
```

```
  LEA DX, PROMPT_2          ; load and display the string PROMPT_2
  MOV AH, 9
  INT 21H

  LEA SI, ARRAY             ; set SI=offset address of ARRAY

  CALL PRINT_ARRAY          ; call the procedure PRINT_ARRAY

  MOV AH, 4CH               ; return control to DOS
  INT 21H
MAIN ENDP



PRINT_ARRAY PROC
 ; this procedure will print the elements of a given array
 ; input : SI=offset address of the array
 ;       : BX=size of the array
 ; output : none

 PUSH AX                    ; push AX onto the STACK
 PUSH CX                    ; push CX onto the STACK
 PUSH DX                    ; push DX onto the STACK

 MOV CX, BX                 ; set CX=BX

 @PRINT_ARRAY:              ; loop label
  XOR AH, AH                ; clear AH
  MOV AL, [SI]              ; set AL=[SI]

  CALL OUTDEC               ; call the procedure OUTDEC

  MOV AH, 2                 ; set output function
  MOV DL, 20H               ; set DL=20H
  INT 21H                   ; print a character

  INC SI                    ; set SI=SI+1
 LOOP @PRINT_ARRAY          ; jump to label @PRINT_ARRAY while CX!=0

 POP DX                     ; pop a value from STACK into DX
 POP CX                     ; pop a value from STACK into CX
 POP AX                     ; pop a value from STACK into AX

 RET                        ; return control to the calling procedure
PRINT_ARRAY ENDP



SELECT_SORT PROC
```

```
; this procedure will sort the array in ascending order
; input : SI=offset address of the array
;       : BX=array size
; output :none

PUSH AX                  ; push AX onto the STACK
PUSH BX                  ; push BX onto the STACK
PUSH CX                  ; push CX onto the STACK
PUSH DX                  ; push DX onto the STACK
PUSH DI                  ; push DI onto the STACK

CMP BX, 1                ; compare BX with 1
JLE @SKIP_SORTING            ; jump to label @SKIP_SORTING if BX<=1

DEC BX                   ; set BX=BX-1
MOV CX, BX                 ; set CX=BX
MOV AX, SI                 ; set AX=SI

@OUTER_LOOP:                 ; loop label
  MOV BX, CX               ; set BX=CX
  MOV SI, AX               ; set SI=AX
  MOV DI, AX               ; set DI=AX
  MOV DL, [DI]              ; set DL=[DI]

  @INNER_LOOP:               ; loop label
    INC SI                 ; set SI=SI+1

    CMP [SI], DL           ; compare [SI] with DL
    JNG @SKIP                ; jump to label @SKIP if [SI]<=DL

    MOV DI, SI             ; set DI=SI
    MOV DL, [DI]            ; set DL=[DI]

    @SKIP:                 ; jump label

    DEC BX                 ; set BX=BX-1
  JNZ @INNER_LOOP            ; jump to label @INNER_LOOP if BX!=0

  MOV DL, [SI]             ; set DL=[SI]
  XCHG DL, [DI]             ; set DL=[DI] , [DI]=DL
  MOV [SI], DL             ; set [SI]=DL

LOOP @OUTER_LOOP             ; jump to label @OUTER_LOOP while CX!=0

@SKIP_SORTING:               ; jump label

POP DI                   ; pop a value from STACK into DI
POP DX                   ; pop a value from STACK into DX
POP CX                   ; pop a value from STACK into CX
POP BX                   ; pop a value from STACK into BX
```

```
  POP AX                    ; pop a value from STACK into AX

  RET                       ; return control to the calling procedure
SELECT_SORT ENDP



OUTDEC PROC
  ; this procedure will display a decimal number
  ; input : AX
  ; output : none

  PUSH BX                   ; push BX onto the STACK
  PUSH CX                   ; push CX onto the STACK
  PUSH DX                   ; push DX onto the STACK

  XOR CX, CX                ; clear CX
  MOV BX, 10                ; set BX=10

  @OUTPUT:                  ; loop label
    XOR DX, DX              ; clear DX
    DIV BX                  ; divide AX by BX
    PUSH DX                 ; push DX onto the STACK
    INC CX                  ; increment CX
    OR AX, AX               ; take OR of Ax with AX
  JNE @OUTPUT               ; jump to label @OUTPUT if ZF=0

  MOV AH, 2                 ; set output function

  @DISPLAY:                 ; loop label
    POP DX                  ; pop a value from STACK to DX
    OR DL, 30H              ; convert decimal to ascii code
    INT 21H                 ; print a character
  LOOP @DISPLAY             ; jump to label @DISPLAY if CX!=0

  POP DX                    ; pop a value from STACK into DX
  POP CX                    ; pop a value from STACK into CX
  POP BX                    ; pop a value from STACK into BX

  RET                       ; return control to the calling procedure
OUTDEC ENDP



END MAIN
```

- ***Take input and search N in String***

```
.MODEL SMALL
.STACK 100H
.DATA

STRING DW 'Hello$'
Char_       DW 'o'
Prompt_found DB 'Found at Position: String + $'
String_Len DW 5
STRING_Data_type DB 2 ; for Word array

.CODE
MAIN PROC
   MOV AX, @DATA
   MOV DS, AX    ;Initalize the Data Segment
   MOV ES, AX    ;and Extra Segment

   ; check if the char is found in string
   ; or not
   CLD       ;Clear Direction Flag, Now SI and DI moves in increasing address
   LEA DI,STRING
   MOV AL,BYTE PTR Char_
   MOV CX,String_Len
   REPNE SCASB      ;scan Char_ in a byte of STRING
   JNZ Exit_

      ;Since Found_,
            MOV AH,9
      LEA DX,Prompt_found
      INT 21h
      ;print : found_index in string
      SUB String_Len,CX
      DEC String_Len ; String_Len holds the Found_index since 1st element is String+0, so deduct 1
      MOV AX,String_Len ;multiply by the STRING_Data_type
      MUL STRING_Data_type
      ;AX hold the product
      CALL OUTDEC

   ;interrupt to exit
   Exit_:
   MOV AH, 4CH
   INT 21H
MAIN ENDP

INCLUDE C:\TEST\OUTDEC.ASM
;INCLUDE C:\TEST\INDEC.ASM
```

```
END MAIN
```

- ***Translate characters in a given string using a translation table and display the translated string***

```
.MODEL SMALL
.STACK 100H
.DATA

table DB  48 DUP (' '), "0123456789", 7 DUP (' ')
     DB  "abcdefghijklmnopqrstuvwxyz", 6 DUP (' ')
     DB  "abcdefghijklmnopqrstuvwxyz", 133 DUP (' ')

string1 DB "THIS IS#1,One"
strLength DB 23

.CODE

   MAIN PROC

      MOV AX,@DATA       ;Initialize Data Segment
      MOV DS,AX

      mov  cx, WORD PTR strLength ; string length
      lea  bx, table   ; addr of translation table
      lea  si, string1  ; address of string
      lea  di, string1  ; destination also string

      mov ah,2          ; INT 21h; function for display
    forIndex:
      lodsb             ; copy next character to AL
      xlat              ; translate character
      mov dl,al
      int 21h           ;display char
      stosb             ; copy character back into string
    loop forIndex       ; repeat for all characters

    exit_:
      MOV AH,4CH        ;return DOS
      INT 21H

   MAIN ENDP

END MAIN
```

- ***1-D ARRAY***

```
.MODEL SMALL

.STACK 100H

.DATA

PROMPT_1 DB 'Enter the Array elements :',0DH,0AH,'$'

PROMPT_2 DB 'The Array elements are : $'

ARRAY DW 10 DUP(0)

.CODE

MAIN PROC

MOV AX, @DATA

MOV DS, AX

MOV BX, 10

LEA DX, PROMPT_1

MOV AH, 9

INT 21H

LEA SI, ARRAY

CALL IN1DARRAY

LEA DX, PROMPT_2

MOV AH, 9

INT 21H

LEA SI, ARRAY

CALL OUT1DARRAY
```

```
MOV AH, 4CH

INT 21H

MAIN ENDP

INCLUDE D:\Codes\IN1DARRAY.ASM

INCLUDE D:\Codes\OUT1DARRAY.ASM

END MAIN
```

- ***TAKE INPUT AND DISPLAY (2-D ARRAY)***

```
.MODEL SMALL
.STACK 100H

.DATA
ARRAY DB 3,3 DUP(0)   ; Define a 3x3 array of bytes
PROMPT1 DB 'Enter array elements:', 0AH,0DH,'$'
PROMPT2 DB 'The array elements are:', 0AH,0DH,'$'
NEWLINE DB 0AH,0DH,'$'

.CODE
MAIN PROC

   MOV AX, @DATA   ; Initialize data segment
   MOV DS, AX

   LEA DX, PROMPT1 ; Display prompt for input
   MOV AH, 9
   INT 21H

   ; Take input for the array
   MOV CX, 3 ; Number of rows
   MOV DX, 3 ; Number of columns
   LEA DI, ARRAY ; Point to start of array
   FOR_ROW:
      FOR_COL:
         MOV AH, 1 ; Read a character
         INT 21H
         SUB AL, 30H ; Convert from ASCII to decimal
         MOV [DI], AL ; Store in array
         INC DI ; Move to next element
      LOOP FOR_COL
      ADD DI, 3 ; Move to start of next row
   LOOP FOR_ROW
```

```
    ; Display the contents of the array
    LEA DX, PROMPT2 ; Display prompt for output
    MOV AH, 9
    INT 21H

    LEA DI, ARRAY ; Point to start of array
    FOR_ROW2:
        FOR_COL2:
            MOV AL, [DI] ; Load element from array
            ADD AL, 30H ; Convert from decimal to ASCII
            MOV AH, 2 ; Display character
            INT 21H
            INC DI ; Move to next element
        LOOP FOR_COL2
        LEA DX, NEWLINE ; Display newline character
        MOV AH, 9
        INT 21H
    LOOP FOR_ROW2

    MOV AH, 4CH     ; Return control to DOS
    INT 21H

MAIN ENDP
END MAIN
```

#(without input)

```
.MODEL SMALL
.STACK 100H
.DATA
    ROWS DB 3
    COLS DB 3
    TWO_D_ARRAY DB 3, 3, 1, 2, 3, 4, 5, 6, 7, 8, 9
    PROMPT DB '2D Array:',13,10,'$'
.CODE
    MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        LEA DX, PROMPT
        MOV AH, 9
        INT 21H

        MOV CL, COLS ; initialize loop counter for columns
        MOV BX, 0 ; initialize index into array
      OUTER_LOOP:
        PUSH CX ; save column counter value
```

```
    MOV CL, COLS ; initialize loop counter for rows
   INNER_LOOP:
    MOV AL, TWO_D_ARRAY[BX] ; load value from array
    ADD AL, 30H ; convert to ASCII
    MOV DL, AL ; prepare to display
    MOV AH, 2
    INT 21H

    INC BX ; increment array index
    DEC CL ; decrement row counter
    JNZ INNER_LOOP ; repeat for all rows

    POP CX ; restore column counter value
    DEC CX ; decrement column counter
    JNZ OUTER_LOOP ; repeat for all columns

    MOV AH, 4CH
    INT 21H
  MAIN ENDP
END MAIN
```

- ***Take input in 1-d array and display***

```
.MODEL SMALL
.STACK 100H
.DATA
  ARR DW 10 DUP(0)
  MSG1 DB 'Enter 10 integer values:',13,10,'$'
  MSG2 DB 'The values entered are:',13,10,'$'
.CODE
  MAIN PROC
    MOV AX, @DATA
    MOV DS, AX

    LEA DX, MSG1
    MOV AH, 9
    INT 21H

    LEA SI, ARR
    MOV CX, 10
    MOV BX, 2   ; set offset to 2 since DW is 2 bytes
    MOV AH, 1   ; set input function to read character
  READ_LOOP:
```

```
    INT 21H
    CMP AL, 0DH ; check if carriage return is entered
    JE READ_DONE
    SUB AL, '0' ; convert character to integer
    MOV [SI+BX], AL ; store the value in the array
    INC BX
    LOOP READ_LOOP

  READ_DONE:
    LEA DX, MSG2
    MOV AH, 9
    INT 21H

    LEA SI, ARR
    MOV CX, 10
    MOV BX, 2   ; set offset to 2 since DW is 2 bytes
    MOV AH, 2   ; set output function to display character
  DISP_LOOP:
    MOV DL, [SI+BX] ; load the value from the array
    ADD DL, '0' ; convert integer to character
    INT 21H
    MOV DL, ',' ; separate values with comma
    INT 21H
    ADD BX, 2 ; move offset to next element
    LOOP DISP_LOOP

    MOV AH, 4CH ; return control to DOS
    INT 21H

  MAIN ENDP
END MAIN
```

#(without input)

```
.MODEL SMALL
.STACK 100H

.DATA
    arr DB 1, 2, 3, 4, 5 ; declare the array

.CODE
    MAIN PROC
        MOV AX, @DATA   ; set up data segment
        MOV DS, AX
```

```
    MOV CX, 5       ; set counter to the number of elements in the array
    LEA SI, arr     ; load the starting address of the array into SI

  displayLoop:
    MOV DL, [SI]    ; move the value of the array element into DL
    ADD DL, 30h     ; convert the value to ASCII
    MOV AH, 02h     ; display the character in DL
    INT 21h
    INC SI          ; increment SI to point to the next element in the array
    LOOP displayLoop ; loop until all elements have been displayed

    MOV AH, 4Ch     ; exit program
    INT 21h
  MAIN ENDP

END MAIN
```

- ***fibonacci series n-terms input and display n-terms***

```
.MODEL SMALL
.STACK 100h

.DATA
  message DB 'Enter the number of terms:', 0AH, 0DH, '$'
  fib_msg DB 'The Fibonacci sequence is:', 0AH, 0DH, '$'
  fib DB 20 DUP (0)

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

    ; Display message to enter number of terms
    LEA DX, message
    MOV AH, 09h
    INT 21h

    ; Read input value of n
    MOV AH, 01h
    INT 21h
    SUB AL, 30h ; Convert ASCII digit to integer
    MOV BL, AL ; Store n in BL for later use

    ; Initialize the Fibonacci sequence
    MOV CX, 0 ; Current term
    MOV AX, 0 ; First term
```

```
  MOV BX, 1 ; Second term
  MOV [fib+CX], AX ; Store first term in array
  INC CX
  MOV [fib+CX], BX ; Store second term in array
  INC CX

  ; Generate the Fibonacci sequence up to the nth term
  FIB_LOOP:
    CMP CX, BL ; Check if we have generated enough terms
    JGE FIB_DONE

    MOV DX, [fib+CX-2] ; Get the previous term
    ADD DX, [fib+CX-1] ; Add it to the current term
    MOV [fib+CX], DX ; Store the sum in the array
    INC CX
    JMP FIB_LOOP

  FIB_DONE:
    ; Display the Fibonacci sequence
    LEA DX, fib_msg
    MOV AH, 09h
    INT 21h

    MOV CX, BL ; Set loop counter to n
    MOV SI, 0 ; Set array index to 0
    FIB_DISPLAY_LOOP:
      MOV DX, [fib+SI] ; Get the current term from the array
      CALL DISPLAY_NUMBER ; Display the current term
      INC SI ; Increment array index
      LOOP FIB_DISPLAY_LOOP

    ; Terminate program
    MOV AH, 4Ch
    INT 21h

MAIN ENDP

; Display a number in decimal format
; Input: DX = number to display
DISPLAY_NUMBER PROC
  MOV BX, 10 ; Set divisor to 10
  XOR CX, CX ; Set digit count to 0
  DISPLAY_LOOP:
    XOR DX, DX ; Clear high word of DX
    DIV BX ; Divide DX:AX by BX
    PUSH DX ; Save remainder on stack
    INC CX ; Increment digit count
    TEST AX, AX ; Check if quotient is zero
    JNZ DISPLAY_LOOP
  DISPLAY_LOOP2:
```

```
    POP DX ; Get remainder from stack
    ADD DL, 30h ; Convert to ASCII digit
    MOV AH, 02h ; Set display function
    INT 21h ; Display digit
    LOOP DISPLAY_LOOP2
  MOV DL, ' ' ; Display a space after the number
  MOV AH, 02h ; Set display function
  INT 21h ; Display space
  RET
DISPLAY_NUMBER ENDP

END MAIN
```

- ***(#using array)***

```
.MODEL SMALL
.STACK 100H

.DATA
  arr DW 12 DUP(?)
  n DW 12    ; number of terms to generate
  index DW ?
  term DW ?
  i DW 02h
  j DW 0
  k DW 1

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

    ; get the number of terms to generate
    MOV AH, 01h
    INT 21h
    SUB AL, 30h   ; convert character to number
    MOV BL, AL
    MOV CX, 0
    MOV AX, 0

  ; initialize the first two terms of the series
  MOV arr[0], AX
  MOV arr[2], CX
  MOV arr[4], CX
  MOV arr[6], CX
```

```
    MOV arr[8], CX
    MOV arr[10], CX
    MOV arr[12], CX
    MOV arr[14], CX
    MOV arr[16], CX
    MOV arr[18], CX
    MOV arr[20], CX

    ; generate the Fibonacci series and store in the array
    MOV DX, OFFSET arr
    MOV SI, 2
    MOV BX, 1

    L1:
        MOV AX, i
        ADD AX, j
        MOV k, AX
        MOV AX, k
        MOV term, AX
        MOV index, SI
        MOV [DX][index], term
        ADD SI, 2
        MOV AX, j
        MOV i, AX
        MOV AX, k
        MOV j, AX
        DEC BL
        JNZ L1

    ; display the Fibonacci series stored in the array
    MOV DX, OFFSET arr
    MOV SI, 0
    MOV CX, n

    L2:
        MOV AX, [DX][SI]
        CALL DISPLAY_NUM
        ADD SI, 2
        LOOP L2

    MOV AH, 4Ch
    INT 21h
    MAIN ENDP

; display a number on the screen
```

```
DISPLAY_NUM PROC
   PUSH AX
   PUSH BX
   PUSH CX
   PUSH DX

   XOR CX, CX
   MOV BX, 10
   MOV AX, 0
   MOV DX, 0
   DIV BX
   PUSH DX
   INC CX

   L1:
      XOR DX, DX
      DIV BX
      PUSH DX
      INC CX
      CMP AX, 0
      JNZ L1

   L2:
      POP DX
      OR DL, 30h
      MOV AH, 02h
      INT 21h
      LOOP L2

   POP DX
   POP CX
   POP BX
   POP AX
   RET
DISPLAY_NUM ENDP

END MAIN
```

PREVIOUS QUESTIONS AND ANSWERS

- In the microprocessor, what is the function of IP and ALU?
  IP : contains the address of the next instruction to be executed by the EU.
  ALU : perform arithmetic & logic operation

- What determines whether a microprocessor is considered an 8 bit, a 16 bit, or a 32 bit device?
  The number of bits a microprocessor's ALU can work with at a time determines whether the microprocessor is considered an 8-bit, 16-bit or 2-bit system.
  Or,
  The number of bits in a microprocessor's data bus determines its classification as an 8-bit, 16-bit, or 32-bit device, with wider data buses allowing for more data to be transferred at once. The width of the data bus also affects the microprocessor's ability to access memory and execute instructions. Wider data buses generally lead to increased processing power and the ability to handle more complex tasks.

- What types of programs are usually written in assembly language?
  Programs which require a lot of hardware control or programs which must run as quickly as possible are usually best written in assembly language.

- Consider a machine language instruction that moves a copy of the contents of register BX in the CPU to a memory word. What happens during it?
  i. the fetch cycle?
  ii. the execution cycle?
  i) • Fetch an instruction from memory.
  • Decode the instruction to determine the operation.
  ii) • Perform the operation if needed.
  • Store the results in memory

- If the stack segment register contains 56C0H and the stack pointer register contains 1358H, what is the physical address of the top of the stack?
  57F58

- Write the 8086 instruction which will perform the indicated operation
  i. Divide DL by BL
  MOV AL, DL

DIV BL

ii. Set the most significant bit of AX to a 1, and invert the lower 4 bits, but do not affect the other bits

```
OR AX,80H
XOR AX,0FH
```

- Perform the indicated operation on the following number
  0111 1001 BCD + 1001 0101 BCD = 0001 0110 0001 0100 BCD

- In order to avoid hand keying programs into an SDK-86 board, we wrote a program to send machine code programs from IBM PC to an SDK-86 board through a serial link. As part of this program, we had to convert each byte of the machine code program to ASCII codes for the two nibbles in the byte. In other word, a byte of 7AH has to be sent as 37H, the ASCII code for 7, and 41H, the ASCII code for A. Once you separate the nibble of the byte, this conversion is a simple IF-THEN-ELSE situation. Write an assembly language program section which does the needed conversion.

  CODE :

```
.MODEL SMALL

.DATA
   num db ?
   num1 db ?
   num2 db ?

.CODE
   MAIN PROC

   MOV AX,@DATA
   MOV DS,AX

   MOV AL,7BH
   MOV num1,AL
   MOV num2,AL

   MOV BL,num1
   AND BL,11110000b
```

```
    MOV CL,4
    SHR BL,CL
    MOV CX,2
begin_:
    CMP BL,9d
    JL digit_
        cmp BL,0Fh
        JG print_
        ADD BL,37h
        JMP print_
    digit_:
        OR BL,00110000b
    print_:
        MOV AH,2
        MOV DL,BL
        INT 21h

        MOV BL,num2
        AND BL,00001111b
    loop begin_

    ;dos return
    MOV AH,4CH
    INT 21H

    MAIN ENDP

END MAIN
```

- Write one or more instructions to multiply the content of DL by 16D without using MUL or IMUL instruction’

```
.MODEL SMALL

.DATA

.CODE
    MAIN PROC

    MOV AX,@DATA
    MOV DS,AX


    MOV DL,00000110b
    MOV CL,4
    SHL DL,CL
```

```
    ;dos return

    MOV AH,4CH
    INT 21H

    MAIN ENDP

END MAIN
```

- Discuss the result of following instruction in brief.
  i. MOV AX , [DX]

- How do we detect when a signed overflow does occur? Discuss three effective methods in brief.

- Write an assembly language program section which puts the product of the first 6 terms of the arithmetic sequence 1, 3, 5, 7 … in DX. Use a looping structure to produce the product

```
.MODEL SMALL

.DATA
    product dw 1
    num     dw 1

.CODE
    MAIN PROC

    MOV AX,@DATA
    MOV DS,AX

    MOV CX,5
LOOP_:
    MOV AX,product
    ADD num,2
    MOV DX,num
    MUL DX
    MOV product, AX
    loop Loop_

    ;dos return

    MOV AH,4CH
    INT 21H
```

```
MAIN ENDP

END MAIN
```

- Write an assembly language program section which clears ZF if DX contains an even number.

```
.MODEL SMALL

.DATA
   product dw 1
   num     dw 1

.CODE
   MAIN PROC

   MOV AX,@DATA
   MOV DS,AX

   MOV BX,5d
   AND BX,0001h
   JNZ exit_


   exit_:
   ;dos return
   MOV AH,4CH
   INT 21H

   MAIN ENDP

END MAIN
```

Table of Contents

Interrupt system **1**
Interrupts in 8086 ..... **2**
Hardware Interrupts 2
Software Interrupts 3
Interrupts in 8085 ..... **4**
Types of Interrupts in 8085 4
Instruction for Interrupts 6
Uses of Interrupts in 8085 microprocessor 7
Issues of Interrupts in 8085 microprocessor 7
Exercises from Book (Marut) **7**
Programming Exercises from Book (Marut) **10**
Other Important Questions (Quiz, Exams, Assignments) **16**

Interrupt system

An interrupt is a condition that halts the microprocessor temporarily to work on a different task and then return to its previous task. Interrupt is an event or signal that requests the attention of the CPU. This halt allows peripheral devices to access the microprocessor.

Interrupt can work for both 8085 and 8086

If we think of a system without any interrupt -
There will be a driver that will be dedicated for a system or hardware (let's say keyboard), then while performing other tasks, it will also go to the driver dedicated to the keyboard and will pull from the keyboard to check if any key is pressed or not.
Disadvantages of this system can be -

- inefficient
- unnecessary pulling (if there is no input but the driver still pulls for input)
- Suppose there is an input device that prepares the inputs every 10 milliseconds but the system's driver pulls from that input device after every 20 milliseconds , at this situation the whole system becomes slow

To handle an interrupt there is an interrupt controller that can be an internal or an external device.

To handle an interrupt there is a routine that is called Interrupt Service Routine. When the service routine is triggered then the microprocessor is interrupted. Then the microprocessor stops its regular task and serves the interrupt service routine . Interrupt service routine is generally a set of functions.

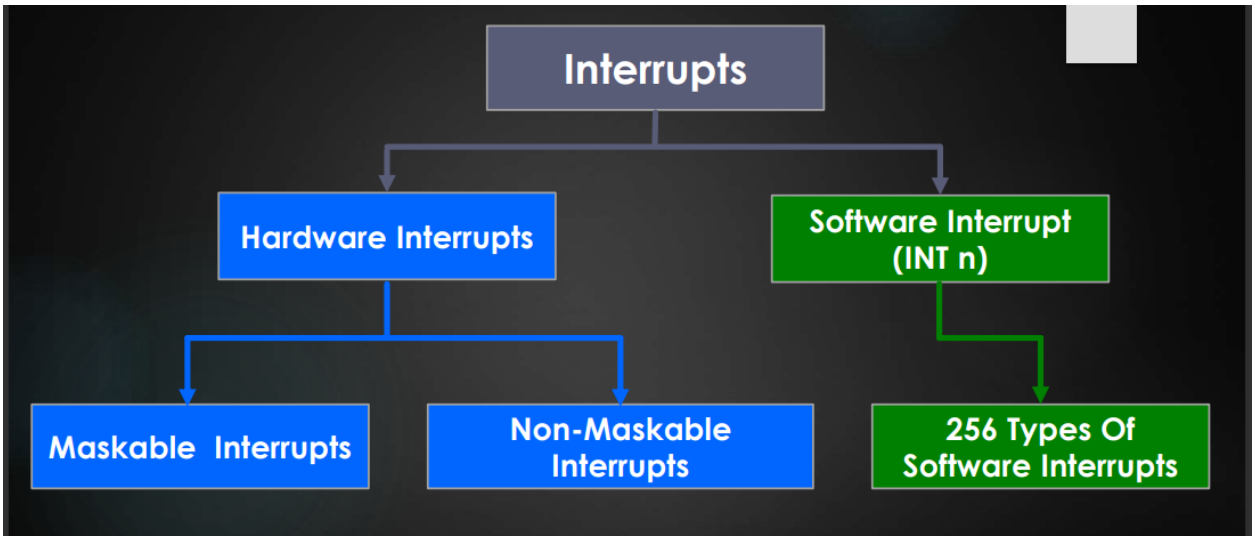
The processor can be interrupted in the following ways

- by an external signal generated by a peripheral,
- by an internal signal generated by a special instruction in the program,
- by an internal signal generated due to an exceptional condition which occurs while executing an instruction

Interrupts in 8086

Interrupts are 2 types-

- Hardware interrupt
- Software interrupt



Hardware Interrupts

Maskable: The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware

interrupts except NMI. The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts

Non-Maskable: The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts.
Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processor all the hardware interrupts initiated through INTR pin are maskable by clearing the interrupt flag (IF). The interrupt initiated through NMI(Non-Maskable Interrupt) pin and all software interrupts are non-maskable.

Non-Maskable Interrupts

- Used during power failure
- Used during critical response time
- Used during non-recoverable hardware errors
- Used watchdog interrupt
- Used during memory parity errors

Software Interrupts

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. **The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.**

Software Interrupt (INT n)

- Used by operating systems to provide hooks into various function
- Used as a communication mechanism between different parts of the program

256 INTERRUPTS OF 8086 ARE DIVIDED IN TO 3 GROUPS

1. TYPE 0 TO TYPE 4 INTERRUPTS
These Are Used For Fixed Operations And Hence Are Called Dedicated Interrupts

- **Type – 0 Divide Error Interrupt**
  Quotient Is Large Cant Be Fit In Al/Ax Or Divide By Zero
- **Type –1 Single Step Interrupt**

Used For Executing The Program In Single Step Mode By Setting Trap Flag

- **Type – 2 Non Maskable Interrupt**
  This Interrupt Is Used For Execution Of NMI Pin.
- **Type – 3 Break Point Interrupt**
  Used For Providing Break Points In The Program
- **Type – 4 Over Flow Interrupt**
  Used To Handle Any Overflow Error.

2. TYPE 5 TO TYPE 31 INTERRUPTS
Not Used By 8086,reserved For Higher Processors Like 80286 80386 Etc
3. TYPE 32 TO 255 INTERRUPTS
Available For User, called User Defined Interrupts These Can Be H/W Interrupts And Activated Through Intr Line Or Can Be S/W Interrupts.

Interrupts in 8085

Types of Interrupts in 8085

There are five interrupt signals in the 8085 microprocessor:

TRAP: The TRAP interrupt is a non-maskable interrupt that is generated by an external device, such as a power failure or a hardware malfunction. The TRAP interrupt has the highest priority and cannot be disabled.
RST 7.5: The RST 7.5 interrupt is a maskable interrupt that is generated by a software instruction. It has the second highest priority.
RST 6.5: The RST 6.5 interrupt is a maskable interrupt that is generated by a software instruction. It has the third highest priority.
RST 5.5: The RST 5.5 interrupt is a maskable interrupt that is generated by a software instruction. It has the fourth highest priority.
INTR: The INTR interrupt is a maskable interrupt that is generated by an external device, such as a keyboard or a mouse. It has the lowest priority and can be disabled.

Interrupts can be classified into various categories based on different parameters:
Hardware and Software Interrupts – When microprocessors receive interrupt signals through pins (hardware) of microprocessors, they are known as Hardware Interrupts. There are 5 Hardware Interrupts in 8085 microprocessors. They are – INTR, RST 7.5, RST 6.5, RST 5.5, TRAP Software Interrupts are those which are inserted

in between the program which means these are mnemonics of microprocessor. There are 8 software interrupts in the 8085 microprocessor. They are – RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6, RST 7.

Vectored and Non-Vectored Interrupts – Vectored Interrupts are those which have a fixed vector address (starting address of sub-routine) and after executing these, program control is transferred to that address. Vector Addresses are calculated by the formula 8 * TYPE

INTERRUPT	VECTOR ADDRESS
TRAP (RST 4.5)	24 H
RST 5.5	2C H
RST 6.5	34 H
RST 7.5	3C H

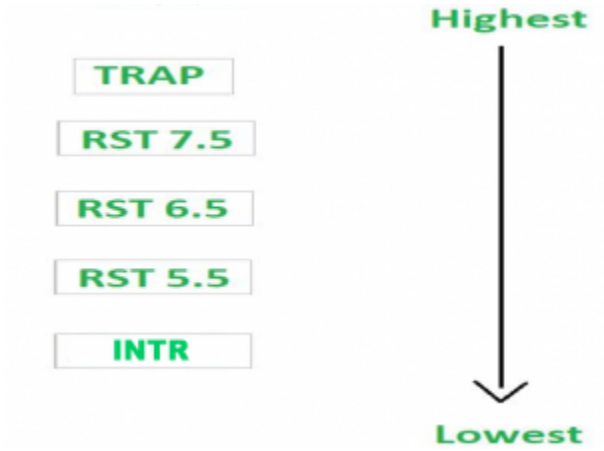
For Software interrupts vector addresses are given by:

INTERRUPT	VECTOR ADDRESS
RST 0	00 H
RST 1	08 H
RST 2	10 H
RST 3	18 H
RST 4	20 H
RST 5	28 H
RST 6	30 H
RST 7	38 H

Non-Vectored Interrupts are those in which the vector address is not predefined. The interrupting device gives the address of the sub-routine for these interrupts. INTR is the only non-vectored interrupt in an 8085 microprocessor.

Maskable and Non-Maskable Interrupts – Maskable Interrupts are those which can be disabled or ignored by the microprocessor. These interrupts are either edge-triggered or level-triggered, so they can be disabled. INTR, RST 7.5, RST 6.5, RST 5.5 are maskable interrupts in 8085 microprocessor. Non-Maskable Interrupts are those which cannot be disabled or ignored by microprocessor. TRAP is a non-maskable interrupt. It consists of both level as well as edge triggering and is used in critical power failure conditions.

Priority of Interrupts – When microprocessor receives multiple interrupt requests simultaneously, it will execute the interrupt service request (ISR) according to the priority of the interrupts.



Instruction for Interrupts

Enable Interrupt (EI) – The interrupt enable flip-flop is set and all interrupts are enabled following the execution of next instruction followed by EI. No flags are affected. After a system reset, the interrupt enable flip-flop is reset, thus disabling the interrupts. This instruction is necessary to enable the interrupts again (except TRAP).
Disable Interrupt (DI) – This instruction is used to reset the value of enable flip-flop hence disabling all the interrupts. No flags are affected by this instruction.
Set Interrupt Mask (SIM) – It is used to implement the hardware interrupts (RST 7.5, RST 6.5, RST 5.5) by setting various bits to form masks or generate output data via the Serial Output Data (SOD) line. First the required value is loaded in accumulator then SIM will take the bit pattern from it.
Read Interrupt Mask (RIM) – This instruction is used to read the status of the hardware interrupts (RST 7.5, RST 6.5, RST 5.5) by loading into the A register a byte

which defines the condition of the mask bits for the interrupts. It also reads the condition of SID (Serial Input Data) bit on the microprocessor.

Uses of Interrupts in 8085 microprocessor

Interrupts in the 8085 microprocessor are used for various purposes, including:

- Real-time processing
- Multi-tasking
- Input/output operations
- Error handling
- Power management

Issues of Interrupts in 8085 microprocessor

There are several issues that need to be considered when using interrupts in the 8085 microprocessor:

- Priority conflicts
- Race conditions
- Interrupt latency
- Interrupt nesting
- Interrupt overhead

Exercises from Book (Marut)

1.Compute the location of the interrupt vector for interrupt 20h.

Ans: The interrupt vector table is a table of addresses that the CPU uses to jump to the appropriate interrupt service routine (ISR) when an interrupt is triggered.
In the case of interrupt 20h, the ISR address is located at offset 20h * 4 in the interrupt vector table, since each entry in the table is 4 bytes long.
Therefore, to compute the location of the interrupt vector for interrupt 20h, we need to multiply the interrupt number (20h) by 4 to get the offset in the interrupt vector table.
20h * 4 = 80h
So the interrupt vector for interrupt 20h is located at offset 80h in the interrupt vector table.

2. Use DEBUG to find the value of the interrupt vector for Interrupt 0.

Ans: In DOS, we can use the DEBUG program to examine the contents of memory. To find the value of the interrupt vector for Interrupt 0, we can follow these steps:
Open a command prompt in DOS. Type "debug" and press Enter to start the DEBUG program. Type the following commands, pressing Enter after each line:
-d 0:0
This will display the first 16 bytes of memory, starting at address 0:0.
Look for the entry for Interrupt 0, which is located at offset 00h * 4 in the interrupt vector table. The value at this location is the interrupt vector for Interrupt 0.
Note that Interrupt 0 is the divide-by-zero interrupt, so its ISR address is critical for handling errors in the CPU's arithmetic operations.

On most x86 systems, the interrupt vector for Interrupt 0 is set to the default ISR address of 0000:0472h. So if we follow the above steps and examine memory at address 0:0, we should see the value 0472h at offset 00h * 4, indicating that this is the interrupt vector for Interrupt 0.

3. Write instructions that use the BIOS interrupt 17h to print the message "Hello".
Ans:

```
.MODEL SMALL
.STACK 100h

.DATA
message DB 'Hello$'

.CODE
MAIN PROC
MOV AX, 0Eh
MOV DX, OFFSET message
INT 17h

MOV AH, 4Ch
INT 21h
MAIN ENDP

END MAIN
```

4.Write instructions that use the INT 2lh, function 2Ah, to display the current date.

Ans:

```
.model tiny
.code
org 100h

main proc near

    mov ah,2ah
    int 21h

    push dx
    mov ax,cx
    mov si,offset year
    call hexToAsc
    pop dx

    push dx
    mov ah,00
    mov al,dh
    mov si,offset month
    call hexToAsc
    pop dx

    push dx
    mov ah,00
    mov al,dl
    mov si,offset day
    call hexToAsc
    pop dx


    mov ah,09h
    mov dx,offset messagecurrentdate
    int 21h

    mov ah,4ch
    mov al,00
    int 21h
```

```
    messagecurrentdate db 0ah,0dh,"Current Date : "
    day db '  '
    separator1 db '/'
    month db '  '
    separator2 db '/'
    year db "    $"

endp


hexToAsc proc near        ;AX input , si point result storage addres
        mov cx,00h
        mov bx,0ah
        hexloop1:
                mov dx,0
                div bx
                add dl,'0'
                push dx
                inc cx
                cmp ax,0ah
                jge hexloop1
                add al,'0'
                mov [si],al
        hexloop2:
                pop ax
                inc si
                mov [si],al
                loop hexloop2
                ret
endp

end main
```

Programming Exercises from Book (Marut)

5. Write a program that will output the message "Hello" once every half second to the screen.

Ans: (not sure about this)

```
.MODEL SMALL
.STACK 100h

.DATA
message DB 'Hello$'

.CODE
MAIN PROC

    MOV AX, @DATA     ; initialize DS register
    MOV DS, AX

    MOV AH, 0Ch       ; set function 0Ch to print character
    MOV CX, 0         ; set color to default
    MOV DX, 0         ; set cursor position to current position
    MOV BL, 07h       ; set attribute to normal (white on black)

    LOOP_START:
        MOV AH, 0Eh   ; set function 0Eh to print string
        MOV DX, OFFSET message ; set DX to the offset of the message
        INT 10h       ; print the message to the screen

        MOV AH, 86h   ; set function 86h to wait for a short period of time
        MOV CX, 5000  ; set CX to the number of microseconds to wait
        INT 15h       ; call interrupt 15h to wait for a short period of time

        JMP LOOP_START ; jump back to the start of the loop

    MOV AH, 4Ch       ; set function 4Ch to exit program
    INT 21h           ; call interrupt 21h to exit program

MAIN ENDP
END MAIN
```

6. Modify PGMIS_Z.ASM so that INT 21h, function 9, Is called to display the time only when the seconds change.

Ans: (not sure)

```
TITLE PGM15_2: DISPLAY_TIME_VR
; program that displays the current time
; and updates the time 18.2 times a second

EXTRN GET_TIME:NEAR,SETUP_INT:NEAR

.stackseg segment stack '100h'
   ; stack segment definition
.stackseg ends

.data
TIME_BUF DB '00:00:00$' ; time buffer hr:min:sec
CURSOR_POS DW ? ; cursor position (row:col)
NEW_VEC DW ? ; new interrupt vector
OLD_VEC DW ? ; old interrupt vector

.code
MAIN PROC
   mov ax, @data
   mov ds, ax ; initialize data segment

   ; save cursor position
   mov ah, 3 ; function 3, get cursor
   mov bh, 0 ; page 0
   int 10h ; get cursor position
   mov CURSOR_POS, dx ; save it

   ; set up interrupt procedure by placing segment of SET_TIME_INT in
NEW_VEC
   mov NEW_VEC, OFFSET SET_TIME_INT ; offset
   mov NEW_VEC+2, SEG SET_TIME_INT ; segment

   lea di, OLD_VEC ; DI points to vector buffer
   lea si, NEW_VEC ; SI points to new vector
   mov al, 0Ch ; timer interrupt
```

```
    call SETUP_INT

    ; read keyboard
    mov ah, 0
    int 16h

    ; restore old interrupt
    lea di, NEW_VEC
    lea si, OLD_VEC
    mov al, 0Ch ; timer interrupt
    call SETUP_INT

    mov ah, 4Ch
    int 21h ; return to DOS
MAIN ENDP

SET_TIME_INT PROC
    ; interrupt procedure
    ; activated by the timer
    push ds
    mov ax, @data
    mov ds, ax ; set data segment

    ; get new time
    lea bx, TIME_BUF
    call GET_TIME

    ; display time
    cmp TIME_BUF+6, '0' ; check if seconds changed
    je skip_display ; if not, skip display
    lea dx, TIME_BUF
    mov ah, 09h ; function 9, display string
    int 21h

    skip_display:
    ; restore cursor position
    mov ah, 02h ; function 2, move cursor
    mov bh, 0 ; page 0
```

```
    mov dx, CURSOR_POS ; cursor position
    int 10h

    pop ds
    iret
SET_TIME_INT ENDP

END MAIN
```

7. Write a memory resident program similar to PGM15_3.ASM using INT 2lh, function 3lh.

Ans:

```
TITLE PGM15_2: DISPLAY_TIME_VR
; program that displays the current time
; and updates the time 18.2 times a second

EXTRN GET_TIME:NEAR,SETUP_INT:NEAR

.stackseg segment stack '100h'
   ; stack segment definition
.stackseg ends

.data
TIME_BUF DB '00:00:00$' ; time buffer hr:min:sec
CURSOR_POS DW ? ; cursor position (row:col)
NEW_VEC DW ? ; new interrupt vector
OLD_VEC DW ? ; old interrupt vector

.code
MAIN PROC
   mov ax, @data
   mov ds, ax ; initialize data segment

   ; save cursor position
   mov ah, 3 ; function 3, get cursor
   mov bh, 0 ; page 0
   int 10h ; get cursor position
```

```
    mov CURSOR_POS, dx ; save it

    ; set up interrupt procedure by placing segment of
SET_TIME_INT in NEW_VEC
    mov NEW_VEC, OFFSET SET_TIME_INT ; offset
    mov NEW_VEC+2, SEG SET_TIME_INT ; segment

    lea di, OLD_VEC ; DI points to vector buffer
    lea si, NEW_VEC ; SI points to new vector
    mov al, 0Ch ; timer interrupt
    call SETUP_INT

    ; read keyboard
    mov ah, 0
    int 16h

    ; restore old interrupt
    lea di, NEW_VEC
    lea si, OLD_VEC
    mov al, 0Ch ; timer interrupt
    call SETUP_INT

    mov ah, 4Ch
    int 21h ; return to DOS
MAIN ENDP

SET_TIME_INT PROC
    ; interrupt procedure
    ; activated by the timer
    push ds
    mov ax, @data
    mov ds, ax ; set data segment

    ; get new time
    lea bx, TIME_BUF
    call GET_TIME

    ; display time
```

```
    cmp TIME_BUF+6, '0' ; check if seconds changed
    je skip_display ; if not, skip display
    lea dx, TIME_BUF
    mov ah, 09h ; function 9, display string
    int 21h

    skip_display:
    ; restore cursor position
    mov ah, 02h ; function 2, move cursor
    mov bh, 0 ; page 0
    mov dx, CURSOR_POS ; cursor position
    int 10h

    pop ds
    iret
SET_TIME_INT ENDP

END MAIN
```

Other Important Questions (Quiz, Exams, Assignments)

1. List the sequence of actions performed by the Intel 8086 microprocessor while handling a Software Interrupt.

Ans: In 8086 microprocessor following tasks are performed when microprocessor encounters an interrupt:

1. The value of the flag register is pushed into the stack. It means that first the value of SP (Stack Pointer) is decremented by 2 then the value of flag register is pushed to the memory address of the stack segment.
2. The value of the starting memory address of CS (Code Segment) is pushed into the stack.
3. The value of IP (Instruction Pointer) is pushed into the stack.
4. IP is loaded from word location (Interrupt type) * 04.
5. CS is loaded from the next word location.
6. Interrupt and Trap flags are reset to 0.

2.Analyze how the 8086 microprocessor handles interrupts and exceptions and how they affect its operation and reliability. (Assignment)

Ans: The 8086 microprocessor handles interrupts and exceptions through its Interrupt Vector Table (IVT). The IVT is a table of 256 addresses, each representing a specific interrupt or exception. When an interrupt or exception occurs, the 8086 first saves the current state of the processor, including the contents of the registers, on the stack. It then looks up the address of the interrupt or exception handler in the IVT, jumps to that address, and executes the corresponding interrupt or exception routine.

Interrupts and exceptions can affect the operation and reliability of the 8086 microprocessor in several ways. On one hand, interrupts and exceptions are essential for the proper functioning of the microprocessor, as they allow it to handle external events and respond to unexpected conditions. For example, interrupts can be used to handle input/output operations, timer events, or other hardware interrupts, while exceptions can detect errors such as divide-by-zero, invalid opcode, or memory access violations.

On the other hand, interrupts and exceptions can also cause issues such as race conditions, resource conflicts, and reliability problems. For instance, if two interrupts occur simultaneously, the 8086 may have to deal with a race condition, where the order of execution is not well-defined and may depend on external factors. Moreover, interrupts and exceptions may also affect the performance and reliability of the microprocessor by introducing delays, conflicts, or other side effects.

To minimize these issues, the 8086 microprocessor implements several mechanisms, such as priority levels, interrupt masking, and error handling. Priority levels allow the microprocessor to handle interrupts in a predefined order, based on their importance or urgency. Interrupt masking allows the microprocessor to selectively enable or disable interrupts, depending on the current context or task. Error handling allows the microprocessor to detect and recover from errors, such as by resetting the system or signaling an error code.

Overall, the 8086 microprocessor's handling of interrupts and exceptions is crucial for its proper functioning and reliability. While these mechanisms can introduce some overhead and complexity, they are necessary to ensure that the microprocessor can handle external events and recover from unexpected conditions in a safe and efficient manner.

Macro

Macro is in-line functions.

A macro Is a block of text that has been given a name. When MASM(MASM stands for Microsoft Macro Assembler.) encounters the name during assembly, it inserts the block Into the program. The text may consist of Instructions, pseudo-ops, comments, or references to other macros. ·
The syntax of macro definition is

```
macro_name  MACRO  dl,d2,. . .dn
            statements
            ENDM
```

Illegal Macro Invocations

There are often restrictions on the arguments for a macro. For example, the arguments in the MOVW macro must be memory words or 16-bit registers. The macro invocation

```
MOVW  AX,1ABCh
```

generates the code

```
PUSH  1ABCh
POP   AX
```

and because an immediate data push is illegal (for the 8086/8088), this results in an assembly error. One way to guard against this situation is to put a comment in the macro; for example,

```
MOVW     MACRO     WORD1,WORD2
;arguments must be memory words or 16-bit registers
         PUSH      WORD2
         POP       WORD1
         ENDM
```

Example 13.1 Define a macro to move a word into a word.

Solution:

```
MOVW    MACRO   WORD1,WORD2
        PUSH    WORD2
        POP     WORD1
        ENDM
```

Here the name of the macro is MOVW. WORD1 and WORD2 are the dummy arguments.

To use a macro in a program, we **invoke** it. The syntax is

```
macro_name    a1,a2, . . . an
```

where a1, a2, . . . an is a list of actual arguments. When MASM encounters the macro name, it **expands** the macro; that is, it copies the macro statements into the program at the position of the invocation, just as if the user had typed them in. As it copies the statements, MASM replaces each dummy argument di by the corresponding actual argument ai and creates the machine code for any instructions.

A macro definition must come before its invocation in a program listing. To ensure this sequence, macro definitions are usually placed at the beginning of a program. It is also possible to create a library of macros to be used by any program, and we do this later in the chapter.

Example 13.2 Invoke the macro MOVW to move B to A, where A and B are word variables.

Solution: `MOVW  A,B`

To expand this macro, MASM would copy the macro statements into the program at the position of the call, replacing each occurrence of WORD1 by A, and WORD2 by B. The result is

```
PUSH  B
POP   A
```

In expanding a macro, the assembler simply substitutes the character strings defining the actual arguments for the corresponding dummy ones. For example, the following calls to the MOVW macro

```
MOVW A,DX        and        MOVW A+2,B
```

cause the assembler to insert this code into the program:

```
PUSH DX          and        PUSH B
POP  A                      POP  A+2
```