

Microprocessor lab

Laboratory Manual

Table of Contents

Sr . No.	Topic	Page. No.
1.	Course Objective	8
2.	Program Outcomes	9
3.	Experiment Learning Outcome (ELO)	10
Sessions		
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.	12
2.	Write an X86/64 ALP to accept a string and to display its length.	17
3.	Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.	21
4.	Write a switch case driven X86/64 ALP to perform 64 -bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation.	25
5.	Write X86/64 ALP to count number of positive and negative numbers from the array	29
6.	Write 64 bit 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. (use of 64-bit registers is expected)	39
7.	Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers	49
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.	56
9.	Write X86/64 ALP to perform overlapped block transfer with string specific instructions. Block containing data can be defined in the data segment.	56
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)	68

11.	Write X86 menu driven Assembly Language Program (ALP) to implement OS (DOS) commands TYPE, COPY and DELETE using file operations. User is supposed to provide command line arguments in all cases	81
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.	87
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.	99

How To Use This Manual

This Manual assumes that the facilitators are aware of Collaborative Learning Methodologies.

This Manual will only provide them tool they may need to facilitate the session on Computer Organization module in collaborative learning environment.

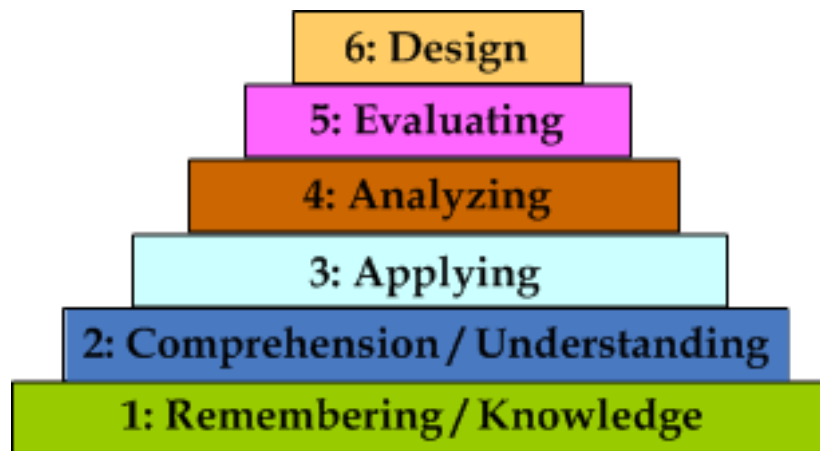
The Facilitator is expected to refer this Manual before the session.

K Applying Knowledge (PO:a)	A Problem Analysis (PO:b)	D Design & Development (PO:c)	I Investigation of problems (PO:d)
M Modern Tool Usage (PO:e)	E Engineer & Society (PO:f)	E Environment Sustainabilit y (PO:h)	T Ethics (PO:i)
T Individual & Team work (PO:g)	O Communication (PO:k)	M Project Management & Finance(PO:j)	I Life Long Learning(PO:l)
















Disk Approach- Digital Blooms Taxonomy



- 1: Remembering / Knowledge
- 2: Comprehension / Understanding
- 3: Applying
- 4: Analyzing
- 5: Evaluating
- 6: Creating / Design



This Manual uses icons as visual cues to the interactivities during the session.

Icons	Graduate Attributes
	Applying Knowledge
	Problem Analysis
	Design and Development
	Investigation of Problem
	Modern Tool Usage
	Engineer and Society
	Environment Sustainability
	Ethics
	Individual and Teamwork
	Communication
	Project Management and Finance
	Lifelong Learning
	Blooms Taxonomy
	Remembering
	Understanding
	Applying
	Analyzing
	Evaluating
	Creating

Course Outcomes:

CO1: Understand and compare Architecture of advanced processors and its resources.

CO2: Apply assembly language programming to develop real life applications.

CO3: Implement parallel processing and math Co-processor.

CO4: Compare different processor configurations.


Program Outcomes: - Students are expected to know and be able –


1.	To apply knowledge of mathematics, science, engineering fundamentals, problem solving skills, algorithmic analysis and mathematical modelling to the solution of complex engineering problems.
2.	To analyze the problem by finding its domain and applying domain specific skills.
3.	To understand the design issues of the product/software and develop effective solutions with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4.	To find solutions of complex problems by conducting investigations applying suitable techniques.
5.	To adapt the usage of modern tools and recent software.
6.	To contribute towards the society by understanding the impact of Engineering on global aspect.
7.	To understand environment issues and design a sustainable system.
8.	To understand and follow professional ethics.
9.	To function effectively as an individual and as member or leader in diverse teams and interdisciplinary settings.
10.	To demonstrate effective communication at various levels.
11.	To apply the knowledge of Computer Engineering for development of projects, finance and management.
12.	To keep in touch with current technologies and inculcate the practice of lifelong learning.


Experiment Learning Outcome: -


ELO1: Apply logical instruction sets to segregate positive and negative numbers. 

ELO2: Apply the concept of overlapped and non-overlapped block transfer in the program with or without using string instruction. 

ELO3: Convert hexadecimal number to BCD and vice versa with the help of assembly language programming. 


ELO4: Choose different multiplication methods such as add & shift and successive addition for multiplication of two numbers without using MUL instruction. 

ELO5: Analyze the difference between near and far procedure to find number of lines, blank spaces & occurrence of character using nasm. 


ELO6: Apply the concept of real mode and protected mode in 8086 ALP to implement a program to display values from GDTR, LDTR, IDTR, TR and MSW registers. 


ELO7: Apply the bubble sort technique in 8086 to sort the input from text file. 

ELO8: Implement DOS commands like TYPE, COPY, DELET using file operations. 

ELO9: Apply the concept of recursion to find factorial of a number. 

ELO10: Implement ALP to Calculate the roots of the quadratic equation with the help of assembly language programming. 

ELO11: Implement ALP to plot Sine Wave, Cosine Wave & Sinc function. 

ELO12: Evaluate mean, variance and standard deviation using 8087 math coprocessor's instruction set. 

ELO13: Create Real Time Application using TSR. 

ELO to CO Mapping

	CO: 1	CO: 2	CO: 3	CO: 4
ELO: 1	2	2	-	-
ELO: 2	2	2	3	-
ELO: 3	2	2	-	-
ELO: 4	1	3	-	-
ELO: 5	1	2	-	-
ELO: 6	-	2	-	-
ELO: 7	-	2	-	-
ELO: 8	-	2	-	-
ELO: 9	-	2	3	-
ELO: 10	-	2	3	-
ELO: 11	-	3	3	-
ELO: 12	-	2	3	-
ELO: 13	-	3	-	-

EXPERIMENT NO.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.

TITLE: 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.

OBJECTIVES:

1. To be familiar with the format of assembly language program structure and instructions.
2. To study the format of assembly language program along with different assembler directives and different functions of the NASM.
3. To learn the procedure how to store N hexadecimal number in memory.

PROBLEM DEFINITION:

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.

WORKING ENVIRONMENT:

- 1) CPU: Intel I5 Processor
- 2) OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
- 3) Editor: gedit, GNU Editor
- 4) Assembler: NASM (Netwide Assembler)
- 5) Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

1. CPU: Intel I5 Processor
2. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
3. Editor: gedit, GNU Editor
4. Assembler: NASM (Netwide Assembler)

INPUT: Five hexadecimal numbers

OUTPUT: Five hexadecimal numbers

THEORY

Assembly language Program is mnemonic representation of machine code. Three assemblers available for assembling the programs for IBM-PC are:

1. Microsoft Micro Assembler(MASM)
2. Borland Turbo Assembler(TASM)
3. Net wide Assembler(NASM)

Assembly Basic Syntax

An assembly program can be divided into three sections:

1. The **data** section
2. The **bss** section
3. The **text** section

The data Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names or buffer size etc. in this section.

The syntax for declaring data section is:

section.data

The bss Section

The bss section is used for declaring uninitialized data or variables. The syntax for declaring bss section is:

section .bss

The text section

The text section is used for writing the actual code. This section must begin with the declaration `global _start`, which tells the kernel where the program execution begins.

The syntax for declaring text section is:

section .text

global _start

_start:

Assembly Language Statements

Assembly language programs consist of three types of statements:

1. Executable instructions or instructions
2. Assembler directives or pseudo-ops
3. Macros

The **executable instructions** or simply **instructions** tell the processor what to do. Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.

The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process.

These are non-executable and do not generate machine language instructions.

Macros are basically a text substitution mechanism.

Assembly System Calls

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

Linux System Calls (32 bit)

You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:

- Put the system call number in the EAX register.
- Store the arguments to the system call in the registers EBX, ECX, etc.
- Call the relevant interrupt (80h)
- The result is usually returned in the EAX register

There are six registers that store the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments then the memory location of the first argument is stored in the EBX register.

Instructions needed:

EQU- Assign single absolute values to symbols.

MOV- Copies byte or word from specified source to specified destination

CALL- The CALL instruction causes the procedure named in the operand to be executed.

JNZ- Jumps if not equal to Zero

JNC-Jumps if no carry is generated

ALGORITHM:

- 1 Start
2. Declare & initialize the variables in .data section.
3. Declare uninitialized variables in .bss section.
4. Declare Macros for print and exit operation.
5. Initialize pointer with source address of array.
6. Initialize count for number of elements.
7. Put the complete summed rbx value to arr[n]
8. Max display of 16 characters and rsi points to _output[16]
9. Dividing by base 16
10. Terminate the process.
11. Declare the Procedure.
12. Stop.

CONCLUSION

Here we have accepted 64 bit Hexadecimal numbers from user and store them in an array and display the accepted numbers

EXPERIMENT NO.2

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

TITLE: Accept a string and to display its length.

OBJECTIVES:

1. To learn the instructions related to String
2. To be familiar with data segments.
3. To learn the instructions related to String operation

PROBLEM DEFINITION:

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

WORKING ENVIRONMENT:

1. CPU: Intel I5 Processor
2. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
3. Editor: gedit, GNU Editor
4. Assembler: NASM (Netwide Assembler)
5. Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

1. CPU: Intel I5 Processor
2. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
3. Editor: gedit, GNU Editor
4. Assembler: NASM (Netwide Assembler)
5. Linker:-LD, GNU Linker

INPUT: String data.

OUTPUT: Display input string and calculate its length

THEORY:

A few data-manipulation instructions implicitly use specialized addressing methods:

- For a few short forms of MOV that implicitly use the EAX register, the offset of the operand is coded as a doubleword in the instruction. No base register, index register, or scaling factors are used.
- String operations implicitly address memory via DS:ESI, (MOVS, CMPS, OUTS, LODS, SCAS) or via ES:EDI (MOVS, CMPS, INS, STOS).

- Stack operations implicitly address operands via SS:ESP registers; e.g., PUSH, POP, PUSHA, PUSHAD, POPA, POPAD, PUSHF, PUSHFD, POPF, POPFD, CALL, RET, IRET, IRETD, exceptions, and interrupts.

The instructions in this category operate on strings rather than on logical or numeric values. Refer also to the section on I/O for information about the string I/O instructions (also known as block I/O).

The primitive string operations operate on one element of a string. A string element may be a byte, a word, or a doubleword. The string elements are addressed by the registers ESI and EDI. After every primitive operation ESI and/or EDI are automatically updated to point to the next element of the string. If the direction flag is zero, the index registers are incremented; if one, they are decremented. The amount of the increment or decrement is 1, 2, or 4 depending on the size of the string element.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

The precise action for each iteration is as follows:

1. If the address-size attribute is 16 bits, use CX for the count register; if the address-size attribute is 32 bits, use ECX for the count register.
2. Check CX. If it is zero, exit the iteration, and move to the next instruction.
3. Acknowledge any pending interrupts.
4. Perform the string operation once.
5. Decrement CX or ECX by one; no flags are modified.
6. Check the zero flag if the string operation is SCAS or CMPS. If the repeat condition does not hold, exit the iteration and move to the next instruction. Exit the iteration if the prefix is REPE and ZF is 0 (the last comparison was not equal), or if the prefix is REPNE and ZF is one (the last comparison was equal).
7. Return to step 1 for the next iteration.

Instructions needed:

MOVS: Move String

CMPS: Compare string

SCAS: Scan string

LODS: Load string

STOS: Store string

ESI: Source index register

EDI: Destination index register

REP: Repeat while ECX not zero

REPE/REPZ: Repeat while equal or zero

REPNE/REPNZ: Repeat while not equal or not zero

ALGORITHM:

- 1 Start
2. Declare & initialize the variables in .data section.
3. Declare uninitialized variables in .bss section.
4. Declare Macros for print and exit operation.
5. Initialize pointer to get input string from user.
6. Initialize counter for calculating no. of chatters in string.
7. Display string entered by user.
8. Put value of count in rax register
9. max size of display , for convinience set to 16 and rsipoints to output[16]
10. setting rdx to null without setting a null byte (a tip i saw on reddit) needed to clean dl for use
11. Declare the Procedure for ascii conversion.
12. Stop.

CONCLUSION

Here we have studied how to accept a string and to display its length.

EXPERIMENT NO.3

**Write an X86/64 ALP to find the largest of given
Byte/Word/Dword/64-bit numbers.**

TITLE: Find the largest of given Byte/Word/Dword/64-bit numbers.

OBJECTIVES:

1. To understanding of basic data structure
2. To compare the different data structure
3. To make usage of different data structure for hexadecimal number

PROBLEM DEFINITION:

Write an X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.

WORKING ENVIRONMENT:

6. CPU: Intel I5 Processor
7. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
8. Editor: gedit, GNU Editor
9. Assembler: NASM (Netwide Assembler)
10. Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

6. CPU: Intel I5 Processor
7. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
8. Editor: gedit, GNU Editor
9. Assembler: NASM (Netwide Assembler)
10. Linker:-LD, GNU Linker

INPUT: No of hexadecimal numbers

OUTPUT: Largest number form entered numbers

THEORY:

Data Types

Integer:

A signed binary numeric value contained in a 32-bit doubleword,16-bit word, or 8-bit byte. All operations assume a 2's complement representation. The sign bit is located in bit 7 in a byte, bit 15 in a word, and bit 31 in a doubleword. The sign bit has the value

zero for positive integers and one for negative. Since the high-order bit is used for a sign, the range of an 8-bit integer is -128 through +127; 16-bit integers may range from -32,768 through +32,767; 32-bit integers may range from -231 through +231-1. The value zero has a positive sign.

Ordinal:

An unsigned binary numeric value contained in a 32-bit doubleword, 16-bit word, or 8-bit byte. All bits are considered in determining magnitude of the number. The value range of an 8-bit ordinal number is 0-255; 16 bits can represent values from 0 through 65,535; 32 bits can represent values from 0 through 232-1.

Near Pointer:

A 32-bit logical address. A near pointer is an offset within a segment. Near pointers are used in either a flat or a segmented model of memory organization.

Far Pointer:

A 48-bit logical address of two components: a 16-bit segment selector component and a 32-bit offset component. Far pointers are used by applications programmers only when systems designers choose a segmented memory organization.

String:

A contiguous sequence of bytes, words, or doublewords. A string may contain from zero bytes to 232-1 bytes (4 gigabytes).

Bit field:

A contiguous sequence of bits. A bit field may begin at any bit position of any byte and may contain up to 32 bits.

Bit string:

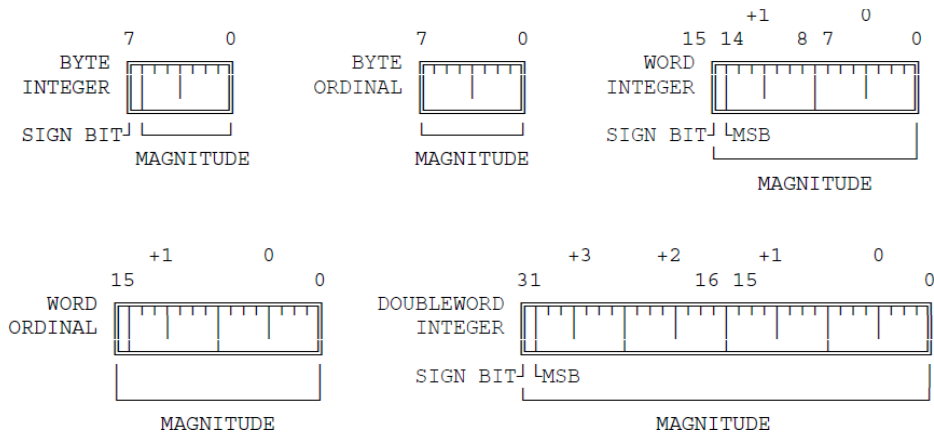
A contiguous sequence of bits. A bit string may begin at any bit position of any byte and may contain up to 232-1 bits.

BCD:

A byte (unpacked) representation of a decimal digit in the range 0 through 9. Unpacked decimal numbers are stored as unsigned byte quantities. One digit is stored in each byte. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers.

Packed BCD:

A byte (packed) representation of two decimal digits, each in the range 0 through 9. One digit is stored in each half-byte. The digit in the high-order half-byte is the most significant. Values 0-9 are valid in each half-byte. The range of a packed decimal byte is 0-99.



Instructions needed:

ROL: Rotate 32 bits r/m dword left CL times

JBE: Jump short if below or equal

INC: Increment dword register by 1

DEC: Decrement dword register by 1

ALGORITHM:

- 1 Start
2. Declare & initialize the variables in .data section.
3. Declare uninitialized variables in .bss section.
4. Declare Macros for print and exit operation.
5. Initialize pointer with source address of array.
6. Initialize count to find the data type of number.
7. Displaying array elements
8. Finding Largest Number by comparing rsi, rax
9. Displaying Largest Number
10. Terminate the process.
11. Declare the Procedure for ascii comparison .
12. Stop.

CONCLUSION:

Thus we have found the largest of given Byte/Word/Dword/64-bit numbers.

EXPERIMENT NO.4

Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation.

TITLE

Switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros.

OBJECTIVES:

1. To understand the looping statements
2. To understand switch case
3. To analyze the different arithmetic operations

PROBLEM DEFINITION:

Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation.

WORKING ENVIRONMENT:

11. CPU: Intel I5 Processor
12. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
13. Editor: gedit, GNU Editor
14. Assembler: NASM (Netwide Assembler)
15. Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

11. CPU: Intel I5 Processor
12. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
13. Editor: gedit, GNU Editor
14. Assembler: NASM (Netwide Assembler)
15. Linker:-LD, GNU Linker

INPUT: Two hexadecimal numbers

OUTPUT: Arithmetic operation performed on two entered numbers

THEORY:

Conditional Transfer Instructions

The conditional transfer instructions are jumps that may or may not transfer control, depending on the state of the CPU flags when the instruction executes.

- **Conditional Jump Instructions**

The conditional jumps that are listed as pairs are actually the same instruction. The assembler provides the alternate mnemonics for greater clarity within a program listing. Conditional jump instructions contain a displacement which is added to the EIP register if the condition is true. The displacement may be a byte, a word, or a doubleword. The displacement is signed; therefore, it can be used to jump forward or backward.

Unsigned Conditional Transfers

Mnemonic	Condition Tested	"Jump If..."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even

Signed Conditional Transfers

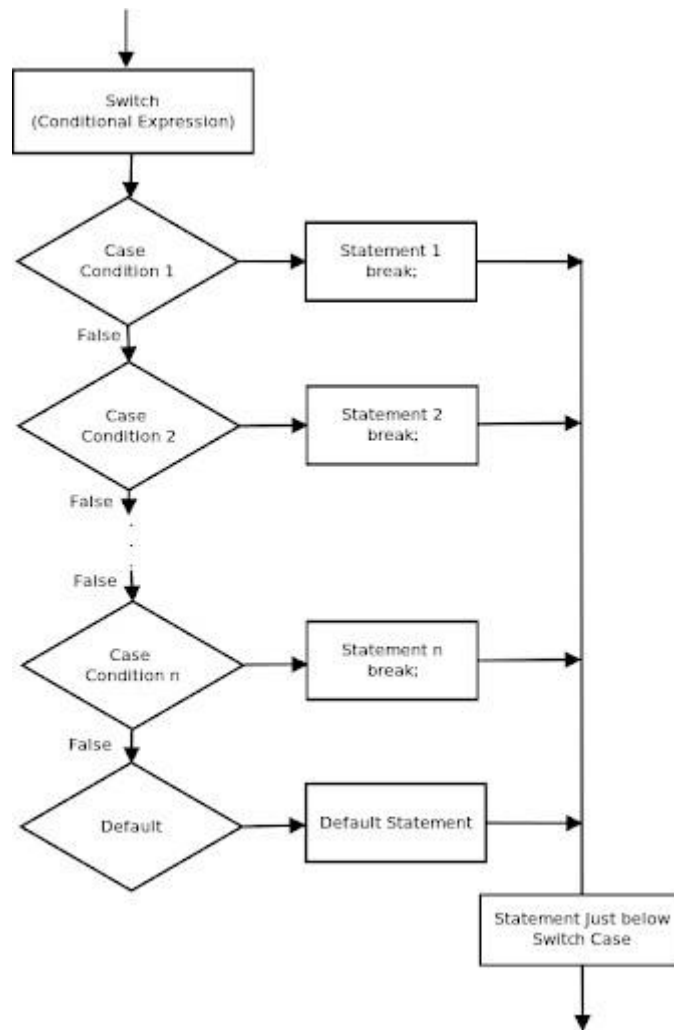
Mnemonic	Condition Tested	"Jump If..."
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (positive, including 0)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

- **Loop Instructions**

The loop instructions are conditional jumps that use a value placed in ECX to specify the number of repetitions of a software loop. All loop instructions automatically decrement ECX and terminate the loop when ECX=0. Four of the five loop instructions specify a condition involving ZF that terminates the loop before ECX reaches zero. LOOP (Loop While ECX Not Zero) is a conditional transfer that automatically decrements the ECX register before testing ECX for the branch condition. If ECX is non-

zero, the program branches to the target label specified in the instruction. The LOOP instruction causes the repetition of a code section until the operation of the LOOP instruction decrements ECX to a value of zero. If LOOP finds ECX=0, control transfers to the instruction immediately following the LOOP instruction. If the value of ECX is initially zero, then the LOOP executes 232 times.

FLOWCHART:



CONCLUSION:

Thus we have studied a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations.

EXPERIMENT NO.5

Count numbers of positive and negative numbers from the array.

Session Plan

Time (min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
10	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
20	Concept of Negative and Positive numbers	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension

TITLE: Write an ALP to count numbers of positive and negative numbers from the array.

OBJECTIVES:

4. To be familiar with the format of assembly language program structure and instructions.
5. To study the format of assembly language program along with different assembler directives and different functions of the NASM.
6. To learn the procedure how to store N hexadecimal number in memory.

PROBLEM DEFINITION:

Write X86/64 Assembly language program (ALP) to count number of positive and negative numbers from array.

WORKING ENVIRONMENT:

- 6) CPU: Intel I5 Processor
- 7) OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
- 8) Editor: gedit, GNU Editor
- 9) Assembler: NASM (Netwide Assembler)
- 10) Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

5. CPU: Intel I5 Processor
6. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
7. Editor: gedit, GNU Editor
8. Assembler: NASM (Netwide Assembler)

INPUT:Hexadecimal numbers

OUTPUT:Number of Negative numbers, Number of Positive numbers

The following code snippet shows the use of the system call sys_exit:

```
MOV EAX, 1      ; system call number (sys_exit)
INT 0x80        ; call kernel
```

The following code snippet shows the use of the system call sys_write:

```
MOV EAX, 4; system call number (sys_write)
MOV EBX, 1; file descriptor (stdout)
MOV ECX, MSG ; message to write
MOV EDX, 4; message length
INT 0x80; call kernel
```

Linux System Calls (64 bit)

Sys_write:

```
MOV RAX, 1
MOV RDI, 1
MOV RSI, message
MOV RDX, message_length
SYSCALL
```

Sys_read:

```
MOV RAX, 0
MOV RDI, 0
MOV RSI, array_name
MOV RDX, array_size
SYSCALL
```

Sys_exit:

```
MOV RAX, 60
MOV RDI, 0
SYSCALL
```

Assembly Variables

NASM provides various **define directives** for reserving storage space for variables. The **define** Assembler directive is used for allocation of storage space. It can be used to reserve as well as initialize one or more bytes.

Allocating Storage Space for Initialized Data

There are five basic forms of the define directive:

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Allocating Storage Space for Uninitialized Data

The reserve directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved. Each define directive has a related reserve directive.

There are five basic forms of the reserve directive:

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Instructions needed:

1. **MOV**-Copies byte or word from specified source to specified destination
2. **ROR**-Rotates bits of byte or word right, LSB to MSB and to CF
3. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word

4. **INC**-Increments specified byte/word by 1
5. **DEC**-Decrements specified byte/word by 1
6. **JNZ**-Jumps if not equal to Zero
7. **JNC**-Jumps if no carry is generated
8. **CMP**-Compares to specified bytes or words
9. **JBE**-Jumps if below or equal
10. **ADD**-Adds specified byte to byte or word to word
11. **CALL**-Transfers the control from calling program to procedure.
12. **RET**-Return from where call is made

MATHEMATICAL MODEL:

Let $S = \{ s, e, X, Y, F_{me}, mem \mid \Phi_s \}$ be the programmer's perspective of Array Addition
Where

S = System

s = Distinct Start of System

e = Distinct End Of System

X = Set of Inputs

Y = Set Of outputs

F_{me} = Central Function

Mem = Memory Required

Φ_s = Constraints

Let X be the input such that

$X = \{ X_1, X_2, X_3, \dots \}$

Such that there exists function $f_{X_1}: X_1 \longrightarrow \{0,1\}$

X2 Source Array

Let $X2 = \{\{b7\text{----}b0\}\{b7\text{-----}b0\}\text{-----}\{b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0\}\}$ where $\forall\ bi \in X1$

There exists a function $f_{X2}: X2 \longrightarrow \{\{00h\text{----}FFh\}\{00h\text{----}FFh\}\{00h\text{----}FFh\}\text{-----}\}$

X3 is the two digit count value

Let $X3 = b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0$ where $\forall\ bi \in X1$

There exists a function $f_{X3}: X3 \longrightarrow \{01h, 02h, \text{--}, \text{--} FFh\}$

Let Y is the Output

Let $Y = b15\ b14\ b13\ b12\ b11\ b10\ b9\ b8\ b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0$

Where $\forall bi \in X1$

$Y \longrightarrow \{0000h, 0001h, \text{-----} FFFFh\}$

Let $Fme = \{F1, F2, F3\}$

Where $F1 = \text{Accept}$

$F2 = \text{Count}$

$F3 = \text{Display}$

$F1 \longrightarrow 1) \text{ Accept the number stored in array.}$

$F2 \longrightarrow Y = \sum (\forall X2)$

$F3 \longrightarrow \text{Display Output}$

ALGORITHM:

1 Start

2. Declare & initialize the variables in .data section.

3. Declare uninitialized variables in .bss section.

4. Declare Macros for print and exit operation.

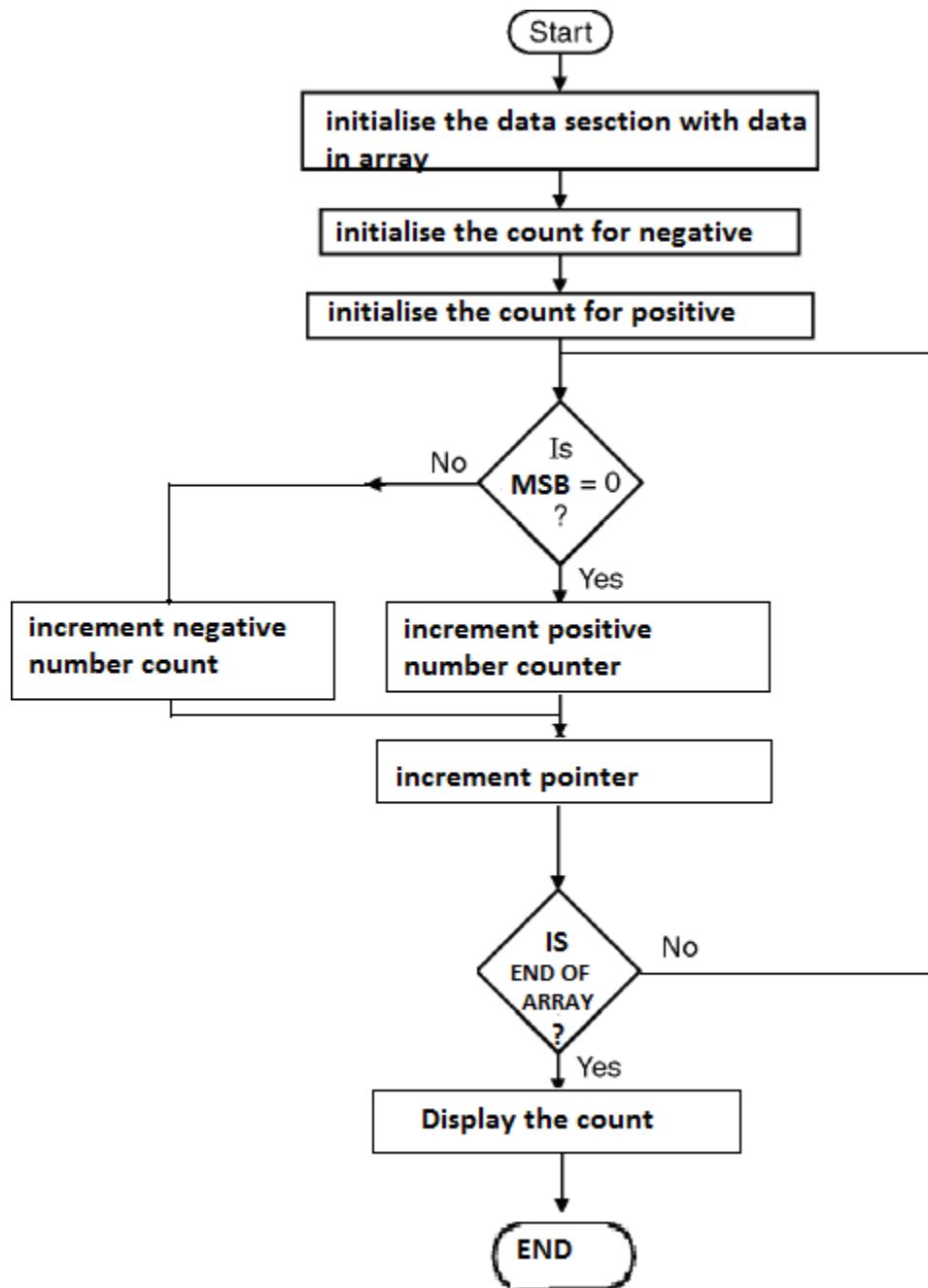
5. Initialize pointer with source address of array.

6. Initialize count for number of elements.

7. Set RBX as Counter register for storing positive numbers count.

8. Set RCX as Counter register for storing negative numbers count.
9. Get the number in RAX register.
10. Rotate the contents of RAX register left 1 bit to check sign bit.
11. Check if MSB is 1. If yes, goto step 12, else goto step 13.
12. Increment count for counting negative numbers.
13. Increment count for counting positive numbers.
14. Increment pointer.
15. Decrement count
16. Check for count = 0. If yes, goto step 17 else goto step 9.
17. Store the positive numbers count and negative numbers count in buffer.
18. Display first message. Call the procedure to display the positive count.
19. Display second message. Call the procedure to display the negative count.
20. Add newline.
21. Terminate the process.
22. Declare the Procedure.
23. Stop.

Flowchart:



CONCLUSION

Here we count the 32-bit numbers of positive and negative numbers in the array in the assembly language.

Output:

```
:[root@comppl2022 ~]# nasm -f elf64 Exp9.asm
:[root@comppl2022 ~]# ld -o Exp9 Exp9.o
:[root@comppl2022 ~]# ./Exp9
;Welcome to count +ve and -ve numbers in an array
;Count of +ve numbers::04
;Count of -ve numbers::03
:[root@comppl2022 ~]#
```

OUTCOME

Upon completion Students will be able to:

ELO1: Apply logical instruction sets to segregate positive and negative numbers. 

FAQ:

Oral Question Bank

- 1) What is logic of program.
- 2) What is meaning of BT instruction.
- 3) What is ascii for newline.
- 4) What are the different sections present in programme.
- 5) Write other instructions used to check positive and negative numbers.
- 6) Explain BT,BTC,BTR,BTS instruction with example
- 7) Write equivalent instruction of ROL BX,1

EXPERIMENT NO.6

Hex to BCD & BCD to Hex Conversion

Session Plan

Time (min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
10	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
20	Concept of Hex to BCD & BCD to Hex Conversion	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension

TITLE: Hex to BCD & BCD to Hex Conversion.

OBJECTIVES:

To Learn the implementation of ALP for conversion of Hex to BCD & vice a versa.

PROBLEM STATEMENT:

To write 64 bit 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.

SOFTWARE REQUIRED:

9. CPU: Intel I5 Processor
10. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
11. Editor: gedit, GNU Editor
12. Assembler: NASM (Netwide Assembler)
13. Linker:-LD, GNU Linker

INPUT:

1. Hexadecimal number
2. BCD Number

OUTPUT:

1. Conversion of hex to BCD number
2. Coersion of BCD to hex number

MATHEMATICAL MODEL:

Let $S = \{ s, e, X, Y, Fme, mem \mid \Phi s \}$ be the programmer's perspective of Hex to BCD & BCD to hex conversion .

Let X be the input such that

$X = \{ X_1, X_2, X_3, \dots \}$

Such that there exists function $f_{X_1}: X_1 \longrightarrow \{0,1\}$

X2 is the four Digit Hex Number.

Let $X_2 = b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ where

$\forall b_i \in X_1$. There exists a function $f_{X_2}: X_2 \longrightarrow \{ 0000h, 0001h, 0002h, \dots, FFFFh \}$

X3 is the Five digit BCD number.

Let $X_3 = b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

where $\forall b_i \in X_1$

There exists a function $f_{X_3}: X_3 \longrightarrow \{ 00000, 00001, \dots, 99999 \}$

X4 is the Single digit choice.

Let $X_4 = b_3 b_2 b_1 b_0$ where $\forall b_i \in X_1$

There exists a function $f_{X_4}: X_4 \longrightarrow \{ 1, 2, 3 \}$

Let Y1 is the 5 Digit BCD Output

Let $Y = b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Where $\forall b_i \in X_1$

$Y_1 \longrightarrow \{ 00000, 00001, \dots, 99999 \}$

Let Y2 is the 4 Digit Hex Output

Let $Y = b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Where $\forall b_i \in X_1$

$Y_2 \longrightarrow \{ 0000h, 0001h, \dots, FFFFh \}$

Fme 1(Hex to bcd) = { F1 ,F2,F3 }

Where F1 = Accept X2

F2 = Repeat (X2/10) till quotient =0 & push Remainder on stack

F3 = Pop remainder from stack & display

Fme 2(bcd to hex) = { F1 ,F2,F3,F4 }

Where F1 = Accept BCD digit X_{3i}

F2 = Result= Result*10 + X_{3i} & $i=i-1$ (Initially Result=0)

F3= repeat F1 onwards untill $i=0$

F4= Display Result

Fme 3(exit)= {F1}

Where F1= Call Sys_exit Call

Fme = { F1,F2,F3,F4}

Where F1= Accept X4

F2= If X4=1, Call Fme1

F3= If X4=2, Call Fme2

F4= If X4=3, Call Fme3

THEORY:

1. Hexadecimal to BCD conversion:

Conversion of a hexadecimal number can be carried out in different ways e.g. dividing number by 000Ah and displaying quotient in reverse way.

2. BCD to Hexadecimal number:

Conversion of BCD number to Hexadecimal number can be carried out by multiplying the BCD digit by its position value and then adding it in the final result.

Special instructions used:

DIV: Unsigned Divide. Result → Quotient in AL and Remainder in AH for 8-bit division and for 16-bit division Quotient in AX and Remainder in DX

MUL: Unsigned Multiply. For 8-bit operand multiplication result will be stored in AX and for 16-bit multiplication result is stored in DX:AX

Commands

- To assemble

nasm -f elf 64 hello.nasm -o hello.o

- To link

ld -o hello hello.o

- To execute -

./hello

ALGORITHM:

1. Start
2. Initialize data section
- 3 Display the Menu Message.
- 4 Accept the choice from the user.
- 5 If choice=1 then call Hex to bcd procedure. If choice=2 then call Bcd to Hex Procedure If choice=3 then call exit procedure

6. Hex to BCD Procedure:

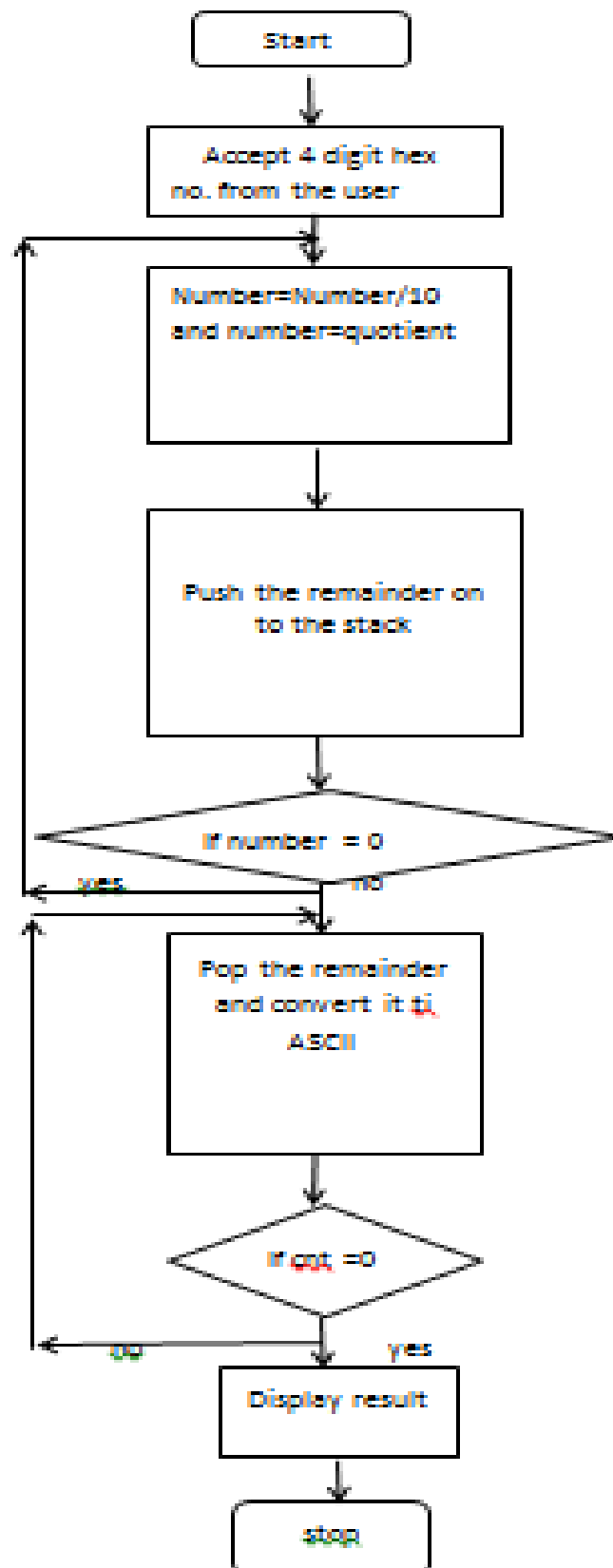
- a) Accept 4 Digit Hexadecimal Number.
- b) $\text{Number} = \text{Number} / 10$ & $\text{Number} = \text{Quotient}$
- c) Push the remainder on the stack
- d) If $\text{Number} = 0$, Go to next step otherwise go to step b
- e) Pop remainder , Convert into ascii & display untill all remainder are popped out

8. Bcd to hex Procedure:

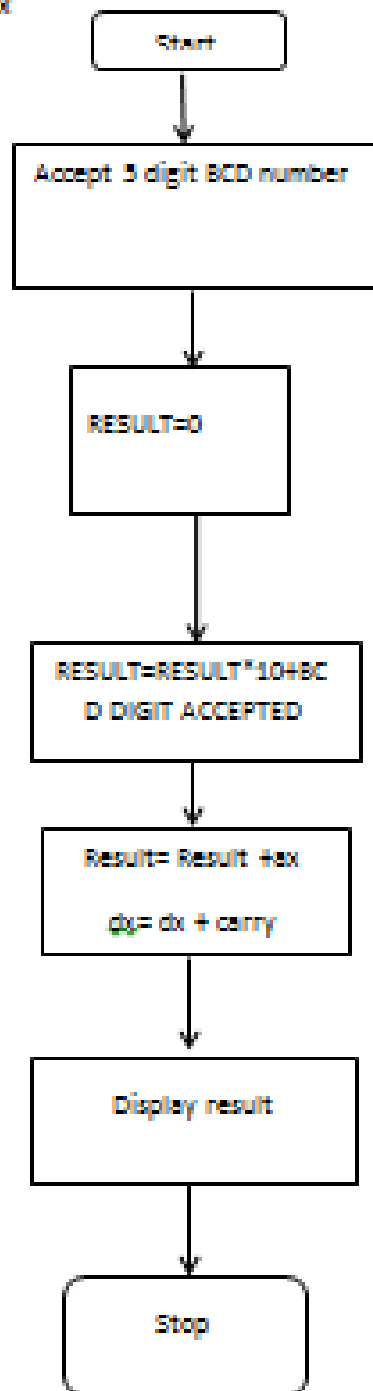
- a) $\text{RESULT} = 0$
- b) Accept BCD Digit
- c) Check whether all BCD Digits are accepted. If YES then go to step e otherwise go to next step
- d) $\text{RESULT} = \text{RESULT} * 10 + \text{BCD Digit accepted}$
- e) Display RESULT

FLOW CHART:

Flowchart of Hex to BCD



Flow chart for BCD to hex



CONCLUSION:

Hence we conclude that we can perform the Hex to BCD conversion & BCD to hex Conversion.

Output:

```
:[admin@localhost ~]$ vi conv.nasm
:[admin@localhost ~]$ nasm -f elf64 conv.nasm -o conv.o
:[admin@localhost ~]$ ld -o conv conv.o
:[admin@localhost ~]$ ./conv
```

```
##### Menu for Code Conversion #####
;1: Hex to BCD
;2: BCD to Hex
;3: Exit
```

```
;Enter Choice:1
```

```
;Enter 4 digit hex number::FFFF
```

```
;BCD Equivalent::65535
```

```
##### Menu for Code Conversion #####
;1: Hex to BCD
;2: BCD to Hex
;3: Exit
```

```
;Enter Choice:1
```

```
;Enter 4 digit hex number::00FF
```

```
;BCD Equivalent::255
```

```
##### Menu for Code Conversion #####
```

;1: Hex to BCD
;2: BCD to Hex
;3: Exit


;Enter Choice:2

;Enter 5 digit BCD number::65535

;Hex Equivalent::0FFFF

OUTCOME

Upon completion Students will be able to:

ELO3: Convert hexadecimal number to BCD and vice versa with the help of assembly language programming. 

QUESTIONS:

Oral Question Bank

- 1 Explain DIV instruction
- 2 Illustrate PUSH operation on stack
- 3 Differentiate conditional & unconditional jump instruction
- 4 Define hexadecimal and BCD number system.
- 5 Illustrate CMP and ROL instruction
- 6 How to convert hex to bcd and vice versa
- 7 Explain JNZ, JZ, JMP, JBE instruction with example.
- 8 Illustrate Read ,Write and exit call for 64 bit program
- 9 Illustrate accept procedure for number.

EXPERIMENT NO.7

Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

Session Plan

Time (min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill Competency Developed. /
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
10	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
20	Concept of GDTR, LDTR, I DTR	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension

TITLE: Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

OBJECTIVES:

1. To be familiar with the format of assembly language program structure and instructions.
2. To study GDTR, LDTR and IDTR.

PROBLEM DEFINITION:

Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

WORKING ENVIRONMENT:

- 1) CPU: Intel I5 Processor
- 2) OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
- 3) Editor: gedit, GNU Editor
- 4) Assembler: NASM (Netwide Assembler)
- 5) Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

1. CPU: Intel I5 Processor
2. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
3. Editor: gedit, GNU Editor
4. Assembler: NASM (Netwide Assembler)

INPUT: system file

OUTPUT: contents of GDTR, LDTR and IDTR.

THEORY:

Four registers of the 80386 locate the data structures that control segmented memory management called as memory management registers:

1. GDTR :Global Descriptor Table Register

These register point to the segment descriptor tables GDT. Before any segment register is changed in protected mode, the GDT register must point to a valid GDT. Initialization of the GDT and GDTR may be done in real-address mode. The GDT (as well as LDTs) should reside in RAM, because the processor modifies the accessed bit of descriptors. The instructions LGDT and SGDT give access to the GDTR.

2. LDTR :Local Descriptor Table Register

These register point to the segment descriptor tables LDT. The LLDT instruction loads a linear base address and limit value from a six-byte data operand in memory into the LDTR. The SLDT instruction always store into all 48 bits of the six-byte data operand.

3. IDTR Interrupt Descriptor Table Register

This register points to a table of entry points for interrupt handlers (the IDT). The LIDT instruction loads a linear base address and limit value from a six-byte data operand in memory into the IDTR. The SIDT instruction always store into all 48 bits of the six-byte data operand.

4. TR Task Register

This register points to the information needed by the processor to define the current task.

These registers store the base addresses of the descriptor tables (A descriptor table is simply a memory array of 8-byte entries that contain

Descriptors and descriptor stores all the information about segment) in the linear address space and store the segment limits.

SLDT: Store Local Descriptor Table Register

Operation: DEST \leftarrow 48-bit BASE/LIMIT register contents;

Description: SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table. SLDT is used only in operating system software. It is not used in application programs.

Flags Affected: None

SGDT: Store Global Descriptor Table Register

Operation: DEST \leftarrow 48-bit BASE/LIMIT register contents;

Description: SGDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined. Otherwise, if the operand-size attribute is 16 bits, the next 4 bytes are assigned the 32-bit BASE field of the register. SGDT and SIDT are used only in operating system software; they are not used in application programs.

Flags Affected: None

SIDT: Store Interrupt Descriptor Table Register

Operation: DEST \leftarrow 48-bit BASE/LIMIT register contents;

Description: SIDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 32 bits, the next three bytes are assigned the BASE field of the register, and the fourth byte is written with zero. The last byte is undefined. Otherwise, if the operand-size attribute is 16 bits, the next 4 bytes are assigned the 32-bit BASE field of the register. SGDT and SIDT are used only in operating system software; they are not used in application programs.

Flags Affected: None

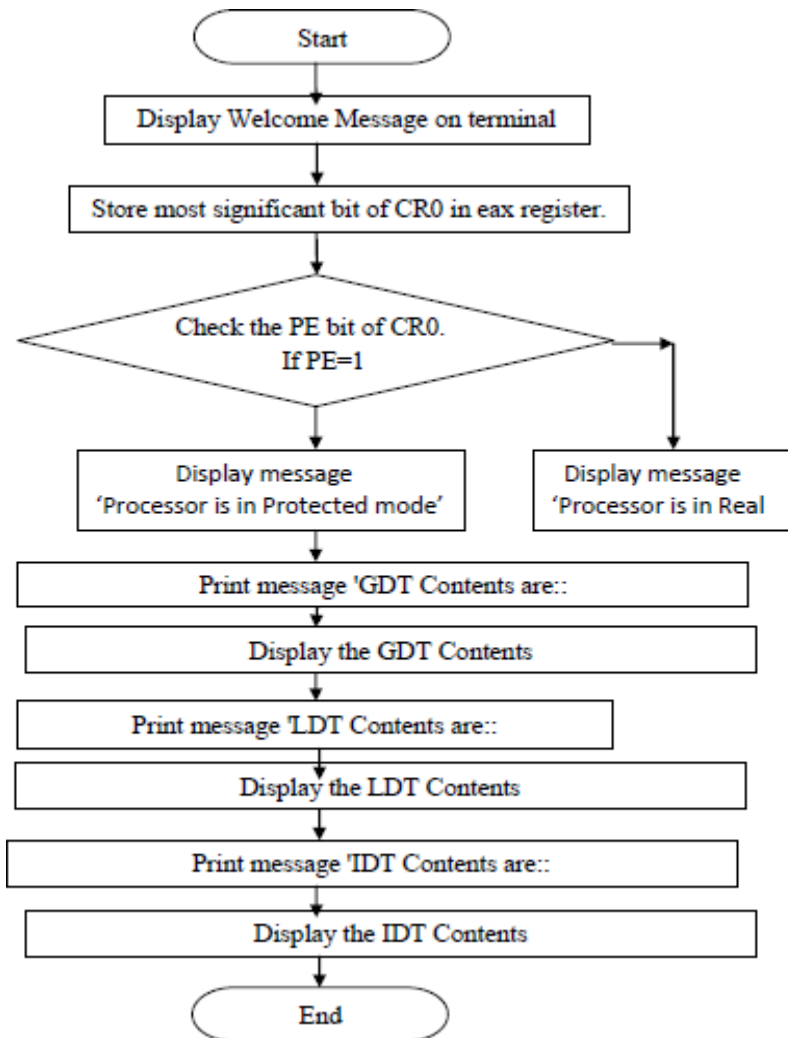
MATHEMATICAL MODEL:

ALGORITHM:

1. Display welcome message on terminal using macro disp.
2. Store most significant bit of CR0 in eax register.
3. Check the PE bit of CR0.

4. If PE=1 then display message “Processor is in Protected mode”.
5. And if PE=0 then display message “Processor is in Real mode”.
6. Then copies/stores the contents of GDT, IDT, LDT using sgdt, sidt, sltd instruction.
7. Display their contents using macro function disp and disp_num.

FLOWCHART:




CONCLUSION:

In this way, we use GDTR, LDTR and IDTR in Real mode.

OUTCOME:

Upon completion Students will be able to:

ELO6: Apply the concept of real mode and protected mode in 8086 ALP to implement a program to display values from GDTR, LDTR, IDTR, TR and MSW registers. 

FAQ:

Oral Question Bank

1. Different types of operating modes?
2. State the difference between different operating modes?
3. Explain descriptor and descriptor table?
4. What is CR0 what are its contents?
5. Explain different types of descriptor table?
6. Explain how to load and store the contents of descriptor table?

EXPERIMENT NO. 08 and 09

Non-overlapped and overlapped block transfer

Session Plan

Time (min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
10	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
20	Concept of Overlapped & Non-overlapped block Transfer	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension

TITLE: Non-overlapped and overlapped block transfer

OBJECTIVES:

4. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
5. To learn the instructions related to String and use of Direction Flag.
6. To be familiar with data segments.
7. Implement non-overlapped and overlapped block transfer.

PROBLEM DEFINITION:

Write X86/64 ALP to perform non-overlapped and overlapped block transfer (with and without string specific instructions). Block containing data can be defined in the data segment.

WORKING ENVIRONMENT:

- 11) CPU: Intel I5 Processor
- 12) OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
- 13) Editor: gedit, GNU Editor
- 14) Assembler: NASM (Netwide Assembler)
- 15) Linker:-LD, GNU Linker

S/W AND H/W REQUIREMENT:

Software Requirements

14. CPU: Intel I5 Processor
15. OS:- Windows XP (16 bit Execution), Fedora 18 (32 & 64 bit Execution)
16. Editor: gedit, GNU Editor
17. Assembler: NASM (Netwide Assembler)
Linker:-LD, GNU Linker

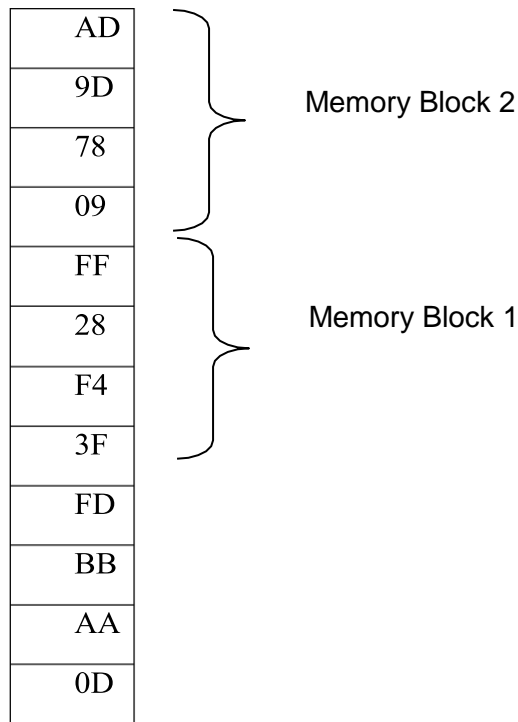
INPUT: Input data specified in program.

OUTPUT: Display the data present in destination block.

THEORY:

1. Non-overlapped blocks:

In memory, two blocks are known as non-overlapped when none of the element is common.



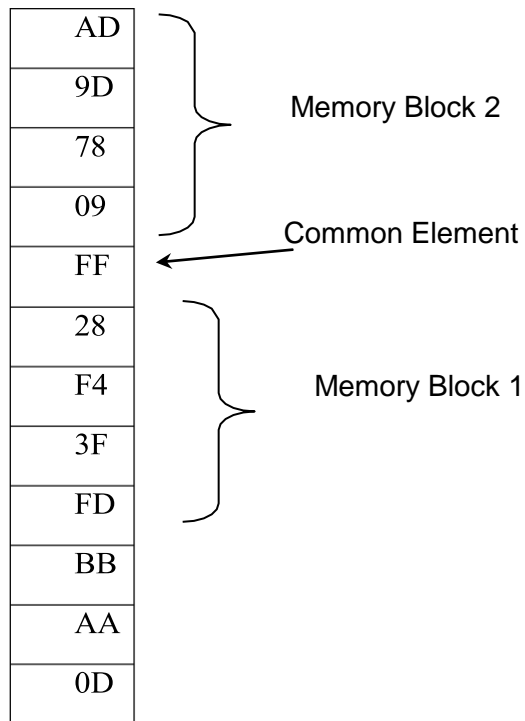
In above example of non-overlapped blocks there is no common element between block 1 & 2.

While performing block transfer in case of non-overlapped blocks, we can start transfer from starting element of source block to the starting element of destination block and then we can transfer remaining elements one by one.

2. Overlapped block:

In memory, two blocks are known as overlapped when at least one element is common between two blocks.

While performing block transfer we have to see which element/s of source block is/are overlapped. If ending elements are overlapped then start transferring elements from last and if starting elements are overlapped then start transfer from first element.



Instructions needed:

1. **MOVS**B-Move string bytes.
2. **JNE**-Jump if not equal
3. **AND**-AND each bit in a byte or word with corresponding bit in another byte or word
4. **INC**-Increments specified byte/word by 1
5. **DEC**-Decrements specified byte/word by 1
6. **JNZ**-Jumps if not equal to Zero
7. **CMP**-Compares to specified bytes or words

8. **JBE**-Jumps if below of equal
9. **CALL**-Transfers the control from calling program to procedure.
10. **RET**-Return from where call is made

ALGORITHM:

1) Non –Overlapped Block Transfer:

In non-overlapped block transfer Source Block & destination blocks are different. Here we can transfer byte by byte data or word by word data from one block to another block.

- i) Start
- ii) Initialize data section
- iii) Initialize RSI to point to source block
- iv) Initialize RDI to point to destination block
- v) Initialize the counter equal to length of block
- vi) Get byte from source block & copy it into destination block
- vii) Increment source & destination pointer
- viii) Decrement counter
- ix) If counter is not zero go to step vi
- x) Display Destination Block
- xi) Stop

2) Overlapped Block Transfer.

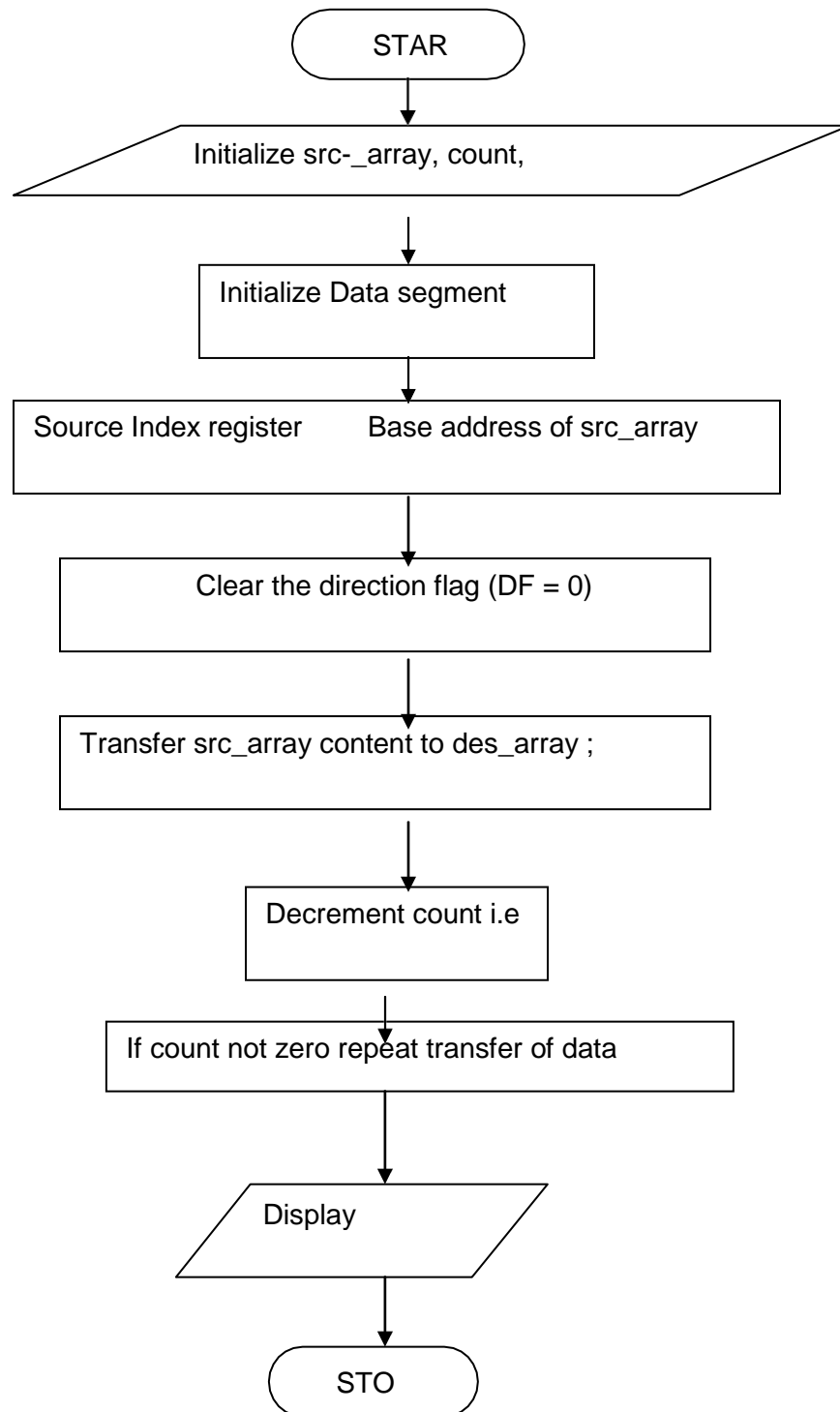
In Overlapped block transfer there is only one block & within the same block We are transferring the data.

- i) Start
- ii) Initialize data section
- iii) Accept the overlapping position from the user
- iv) Initialize RSI to point to the end of source block
- v) Add RSI with overlapping Position & use it as pointer to point to End of Destination Block.
- vi) Initialize the counter equal to length of block
- vii) Get byte from source block & copy it into destination block
- viii) Decrement source & destination pointer
- ix) Decrement counter
- x) If counter is not zero go to step vii

- xi) Display Destination Block
- xii) Stop

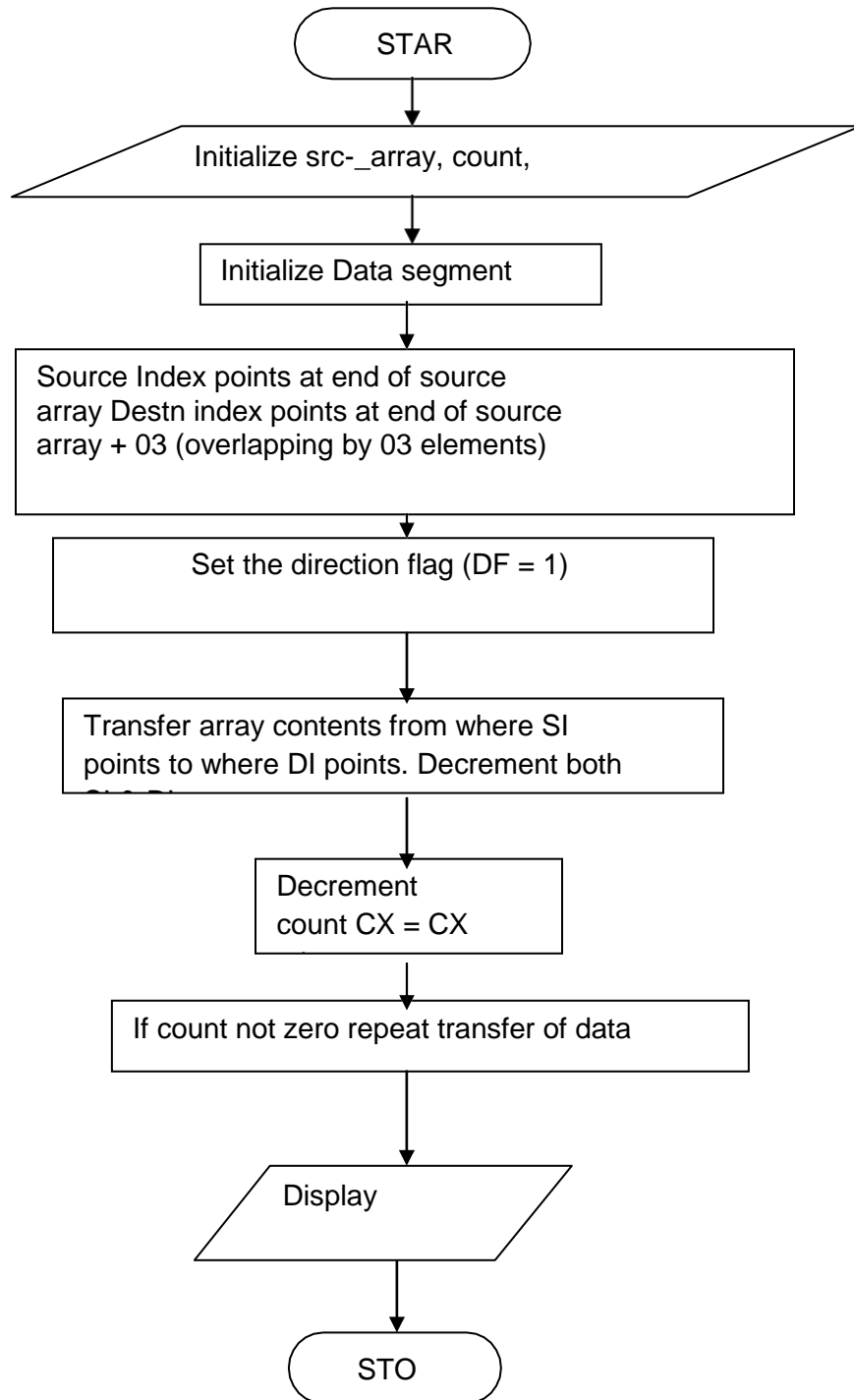
Flowchart:

NON-OVERLAPPED BLOCK TRANSFER



Flowchart:

OVERLAPPED BLOCK TRANSFER



Mathematical Model:

Let $S = \{s, e, X, Y, Fme, mem \mid \Phi_s\}$ be the programmer's perspective of Non-over Lapped & overlapped Block Transfer.

1) Non-Overlapped Block Transfer.

Let X be the input such that

$X = \{X_1, X_2, X_3, \dots\}$

Such that there exists function $f_{X_1}: X_1 \longrightarrow \{0,1\}$

X2 Source Array

Let $X_2 = \{\{b_7 \dots b_0\} \{b_7 \dots b_0\} \dots \{b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0\}\}$ where $\forall b_i \in X_1$

There exists a function $f_{X_2}: X_2 \longrightarrow \{\{00h \dots FFh\} \{00h \dots FFh\} \{00h \dots FFh\} \dots\}$

X3 is destination array before transfer

Let $X_3 = \{\{b_7 \dots b_0\} \{b_7 \dots b_0\} \dots \{b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0\}\}$ where $\forall b_i \in X_1$

There exists a function $f_{X_3}: X_3 \longrightarrow \{\{00h \dots FFh\} \{00h \dots FFh\} \{00h \dots FFh\} \dots\}$

X4 is the two digit count value equal to length of block

Let $X_4 = b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$ where $\forall b_i \in X_1$

There exists a function $f_{X_4}: X_4 \longrightarrow \{01h, 02, \dots, FFh\}$

X5 is the Single digit choice.

Let $X_5 = b_3 \ b_2 \ b_1 \ b_0$ where $\forall b_i \in X_1$

There exists a function $f_{X_5}: X_5 \longrightarrow \{1, 2, 3\}$

Let Y is the Destination Block after transfer

Let $Y = \{\{b_7 \dots b_0\} \{b_7 \dots b_0\} \dots \{b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0\}\}$ where $\forall b_i \in X_1$

Where $\forall b_i \in X_1$

$Y \longrightarrow \{\{00h \dots FFh\} \{00h \dots FFh\} \{00h \dots FFh\} \dots\}$

Fme = { F1 ,F2,F3 ,F4}

Where F1 = Display X2 & X3 (Source block & Destination block before Transfer)

F2 = Copy Byte from X2 (source block) to Y(destination block)

F3 = Increment Source & destination pointer, $X_4 = X_4 - 1$, If not zero repeat F2 & F3

F4= Display Y

2) Overlapped Block Transfer.

Let X be the input such that

$X = \{ X1, X2, X3, \dots \}$

Such that there exists function $f_{X1}: X1 \longrightarrow \{0,1\}$

X2 Source Array

Let $X2 = \{ \{b7 \dots b0\} \{b7 \dots b0\} \dots \{b7b6 b5 b4 b3 b2 b1$

$b0\} \{00H\} \{00H\} \{00H\} \{00H\} \}$ where $\forall bi \in X1$

There exists a function $f_{X2}: X2 \longrightarrow \{ \{00h \dots FFh\} \{00h \dots FFh\} \{00h \dots FFh\} \dots \{00h\} \{00h\} \{00h\} \{00h\} \{00h\} \}$

X3 is the two digit count value equal to length of block

Let $X3 = b7 b6 b5 b4 b3 b2 b1 b0$ where $\forall bi \in X1$

There exists a function $f_{X3}: X3 \longrightarrow \{ 01h, 02, \dots, FFh \}$

X4 is the Single digit choice.

Let $X4 = b3 b2 b1 b0$ where $\forall bi \in X1$

There exists a function $f_{X4}: X4 \longrightarrow \{ 1, 2, 3 \}$

X5 is the position of overlapping

Let $X4 = b7 b6 b5 b4 b3 b2 b1 b0$ where $\forall bi \in X1$

There exists a function $f_{X5}: X5 \longrightarrow \{ 01h, 02h, 03h, 04h, 05h \}$

Let Y is the Destination Block after transfer

Let $Y = \{ \{b7 \dots b0\} \{b7 \dots b0\} \dots \{b7 b6 b5 b4 b3 b2 b1 b0\} \}$ where $\forall bi \in X1$

Where $\forall bi \in X1$

$Y \longrightarrow \{ \{00h \dots FFh\} \{00h \dots FFh\} \{00h \dots FFh\} \dots \}$

Fme = { F1 ,F2,F3 ,F4,F5,F6}

Where F1 = Accept X5

F2 = Point to end of source block X2

F3 = pointer to Y= (pointer to X2)+X5

F4 = Copy Byte from X2 (source block) to Y(destination block)

F5 = Decrement Source & destination pointer, $X3=X3-1$, If not zero repeat F4&F5

F6 = Display Y

Commands

- To assemble

```
nasm -f elf 64 hello.nasm -o hello.o
```

- To link

```
ld -o hello hello.o
```

- To execute -

```
./hello
```

CONCLUSION

Hence we conclude that we can perform non-overlapped & overlapped block Transfer with & without using string instructions.

Output:

```
[root@comppl208 nasm-2.10.07]# gedit nonoverlap26.asm
[root@comppl208 nasm-2.10.07]# nasm -f elf64 nonoverlap26.asm
[root@comppl208 nasm-2.10.07]# ld -o nonoverlap26 nonoverlap26.o
[root@comppl208 nasm-2.10.07]# ./nonoverlap26
```

****Block contents before transfer:**

***_*Source block contents 01 02 03 04 05**

***_*Destination block contents 00 00 00 00 00**

*****Nonoverlap block transfer*****

1.Block transfer without string

2.Block transfer with string

3.exit 1

****Block contents after transfer:**

***_*Source block contents 01 02 03 04 05**

***_*Destination block contents 01 02 03 04 05**

```
[root@comppl208 nasm-2.10.07]# ./nonoverlap26
```

****Block contents before transfer:**

```
*_*Source block contents 01 02 03 04 05
*_Destination block contents 00 00 00 00 00
```

Nonoverlap block transfer

- 1.Block transfer without string
- 2.Block transfer with string
- 3.exit 2

```
*_*Source block contents 01 02 03 04 05
*_Destination block contents 01 02 03 04 05
**Block contents after transfer:
*_Source block contents 01 02 03 04 05
*_Destination block contents 01 02 03 04 05
```

```
[root@comppl208 nasm-2.10.07]# ./nonoverlap26
```

```
**Block contents before transfer:
*_Source block contents 01 02 03 04 05
*_Destination block contents 00 00 00 00 00
```

Nonoverlap block transfer

- 1.Block transfer without string
 - 2.Block transfer with string
 - 3.exit 3
- ```
[root@comppl208 nasm-2.10.07]#
```

## OUTCOME

**Upon completion Students will be able to:**

**ELO2:** Apply the concept of overlapped and non-overlapped block transfer in the program with or without using string instruction. 

### Oral Question Bank

- 
1. What is difference between overlapped & non-overlapped block Transfer?
  2. What is the difference between without & with string instruction Execution?
  3. What is the use of direction flag?
  4. What is different string instructions used in the program?
  5. What is different string instructions used in the program?

## **EXPERIMENT NO.10**

**Multiplication of two 8 bit nos. using Successive addition  
and Shift and add method**

## Session Plan

| <b>Time<br/>( min)</b> | <b>Content</b>                                     | <b>Learning Aid /<br/>Methodology</b> | <b>Faculty<br/>Approach</b>             | <b>Typical<br/>Student<br/>Activity</b> | <b>Skill /<br/>Competency<br/>Developed.</b>                 |
|------------------------|----------------------------------------------------|---------------------------------------|-----------------------------------------|-----------------------------------------|--------------------------------------------------------------|
| 10                     | Relevance and significance of Problem statement    | Chalk & Talk ,<br>Presentation        | Introduces,<br>Explains                 | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal                                  |
| 10                     | Explanation of Problem statement                   | Chalk & Talk ,<br>Presentation        | Introduces,<br>Facilitates,<br>Explains | Listens,<br>Participates,               | Knowledge,<br>intrapersonal,<br>Application                  |
| 20                     | Concept of Successive odd and Shift and add method | Demonstration,<br>Presentation        | Explains,<br>Facilitates,<br>Monitors   | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>interpersonal<br>Application |
| 60                     | Implementation of problem statement                | N/A                                   | Guides, Facilitates<br>Monitors         | Participates,<br>Discusses              | Comprehension,<br>Hands on<br>experiment                     |
| 10                     | Assessment                                         | N/A                                   | Monitors                                | Participates,<br>Discusses              | Knowledge,<br>Application                                    |
| 10                     | Conclusions                                        | Keywords                              | Lists, Facilitates                      | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>Comprehension                |

**TITLE:** Multiplication of two 8 bit nos. using Successive addition and Shift and add method

**OBJECTIVES:**

1. Understand the implementation.
2. To interpret the Microprocessor Interfacing paradigms.
3. To express and apply the method of odd, add and shift method.
4. Understand implementation of arithmetic instruction of 8086.

**PROBLEM STATEMENT:**

Write 8086/64 ALP to perform multiplication of two 8 bit hexadecimal nos. Use successive addition & shift & add method, Accept i/p from the user.

**HARDWARE REQUIRED:**

**CPU:** Intel i5 Processor

**OS:** Windows XP (16 bit execution), Fedora 18 32 & 64 bit execution

**SOFTWARE REQUIRED:**

**Editor:** gedit, GNU Editor

**Assembler:** NASM (Netwide Assembler)

**Linker:** GNU Linker

**INPUT:** Two hex nos.

For e.g. AL=12H, BL= 10H

**OUTPUT:**

Result : D120H

**THEORY:**

There are 5 basic form of define reverse directives.

**Directives**

DD

DW

DD

DQ

DT

RESB

RESW

RESQ

REST

**Purpose**

Define byte

Define word

Define doubleword

Define quad word

Define Ten byte

Reserve byte

Reserve word

Reserve quad word

Reserve ten word

**Instructions Needed:**

MOV : Move or copy word

ROR : Rotate to right

AND : Logical AND

INC : Increment

DEC : Decrement

JNZ : Jump if not zero

CMP : Compare

JNC : Jump if no carry

JBE : Jump if below

**Shift & Add method:**

The method taught in school for multiplying decimal no. is based on calculated partial products, shifting it to the left & then adding them together. Shift & add multiplication is similar to the multiplication performed by paper & pencil. This method adds the multiplicand X to itself Y

times where Y denotes the multiplier. To multiply two nos. by paper & pencil placing the intermediate product in the appropriate positions to the left of earlier product.

1. Consider 1 byte is in AL & another in BL
2. We have to multiply byte in AL with byte in BL
3. In this method, you add 1 with itself & rotate other no. each times & shift it by 1 bit n left along with carry
4. If carry is present add 2 NOS.
5. Initialize count to n as we are scanning for n digit decrement counter each time, the bits are added

The result is stored in AX, display the result.

Eg., AH=11H, BL=10H, Count=n

**Step 1:**

$$\mathbf{AX=11 + 11 = 22H}$$

Rotate BL by 1 bit to left along with carry 0001 0000

$$\mathbf{B1=10H \quad \underline{0010\ 0000} \ (20)}$$

**Step 2:**

Decrement count =3

Check for carry, carry is not there So Add with itself

$$\mathbf{AX=22+22=44H}$$

Rotate BL to left

$$\mathbf{BL=0 \quad \underline{0000\ 0000} \ (00)}$$

No carry

**Step 3:**

Decrement count=2

Add no. with itself

$$\mathbf{AX=44+44=88H}$$

Rotate BL to left

$$\mathbf{B2=0 \text{ (carry) } \underline{1000\ 0000} \text{ (80)}}$$

**Step 4:**

Decrement count=0

Add no. with itself,

$$\mathbf{AX=88+88=110H}$$

Rotate BL to left

$$\mathbf{BL=0 \text{ (carry) } \underline{1000\ 0000} \text{ (80)}}$$

**Step 5:**

Decrement count =0, carry is generated

Add Ax, BX

$$\mathbf{0110+0000=0110H}$$

i.e.,

$$\mathbf{11H+10H=0110H}$$

**MATHEMATICAL MODEL:**



**Let  $S=\{s, e, x, y, \text{time}, \text{mem } \Phi s\}$**  be program perspective of multiplication of two 8 bit hexadecimal Nos.

Let X be the input such that

$X=\{x_1, x_2, x_3, \dots\}$

Such that there exists function  $f_x: x \rightarrow \{0, 1\}$

$X_2=\{s \text{ the two digit multiply}\}$

Let  $x_2 = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Where  $x_{b1} \in x_1$  Here exists a function  $f_{x2}: x_2 \rightarrow \{00h, 01h, 02h, \dots, ffh\}$

$X_3$  is the single digit choice

Let  $x_4 = b_3 b_2 b_1 b_0$  where  $(b_i \in x_1 \text{ there Let } y \text{ is the output})$

$Y = b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$  (where  $b_i \in x_1$ )

$Y = \{0000b, 0001b, \dots, fffb\}$

Time ( for successive addition) =  $\{f_1, f_2, f_3, f_4\}$

Where  $f_1 = \text{Accept } x_2 \ \& \ x_3$

$F_2 = \text{Addition (Repeat } x_2+x_1+ \dots \& \text{ till } x_3=00)$

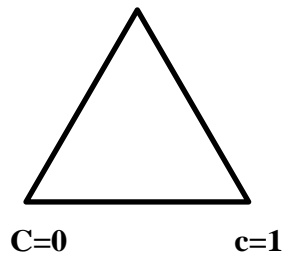
$F_3 = \text{display}$

Time (for shift and add)=  $\{f_1, f_2, f_3\}$

Where

$F_1 = \text{Accept } x_1 \& x_3$

F2= shift x3>>



Shift x2 << & x5 = x5-1

Shift x2 << & add with y, x5=x5-1

Repeat till x5=0

F3 = display y

C=0                      c=1

Shift x2 << x5 << x5-1

Shift x2 << & add with y x5 = x5-1

Repeat till x5=0

F3= display

Add it with result & then decrement counter display result.

## **ALGORITHM:**

### **1. Successive Addition**

1. Start
2. Get the 1<sup>st</sup> no. from user

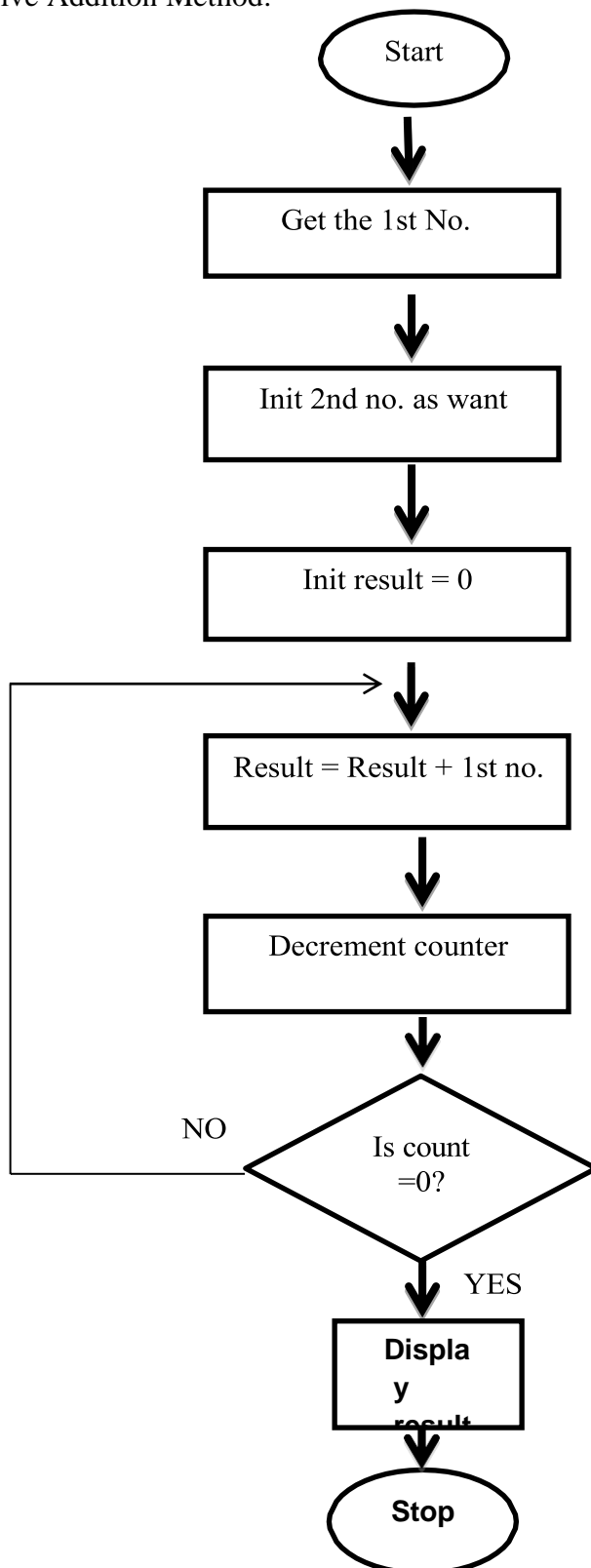
3. Get 2<sup>nd</sup> no. from user the no. will get as counter.
4. Initialize result=0
5. Add the 1<sup>st</sup> no. of itself as multi times
6. Decrement counter
7. Compare the counter with „0“
  1. If count  $\neq 0$   
Goto step 5
  2. Else
    1. Display the result
    2. Stop

### **Shift Addition Method**

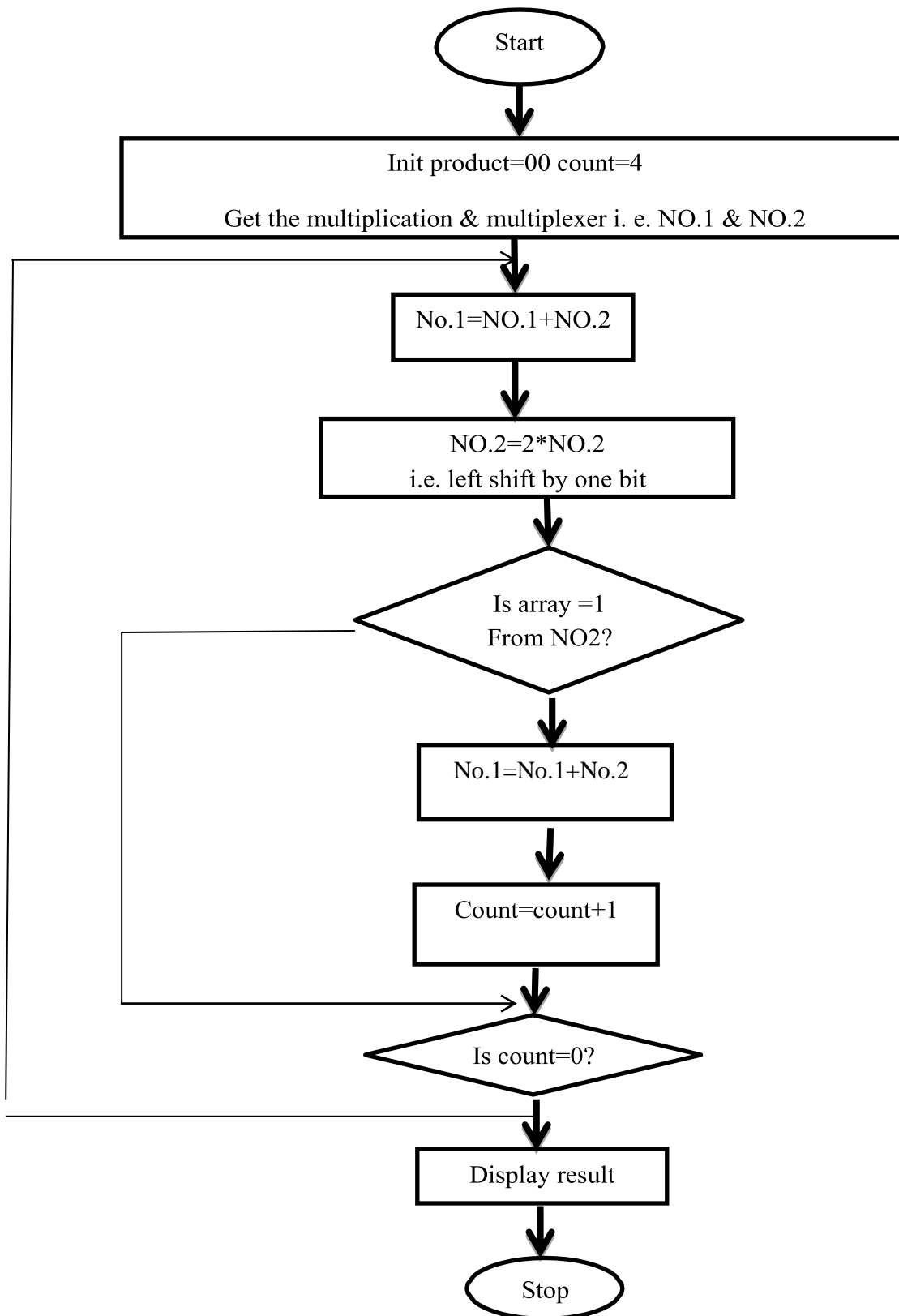
1. Start
2. Get the 1<sup>st</sup> no. from user
3. Initialize count =0
4. No.1 = no\*2
5. Get the 2<sup>nd</sup> no. from user
6. Shift multiplier to left along with carry
7. Check for carry, if present goto step 4
8. No. 1 = no.1 + shifted no.2
9. Decrement counter
10. If not zero, goto step 6
11. Display result
12. Stop

## FLOWCHART:

Successive Addition Method:



## 1. Add & shift Method



## CONCLUSION:

From this program we have studied the multiplication of 8 bit nos. and in this we have studied and implemented the program of successive addition and shift & add method.

Output:

```
:[root@comppl208 nasm-2.10.07]# nasm -f elf64 multi26.asm
[root@comppl208 nasm-2.10.07]# ld -o multi26 multi26.o
[root@comppl208 nasm-2.10.07]# ./multi26
```

\*\*\*Multiplication by add & shift\*\*\*

Enter two digit number: 50

Enter two digit number: 02

Multiplication is: 00A0

```
:[root@comppl208 nasm-2.10.07]# nasm -f elf64 muladd26.asm
[root@comppl208 nasm-2.10.07]# ld -o muladd26 muladd26.o
[root@comppl208 nasm-2.10.07]# ./muladd26
```

\*\*\*Multiplication by successive addition\*\*\*



Enter two digit number: 05

Enter two digit number: 20

Multiplication is: 00A0

## OUTCOME

**Upon completion Students will be able to:**

**ELO4:** Choose different multiplication methods such as add & shift and successive addition for multiplication of two numbers without using MUL instruction.  

### **Oral Question Bank**

- 1 What are the two methods of multiplication ?
- 2 What is concept of Successive Add?
- 3 What is concept of Add & Shift?
- 4 Discuss logic of successive addition?
- 5 What is the size of each stack?
- 6 What are the different data types?
- 7 Which co-processor you have used?
- 8 Which arithmetic instruction you used?

## **EXPERIMENT NO.11**

**ALP to implement DOS commands TYPE,COPY, &  
DELETE using File Operations.**



## Session Plan

| Time ( min) | Content                                         | Learning Aid / Methodology  | Faculty Approach                  | Typical Student Activity         | Skill / Competency Developed.                       |
|-------------|-------------------------------------------------|-----------------------------|-----------------------------------|----------------------------------|-----------------------------------------------------|
| 10          | Relevance and significance of Problem statement | Chalk & Talk , Presentation | Introduces, Explains              | Listens, Participates, Discusses | Knowledge, intrapersonal                            |
| 10          | Explanation of Problem statement                | Chalk & Talk , Presentation | Introduces, Facilitates, Explains | Listens, Participates,           | Knowledge, intrapersonal, Application               |
| 15          | Concept of DOS commands                         | Demonstration, Presentation | Explains, Facilitates, Monitors   | Listens, Participates, Discusses | Knowledge, intrapersonal, interpersonal Application |
| 10          | Concept of FILE operations                      | Demonstration, Presentation | Explains, Facilitates, Monitors   | Listens, Participates, Discusses | Knowledge, intrapersonal, interpersonal Application |
| 60          | Implementation of problem statement             | N/A                         | Guides, Facilitates Monitors      | Participates, Discusses          | Comprehension, Hands on experiment                  |
| 10          | Assessment                                      | N/A                         | Monitors                          | Participates, Discusses          | Knowledge, Application                              |
| 05          | Conclusions                                     | Keywords                    | Lists, Facilitates                | Listens, Participates, Discusses | Knowledge, intrapersonal, Comprehension             |

**TITLE:**

ALP to implement DOS commands TYPE,COPY, & DELETE using File Operations.

**OBJECTIVES:**

8. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
9. To be familiar with DOS Commands.
10. Implement file operations.

**PROBLEMDEFINITION:**

Write x86 menu driven ALP to implement DOS commands TYPE,COPY, & DELETE using File Operations. User is supposed to provide command line arguments in all cases.

**WORKING ENVIRONMENT:**

- 16) CPU: Intel I5 Processor
- 17) OS:- Windows XP (16 bit Execution ), Fedora 18 (32 & 64 bit Execution)
- 18) Editor: gedit, GNU Editor
- 19) Assembler: NASM (Netwide Assembler)
- 20) Linker:-LD, GNU Linker

**S/W AND H/W REQUIREMENT:****Software Requirements**

18. CPU: Intel I5 Processor
19. OS:- Windows XP (16 bit Execution ), Fedora 18 (32 & 64 bit Execution)
20. Editor: gedit, GNU Editor
21. Assembler: NASM (Netwide Assembler)  
Linker:-LD, GNU Linker

**INPUT:** Input data specified in program( TEXT FILE).

**OUTPUT:** Display the data Present in File.

## **THEORY:**

### **PSP-:**

Whenever DOS loads a program for execution it creates a 256 byte data structure called PSP.

It is available at 1st paragraph of true memory program itself is located and load above psp.

There are two types of disk access method.

1. ASCII method
2. Handles.

### **Handles-**

Use of file handles to file operation the file management function access the file in fashion similar

To that used under UNIX.

### **ASCII Function calls -:**

1. 3H- Creates a file DS:DX.
2. 30H- Open a file.
3. 3EH- Close file.
4. 40H- Write into file.
5. 41H- Delete a file.
6. 17H- Rename a file.
7. 4EH- To check if file exists.

### **File Control Block (FCB)-**

As FCB is a 37 file data structure allocated with the application program memory.

### **Common FCB Record Operation-:**

0FH: Open a File.

10H: Close file.

16H: Create a file.

14H: perform segment Read.

15H: perform segment write.

22H: perform random.

### **ALGORITHM FOR TYPE:**

1. Start
2. Get source file name and destination file name from command tail.
3. If file is not present display error message as “File not found” and stop.
4. If present, open the file in read mode.
5. Read the contents of file and print the data on the screen
6. Stop

### **ALGORITHM FOR COPY:**

1. Start.
2. Get source file name and destination file name from command tail.
3. If file is not present display error message as “File not found” and stop.
4. If present, open the file in read mode.
5. Read name of destination file and open it in read mode.
6. Read the contents of file source and write it into destination file.
7. Stop.

### **CONCLUSION**

Hence we conclude that we Implement DOS commands like TYPE,COPY, DELET using file

operations

**Output:**

```
C:\>CD TASM
C:\tasm>COPY CON TEST.TXT
HELLO,THIS IS A TEST
Overwrite TEST.TXT? (Yes/No/All): Y
^Z
1 file(s) copied.
C:\tasm>ASS2 TEST.TXT
HELLO, THIS IS A TEST
C:\>CD TASM
C:\tasm>COPY CON TEST.TXT
HELLO,THIS IS A TEST
Overwrite TEST.TXT? (Yes/No/All): Y^Z
1 file(s) copied.

C:\tasm>ASS2 TEST.TXT TEXT1.TXT
File Copied
```

**OUTCOME**

**ELO8:** Implement DOS commands like TYPE,COPY, DELET using file operations.



**FAQ:**

**Oral Question Bank**

1. What are command Line arguments?
2. List and Explain Different DOS Commands?
3. Explain different File operations?
4. Explain Type ,COPY Command.

## **EXPERIMENT NO.12**

**Program to analyze the difference between near and far  
procedure to find number of lines, blank spaces & occurrence  
of character using nasm.**

## Session Plan

| <b>Time<br/>( min)</b> | <b>Content</b>                                             | <b>Learning Aid /<br/>Methodology</b> | <b>Faculty Approach</b>              | <b>Typical<br/>Student<br/>Activity</b> | <b>Skill / Competency<br/>Developed.</b>                     |
|------------------------|------------------------------------------------------------|---------------------------------------|--------------------------------------|-----------------------------------------|--------------------------------------------------------------|
| 10                     | Relevance and significance of Problem statement            | Chalk & Talk ,<br>Presentation        | Introduces, Explains                 | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal                                  |
| 10                     | Explanation of Problem statement                           | Chalk & Talk ,<br>Presentation        | Introduces,<br>Facilitates, Explains | Listens,<br>Participates,               | Knowledge,<br>intrapersonal,<br>Application                  |
| 15                     | Concept of FAR PROCEDURES and PUBLIC and EXTERN directives | Demonstration,<br>Presentation        | Explains, Facilitates,<br>Monitors   | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>interpersonal<br>Application |
| 10                     | Concept of FILE AND FILE OPERATION                         | Demonstration,<br>Presentation        | Explains, Facilitates,<br>Monitors   | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>interpersonal<br>Application |
| 60                     | Implementation of problem statement                        | N/A                                   | Guides, Facilitates<br>Monitors      | Participates,<br>Discusses              | Comprehension,<br>Hands on experiment                        |
| 10                     | Assessment                                                 | N/A                                   | Monitors                             | Participates,<br>Discusses              | Knowledge,<br>Application                                    |
| 05                     | Conclusions                                                | Keywords                              | Lists, Facilitates                   | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>Comprehension                |

**TITLE:**

Program to analyze the difference between near and far procedure to find number of lines, blank spaces & occurrence of character using nasm.

**OBJECTIVES:**

11. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
12. To be familiar with FAR PROCEDURES and PUBLIC and EXTERN directives.
13. Implement file operations.

**PROBLEM DEFINITION:**

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.

**WORKING ENVIRONMENT:**

- 21) CPU: Intel I5 Processor
- 22) OS:- Windows XP (16 bit Execution ), Fedora 18 (32 & 64 bit Execution)
- 23) Editor: gedit, GNU Editor
- 24) Assembler: NASM (Netwide Assembler)
- 25) Linker:-LD, GNU Linker

**S/W AND H/W REQUIREMENT:****Software Requirements**

22. CPU: Intel I5 Processor
23. OS:- Windows XP (16 bit Execution ), Fedora 18 (32 & 64 bit Execution)
24. Editor: gedit, GNU Editor
25. Assembler: NASM (Netwide Assembler)  
Linker:-LD, GNU Linker

**INPUT:** Input data specified in program( TEXT FILE).

**OUTPUT:** Display the data Present in File.



## THEORY:

In this program the user is going to enter the text file name on command line. Suppose user has enter file named abc.txt, which is present on stack memory in following form :

|                                 |          |
|---------------------------------|----------|
| Arguments                       | abc.txt  |
| .asm program name               | file.asm |
| no. of arguments in the program | 1        |
| Current SP                      |          |

By accessing this stack we can get the file name in the program. Then by calling the interrupt for opening the file and closing the file we can access the contents of a file. The file contents are taken into buffer memory for display.

### System calls for File –

#### Open File –

open the file for reading

```
mov eax,5 ;system call number(sys_open)
movebx,file_name ;file name
mov ecx,0 ;file access mode
mov edx,0777 ;read,write and execute by all
int 80h ;call kernel

mov [fd_in],eax ;file descriptor
bt eax,31 ;negative value in eax indicates error
jnc conti1
```

```
printfnotmsg,fmsg_len
jmp exit
```

conti1:

```
printopenmsg,msg_len
printfilemsg,fmsg_len
```

## Read File -

read from file

readfile:

```
mov eax,3 ;system call number(sys_read)
movebx,[fd_in] ;file descriptor
movecx,fbuff ;pointer to the input buffer
movedx,fb_len ;buffer size i.e the number of bytes to read
int 80h ;call kernel

mov [act_len],eax
cmp eax,0
je next1
printfbuff,[act_len]
jmpreadfile
```

next1:

## Close File -

close the file

```
mov eax,6 ;system call number(sys_close)
movebx,[fd_in] ;file decriptor
int 80h
```

## Instruction:

### 1) POP –

**Description** - This instruction going to pop the contents from stack into destination mentioned in the instruction. The stack pointer is first incremented and then the contents are popped.

e. g. pop ebx

### 2).JNS -

**Description**-This instruction is going jump on label mentioned if sign flag is not set.

**Flag:** SF

e.g. jns up

### 3) JS –

**Description-** This instruction is going jump on label mentioned if sign flag is set.

**Flag:** SF

e. g. js up

### 4) JZ-

**Description-** This instruction is going jump on label mentioned if zero flag is set.

**Flag:** ZF

e. g. jz up

#### ➤ **EXTERN**

Informs the assembler that the names, procedures, labels declared after this directive have already been defined in some other assembly language module while in the other module where the names, procedures & labels actually appear, they must be declared Global using GLOBAL directive

#### ➤ **GLOBAL**

The labels, variables, constants or procedures declared GLOBAL may be used by other modules of program. Once the variable is declared GLOBAL, it can be used by any module in the program.

#### ➤ **FAR**

Used to declare the procedure far from the segment from where we are calling it.

#### ➤ **PUBLIC**

Used to declare procedure publically

**When executing a far call, the processor performs these actions:**

1. Pushes current value of the CS register on the stack.
2. Pushes the current value of the IP register on the stack.
3. Loads the base address of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the IP register.
5. Begins execution of the called procedure.

**When executing a far return, the processor does the following:**

1. Pops the top-of-stack value (the return instruction pointer) into the IP register.

2. Pops the top-of-stack value (the segment selector for the code segment being returned to) into the CS register.
3. (If the RET instruction has an optional  $n$  argument.) Increments the stack pointer by the number of bytes specified with the  $n$  operand to release parameters from the stack.
4. Resumes execution of the calling procedure.

#### **Commands:**

- To assemble

**nasm -f elf 64 prog5a.asm**

**nasm -f elf 64 prog5b.asm**

- To link

**ld prog5a.o prog5b.o**

- To execute -

**./a.out**

#### **ALGORITHM:**

##### **Algorithm to Read a file**

- 1) Call file in the program
  - 2) put a pointer to the stack
  - 3) Get file name from the stack.
  - 4) Call interrupt to open the file.
  - 5) The file descriptor is now available in eax. Test that descriptor.
  - 6) If file descriptor is not valid then close the file and exit from program.
  - 7) If valid file descriptor then copy the file contents into buffer.
  - 8) It will display the no of blank spaces in the file.
  - 9) It will display the no of lines in the file
  - 10) It will display the occurrences of character (o).
  - 11) Again test file descriptor. If it returns null then display the contents from the buffer.
  - 12) Close the file.
- 10) Exit from the program

##### **A1: Algorithm for Number of Blank spaces in the file**

- i. Start
- ii. Initialize RSI to start of text file and RDI to end of text file,
- iii. Start checking for the blank space if space is detected count will increase.
- iv. Display Number of blank spaces
- v. Exit

#### **A2: Algorithm for Number of lines in the file**

- i. Start
- ii. Initialize RSI to start of text file and RDI to end of text file,
- iii. Start checking the line if enter is detected count of line will increase
- iv. Display Number of lines
- v. Ret

#### **A3: Algorithm to count occurrences of character**

- ii. Start
- iii. Initialize RSI to start of text file and RDI to end of text file,
- iv. Start checking the occurrences of character (o) and count it,
- v. Display the count value if character (o) is present.
- vi. If character (o) is not present Display the message that character (o) is not present.
- vii. Stop

#### **Mathematical Model:**

Let  $S = \{ s, e, X, Y, Fme, mem \mid \Phi s \}$  be the programmer's perspective of String Manipulations Where

Let X be the input such that

$X = \{ X1, X2, X3, \dots \}$

Such that there exists function  $f_{X1}: X1 \longrightarrow \{0,1\}$

X2 is the Ascii Value of String Character

Let  $X2 = b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0$  where  $\forall bi \in X1$

There exists a function  $f_{X2}:X2 \longrightarrow \{ 41h,42h,-----,61h,-----\}$  i.e.  $41h$  is ASCII equivalent of A &  $61h$  is Ascii equivalent of a.

$X3$  is String1 Array

Let  $X3 = \{ \{b7-----b0\} \{b7----b0\}-----\{ b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0\} \}$  where  $\forall bi \in X1$   
 $= \{ X2_{n1}, X2_{n1-1}, ----- 0 \}$  Where  $n1 = \text{Length of String1}$

e.g  $X3 = \{ Abcd \}-----$  Entered String

then how it is stored  $\{ 41h,61h,62h,63h \}$

$X4$  is String2 Array

Let  $X4 = \{ \{b7-----b0\} \{b7----b0\}-----\{ b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0\} \}$  where  $\forall bi \in X1$   
 $= \{ X2_{n2}, X2_{n2-1}, ----- 0 \}$  Where  $n2 = \text{Length of String2}$

$X5$  is single digit choice

Let  $X4 = b3\ b2\ b1\ b0$  where  $\forall bi \in X1$

There exists a function  $f_{X5}:X5 \longrightarrow \{ 1,2,3 \}$ .

$Y1$  is concatenated String Array

Let  $Y1 = \{ \{b7-----b0\} \{b7----b0\}-----\{ b7\ b6\ b5\ b4\ b3\ b2\ b1\ b0\} \}$  where  $\forall bi \in X1$   
 $= \{ \{ X2_{n1}, X2_{n1-1}, ----- X0 \} \{ X2_{n2}, X2_{n2-1}, ----- 0 \} \}$  Where  $n1 = \text{Length of String1}$   
&  $n2 = \text{Length of String2}$

Let  $Y2 = n1$  is length of string1 which is returned by accept macro (Returned in RAX, 64 bit Register)

Let  $Y2 = b63\ b62\ b61 ----- b3\ b2\ b1\ b0$

Where  $\forall bi \in X1$

$Y2 \longrightarrow \{ 0000000000000001h, ----- FFFFFFFFFFFFFFFFFFh \}$

Let Y3= n2 is length of string2 which is returned by accept macro (Returned in RAX,64 bit Register)

Let Y3= b63 b62 b61 ----- b3 b2 b1 b0

Where  $\forall b_i \in X1$

Y3  $\longrightarrow$  { 0000000000000001h, -----FFFFFFFFFFFFFFFFh}

Y4 is the substring Count

Let Y4 = b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 where  $\forall b_i \in X1$

Y4  $\longrightarrow$  {0000H,0001H, ----- ,FFFFH}

Let Fme = { F1 ,F2,F3,F4 }

Where F1 = Accept String1 & store n1

F2 = Accept String2 & store n2

F3= Accept X5 i.e choice.

F4 = If X5=1 call Fme1

F5 = If X5=2 call Fme2

F6 = If X5=3 call Fme3

Fme1 ( concatenated String)= { F1,F2,F3}

Where F1= Copy X3 in Y1

F2= Copy X4 in Y1

F3= Display Y1

Fme2 ( Substring)= { F1,F2}

Where F1= Compare X4 with X3 for count=n2

if Equal Y4=Y4+1

F2= If not equal increment pointer to X3 &

Repeat F1 till the end of X3

F3 = Display Y4

Fme3 ( Exit)= { F1 }

Where F1= Call Sys\_exit

### Commands:

- To assemble

**nasm -f elf 64 prog5a.asm**

**nasm -f elf 64 prog5b.asm**

- To link

**ld prog5a.o prog5b.o**

- To execute -

**./a.out**

## CONCLUSION

Hence we conclude that we have find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character from a text file.

Output

Case 1: If file exist

```
[fedora@localhost ~]$ nasm -f elf64 prog5a.asm
```

```
[fedora@localhost ~]$ nasm -f elf64 prog5b.asm
```

```
[fedora@localhost ~]$ ld prog5a.o prog5b.o
```

```
[fedora@localhost ~]$./a.out
```

FILE OPENED SUCESSFULLY!

Number of Blank spaces – 5

Number of lines – 3

Occurrence of a character (o) - 7



```
[fedora42@localhost ~]$
```



Case 2: If file does not exist

```
[fedora@localhost ~]$./a.out
FIE NOT FOUND...
[fedora42@localhost ~]$
```

## OUTCOME

**ELO5:** Analyze the difference between near and far procedure to find number of lines, blank spaces & occurrence of character using nasm.  

### **Oral Question Bank**

1. What is difference between NEAR& FAR procedure?
2. List and Explain Different File System Calls?
3. Explain PUBLIC and EXTERN directives?

## **EXPERIMENT NO.13**

**Find factorial of a given integer number on a  
command line by using recursion**

## Session Plan:

| <b>Time<br/>( min)</b> | <b>Content</b>                                  | <b>Learning Aid /<br/>Methodology</b> | <b>Faculty Approach</b>              | <b>Typical<br/>Student<br/>Activity</b> | <b>Skill / Competency<br/>Developed.</b>                     |
|------------------------|-------------------------------------------------|---------------------------------------|--------------------------------------|-----------------------------------------|--------------------------------------------------------------|
| 10                     | Relevance and significance of Problem statement | Chalk & Talk ,<br>Presentation        | Introduces, Explains                 | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal                                  |
| 10                     | Explanation of Problem statement                | Chalk & Talk ,<br>Presentation        | Introduces,<br>Facilitates, Explains | Listens,<br>Participates,               | Knowledge,<br>intrapersonal,<br>Application                  |
| 20                     | Concept of Math Coprocessor of 8087             | Demonstration,<br>Presentation        | Explains, Facilitates,<br>Monitors   | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>interpersonal<br>Application |
| 60                     | Implementation of problem statement             | N/A                                   | Guides, Facilitates<br>Monitors      | Participates,<br>Discusses              | Comprehension,<br>Hands on experiment                        |
| 10                     | Assessment                                      | N/A                                   | Monitors                             | Participates,<br>Discusses              | Knowledge,<br>Application                                    |
| 10                     | Conclusions                                     | Keywords                              | Lists, Facilitates                   | Listens,<br>Participates,<br>Discusses  | Knowledge,<br>intrapersonal,<br>Comprehension                |

**TITLE:** Find factorial of a given integer number on a command line by using recursion.

**OBJECTIVES:**

- 14. To be familiar with the format of assembly language program along with different assembler directives and different functions of the DOS Interrupt.
- 15. To learn the instructions related to 80386.
- 16. To be familiar with Math Coprocessor.
- 4. Implement factorial of a integer number.

**PROBLEM DEFINITION:**

Write 80386 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.

**WORKING ENVIRONMENT:**

- 26) CPU: Intel I5 Processor
- 27) OS:- Windows XP (16 bit Execution ), Fedora 18 (32 & 64 bit Execution)
- 28) Editor: gedit, GNU Editor
- 29) Assembler: NASM (Netwide Assembler)
- 30) Linker:-LD, GNU Linker

**S/W AND H/W REQUIREMENT:**

**Software Requirements**

- 26. CPU: Intel I5 Processor
- 27. OS:- Windows XP (16 bit Execution ), Fedora 18 (32 & 64 bit Execution)
- 28. Editor: gedit, GNU Editor
- 29. Assembler: NASM (Netwide Assembler)  
Linker:-LD, GNU Linker

**INPUT:** Input data specified in program.

**OUTPUT:** Display the data present in destination block.

## **THEORY:**

A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.

Recursion could be observed in numerous mathematical algorithms. For example, consider the case of calculating the factorial of a number. Factorial of a number is given by the equation –

$$\text{Fact}(n) = n * \text{fact}(n-1) \text{ for } n > 0$$

For example: factorial of 5 is  $1 \times 2 \times 3 \times 4 \times 5 = 5 \times \text{factorial of } 4$  and this can be a good example of showing a recursive procedure. Every recursive algorithm must have an ending condition, i.e., the recursive calling of the program should be stopped when a condition is fulfilled. In the case of factorial algorithm, the end condition is reached when  $n$  is 0.

## **Instructions needed:**

1. AND-AND each bit in a byte or word with corresponding bit in another byte or word
2. INC-Increments specified byte/word by 1
3. DEC-Decrements specified byte/word by 1
4. JG - The command JG simply means: Jump if Greater.
5. CMP-Compares to specified bytes or words
6. MUL - The MUL (Multiply) instruction handles unsigned data
7. CALL-Transfers the control from calling program to procedure.
8. ADD- ADD instructions are used for performing simple addition of binary data in byte, word and doubleword size, i.e., for adding 8-bit, 16-bit or 32-bit operands, respectively.
9. RET-Return from where call is made

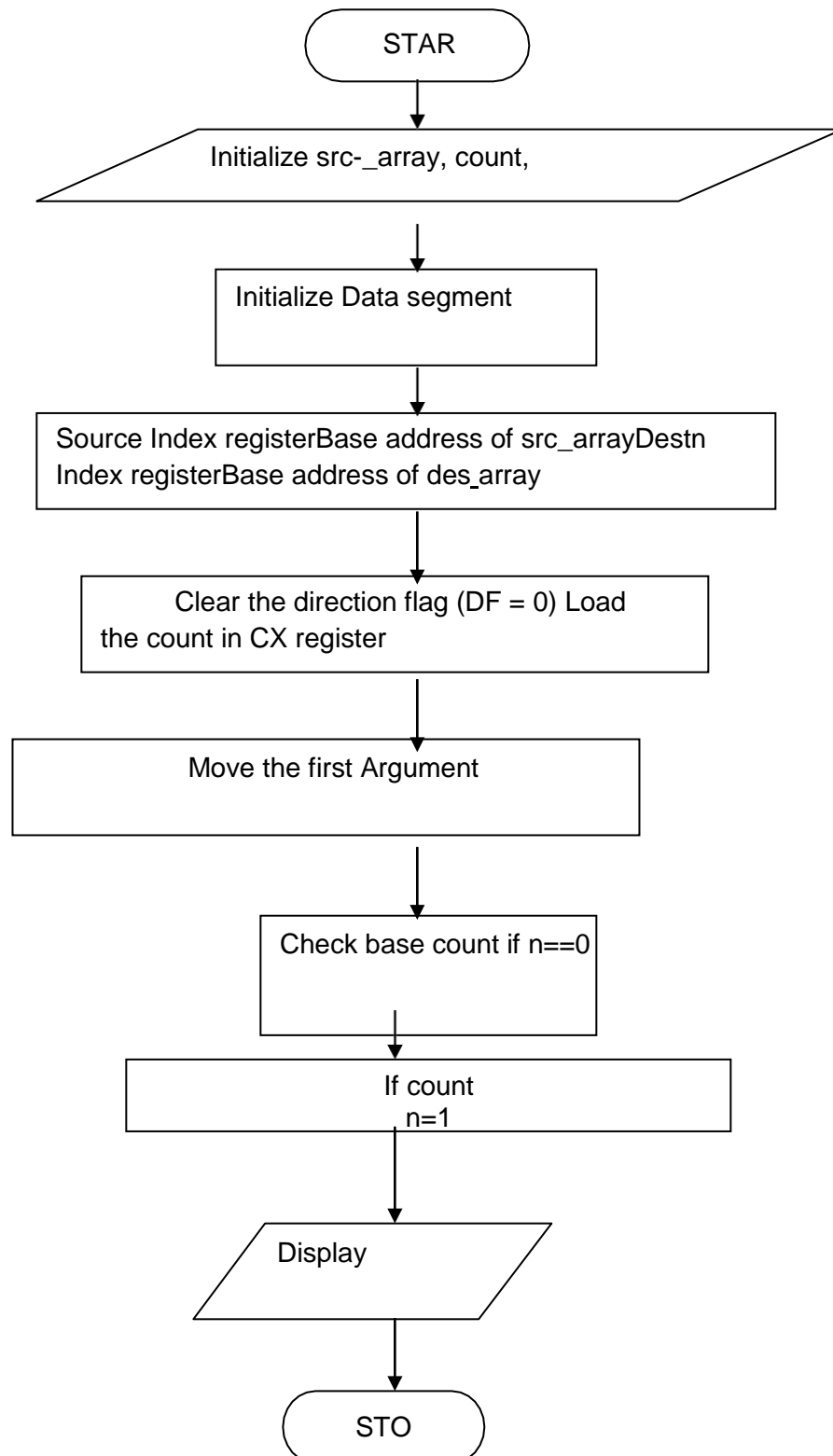
## **ALGORITHM:**

This algorithm use recursive approach to find factorial of N.

1. Start
2. Read: Take input N
3. Retrieve parameter and put it into Register-PUSH
4. Check for base case if  $n==0$
5. move the first argument to %eax
6. If the number is 1, that is our base case, and we simply return.
7. multiply by the result of the last call to factorial.
8. Restore %ebp and %esp to where they were before the function started.
9. return to the function

## FLOWCHART:

To find factorial of number



## MATHEMATICAL MODEL:

Let  $S = \{ s, e, X, Y, \mid \Phi s \}$

The factorial function is formally defined by the [product](#)

$$n! = \prod_{k=1}^n k$$

This notation works in a similar way to **summation notation** ( $\Sigma$ ), but in this case we multiply rather than add terms. For example, if  $n = 4$ , we would substitute  $k = 1$ , then  $k = 2$ , then  $k = 3$  and finally  $k = 4$  and write:

$$4! = \prod_{k=1}^4 k = 1 \times 2 \times 3 \times 4 = 24$$

In math, we often come across the following expression:

$n!$

This is "n factorial", or the product

$$n(n-1)(n-2)(n-3) \dots (3)(2)(1).$$

## Commands

- To assemble

```
nasm -f elf64 hello.nasm -o hello.o
```

- To link

```
ld -o hello hello.o
```

- To execute -

```
./hello
```



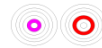
## CONCLUSION

Hence we conclude that we can perform ALP to find out factorial of a given integer number.

## OUTCOME

**Upon completion Students will be able to:**

**ELO9:** Apply the concept of recursion to find factorial of a number.



## FAQ:

### Oral Question Bank

1. What is difference between 8087 and 80386.
2. What are the Instructions sets to be used for 8086-87
3. What different conditions used to find factorial of an integer number.
4. Explain CALL, JG, ADD instructions