Assignment No-5

Aim: Write X86 ALP to find, a) Number of Blank spaces b) Number of lines c) Occurrence of a particular character. Accept the data from the text file. The text file has to be accessed during Program_1 execution and write FAR PROCEDURES in Program_2 for the rest of the processing. Use of PUBLIC and EXTERN directives is mandatory.

Prerequisites:
Concept of file
Learning objectives:
Understanding FAR procedure, near procedure, Extern and public Directives, File mode, file permission.
New Concept:
Implementation FAR procedure
Theory:

File Handle: A number that the operating system assigns temporarily to a file when it is opened. The operating system uses the file handle internally when accessing the file. A special area of main memory is reserved for file handles, and the size of this area determines how many files can be open at once.

File Permission:
Linux permissions dictate 3 things you may do with a file, read, write and execute. They are referred to in Linux by a single letter each.

- **read** - The Read permission refers to a user's capability to read the contents of the file.
- **write** - The Write permissions refer to a user's capability to write or modify a file or directory.
- **execute** - The Execute permission affects a user's capability to execute a file or view the
  contents of a directory.

For every file we define 3 sets of people for whom we may specify permissions.

- **Owner** - a single person who owns the file. (typically the person who created the file but ownership may be granted to someone else by certain users)
- **Group** - every file belongs to a single group.
- **Others** - everyone else who is not in the group or the owner.

Permission Types

- **u** - Owner
- **g** - Group
- **o** or **a** - All Users

Value	Meaning
777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.
755	*(rwxr-xr-x)* The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
700	*(rwx------)* The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
666	*(rw-rw-rw-)* All users may read and write the file.
644	*(rw-r--r--)* The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.
600	*(rw-------)* The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

File Mode:

File is opened in different mode like read-only , write-only , and read-write , Append etc.Among the file access modes, most commonly used are: read-only (0), write-only (1), and read-write (2).

A system call table

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count
2	sys_open	const char *filename	int flags	int mode
3	sys_close	unsigned int fd		

Creating and Opening a File

For creating and opening a file, perform the following tasks −

- Put the system call sys_open() number 2, in the RAX register.
- Put the filename in the RDI register.
- Put the file opening mode in the RSI register.
- Put the file permissions in the RDX register.

Return:

The system call returns the file descriptor (FILE Handle) of the created/opened file in the RAX register, in case of error, the error code is in the RAX register.

Reading from a File

For reading from a file, perform the following tasks −

- Put the system call sys_read() number 0, in the RAX register.

- Put the file descriptor(File Handle) in the RDI register.
- Put the pointer to the input buffer in the RSI register.
- Put the buffer size, i.e., the number of bytes to read, in the RDX register.

Return:
The system call returns the number of bytes read in the RAX register, in case of error, the error code is in the RAX register.

Writing to a File
For writing to a file, perform the following tasks −

- Put the system call sys_write() number 1, in the RAX register.
- Put the file descriptor in the RDI register.
- Put the pointer to the output buffer in the RSI register.
- Put the buffer size, i.e., the number of bytes to write, in the RDX register.

Return:
The system call returns the actual number of bytes written in the RAX register, in case of error, the error code is in the RAX register.

Closing a File
For closing a file, perform the following tasks −

- Put the system call sys_close() number 3, in the RAX register.
- Put the file descriptor in the RDI register.

The system call returns, in case of error, the error code in the RAX register.

Near and Far Procedure:
Near calls and returns transfer control between procedures in the same code segment. Far calls and returns pass control between different segments.

Near CALL and RET Operation
When executing a near call, the processor does the following (see Figure 28-2):
1. Pushes the current value of the EIP register on the stack.
2. Loads the offset of the called procedure in the EIP register.
3. Begins execution of the called procedure.

When executing a near return, the processor performs these actions:
1. Pops the top-of-stack value (the return instruction pointer) into the EIP register.
2. (If the RET instruction has an optional *n* argument.) Increments the stack pointer by the number of bytes specified with the *n* operand to release parameters from the stack.

Far CALL and RET Operation
When executing a far call, the processor performs these actions:
1. Pushes current value of the CS register on the stack.
2. Pushes the current value of the EIP register on the stack.
3. Loads the segment selector of the segment that contains the called procedure in the CS register.
4. Loads the offset of the called procedure in the EIP register.
5. Begins execution of the called procedure.

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

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

EXTERN: Importing Symbols from Other Modules

EXTERN directive is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one.

The EXTERN directive takes as many arguments as you like. Each argument is the name of a symbol:

```
extern _printf
extern _sscanf, _fscanf
```

GLOBAL: Exporting Symbols to Other Modules

GLOBAL is the other end of EXTERN: if one module declares a symbol as EXTERN and refers to it, then in order to prevent linker errors, some other module must actually *define* the symbol and declare it as GLOBAL. Some assemblers use the name PUBLIC for this purpose.

The GLOBAL directive applying to a symbol must appear *before* the definition of the symbol. GLOBAL uses the same syntax as EXTERN, except that it must refer to symbols which *are* defined in the same module as the GLOBAL directive. For example:

```
global _main
_main: ; some code
```

Algorithm:

Algorithm 1:

1. Create file "macro.asm" which includes all macros for open, read and write the file.
2. Include that file in program1 and program 2.
3. In Programme1 accept file name and character from user then open specified file, read that file and call FAR procedure for processing of file i.e. programme2
4. In proggramme2 ,write FAR procedure for calculation of blank spaces, new line characters and special symbols.
5. Close the file.

Programme1 Algorithm

1. Declare necessary messages and variables under .data and .bss,declare procedure which is to be called as extern and include macro.asm
2. Accept file name and character which is to be searched from the user,appen null character „0" at the end of filename.
3. Open the file(filename) and it returns file descriptor in rax.

4. Compare File descriptor with -1 if equals then give error message or otherwise save it in variable
   FILEHANDLE.
5. Read the file,it returns no. of character to be read in RAX register.
6. Move that no. of characters in variable abuflen.
7. Call FAR procedure.

Programme2 Algo.

1) Declare necessary messages and variables under .data and .bss,declare procedure which is called as global and include macro.asm
2) Point to first character of file which is read in buffer
3) Move that character into register.
4) Compare it with space(20h) if not equal then goto next comparison and if equal increment scount and then go to next where next character is pointed.
5) Compare it with newline(0Ah) if not equal then goto next comparison and if equal increment ncount and then go to next where next character is pointed.
6) Compare it with special character if equal increment ccount and then go to next where next character is pointed.
7) Point to next character ,decrement abuflen and repeat steps 3 to 6 until abuflen becomes zero
8) Display space count(scount),newline count(ncount) and character count(ccount).

Conclusion: Thus, we have successfully studied how to open ,read the file and process the file using assembly language.

Assignment No: 6

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

Prerequisites: Format of

- LDTR
- IDTR
- TR
- MSW Registers

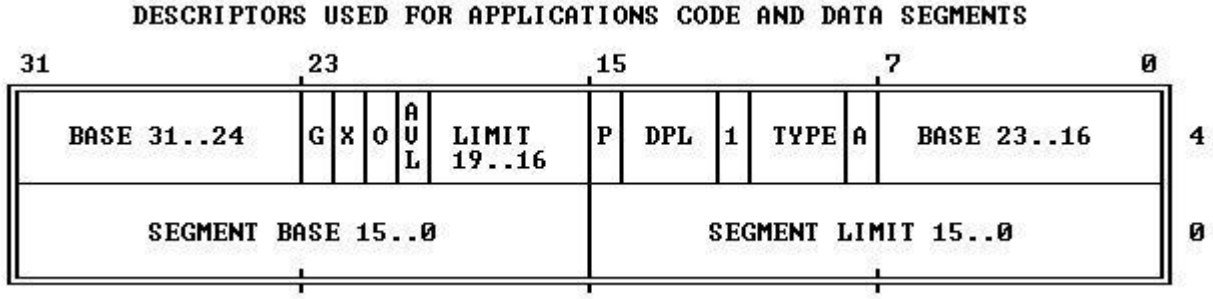
Learning Objectives:

- Understand load and store instruction for LDTR, IDTR, TR, MSW Register.

Theory:

Desciptor:

The segment descriptor provides the processor with the data it needs to map a logical address into a linear address. Descriptors are created by compilers, linkers, loaders, or the operating system, not by applications programmers. All types of segment descriptors take one of these formats. Segment-descriptor fields are:



BASE: Defines the location of the segment within the 4 gigabyte linear address space. The processor concatenates the three fragments of the base address to form a single 32-bit value.

LIMIT: Defines the size of the segment. When the processor concatenates the two parts of the limit field, a 20-bit value results. The processor interprets the limit field in one of two ways, depending on the setting of the granularity bit:

1. In units of one byte, to define a limit of up to 1 megabyte.
2. In units of 4 Kilobytes, to define a limit of up to 4 gigabytes. The limit is shifted left by 12 bits when loaded, and low-order one-bits are inserted.

Granularity bit: Specifies the units with which the LIMIT field is interpreted. When thebit is clear, the limit is interpreted in units of one byte; when set, the limit is interpreted in units of 4 Kilobytes.

TYPE: Distinguishes between various kinds of descriptors.

DPL (Descriptor Privilege Level): Used by the protection mechanism .

Segment-Present bit: If this bit is zero, the descriptor is not valid for use in address transformation; The operating system is free to use the locations marked AVAILABLE. Operating systems that implement segment-based virtual memory clear the present bit in either of these cases:

- When the linear space spanned by the segment is not mapped by the paging mechanism.
- When the segment is not present in memory.

Accessed bit: The processor sets this bit when the segment is accessed; i.e., a selector for the descriptor is loaded into a segment register or used by a selector test instruction. Operating systems that implement virtual memory at the segment level may, by periodically testing and clearing this bit, monitor frequency of segment usage.

Creation and maintenance of descriptors is the responsibility of systems software, usually requiring the cooperation of compilers, program loaders or system builders.

Segment descriptors are stored in either a Global Descriptor Table (GDT) or Local Descriptor Table (LDT).

The 80386+ locates the GDT and the current LDT in memory by means of the GDTR and LDTR registers.

Descriptor tables

Descriptor tables define all the segments used in the protected mode system. The 3 types of tables are:

. Global Descriptor Table (GDT)
. Local Descriptor Table (LDT)
. Interrupt Descriptor Table (IDT)

Descriptor tables are variable-length memory arrays, with 8-byte entries that contain descriptors. In the 80386+, they range in size from 8 bytes to 64K, and each table holds up to 8192 8-byte descriptors. The upper 13 bits of a selector are used as an index into the descriptor table.

GDT

Every protected mode 80386 system contains a Global Descriptor Table. The GDT holds descriptors that are available to all the tasks in a system. Except for descriptors that control interrupts or exceptions, the GDT can contain any other kind of segment descriptor.

Generally, the GDT contains 3 types of descriptors: code and data segments used by the operating system, descriptors for the Local Descriptors in a system, and task state segments (TSS). The first slot of the GDT is not used; it corresponds to the null selector which defines a null pointer value.

LDT

Operating systems are generally designed so that each task has a separate Local Descriptor Table. LDTs provide a way for isolating a given task's code and data segments from the rest of the operating system. The **G**DT contains descriptors for segments that are common to all

tasks. The LDT is associated with a given task and may contain only code, data, stack, task gate, and call gate descriptors. A segment cannot be accessed by a task if its segment descriptor

does not exist either in the current LDT or the GDT. This both isolates and protects that task's segments, while still allowing global **data** to be shared among tasks.Unlike the 48-bit GDTR and IDTR, the LDTR contains only a 16-bit selector which refers to an LDT descriptor in the GDT.

IDT

The Interrupt Descriptor Table contains the descriptors that point to the location of up to 256 interrupt service routines. The IDT can only contain trap gates, task gates, and interrupt gates. The IDT should be at least 256 bytes in size so it can hold descriptors for the 32 Intel-reserved interrupts. Every interrupt used by the system **must** have an entry in the IDT.

Algorithm:

1. Declare memory variable for storing contents of GDTR,LDTR,IDTR,TR and MSW under bss section.
2. Store contents of GDTR into that memory variable by using store instruction.
3. Move contents of lower 2 bytes of GDTR into BX register and call display procedure(hex to Ascii conversion)
4. Move contents of middle 2 bytes of GDTR into BX register and call display procedure(hex to Ascii conversion)
5. Move contents of upper 2 bytes of GDTR into BX register and call display procedure(hex to Ascii conversion)
6. Repeat steps 2 to 5 for IDTR(6 bytes),LDTR(2 bytes),TR and MSW register accordingly.
7. Exit by using Syscall 60.

Conclusion: Thus, we have studied architecture format of GDTR, LDTR, TR and MSW register, how to display contents of these registers.

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: Concepts of

- Bubble Sort
- File Handling

Learning Objectives:

- Understand the memory
- Understand the file operations
- Understand the localization

Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Performance:

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other $O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted efficiently is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.

Step-by-step example.

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

Step-by-step example [edit]

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass

(**5 1** 4 2 8) → (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass

(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)
Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Algorithm:

1. Create macro for opening , reading, writing the file
2. Accept filename from user and append 0 or NULL character at the end of filename.
3. Open the Input file by using macro for opening the file. It returns FILE HANDLE in RAX register.
4. Compare FILE HANDLE with -1, if below or equal then display error message otherwise move that FILE HANDLE in one variable.
5. Read that file by using macro for reading the FILE .It returns length of file within RAX.
6. Decrement RAX (to remove end character) and move that length into variable say flen
7. Point to memory(RSI) in which file contents are read and also point to ARRAY(RDI) in which we want to transmit file data.(input)
8. Transmit data from RSI to RDI, Increment RSI by 2(to exclude new line character) and Increment RDI by one. Increment variable Count (actual length of array) at each transformation.
9. Repeat step 8 until flen becomes zero.
10. Point RSI to array and apply Bubble sort on that array.
11. Write final result of array to input file.
12. Display sorted array.
13. Close the file by using macro for closing the file.
14. end

Algorithm for Bubble Sort:

1. Point RSI to the first number of array and RDI to second number of an ARRAY

2. Move the content of RSI into al register. Compare contents of al and RDI.
3. If AL> [RDI] go to swap. Otherwise increment RSI, RDI and decrement Count.
4. Swap: MOV DL,[RSI]
   MOV [RDI], DL
   MOV [RSI], AL
5. Continue steps 2 to 4 until count becomes zero.

Conclusion: Thus, We have successfully studied how to take input from file, sort the input using bubble sort and rewrite sorted output into same file.

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: Concepts of

- File/ Directory Commands

Learning Objectives:

- Understand the File Operation using Assembly Language.

Theory:

In computing, a **file system** is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names are called a "file system".

There are many different kinds of file systems. Each one has different structure and logic, properties of speed, flexibility, security, size and more.

File systems can be used on numerous different types of storage devices that use different kinds of media. The most common storage device in use today is a hard disk drive. Other kinds of media that are used include flash memory, magnetic tapes, and optical discs. In some cases, computer's main memory (random-access memory, RAM) is used to create a temporary file system for short-term use.

Some file systems are used on local data storage devices others provide file access via a network protocol .Some file systems are "virtual", meaning that the supplied "files" (called **virtual files**) are computed on request (e.g. procfs) or are merely a mapping into a different file system used as a backing store. The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

COPY COMMAND:

Ex:
CP NEWFILE, OLDFILE.

Type **cp** followed by the name of an existing file and the name of the new file.

To copy a file to a different directory (without changing the file's name), specify the directory instead of the new filename.

Ex:

cp newfile testdir

To copy a file to a different directory and create a new file name, you need to specify a directory/a new file name.

Ex:
cp newfile testdir/newerfile
cp newfile ../newerfile
The "…" represents one directory up in the hierarchy.

TYPE COMMAND:
Ex:
Type Filename
It allows displaying contents of filename.

DELETE COMMAND:
Ex:
DEL newfile
DEL followed by the name of a file to remove the file.
Use the wildcard character to remove several files at once. Ex:
DEL n*
This command removes all files beginning with n.
Type DEL -i followed by a filename if you"d like to be prompted before the file is actually removed.
Ex:
DEL -i newfile
DEL -i n*
By using this option, you have a chance to verify the removal of each file. The -i option is very handy when removing a number of files using the wildcard character *

Algorithm:

Copy Command

1. Read the command from command line and store it into one variable.
2. Compare that command with "COPY" if equal then go to step 3 otherwise display No command msg
3. Read the file1name and file2name from command line into variables.
4. Open the file file1name by using open macro and compare returned FILEHANDLE with -1 ,if below or equal then display error message else store that file handle into variable FILEHANDLE.
5. Read the file file1name by using macro for reading file.
6. Open the file file2name by using macro for opening the file ,and write the contents from file1name to file2name(by using macro for writing the file).
7. Close both file1name and file2name.
8. Exit.

TYPE Command:

1. Read the command from command line and store it into one variable.

2. Compare that command with "TYPE" if equal then go to step 3 otherwise display
   Nocommand msg
3. Read the filename from command line into variables.
4. Open the file filename by using open macro and compare returned FILEHANDLE with - 1 ,if below or equal then display error message else store that file handle into variable
   FILEHANDLE.
5. Read the file filename by using macro for reading the file.It reads file contents into memory buffer.
6. Display the buffer.
7. Close the file filename
8. Exit.

DEL Command

1. Read the command from command line and store it into one variable.
2. Compare that command with "DEL" if equal then go to step 3 otherwise display
   Nocommand msg
3. Read the filename from command line into variables
4. By using syscall 87 , Delete the file.
5. Exit.

Conclusion: Thus, We have successfully studied implementation of different file commands.

Assignment No: 9

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: Concepts of
- Factorial
- Recursive Function

Learning Objectives:
- Understand the stack operation in calling recursive function

Recursive Function:

A recursive procedure is one that calls itself. There are two kind of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure. Consider a function which calls itself: we call this type of recursion **immediate** recursion.

What is recursion used for?

Recursion is best used for problems where a large task can be broken down into a repetitive "sub-task". Because a recursive routine calls itself to perform those sub-tasks, eventually the routine will come across a sub-task that it can handle without calling itself. This is known as a <b *base case – and it is needed to prevent the routine from calling itself over and over again without stopping. So, one can say that the base case stops the recursion. </b*

Base cases and Recursion

In the base case, the routine does not call itself. But, when a routine does have to call itself in order to complete its sub-task, then that is known as the *recursive case. So, there are 2 types of cases when using a recursive algorithm: base cases and recursive cases. This is very important to remember when using recursion, "What is my base case and what is the recursive case?"*

```
function factorial (x) {
return (x * factorial(x-1) ) ;
}
```

Compute n! Recursively.

We are given the mathematical function for computing the factorial:

n! = n * (n - 1)!

Why this is a recursive function? To compute n! we are required to compute (n - 1)!.
We are also given the simple case. We define mathematically 0!

0! = 1

To understand factorial in a recursive sense, consider a few simple cases:

1! = 1 * 0! = 1 * 1 = 1

2! = 2 * 1! = 2 * 1 = 2

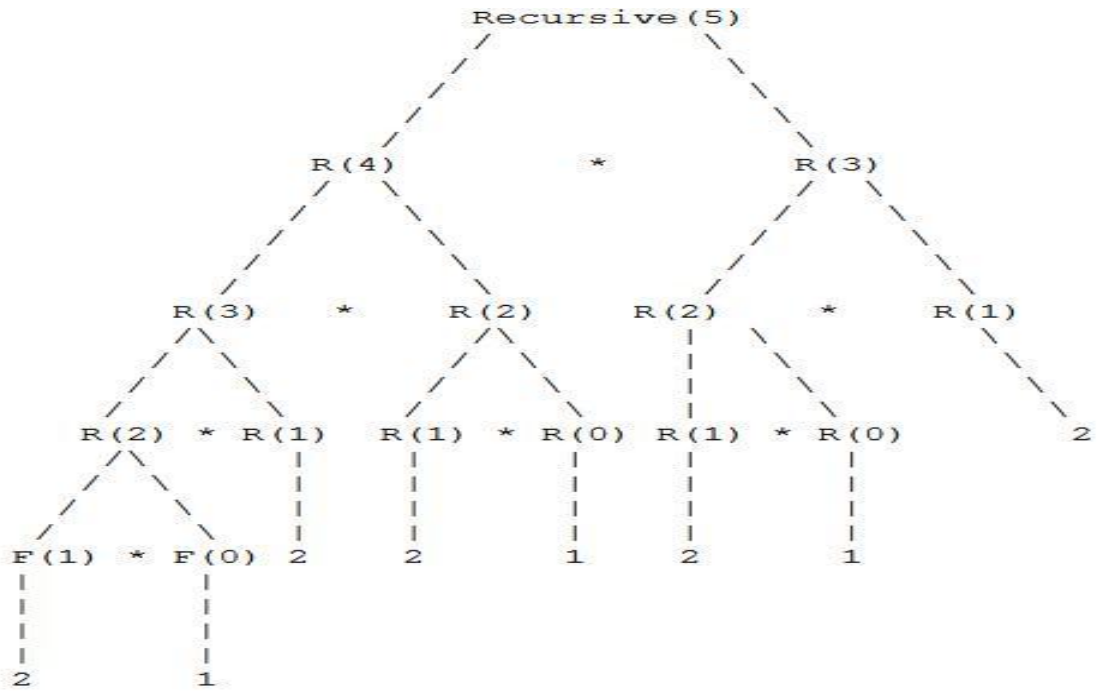
3! = 3 * 2! = 3 * 2 = 6

4! = 4 * (3 * 2 * 1) = 4 * 3! = 24

Consider a more complicated instance:

12! = 12 * 11! = 12 * (11 * 10 * ... * 1) = = ?

Suppose ,F(n)=F(n-1)*f(n-2) ,then recursive tree generated is as follows.



If we evaluate these recursive calls we have

(((2 * 1) * 2) * (2 * 1)) * ((2 * 1) * 2) = 32

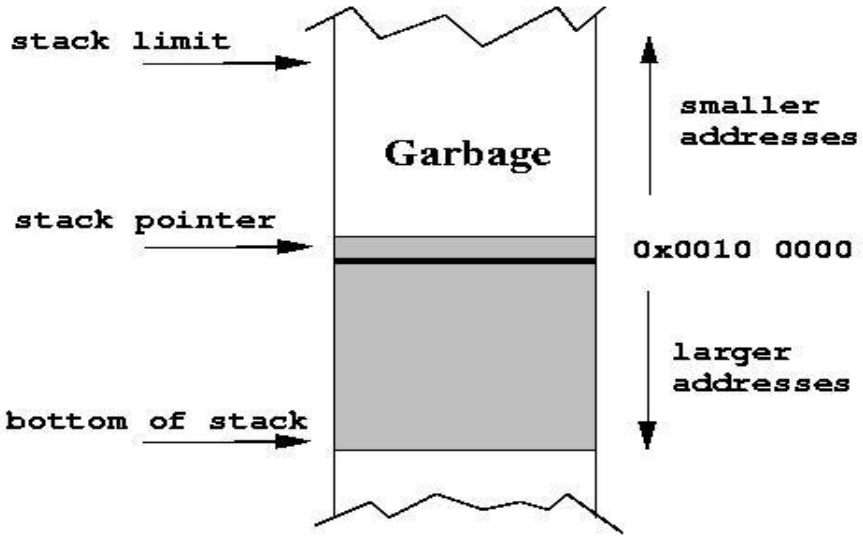
Thus, Recursive(5) = 32.

Recursion Function and Stack:

Stack

When a program starts executing, a certain contiguous section of memory is set aside for the program called the **stack**.

Let's look at a stack.



The stack pointer is usually a register that contains the top of the stack. The stack pointer contains the smallest address **x** such that any address smaller than **x** is considered garbage, and any address greater than or equal to **x** is considered valid.
In the example above, the stack pointer contains the value **0x0000 1000**, which was somewhat arbitrarily chosen.
The shaded region of the diagram represents valid parts of the stack.
It's useful to think of the following aspects of a stack.

- **stack bottom** The largest valid address of a stack. When a stack is initialized, the stack pointer points to the stack bottom.
- **stack limit** The smallest valid address of a stack. If the stack pointer gets smaller than this, then there's a *stack overflow* Other sections of memory are used for the program and for the heap (the section of memory used for dynamic memory allocation).

In the X86, when recursive call to function then at each calling of recursive function value passed to function and return address are stored on to the stack

Push and Pop

Like the data structure ,there are two operations on the stack: push and pop.
Usually, push and pop are defined as follows:

- **push** You can push one or more registers, by setting the stack pointer to a smaller value (usually by subtracting 4 times the number of registers to be pushed on the stack) and copying the registers to the stack.

- **pop** You can pop one or more registers, by copying the data from the stack to the registers, then to add a value to the stack pointer (usually adding 4 times the number of registers to be popped on the stack)

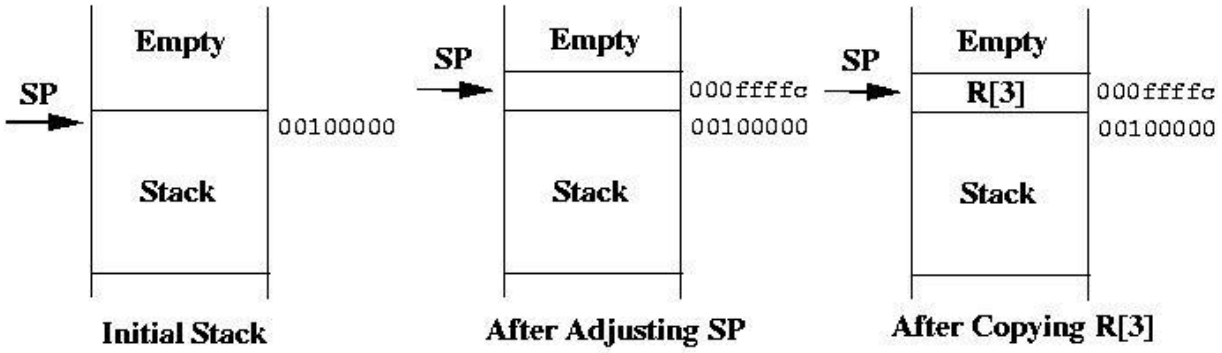
Thus, pushing is a way of saving the contents of the register to memory, and popping is a way of restoring the contents of the register from memory.

.

push: addi $sp, $sp, -4 # Decrement stack pointer
by 4 sw $r3, 0($sp) # Save $r3 to stack

The label "push" is unnecessary. It's there just to make it easier to tell it's a push. The code would work just fine without this label.

Here's a diagram of a push operation.

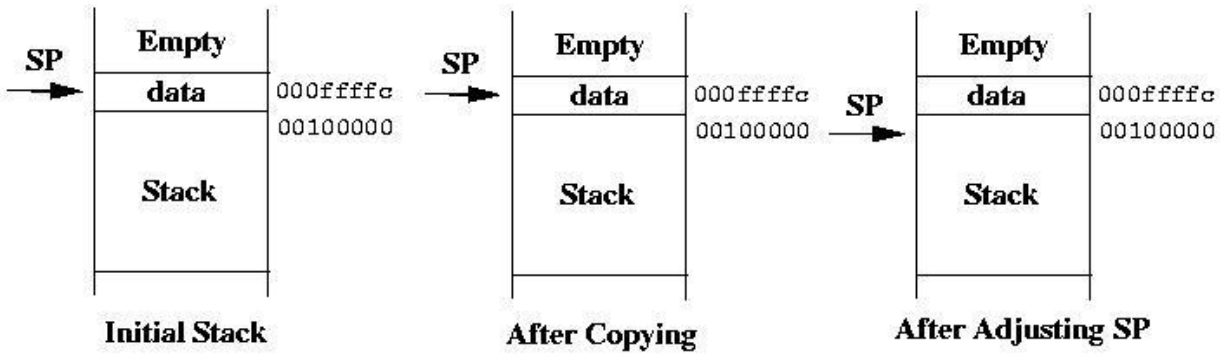


Popping off the stack

Popping off the stack is the opposite of pushing on the stack. First, you copy the data from the stack to the register, then you adjust the stack pointer.

pop: lw $r3, 0($sp) # Copy from stack to $r3
addi $sp, $sp, 4 # Increment stack pointer by 4

Here's a diagram of the pop operation.



Algorithm:

1. Input the number whose factorial is to be find from user
2. If input no=0 then display 1(0!=1)else go to step 3
3. Move input number into RAX register.
4. Compare RAX with 1 if equal go to step 5
5. POP contents of top of stack into RBX register(return address) and go to step 6
6. POP contents of top of stack into RBX register(value passed to function) and go to step 7
7. Multiply content of RBX and RAX.
8. Continue step 5 to 7 until stack becomes empty.
9. Display the final result by using display procedure.
10. Exit.

Conclusion: Thus, we have successfully studied how factorial is calculated by using recursive function.

Assignment No-11

Title: - ALP to plot Sine Wave, Cosine Wave and Sin function

Assignment Name: - Write 80387 ALP to plot Sine Wave, Cosine Wave and Sin function. Access video memory directly for plotting.

Objective-

- To understand trigonometric function

Outcome-

- Students will be able to access video memory directly for plotting.

Prerequisite–Linux system call

Hardware Requirement- Desktop PC

Software Requirement- Ubuntu 14.04,NASM

Introduction:-

A sine wave or sinusoid is a mathematical curve that describes a smooth epetitive oscillation. A sine wave is a continuous wave. It is named after the function sine, of which it is the graph. It occurs often in pure and applied mathematics, as well as physics, engineering, signal processing and many other fields. A cosine wave is a signal waveform with a shape identical to that of a sine wave , except each point on the cosine wave occurs exactly 1/4 cycle earlier than the corresponding point on the sine wave. A cosine wave and its corresponding sine wave have the same frequency, but the cosine wave leads the sine wave by 90 degrees of phase .

The sine function is one of the basic functions encountered in trigonometry (the others being the cosecant, cosine, cotangent, secant, and tangent). Let be an angle measured counter clockwise from the x-axis along an arc of the unit circle. Sine is an entire function and is implemented in the Wolfram Language as Sin[z].

Instructions:

FLDPI: (load pi") loads (pushes) pi" onto the NPX stack.

FDIV: Divide real

FMUL: Multiply Real

FSIN: When complete, this function replaces the contents of ST with SIN(ST). FSIN is equivalent to FCOS in the way it reduces the operand. ST is expressed in radians.

FLD: (load real) loads (pushes) the source operand onto the top of the register stack.

FBSTP: Packed decimal (BCD) store and pop

LEA: Load Effective Address

Algorithm for SIN Wave:-

Step 1: Display the message of Sin wave

Step 2: Initialize x-axis & y-axis for display.

Step 3: Constant represented in .data transfer into accumulator

Step 4: Transfer this data into data segment

Step 5: Set Video mode [mov ah,00]

Step 6: Push pi on FPU register stack

Step 7: Convert degree into radian (PI/180)

Step 8: (PI/180)*X

Step 9: Sin value will be generated

Step 10: (PI/180)*30

Step 11: Load floating point data to stack(fld hundred)

Step 12: Subtract top from top-1

Step 13: Store BCD integer and pop it

Step 14: Load Effective Address to register

Step 15: Create Graphics Pixel

Step 16: if not completed jump to step 6 else Step 17

Step 17: Display Graph

Step 18: [Finished] Stop

Algorithm for COS Wave:-

Step 1: Display the message of Sin wave

Step 2: Initialize x-axis & y-axis for display.

Step 3: Constant represented in .data transfer into accumulator

Step 4: Transfer this data into data segment

Step 5: Set Video mode [mov ah, 00]

Step 6: Push pi on FPU register stack

Step 7: Convert degree into radian (PI/180)

Step 8: (PI/180)*X

Step 9: COS value will be generated

Step 10: (PI/180)*30

Step 11: Load floating point data to stack(fld hundred)

Step 12: Subtract top from top-1

Step 13: Store BCD integer and pop it

Step 14: Load Effective Address to register

Step 15: Create Graphics Pixel

Step 16: if not completed jump to step 6 else Step 17

Step 17: Display Graph

Step 18: [Finished] Stop

Execution Steps:

For SIN Wave:

1. MOUNT c c:\Tasm
2. c:
3. tasm sin.asm
4. tlink sin
5. sin

For COS Wave:

1. MOUNT c c:\Tasm
2. c:
3. tasm cos.asm
4. tlink cos
5. cos

Conclusion:-

In this way we have implemented ALP to plot Sine Wave, Cosine Wave and Sin function. Access video memory directly for plotting.

References:- 1. Kenneth Ayala, "The 8086 Microprocessor: Programming & Interfacing the PC", CengageLearning,Indian Edition, 2008

2. RayDunkon, "Advanced MSDOS Programming", 2 nd Edition, BPB Publication.

Assignment No-12

Title: - Write ALP for performing Mean, Variance, and Standard Deviation.

Assignment Name: - 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.

Objective-

- To understand Concept of Mean, Variance and Standard Deviation

Outcome-

- Students will be able Execute the Mean, Variance and Standard Deviation

Prerequisite– Linux system calls

Hardware Requirement- Desktop PC

Software Requirement- Ubuntu 14.04, NASM

Theory:-

80X87 Architecture

The 80X87 is designed to operate concurrently with microprocessor. The 80X87 executes 68 different instructions. The microprocessor executes all normal instruction & 80X87 arithmetic coprocessor instructions. Both the microprocessor & coprocessor can execute their respective instructions simultaneously or concurrently. The numeric or arithmetic coprocessor is a special-purpose microprocessor i.e. especially designed to efficiently execute arithmetic & transcendental operations. The microprocessor intercepts & executes the normal instruction set, & coprocessor intercepts & executes only the co-processor instructions. The ESC instruction used by microprocessor, to generate a memory address for coprocessor so that coprocessor can execute a coprocessor instruction.

Internal Structure of The 80X87:

Figure shows internal structure of arithmetic coprocessor. It is divided into two major sections; control unit & numeric execution unit.

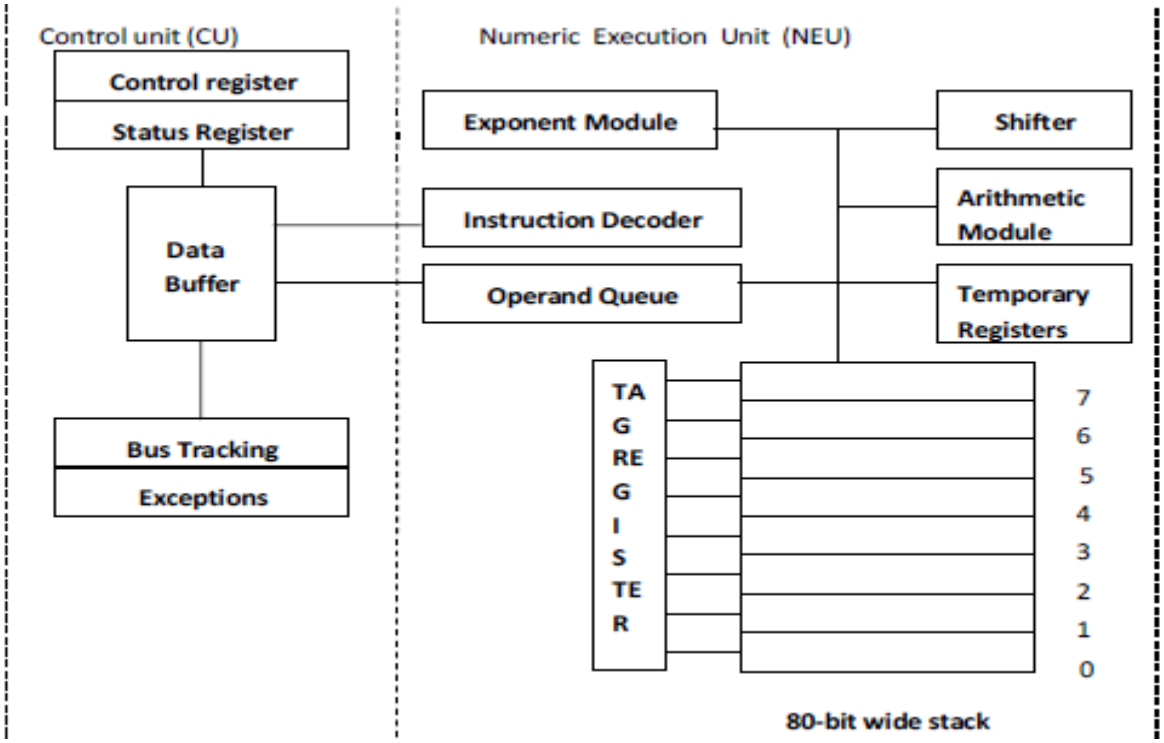
Control Unit:

It interfaces the coprocessor data bus. Both the devices monitor the instruction stream coprocessor) instruction, the coprocessor executes it; if not,

Numeric Execution Unit (NEU):

It is responsible for executing all coprocessor instructions. The NEU has an 8- register stack that holds operands for arithmetic instructions & the result of arithmetic instructions. Instructions either address data in specific stack data register or use a push & pop mechanism to store & retrieve data on the top of stack. Other registers in the NEU are status, control, tag & exception pointers. A few instructions transfer data between the coprocessor & the AX register in microprocessor. The FSTSW AX instruction is the only instruction available to coprocessor that allows direct communications to microprocessor through the AX register. 8087 does not contain FSTSW AX instruction. The stack within the coprocessor contains eight registers that are each 80 bits wide. These stack registers always contains an 80 bit extended precision floating pt. number. The only time that data appear as any other form is

when they reside in the memory system. The coprocessor converts from signed integer, BCD, single precision, or double precision form as data are moved between the memory & coprocessor register stack.



Instruction Set:

The arithmetic coprocessor executes over 68 different instructions. Whenever a coprocessor instruction references memory, the microprocessor automatically generates memory address for the instruction. The coprocessor uses the data bus for data transfer instruction exchanges the top of stack.

Instructions:

FINIT: Initialize Co-processor
FLDZ: Load zero on stack top
FILD: Load Integer on stack
FIDIV: Divide stack top by an integer value
FIMUL: Multiply stack top by an integer value
FST: Store stack top
FADD: Add in stack top
FBSTP: Store integer part of stack top in 10 byte packed BCD format
FMUL: Multiply stack top
FSQRT: Square Root of Stack Top
FSTSW: Stores the coprocessor status word
FTS: compares ST0 and 0
FSQRT: Contents of ST are replaced with its square root.
FABS: Replaces ST by its absolute value. Instruction simply makes sign positive.
FCHS: Complements the sign of the number in ST.

g) Instructions which Loads Constants

These instructions simply push the indicated constant onto the Stack.
FLDZ: Push 0.0 onto stack.
FLD1: Push +1.0 onto stack.
FLDPI: Push the value of pi(3.44) onto stack.
FLD2T: Push log of 10 to the base 2 onto stack(log2 10)
FLD2E: Push log of e to the base 2 onto stack (log2 e)
FLDG2: Push log of 2 to the base 10 onto stack (log10 2)

h) Coprocessor Control Instructions:
The coprocessor has to control instructions for initialization, exception handling, task switching. The control instructions have two forms.
FINIT/FNINIT: Performs a reset operation on the arithmetic coprocessor. It sets register 0 as the top of the stack.
FSETPM: Setup Coprocessor for addressing in protected mode .
FLDCW: Loads the control register operand.
FSTCW / FNSTCW: Stores control register operand into memory operand.

Algorithm

Step 1: Initialize 8087
Step 2: Make stack top zero
Step 3: Load and add each no to stack top
Step 4: Divide the total by n no of elements
Step 5: Store the average in mean
Step 6: Make stack top zero
Step 7: Load each no. To stack top, Subtract mean from the no.(xi-mean), Square the (ximean)
and add the contents
Step 8: Store the addition in temp1
Step 9: Load 1 on stack top
Step 10: Load n on stack top
Step 11: Find n-1
Step 12: Store n-1 in temp2
Step 13: Load n-1 on stack top
Step 14: Load sum of (xi-mean)2 on stack top
Step 15: Divide sum by n-1
Step 16: Store as variance
Step 17: Find the square root
Step 18: Store as standard deviation
Step 19: Display mean, variance and standard deviation
Step 20: Stop

Conclusion: - Thus, we have successfully implemented the assembly programme to find mean, variance and standard deviation.

References:- 1. Kenneth Ayala, "The 8086 Microprocessor: Programming & Interfacing the PC", CengageLearning,Indian Edition, 2008