A Combined Report of the Experiments Conducted Under

Microprocessor Laoratory

By

XYZ
Seat No: ****

SE (Computer Engineering)

Department of Computer Engineering
Sinhgad College of Engineering,

Vadgaon (Bk.), Pune-411041

Affiliated to Savitribai Phule Pune University, Pune.

2021-22

Sinhgad Technical Education Society,
Sinhgad College of Engineering , Pune-41
Department of Computer Engineering

Sinhgad Institutes

CERTIFICATE

This is to Certify that XYZ has completed all the activities required for completion of course **Microprocessor Laboratory.**

Prof. Shweta Kambare	**Prof. M. P. Wankhade**	**Dr S. D. Lokhande**
Subject Incharge	**Head of Department of**	**Principal**
Department of Computer Engineering	**Computer Engineering**	**Sinhgad College of Engineering**

PROGRAMME OUTCOMES: As prescribed by NBA

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

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

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

4. Conduct investigations of complex problems: The problems that cannot be solved by straightforward application of knowledge, theories and techniques applicable to the engineering discipline.

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.

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.

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.

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

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

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with t h e 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.

11. Project management and finance: Demonstrate knowledge and understanding of t h e 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.

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.

COURSE OBJECTIVES:

1. To understand assembly language programming instruction set
2. To understand different assembly directives with example
3. To apply instruction set for implementing x86/64 bit assembly language programs

COURSE OUTCOMES:

On completion of this course the student should be able to:

CO1: **Understand** and **apply** various addressing modes and instruction sets to implement assembly language programs.

CO2: **Apply** logic to **implement** code conversion

CO3: **Analyze** and **apply** logic to **demonstrate** processor mode of operation

Savitribai Phule Pune University
Second Year of Computer Engineering (2019 Course)
210257: Microprocessor Laboratory

Teaching Scheme	Credit Scheme	Examination Scheme and Marks
Practical: 02 Hours/Week	01	Term Work: 25 Marks Practical: 25 Marks

Companion Course : 210254: Microprocessor

Course Objectives:

- To understand assembly language programming instruction set
- To understand different assembler directives with example
- To apply instruction set for implementing X86/64 bit assembly language programs

Course Outcomes:

On completion of the course, learner will be able to–

CO1. **Understand** and **apply** various addressing modes and instruction set to implement assembly language programs

CO2. **Apply** logic to **implement** code conversion

CO3. **Analyze** and **apply** logic to **demonstrate** processor mode of operation

Guidelines for Laboratory /Term Work Assessment

Continuous assessment of laboratory work is based on overall performance and Laboratory assignments performance of student. Each Laboratory assignment assessment will assign grade/marks based on parameters with appropriate weightage. Suggested parameters for overall assessment as well as each Laboratory assignment assessment include- timely completion, performance, innovation, efficient codes, punctuality and neatness.

Guidelines for Laboratory Conduction

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy need to address the average students and inclusive of an element to attract and promote the intelligent students. The instructor may set multiple sets of assignments and distribute among batches of students. It is appreciated if the assignments are based on real world problems/applications. Use of open source software is encouraged.

In addition to these, instructor may assign one real life application in the form of a mini-project based on the concepts learned. Instructor may also set one assignment or mini-project that is suitable to respective branch beyond the scope of syllabus.

Operating System: 64-bit Open source Linux or its derivative.

Programming Tools: Preferably using Linux equivalent or MASM/TASM/NASM/FASM.

Guidelines for Practical Examination

Both internal and external examiners should jointly set problem statements. During practical assessment, the expert evaluator should give the maximum weightage to the satisfactory implementation of the problem statement. The supplementary and relevant questions may be asked at the time of evaluation to test the student's for advanced learning, understanding of the fundamentals, effective and efficient implementation. So encouraging efforts, transparent evaluation and fair approach of the evaluator will not create any uncertainty or doubt in the minds of the students. So adhering to these principles will consummate our team efforts to the promising start of the student's academics.

Virtual Laboratory:

· http://209.211.220.205/vlabiitece/mi/MI3.php

Suggested List of Laboratory Experiments/Assignments(any 10)

Sr. No.	Assignments
1	Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers.
2	Write an X86/64 ALP to accept a string and to display its length.
3	Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.
4	Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation.
5	Write an X86/64 ALP to count number of positive and negative numbers from the array.
6	Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for: (a) HEX to BCD b) BCD to HEX (c) EXIT. Display proper strings to prompt the user while accepting the input and displaying the result. (Wherever necessary, use 64-bit registers).
7	Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.
8	Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

9	Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.
10	Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).
11	Write X86 Assembly Language Program (ALP) to implement following OS commands i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments
12	Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.
13	Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.
14	Write an X86/64 ALP password program that operates as follows: a. Do not display what is actually typed instead display asterisk ("*"). If the password is correct display, "access is granted" else display "Access not Granted"
15	Study Assignment: Motherboards are complex. Break them down, component by component, and Understand how they work. Choosing a motherboard is a hugely important part of building a PC. Study- Block diagram, Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio, Networking, Overclocking, and Cooling. 4. https://www.intel.in/content/www/in/en/support/articles/000006014/boards-and kits/desktop-boards.html

@The CO-PO Mapping Matrix

CO\PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	-	1	2	1	-	-	-	-	-	-	-	-
CO2	2	1	-	1	-	-	-	-	-	-	-	-
CO3	-	1	-	1	-	-	-	-	-	-	-	-

INDEX

Sr. No.	Assg. No.	Title of Assignment	CO	PO
1.	**1.**	To study Assembly Language Programming	**1**	**2,3,4**
2.	**2.**	Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers	**1**	**2,3,4**
3.	**3.**	Write an X86/64 ALP to accept a string and to display its length.	**1**	**2,3,4**
4.	**4.**	Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.	**1**	**2,3,4**
5.	**5.**	Write an X86/64 ALP to count number of positive and negative numbers from the array.	**1**	**2,3,4**
6.	**6.**	Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.	**1,2**	**1,2,3,4**
7.	**7**	Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.	**1,2**	**1,2,3,4**
8.	**8.**	Write X86/64 ALP to perform overlapped block transfer with string specific instructions. Block containing data can be defined in the datasegment.	**1,2**	**1,2,3,4**
9.	**9.**	Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).	**1,2**	**1,2,3,4**
10.	**10.**	Write X86 Assembly Language Program (ALP) to implement following OS commands i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments	**1,2,3**	**2,3,4**
11.	**11.**	Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.	**1,2**	**1,2,3,4**

Software Required:

1. OS- 64 bit Linux Mint
2. CPU- 64 bit core-2Duo
3. Assembler Used- NASM/ TASM
4. Linker Used- Ld (GNU Linker)

Write-ups must include:

- **Assignment No.**
- **Title**
- **Problem Statement**
- **Course Objectives**
- **Outcome**
- **Theory(in brief)**
- **Algorithm**
- **Test Cases**
- **Conclusion**
- **FAQs**
- **Output:**

ASSIGNMENT NO: 1

TITLE: To study Assembly Language Programming

PROBLEM STATEMENT: To study Assembly Language Programming

COURSE OBJECTIVE: To understand assembly language programming instruction set
To understand different assembly directives with example

OUTCOME: 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

THEORY:

Introduction to Assembly Language Programming:

Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities.

Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction'.

Processor understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in software development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

Advantages of Assembly Language

- An understanding of assembly language provides knowledge of:
- Interface of programs with OS, processor and BIOS;
- Representation of data in memory and other external devices;
- How processor accesses and executes instruction;
- How instructions accesses and process data;
- How a program access external devices.

Other advantages of using assembly language are:

- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;

ALP Step By Step:

Installing NASM:

If you select "Development Tools" while installed Linux, you may NASM installed along with the

Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

- Open a Linux terminal.
- Type ***whereis nasm*** and press ENTER.
- If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just*nasm:*, then you need to install NASM.

To install NASM take the following steps:

- Check the netwide assembler (NASM) website for the latest version.
- Download the Linux source archive nasm-X.XX. ta .gz, where X.XX is the NASM version number in the archive.
- Unpack the archive into a directory, which creates a subdirectory nasm-X. XX.
- cd to *nasm-X. XX* and type ***./configure*** . This shell script will find the best C compiler to use and set up Makefiles accordingly.
- Type ***make*** to build the nasm and ndisasm binaries.
- Type ***make install*** to install nasm and ndisasm in /usr/local/bin and to install the man pages.

This should install NASM on your system. Alternatively, you can use an RPM distribution for the Fedora Linux. This version is simpler to install, just double-click the RPM file.

Assembly Basic Syntax:

An assembly program can be divided into three sections:

- The **data** section
- The **bss** section
- The **text** section

The order in which these sections fall in your program really isn't important, but by convention the .data section comes first, followed by the .bss section, and then the .text section.

The .data Section

The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution. You don't have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the program as a whole into memory.

The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

```
section .data
```

The .bss Section

Not all data items need to have values before the program begins running. When you're reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk. Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don't say what values are to be present in the buffer.

There's a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

```
section .bss
```

The .text Section

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data items are defined in .text. The .text section contains symbols called *labels* that identify locations in the program code for jumps and calls, but beyond your instruction mnemonics, that's about it.

All global labels must be declared in the .text section, or the labels cannot be ''seen'' outside your program by the Linux linker or the Linux loader. Let's look at the labels issue a little more closely.

```
section .text
```

Labels

A label is a sort of bookmark, describing a place in the program code and giving it a name that's easier to remember than a naked memory address.

Labels are used to indicate the places where jump instructions should jump to, and they give names to callable assembly language procedures.

Here are the most important things to know about labels:

- *Labels must begin with a letter, or else with an underscore, period, or question mark.* These last three have special meanings to the assembler, so don't use them until you know how NASM interprets them.
- *Labels must be followed by a colon when they are defined.* This is basically what tells NASM that the identifier being defined is a label. NASM will punt if no colon is there and will not flag an error, but the colon nails it, and prevents a mistyped instruction mnemonic from being mistaken for a label. Use the colon!
- *Labels are case sensitive.* So yikes:, Yikes:, and YIKES: are three completely different labels. This differs from practice in a lot of other languages (Pascal particularly), so keep it in mind.

Assembly Language Statements

Assembly language programs consist of three types of statements:

- Executable instructions or instructions
- Assembler directives or pseudo-ops
- Macros

Syntax of Assembly Language Statements

```
[label]    mnemonic    [operands]                    [;comment]
```

Function of Assemblers, Linker and Loader:

Assembler (meaning *one that assembles*) may refer to:

- Assembler (computing), a computer program which translates from assembly language to an object file or machine language format

To assemble a file, you issue a command of the form

- nasm −f <format> <filename> [−o <output>]
- For example,
- nasm −f elf myfile.asm
- will assemble myfile.asm into an ELF object file myfile.o.

Linker

To translate the object files into executable programs, an appropriate linker must be used, such as the ld for UNIX-like systems.

- ld -m elf_i386 –o myfile myfile.o
- In computer science, a linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Loader

- In computing, a loader is the part of an operating system that is responsible for one of the essential stages in the process of starting a program, loading programs, that is, starting up programs by reading the contents of executable files (executables- files containing program text) into memory, then carrying out other required preparatory tasks, after which the program code is finally allowed to run and is started when the operating system passes control to the loaded program code.
- loads the executable code into memory for execution
- ./myfile

System Calls:

System calls are APIs for the interface between user space and kernel space. We have already used the system calls sys_write and sys_exit for writing into the screen and exiting from the program respectively.

The Hello World Program in Assembly

```
section .data
                msg    db  “Hello World”,0Ah
                msglen  equ $-msg
section .bss
section .text
                global _start
                _start:
;display
                MOV Rax, 1
                MOV Rdi,  1
                MOV Rsi,  msg
                MOV Rdx, msglen
                syscall
;Exit System call
                mov Rax,60
                        mov Rdi,0
                syscall
```

Compiling and Linking an Assembly Program in NASM

- Type the above code using a text editor and save it as hello.asm.
- Make sure that you are in the same directory as where you saved **hello.asm**.
- To assemble the program, type **nasm -f elf hello.asm**
- If there is any error, you will be prompted about that at this stage. Otherwise an object file of your program named **hello.o** will be created.
- To link the object file and create an executable file named hello, type **ld -m elf_i386 -o hello hello.o**
- Execute the program by typing **./hello**

If you have done everything correctly, it will display **Hello, world!** on the screen.

CONCLUSION: Hence we studied basic concepts of assembly language programming and displayed 'Hello World' message.

FAQs:

1. MINT is how many bit operating system?
Ans.

2. Assembly language is Low-level language or High-level language?
Ans.

3. List advantages of assembly language.
Ans.

4. List different sections in the assembly language program.
Ans.

5. How to write comments in program?

Ans:

Program and output

ASSIGNMENT NO: 2

TITLE: ALP to accept numbers in array from user and display

PROBLEM STATEMENT: Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers

COURSE OBJECTIVE: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME: 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

THEORY:

Microprocessor is the CPU of a microcomputer. It is a 16-bit Microprocessor(μp).It's ALU, internal registers works with 16bit binary words. 8086 has a 20 bit address bus can access up to 220= 1 MB memory locations. 8086 has a 16bit data bus. It can read or write data to a memory/port either 16bits or 8 bit at a time. It can support up to 64K I/O ports. It provides 14, 16 -bit registers. Frequency range of 8086 is 6-10 MHz It has a multiplexed address and data bus AD0- AD15 and A16 – A19. It requires a single phase clock with 33% duty cycle to provide internal timing. It can prefetch upto 6 instruction bytes from memory and queues them in order to speed up instruction execution. It requires +5V power supply. A 40 pin dual in line package. 8086 is designed to operate in two modes, Minimum mode and Maximum mode. The minimum mode is selected by applying logic 1 to the MN / MX# input pin. This is a single microprocessor configuration. The maximum mode is selected by applying logic 0 to the MN / MX# input pin. This is a multi micro processors configuration.

The different segment registers are:- CS Register: - Code segment.

DS register: - Data segment.

ES register: - Extra segment.

SS register: - Stack segment.

Terminology Used: memory - Refers to an 8 or 16-bit memory location determined by an effective address. register - AX, BX, CX, DX, SI, DI, BP, or SP as well as the 8-bit derivatives of AX, BX, CX, and DX (other registers or flags are not allowed). immediate - A numeric constant or label. REG1::REG2 - The concatenation of two registers (e.g., the 32-bit value DX::AX) A single colon is used for memory addresses. XF or XF=b - A flag's value after an instruction can be 0 or 1 and

usually depends on the result of the instruction. A flag being set to '?' by an instruction indicates that the flag is undefined after the operation.

ALGORITHM:

1. Declare array to store numbers
2. Accept 5-16digit numbers using system call
3. Display 5-16digit numbers using system call
4. Exit

TEST CASES:

Sr. NO	**Test ID**	**Steps**	**Input**	**Expected Result**	**Actual Result**	**Status (Pass/ Fail)**
1	**ID1**					
2	**ID2**					

CONCLUSION: Hence we accepted five 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers

FAQs:

1. What is an Array?

Ans.

2. Explain read system call?

Ans.

3. Explain exit system call ?

Ans:

4. Explain DB,DW,DQ,DD

Ans:-

5. How to calculate digits for 64 bit input

Ans:-

Program & Output

ASSIGNMENT NO: 3

TITLE: ALP to accept string and display its length

PROBLEM STATEMENT: Write an X86/64 ALP to accept a string and to display its length.

COURSE OBJECTIVE: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME: 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

THEORY:

The variable length strings can have as many characters as required. Generally, we specify the length of the string by either of the two ways –

- Explicitly storing string length
- Using a sentinel character

We can store the string length explicitly by using the $ location counter symbol that represents the current value of the location counter. In the following example –

```
msg  db  'Hello, world!',0xa ;our dear string

len  equ  $ - msg          ;length of our dear string
```

$ points to the byte after the last character of the string variable *msg*. Therefore, *$-msg* gives the length of the string. We can also write

```
msg db 'Hello, world!',0xa ;our dear string

len equ 13                 ;length of our dear string
```

Alternatively, you can store strings with a trailing sentinel character to delimit a string instead of storing the string length explicitly. The sentinel character should be a special character that does not appear within a string.

For example –

```
message DB 'I am loving it!', 0
```

String Instructions

Each string instruction may require a source operand, a destination operand or both. For 32-bit segments, string instructions use ESI and EDI registers to point to the source and destination

operands, respectively.

For 16-bit segments, however, the SI and the DI registers are used to point to the source and destination, respectively.

There are five basic instructions for processing strings. They are –

- MOVS – This instruction moves 1 Byte, Word or Doubleword of data from memory location to another.
- LODS – This instruction loads from memory. If the operand is of one byte, it is loaded into the AL register, if the operand is one word, it is loaded into the AX register and a doubleword is loaded into the EAX register.
- STOS – This instruction stores data from register (AL, AX, or EAX) to memory.
- CMPS – This instruction compares two data items in memory. Data could be of a byte size, word or doubleword.
- SCAS – This instruction compares the contents of a register (AL, AX or EAX) with the contents of an item in memory.

Each of the above instruction has a byte, word, and doubleword version, and string instructions can be repeated by using a repetition prefix.

These instructions use the ES:DI and DS:SI pair of registers, where DI and SI registers contain valid offset addresses that refers to bytes stored in memory. SI is normally associated with DS (data segment) and DI is always associated with ES (extra segment).

The DS:SI (or ESI) and ES:DI (or EDI) registers point to the source and destination operands, respectively. The source operand is assumed to be at DS:SI (or ESI) and the destination operand at ES:DI (or EDI) in memory.

For 16-bit addresses, the SI and DI registers are used, and for 32-bit addresses, the ESI and EDI registers are used.

ALGORITHM:

1. Accept string from user
2. ;When you accept string, length automatically returns in RAX register
3. Call display procedure, to display length

Display Procedure

When you want to display number on screen you need to convert number from hex to ascii format

Algorithm for number display on screen-Hex to Ascii conversion
Display Procedure

1. Take count value as 16 into cnt variable; count for 16 digits to display
2. Move address of “result” variable into rdi.
3. Rotate left rbx register by 4 bits. ;rbx reg =16 digit number to display
4. Move bl into al.
5. And al with 0FH.
6. Compare al with 09H.

7. if greater Add 37H into al.
8. Else Add 30H into al
9. Move al into memory location pointed by rdi.
10. Increment rdi.
11. Loop the statements till count value becomes zero.

Return from procedure.

TEST CASES:

Sr. NO	Test ID	Steps	Input	Expected Result	Actual Result	Status (Pass/ Fail)
1	ID1	Take hard coded dataand displayresult	WELCOME	0000000000000008	0000000000000008	Pass

CONCLUSION: Hence we accepted string and display its length

FAQs:

1. Why ascii to hex conversion required?

Ans.

2. What is string?

Ans.

3. Explain AND operation ?

Ans:

4. Explain RESB,RESW,RESQ,RESD?

Ans:-

5. What is Hex number?

Ans:-

Program & Output

ASSIGNMENT NO: 4

TITLE: ALP to find the largest of givenByte/Word/Dword/64-bit numbers.

PROBLEM STATEMENT: Write an x86/64 ALP to find largest of given BYTE/WORD/DWORD/ 64-bit numbers

COURSE OBJECTIVE: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME: 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

THEORY:

Numerical data is generally represented in binary system. Arithmetic instructions operate on binary data. When numbers are displayed on screen or entered from keyboard, they are in ASCII form.

So far, we have converted this input data in ASCII form to binary for arithmetic calculations and converted the result back to binary.

Such conversions, however, have an overhead, and assembly language programming allows processing numbers in a more efficient way, in the binary form. Decimal numbers can be represented in two forms –

- ASCII form
- BCD or Binary Coded Decimal form

ASCII Representation

In ASCII representation, decimal numbers are stored as string of ASCII characters. For example, the decimal value 1234 is stored as –

```
31    32    33    34H
```

Where, 31H is ASCII value for 1, 32H is ASCII value for 2, and so on. There are four instructions for processing numbers in ASCII representation –

- AAA – ASCII Adjust After Addition
- AAS – ASCII Adjust After Subtraction
- AAM – ASCII Adjust After Multiplication
- AAD – ASCII Adjust Before Division

These instructions do not take any operands and assume the required operand to be in the AL register.

ALGORITHM:

1. Declare array with numbers (input)
2. Take variable for counter
3. Assign rsi to array to get current number
4. assign 0 value to al register
5. compare value of al register with value pointed by rsi in array (Mov al.[rsi])
6. If number of rsi is less than al, then perform exchange operation
7. otherwise increment rsi and compare again with al
8. continue above comparison until counter becomes zero
9. largest number store in al
10. call display procedure to display largest number

TEST CASES:

Sr. NO	**Test ID**	**Steps**	**Input**	**Expected Result**	**Actual Result**	**Status (Pass/ Fail)**
1	ID1	Take hard coded data and display result	11h,22h,55h,33h,44h	55h	55h	Pass

CONCLUSION: Hence we accepted the numbers and find the largest number from them and display.

FAQs:

1. **What are the different registers present in 8086?**

Ans.

2. What are the different pointers and index registers in 8086?

Ans.

3. Explain logic for find largest of given number

Ans:

4. What is the syntax of MOV instruction?

Ans:-

5. What is flag?

Ans:-

Program & Output

ASSIGNMENT NO: 5

TITLE: ALP to count number of positive and negative numbers from the array

PROBLEM STATEMENT: Write X86/64 ALP to count number of positive and negative numbers from the array

COURSE OBJECTIVE: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME: 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

THEORY:

In computing, signed number representations are required to encode negative numbers in binary number systems.

The early days of digital computing were marked by a lot of competing ideas about both hardware technology and mathematics technology (numbering systems). One of the great debates was the format of negative numbers, with some of the era's most expert people having very strong and different opinions. One camp supported two's complement, the system that is dominant today. Another camp supported ones' complement, where any positive value is made into its negative equivalent by inverting all of the bits in a word. A third group supported "sign & magnitude" (sign-magnitude), where a value is changed from positive to negative simply by toggling the word's sign (high-order) bit.

In mathematics, negative numbers in any base are represented by prefixing them with a − sign. However, in computer hardware, numbers are represented in bit vectors only, without extra symbols. The four best-known methods of extending the binary numeral system to represent signed numbers are: sign -and- magnitude, ones' complement, two's complement, and excess-*K*. Some of the alternative methods use implicit instead of explicit signs, such as negative binary, using the base −2. Corresponding methods can be devised for other bases, whether positive, negative, fractional, or other elaborations on such themes. There is no definitive criterion by which any of the representations is universally superior. The representation used in most current computing devices is two's complement

ALGORITHM:

1. Initialize index register with the offset of array of signed numbers
2. Initialize ECX with array element count
3. Initialize positive number count and negative number count to zero
4. Perform MSB test of array element
5. If set jump to step 7
6. Increment positive number count and jump to step 8
7. Increment negative number count and continue
8. Point index register to the next element
9. Decrement the array element count from ECX, if not zero jump to step 4, else continue
10. Display Positive number message and then display positive number count
11. Display Negative number message and then display negative number count
12. EXIT

TEST CASES:

Sr. NO	**Test ID**	**Steps**	**Input**	**Expected Result**	**Actual Result**	**Status (Pass/ Fail)**
1	ID1	Take hard coded data and display result	855h, 90ffh, 87h, 88h, 8a9fh, 0afh, 02h	Count of Positive no.: 6 Count of Negative no.: 1	Count of Positive no.: 6 Count of Negative no.: 1	Pass

CONCLUSION: Hence we counted and displayed the number of positive and negative numbers from the array of signed numbers.

FAQs:

1. What is an Array?

Ans.

2. What is the working of instruction INC?

Ans.

3. Explain .data, .bss, .text ?

Ans:

4. What is Microprocessor

Ans:-

5. What is Assembly level language

Ans:-

Program & Output

ASSIGNMENT NO: 6

TITLE: ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

PROBLEM STATEMENT: Write X86/64 ALP to detect protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers also identify CPU type using CPUID instruction.

COURSE OBJECTIVES: To understand assembly language programming instruction set
To understand different assembly directives with example

OUTCOME : 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

2: **Apply** logic to **implement** code conversion.

THEORY :

REAL MODE ARCHITECTURE:

When the processor is reset or powered up it is initialized in Real Mode. Real Mode has the same base architecture as the 8086, but allows access to the 32-bit register set of the Intel386 DX. The addressing mechanism, memory size, interrupt handling, are all identical to the Real Mode on the 80286.

PROTECTED MODE ARCHITECTURE: The complete capabilities of the Intel386 DX are unlocked when the processor operates in Protected Virtual Address Mode (Protected Mode).

Protected Mode vastly increases the linear address space to four gigabytes (2 32 bytes) and allows the running of virtual memory programs of almost unlimited size (64 terabytes or 2 46 bytes). In addition Protected Mode allows the Intel386 DX to run all of the existing 8086 and 80286 software, while providing a sophisticated memory management and a hardware-assisted protection mechanism. Protected Mode allows the use of additional instructions especially optimized for supporting multitasking operating systems. The base architecture of the Intel386 DX remains the same, the registers, instructions, and addressing modes.

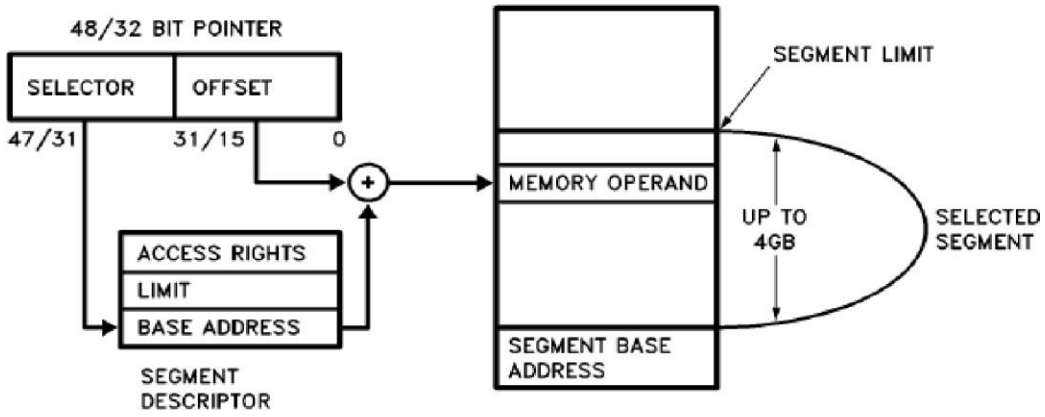
Protected Mode Addressing Mechanism:

DESCRIPTOR TABLES INTRODUCTION

The descriptor tables define all of the segments which are used in an Intel386 DX system. There are three types of tables on the Intel386 DX which hold descriptors: the Global Descriptor Table, Local Descriptor Table, and the Interrupt Descriptor Table. All of the tables are variable length memory arrays. They can range in size between 8 bytes and 64K bytes. Each table can hold up to 8192 8 byte descriptors.

The upper 13 bits of a selector are used as an index into the descriptor table. The tables have registers associated with them which hold the 32-bit linear base address, and the 16-bit limit of each table. Each of the tables has a register associated with it the GDTR, LDTR, and the IDTR.

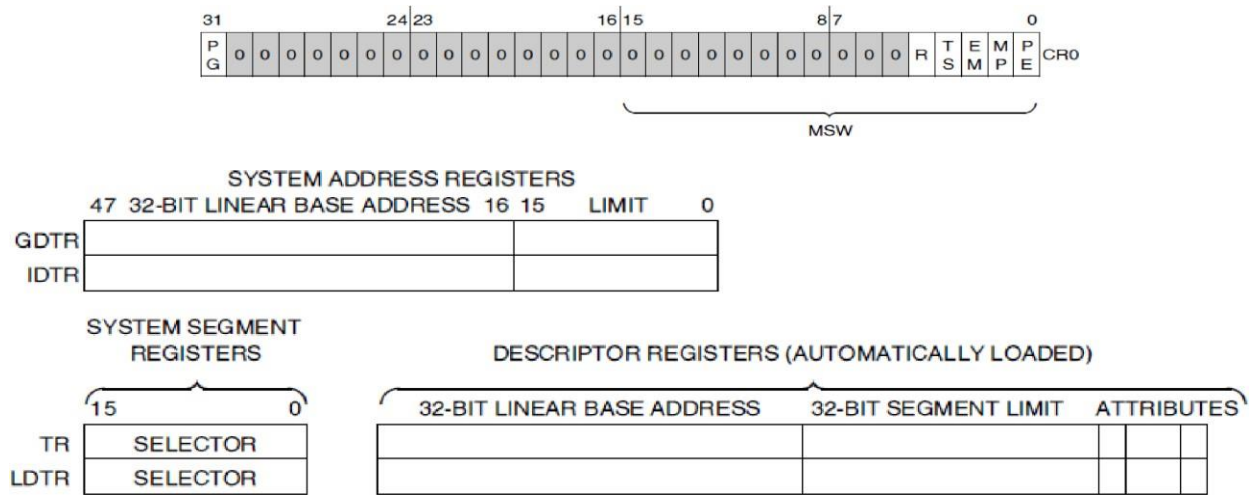
The LGDT, LLDT, and LIDT instructions, load the base and limit of the Global, Local, and Interrupt Descriptor Tables, respectively, into the appropriate register. The SGDT, SLDT, and SIDT instructions store the base and limit values. These tables are manipulated by the operating



system. Therefore, the load descriptor table instructions are privileged instructions. Local Descriptor Table Register (LDTR) and Task Register (TR): These registers hold the 16-bit selector for the LDT descriptor and the TSS descriptor, respectively. The LDT and TSS segments, since they are task specific segments, are defined by selector values stored in the system segment registers.

Note that a segment descriptor register (programmer-invisible) is associated with each system segment register.

CR0:



PG (Paging Enable, bit 31): The PG bit is set to enable the on-chip paging unit. It is reset to disable the on-chip paging unit.

R (reserved, bit 4): This bit is reserved by Intel. When loading CR0 care should be taken to not alter the value of this bit.

TS (Task Switched, bit 3): TS is automatically set whenever a task switch operation is performed.

EM (Emulate Coprocessor, bit 2): The EMulate coprocessor bit is set to cause all coprocessor opcodes to generate a Coprocessor Not Available fault (exception 7). It is reset to allow coprocessor opcodes to be executed on an actual Intel387 DX coprocessor (this is the default case after reset).

MP (Monitor Coprocessor, bit 1): The MP bit is used in conjunction with the TS bit to determine if the WAIT opcode will generate a Coprocessor Not Available fault (exception 7) when TS = 1. When both MP = 1 and TS = 1, the WAIT opcode generates a trap. Otherwise, the WAIT opcode does not generate a trap. Note that TS is automatically set whenever a task switch operation is performed.

PE (Protection Enable, bit 0): The PE bit is set to enable the Protected Mode. If PE is reset, the processor operates again in Real Mode. PE may be set by loading MSW or CR0. PE can be reset only by a load into CR0. Resetting the PE bit is typically part of a longer instruction sequence needed for proper transition from Protected Mode to Real Mode.

ALGORITHM:

1. Read CR0 i.e. MSW and store it in memory
2. Check LSB of same, if 0, display real mode message and jump to step 4 else continue
3. Display protected mode message

4. Read GDTR, IDTR, LDTR, TR in memory
5. Display GDTR, IDTR, LDTR, TR & MSW from memory
6. Use CPUID instruction
7. Exit

CONCLUSION: Hence we check the mode of operation of 80386 and read and displayed the GDTR, IDTR, LDTR, TR and MSW with appropriate messages.

FAQs:

1. What is Protected Mode?
Ans..

2. What is TR?
Ans.

1. How to enter into protected mode from real mode?

Ans.
4. What are the Interrupts of 8086?

Ans:
5. How labels are defined in an assembly program?
Ans: .

Program & Output

ASSIGNMENT NO: 7

TITLE: Write X86/64 Assembly language program (ALP) to perform non-overlapped block transfer.

PROBLEM STATEMENT: Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions. Block containing data can be defined in the data segment.

COURSE OBJECTIVES: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME : 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

2: **Apply** logic to **implement** code conversion.

THEORY

1. Registers:

Registers are places in the CPU where a number can be stored and manipulated. There are three sizes of registers: 8-bit, 16-bit and on 386 and above 32-bit. There are four different types of registers:

1. General Purpose Registers,

2. Segment Registers,

3. Index Registers,

4. Stack Registers.

General Registers – Following are the registers that are used for general purposes in 8086

AX accumulator (16 bit)

AH accumulator high-order byte (8 bit) AL accumulator low-order byte (8 bit)

BX accumulator (16 bit)

BH accumulator high-order byte (8 bit) BL accumulator low-order byte (8 bit)

CX count and accumulator (16 bit)

CH count high order byte (8 bit) CL count low order byte (8 bit)

DX data and I/O address (16 bit)

DH data high order byte (8 bit) DL data low order byte (8 bit)

Segment Registers - These registers are used to calculate 20 bit address from 16 bit registers.

CS code segment (16 bit) DS data segment (16 bit) SS stack segment (16 bit)

ES extra segment (16 bit)

Index Registers - These registers are used with the string instructions.

DI destination index (16 bit) SI source index (16 bit)

Pointers - These regist ister to obtain 20 bit addresses

SP stack pointer (16 bit) BP base pointer (16 bit) IP instruction pointer (16 bit)

CS, Code Segment

Used to "point" to Instructions

Determines a Memory Address (along with IP)

Segmented Address written as CS:IP

DS, Data Segment

Used to "point" to Data

Determines Memory Address (along with other registers)

ES, Extra Segment allows 2 Data Address Registers

SS, Stack Segment

Used to "point" to Data in Stack Structure (LIFO)

Used with SP or BP

SS:SP or SS:BP are valid Segmented Addresses

IP, Instruction Pointer

Used to "point" to Instructions

Determines a Memory Address (along with CS)

Segmented Address written as CS:IP

SI, Source Index; DI, Destination Index

Used to “point” to Data

Determines Memory Address (along with other registers)

DS, ES commonly used

SP, Stack Pointer; BP, Base Pointer

Used to “point” to Data in Stack Structure (LIFO)

Used with SS

SS:SP or SS:BP are valid Segmented address

Memory address calculations

The 8086 uses a 20 bit address bus but the registers are only sixteen bit. To derive twenty bit addresses from the registers two registers are combined. Every memory reference uses one of the four segment registers plus an offset and/or a base pointer and/or a index register. The segment register is multiplied by sixteen (shifted to the left four bits) and added to the sixteen bit result of the offset calculation.

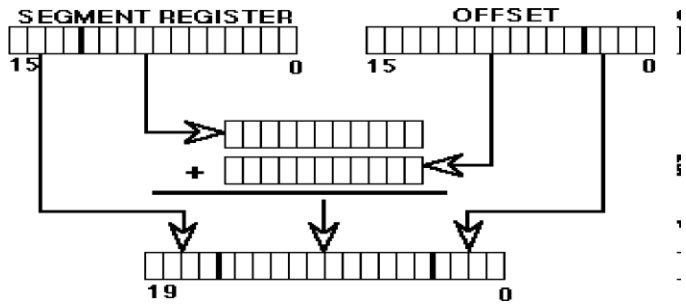


Figure 1Diagrammatic Representation of Address calculation

The 8086 provides four segment registers for address calculations. Each segment register is assigned a different task. The code segment register is always used with the instruction pointer (also called the program counter) to point to the instruction that is to be executed next. The stack segment register is always used with the stack pointer to point to the last value pushed onto the stack. The extra segment is general purpose segment register. The data segment register is the default register to calculate data operations, this can be over ridden by specifying the segment register. For example *mov ax,var1* would use the offset var1 and the data segment to calculate the memory reference but *mov ax,ss:var1* would use the offset var1 and the stack segment register to calculate the memory reference.

The offset can be calculated in a number of ways. Their are three elements that can make up an offset. The first element is a base register, this can be one of the BX of BP registers (the BP register

defaults to the stack segment). The second element is one of the index register, SI or DI. The third element is a displacement. A displacement can be a numerical value or an offset to a label. An offset can contain one to three of these elements, making a total of sixteen possibilities.

BX SI
or + or + Displacement
BP DI
(base) (index)

Memory Segmentation

x86 Memory Partitioned into Segments

– x386: maximum size is 4GB (32-bit index reg.)

– x386: can have 6 active segments (CS, SS, DS, ES, FS, GS)

Program for non-overlapped and overlapped block transfer of array elements.Takes the array elements from the user and also the number of elements to be overlapped in overlapped transfer.Block transfer here, refers to moving of block of data within the memory to a different location.In non-overlapped transfer we move the data to a completely new location. It is easily accomplished by copying the data using two pointers, one data byte at a time.In overlapped transfer the data block is shifted slightly from the present position,thus, some of the starting elements may overlap with the old position of the last elements in the array. They are therefore copied in reverse order.

Read 2 arrays (block) of 2 digit numbers. The numbers may be HEX or BCD. In nonoverlapped block transfer data block is transferred in empty memory space. In overlapped block transfer data block transferred in occupied memory space. New data will overwrite on old. Display both block.

ALGORITHM

Non-overlapped mode

1. Initialize 2 memory blocks pointed by source and destination registers.

2. Initialize counter.

3. Move the contents pointed by source register to a register.

4. Increment address of source register.

5. Move the contents from register into location pointed by destination register.

6. Increment destination registers.

7. Decrement counter.

8. Repeat from steps 3 to step 6 until counter is 0.

9. End.

TEST CASES:

Sr. NO	**Test ID**	**Steps**	**Input**	**Expected Result**	**Actual Result**	**Status (Pass/ Fail)**
1	ID1	Take hard coded data and display result for Non Overlapped memory block	Source block: 11 22 33 44 55	Destination block: 11 22 33 44 55	Destination block: 11 22 33 44 55	Pass
2	ID2	Take hard coded data and display result for Non Overlapped memory block	Source block: 11 22 33 44 55	Destination block: 11 22 33 44 55	22 11 33 44 55	Fail

CONCLUSION: Hence we have executed programs to perform non-overlapped block transfer.

FAQ's:

1. What is the use of Direction flag in Block transfer?
Ans:

2. What is use of Source and Destination Index in above program
Ans:

3. Which interrupt is used to terminate the program in 8086 ?
Ans:

4. Explain .data?
Ans.

5. Explain .bss?
Ans.

Program & Output

ASSIGNMENT NO: 8

TITLE: Write X86/64 Assembly language program (ALP) to perform overlapped block transfer.

PROBLEM STATEMENT: Write X86/64 ALP to perform overlapped block transfer with string specific instructions Block containing data can be defined in the data segment.

COURSE OBJECTIVES: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME : 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

2: **Apply** logic to **implement** code conversion.

THEORY-

Same as Non-overlapped(no need to write again)

ALGORITHM

Overlapped mode

1. Initialize 2 memory blocks pointed by source and destination registers.

2. Initialize counter.

3. Move the contents pointed by source register [si+count] to a variable.

4. Decrement address of source register.

5. Move the contents from variable into location pointed by destination register [di+count]

6. Decrement destination registers.

7. Decrement counter.

8. Repeat from steps 3 to step 6 until counter is 0.

9. End.

TEST CASES:

Sr. NO	Test ID	Steps	Input	Expected Result	Actual Result	Status (Pass/ Fail)
1	ID3	Take hard coded data and display result for Overlapped memory block	Source block: 11 22 33 44 55	Destination block: 11 22 11 22 33 44 55	Destination block: 11 22 11 22 33 44 55	Pass
2	ID4	Take hard coded data and display result for Overlapped memory block	Source block: 11 22 33 44 55	Destination block: 11 22 11 22 33 44 55	55 44 33 22 11 11 22 33	Fail

CONCLUSION: Hence we have executed programs to perform overlapped block transfer.

FAQ's:

1. What is difference between overlapped and non-overlapped block transfer
Ans:

2. Explain the logic of overlapped block transfer
Ans:

3. Which string instruction we can use for overlapped block transfer
Ans:

4. Explain write macro
Ans.

5. Explain print macro
Ans.

Program & Output

ASSIGNMENT NO: 9

TITLE: Write ALP to Perform multiplication

PROBLEM STATEMENT: Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. (use of 64-bit registers is expected).

COURSE OBJECTIVES: To understand assembly language programming instruction set
To understand different assembly directives with example

OUTCOME : 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

2: **Apply** logic to **implement** code conversion.

THEORY:

Multiplying unsigned numbers

Multiplying unsigned numbers in binary is quite easy. Recall that with 4 bit numbers we can represent numbers from 0 to 15. Multiplication can be performed done exactly as with decimal numbers, except that you have only two digits (0 and 1) The only number facts to remember are that 0*1=0, and 1*1=1 (this is the same as a logical "and").

Multiplication is different than addition in that multiplication of an n bit number by an m bit number results in an n+m bit number. Let's take a look at an example where n=m=4 and the result is 8 bits

Multiplying signed numbers

There are many methods to multiply 2's complement numbers. The easiest is to simply find the magnitude of the two multiplicands, multiply these together, and then use the original sign bits to determine the sign of the result. If the multiplicands had the same sign, the result must be positive, if they had different signs, the result is negative. Multiplication by zero is a special case (the result is always zero, with no sign bit).

Multiplication and division can be performed on signed or unsigned numbers. For unsigned numbers, MUL and DIV instructions are used, while for signed numbers IMUL and IDIV are used.

In Successive addition multiplicand is added in result multiplier number of times.

Successive Addition Method for 8-bit numbers

- Consider that a byte is present in the AL register and second byte is present in BL Register.
  - We have to multiply the in AL with the byte in BL
    - Multiply the number using Successive Addition Method.

- In this method, one number is accepted and other number is taken as a counter.
- The first number is added with itself, till the counter decrements to zero.
- Result is stored in DX register, Display the result, using display routine

In add & shift method multiplicand is added after shifting if multiplier bit is rotated through carry generator operation is perform on multiplicand then shift multiplier towards left by

1.**Sequential Shift/Add-Method**

Method to avoid adder arrays

• shift register for partial product and multiplier with each cycle,

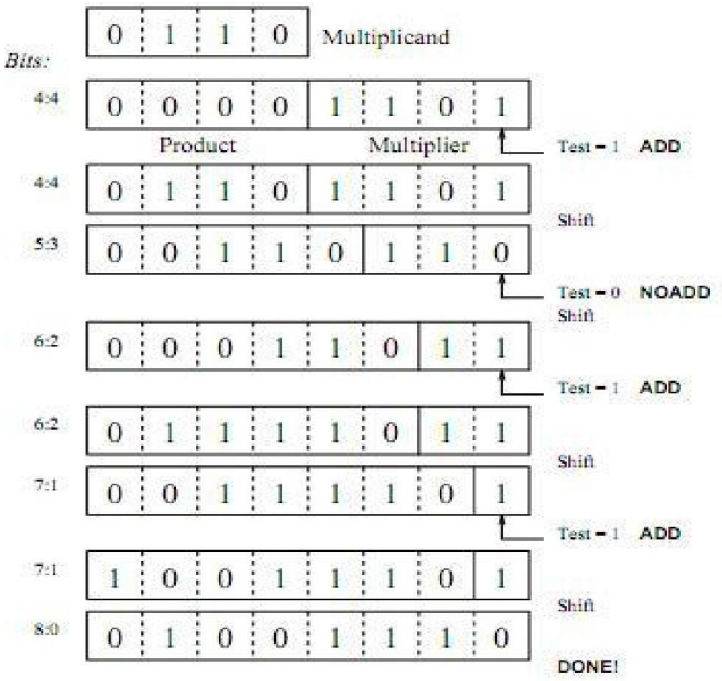
1. Partial product increases by one digit

2. Multiplier is reduced by one digit

• MSBs of partial product and multiplicand are aligned in each cycle

• not the multiplicand is shifted

⇒ Partial product and multiplier are



ALGORITHM :

1. Initialize data section.
2. Initialize .bss section.
3. Write macro, accept macro

4. Accept the number.
5. Convert into ASCII to HEX
6. Accept another number.
7. Initialize the counter.
8. Store the count in ARRAY.
9. For Successive addition
10. Assign sum=first number count=0
11. Add the first number with counter and store in register.
12. Repeat the loop with defragmentation of second number.
13. sum=sum+ Multiplicand
14. Decrement count.
15. print sum
16. ADD and Shift method
17. Assign count=no of digits in multiplier(edx=08h), sum=0,shiftvar=0,Count=0
18. Store the first number in register
19. Right carry on first number generate then go to step 20,not generate carry then go to step 24
20. Perform Shift left Multiplier by count value
21. If carry flag set, sum=sum+(Left_shifted multiplicand by shiftvar)
22. shiftvar=shiftvar+1
23. Add value in the temporary variable
24. Increment counter, decrements assign count (edx).
25. If assign =count 0 goto step 26 if not go to step 19.
26. Print sum
27. End

TEST CASES:

Sr. NO	**Test ID**	**Steps**	**Input**	**Expected Result**	**Actual Result**	**Status (Pass/ Fail)**
1	ID1	Take data from user and display result (multiplication) for Successive Addition	Operand 1 : 03 Operand 2 : 02	Multiplication : 06	Multiplication : 06	Pass

2	ID2	Take data from user and display result (multiplication) for Successive Addition	Operand 1 : 03 Operand 2 : 02	Multiplication : 06	Multiplication : 03	Fail
3	ID3	Take data from user and display result (multiplication) for Add & Shift method	Operand 1 : 03 Operand 2 : 02	Multiplication : 06	Multiplication : 06	Pass
4	ID4	Take data from user and display result (multiplication) for Add & Shift method	Operand 1 : 03 Operand 2 : 02	Multiplication : 06	Multiplication : 03	Fail

CONCLUSION: Hence we have executed programs to perform multiplication of two 8-bit hex numbers using successive addition and add-shift method.

FAQ's :

1. Write shift instructions with syntax.

Ans:

2. Write any two rotate instructions.

Ans:

3. Result stored in which register

Ans:

4. Which registers are used to store arguments?

Ans:

5. Explain the logic of multiplication using successive addition method

Ans.

Program & Output

ASSIGNMENT NO: 10

TITLE: Implement x86 program for File Operation

PROBLEM STATEMENT:

Write X86 Assembly Language Program (ALP) to implement following OS commands
i) COPY, ii) TYPE Using file operations. User is supposed to provide command line arguments

COURSE OBJECTIVES: To understand assembly language programming instruction set

To understand different assembly directives with example
To apply instruction set for implementing x86/64 bit assembly language programs

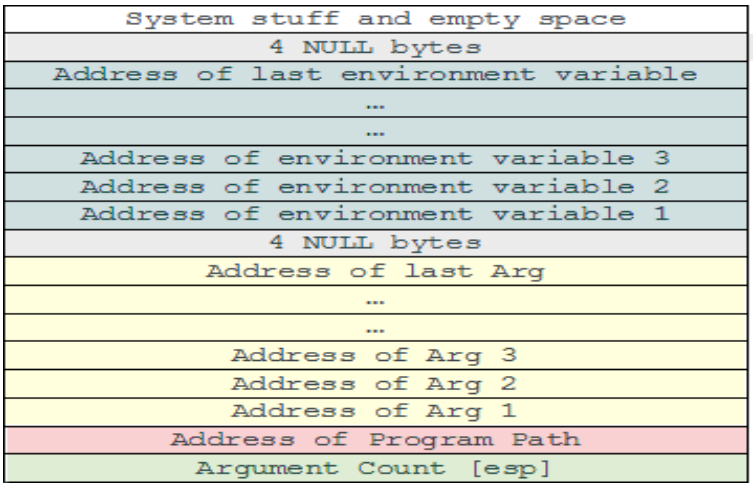
OUTCOMES Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

2: **Apply** logic to **implement** code conversion

3: **Analyze** and **apply** logic to **demonstrate** processor mode of operation

THEORY:

Stack When You Run Program



SYSTEM CALL

- **OPEN File**

```
mov rax, 2          ; 'open' syscall
mov rdi, fname1     ; file name
```

```
mov rsi, 0          ;
mov rdx, 0777       ; permissions set
Syscall
mov [fd_in], rax
```

- **OPEN File/Create file**

```
mov rax, 2          ; 'open' syscall
mov rdi, fname1  ; file name
mov rsi, 0102o      ; read and write mode,                    create if not
mov rdx, 0666o      ; permissions set
Syscall
mov [fd_in], rax
```

- **READ File**

```
mov rax, 0          ; ‘Read' syscall
mov rdi, [fd_in]    ; file Pointer
mov rsi, Buffer     ; Buffer for read
mov rdx, length     ; len of data want to read
Syscall
```

- **WRITE File**

```
mov rax, 01         ; ‘Write' syscall
mov rdi, [fd_in]    ; file Pointer
mov rsi, Buffer     ; Buffer for write
mov rdx, length     ; len of data want to read
Syscall
```

- **DELETE File**

```
mov rax,87
mov rdi,Fname
syscall
```

- **CLOSE File**

```
mov rax,3
mov rdi,[fd_in]
syscall
```

ALGORITHM:

COPY:

- Open file in read mode using open interrupt.
- Read contents of file using read interrupt.
- Create another file using read interrupt change only attributes.
- Open another file using open interrupt.
- Write contents of buffer into opened file.
- Close both files using close interrupt.

TYPE:

- Open file in read mode using open interrupt.
- Read contents of file using read interrupt.
- Display contents of file using write interrupt.
- Close file using close interrupt

DELETE:

- DELETE file using delete interrupt.

Definitions: Refer Ray Duncan

TEST CASES:

Sr. NO	Test ID	Steps	Input	Expected Result	Actual Result	Status (Pass/ Fail)
1	ID1	Enter program name with text file	Mlpro p.txt ml.txt	Correct file name correct extension Correct file name correct extension file written successfully	Correct file name correct extension Correct file name correct extension file written successfully	Pass
2	ID2	Enter program name with text file	Mlpro p.txt mlprog.txt	CCorrect file name correct extension Correct file name correct extension file written successfully	Correct file name correct extension Incorrect file name	Fail
3	ID3	Enter program name with text file	Mlpro p.txt	Correct file name correct extension file opened successfully Successfully reading of file: Contents	Correct file name correct extension file opened successfully Successfully reading of file: Contents	Pass
4	ID4	Enter program name with text file	Mlpro	Correct file name correct extension file opened successfully Successfully reading of file: Contents	Incorrect file name	Fail

CONCLUSION: All options stated above are tested and executed successfully.

FAQ's:

1. What is File Descriptor?

Ans:

2. What is File Pointer?

Ans

3. List String instruction

Ans:

4. What is a dangling pointer?

Ans:

5. Which function is used to open filename?

Ans:

Program & Output

ASSIGNMENT NO: 11

TITLE: Implement x86 program to find the factorial of a number.

PROBLEM STATEMENT: WWrite x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code.

COURSE OBJECTIVES: To understand assembly language programming instruction set

To understand different assembly directives with example

OUTCOME : 1. Understand and **apply** various addressing modes and instruction sets to implement assembly language programs.

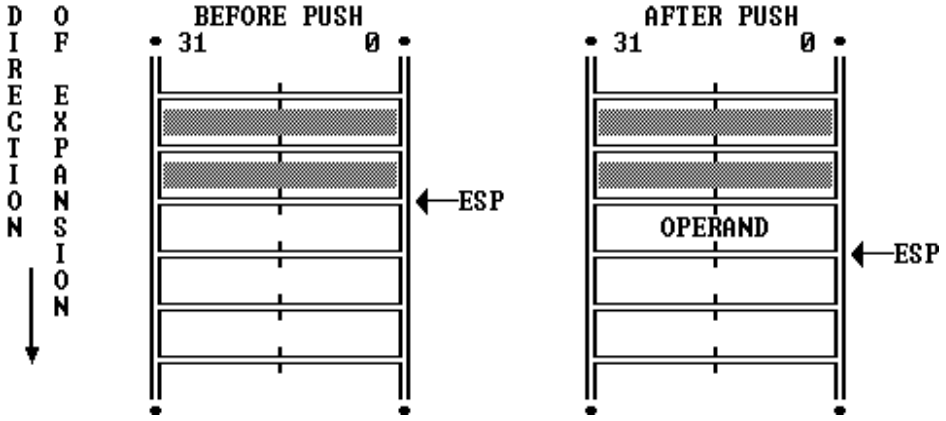
2: **Apply** logic to **implement** code conversion.

THEORY:

1. To compute the factorial of a number means to multiply a number n with (n-1) (n-2) …× 2 × 1.
   Example : to compute 5! = 5×4×3×2×1=120.
2. We will initialize AX=1 and load the number whose factorial is to be computed in BX. Call procedure fact, which will calculate the factorial of the number.

PUSH (Push) decrements the stack pointer (ESP), then transfers the source operand to the top of stack indicated by ESP (see Figure 3-1).

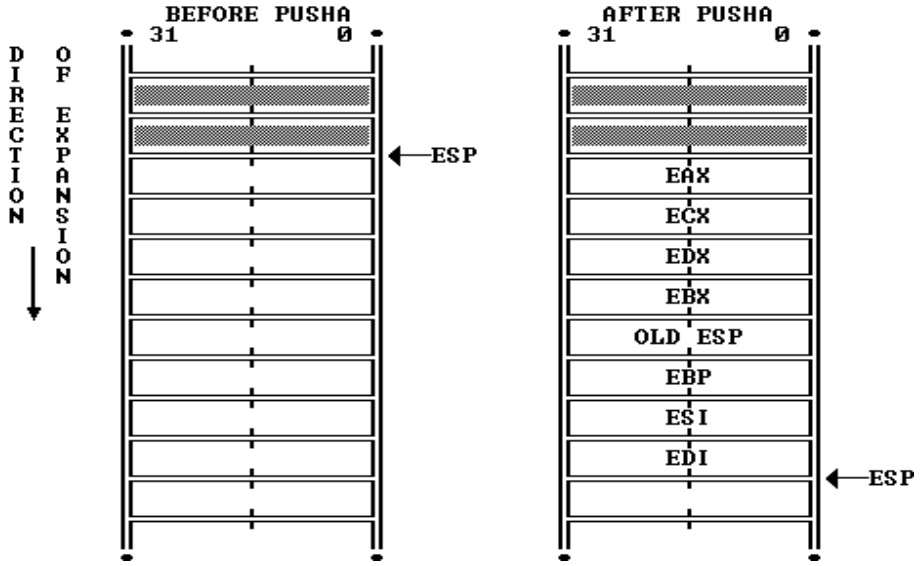
Figure 3-1. PUSH



PUSH is often used to place parameters on the stack before calling a procedure; it is also the basic means of storing temporary variables on the stack. The PUSH instruction operates on memory

operands, immediate operands, and register operands (including segment registers).

Figure 3-2. PUSHA



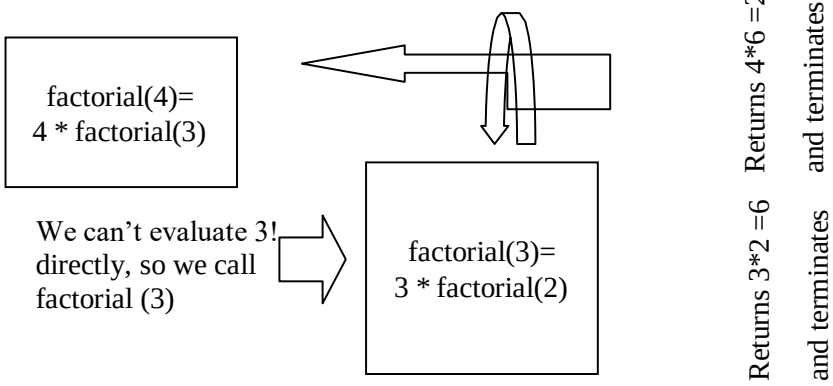
PUSHA (Push All Registers) saves the contents of the eight general registers on the stack. This instruction simplifies procedure calls by reducing the number of instructions required to retain the contents of the general registers for use in a procedure. The processor pushes the general registers

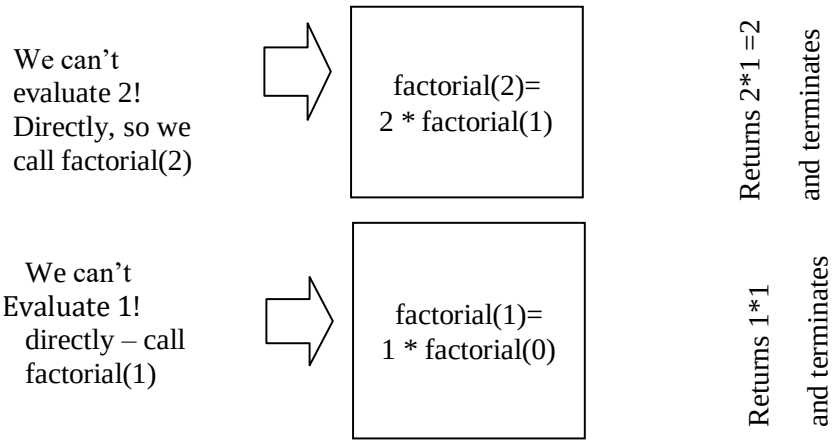
on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI.PUSHA is complemented by the POPA instruction.

POP (Pop) transfers the word or doubleword at the current top of stack (indicated by ESP) to the destination operand, and then increments ESP to point to the new top of stack.

POPA (Pop All Registers) restores the registers saved on the stack by PUSHA, except that it ignores the saved value of ESP.

Trace of a call to Factorial: int z = factorial(4)





ALGORITHM:

4. Initialize the Data Segment.
5. Initialize AX = 1.
6. Load the number in BX.
7. Call procedure fact.
8. Compare BX with 1, if not go to step 7.
9. AX = 1 and return back to the calling program.
10. AX = AX × BX.
11. Decrement BX.
12. Compare BX with 1, if not go to step 7.
13. Return back to calling program.
14. Stop.

TEST CASES:

Sr. NO	**Test ID**	**Steps**	**Input**	**Expected Result**	**Actual Result**	**Status (Pass/ Fail)**
1	ID1	Enter integer number	4	24	24	Pass
2	ID2	Enter integer number	4	24	4	Fail

CONCLUSION: In this way, we have implemented factorial of a given number.

FAQs:

1. What is stack?
Ans.

2. Explain the use and operation of stack and stack pointer?
Ans:

3. Explain RESD, RESW, RESB, RESQ ?
Ans:
4. Explain Stack data structure
Ans:
5. How to calculate factorial
Ans:

Program & Output