Geethanjali College of Engineering and Technology

Cheeryal (V), Keesara (M), Medchal Dt – 501 301, Telangana

(AUTONOMOUS)

Computer Architecture and Assembly Language Programming Lab

Laboratory Manual

DEPARTMENT OF COMPUTER SCIENCE and ENGINEERING
(2021-22)

Lab Incharge

HOD-CSE
Dr. A Sreelakshmi
Prof. & Head.

Geethanjali College of Engineering and Technology

Department of COMPUTER SCIENCE & ENGINEERING

Computer Organization and Assembly Language Programming Lab Manual

(JNTU CODE): 20CS22L02 **Programme : UG**

Branch : CSE A,B, C , D,E	**Version No : 1**
Year : II	**Updated on :**
Semester : II	**No.of pages :**

Classification status (Unrestricted / Restricted) **Distribution List : Department, Lab, Library, Lab In Charge**
Prepared by : **1) Name : Dr. Puja S Prasad ,Mrs M.Alphonsa,** **2) Sign :** **3) Design :** **4) Date :**

Verified by: **1)Name :** **2) Sign :** **3) Designation: 4)Date**	**<u>* For Q.C Only.</u> 1)Name: 2)Sign:** **3)Designation:** **4)Date:**

Approved by :(HOD) **1)Name: Dr. A. Sreelakshmi** **2)Sign:** **3)Date:**

<u>LIST OF LAB EXERCISES</u>

Computer Organization and Assembly Language Programming Lab		
S. NO	**NAME OF THE PROGRAM**	**PAGE NO.**
Week-1	1. Architecture of 8086 Microprocessor. 2. Instruction Set of 8086.	28
Week-2	1. Write a program to display string "Computer Science and Engineering “. 2. Write an Assembly Language Program (ALP) to display multiple strings line by line. 3. Write an Assembly Language Program(ALP) to find the maximum of three numbers.	36
Week-3	1. Write an assembly language program (ALP) to Print numbers from 0 to 9. 2. Write an Assembly Language Program (ALP) to check whether a given number is even or odd .	39
Week-4	1. Write an Assembly Language Program (ALP) to find	41

	Factorial of a number. 2. Write an Assembly Language Program (ALP) to print Fibonacci series up to 5 numbers.	
Week-5	1. Write an Assembly Language Program (ALP) to take n values from user and calculate their sum for 8086. BL contains the result. 2. Write an Assembly Language Program(ALP) to take n values from user and calculate maximum and minimum values.	42
Week-6	1. Write 8086 assembly language program to transfer a block of data from one location to another. 2. Write an assembly language program to reverse the given string for 8086 3. Write an Assembly Language Program (ALP) to perform Addition of two 2X2 matrices.	47
Week-7	1. Write an Assembly Language Program (ALP) for linear search. 2. Write an Assembly Language Program (ALP) to take n values from user and sort them in ascending order.	51

ADDITIONAL PROGRAMS

Computer Organization and Assembly Language Programming Lab		
S.NO	**NAME OF THE PROGRAM**	**PAGE NO.**
1	Write an as assembly language program to evaluate Arithmetic Expression using 8 bit and 16bit i) a = b + c – d * e ii) z = x * y + w – v +u / k	53
2	Write an ALP of 8086 to take N numbers as input. And do the following operations on them. a. Arrange in Descending order b. Arrange in ascending	55
3	Write an ALP of 8086 to take N numbers as input and find average	57
4	Write an ALP of 8086 to take a string of as input Find the length of the string	58
5	Write an ALP of 8086 to take a string of as input Find the given string is Palindrome or not	59

Vision of the institute

Geethanjali visualizes dissemination of knowledge and skills to students, who would eventually contribute to well being of the people of the nation and global community.

Mission of the institute

- To impart adequate fundamental knowledge in all basic sciences and engineering, technical and Inter-personal skills to students.
- To bring out creativity in students that would promote innovation, research and entrepreneurship
- To Preserve and promote cultural heritage, humanistic and spiritual values promoting peace and harmony in society.

Vision of the CSE Department

To produce globally competent and socially responsible computer science engineers contributing to the advancement of engineering and technology which involves creativity and innovation by providing excellent learning environment with world class facilities.

Mission of the CSE Department

1. To be a center of excellence in instruction, innovation in research and scholarship, and service to the stake holders, the profession, and the public.

2. To prepare graduates to enter a rapidly changing field as a competent computer science engineer.

3. To prepare graduates capable in all phases of software development, possess a firm understanding of hardware technologies, have the strong mathematical background necessary for scientific computing, and be sufficiently well versed in theory and practice to allow growth within the discipline as it advances.

4. To prepare graduates to assume leadership roles by possessing good communication skills, the ability to work effectively as team members, and an appreciation for their social and ethical responsibility in a global setting.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs): Program Educational Objectives (PEOs) are broad statements that describe what graduates are expected to attain within a few years of graduation. The PEOs for Computer Science and Engineering graduates are:

PEO-I: To provide graduates with a good foundation in mathematics, sciences and engineering fundamentals required to solve engineering problems that will facilitate them to find employment in industry and/or to pursue postgraduate studies with an appreciation for lifelong learning.

PEO-II: To provide graduates with analytical and problem solving skills to design algorithms, other hardware/ software systems, and inculcate professional ethics, interpersonal skills to work in a multi-cultural team.

PEO-III: To facilitate graduates get familiarized with state of the art software/hardware tools, imbibing creativity and innovation that would enable them to develop cutting-edge technologies of multi-disciplinary nature for societal development.

Program Outcomes (CSE)

PROGRAM OUTCOMES (POs): Program Outcomes (POs) describe what students are expected to know and be able to do by the time of graduation to accomplish Program Educational Objectives (PEOs). The Program Outcomes for Computer Science and Engineering graduates are: Engineering Graduates would be able to:

PO 1:**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4: **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO 6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO 8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO 11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12: **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs):

PSO 1: Demonstrate competency in Programming and problem solving skills and apply these skills in solving real world problems

PSO 2: Select appropriate programming languages, Data structures and algorithms in combination with modern technologies and tools, apply them in developing creative and innovative solutions .

PSO 3:Demonstrate adequate knowledge in emerging technologies

Lab Course Objectives

Develop ability to

1. Introduce principles of computer organization and the basic architectural concepts.
2. Recommend instruction formats, addressing modes, micro instructions for design of control unit
3. Write assembly level programs using 8086 microprocessor.
4. Understand the I/O and memory organizations of a Computer system
5. Recognize different parallel processing architectures

Lab Course Outcomes

After completion of the course, student would be able to

20CS22L02.1 Demonstrate an understanding of the design of the functional units of a digital computer system.

20CS22L02.2 Design micro instructions for different kinds of CPU organizations with proper understanding of instruction formats and addressing modes

20CS22L02.3 Write assembly language programs using 8086 microprocessor with the knowledge of pin diagram, registers and instruction formats of 8086 microprocessor.

20CS22L02.4 Identify different hardware components associated with the memory and I/O organization of a computer

20CS22L02.5 Differentiate different parallel processing architectures

Mapping of Lab Course with Programme Educational Objectives

S.No	Course Component	Code	Course	Semester	PEO 1	PEO 2	PEO 3
1	Professional Course	20CS22L02	CAALP	II	1	3	3

Mapping of Lab Course outcomes with Programme outcomes:

Pos	Program Outcomes													
	1	2	3	4	5	6	7	8	9	10	11	12	PSO 1	PSO2
CAALP														
20CS22L02.1 Demonstrate an understanding of the design of the functional units of a digital computer system.	**2**	**2**	**1**	**1**			**1**				**2**	**1**	**1**	**1**
20CS22L02.2 Design micro instructions for different kinds of CPU organizations with proper understanding of instruction formats and addressing modes	**2**	**2**	**1**	**1**			**1**				**2**	**1**	**1**	**1**
20CS22L02.3 Write assembly language programs using 8086 microprocessor with the knowledge of pin diagram, registers and instruction formats of 8086 microprocessor.	**2**	**2**	**1**	**1**		**1**	**1**				**2**	**1**	**1**	**1**
20CS22L02.4 Identify different hardware components associated with the memory and I/O organization of a computer	**2**	**2**	**1**	**1**		**1**	**1**				**2**	**1**	**1**	**1**
20CS22L02.5 Differentiate different parallel processing architectures	**3**	**3**	**3**	**2**	**2**		**1**				**2**	**1**	**2**	**2**

Prerequisitesi

Digital Logic Design

Instructions to the students:

1. Students are required to attend all labs.
2. Students will work in a group of two in hardware laboratories and individually in computer laboratories.
3. While coming to the lab bring the lab workbook cum observation book.
4. Before coming to the lab, prepare the prelab questions.
5. Utilize 3 hours time properly to execute the program and note down the executed program in workbook with output and take signature from the instructor.
6. If the experiment is not completed in the prescribed time, the pending work has to be done in the leisure hour or extended hours.
7. You will be expected to submit the completed workbook according to the deadlines set up by your instructor.
8. For practical subjects there shall be a continuous evaluation during the semester for 25 internal marks and 50 end examination marks.

Instructions to laboratory teachers:

1. Observation book and lab records submitted for the lab work are to be checked and signed before the next lab session.
2. Students should be instructed to switch ON the power supply after the connections are checked by the lab assistant / teacher.
3. The promptness of submission should be strictly insisted by awarding the marks accordingly.
4. Ask viva questions at the end of the experiment.
5. Do not allow students who come late to the lab class.
6. Encourage the students to do the experiments innovatively
7. Fill continuous Evaluation sheet, on continuous basis.
8. Ensure that the students are dressed in a formal way.

Scheme of Lab Exam Evaluation:

Evaluation of Internal Marks: 30M

a) 15 Marks are awarded for day to day work

Sl. No.	Particulars	Marks
1	Record and Observation book	5
2	Attendance and behavior of student	5
3	Viva and performance	5

b) 15 Marks are awarded for conducting laboratory test as follows:

Sl. No.	Particulars	Marks
1	Write up of program	5
2	Execution of Program	5
3	Viva performance	5

Evaluation of External Marks:

70 Marks are awarded for conducting laboratory test as follows:

Sl. No.	Particulars	Marks
1	Algorithm	25
2	Write up of program	15
3	Execution of Program	15
4	Viva	15

INTRODUCTION

8086 Architecture

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

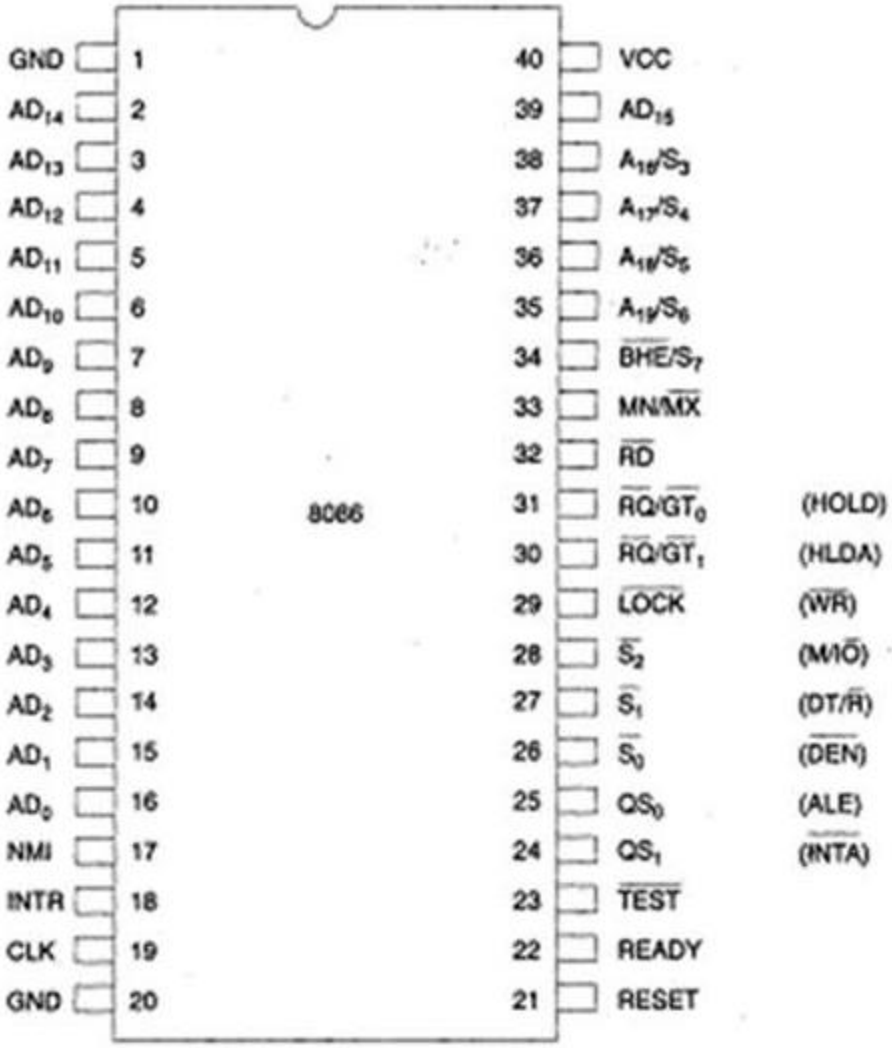
Features of 8086

The most prominent features of a 8086 microprocessor are as follows −

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

Comparison between 8085 & 8086 Microprocessor

- **Size** − 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.
- **Address Bus** − 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- **Memory** − 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- **Instruction** − 8085 doesn't have an instruction queue, whereas 8086 has an instruction queue.
- **Pipelining** − 8085 doesn't support a pipelined architecture while 8086 supports a pipelined architecture.
- **I/O** − 8085 can address 2^8 = 256 I/O's, whereas 8086 can access 2^16 = 65,536 I/O's.
- **Cost** − The cost of 8085 is low whereas that of 8086 is high.
- 8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip. Let us now discuss in detail the pin configuration of a 8086 Microprocessor.
- **8086 Pin Diagram**
- Here is the pin diagram of 8086 microprocessor −



- Let us now discuss the signals in detail −
- **Power supply and frequency signals**
- It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.
- **Clock signal**
- Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.
- **Address/data bus**
- AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.
- **Address/status bus**
- A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.
- **S7/BHE**
- BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.
- **Read(*$\overline{RD}$*)**
- It is available at pin 32 and is used to read signal for Read operation.
- **Ready**
- It is available at pin 32. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

- **RESET**
- It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.
- **INTR**
- It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.
- **NMI**
- It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.
- $\overline{TEST}$
- This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.
- **MN/$\overline{MX}$**
- It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-aversa.
- **INTA**
- It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.
- **ALE**
- It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.
- **DEN**
- It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.
- **DT/R**
- It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transreceiver. When it is high, data is transmitted out and vice-a-versa.
- **M/IO**
- This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.
- **WR**
- It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.
- **HLDA**
- It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.
- **HOLD**
- This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.
- **QS_1 and QS_0**
- These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table −

QS_0	QS_1	Status
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

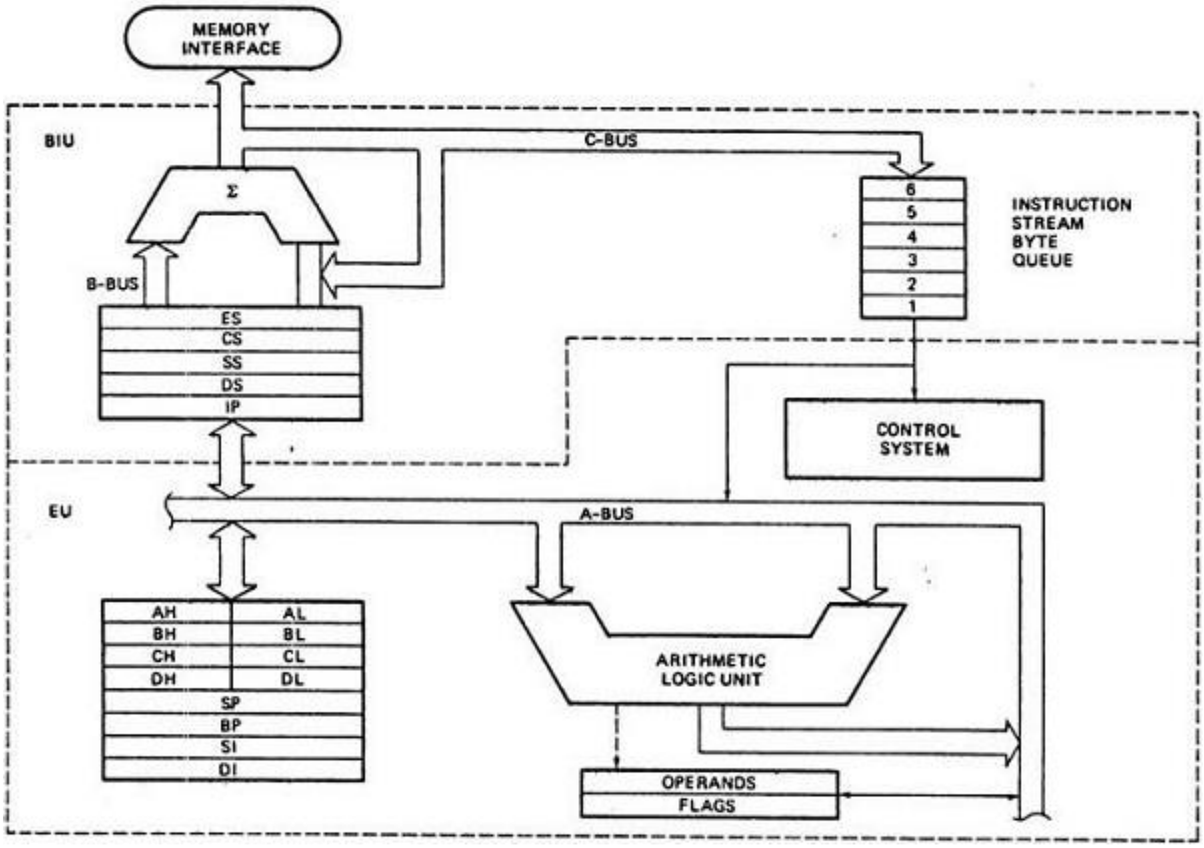
- **S_0, S_1, S_2**
- These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status −

S_2	S_1	S_0	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

- **LOCK**
- When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.
- **RQ/GT_1 and RQ/GT_0**
- These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT_0 has a higher priority than RQ/GT_1.

Architecture of 8086

The following diagram depicts the architecture of a 8086 Microprocessor −



Registers

In 16-bit mode, such as provided by the Pentium processor when operating as a Virtual 8086 (this is the mode used when Windows 95 displays a DOS prompt), the processor provides the programmer with 14 internal registers, each 16 bits wide. They are grouped into several categories as follows:

4. Four general-purpose registers, AX, BX, CX, and DX. Each of these is a combination of two 8-bit registers which are separately accessible as AL, BL, CL, DL (the "low" bytes) and AH, BH, CH, and DH (the "high" bytes). For example, if AX contains the 16-bit number 1234h, then AL contains 34h and AH contains 12h.
5. Four special-purpose registers, SP, BP, SI, and DI.
6. Four segment registers, CS, DS, ES, and SS.
7. The instruction pointer, IP (sometimes referred to as the program counter).
8. The status flag register, FLAGS.

The first four registers as "general-purpose", each of them is designed to play a particular role in common use:

6. AX is the "accumulator"; some of the operations, such as MUL and DIV, require that one of the operands be in the accumulator. Some other operations, such as ADD and SUB, may be applied to any of the registers (that is, any of the eight general- and special-purpose registers) but are more efficient when working with the accumulator.
7. BX is the "base" register; it is the only general-purpose register which may be used for indirect addressing. For example, the instruction MOV [BX], AX causes the contents of AX to be stored in the memory location whose address is given in BX.
8. CX is the "count" register. The looping instructions (LOOP, LOOPE, and LOOPNE), the shift and rotate instructions (RCL, RCR, ROL, ROR, SHL, SHR, and SAR), and the string instructions (with the prefixes REP, REPE, and REPNE) all use the count register to determine how many times they will repeat.

9. DX is the "data" register; it is used together with AX for the word-size MUL and DIV operations, and it can also hold the port number for the IN and OUT instructions, but it is mostly available as a convenient place to store data, as are all of the other general-purpose registers.

Here are brief descriptions of the four special-purpose registers:

- SP is the stack pointer, indicating the current position of the top of the stack. You should generally never modify this directly, since the subroutine and interrupt call-and-return mechanisms depend on the contents of the stack.
- BP is the base pointer, which can be used for indirect addressing similar to BX.
- SI is the source index, used as a pointer to the current character being read in a string instruction (LODS, MOVS, or CMPS). It is also available as an offset to add to BX or BP when doing indirect addressing; for example, the instruction MOV [BX+SI], AX copies the contents of AX into the memory location whose address is the sum of the contents of BX and SI.
- DI is the destination index, used as a pointer to the current character being written or compared in a string instruction (MOVS, STOS, CMPS, or SCAS). It is also available as an offset, just like SI.

Since all of these registers are 16 bits wide, they can only contain addresses for memory within a range of 64K (=2^16) bytes. To support machines with more than 64K of physical memory, Intel implemented the concept of *segmented* memory. At any given time, a 16-bit address will be interpreted as an offset within a 64K segment determined by one of the four segment registers (CS, DS, ES, and SS).

Each segment register has its own special uses:

- CS determines the "code" segment; this is where the executable code of a program is located. It is not directly modifiable by the programmer, except by executing one of the branching instructions. One of the reasons for separating the code segment from other segments is that well-behaved programs never modify their code while executing; therefore, the code segment can be identified as "read-only". This simplifies the work of a cache, since no effort is required to maintain consistency between the cache and main memory. It also permits several instances of a single program to run at once (in a multitasking operating system), all sharing the same code segment in memory; each instance has its own data and stack segments where the information specific to the instance is kept. Picture multiple windows, each running Word on a different document; each one needs its own data segment to store its document, but they can all execute the same loaded copy of Word.
- DS determines the "data" segment; it is the default segment for most memory accesses.
- ES determines the "extra" segment; it can be used instead of DS when data from two segments need to be accessed at once. In particular, the DI register gives an offset relative to ES when used in the string instructions; for example, the MOVSB instruction copies a byte from DS:SI to ES:DI (and also causes SI and DI to be incremented or decremented, ready to copy the next byte).
- SS determines the "stack" segment; the stack pointer SP gives the offset of the current top-of-stack within the stack segment. The BP register also gives an offset relative to the stack segment by default, for convenient access to data further down in the stack without having to modify SP. Just as with SP, you should not modify SS unless you know exactly what you are doing.

The instruction pointer, IP, gives the address of the *next* instruction to be executed, relative to the code segment. The only way to modify this is with a branch instruction.

The status register, FLAGS, is a collection of 1-bit values which reflect the current state of the processor and the results of recent operations. Nine of the sixteen bits are used in the 8086:

3. Carry (bit 0): set if the last arithmetic operation ended with a leftover carry bit coming off the left end of the result. This signals an overflow on unsigned numbers.
4. Parity (bit 2): set if the low-order byte of the last data operation contained an even number of 1 bits (that is, it signals an even parity condition).
5. Auxiliary Carry (bit 4): used when working with binary coded decimal (BCD) numbers.

6. Zero (bit 6): set if the last computation had a zero result. After a comparison (CMP, CMPS, or SCAS), this indicates that the values compared were equal (since their difference was zero).
7. Sign (bit 7): set if the last computation had a negative result (a 1 in the leftmost bit).
8. Trace (bit 8): when set, this puts the CPU into single-step mode, as used by debuggers.
9. Interrupt (bit 9): when set, interrupts are enabled. This bit should be cleared while the processor is executing a critical section of code that should not be interrupted (for example, when processing another interrupt).
10. Direction (bit 10): when clear, the string operations move from low addresses to high (the SI and DI registers are incremented after each character). When set, the direction is reversed (SI and DI are decremented).
11. Overflow (bit 11): set if the last arithmetic operation caused a signed overflow (for example, after adding 0001h to 7FFFh, resulting in 8000h; read as two's complement numbers, this corresponds to adding 1 to 32767 and ending up with -32768).

There are numerous operations that will test and manipulate various of these flags, but to get the contents of the entire FLAGS register one has to push the flags onto the stack (with PUSHF or by calling an appropriate interrupt handler with INT) and then pop them off into another register. To set the entire FLAGS register, the sequence is reversed (with POPF or IRET). For example, one way to set the carry flag (there are much better ways, including the STC instruction) is the following:

```
PUSHF
POP     AX
OR      AX, 1
PUSH    AX
POPF
```

Most of the time you will not have to deal with the FLAGS register explicitly; instead, you will execute one of the conditional branch instructions, Jcc, where cc is one of the following mnemonic condition codes:

- O, Overflow
- NO, Not Overflow
- B, Below; C, Carry; NAE, Not Above or Equal
- NB, Not Below; NC, Not Carry; AE, Above or Equal
- E, Equal; Z, Zero
- NE, Not Equal; NZ, Not Zero
- BE, Below or Equal; NA, Not Above (true if either Carry or Zero is set)
- NBE, Not Below or Equal; A, Above
- S, Sign
- NS, Not Sign
- P, Parity; PE, Parity Even
- NP, Not Parity; PO, Parity Odd
- L, Less; NGE, Not Greater or Equal (true if Sign and Overflow are different)
- NL, Not Less; GE, Greater or Equal
- LE, Less or Equal; NG, Not Greater (true if Sign and Overflow are different, or Zero is set)
- NLE, Not Less or Equal; G, Greater

All of the conditions on the same line are synonyms. The Above and Below conditions refer to comparisons of unsigned numbers, and the Less and Greater conditions refer to comparisons of signed (two's complement) numbers.

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

3. **LEA** − Used to load the address of operand into the provided register.
4. **LDS** − Used to load DS register and other provided register from the memory
5. **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 $CX \neq 0$.
- **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

Interrupt vectors used by DOS

Interrupt vector	Description	Version	Notes
20h	Terminate program	1.0+	Implemented in DOS kernel
21h	Main DOS API	1.0+	Implemented in DOS kernel
22h	Program terminate address	1.0+	Return address in calling program
23h	Control-C handler address	1.0+	Default handler is in the command shell (usually COMMAND.COM)

24h	Critical error handler address	1.0+	Default handler is in the command shell (usually COMMAND.COM)
25h	Absolute disk read	1.0+	Implemented in DOS kernel, enhanced in DOS 3.31 to support up to 2 GB partitions
26h	Absolute disk write	1.0+	Implemented in DOS kernel, enhanced in DOS 3.31 to support up to 2 GB partitions
27h	Terminate and stay resident	1.0+	Implemented in COMMAND.COM in DOS 1.0, DOS kernel in DOS 2.0+
28h	Idle callout	2.0+	Called by DOS kernel when waiting for input
29h	Fast console output	2.0+	Implemented by the builtin console device driver or a replacement driver like ANSI.SYS
2Ah	Networking and critical section	3.0+	Called by DOS kernel to interface with networking software
2Bh	Unused		
2Ch	Unused		
2Dh	Unused		
2Eh	Reload transient	2.0+	Implemented in COMMAND.COM
2Fh	Multiplex	3.0+	Implemented in DOS kernel and various programs (PRINT, MSCDEX, DOSKEY, APPEND, etc.) depending on subfunction number

DOS INT 21h services

AH	Description	Version
00h	Program terminate	1.0+
01h	Character input	1.0+
02h	Character output	1.0+
03h	Auxiliary input	1.0+
04h	Auxiliary output	1.0+
05h	Printer output	1.0+
06h	Direct console I/O	1.0+
07h	Direct console input without echo	1.0+
08h	Console input without echo	1.0+
09h	Display string	1.0+
0Ah	Buffered keyboard input	1.0+
0Bh	Get input status	1.0+
0Ch	Flush input buffer and input	1.0+
0Dh	Disk reset	1.0+
0Eh	Set default drive	1.0+
0Fh	Open file	1.0+
10h	Close file	1.0+
11h	Find first file	1.0+
12h	Find next file	1.0+
13h	Delete file	1.0+
14h	Sequential read	1.0+
15h	Sequential write	1.0+
16h	Create or truncate file	1.0+
17h	Rename file	1.0+
18h	Reserved	1.0+
19h	Get default drive	1.0+
1Ah	Set disk transfer address	1.0+
1Bh	Get allocation info for default drive	1.0+
1Ch	Get allocation info for specified drive	1.0+
1Dh	Reserved	1.0+
1Eh	Reserved	1.0+
1Fh	Get disk parameter block for default drive	1.0+
20h	Reserved	1.0+
21h	Random read	1.0+
22h	Random write	1.0+
23h	Get file size in records	1.0+
24h	Set random record number	1.0+
25h	Set interrupt vector	1.0+
26h	Create PSP	1.0+
27h	Random block read	1.0+

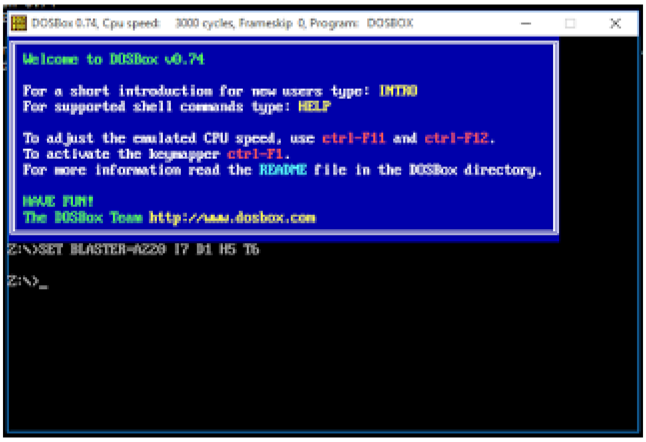
28h	Random block write	1.0+
29h	Parse filename	1.0+
2Ah	Get date	1.0+
2Bh	Set date	1.0+
2Ch	Get time	1.0+
2Dh	Set time	1.0+
2Eh	Set verify flag	1.0+
2Fh	Get disk transfer address	2.0+
30h	Get DOS version	2.0+
31h	Terminate and stay resident	2.0+
32h	Get disk parameter block for specified drive	2.0+
33h	Get or set Ctrl-Break	2.0+
34h	Get InDOS flag pointer	2.0+
35h	Get interrupt vector	2.0+
36h	Get free disk space	2.0+
37h	Get or set switch character	2.0+
38h	Get or set country info	2.0+
39h	Create subdirectory	2.0+
3Ah	Remove subdirectory	2.0+
3Bh	Change current directory	2.0+
3Ch	Create or truncate file	2.0+
3Dh	Open file	2.0+
3Eh	Close file	2.0+
3Fh	Read file or device	2.0+
40h	Write file or device	2.0+
41h	Delete file	2.0+
42h	Move file pointer	2.0+
43h	Get or set file attributes	2.0+
44h	I/O control for devices	2.0+
45h	Duplicate handle	2.0+
46h	Redirect handle	2.0+
47h	Get current directory	2.0+
48h	Allocate memory	2.0+
49h	Release memory	2.0+
4Ah	Reallocate memory	2.0+
4Bh	Execute program	2.0+
4Ch	Terminate with return code	2.0+
4Dh	Get program return code	2.0+
4Eh	Find first file	2.0+
4Fh	Find next file	2.0+
50h	Set current PSP	2.0+
51h	Get current PSP	2.0+
52h	Get DOS internal pointers (SYSVARS)	2.0+

53h	Create disk parameter block	2.0+
54h	Get verify flag	2.0+
55h	Create program PSP	2.0+
56h	Rename file	2.0+
57h	Get or set file date and time	2.0+
58h	Get or set allocation strategy	2.11+
59h	Get extended error info	3.0+
5Ah	Create unique file	3.0+
5Bh	Create new file	3.0+
5Ch	Lock or unlock file	3.0+
5Dh	File sharing functions	3.0+
5Eh	Network functions	3.0+
5Fh	Network redirection functions	3.0+
60h	Qualify filename	3.0+
61h	Reserved	3.0+
62h	Get current PSP	3.0+
63h	Get DBCS lead byte table pointer	3.0+
64h	Set wait for external event flag	3.2+
65h	Get extended country info	3.3+
66h	Get or set code page	3.3+
67h	Set handle count	3.3+
68h	Commit file	3.3+
69h	Get or set media id	4.0+
6Ah	Commit file	4.0+
6Bh	Reserved	4.0+
6Ch	Extended open/create file	4.0+

MASM installation

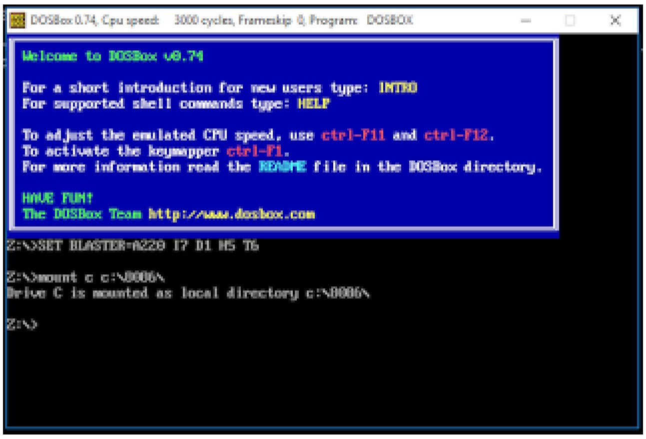
4. Download the MASM software.
5. Extract the downloaded zip file and you will find two folders named as "DOSBox-0.74" and "8086".

3. Move the folder "8086" into the "C:\" directory of your PC's hard disk.
4. Now open your "DOSBox-0.74" and find the "DOSBox.exe" launcher. Launch it by a double click.

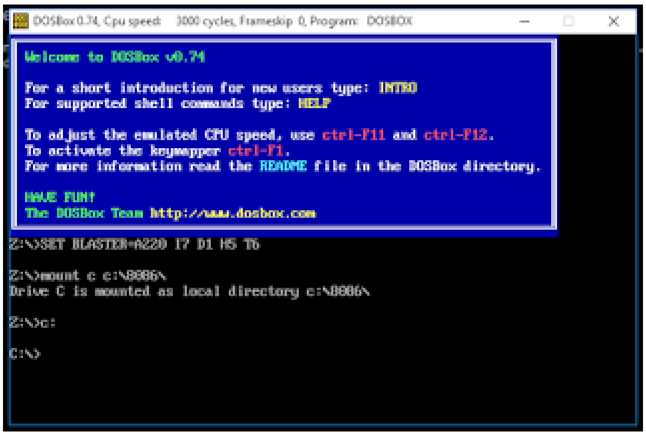


5. Now Type the exact lines found below in the DOSBox and hit enter.

"mount c c:\8086\"



6. Next, type "**c:**" and hit enter to get into C drive through *DOSBox*.



7. Now you are ready to compile and execute .asm programs.

Compiling and executing .asm programs

1. First, write the assembly language code in notepad(any text editors) and save it exactly in "**C:\8086**" with a "**.asm**" file extension.

2. Now that your assembly language code is present in "*C:\8086*", go to *DOSBox* and get into ***C*** drive using the above said steps.

3. Comypile your code using the command,

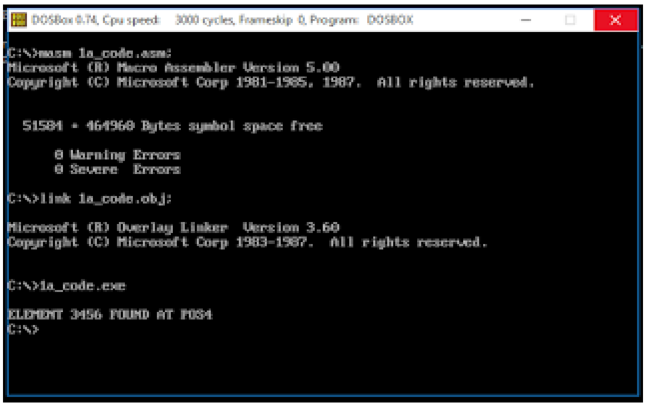
"masm program_name.asm;"

4. Link your object file of the code using the command,

"link program_name.obj;"

5. Run your code using commands:
"**program_name.exe**" to run it normally.
or type,
"**debug program_name.exe**" if you want to debug the code in the debugging screen.

1. Running normally using "program_name.exe"

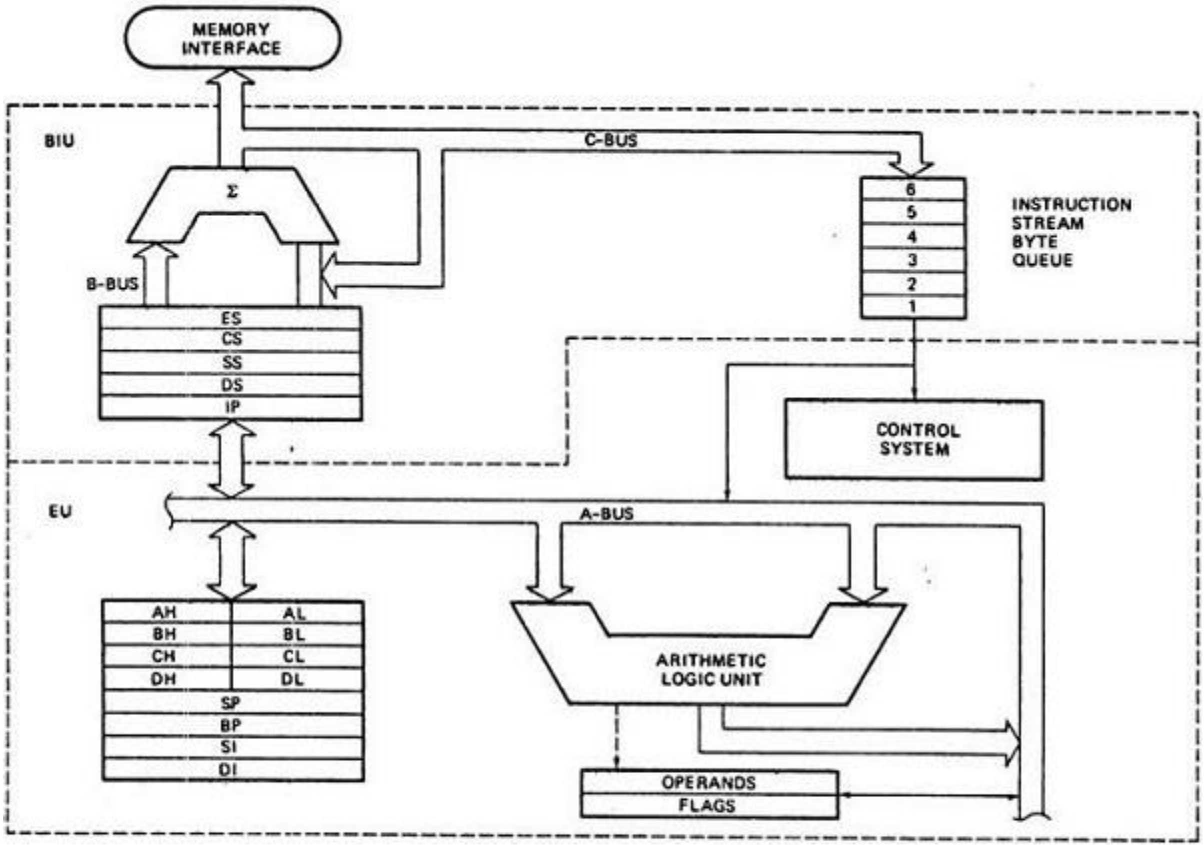


2. Running in debugging mode using "debug program_name.exe"

WEEK-1:

1) **ARCHITECTURE OF 8086 MICROPROCESSOR.**

Objective: To Draw and explain the Architecture of 8086 microprocessor.



8086 Microprocessor is divided into two functional units

2. **EU** (Execution Unit)
3. **BIU** (Bus Interface Unit).

1.**EU (Execution Unit)**

Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses it performs operations over data through BIU.

1.ALU

It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.

2.Flag Register

It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups − Conditional Flags and Control Flags.

2.1Conditional Flags

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags −

- **Carry flag** − This flag indicates an overflow condition for arithmetic operations.
- **Auxiliary flag** − When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.
- **Parity flag** − This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.
- **Zero flag** − This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.
- **Sign flag** − This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.
- **Overflow flag** − This flag represents the result when the system capacity is exceeded.

2.2 Control Flags

Control flags controls the operations of the execution unit. Following is the list of control flags −

- **Trap flag** − It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.
- **Interrupt flag** − It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.
- **Direction flag** − It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

3.General purpose register

There are 8 general purpose registers.(AX,BX,CX,DX,SP,BP,SI,DI)

These registers can be used individually to store 8-bit data and can be used in pairs to store 16bit data. The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. It is referred to the AX, BX, CX, and DX respectively.

- **AX register** − It is also known as accumulator register. It is used to store operands for arithmetic operations.
- **BX register** − It is used as a base register. It is used to store the starting base address of the memory area within the data segment.
- **CX register** − It is referred to as counter. It is used in loop instruction to store the loop counter.
- **DX register** − This register is used to hold I/O port address for I/O instruction.

Stack pointer register

It is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack.It points to the topmost item of the stack. If the stack is empty the stack pointer will be (FFFE)H. It's offset address relative to stack segment.

Base Pointer Register – . It is of 16 bits.
It is primary used in accessing parameters passed by the stack. It's offset address relative to stack segment.

Source Index Register – It is of 16 bits.
It is used in the pointer addressing of data and as a source in some string related operations.
It's offset is relative to data segment.

Destintion Index Register . It is of 16 bits.
It is used in the pointer addressing of data and as a destination in some string related operations.
It's offset is relative to extra segment.

2.BIU (Bus Interface Unit)

BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory. EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

It has the following functional parts −

3. **Instruction queue** − BIU contains the instruction queue. BIU gets upto 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.
4. Fetching the next instruction while the current instruction executes is called **pipelining**.
5. **Segment register** − BIU has 4 segment buses, i.e. CS, DS, SS& ES. It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to executed by the EU.
   - **CS** − It stands for Code Segment. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.
   - **DS** − It stands for Data Segment. It consists of data used by the program andis accessed in the data segment by an offset address or the content of other register that holds the offset address.
   - **SS** − It stands for Stack Segment. It handles memory to store data and addresses during execution.
   - **ES** − It stands for Extra Segment. ES is additional data segment, which is used by the string to hold the extra destination data.
6. **Instruction pointer** − It is a 16-bit register used to hold the address of the next instruction to be executed.

2) INSTRUCTION SET OF 8086 MICROPROCESSOR.

Objective: To explain the function of each instruction with an example.

Instructions are classified on the basis of functions they perform. They are categorized into the following main types:

1.Data Transfer instruction

All the instructions which perform data movement come under this category. The source data may be a register, memory location, port etc. the destination may be a register, memory location or port.

Instruction	Description
MOV	Moves data from register to register, register to memory, memory to register, memory to accumulator, accumulator to memory, etc.
LDS	Loads a word from the specified memory locations into specified register. It also loads a word from the next two memory locations into DS register.
LES	Loads a word from the specified memory locations into the specified register. It also loads a word from next two memory locations into ES register.
LEA	Loads offset address into the specified register.
LAHF	Loads low order 8-bits of the flag register into AH register.
SAHF	Stores the content of AH register into low order bits of the flags register.
XLAT/XLATB	Reads a byte from the lookup table.
XCHG	Exchanges the contents of the 16-bit or 8-bit specified register with the contents of AX register, specified register or memory locations.
PUSH	Pushes (sends, writes or moves) the content of a specified register or memory location(s) onto the top of the stack.
POP	Pops (reads) two bytes from the top of the stack and keeps them in a specified register, or memory location(s).
POPF	Pops (reads) two bytes from the top of the stack and keeps them in the flag register.
IN	Transfers data from a port to the accumulator or AX, DX or AL register.
OUT	Transfers data from accumulator or AL or AX register to an I/O port identified by the second byte of the instruction.

2.Arithmetic Instructions

Instructions of this group perform addition, subtraction, multiplication, division, increment, decrement, comparison, ASCII and decimal adjustment etc.423ory of Java

Instruction	Description
ADD	Adds data to the accumulator i.e. AL or AX register or memory locations.
ADC	Adds specified operands and the carry status (i.e. carry of the previous stage).
SUB	Subtract immediate data from accumulator, memory or register.
SBB	Subtract immediate data with borrow from accumulator, memory or register.
MUL	Unsigned 8-bit or 16-bit multiplication.
IMUL	Signed 8-bit or 16-bit multiplication.
DIV	Unsigned 8-bit or 16-bit division.

IDIV	Signed 8-bit or 16-bit division.
INC	Increment Register or memory by 1.
DEC	Decrement register or memory by 1.
DAA	**Decimal Adjust after BCD Addition:** When two BCD numbers are added, the DAA is used after ADD or ADC instruction to get correct answer in BCD.
DAS	**Decimal Adjust after BCD Subtraction:** When two BCD numbers are added, the DAS is used after SUB or SBB instruction to get correct answer in BCD.
AAA	**ASCII Adjust for Addition:** When ASCII codes of two decimal digits are added, the AAA is used after addition to get correct answer in unpacked BCD.
AAD	**Adjust AX Register for Division:** It converts two unpacked BCD digits in AX to the equivalent binary number. This adjustment is done before dividing two unpacked BCD digits in AX by an unpacked BCD byte.
AAM	**Adjust result of BCD Multiplication:** This instruction is used after the multiplication of two unpacked BCD.
AAS	**ASCII Adjust for Subtraction:** This instruction is used to get the correct result in unpacked BCD after the subtraction of the ASCII code of a number from ASCII code another number.
CBW	Convert signed Byte to signed Word.
CWD	Convert signed Word to signed Doubleword.
NEG	Obtains 2's complement (i.e. negative) of the content of an 8-bit or 16-bit specified register or memory location(s).
CMP	Compare Immediate data, register or memory with accumulator, register or memory location(s).

3.Logical Instructions

Instruction of this group perform logical AND, OR, XOR, NOT and TEST operations.

Instruction	Description
AND	Performs bit by bit logical AND operation of two operands and places the result in the specified destination.
OR	Performs bit by bit logical OR operation of two operands and places the result in the specified destination.
XOR	Performs bit by bit logical XOR operation of two operands and places the result in the specified destination.
NOT	Takes one's complement of the content of a specified register or memory location(s).
TEST	Perform logical AND operation of a specified operand with another specified operand.

4.Rotate Instructions

Instruction	Description
RCL	Rotate all bits of the operand left by specified number of bits through carry flag.
RCR	Rotate all bits of the operand right by specified number of bits through carry flag.
ROL	Rotate all bits of the operand left by specified number of bits.
ROR	Rotate all bits of the operand right by specified number of bits.

5.Shift Instructions

Instruction	Description

SAL or SHL	Shifts each bit of operand left by specified number of bits and put zero in LSB position.
SAR	Shift each bit of any operand right by specified number of bits. Copy old MSB into new MSB.
SHR	Shift each bit of operand right by specified number of bits and put zero in MSB position.

6.Branch Instructions

It is also called program execution transfer instruction. Instructions of this group transfer program execution from the normal sequence of instructions to the specified destination or target. The following instructions come under this category:

Instruction	Description
JA or JNBE	Jump if above, not below, or equal i.e. when CF and ZF = 0
JAE/JNB/JNC	Jump if above, not below, equal or no carry i.e. when CF = 0
JB/JNAE/JC	Jump if below, not above, equal or carry i.e. when CF = 0
JBE/JNA	Jump if below, not above, or equal i.e. when CF and ZF = 1
JCXZ	Jump if CX register = 0
JE/JZ	Jump if zero or equal i.e. when ZF = 1
JG/JNLE	Jump if greater, not less or equal i.e. when ZF = 0 and CF = OF
JGE/JNL	Jump if greater, not less or equal i.e. when SF = OF
JL/JNGE	Jump if less, not greater than or equal i.e. when SF ≠ OF
JLE/JNG	Jump if less, equal or not greater i.e. when ZF = 1 and SF ≠ OF
JMP	Causes the program execution to jump unconditionally to the memory address or label given in the instruction.
CALL	Calls a procedure whose address is given in the instruction and saves their return address to the stack.
RET	Returns program execution from a procedure (subroutine) to the next instruction or main program.
IRET	Returns program execution from an interrupt service procedure (subroutine) to the main program.
INT	Used to generate software interrupt at the desired point in a program.
INTO	Software interrupts to indicate overflow after arithmetic operation.
LOOP	Jump to defined label until CX = 0.
LOOPZ/LOOPE	Decrement CX register and jump if CX ≠ 0 and ZF = 1.
LOOPNZ/LOOPNE	Decrement CX register and jump if CX ≠ 0 and ZF = 0.

CF=Carry Flag
ZF=Zero Flag
OF=Overflow Flag
SF=Sign Flag

7.Flag Manipulation and Processor Control Instructions

Instructions of this instruction set are related to flag manipulation and machine control. The following instructions come under this category:

Instruction	Description
CLC	**Clear Carry Flag:** This instruction resets the carry flag CF to 0.

CLD	**Clear Direction Flag:** This instruction resets the direction flag DF to 0.
CLI	**Clear Interrupt Flag:** This instruction resets the interrupt flag IF to 0.
CMC	This instruction take complement of carry flag CF.
STC	Set carry flag CF to 1.
STD	Set direction flag to 1.
STI	Set interrupt flag IF to 1.
HLT	Halt processing. It stops program execution.
NOP	Performs no operation.
ESC	**Escape:** makes bus free for external master like a coprocessor or peripheral device.
WAIT	When WAIT instruction is executed, the processor enters an idle state in which the processor does no processing.
LOCK	It is a prefix instruction. It makes the LOCK pin low till the execution of the next instruction.

8.String Instructions

String is series of bytes or series of words stored in sequential memory locations. The 8086 provides some instructions which handle string operations such as string movement, comparison, scan, load and store.

Instruction	Description
MOVS/MOVSB/MOVSW	Moves 8-bit or 16-bit data from the memory location(s) addressed by SI register to the memory location addressed by DI register.
CMPS/CMPSB/CMPSW	Compares the content of memory location addressed by DI register with the content of memory location addressed by SI register.
SCAS/SCASB/SCASW	Compares the content of accumulator with the content of memory location addressed by DI register in the extra segment ES.
LODS/LODSB/LODSW	Loads 8-bit or 16-bit data from memory location addressed by SI register into AL or AX register.
STOS/STOSB/STOSW	Stores 8-bit or 16-bit data from AL or AX register in the memory location addressed by DI register.
REP	Repeats the given instruction until $CX \neq 0$

REPE/ REPZ	Repeats the given instruction till CX ≠ 0 and ZF = 1
REPNE/REPNZ	Repeats the given instruction till CX ≠ 0 and ZF = 0

WEEK-2

1. **Write a program to display string "Computer Science and Engineering" for 8086.**

Objective: To write a program to display string "Computer Science and Engineering for 8086.

Algorithm:

Step 1: Start
Step 2: Initialize the data memory
Step 3: Load the data into AX (Accumulator) register
Step 4: Move the data from AX(Accumulator) register to DS(Data Segment)
Step 5: Store the offset address in DX
Step 6: Initiate the appropriate interrupt to display
Step 7: Stop.

Program:

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
MSGE DB "Computer Science and Engineering$"
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV DX,OFFSET MSGE
MOV AH,09H
INT 21H
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

Input/Output:

Output: Computer Science and Engineering

2. **Write an Assembly Language Program (ALP) to display multiple strings line by line.**

Objective: To write a program to print multiple strings line by line.

Program:

```
ASSUME CS: CODE, DS:DATA
DATA SEGMENT
MSG DB "GEETHANJALI$"
    MSG1 DB 0AH,"CSE$"
    DATA ENDS
    CODE SEGMENT
    START:
    MOV AX, DATA
    MOV DS, AX
    MOV DX, OFFSET MSG
    MOV AH, 09H
    INT 21H
    LEA DX, MSG1
    INT 21H
    MOV AH, 4CH
     INT 21H
    CODE ENDS
    END START
```

Output:

GEETHANJALI
CSE

3. **WRITE AN ASSEMBLY LANGUAGE PROGRAM (ALP) TO FIND THE MAXIMUM OF THREE NUMBERS**

Objective: To write a program to find the maximum of three numbers.

Algorithm:

Step 1: Start
Step 2: Load Accumulator (A) with value1
Step 3: Load register B with value2
Step 4: Load register C with value3
Step 5: Compare B with A, gives carry if value2 is greater than value1 (i.e. B>A)
Step 6: When no carry from Step5: go to Step 7: (i.e. when A>B)
Step 7: Move content of register B to A (i.e. when B>A)
Step 8: Compare C with A, gives carry if value3 is greater than value1 (i.e. C>A)
Step 9: When no carry from Step 8: go to Step10: (i.e. when A>C)
Step 10: Move content of register C to A (i.e. when C>A)
Step 11: Store content of Accumulator to memory location 4200H
Step 12: Stop

Program:

```
        ASSUME CS:CODE, DS:DATA
org 100h
DATA SEGMENT
        LIST DB 1H,5H,3H
        COUNT EQU 03H
        MAX DB 01H DUP(?)
DATA ENDS
CODE SEGMENT
        START:
                MOV AX,DATA
                MOV DS,AX
                MOV SI,OFFSET LIST
                MOV CL, COUNT
                MOV AL, [SI]
        AGAIN:
                CMP AL,[SI+1]
                JNL NEXT
                MOV AL,[SI+1]
        NEXT:
                INC SI
                DEC CL
                CMP CL,0H
                JG AGAIN
                mov bl,al
                MOV AH, 4CH
                mov al,0H
                INT 21H
CODE ENDS
END START
```

Input/Output:

Output: 5

WEEK-3

1. Write an Assembly Language Program (ALP) to print numbers from 0 to 9

Program

```
data segment

Data ends
Code segment
Assume cs: code, ds:data
Begin: mov ax,data
mov ds,ax
mov cx,10
mov dl,48

L1:MOV ah,2
   int 21h
inc dl
loop l1

 mov ah,4ch
int 21h
code ends
end begin
```

OUTPUT:

0123456789

2.Write an Assembly Language Program (ALP) to check whether a given number is even or odd

ASSUME CS:CODE,DS:DATA

```
DATA SEGMENT
NL1 DB 10,'ENTER NUMBER:$'
NL2 DB 10,'ODD$'
NL3 DB 10,'EVEN$'
SMLST DB ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
LEA DX,NL1
MOV AH,09H
INT 21H
MOV AH,01H
INT 21H
SUB AL,30H
TEST AX,01H
JE SKIP1
LEA DX,NL2
MOV AH,09H
INT 21H
JMP SKIP2
SKIP1:
LEA DX,NL3
MOV AH,09H
INT 21H
SKIP2:
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

AH

AL

AX

9, 1, 30, 1

OUTPUT:

ENTER NUMBER: 3
ODD

Week 4:

1. Write an Assembly Language Program (ALP) to find the factorial of a number

Objective: To write a program to find the factorial of a numbers for 8086.

Algorithm:

Step 1: Start.
Step 2: Initialize data segment.
Step 3: Get the number in AL.
Step 4: Multiply the number with 8-bit number present in CL.
Step 5: Increment the counter.
Step 6: Compare with no.1
Step 7: Display factorial of number.
Step 8: Stop.

Program:

```
ASSUME CS:CODE
CODE SEGMENT
START:
            MOV CX,5H
            MOV AX,1H
    NEXT:
            MUL CX
            DEC CX
            CMP CX,1H
            JNZ NEXT

            MOV AH,4CH
            MOV BL,AL
            MOV AL,0H
            INT 21H
CODE ENDS
END START
```

Input/Output:

Output: 78H

2. Write an Assembly Language Program (ALP) to print fibo series up to 5 number

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
```

```
NL1 DB 10,'ENTER NUMBER:$'
NL2 DB 10,'$'
TEMP DB ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
LEA DX,NL1
MOV AH,09H
INT 21H
MOV AH,01H
INT 21H
SUB AL,30H        same
MOV CL,AL
MOV AL,0          C A$
MOV BL,1
LBL1:
MOV TEMP,AL
LEA DX,NL2
9  MOV AH,09H     3+1
INT 21H
ADD AX,3030H
MOV DX,AX         3+1
2  MOV AH,02H
INT 21H
MOV AL,TEMP
ADD AL,BL
MOV BL,TEMP       5+1
LOOP LBL1
4  MOV AH,4CH
INT 21H
CODE ENDS
END START
```

<u>OUTPUT:</u>

ENTER NUMBER:5
0
1
1
2
3

Week 5:

1. Write an Assembly Language Program (ALP) to take n values from user and calculate their sum.(BL contains the result)

Objective: To write a program to take n values from user and calculate their sum for 8086. BL contains the result

Algorithm:

Step 1: Start
Step 2: Initialize the data memory
Step 3: Load the data into AX (Accumulator) register
Step 4: Move the data from AX (Accumulator) register to DS (Data Segment)
Step 5: Load Effective Addresses from NL1 to Dx (Accumulator)
Step 6: MOV 09H to AH
Step 7: MOV 01H to AH
Step 8: SUB 30H to AL
Step 9: MOV CL to AL
Step 10: MOV AL to BL
Step 11: MOV 00 to AL
Step 12: MOV AL to VAL1
Step 13:LBL1-Load Effective Addresses from NL2 to DX\
Step 14: MOV 09H to AH
Step 15: MOV 01H to AH
Step 16:SUB 30H to AL
Step 17: ADD VAL1 to AL
Step 18:MOV AL to VAL1
Step 19: LOOP LBL1
Step 20: MOV 00 to AX
Step 21: MOV VAL1 to BL
Step 22: MOV 4CH to AH.
Step 23:Stop

Program:

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
      VAL1    DB     ?
      NL1     DB     0AH,0DH,'ENTER HOW MANY NO U WANT:','$'
      NL2     DB     0AH,0DH,'ENTER NO:','$'
DATA ENDS
CODE SEGMENT
MAIN    PROC
      MOV AX,DATA
      MOV DS,AX

      LEA DX,NL1
      MOV AH,09H
      INT 21H
```

Sand

```
        MOV AH,01H
        INT 21H
        SUB AL,30H

        MOV CL,AL
        MOV BL,AL
        MOV AL,00
        MOV VAL1,AL

LBL1:
        LEA DX,NL2
        MOV AH,09H
        INT 21H

        MOV AH,01H
        INT 21H
        SUB AL,30H

        ADD AL,VAL1
        MOV VAL1,AL
        LOOP LBL1

        MOV AX,00
        MOV BL,VAL1
        MOV AH,4CH
        INT 21H


MAIN    ENDP
CODE ENDS
        END     MAIN
```

2. Write an Assembly Language Program (ALP) to take n values from user and calculate maximum and minimum

<u>Objective:</u> To write a program to take n values from user and calculate maximum and minimum values for 8086.

<u>Algorithm:</u>

Step 1: Start
Step 2: Initialize the data memory
Step 3: Load the data into AX (Accumulator) register
Step 4: Move the data from AX (Accumulator) register to DS (Data Segment)
Step 5: Load effective addresses from NL1 to DX
Step 6:Mov 09H to AH
Step 7:INt 21H
Step 8: Mov 01H to AH
Step 9: INT 21H
Step 10:SUB 30H,AL
Step 11:MOV AL,CL
Step 12: MOV AL,BL

Step 13:MOV 00,AL
Step 14:MOV AL,VAL1
Step 15:LBL1:Load Effective Addresses from NL2 to DX
Step 16:MOV 09H to AH
Step 17:INT 21H
Step 18:MOV 01H to AH
Step 19:INT 21H
Step 20:SUB 30H to AL
Step 21:CMP MOV VAL1 to BH and MOV VAL1 to BL
Step 22:MOV AL to VAL1
Step 23:MOV 00 to AX
Step 24:MOV VAL1 to BL
Step 25:MOV 4CH to AH
Step 26:INT 21H
Step 27:STOP

Program:

```
DATA SEGMENT
    VAL1   DB    ?
    NL1    DB    0AH,0DH,'ENTER HOW MANY NO U WANT:','$'
    NL2    DB    0AH,0DH,'ENTER NO:','$'

DATA ENDS
CODE SEGMENT
MAIN   PROC
    MOV AX,DATA
    MOV DS,AX

    LEA DX,NL1
    MOV AH,09H
    INT 21H

    MOV AH,01H
    INT 21H
    SUB AL,30H

    MOV CL,AL
    MOV BL,AL
    MOV AL,00
    MOV VAL1,AL

LBL1:
    LEA DX,NL2
    MOV AH,09H
    INT 21H

    MOV AH,01H      } Same
    INT 21H
    SUB AL,30H

    CMP
```

```
        MOV BH,VAL1
        MOV BL,VAL1

    MOV VAL1,AL
    LOOP LBL1


    MOV AX,00
    MOV BL,VAL1
    MOV AH,4CH
    INT 21H


MAIN    ENDP
CODE ENDS
    END     MAIN
```

Input/Output:

Week 6:

1. **Write 8086 Assembly Language Program (ALP) to transfer a block of data from one location to another.**

   Objective: To write a program to transfer a block of data from one location to another

Algorithm:

Step 1:Start
Step 2:MOV 08H to CX
Step 3:MOV 10H to BX
Step 4: MOV 21H to DX
Step 5: MOV bx to SI
Step 6: MOV dx to DI
Step 7: MOV CL to AL
Step 8:STORE
Step 9: MOV AL to ptr[SI]
Step 10: dec AL
Step 11:cmp 0H to AL
Step 12:jne store.
Step 13:Again MOV ptr[SI] to AL
Step 14:MOV AL to ptr[DI]
Step 15:Loop again
Step 16:MOV 4ch to AH
Step 17:INT 21H

Program:

```
org 100H
assume cs:code
code segment
   start:
MOV Cx, 08H
MOV BX, 10H
MOV DX, 21H

mov si,bx
mov di,dx

mov al,cl
store:
mov byte ptr[si],al
dec al
cmp al,0h
jne store

again:
MOV Al, byte ptr[si]
MOV byte ptr[Di], Al
loop again

mov ah,4ch
int 21H
```

```
code ends
end start
```

Input/Output

AX=4C01 BX=0010 CX=0000 DX=0021 SP=0000 BP=0000 SI=0010 DI=0021

2. Write an Assembly Language Program (ALP) to reverse the given string.

Objective: To write a program to reverse the given string for 8086

Algorithm:

Step 1 : Initialize the data segment(DS)
Step 2 : In the Data segment . initialize element in an array named as Src,initialize the empty array size as DS and Count the value
Step 3 : In code segment move the data segment value to data segment register
Step 4 : Move count value (count +1) to count register and define offset address of destination to DI and move 04 H to DX
Step 5 : Define offset address of src to SI then move SI to BX and then BX to DX
Step 6 : Decrement the destination index then subtract source index value
Step 7 : Decrement CX if non zero go to step V
Step 8 : Store the result
Step 9 : Stop

Program:

```
Data Segment
  ostr db 'Computer','$'
  slen dw $-ostr
  rstr db 20 dup(' ')
Data Ends

Code Segment
  Assume cs:code, ds:data
  Begin:
    mov ax, data
    mov ds, ax
    mov es, ax
    mov cx, slen
    add cx, -2
    lea si, ostr
    lea di, rstr
    add si, slen
    add si, -2
    L1:
       mov al, [si]
       mov [di], al
       dec si
       inc di
```

```
    loop L1
    mov al, [si]
    mov [di], al
    inc di
    mov dl, '$'
    mov [di], dl
  Print:
    mov ah, 09h
    lea dx, rstr
    int 21h
  Exit:
    mov ax, 4c00h
    int 21h
Code Ends
End Begin
```

Input/Output:

retupmoc

3.. Write an Assembly Language Program (ALP) to perform addition of two 2X2 matrices.

Objective: To write a program to perform Addition of 2X2 matrices.

Algorithm:

Step1: Store 500 to SI and 601 to DI and Load data from offset 500 to register CL and set register CH to 00 (for count).
Step2: Increase the value of SI by 1.
Step3: Load first number (value) from next offset (i.e 501) to register AL.
Step4: Add the value in register AL by value at offset DI.
Step5: Store the result (value of register AL) to memory offset SI.
Step6: Increase the value of SI by 1.
Step7: Increase the value of DI by 1.
Step8: Loop above 5 till register CX gets 0.

Program:

```
MODEL
SMALL
.DATA
M DB 01H,02H,03H,04H
N DB 05H,06H,07H,08H
CNT DB 04H
RES DB ?
.CODE
START: MOV AX,@DATA
MOV DS,AX
MOV CL,CNT
LEA SI ,M
LEA DI,N
;Matrix Adiition Part
L1:MOV AL,[SI]
MOV BL,[DI]
ADD AL,BL
DAA
PUSH AX
INC SI
INC DI
LOOP L1
```



```
CALL STORE
;Store Part
```

```
    STORE PROC
    POP AX
    LEA SI,RES
    ADD SI,04H
    MOV CL,04H
    L2:POP AX
    MOV [SI],AL
    DEC SI
    LOOP L2
    INT 3H
    RET
    STORE ENDP
END START
```

Week 7:

1. Write an Assembly Language Program (ALP) for linear search.

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
LIST DB 2H,3H,5H,8H
E DB ?
COUNT DB 4H
M DB 10,"ENTER ELEMENT $"
MSG DB 10," ELEMENT FOUND $"
MSG1 DB  10,"ELEMENT NOT FOUND $"
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
LEA DX,M
MOV AH,09H
INT 21H
MOV AH,01H
INT 21H
SUB AL,30H
MOV E,AL
MOV SI, OFFSET LIST
MOV CL,COUNT
MOV AL,E
FIRST:
CMP AL,[SI]
JE NEXT
INC SI
LOOP FIRST
LEA DX,MSG1
MOV AH,09H
INT 21H
MOV AH,4CH
INT 21H
NEXT:
LEA DX,MSG
MOV AH,09H
INT 21H
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

INPUT/OUTPUT:

Enter Element:6
ELEMENT NOT FOUND

2. Write an Assembly Language Program (ALP) to take n values from user and sort them in ascending order

```
ASSUME CS:CODE, DS:DATA
org 100H
DATA SEGMENT
   V DB 9,6,7,8
DATA ENDS
CODE SEGMENT
   START:
                  mov ax,data
                  mov ds,ax
                  MOV CH,4
                  MOV AL,0
         ITER:
                  MOV BX,0
         NEXT:
                  MOV AL,V[BX]
                  CMP AL,V[BX+1]
                  JNL SWAP
                  JL CON
         SWAP:
                  MOV DL,V[BX]
                  MOV AL,V[BX+1]
                  MOV V[BX+1],DL
                  MOV V[BX],AL
         CON:
                  INC BX
                  CMP BX,3
                  JNZ NEXT
                  DEC CH
                  CMP CH,0
                  JNZ ITER
                  MOV CL,04H
                  MOV BL,0H
         DISP:
                  MOV DL,V[BX]
                  ADD DL,30H
                  INC BX
                  MOV AH,02H
                  INT 21H
                  LOOP DISP

                  MOV AH,4CH
                  INT 21H
CODE ENDS
END START
```

Output:
6789

ADDITIONAL PROGRAMS

1. Write an as assembly language program to evaluate Arithmetic Expression using 8 bit and 16bit

   i) a = b + c – d * e

   ii) z = x * y + w – v +u / k

PROGRAM:

i) a=b+c-d*e

```
 ASSUME CS:CODE, DS:DATA
DATA SEGMENT
B DB 02
C DB 04
D DB 01
E DB 03
A DB 01 DUP()
DATA ENDS

CODE SEGMENT
START:MOV AX,DATA
MOV DS,AX
MOV AL,B
MOV BL,C
ADD AL,BL
MOV CL,D
SUB AL,CL
MOV DL,E
MUL DL
MOV A,AL
INT 03
CODE ENDS
END START
```

RESULT:AX=000F

ii) Z=X*Y+W-V+U/K

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
X DB 02
Y DB 04
W DB 09
V DB 03
U DW 0006
K DB 03
Z DB 01 DUP()
DATA ENDS
CODE SEGMENT
START:MOV AX,DATA
MOV DS,AX
MOV AL,X
MOV BL,Y
MUL BL
MOV CL,W
ADD AL,CL
MOV DL,V
SUB AL,DL
MOV CL,AL
MOV AX,U
MOV BL,K
DIV BL
ADD CL,AL
MOV Z,CL
INT 03
CODE ENDS
END START
```

RESULT:CL=0010

EXPERIMENT 2

AIM: Write an ALP of 8086 to take N numbers as input. And do the following operations on them.

a. Arrange in Descending order
b. Arrange in ascending

PROGRAM:

1) **Descending order**

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
            LIST DB 02H,05H,01H, 07H, 04H, 03H,06H
            COUNT EQU 0007H
DATA ENDS
CODE SEGMENT
START: MOV AX, DATA
              MOV DS, AX
              MOV DX,COUNT
AGAIN:MOV CX, DX
             MOV SI,OFFSET LIST
BACK: MOV AL,[SI]
            CMP AL, [SI+1]
           JG NEXT
           XCHG  AL,[SI+1]
           XCHG AL,[SI]
NEXT:INC SI
          LOOP BACK
          DEC DX
          JNZ AGAIN
          INT 03
CODE ENDS
END START
```

RESULT:

```
  -g         ax=0B04  bx=0000 cx=0000 dx=0000 sp=0000 bp=0000 si=0006 di=0000
         ds= 0B57 es=0B47    ss=0B57  cs=0B58  ip=001F

-d ds:0   04 03 02 01
```

2) **Ascending order**

```
 ASSUME CS: CODE, DS: DATA
DATA SEGMENT
       LIST DB 02H, 01H, 04H, 03H
      COUNT EQU 0003H
DATA ENDS
CODE SEGMENT
START: MOV AX, DATA
        MOV DS, AX
        MOV DX,COUNT
AGAIN: MOV CX, DX
        MOV SI,OFFSET LIST
BACK: MOV AL,[SI]
        CMP AL, [SI+1]
        JL NEXT
        XCHG  AL,[SI+1]
        XCHG AL,[SI]
NEXT:   INC SI
        LOOP BACK
       DEC DX
       JNZ AGAIN
       INT 03
CODE ENDS
END START
```

RESULT:

```
-g     ax=0B01  bx=0000 cx=0000 dx=0000 sp=0000 bp=0000 si=0006 di=0000
    ds= 0B57 es=0B47    ss=0B57  cs=0B58  ip=001F

-d ds:0   01 02 03 04
```

EXPERIMENT 3

AIM: Write an ALP of 8086 to take N numbers as input and find average

PROGRAM:

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
        LIST DB 02H, 01H, 04H, 03H
        COUNT EQU 0003H
 DATA ENDS
 CODE SEGMENT
 START:  MOV AX, DATA
         MOV DS, AX
         MOV AX, 0000H
         MOV CX, COUNT
         MOV SI,OFFSET LIST
         MOV AL,[SI]
AGAIN:   ADD AL,[SI+1]
INC SI
         DEC CX
 JNZ AGAIN
         MOV CL,04H
         DIV CL
         INT 03H
CODE ENDS
END START
```

RESULT:

```
 -g     ax=0003  bx=0000 cx=0000 dx=0000 sp=0000 bp=0000 si=0006di=0000
     ds: 10a3  es=1093 ss=10a3 cs=10a4 ip=00231 -d ds:0  02 01 04 03 04
-u cs:0
```

EXPERIMENT 4

AIM: Write an ALP of 8086 to take a string of as input Find the length of the string

PROGRAM:

```
ASSUME CS:CODE,DS:DATA
DATA SEGMENT
        STRING DB "MASM$"
        REFNO EQU '$'
        COUNT EQU 0000
DATA ENDS
CODE SEGMENT
START:MOV AX,DATA
        MOV DS,AX
        MOV CX,COUNT
        MOV SI,OFFSET STRING
        MOV AL,REFNO
BACK: CMP AL,[SI]
        JE STOP
        INC SI
        INC CX
        JNZ BACK
STOP: MOV AX,CX
        INT 03
CODE ENDS
END START
```

RESULT: -g and press enter

CX= 0004.

EXPERIMENT 5

AIM: Write an ALP of 8086 to take a string of as input Find the given string is Palindrome or not

PROGRAM:

```
ASSUME DS:DATA, CS:CODE, ES:EXTRA
DATA SEGMENT
        STRING1 DB "MADAM"
        STRLEN EQU ($-STRING1)
        MSG1 DB "THE GIVEN STRING IS A PALINDROME$"
        MSG2 DB "THE GIVEN STRING IS NOT A PALINDROME$"
DATA ENDS
EXTRA SEGMENT
        STRING2 DB STRLEN DUP(0)
EXTRA ENDS
CODE SEGMENT
START:MOV AX, DATA
        MOV DS, AX
        MOV AX, EXTRA
        MOV ES, AX
        MOV AX, 0000H
        MOV SI, OFFSET STRING1
        MOV DI, OFFSET STRING2
        ADD DI, STRLEN-1
        MOV CX, STRLEN
  L1:   MOV AL, DS:[SI]
        MOV ES:[DI], AL
        INC SI
        DEC DI
        LOOP L1
        MOV SI,OFFSET STRING1
        MOV DI,OFFSET STRING2
        MOV CX, STRLEN
        CLD
        REP CMPSB
        JE PAL
        MOV DX, OFFSET MSG2
        MOV AH,09
        INT 21H
        JMP NEXT
 PAL: MOV DX, OFFSET MSG1
        MOV AH, 09H
         INT 21H
        NEXT:INT 03
CODE ENDS
END START
```

RESULT:

MASM-The given string is not a palindrome.

RADAR- The given string is a palindrome.