

# PUNE INSTITUTE OF COMPUTER TECHNOLOGY

## Department of Computer Engineering

Academic Year: 2022-23

---

### LAB MANUAL

SUBJECT: MICROPROCESSOR LAB (210257)

CLASS: S.E.

SEMESTER: IV

### ASSIGNMENT NUMBER: 1

<b>ASSIGNMENT NO.</b>	1
<b>TITLE</b>	Display the five 64 bit Hexadecimal numbers from the user
<b>PROBLEM STATEMENT / DEFINITION</b>	Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers from the user and store them in an array and display the accepted numbers.
<b>OBJECTIVE</b>	To learn the procedure how to store and display N hexadecimal numbers in memory..
<b>OUTCOME</b>	Students are able to Understand and apply various addressing modes and instruction set to implement assembly language programs.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	-Core 2 duo/i3/i5/i7 - 64 bit processor -Editor: gedit, GNU Editor - Assembler: NASM (Netwide Assembler) -Linker:-LD, GNU Linker
<b>REFERENCES</b>	1. “Microprocessor and Interfacing Techniques”, Douglas Hall 2. “IBM PC Assembly language and programming”, Peter Able 3. “Advances MS-DOS programming ”, Ray Duncan

<b>STEPS</b>	<ol style="list-style-type: none"> <li>1.Start</li> <li>2. Declare &amp; initialize the variables in the .data section.</li> <li>3. Declare uninitialized variables in the .bss section.</li> <li>4. print message1 accepts the five 64-bit numbers</li> <li>5. Accept five 64-bit numbers</li> <li>6.Initialize pointer with source address of array.</li> <li>7. Initialize count for number of elements.</li> <li>8. Put the complete summed rbx value to arr[n]</li> <li>9. Decrement counter with conditional instruction JNZ</li> <li>10..Max accept of 16 decimal number with enter value and rsi points to accept[17]</li> <li>11. print message2 display the five 64-bit numbers</li> <li>12.Repeat steps no.s from 5 to 9 to display the five 64-bits numbers</li> <li>13.Terminate the process.</li> <li>14.Stop.</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>

### **Prerequisites: COA**

### **Concepts related Theory:**

The Assembler is a system software that converts an assembly language code to machine code. It takes basic Computer commands and converts them into Binary Code that Computer's Processor can use to perform its Basic Operations.

These instructions are assembler language or assembly language.Assembly language uses opcode for the

instructions.

An opcode basically gives information about the particular instruction. The symbolic representation of the opcode (machine level instruction) is called mnemonics. Programmers use them to remember the operations in assembly language.

### **For example ADD A,B**

Here, ADD is the mnemonic that tells the processor that it has to perform additional functions. Moreover, A and B are the operands. Also, SUB, MUL, DIVC, etc. are other mnemonics.

. **There are many good assembler programs, such as –**

- Microsoft Assembler (MASM)
- Borland Turbo Assembler (TASM)
- The GNU assembler (GAS)

Assembly language is dependent upon the instruction set and the architecture of the processor. We focus on Intel-32 processors like Pentium. minimum requirements are:

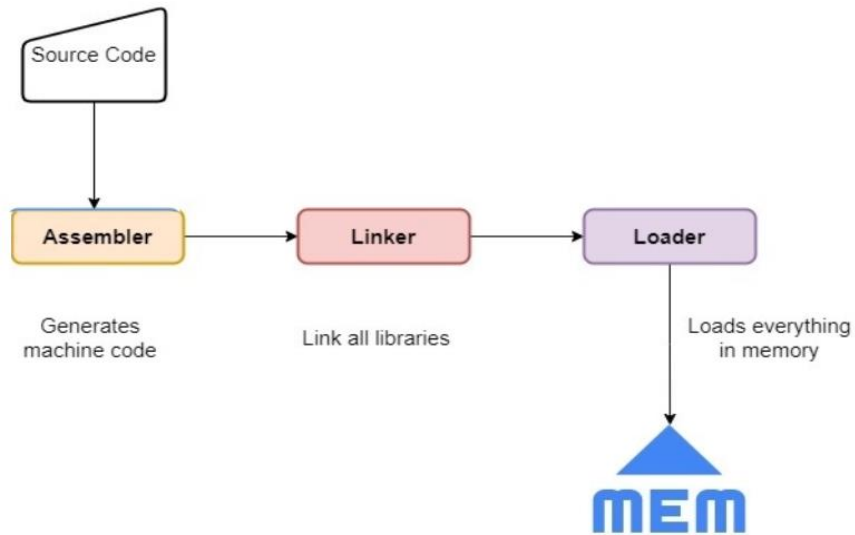
- An IBM PC or any equivalent compatible computer
- A copy of Linux operating system
- A copy of NASM assembler program

### **The NASM assembler, as it is –**

- The Netwide Assembler, NASM, is an 80x86 and x86-64 assembler designed for portability and modularity.
- It supports a range of object file formats, including :Linux and \*BSD a.out, ELF (Executable and Linkable Format) , Mach-O, 16-bit and 32-bit ,.obj (OMF),binary format.
- COFF (Common Object File Format)is a format for executable, object code, and shared library (including its Win32 and Win64 variants.)
- Open Source Software
- NASM Is Case-Sensitive
- NASM Requires Square Brackets For Memory References
- It can also output plain binary files, Intel hex and Motorola'S-Record formats.
- It supports all currently known x86 architectural extensions, and has strong support for macros.

### **Role of assembler,loader & linker:**

1. **Assembler:**It translates assembly code to the machine code and then stores the result in an object file. This file contains the binary representation of our program.
2. **Linker:**The linker takes all object files as input, resolves all memory references, and finally merges these object files to make an executable file.
3. **Loader:**It loads the final executable code into memory.



## Basic syntax of nasm program

### 1.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 ;define constants
```

### 2.bss section:

- The bss(block started by symbol) section is used for declaring variables.
- Contain object file where statically allocated variables are stored
- The syntax for declaring bss section is –

```
section .bss ;define all reserved variable here
```

### 3.text section

- It is used for keeping 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 ;must be declared for linker (ld)
```

```
_start: ;start actual Program coding or tells linker entry point
```

## Syntax of assembly language statement

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

- \* Typically one statement per line
- \* Fields in [ ] are optional
- \* **label** serves two distinct purposes:
  - » To label an instruction
    - Can transfer program execution to the labeled instruction
  - » To label an identifier or constant
- \* **mnemonic** identifies the operation (e.g., **add**, **or**)
- \* **operands** specify the data required by the operation
  - » Executable instructions can have zero to three operands

### For Example:



## 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:

1. Put the system call number in the EAX register.
2. Store the arguments to the system call in the registers EBX, ECX, etc.
3. Call the relevant interrupt (80h)
4. 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.

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories –

- General registers,
- Control registers, and
- Segment registers.

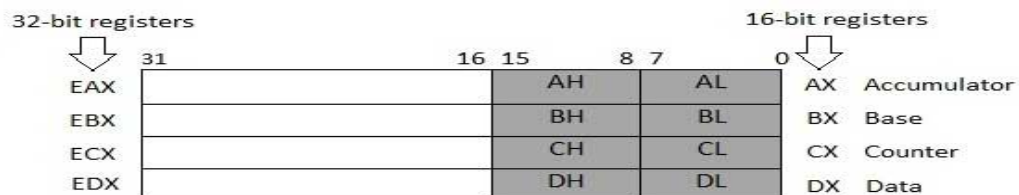
### The general registers are further divided into the following groups –

1. Data registers,
2. Pointer registers, and
3. Index registers.

#### 1. Data registers

Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways –

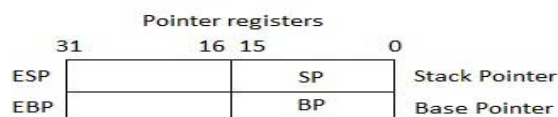
- ▣ As complete 32-bit data registers: EAX, EBX, ECX, EDX.
- ▣ Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
- ▣ Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.



#### 2. Pointer registers

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers –

- ▣ **Instruction Pointer (IP)** – The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- ▣ **Stack Pointer (SP)** – The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- ▣ **Base Pointer (BP)** – The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

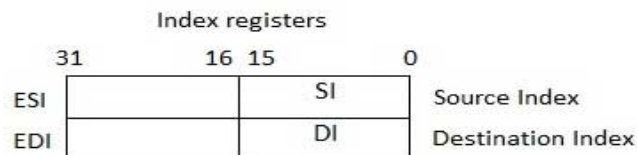


### 3. Index registers

#### Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers –

- **Source Index (SI)** – It is used as source index for string operations.
- **Destination Index (DI)** – It is used as destination index for string operations.



#### 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.

#### Instructions Needed:

1. **EQU:** The **equate directive is used to substitute values for symbols or labels**. The format is 'label: EQU value', so whenever the assembler encounters 'label', it replaces this with 'value'.
2. **MOV:** copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory).
3. **CALL:** The **call instruction calls** near procedures using a full pointer. **call** causes the procedure named in the operand to be executed.
4. **JNZ:** The Jump if not equal to zero **transfers control to the specified address if the value in the accumulator is not 0**. If the accumulator has a value of 0, the next instruction is executed. Neither the accumulator nor any flags are modified by this instruction.
5. **JNC:** The JNC instruction **transfers program control to the specified address if the carry flag is 0**. Otherwise, execution continues with the next instruction. No flags are affected by this instruction.

#### Algorithm

- 1 Start
2. Declare & initialize the variables in the .data section.

3. Declare uninitialized variables in the .bss section.
4. print message1 accepts the five 64-bit numbers
5. Accept five 64-bit numbers
6. Initialize pointer with source address of array.
7. Initialize count for number of elements.
8. Put the complete summed rbx value to arr[n]
9. Decrement counter with conditional instruction JNZ
- 10..Max accept of 16 decimal number with enter value and rsi points to accept[17]
11. print message2 display the five 64-bit numbers
- 12.Repeat steps no.s from 5 to 9 to display the five 64-bits numbers
- 13.Terminate the process.
- 14.Stop.

**Output:** The output shows us Five 64-bits of numbers displayed.

**CONCLUSION:** Here we have accepted store them in an array and display the accepted numbers

### **Review Questions:**

1. What is an assembler? compare assembler and compiler?
2. What are functions of assembler, linker and loader in assembly programming ?
3. Explain the Nasm with its features?
4. With the syntax statement representation explain the instruction of assembly language?
5. Explain segmentation in 80386? role of sections: data, bss, text
6. With the examples explain assembly system calls?
7. What are general purpose registers?
8. Explain various instructions used in assignment no 1 program?



## ASSIGNMENT NUMBER: 2

<b>ASSINGMENT NO.</b>	2
<b>TITLE</b>	To display length of the String
<b>PROBLEM STATEMENT /DEFINITION</b>	Write an X86/64 ALP to accept a string and to display its length.
<b>OBJECTIVE</b>	To learn and understand the operation of accept and display string
<b>OUTCOME</b>	On completion of this practical, students will be able to calculate length of string without using inbuilt function and HEX to ASCII conversion
<b>S/W PACKAGES AND HARDWARE APPARATUSUSED</b>	Core 2 duo/i3/i5/i7 - 64bit processorOS – Linux 64bit OS Editor-any Text Editor Assembler – NASM Debugger- GDB/TD
<b>REFERENCES</b>	1. “Microprocessor and Interfacing Techniques”, DouglasHall 2. “IBM PC Assembly language and programming”, PeterAble 3. “Advances MS-DOS programming”, Ray Duncan

<b>STEPS</b>	<ol style="list-style-type: none"> <li>1. START</li> <li>2. Get string from user stored in variable</li> <li>3. Check string character by character till end of the string</li> <li>4. Stored the counter in a cx register</li> <li>5. Then convert that number in to ASCII number to display</li> <li>6. Display length of string</li> <li>7. STOP</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>

Prerequisites: DEL

### **Theory:**

All members of the 80 x 86 families support five different string instructions: MOVSB, CMPSB, SCASB, LODSB and STOSB. They are the string primitives since you can build most other string operations from these five instructions.

**How the String Instructions Operate:** The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVSB instruction moves a sequence of bytes from one memory location to another. The CMPSB instruction compares two blocks of memory. The SCASB instruction scans a block of memory for a particular value. These string instructions often require three operands a destination block address a source block address and (optionally) an element count. For example, when using the MOVSB instruction to copy a string you need a source address a destination address and a count (the number of string elements to move).

Unlike other instructions which operate on memory the string instructions are single-byte instructions which don't have any explicit operands. The operands for the string instructions include

- The SI (source index) register
- The DI (destination index) register
- The CX (count) register
- The AX register and
- The direction flag in the FLAGS register.

For example one variant of the MOVS (move string) instruction copies a string from the source address specified by DS:SI to the destination address specified by ES:DI of length CX. Likewise, the CMPS instruction compares the string pointed at by DS:SI of length CX to the string pointed at by ES: DI.

Not all instructions have source and destination operands (only MOVS and CMPS support them). For example, the SCAS instruction (scan a string) compares the value in the accumulator to values in memory. Despite their differences the 80x86's string instructions all have one thing in common - using them requires that you deal with two segments the data segment and the extra segment.

### **Flag**

Besides the SI, DI and ax registers one other register controls the 80x86's string instructions - the flags register. Specifically, the direction flag in the flags register controls how the CPU processes strings.

If the direction flag is clear the CPU increments SI and DI after operating upon each string element. For example if the direction flag is clear then executing MOVS will move the byte word or double word at DS:SI to ES:DI and will increment SI and DI by one two or four. When specifying the REP prefix before this instruction the CPU increments SI and DI for each element in the string. At completion the SI and DI registers will be pointing at the first item beyond the string.

If the direction flag is set, then the 80x86 decrements si and di after processing each string element. After a repeated string operation, the si and di registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the cld (clear direction flag) and std (set direction flag) instructions. When using these instructions inside a procedure keep in mind that they modify the machine state. Therefore you may need to save the direction flag during the execution of that procedure.

### **ALGORITHM**

STEP1: Get string from user by write system call

STEP2: Decrement EAX register by 1 stored in RBX (length is stored in EAX register after write system call)

STEP3: point rdi to variable which is holding the length of string

STEP4: set count as 16

STEP5: rotate rbx by 4

STEP6: move content of BL to AL and ANDING with 0FH check AL >9

    If yes: ADD 37H

    No: ADD 30 H

STEP7: Store converted data into output variable

STEP8: increment RDI

STEP9: decrement counter

STEP10: repeat STEPS 5 to 9 till counter 0

**Output :** The output shows us actual length of string (64 bit) without using inbuilt string Function.

**Conclusion:** In this way we studied about how to calculate length of string without inbuilt function and display length of string using HEX to ASCII conversion strings using 80386 microprocessors

**Review Questions (write-ups)**

1. What is the use of addressing mode? And Explain its type.
2. Explain File Descriptor of Assembly program?
3. Explain with example string instructions of 80386 Microprocessor.
4. What is the maximum size of the instruction in 80386?
5. Which are string instructions?
6. In string operations which is by default a string source pointer?
7. In string operations which is by default a string destination pointer?
8. Write Code of Hex to Ascii Procedure?

### ASSIGNMENT NUMBER: 3

<b>ASSINGMENT NO.</b>	2
<b>TITLE</b>	To display greatest number from given array element
<b>PROBLEM STATEMENT /DEFINITION</b>	Write an X86/64 ALP to find the largest of the given BYTE/WORD/DWORD/64 bit numbers
<b>OBJECTIVE</b>	To learn and understand the operation of accept and display string
<b>OUTCOME</b>	On completion of this practical, students will be able to find largest number from the given array by compare instruction
<b>S/W PACKAGES AND HARDWARE APPARATUSUSED</b>	Core 2 duo/i3/i5/i7 -  64bit processorOS –  Linux 64bit OS  Editor-any  Text Editor  Assembler –  NASM  Debugger- GDB/TD
<b>REFERENCES</b>	<ol style="list-style-type: none"> <li>1. “Microprocessor and Interfacing Techniques”, DouglasHall</li> <li>2. “IBM PC Assembly language and programming”, PeterAble</li> <li>3. “Advances MS-DOS programming”, Ray Duncan</li> </ol>
<b>STEPS</b>	<ol style="list-style-type: none"> <li>1. START</li> <li>2. Get 5 elements of array from user</li> <li>3. Find largest number and stored in AL</li> <li>4. Then convert that number in to ASCII number to display</li> <li>5. Display largest number</li> <li>6. STOP</li> </ol>

<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>9. 10. Conclusion/Analysis</li> </ol>
---	--

Prerequisites: DEL

### **Theory:**

All members of the 80 x 86 families support five different string instructions: MOVSB, CMPSB, SCASB, LODSB and STOSB. They are the string primitives since you can build most other string operations from these five instructions.

**How the String Instructions Operate:** The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVSB instruction moves a sequence of bytes from one memory location to another. The CMPSB instruction compares two blocks of memory. The SCASB instruction scans a block of memory for a particular value. These string instructions often require three operands a destination block address a source block address and (optionally) an element count. For example, when using the MOVSB instruction to copy a string you need a source address a destination address and a count (the number of string elements to move).

Unlike other instructions which operate on memory the string instructions are single-byte instructions which don't have any explicit operands. The operands for the string instructions include

- The SI (source index) register
- The DI (destination index) register
- The CX (count) register
- The AX register and
- The direction flag in the FLAGS register.

For example one variant of the MOVSB (move string) instruction copies a string from the source address specified by DS:SI to the destination address specified by ES:DI of length CX. Likewise, the CMPSB instruction compares the string pointed at by DS:SI of length CX to the string pointed at by ES: DI.

Not all instructions have source and destination operands (only MOVSB and CMPSB support them). For example, the SCASB instruction (scan a string) compares the value in the accumulator to values in memory. Despite their differences the 80x86's string instructions all have one thing in common - using them requires that you deal with two segments the data segment and the extra segment.

### **Flag**

Besides the SI, DI and ax registers one other register controls the 80x86's string instructions - the flags register. Specifically, the direction flag in the flags register controls how the CPU processes strings.

If the direction flag is clear the CPU increments SI and DI after operating upon each string element. For example if the direction flag is clear then executing MOVS will move the byte word or double word at DS:SI to ES:DI and will increment SI and DI by one two or four. When specifying the REP prefix before this instruction the CPU increments SI and DI for each element in the string. At completion the SI and DI registers will be pointing at the first item beyond the string.

If the direction flag is set, then the 80x86 decrements si and di after processing each string element. After a repeated string operation, the si and di registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the cld (clear direction flag) and std (set direction flag) instructions. When using these instructions inside a procedure keep in mind that they modify the machine state. Therefore you may need to save the direction flag during the execution of that procedure.

### **ALGORITHM**

STEP1: Get array content from user and print

STEP2: set counter =5 ( array stored 5 elements)

STEP3: point RDI to array

STEP4: move 0 as maximum number in al register

STEP5: compare AL with next array element

STEP6: if AL content largest number

    If yes: got to STEP 7

    No: move array element in to AL got to STEP 7

STEP7: Increment RDI

STEP8: Decrement COUNTER

STEP9: repeate STEPS 5 to 9 till counter 0

STEP10: Display largest number which is in AL register using HEX to ASCII conversion

**Output :** The output shows us largest 64bit number from given array.

**Conclusion:** In this way we studied about compare instruction using 80386 microprocessors

### **Review Questions (write-ups)**

1. How to stored offset of array?
2. What is use of compare instruction?

3. What is use of counter

**ASSIGNMENT NUMBER: 4**

<b>ASSINGMENT NO.</b>	4
<b>TITLE</b>	Menu driven basic arithmetic operations
<b>PROBLEM STATEMENT /DEFINITION</b>	Write a switch case driven X86/64 ALP to perform 64-bit hexadecimal arithmetic operations (+,-,*, /) using suitable macros. Define procedure for each operation
<b>OBJECTIVE</b>	<ol style="list-style-type: none"><li>1. To learn the procedures in assembly language programing</li><li>2. To learn code conversion(ascii to Hexa decimal)</li><li>3. To perform arithmetic operations using menu driven</li></ol>
<b>OUTCOME</b>	Students will be able to do menu driven arithmetic operations
<b>S/W PACKAGES AND HARDWARE APPARATUSUSED</b>	Core 2 duo/i3/i5/i7 - 64bit  processorOS – Linux 64bit OS  Editor- gedit /vi  Assembler –NASM  Debugger- GDB/TD
<b>REFERENCES</b>	<ol style="list-style-type: none"><li>1. “Microprocessor and Interfacing Techniques”, Douglas Hall</li><li>2. “IBM PC Assembly language and programming”, Peter Able</li><li>3. INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986</li></ol>



<b>STEPS</b>	<ol style="list-style-type: none"> <li>1. Display menu of 4 basic arithmetic operations</li> <li>2. Accept choice from user</li> <li>3. Convert ascii value to hexa-decimal value</li> <li>4. Based on choice perform the arithmetic operation</li> <li>5. Convert the result into ascii code for displaying the result</li> <li>6. Print the result on the screen and stop</li> </ol>
--------------	--

<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>
---	---

### **Prerequisites'**

### **Concepts related Theory:**

### **Procedure and Call**

#### **Procedure**

A called procedure (or subroutine) is a section of code that perform a clearly defined task. A code block may contain any number of procedures.

Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size. Procedures are identified by a name.

Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.

#### **Syntax**

Following is the syntax to define a procedure –

Proc\_name :

Procedure body

...

ret

The procedure is called from another function by using the CALL instruction. The CALL instruction should have the name of the called procedure as an argument as shown below –

CALL proc\_name

The called procedure returns the control to the calling procedure by using the RET instruction.

### **Call and ret**

The CALL instruction is to transfer control to a called procedure (subroutine).

The RET instruction is to return from the called procedure (subroutine) to the original calling procedure. It effects the counterpart of CALL.

Assembly Language allows us to perform basic unsigned integer arithmetic operations. These operations are:

1. Addition (add)
2. Subtraction (sub)
3. Multiplication (mul)
4. Division (div)

## **1. Arithmetic Instructions**

### **Addition**

ADD destination, source

ADD (Add Integers) replaces the destination operand with the sum of the source and destination operands. Sets CF if overflow.

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF

### **Subtraction**

P:F-LTL-UG/03/R1

SUB destination, source

SUB (Subtract Integers) subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF is set. The operands may be signed or unsigned bytes, words, or doublewords.

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

Flags Affected

OF, SF, ZF, AF, PF, and CF

### Multiplication

The 80386 has separate multiply instructions for unsigned and signed operands. MUL operates on unsigned numbers, while IMUL operates on signed integers as well as unsigned.

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX:AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1.

Flags Affected

OF and CF

### Division

The 80386 has separate division instructions for unsigned and signed operands. DIV operates on unsigned numbers, while IDIV operates on signed integers as well as unsigned. In either case, an exception (interrupt zero) occurs if the divisor is zero or if the quotient is too large for AL, AX, or EAX.

DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use:

Size	Dividend	Divisor	Quotient	Remainder
byte	AX	r/m8	AL	AH
word	DX:AX	r/m16	AX	DX

dword      EDX:EAX      r/m32      EAX      EDX

Flags Affected  
OF, SF, ZF, AR, PF, CF

### **Algorithm:**

1. Define the macro for input and output statements
2. Begin program and display menu in front of user
3. Read the choice of arithmetic operation from the user
4. Check the value of choice and accordingly perform the respective operation using procedure call
5. Convert the result into ascii value to display the result on the screen

### **Conclusion:**

Hence we performed switch case driven arithmetic operations using procedure calls.

### **Review Questions:**

1. Define procedure? Write syntax
2. What is the use of procedure in assembly language?
3. How the procedure is called from the main program in microprocessor?
4. How does a near procedure differs from far procedure call
5. What are call and ret instructions? Explain with code
6. How procedures are invoked
7. Where is the control passed after ret instruction
8. Explain instructions with example(add,mul,div,sub)
9. What is the difference between mul and imul

### ASSIGNMENT NUMBER: 5

<b>ASSINGMENT NO.</b>	1
<b>TITLE</b>	To find positive and negative numbers.
<b>PROBLEM STATEMENT /DEFINITION</b>	Write X86/64 ALP to count number of positive and negative numbers from the array
<b>OBJECTIVE</b>	To understand how to identify a positive number or negative number.
<b>OUTCOME</b>	Students will be able to use different index registers and find positive and negative numbers from array.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	Core 2 duo/i3/i5/i7 - 64bit processor  OS – Linux 64bit OS  Editor- gedit /vi  Assembler –NASM  Debugger- GDB/TD
<b>REFERENCES</b>	4. “Microprocessor and Interfacing Techniques”, Douglas Hall  5. “IBM PC Assembly language and programming”, Peter Able  6. “Advances MS-DOS programming ”, Ray Duncan

<b>STEPS</b>	<p>Start.</p> <ol style="list-style-type: none"> <li>Initialize an array of 10 numbers or accept 10 numbers from user and store them in one array.</li> <li>Initialize pos_counter=0, neg_counter=0, index_reg=array address, counter=10</li> <li>Read the number from index_reg into a register.</li> <li>Perform addition with 00H and check sign bit</li> <li>If sign bit==1 then increment neg_counter=neg_counter+1 else increment pos_counter=pos_counter+1 end if</li> <li>Increment index_reg= index_reg+1</li> <li>Decrement counter=counter-1</li> <li>If counter!=0 then goto step number 4 else continue</li> <li>Print message “Positive numbers are:” and print pos_counter.</li> <li>Print message “Negative numbers are:” and print neg_counter.</li> <li>Exit.</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>Date</li> <li>Assignment no.</li> <li>Problem definition</li> <li>Learning objective</li> <li>Learning Outcome</li> <li>Concepts related Theory</li> <li>Algorithm</li> <li>Test cases</li> <li>Conclusion/Analysis</li> </ol>

**Prerequisites:** COA

**Concepts related Theory:**

Any number above zero is a positive number. Positive numbers are written with no sign or a + sign in front of them and they are counted from zero to the right. Any number **below zero** is a negative number. Negative numbers are written with a - sign in front of them and they are counted from zero to the left. Always look at the **sign** in front of a number to see if it is positive or negative.

In Computer Systems, the negative number is represented by different ways:

### 1. Sign-Magnitude representation: -

Negative numbers are essential and any computer not capable of dealing with them would not be particularly useful. But how can such numbers be represented? There are several methods which can be used to represent negative numbers in Binary. One of them is called the Sign-Magnitude Method.

The Sign-Magnitude Method is quite easy to understand. In fact, it is simply an ordinary binary number with one extra digit placed in front to represent the sign. If this extra digit is a '1', it means that the rest of the digits represent a negative number. However if the same set of digits are used but the extra digit is a '0', it means that the number is a positive one. The following examples explain the Sign-Magnitude method better. Let us assume that we have an 8-bit register. This means that we have 7 bits which represent a number and the other bit to represent the sign of the number (the **Sign Bit**). This is how numbers are represented:

0	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

The red digit means that the number is positive. The rest of the digits represent 37. Thus, the above number in sign-magnitude representation means +37.

And this is how -37 is represented:

1	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

Binary Value

0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-0
1001	-1
1010	-2
1011	-3
1100	-4
1101	-5
1110	-6
1111	-7

Assign the leftmost (most significant) bit to be the sign bit. If the sign bit is 0, this means the number is positive. If the sign bit is 1, then the number is negative. The remaining  $m-1$  bits are used to represent the magnitude of the binary number in the unsigned binary notation.

## 2. 1's Complement representation: -

The 1's complement of a binary number is the number that results when we change all 1s to zeros and the zeros to ones.

## 3. 2's Complement representation: -

Another common method of representing negative numbers in binary is the Twos Complement. The 2's complement is the binary number that results when we add 1 to the 1's complement. It is given as

$$2\text{'s complement} = 1\text{'s complement} + 1$$

e.g. 2's complement of  $(11000100)_2$

$$1\text{'s complement of number} = 00111011$$

$$\begin{array}{r} \text{Add } 1 \\ \hline \end{array} = 1$$

$$2\text{'s complement of number} = 00111100$$

A complete binary table for four bits is shown below:

Binary	Two's Comp Value	Binary	Two's Comp Value
0000	0	1111	-1
0001	+1	1110	-2
0010	+2	1101	-3
0011	+3	1100	-4
0100	+4	1011	-5
0101	+5	1010	-6
0110	+6	1001	-7
0111	+7	1000	-8



- **Advantages of 2's complement representation: -**

1. There are distinct +0 and -0 representations in both the sign-magnitude and 1's complement systems, but the 2's complement system has only a +0 representation.
2. For 4 bit numbers, the value -8 is represent able only in the 2's complement system and not in other systems.
3. It is more efficient for logic circuit implementation, and one often used in computers for addition and subtraction operations.

## Bit Test Instruction

Opcode	Mnemonic	Description
0F B3	BTR r/m16, r16	Store selected bit in CF flag and clear
0F B3	BTR r/m32, r32	Store selected bit in CF flag and clear
0F BA /6 ib	BTR r/m16, imm8	Store selected bit in CF flag and clear
0F BA /6 ib	BTR r/m32, imm8	Store selected bit in CF flag and clear

Description
<p>Selects the bit in a bit string (specified with the first operand, called the bit base) at the bitposition designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value. If the bit base operand specifies a register, the instruction takes the modulo 16 or 32 (depending on the register size) of the bit offset operand, allowing any bit position to be selected in a 16- or 32-bit register, respectively. If the bit base operand specifies a memory location, it represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The offset operand then selects a bit position within the range <math>-2^{31}</math> to <math>2^{31} - 1</math> for a register offset and 0 to 31 for an immediate offset.</p> <p>Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT-Bit Test" in this chapter for more information on this addressing mechanism.</p> <p>This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.</p>

## Algorithm:

1. Start.
2. Initialize an array of 10 numbers or accept 10 numbers from user and store them in one array.
3. Initialize pos\_counter=0, neg\_counter=0, index\_reg=array address, counter=10
4. Read the number from index\_reg into a register.
5. Perform addition with 00H and check sign bit
6. If sign bit==1 then  
     increment neg\_counter=neg\_counter+1  
     else  
     increment pos\_counter=pos\_counter+1  
     end if
7. Increment index\_reg= index\_reg+1
8. Decrement counter=counter-1
9. If counter!=0 then goto step number 4 else

continue

10. Print message “Positive numbers are:” and print pos\_counter.
11. Print message “Negative numbers are:” and print neg\_counter.
12. Exit.

### **Conclusion:**

We are able to use different index registers and find positive and negative numbers from array.

### **Review Questions:**

1. Defined various signed data types in 80386.
2. Explain BT and BTR instruction with example.
3. Explain basic structure of ALP.
4. Write '-93' in sign-magnitude representation, using an 8-bit register.
5. What is the largest positive number which can be represented in an 8-bit register, using the sign-magnitude method of representation?
6. What is the largest negative number which can be represented in an 8-bit register, using the sign-magnitude method of representation?
7. Give the range of possible numbers which can be represented in a 16-bit register, using the sign-magnitude method.

## ASSIGNMENT NUMBER:8

<b>ASSINGMENT NO.</b>	8
<b>TITLE</b>	Data block Transfer
<b>PROBLEM STATEMENT /DEFINITION</b>	Write x86 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.
<b>OBJECTIVE</b>	To learn <ul style="list-style-type: none"><li>• Overlapped / Non – overlapped data transfer in segments</li><li>• Block transfer instruction of 8086</li><li>• Data storage in the memory and segments</li></ul>
<b>OUTCOME</b>	Students will study different block transfer instructions and also understood block transfer within different segments.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	Core 2 duo/i3/i5/i7 - 64bit processor OS – Linux 64bit OS Editor- gedit /vi Assembler –NASM Debugger- GDB/TD
<b>REFERENCES</b>	1. “Microprocessor and Interfacing Techniques”, Douglas Hall 2. “IBM PC Assembly language and programming”, Peter Able 3. “Advances MS-DOS programming ”, Ray Duncan
<b>STEPS</b>	<b>A] Overlapping</b> <ol style="list-style-type: none"><li>1. Study system call to read and display character on the screen.</li><li>2. Accept the Value of „N“ i.e. how many numbers to add</li><li>3. initialize Sum =0</li><li>4. Read a number (two digit)</li><li>5. add it to sum</li></ol>

	<ol style="list-style-type: none"> <li>6. repeat the steps 4 and 5 to add all N numbers</li> <li>7. Print the result /Sum</li> <li>8. End.</li> </ol> <p style="text-align: center;"><b>B] Non Overlapping</b></p> <ol style="list-style-type: none"> <li>1. Declare a source array.</li> <li>2. Load the address of source array in one of the registers. (Index register)</li> <li>3. Read the first byte from the source array.</li> <li>4. Increment the pointer/SI by length of array which becomes the starting Address of destination array.</li> <li>5. Move the source element to the destination address.</li> <li>6. In case of overlapping mode based on the destination address either move the first Element or last element in beginning of transfer operation.</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>

**Prerequisites:** Instruction set of 80386

**Concepts related Theory:** One of the frequent operations used in programming is shifting/

transferring the data from one memory location to another memory location. These operations can be with simple mov instructions which may result in more number of operations. We can make use of instructions like MOVSB to transfer the data. The relevant instructions are LOOP / MOVSB.

The data can be transferred either in overlapped fashion or non overlapped fashion. In case of overlapped address the two possible situations are for the Source address to be greater than the destination address in which case the first element in the source is to be moved first or for the source address to be less than the destination address which requires the last element of the source to be moved first.

#### **Algorithm: A] Overlapping**

1. Study system call to read and display character on the screen.
2. Accept the Value of „N“ i.e. how many numbers to add
3. initialize Sum =0
4. Read a number (two digit)
5. add it to sum
6. repeat the steps 4 and 5 to add all N numbers
7. Print the result /Sum
8. End.

#### **B] Non Overlapping**

1. Declare a source array.
2. Load the address of source array in one of the registers. (Index register)
3. Read the first byte from the source array.
4. Increment the pointer/SI by length of array which becomes the starting Address of destination array.
5. Move the source element to the destination address.
6. In case of overlapping mode based on the destination address either move the first Element or last element in beginning of transfer operation.

**Conclusion:**

We have studied different block transfer instructions and also understood block transfer within different segments.

**Review Questions:**

1. Explain various block transfer instruction with examples.
2. Explain use of index registers in ALP.
3. Explain different string instructions in 80386.
4. What are the different addressing modes?
5. Explain the addressing mode with suitable example.

## ASSIGNMENT NUMBER: 6

<b>ASSINGMENT NO.</b>	6
<b>TITLE</b>	Number Conversion
<b>PROBLEM STATEMENT /DEFINITION</b>	<p>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:</p> <p>a) HEX to BCD            b) BCD to HEX            c) EXIT</p>
<b>OBJECTIVE</b>	<p>To learn</p> <ul style="list-style-type: none"><li>• Data representation and conversion</li><li>• Understand the stack operations</li></ul>
<b>OUTCOME</b>	Students will studuy use of stack operations and number conversion in ALP.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<p><b>Processor:</b> Core 2 duo/i3/i5/i7</p> <p><b>OS:</b> Linux 32bit/64bit OS</p> <p><b>Editor:</b> gedit/vi</p> <p><b>Assembler:</b> NASM</p> <p><b>Debugger:</b> GDB</p>
<b>REFERENCES</b>	<ol style="list-style-type: none"><li>1. Douglas Hall, "Microprocessors &amp; Interfacing", McGraw Hill, Revised 2nd Edition, 2006 ISBN 0-07-100462-9</li><li>2. A.Ray, K.Bhurchandi "Advanced Microprocessors and peripherals: Arch, Programming &amp; Interfacing"</li></ol>
<b>STEPS</b>	<ol style="list-style-type: none"><li>1. Start.</li><li>2. Take i/p as hex to bcd or bcd to hex.</li><li>3. Convert input to ASCII.</li><li>4. Convert i/p to hex or bcd as required using function.</li><li>5. Convert answer to ASCII to display</li></ol>

	6. Display answer 7. Exit
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 10. Conclusion/Analysis

**Prerequisites: COA, DEL**

**Concepts related Theory:**

The different numbers systems can be used with the computer such as Hexadecimal, Octal, Binary and BCD. The number can be converted to any number system from any source other number system.

For example, in applications such as frequency counters, digital voltmeters or calculator where the output is a decimal display, a Binary Coded Decimal (BCD) is often used. BCD uses bit binary code to individually represent each decimal digit in a number.

Decimal

5      2      9

BCD      0101   0010   1001



### Example:

HEX number: 1234

0A) 1234(1D2

1234

-----

0

Push (0)

0A) 1D2 (2E

1CC

-----

6

Push (6)

0A) 24(3

1E

----

6

Push (6)

0A) 4(0

0

--

4

Push (4)

4
6
6
0

Pop from stack and display result = 4660

### BCD to HEX

- Initialize sum = 0
- Accept the first digit multiplicand
- Multiply the number by 10,000 multiplier
- Add result to sum
- Divide the multiplier by 10
- Repeat the step 2 to 4 till multiplier becomes zero
- Display the sum which is result of converting BCD number to HEX

Example

BCD number =4660

$$4*1000 + 6*100 + 6*10 + 6*0 = 1234H$$

$$FA0 + 258 + 3C + 0 = 1234H$$

**Algorithm:**

1. Start.
2. Take i/p as hex to bcd or bcd to hex.
3. Convert input to ASCII.
4. Convert i/p to hex or bcd as required using function.
5. Convert answer to ASCII to display
6. Display answer
7. Exit

**Conclusion:**

We have studied use of stack operations and number conversion in ALP.

**Review Questions:**

1. What is ASCII format?
2. Explain Hex to BCD and BCD to Hex conversion with example.
3. Explain MUL/IMUL instruction
4. Explain DIV/IDIV instruction

## ASSIGNMENT NUMBER:

<b>ASSINGMENT NO.</b>	
<b>TITLE</b>	Multiplication of two digit Numbers.
<b>PROBLEM STATEMENT /DEFINITION</b>	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)
<b>OBJECTIVE</b>	To understand following multiplication techniques in ALP.  1. Successive Addition  2. Add and Shift Method.
<b>OUTCOME</b>	Students will be able to do multiplication in ALP.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	Core 2 duo/i3/i5/i7 - 64bit processor  OS – Linux 64bit OS  Editor- gedit /vi  Assembler –NASM  Debugger- GDB/TD
<b>REFERENCES</b>	4. “Microprocessor and Interfacing Techniques”, Douglas Hall  5. “IBM PC Assembly language and programming”, Peter Able  6. “Advances MS-DOS programming ”, Ray Duncan
<b>STEPS</b>	Successive Addition: 1. Start 2. Accept two 2-digit numbers.(Multiplier and Multiplicand) 3. Set Multiplicand value as acounter value. 4. Add Multiplier with itself “Counter-1” number of times. 5. Print The answer.  Add and Shift Method

	<ol style="list-style-type: none"> <li>1. Start</li> <li>2. Accept two 2-digit numbers.(Multiplier and Multiplicand)</li> <li>3. Store multiplier to BL and Multiplicand to CL, Initialize AX with 00.</li> <li>4. Shift BL to left by 1 bit. (Shifted bit will be stored to carry flag)</li> <li>5. If carry flag is set, Add CL to AL, and shift AL to left by 1 bit.</li> <li>6. If carry flag is reset, shift AL to left by 1 bit.</li> <li>7. Repeat step 4 to 6 for 8 times.(As 2 digit numbers contains 8 bits)</li> <li>8. Print the result from AX.</li> <li>9. Stop</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>

**Prerequisites: COA**

**Concepts related Theory:**

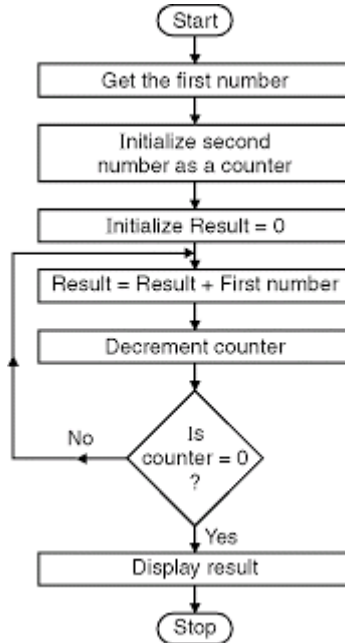
**Successive Addition Method:**

Consider that a byte is present in the AL register and second byte is present in the BL register.

- We have to multiply the byte in AL with the byte in BL.
- We will multiply the numbers using successive addition method.
- In successive addition method, one number is accepted and other number is taken as a counter. The first number is added with itself, till the counter decrements to zero.
- Result is stored in DX register. Display the result, using display routine.

➤ For example :  $AL = 12\text{ H}$ ,  $BL = 10\text{ H}$

- Result = 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H +  
12H + 12H
- Result = 0120 H



### Add and Shift Method:

Consider that one byte is present in the AL register and another byte is present in the BL register. We have to multiply the byte in AL with the byte in BL.

We will multiply the numbers using add and shift method. In this method, you add number with itself and rotate the other number each time and shift it by one bit to left alongwith carry. If carry is present add the two numbers.

Initialize the count to 4 as we are scanning for 4 digits. Decrement counter each time the bits are added. The result is stored in AX. Display the result.

For example : AL = 11 H, BL = 10 H, Count = 4

### Step I :

$$\begin{array}{r} \text{AX} = 11 \\ + \quad 11 \\ \hline 22 \text{ H} \end{array}$$

Rotate BL by one bit to left along with carry.

$$\text{BL} = 10 \text{ H} \quad 0 \quad \neg \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0$$

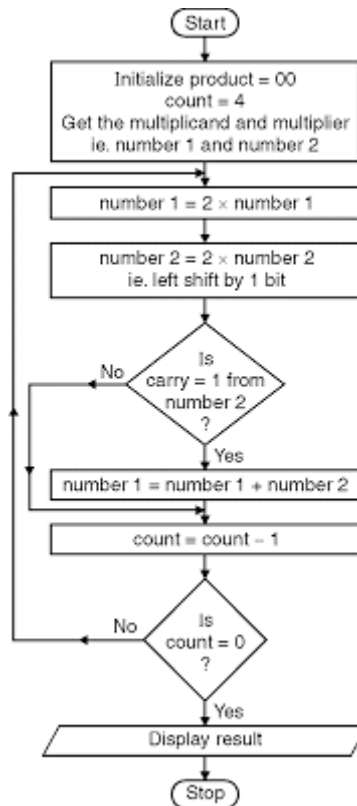
$$\text{BL} = \begin{array}{ccccc} & & \text{CY} & & \\ & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ & \text{CY} & & 2 & & & 0 & & & \end{array}$$

**Step II** : Now decrement counter count = 3.

Check for carry, carry is not there so add number with itself.

$$\mathbf{AX} = \begin{matrix} 22 \\ + 22 \end{matrix}$$

		44 H	
Rotate BL to left,			
BL = 0 -0	100		0000
CY	4		0
Carry is not there.			
Decrement count, count=2			
<b>Step III :</b> Add number with itself			
	AX = 44		
	+ 44		
	88 H		
Rotate BL to left,			
BL = 0 1	000		0000
CY	8		0
Carry is not there.			
<b>Step IV :</b> Decrement counter count = 1.			
Add number with itself as carry is not there.			
	AX = 88		
	+ 88		
	110 H		
Rotate BL to left,			
BL = 1	0000		0000
CY	0		0
Carry is there.			
<b>Step V :</b> Decrement counter = 0.			
Carry is present.			
\ add AX, BX			
	\ 0110 i.e. 11 H		
	+ 0000     ' 10 H		
	0110 H    0110 H		



### Algorithm:

#### Successive Addition:

1. Start
2. Accept two 2-digit numbers. (Multiplier and Multiplicand)
3. Set Multiplicand value as a counter value.
4. Add Multiplier with itself “Counter-1” number of times.
5. Print The answer.

#### Add and Shift Method

1. Start
2. Accept two 2-digit numbers. (Multiplier and Multiplicand)
3. Store multiplier to BL and Multiplicand to CL, Initialize AX with 00.
4. Shift BL to left by 1 bit. (Shifted bit will be stored to carryflag)
5. If carry flag is set, Add CL to AL, and shift AL to left by 1 bit.
6. If carry flag is reset, shift AL to left by 1 bit.
7. Repeat step 4 to 6 for 8 times. (As 2 digit numbers contains 8)

- bits)
8. Print the result from AX.
  9. Stop

**Conclusion:**

We have studied following multiplication techniques in ALP.

1. Successive Addition
2. Add and Shift Method.

**Review Questions:**

1. Explain different multiplication techniques in ALP.
2. Explain booth's algorithm.



## ASSIGNMENT NUMBER: 9

<b>ASSINGMENT NO.</b>	9
<b>TITLE</b>	Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character
<b>PROBLEM STATEMENT /DEFINITION</b>	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 GLOBAL and EXTERN directives is mandatory.
<b>OBJECTIVE</b>	To understand how to implement NEAR and FAR procedure.
<b>OUTCOME</b>	Students will study NEAR and FAR procedure and there application.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>➤ “The Intel microprocessor”, Barry B. Brey.</li><li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>➤ “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>

<p><b>STEPS</b></p>	<p><b>Create a one Folder which contain 2 program file and 1 text file.</b></p> <p>Write all input text to the text file.</p> <p>Define all parameter required to execute above procedure in 1<sup>st</sup> program file.</p> <p>Define all the procedure (required to count and print the number of blank space,procedure required to count and print the number of ENTER in text file and procedure required to count number of occurrences of given character in text file ) in 2<sup>nd</sup> program file</p> <p>Using GLOBAL keyword.</p> <p>Declare a file name in section .data</p> <p>Open a file using OPEN system call.</p> <p>Check the descriptor value of open file.(if value is negative means file is not open and if value is positive means file is open )</p> <p>Save the file descriptor for further use.</p> <p>Read the content of text file using read system call</p> <p>To find the Number of Spaces from the text file compare each entry from the text with 20H (ASCII Value of space) ,count the value and print.</p> <p>To find the Number of Enter from the text file compare each entry from the text with 0x0A (ASCII Value of enter) ,count the value and print.</p> <p>Accept the letter whose number of occurrences find in the text file.</p> <p>To find the Number of occurrences from the text file compare each entry from the text with ASCII Value of entered text,count the value and print.</p> <p>Close the file.</p> <p>Exit.</p>
---------------------	--

<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Title</li> <li>2. Problem Definition</li> <li>3. Objective: Intention behind study</li> <li>4. Software &amp; Hardware requirements</li> <li>5. Explanation of the assignment</li> <li>6. Algorithm</li> <li>7. State Diagram</li> <li>8. Test cases for program.</li> <li>9. Print outs</li> <li>10. Conclusion.</li> </ol>
---	--

### Assignment No. 5

- **Aim:** Write X-86 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 GLOBAL and EXTERN directives is mandatory.
- **Prerequisites :** Program Transfer control instruction, File system
- **Concept related Theory:**
  - **OPEN File**

```

mov rax, 2      ; 'open' syscall
mov rdi, fname1 ; file name
mov rsi, 2      ; File access mode
mov rdx, 0777   ; permissions set
Syscall
mov [fd_in], rax

```

- **READ File**

```

mov rax, 0      ; 'Read' syscall

```

```
mov rdi, [fd_in] ; file Pointer
mov rsi, Buffer ; Buffer for read
mov rdx, length ; len of data want to read
```

**Syscall**

- **WRITE File**

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

**Syscall**

- **CLOSE File**

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

- **Conclusion**

We have studied Near and Far process and there application.

- **Review Questions:**

1. Difference between Near and Far procedure.
2. Explain GLOBAL and Extern Assembler Directives.
3. What is difference between procedure and Macro?
4. What is difference between procedure and interrupt?
5. How to call a procedure?

## ASSIGNMENT NUMBER: 7

<b>TITLE</b>	<b>To read and display contents pointed by GDTR, LDTR and IDTR.</b>
<b>PROBLEM STATEMENT /DEFINITION</b>	<b>Write an ALP to read and display the table content pointed by GDTR/LDTR and IDTR</b>
<b>OBJECTIVE</b>	To understand how to read and display contents of GDTR, LDTR and IDTR registers.
<b>OUTCOME</b>	Students will study different Descriptor tables in system also different registers associated with it.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>➤ “The Intel microprocessor”, Barry B. Brey.</li><li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>➤ “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>
<b>STEPS</b>	<ol style="list-style-type: none"><li>1. Start</li><li>2. Store the value of GDTR,IDTR and LDTR in respective variable.</li><li>3. Display the result</li><li>4. Stop</li></ol>

<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Title</li> <li>2. Problem Definition</li> <li>3. Objective: Intention behind study</li> <li>4. Software &amp; Hardware requirements</li> <li>5. Explanation of the assignment</li> <li>6. Algorithm</li> <li>7. State Diagram</li> <li>8. Test cases for program.</li> <li>9. Print outs</li> <li>10. Conclusion.</li> </ol>
---	--

### Assignment No. 7

- **Aim: To display the contents pointed by System Address Registers**
- **Prerequisites :** 80386 Architecture
- **Concept related Theory:**

#### GDTR and IDTR

These registers hold the 32-bit linear base address and 16-bit limit of the GDT and IDT, respectively.

The GDT and IDT segments, since they are global to all tasks in the system, are defined by 32-bit linear addresses (subject to page translation if paging is enabled) and 16-bit limit values.

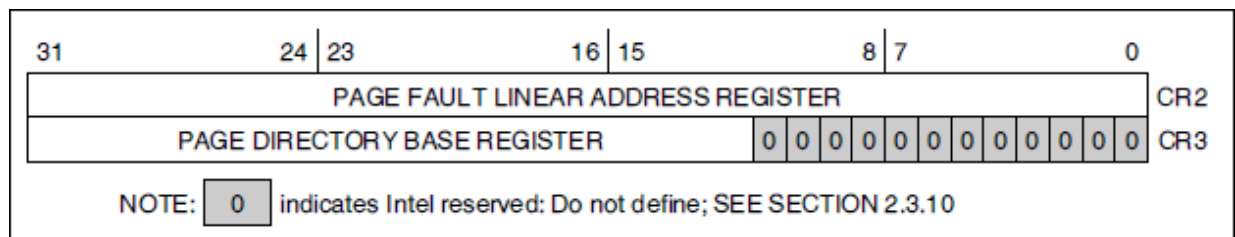


Figure 1. Control Registers 2 and 3

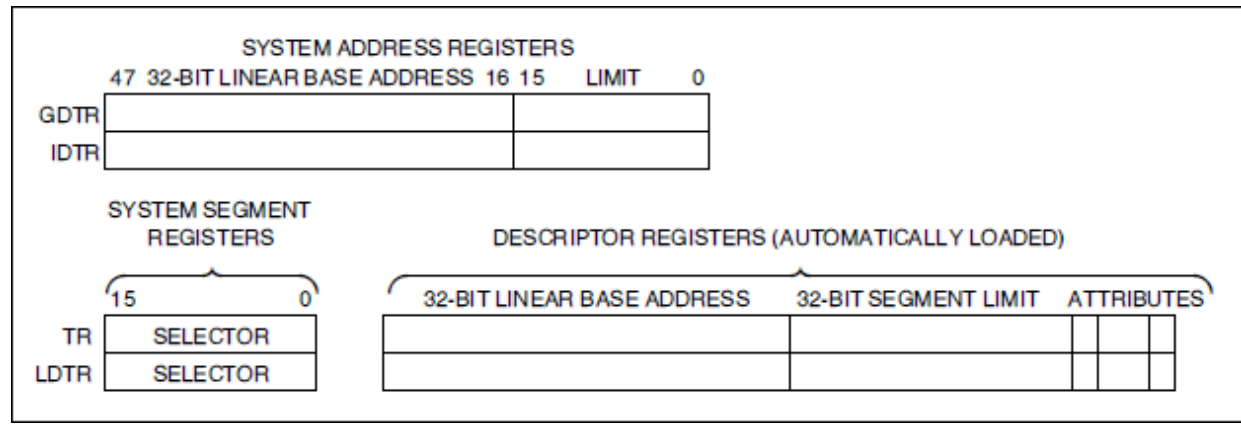


Figure2. System Address and System Segment Registers

### **LDTR and TR**

These registers hold the 16-bit selector for the LDT descriptor and the TSS descriptor, respectively. The LDT and TSS segments, since they are task specific segments, are defined by selector values stored in the system segment registers. Note that a segment descriptor register (programmer-invisible) is associated with each system segment register.

- **Conclusion**

We have studied different Descriptor tables in system also different registers associated with it.

- **Review Questions:**

1. What are the different descriptor tables of 80386 processors?
2. What is Selector?
3. Explain the descriptor format.
4. Explain descriptor cache.
5. What are the types of Descriptors?
6. What is size of GDTR, LDTR IDTR, TR?
7. How to get the base address of GDT/LDT/IDT?

## ASSIGNMENT NUMBER :

<b>TITLE</b>	Write X86 program to sort the list of integers in ascending/descending order. Read the input from the text file and write the sorted data back to the same text file using bubble sort
<b>PROBLEM STATEMENT /DEFINITION</b>	Write X86 program to sort the list of integers in ascending/descending order. Read the input from the text file and write the sorted data back to the same text file using bubble sort .
<b>OBJECTIVE</b>	To understand how to implement Bubble sort using ALP.
<b>OUTCOME</b>	Students will study sorting of number in ALP.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>➤ “The Intel microprocessor”, Barry B. Brey.</li><li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>➤ “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>
<b>STEPS</b>	<ol style="list-style-type: none"><li>1. Create a text file which contain the single digit number.</li><li>2. Create one assembly language program file which contain the following step.</li><li>3. Open the text file and check that is it successfully opened or not.</li><li>4. Read the content of file and store it in buffer.</li><li>5. Sort the contain of buffer using bubble sort method.</li><li>6. write the content of buffer in the text file.</li><li>7. close the file.</li></ol>



<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Title</li> <li>2. Problem Definition</li> <li>3. Objective: Intention behind study</li> <li>4. Software &amp; Hardware requirements</li> <li>5. Explanation of the assignment</li> <li>6. Algorithm</li> <li>7. State Diagram</li> <li>8. Test cases for program.</li> <li>9. Print outs</li> <li>10. Conclusion.</li> </ol>
---	--

### Assignment No. 7

- **Aim:** Write X86 program to sort the list of integers in ascending/descending order. Read the input from the text file and write the sorted data back to the same text file using bubble sort

- **Prerequisites :** Data Structure, Instruction set 80386, File System

- **Concept related Theory:**

- **OPEN File**

**mov rax, 2 ; 'open' syscall**

**mov rdi, fname1 ; file name**

**mov rsi, 2 ; File access mode**

**mov rdx, 0777 ; permissions set**

**Syscall**

**mov [fd\_in], rax**

- **READ File**

**mov rax, 0 ; 'Read' syscall**

```
mov rdi, [fd_in] ; file Pointer
mov rsi, Buffer ; Buffer for read
mov rdx, length ; len of data want to read
```

**Syscall**

- **WRITE File**

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

**Syscall**

- **CLOSE File**

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

- **Conclusion**

**We have studied how to implement bubble sort using ALP.**

- **Review Questions**

## ASSIGNMENT NUMBER :

<b>TITLE</b>	<b>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.</b>
<b>PROBLEM STATEMENT /DEFINITION</b>	<b>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.</b>
<b>OBJECTIVE</b>	To understand how to implement OS (DOS) commands using file operations.
<b>OUTCOME</b>	Students will study different DOS commands and file operations.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>➤ “The Intel microprocessor”, Barry B. Brey.</li><li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>➤ “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>

<b>STEPS</b>	<p><b>DOS COPY Command</b></p> <ol style="list-style-type: none"> <li>1. Start</li> <li>2. pop no. of arguments, exec arguments and source filename.</li> <li>3. Save source file name.</li> <li>4. Pop destination filename.</li> <li>5. Save destination filename.</li> <li>6. Open the file for reading.</li> <li>7. Open or create the file for writing.</li> <li>8. Close source file.</li> <li>9. Close destination file.</li> <li>10. Exit.</li> </ol> <p><b>DOS DELETE Command</b></p> <ol style="list-style-type: none"> <li>1. Start</li> <li>2. pop no. of arguments, exec arguments and source filename.</li> <li>3. Save source file name.</li> <li>4. Delete file</li> <li>5. Exit.</li> </ol> <p><b>DOS TYPE Command</b></p> <ol style="list-style-type: none"> <li>1. Start</li> <li>2. pop no. of arguments, exec arguments and source filename.</li> <li>3. Save source file name.</li> <li>4. Open the file for reading.</li> <li>5. Write to the terminal</li> <li>6. Close source file</li> <li>7. Exit.</li> </ol>
--------------	--

<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Title</li> <li>2. Problem Definition</li> <li>3. Objective: Intention behind study</li> <li>4. Software &amp; Hardware requirements</li> <li>5. Explanation of the assignment</li> <li>6. Algorithm</li> <li>7. State Diagram</li> <li>8. Test cases for program.</li> <li>9. Print outs</li> <li>10. Conclusion.</li> </ol>
---	--

### Assignment No. 8

- **Aim:** 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.
- **Prerequisites :** Linux commands, Command Line
- **Concept related Theory:**
  - OPEN File

```
mov rax, 2      ; 'open' syscall
```

```
mov rdi, fname1 ; file name
```

```
mov rsi, 0      ;
```

```
mov rdx, 0777   ; permissions set
```

Syscall

```
mov [fd_in], rax
```

- OPEN File/Create file

```
mov rax, 2      ; 'open' syscall
```

```
mov rdi, fname1 ; file name
```

```
mov rsi, 0102o  ; read and write mode, create if not
```

```
mov rdx, 0666o    ; permissions set
```

Syscall

```
mov [fd_in], rax
```

- READ File

```
mov rax, 0        ; „Read' syscall
```

```
mov rdi, [fd_in]  ; file Pointer
```

```
mov rsi, Buffer    ; Buffer for read
```

```
mov rdx, length   ; len of data want to read
```

Syscall

- WRITE File

```
mov rax, 01       ; „Write' syscall
```

```
mov rdi, [fd_in]  ; file Pointer
```

```
mov rsi, Buffer    ; Buffer for write
```

```
mov rdx, length   ; len of data want to read
```

Syscall

- DELETE File

```
mov rax, 87
```

```
mov rdi, Fname
```

```
syscall
```

- CLOSE File

```
mov rax, 3
```

```
mov rdi, [fd_in]
```

```
syscall
```

TYPE Command:

- Open file in read mode using open interrupt.

- Read contents of file using read interrupt.
- Display contents of file using write interrupt.
- **Close file using close interrupt**

#### **COPY Command:**

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

#### **DELETE Command:**

- DELETE file using delete interrupt.

- **Conclusion**

We have studied different DOS commands and file operations.

- **Review Questions:**

1. What are the different DOS commands?
2. How to provide read, write and execute permission?
3. How to write to a file?
4. How to read from a file?
5. What is the return value of Open system call?
6. What is the return value of Read system call?

## ASSIGNMENT NUMBER: 10

<b>TITLE</b>	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.
<b>PROBLEM STATEMENT /DEFINITION</b>	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.
<b>OBJECTIVE</b>	To understand how to use stack segment for recursion.
<b>OUTCOME</b>	Students will study recursion using stack in ALP.
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>• “The Intel microprocessor”, Barry B. Brey.</li><li>• “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>• “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>



<b>STEPS</b>	<ul style="list-style-type: none"> <li>➤ Start.</li> <li>➤ Accept the number from user.</li> <li>➤ Convert that number into Hexadecimal(ASCII TO HEX).</li> <li>➤ Compare accepted number with 1,If it is equal to 1 go to step 5,else push the number on stack and decrement the number and go to step 4</li> <li>➤ pop the content of stack and multiply with number</li> <li>➤ repeat the step until stack become empty.</li> <li>➤ Convert the number from HEX to ASCII</li> <li>➤ Print the number.</li> <li>➤ End.</li> </ul>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ul style="list-style-type: none"> <li>• Title</li> <li>• Problem Definition</li> <li>• Objective: Intention behind study</li> <li>• Software &amp; Hardware requirements</li> <li>• Explanation of the assignment</li> <li>• Algorithm</li> <li>• State Diagram</li> <li>• Test cases for program.</li> <li>• Print outs</li> <li>• Conclusion.</li> </ul>

### Assignment No. 10

- **Aim:** 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.
- **Prerequisites :** COA
- **Concept related Theory:**

## **PUSH -- Push Operand onto the Stack**

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 80386 PUSH eSP instruction pushes the value of eSP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

Opcode		Instruction	Clocks	Description
FF	/6	PUSH m16	5	Push memory word
FF	/6	PUSH m32	5	Push memory dword
50	+ /r	PUSH r16	2	Push register word
50	+ /r	PUSH r32	2	Push register dword
6A		PUSH imm8	2	Push immediate byte
68		PUSH imm16	2	Push immediate word
68		PUSH imm32	2	Push immediate dword
0E		PUSH CS	2	Push CS
16		PUSH SS	2	Push SS
1E		PUSH DS	2	Push DS
06		PUSH ES	2	Push ES
0F	A0	PUSH FS	2	Push FS
0F	A8	PUSH GS	2	Push GS

## **POP -- Pop a Word from the Stack**

POP replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the 80386 stack, addressed by SS:SP (address-size attribute of 16 bits) or SS:ESP (addresssize attribute of 32 bits). The stack pointer SP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

Opcode		Instruction	Clocks	Description
8F	/0	POP m16	5	Pop top of stack into memory word
8F	/0	POP m32	5	Pop top of stack into memory dword
58	+ rw	POP r16	4	Pop top of stack into word register
58	+ rd	POP r32	4	Pop top of stack into dword register
1F		POP DS	7, pm=21	Pop top of stack into DS
07		POP ES	7, pm=21	Pop top of stack into ES
17		POP SS	7, pm=21	Pop top of stack into SS
0F	A1	POP FS	7, pm=21	Pop top of stack into FS
0F	A9	POP GS	7, pm=21	Pop top of stack into GS

**11. Conclusion:** In this way we studied to use stack segment for recursion.

## ASSIGNMENT NUMBER:

<b>ASSINGMENT NO.</b>	
<b>TITLE</b>	Write 80387 ALP to find the roots of the quadratic equation. All the possible cases must be considered in calculating the roots.
<b>PROBLEM STATEMENT / DEFINITION</b>	Write 80387 ALP to find the roots of the quadratic equation. All the possible cases must be considered in calculating the roots.
<b>OBJECTIVE</b>	To be able to solve mathematical problems in Assembly language programming
<b>OUTCOME</b>	Students will be efficient in handling and solving mathematical problems using ALP
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<p><b>Processor:</b> Core 2 duo/i3/i5/i7</p> <p><b>OS:</b> Linux 32bit/64bit OS</p> <p><b>Editor:</b> gedit/vi</p> <p><b>Assembler:</b> NASM</p> <p><b>Debugger:</b> GDB</p>
<b>REFERENCES</b>	<ul style="list-style-type: none"> <li>➤ “The Intel microprocessor”, Barry B. Brey.</li> <li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li> <li>“80386 Microprocessor Handbook”, Chris H. Papas.</li> </ul>
<b>STEPS</b>	<ol style="list-style-type: none"> <li>1. Write msg to enter the Quadratic equation</li> <li>2. Write msg to enter first coefficient a</li> <li>3. Read first coefficient</li> <li>4. Convert the no from ascii to hex</li> <li>5. Write msg to enter second coefficient b</li> <li>6. Read second coefficient</li> <li>7. Convert the no from ascii to hex</li> <li>8. Write msg to enter third coefficient c</li> <li>9. Read third coefficient</li> <li>10. Convert the no from ascii to hex</li> <li>11. Write down the quadratic formula <math>x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}</math></li> <li>12. Identify the values of a, b, and c in the quadratic equation.</li> </ol>

	<p>The variable <math>a</math> is the coefficient of the <math>x^2</math> term, <math>b</math> is the coefficient of the <math>x</math> term, and <math>c</math> is the constant. For the equation <math>3x^2 - 5x - 8 = 0</math>, <math>a = 3</math>, <math>b = -5</math>, and <math>c = -8</math>. Write this down.</p> <p>13. Substitute the values of <math>a</math>, <math>b</math>, and <math>c</math> into the equation. Now that you know the values of the three variables, you can just plug them into the equation like this:</p> <ol style="list-style-type: none"> <li><math>\{-b \pm \sqrt{b^2 - 4ac}\}/2</math></li> <li><math>\{-(-5) \pm \sqrt{(-5)^2 - 4(3)(-8)}\}/2(3) =</math></li> <li><math>\{-(-5) \pm \sqrt{(-5)^2 - (-96)}\}/2(3)</math></li> </ol> <p>14. Do the math. After you've plugged in the</p> <ol style="list-style-type: none"> <li>numbers, do the remaining math to simplify</li> <li>positive or negative signs, multiply, or square the</li> <li>remaining terms. Here's how you do it:</li> <li><math>\{-(-5) \pm \sqrt{(-5)^2 - (-96)}\}/2(3) =</math></li> <li><math>\{5 \pm \sqrt{25 + 96}\}/6</math></li> <li><math>\{5 \pm \sqrt{121}\}/6</math></li> </ol> <p>15. Simplify the square root. If the number under the radical symbol is a perfect square, you will get a whole number. If the number is not a perfect square, then simplify to its simplest radical version. If the number is negative, <i>and you're sure it's supposed to be negative</i>, then the roots will be complex. In this example, <math>\sqrt{121} = 11</math>. You can write that <math>x = (5 \pm 11)/6</math>.</p> <p>16. Convert the final roots from hex to ascii</p> <p>17. Print the final roots. Real and imaginary.</p>
<p><b>INSTRUCTIONS FOR WRITING JOURNAL</b></p>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>

## Assignment No. 10

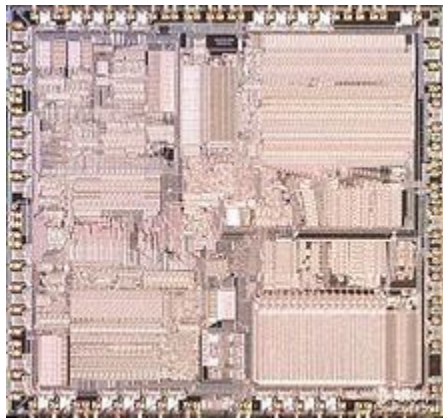
**Aim:** Write 80387 ALP to find the roots of the quadratic equation. All the possible cases must be considered in calculating the roots.

**Prerequisites:** FPU instruction set

**Concepts related Theory:**

### 80387 Microprocessor:

The **80387 (387 or i387)** is the first Intel coprocessor to be fully compliant with the IEEE 754-1985 standard. Released in 1987, a full two years after the 386 chip, the i387 includes much improved speed over Intel's previous 8087/80287 coprocessors, and improved characteristics of its trigonometric functions. The 8087 and 80287's FPTAN and FPATAN instructions are limited to an argument in the range  $\pm\pi/4$  ( $\pm 45^\circ$ ), and the 8087 and 80287 have no *direct* instructions for the sin and cos functions.



Intel 80387 CPU Die Image

Without a coprocessor, the 386 normally performs floating-point arithmetic through (slow) software routines, implemented at runtime through a software exception-handler. When a math coprocessor is paired with the 386, the coprocessor performs the floating point arithmetic in hardware, returning results much faster than an (emulating) software library call.

The i387 is compatible only with the standard i386 chip, which has a 32-bit processor bus. The later cost-reduced i386SX, which has a narrower 16-bit data bus, can not interface with the i387's 32-bit bus. The i386SX requires its own coprocessor, the 80387SX, which is compatible with the SX's narrower 16-bit data bus.

### Instruction set:

All instructions of 80386 MOV, JC,JNC,JG,JZ,JNZ,PUSH POP,INC DEC,CMP, ADD ETC are used in this program.

### Algorithm:

1. Write msg to enter the Quadratic equation
2. Write msg to enter first coefficient a
3. Read first coefficient
4. Convert the no from ascii to hex
5. Write msg to enter second coefficient b
6. Read second coefficient
7. Convert the no from ascii to hex
8. Write msg to enter third coefficient c
9. Read third coefficient
10. Convert the no from ascii to hex
11. Write down the quadratic formula  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
12. Identify the values of a, b, and c in the quadratic equation. The variable  $a$  is the coefficient of the  $x^2$  term,  $b$  is the coefficient of the  $x$  term, and  $c$  is the constant. For the equation  $3x^2 - 5x - 8 = 0$ ,  $a = 3$ ,  $b = -5$ , and  $c = -8$ . Write this down.
13. Substitute the values of a, b, and c into the equation. Now that you know the values of the three variables, you can just plug them into the equation like this:

$$\{-b \pm \sqrt{b^2 - 4ac}\} / 2$$

$$\{-(-5) \pm \sqrt{((-5)^2 - 4(3)(-8))}\} / 2(3) =$$

$$\{-(-5) \pm \sqrt{((-5)^2 - (-96))}\} / 2(3)$$

14. Do the math. After you've plugged in the numbers, do the remaining math to simplify positive or negative signs, multiply, or square the remaining terms. Here's how you do it:

$$\{-(-5) \pm \sqrt{((-5)^2 - (-96))}\} / 2(3) =$$

$$\{5 \pm \sqrt{(25 + 96)}\} / 6$$

$$\{5 \pm \sqrt{(121)}\} / 6$$

14. Simplify the square root. If the number under the radical symbol is a perfect square, you will get a whole number. If the number is not a perfect square, then simplify to its simplest radical version. If the number is negative, *and you're sure it's supposed to be negative*, then the roots will be complex. In this example,  $\sqrt{(121)} = 11$ . You can write that  $x = (5 \pm 11) / 6$ .
15. Convert the final roots from hex to ascii
16. Print the final roots. Real and imaginary.

**Conclusion:** Thus Roots of Quadratic equations are calculated following above steps.

**Review Questions:** Difference between 80386 and 80387 microprocessor?

### ASSIGNMENT NUMBER:

<b>TITLE</b>	<b>Write 80387 ALP to plot Sine Wave, Cosine Wave and Sinc function. Access video memory directly for plotting.</b>
<b>PROBLEM STATEMENT /DEFINITION</b>	<b>Write 80387 ALP to plot Sine Wave, Cosine Wave and Sinc function. Access video memory directly for plotting.</b>
<b>OBJECTIVE</b>	To understand how to how to access video memory for plotting.
<b>OUTCOME</b>	Students will study video memory plotting and regen buffer concepts..
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>➤ “The Intel microprocessor”, Barry B. Brey.</li><li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>➤ “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>
<b>STEPS</b>	



<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Title</li> <li>2. Problem Definition</li> <li>3. Objective: Intention behind study</li> <li>4. Software &amp; Hardware requirements</li> <li>5. Explanation of the assignment</li> <li>6. Algorithm</li> <li>7. State Diagram</li> <li>8. Test cases for program.</li> <li>9. Print outs</li> <li>10. Conclusion.</li> </ol>
---	--

### Assignment No. 11

- **Aim: Write 80387 ALP to plot Sine Wave, Cosine Wave and Sinc function. Access video memory directly for plotting.**
- **Prerequisites : Dos Interrupts, FPU instruction**
- **Concept related Theory:**
  - The overall display characteristics, such as vertical and horizontal resolution, background color, and palette, are controlled by values written to I/O ports whose addresses are hardwired on the adapter,
  - whereas the appearance of each individual character or graphics pixel on the display is controlled by a specific location within an area of memory called the regen buffer or refresh buffer.
  - Both the CPU and the video controller access this memory;
  - The software updates the display by simply writing character codes or bit patterns directly into the regen buffer. (This is called memory-mapped I/O.)

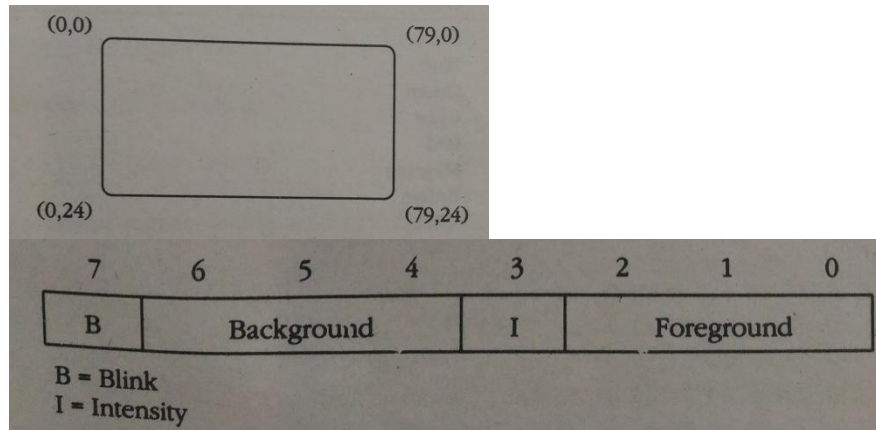
E.g. MDA, CGA, EGA, MCGA, VGA etc.

### Text Mode:

- Sometimes also called as alphanumeric display mode
- Use 16KB of memory starting at segment B800H

- 2 Bytes per character (ASCII Code & Attribute)
- For 80-by-25 text mode

$$\text{offset} = (y * 80 + x) * 2$$

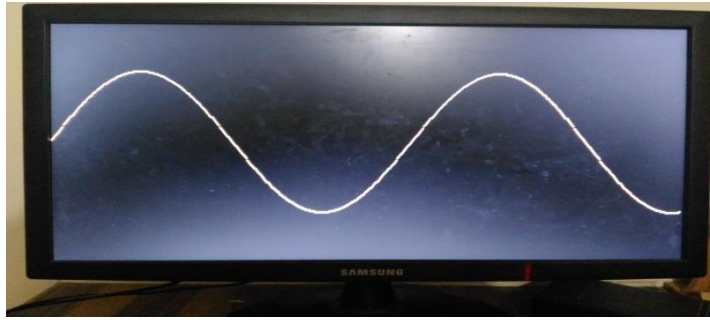


#### Graphics Mode:

- More complicated
- Each bit or group of bits in regen buffer corresponds to an addressable point or pixel on the screen
- E.g. 640-by-200, 2-color graphics display mode of the CGA
- Each pixel is represented by a single bit
- (x,y) range (0,0) through (639,199)
- The memory map is set up so that all the even y coordinates are scanned as a set and all the odd y coordinates are scanned as a set; this mapping is referred to as the memory interlace.
- $\text{Offset} = ((y \text{ AND } 1) * 2000\text{H}) + (y/2 * 50\text{H}) + (x/8)$
- $\text{Bit position} = 7 - (x \text{ MOD } 8)$   
( 8 byte table or array of bit masks and operation „x AND 7“)

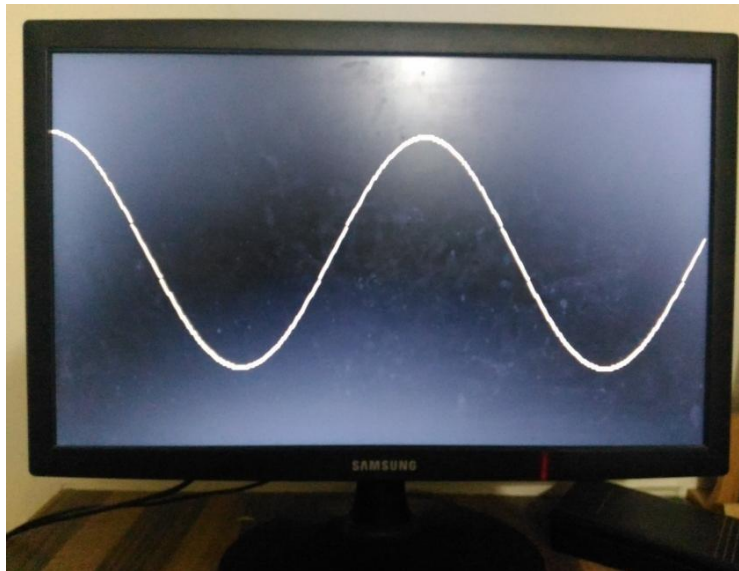
#### Sine Wave:

- $Y = 100 - 60 \sin (\{ \pi / 180 \} * x)$



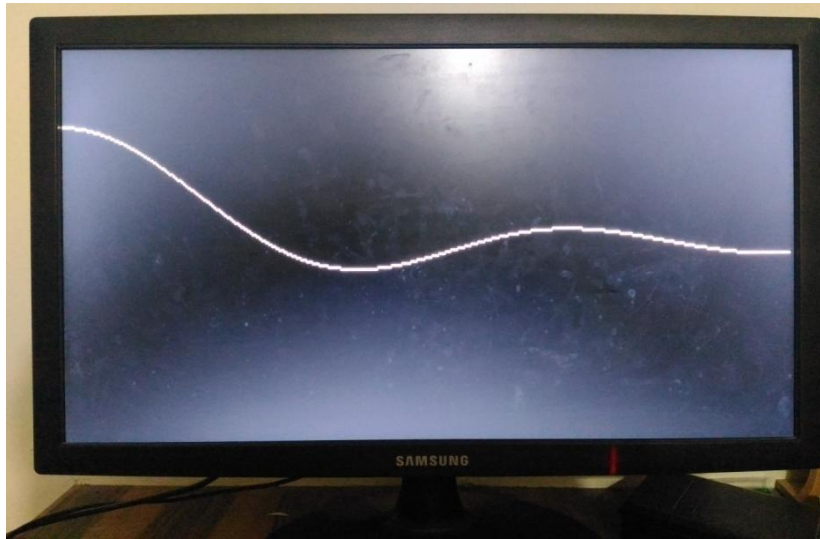
**Cosine Wave:**

- $Y = 100 - 60 \cos (\{\pi/180\} * x)$



**Sinc Function:**

- $\text{Sinc}(x) = \sin(x)/x$



- **Conclusion**

We have studied video memory plotting to plot sine wave, cosine wave and sin function.

- **Review Questions:**

1. What is the starting address of video memory buffer?
2. What is the size of regen buffer? It is divided in how many parts?
3. Calculate offset for 80 by 25 text mode.
4. Explain alphanumeric display mode.

## ASSIGNMENT NUMBER: 12

<b>ASSINGMENT NO.</b>	12
<b>TITLE</b>	Write 80387 ALP to obtain: i) Mean ii) Variance iii) Standard Deviation Also plot the histogram for the data set. The data elements are available in a text file.
<b>PROBLEM STATEMENT /DEFINITION</b>	Write 80387 ALP to obtain: i) Mean ii) Variance iii) Standard Deviation Also plot the histogram for the data set. The data elements are available in a text file.
<b>OBJECTIVE</b>	To be able to solve mathematical problems in Assembly language programming  To be able to handle file and data set from file in ALP,  To be able to include mathematical histogram using ALP
<b>OUTCOME</b>	Students will be efficient in handling and solving mathematical problems through file using ALP
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>Editor:</b> gedit/vi  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"> <li>➤ “The Intel microprocessor”, Barry B. Brey.</li> <li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li> <li>“80386 Microprocessor Handbook”, Chris H. Papas.</li> </ul>
<b>STEPS</b>	<ol style="list-style-type: none"> <li>1. Create a one Folder which contain 1 program file and 1 text file.</li> <li>2. Write all input text to the text file.</li> <li>3. Define all parameter required to execute above procedure in 1<sup>st</sup> program file.</li> </ol>

	<ol style="list-style-type: none"> <li>4. Define all the procedures <ol style="list-style-type: none"> <li>a. Ascii to Hex</li> <li>b. Hex to ascii</li> <li>c. Find Mean</li> <li>d. Find Variance</li> <li>e. Find Standard Deviation</li> </ol> </li> <li>5. Accept /Read the data set</li> <li>6. Perform Ascii to Hex</li> <li>7. Calculate mean</li> <li>8. Print hex Mean value</li> <li>9. Use mean value to calculate variance</li> <li>10. Print hex Variance</li> <li>11. Calculate Standard Deviation</li> <li>12. Print hex Deviation.</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>1. Date</li> <li>2. Assignment no.</li> <li>3. Problem definition</li> <li>4. Learning objective</li> <li>5. Learning Outcome</li> <li>6. Concepts related Theory</li> <li>7. Algorithm</li> <li>8. Test cases</li> <li>10. Conclusion/Analysis</li> </ol>

### Assignment No. 12

**Aim:** Write 80387 ALP to obtain: i) Mean ii) Variance iii) Standard Deviation Also plot the histogram for the data set. The data elements are available in a text file.

**Prerequisites:** Basic instructions of Assembly Language programming

**Concepts related Theory:**

- **OPEN File**

**mov rax, 2 ; 'open' syscall**

```
mov rdi, fname1 ; file name
mov rsi, 2       ; File access mode
mov rdx, 0777    ; permissions set
```

**Syscall**

```
mov [fd_in], rax
```

- **READ File**

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

**Syscall**

- **WRITE File**

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

**Syscall**

- **CLOSE File**

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

**Algorithm:**

**Data set:** Measured height of the dogs

The heights (at the shoulders) are: 600mm, 470mm, 170mm, 430mm and 300mm.

Mean = Addition of all the values /no of values

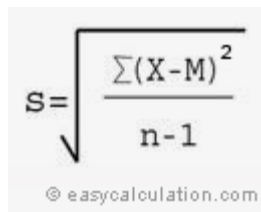
$$\text{Mean} = 600 + 470 + 170 + 430 + 300 \div 5 = 1970 \div 5 = 394$$

**Variance :** Variance =  $s^2$

To calculate the Variance, take each difference, square it, and then average the result:

$$\begin{aligned}\text{Variance: } \sigma^2 &= \frac{206^2 + 76^2 + (-224)^2 + 36^2 + (-94)^2}{5} \\ &= \frac{42,436 + 5,776 + 50,176 + 1,296 + 8,836}{5} \\ &= \frac{108,520}{5} = 21,704\end{aligned}$$

**Sample Standard Deviation :**



The image shows a formula for sample standard deviation:  $s = \sqrt{\frac{\sum (X-M)^2}{n-1}}$ . Below the formula is a small copyright notice: © easycalculation.com.

And the Standard Deviation is just the square root of Variance, so:

*Standard Deviation*

$$\begin{aligned}\sigma &= \sqrt{21,704} \\ &= 147.32... \\ &= 147 \text{ (to the nearest mm)}\end{aligned}$$

**Conclusion:.** Thus we have calculated Mean ,Variance and Standard deviation using ALP

**Review Questions:**

1. How to find Mean , Variance, and Standard deviations?
2. What are the different Floating point Data type



### ASSIGNMENT NUMBER: 13

<b>ASSIGNMENT NO</b>	<b>13</b>
<b>TITLE</b>	<b>Write a Terminate but Stay Resident (TSR) program for a key-logger. The key-presses during the stipulated time need to be displayed at the center of the screen</b>
<b>PROBLEM STATEMENT /DEFINITION</b>	<b>Write a Terminate but Stay Resident (TSR) program for a key-logger. The key-presses during the stipulated time need to be displayed at the center of the screen</b>
<b>OBJECTIVE</b>	To understand Terminate but Stay Resident program.
<b>OUTCOME</b>	Students will study Terminate but Stay Program for a key-logger..
<b>S/W PACKAGES AND HARDWARE APPARATUS USED</b>	<b>Processor:</b> Core 2 duo/i3/i5/i7  <b>OS:</b> Linux 32bit/64bit OS  <b>S/W:</b> key-logger  <b>Assembler:</b> NASM  <b>Debugger:</b> GDB
<b>REFERENCES</b>	<ul style="list-style-type: none"><li>➤ “The Intel microprocessor”, Barry B. Brey.</li><li>➤ “Introduction to 64 bit Intel Assembly Language Programming for Linux”, 2<sup>nd</sup> Edition, Ray Seyfarth,</li><li>➤ “80386 Microprocessor Handbook”, Chris H. Papas.</li></ul>

<b>STEPS</b>	<ol style="list-style-type: none"> <li>1. Far JUMP to transient portion of memory</li> <li>2. Clear IF</li> <li>3. Get vector address</li> <li>4. Save vector address</li> <li>5. Set new vector address</li> <li>6. Make program resident</li> </ol>
<b>INSTRUCTIONS FOR WRITING JOURNAL</b>	<ol style="list-style-type: none"> <li>Title</li> <li>2. Problem Definition</li> <li>3. Objective: Intention behind study</li> <li>4. Software &amp; Hardware requirements</li> <li>5. Explanation of the assignment</li> <li>6. Algorithm</li> <li>7. State Diagram</li> <li>8. Test cases for program.</li> <li>9. Print outs</li> <li>10. Conclusion.</li> </ol>

### Assignment No. 13

**Aim: Write a Terminate but Stay Resident (TSR) program for a key-logger. The key-presses during the stipulated time need to be displayed at the center of the screen**

- **Prerequisites :** MS DOS, DOS interrupts
- **Concept related Theory:**
  - Terminate and Stay Resident (TSR) generally refers to a special class of programs for PC-compatible computers running DOS.
  - When a user exits a normal program running in DOS, the memory that the program used is usually freed for other programs and tasks; therefore, the program must be reloaded from a disk back into memory for it to be used again. When you run a TSR program, however, it loads itself into the computer's memory and remains there for later use. You may run other programs while the TSR is still alive in memory, and these programs may

invoke the TSR or be affected by the behavior of the TSR program. For this reason, TSR programs may give DOS the appearance of multitasking (the ability to perform several tasks at once) which is built into many other operating systems.

- You may use TSR programs for a wide variety of tasks. Some of these programs are active only when you press a hot key (a special key or combination of keystrokes that activates the TSR). An example would be a pop-up calculator program that appears on the screen whenever you press Alt-Shift-c, even from within a separate word processing program. Other TSR programs run continuously in the background and may normally be invisible. Examples of such programs are some network and communications programs and special virus scanner programs that monitor the use of a computer's memory and disk drives. Still other TSR programs may operate in both ways. A visually impaired user might call up a TSR that intercepts text information sent to the screen and displays it in a larger, easier-to-read format.
- TSR programs may be quite complex, and are often difficult to program reliably. TSR programmers must make sure that their programs do not conflict with other programs active in the computer's memory. The TSR must also not interfere with other programs' use of the disk drives or other hardware. Whenever a TSR becomes active, it must carefully record information in memory being used by another program, and restore this information exactly when it transfers control.
- Because of this complexity, TSR programs are often likely culprits when a PC is behaving strangely or crashing. Some TSR programs may not be compatible with other programs because of the way they use memory, or they may conflict with other TSR programs that are also active. When installing TSR programs, it is a good idea to install them one at a time, and make sure that other programs such as word processors and spreadsheets are behaving normally despite the presence of the TSR.
- Another drawback to TSR programs is their consumption of memory. Because a TSR retains a block of the computer's memory as its own, less space is then available to other programs. If several TSR programs are present in memory, there may not be enough space left over to load other large, demanding programs such as spreadsheets. Beginning with version 5.0, however, DOS supports the ability to load high most of a TSR program's contents into expanded and extended memory, which have fewer demands on their use than the computer's main memory, where most programs are executed and consumption of space is critical.
- Key-logger:

key-logger is a type of surveillance software (considered to be either software or spyware) that has the capability to record every keystroke you make to a log file, usually encrypted. A key-logger recorder can record instant message, e-mail, and any information you type at any time using your keyboard. The log file created by the key-logger can then be sent to a specified receiver. Some key-

logger programs will also record any e-mail addresses you use and Web Site URL's you visit.

- **Conclusion**

We have studied Terminate but Stay Resident program for key-logger

- **Review Questions:**

1. What is Terminate but Stay Resident program?
2. What is the key-logger?
3. What are the limitations of Terminate but Stay Resident program?
4. What are the advantages of Terminate but Stay Resident program?.
5. How to make your program TSR?
6. How to get vector address?
7. Hoe to set vector address?
8. What are the types of model?
9. What do you mean by TINY model?