|  |  |
| --- | --- |
| Sr. No. | INDEX |
| 1 | Bridge the Gap : Write X86/64 ALP to display Hello World using Macros. |
| 2 | Write X86/64 ALP to count number of positive and negative numbers from the array. |
| 3 | Write X86/64 ALP to perform non-overlapped and overlapped block transfer (with and without string specific instructions). Block containing data can be defined in the data segment. |
| 4 | Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for:  (a) HEX to BCD b) BCD to HEX (c) EXIT.  Display proper strings to prompt the user while accepting the input and displaying the result. (wherever necessary, use 64-bit registers) |
| 5 | Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. Accept input from the user. (use of 64-bit registers is expected) |
| 6 | 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. |
| 7 | Write X86/64 ALP to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers. |
| 8 | 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 |
| 9 | 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. |
| 10 | Write 80387 ALP to plot Sine Wave, Cosine Wave function. Access video memory directly for plotting. |
| 11 | Write x86 ALP to find the factorial of a given integer number on a command line by using recursion. Explicit stack manipulation is expected in the code. |
| 12 | 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. |
| 13 | 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. |

Assignment No. 1

|  |
| --- |
| **Title:** Bridge the Gap: Write X86/64 ALP to display Hello World using Macros. |
| **Learning Objective:** To acquaint the learner with application instruction set and logic to build assembly language programs. |
| **Learning Outcome:** To apply the assembly language programming to develop application programs in assembly using various instructions. |
| **Aim:** Write X86/64 ALP to display Hello World using Macros. |
| **Theory:**  **Assembly language**: Assembly language is a “low level” language and provides the basic instructional interface to the computer processor. Assembly language is as close to the processor as you can get as a programmer. Programs written in a high-level language are translated into assembly language in order for the processor to execute the program. The highlevel language is an abstraction between the language and the actual processor instructions.  Assembly language gives you direct control of the system's resources. This involves setting processor registers, accessing memory locations, and interfacing with other hardware elements. This requires a significantly deeper understanding of exactly how the processor and memory work.  **Macros:** An assembly language macro is a predefined set of instructions that can easily be inserted wherever needed. Once defined, the macro can be used as many times as necessary. It is useful when the same set of code must be utilized numerous times. A macro can be useful to reduce the amount of coding, streamline programs, and reduce errors from repetitive coding  Macro Definition Before using a multi-line macro, it must first be defined. The general format is as follows:  %macro <name> <number of argument>  ; [body of macro]  %endmacro  The arguments can be referenced within the macro by %, with %1 being the first argument, and %2 the second argument, and so forth. In order to use labels, the labels within the macro must be prefixing the label name with a %%  **System Call:** There are many operations that an application program must use the operating system to perform. Such operations include console output, keyboard input, file services (open, read, write, close, etc.), obtaining the time or date, requesting memory allocation, and many others.  Accessing system services is how the application requests that the operating system perform some specific operation (on behalf of the process). More specifically, the system call is the interface between an executing process and the operating system.  A system service call is logically similar to calling a function, where the function code is located within the operating system. The function may require privileges to operate which is why control must be transferred to the operating system. When calling system services, arguments are placed in the standard argument registers. System services do not typically use stack-based arguments. This limits the arguments of a system services to six (6), which does not present a significant limitation  If any are needed, the arguments for system services are placed in the rdi, rsi, rdx, r10, r8, and r9 registers (in that order). The following table shows the argument locations which are consistent with the standard calling convention    ConsoleOutput  Console Input  The arguments for the read system service are as follows: |
| **Algorithm/Pseudo code: N/A** |
| **S/W and H/W Required:** Fedora 20 O/S  NASM |
| **Input: N/A** |
| **Expected Output:** Hello World |
| **Steps for Execution:**  nasm -f elf64 test.asm  ld test.o -o test  . / test |
| **Conclusion :** Thu we have implemented an ASM program to display hello word by using Macro and understood how to write basic program in assembly language, and the steps required to execute the same. |

Assignment No. 2

|  |
| --- |
| **Title:** Write X86/64 ALP to count number of positive and negative numbers from the array. |
| **Learning Objective:** To acquaint the learner with application instruction set and logic to build assembly language programs. |
| **Learning Outcome:** To apply the assembly language programming to develop application programs in assembly using various instructions. |
| **Aim:** Write X86/64 ALP to count number of positive and negative numbers from the array. |
| Theory: Assembly - Basic Syntax An assembly program can be divided into three sections −   * The **data** section, * The **bss** section, and * The **text** section.  The *data* Section The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.  The syntax for declaring data section is −  section.data The *bss* Section The **bss** section is used for declaring variables. The syntax for declaring bss section is −  section.bss The *text* section The **text** section 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  \_start: Comments Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like −  ; This program displays a message on screen Allocating Storage Space for Initialized Data The syntax for storage allocation statement for initialized data is −  [variable-name] define-directive initial-value [,initial-value]...  There are five basic forms of the define directive –   Allocating Storage Space for Uninitialized Data The reserve directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved. Each define directive has a related reserve directive.  There are five basic forms of the reserve directive −    **Conditional Control Instructions**  The conditional control instructions provide a conditional jump based on a comparison. This provides the functionality of a basic IF statement.  The general form of the signed conditional instructions along with an explanatory comment are as follows:      **Data Movement**  Typically, data must be moved into a CPU register from RAM in order to be operated  upon. Once the calculations are completed, the result may be copied from the register  and placed into a variable. There are a number of simple formulas in the example  program that perform these steps. This basic data movement operation is performed  with the move instruction.  The general form of the move instruction is:  **mov <dest>, <src>** |
| **Algorithm/Pseudo code: N/A** |
| **S/W and H/W Required:** Fedora 20 O/S  NASM |
| **Input: N/A** |
| **Expected Output:**  Total Number of elements is array: 5  Total number of Negative: 2  Total number of positive: 3 |
| **Steps for Execution:**  nasm -f elf64 test.asm  ld test.o -o test  . / test |
| **Conclusion:** Thu we have implemented an ASM program to display number of positive and number of negative integers using CMP and Jump instructions. |

Assignment No. 3

|  |
| --- |
| **Title:** Write X86/64 ALP to perform non-overlapped and overlapped block transfer (with and without string specific instructions). Block containing data can be defined in the data segment. |
| **Learning Objective:** To acquaint the learner with application instruction set and logic to build assembly language programs. |
| **Learning Outcome:** To apply the assembly language programming to develop application programs in assembly using various instructions for block transfer from memory. |
| **Aim:** Write X86/64 ALP to perform non-overlapped and overlapped block transfer (with and without string specific instructions). Block containing data can be defined in the data segment. |
| **Theory:**  Non Over Lap Block Transfer   * Consider that a block of data of N bytes is present at source location. * Now this block of We know that source address is in the ESI register and destination address is in the EDI. * Let the number of bytes N = 05. * We will have to initialize this as count. * We know that source address is in the ESI register and destination address is in the EDI register. * For block transfer without string instruction, move contents at ESI to accumulator and from accumulator to memory location of EDI and increment ESI and EDI for next content transfer. * For block transfer with string instruction, clear the direction flag. Move the data from source location to the destination location using string instruction.   Over Lap Block Transfer   * Consider that a block of data of N bytes is present at source location. * Now this block of N bytes is to be moved from source location to a destination location. * Let the number of bytes N = 05. * We will have to initialize this as count. * Overlap the source block and destination block * We know that source address is in the ESI register and destination address is in the EDI * For block transfer without string instruction, move contents at ESI to accumulator and register. From accumulator to memory location of EDI and decrement ESI and EDI for next For block transfer with string instruction, set the direction flag. Move the data from source location to the destination location using string instruction.   Instructions needed:   * MOVSB-This is a string instruction and it moves string byte from source to destination. * REP- This is prefix that are applied to string operation. Each prefix cause the string instruction that follows to be repeated the number of times indicated in the count register. * CLD- Clear Direction flag. ESI and EDI will be incremented and DF = 0 * STD- Set Direction flag. ESI and EDI will be incremented and DF = 1 * ROL-Rotates bits of byte or word left. * AND-AND each bit in a byte or word with corresponding bit in another byte or word. * INC-Increments specified byte/word by1. * DEC-Decrements specified byte/word by1. * JNZ-Jumps if not equal to Zero. * JNC-Jumps if no carry is generated. * CMP-Compares to specified bytes or words. * JBE-Jumps if below or equal. * ADD-Adds specified byte to byte or word to word. * CALL-Transfers the control from calling program to procedure. * RET-Return from where call is made |
| **Algorithm/Pseudo code:**  Non Over Lap Block Transfer  1. Initialize ESI and EDI with source and destination address.  2. Move count in ECX register.  3. Move contents at ESI to accumulator and from accumulator to memory location of EDI.  4. Increment ESI and EDI to transfer next content.  5. Repeat procedure till count becomes zero  Over Lap Block Transfer  1. Initialize ESI and EDI with source and destination address.  2. Move count in ECX register.  3. Move source block’s and destination block’s last content address in ESI and EDI.  4. Move contents at ESI to accumulator and from accumulator to memory location of EDI.  5. Decrement ESI and EDI to transfer next content.  6. Repeat procedure till count becomes zero |
| **S/W and H/W Required:** Fedora 20 O/S  NASM |
| **Input: N/A** |
| **Expected Output:**  the source block is:  se computer  the destination block is:  se computer  the source block is:  se computer  the destination block is:  se secomputer |
| **Steps for Execution:**  nasm -f elf64 test.asm  ld test.o -o test  . / test |
| **Conclusion:** Thu we have implemented an ASM program to display number of positive and number of negative integers using CMP and Jump instructions. |

Assignment No. 5

|  |
| --- |
| **Title:** Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for:  (a) HEX to BCD b) BCD to HEX (c) EXIT.  Display proper strings to prompt the user while accepting the input and displaying the result. (wherever necessary, use 64-bit registers) |
| **Learning Objective:** To acquaint the learner with application instruction set and logic to build assembly language programs. |
| **Learning Outcome:** To apply the assembly language programming to develop application programs in assembly using various instructions.. |
| **Aim:** Write X86/64 ALP to convert 4-digit Hex number into its equivalent BCD number and 5-digit BCD number into its equivalent HEX number. Make your program user friendly to accept the choice from user for:  (a) HEX to BCD b) BCD to HEX (c) EXIT.  Display proper strings to prompt the user while accepting the input and displaying the result. (wherever necessary, use 64-bit registers) x |
| **Theory:**    Instruction required: MUL  The general form of the unsigned multiplication is as follows:  **mul <src>**  Where the source operand must be a register or memory location. An immediate  operand is not allowed.  For the single operand multiply instruction, the **A** register (**al**/**ax**/**eax/rax**) must be used  for one of the operands (**al** for 8-bits, **ax** for 16-bits, **eax** for 32-bits, and **rax** for 64-bit).  The other operand can be a memory location or register, but not an immediate.  Additionally, the result will be placed in the **A** and possibly **D** registers, based on the  sizes being multiplied. The following table shows the various options for the byte,  word, double-word, and quadword unsigned multiplications.    **Integer Division**  The division instruction divides two integer operands. Mathematically, there are special  rules for handling division of signed values. As such, different instructions are used for  unsigned division (**div**) and signed division (**idiv**).  Recall that  ***Dividend*** = ***quotient***  ***divisor***  Division requires that the dividend must be a larger size than the divisor. In order to  divide by an 8-bit divisor, the dividend must be 16-bits (i.e., the larger size). Similarly,  a 16-bit divisor requires a 32-bit dividend. And, a 32-bit divisor requires a 64-bit  dividend.  Like the multiplication, for most cases the integer division uses a combination of the **A**  and **D** registers. This pairing of registers is due to legacy support for previous earlier  versions of the architecture. While this helps ensure backwards compatibility, it can be  quite confusing.  Further, the **A**, and possibly the **D** register, must be used in combination for the  dividend.  • Byte Divide: **ax** for 16-bits  • Word Divide: **dx:ax** for 32-bits  • Double-word divide: **edx:eax** for 64-bits  • Quadword Divide: **rdx:rax** for 128-bits |
| **Algorithm/Pseudo code:**  **Conversion HEX to BCD**  Using DIV instruction FFFF/0AH  If Input number is FFFF  FFFF RAX- FFFF,EAX- 0A  Quotient-EAX Remainder –EDX  After each Pass   1. EAX- 1999H EDX-5 2. EAX=28F,EDX 3 3. EAX=41H,EDX=5 4. EAX=06H 5. EDX=5 6. EAX=0,EDX=6   Output=65535  **BCD to HEX Conversion-**  Initially inputnumber is 65535  BL=6, DX=0A,AX=0,MUL DX= 0\*6=0 ADD AL,BL=06  BL=5, DX=0A,AX=6,MUL DX= 6\*A=3C ADD AL,BL=3C+5=41H  BL=5, DX=0A,AX=41H ,MUL DX= 41H\*A=28A ADD AL,BL=28+5H=28F  BL=3, DX=0A,AX=28F ,MUL DX= 28F\*A=19996 ADD AL,BL=1996+3H=1999H  BL=5, DX=0A,AX=1999 ,MUL DX= 1999\*A=FFFA ADD AL,BL=FFFA+5H=FFFFH |
| **S/W and H/W Required:** Fedora 20 O/S  NASM |
| **Input: N/A** |
| **Expected Output:**  =====================================                  MENU                 ===================================== Enter your choice 1. Hex to BCD 2.BCD to hex 3.Exit 1 Enter the number 1234 4660 =====================================                  MENU                 ===================================== Enter your choice 1. Hex to BCD 2.BCD to hex 3.Exit 2 Enter the number 04660 1234 =====================================                  MENU                 ===================================== Enter your choice 1. Hex to BCD 2.BCD to hex 3.Exit 1 Enter the number AABB 43707 |
| **Steps for Execution:**  student@comp:~$ nasm -f elf64 third.asm student@comp:~$ ld -o third third.o  student@comp:~$ ./third |
| **Conclusion:** Thu we have implemented an ASM program to convert HEX to BCD and BCD to HECX numbers using MUL abd DIV instructions. |

Assignment No. 5

|  |
| --- |
| **Title:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. Accept input from the user. (use of 64-bit registers is expected) |
| **Learning Objective:** To acquaint the learner with application instruction set and logic to build assembly language programs. |
| **Learning Outcome:** To apply the assembly language programming to develop application programs in assembly using various instructions.. |
| **Aim:** Write X86/64 ALP to perform multiplication of two 8-bit hexadecimal numbers. Use successive addition and add and shift method. Accept input from the user. (use of 64-bit registers is expected) |
| **Theory:**  **Successive Addition Multiplication**  Program should take first number and counter as input for the program and after implementing given below logic it should produce desired result .  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 H,  BL = 10 H                                  Result = 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H + 12H                                  Result = 0120 H  **Shift-and-Add Multiplication**  Shift-and-add multiplication is similar to the multiplication performed by paper and pencil. This method adds the multiplicand *X* to itself *Y* times, where *Y* denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results. As an example, consider the multiplication of two unsigned 4-bit numbers, 8 (1000) and 9 (1001).    The 2*n*-bit product register (*A*) is initialized to 0. Since the basic algorithm shifts the multiplicand register (*B*) left one position each step to align the multiplicand with the sum being accumulated in the product register, we use a 2*n*-bit multiplicand register with the multiplicand placed in the right half of the register and with 0 in the left half. |
| **Algorithm/Pseudo code:** Algorithm  to Multiply Two 8 Bit Numbers  Successive Addition Method Step I          :   Initialize the data segment.Step II        :   Get the first number. Step III      :   Get the second number as counter.  Step IV       :   Initialize result = 0.  Step V        :   Result = Result + First number.  Step VI       :   Decrement counter  Step VII     :   If count ¹ 0, go to step V.  Step VIII   :   Display the result.  Step IX      :   Stop.  Multiply example using the Add and shift algorithm. |
| **S/W and H/W Required:** Fedora 20 O/S  NASM |
| **Input:** Enter First 8 bit number: 02  Enter second 8 bit number: 04 |
| **Expected Output:**  Multiplication of two HEX 8 bit number is 0008 |
| **Steps for Execution:**  student@comp:~$ nasm -f elf64 third.asm student@comp:~$ ld -o third third.o  student@comp:~$ ./third |
| **Conclusion:** Thu we have implemented an ASM program to multiply two numbers using successive addition and add and shift method. |

**Assignment No: 7**

**Problem Statement:**

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

**Theory:**

**Bubble Sort** is a simple algorithm which is used to sort a given set of ‘n’ elements provided in form of an array with ‘n’ number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

If we have total n elements, then we need to repeat this process for n-1 times.

**Algorithm:**

1. Start.
2. Initialize the data segment.
3. Initialize the number of elements counter.
4. Initialize the number of comparison counter.
5. Compare the element, if first element is less than second element, go to step 9

Otherwise go to step 6.

1. Swap the elements.
2. Decrement the comparison counter.
3. If count = 0, go to step 9, otherwise we go to step 5.
4. Insert the element in proper position.
5. Increment the number of element counter.
6. If count = n, go to step 12, otherwise go to step 3.
7. Display the result.
8. Stop.

**Expected Input:**

Text file having list of unsorted integer.

**Expected Output:**

Text file having list of sorted integer.

**Conclusion:**

Hence the ALP to sort the list of integers in ascending order using bubble sort .

**Assignment No: 8**

**Problem Statement:**

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.

**Theory:**

**TYPE command:**

**TYPE** it displays the content of saved file;

e.g: C:\> Type> file name.

**COPY command:**

It copies the given file or files from the source directory to the largest directory;

Syntax: C:\> copy<source file name> <target file name>.

**DELETE command:**

Delete a file from current directory;

Syntax: C;\> del<file name>

**Algorithm:**

1. Start.
2. Display the menu and accept the choice from user.
3. Move byte [choice] to al.
4. Move 30h to al.
5. Compare 01h with [al]. if not equal then go to step 8.
6. Call type command procedure.
7. Jump to step 15.
8. Compare 02h with [al]. if not equal go to step 11.
9. Call copy command procedure.
10. Jump to step 15.
11. Compare 03h with [al]. if not equal go to step 14.
12. Call delete command procedure.
13. Jump to step 15.
14. Call exit procedure.
15. Stop.

**Type procedure:**

1. Accept the file name from the user.
2. Decrement [rax].
3. Move 00h to byte [file name + rax].
4. Open file.
5. Compare -1h with contents of rax register. If less or equal, then display error message, Otherwise move contents of rax register to [fhandle].
6. Reads contents of in file.
7. Decrement [rax].
8. Move [rax] to [act\_len].
9. Return.

**Copy command:**

1. Accept file name from which you want to copy the command.
2. Decrement the contents of rax register.
3. Move 00h to byte [filename + rax].
4. Accept file name into which data to copied.
5. Decrement [rax].
6. Move 00h to byte [filename + rax]
7. Open file 2.
8. Compare -1h with [rax], if less or equal then display “file not found”.
9. Move [rax] to [filehandle].
10. Open file 2.
11. Compare -1h with [rax], if less or equal then display “file not found”.
12. Move [rax] to [filehandle].
13. Write the data in file 2.
14. Display message “file successfully copied”.
15. Close file.
16. Return.

**Delete command:**

1. Accept file name which want to delete.
2. Decrement [rax].
3. Move 00h to byte [filehandle +rax]
4. Move 87 to rax register.
5. Move filename to rdi register.
6. Compare -1h with [rax], if less or equal then display “file not found”.
7. Return**.**

**Expected Input:**

File on which operation to be perform.

**Expected Output:**

File after performing different operation i.e. TYPE, COPY, DELETE.

**Conclusion:**

Assembly level program to implement OS (DOS) commands TYPE, COPY and DELETE using file operations is executed successfully.

**Assignment No: 9**

**Problem Statement:**

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.

**Theory:**

For an integer n greater than or equal to 1, the factorial is the product of all integers less than or equal to n but greater than or equal to 1. The factorial value of 0 is defined as equal to 1. The factorial values for negative integers are not defined.

**Algorithm:**

1. Start.
2. Take the input 8 bit number from user.
3. Call accept procedure.
4. Xor contents of rax register.
5. Move contents of bx to ax register.
6. Compare 01h with contents of ax, if below or equal go to step 10.
7. Call factorial procedure.
8. Call display procedure.
9. Display message “factorial of 1 and zero :” using display macro.
10. Display 0001h .
11. Move 60h to rax register.
12. Move 0 to rdi.
13. Stop.

**Accept procedure:**

1. Move contents of num to rsi register.
2. Move 04 to cl register.
3. Xor the contents of rbx.
4. Move 02 to ch register.
5. Compare 39h with byte[rsi]. If not greater go to step 7.
6. Subtract 07h from byte[rsi].
7. Subtract 30h from byte[rsi].
8. Rotate left content of bl by cl times.
9. Add contents of rsi with bl register.
10. Increment rsi.
11. Decrement cl.
12. If not zero then return.

**Factorial procedure:**

1. Xor contents of rbx.
2. Move rax register contents to rbx.
3. Subtract 01 from rbx register.
4. Multiply contents of rbx with rax register.
5. Compare 01h with rbx contents if not equal then go to step 3.
6. Return.

**Display procedure:**

1. Move results to rsi register AND 16h to ch register.
2. Move 04 to cl.
3. Rotate left contents of rax by 4 times.
4. Move contents of al in bl.
5. Perform anding of 0fh with bl register.
6. Compare 09h with contents of bl if not greater then go to step 8.
7. Add 07h to bl register.
8. Add 30h to bl register.
9. Move b contents to rsi.
10. Increment rsi.
11. Decrement cl.
12. If not zero, go to 3.
13. Display.
14. Return.

**Expected Input:**

Enter the number to find factorial of that number.

**Expected Output:**

Factorial of number.

**Conclusion:**

An ALP to find factorial of 8 bit number is executed successfully**.**

**Assignment No: 10**

**Problem Statement:**

Write 80387 ALP to plot Sine Wave, Cosine Wave and Sinc function. Access video memory directly for plotting.

**Theory:**

**CALCULATION OF SINE WAVE:** Y = 50– 30 sin ({pi/180}\*x)

**CALCULATION OF COSINE WAVE:** Y = 50 – 30 cos ({pi/180}\*x)

**Instruction of co-processor used in the assignment: (Explain it by your own)**

FIMUL:

FSIN:

FLDPI:

FIDIV:

FSUB:

FILD:

FISTP:

**Algorithm:**

**Algorithm for Plotting Sine wave:**

1. Start
2. Initialize x=0.0, xadd=3.0, one80=180.0,count=360,x1=0,ycursor=0 in data segment.
3. Initialize DS (needed for .exe-program) mov ax,@data and mov ds, ax
4. Move address of video memory 0A000H in AX and the Move AX in ES
5. Select Desired video mode Here we selected mode 6H (640x200 256 color graphics (MCGA,VGA)). For this Move AX=6H [or AH=00 (Set Video mode) and AL=6H (Video mode)] and then call int 10H BIOS interrupt to enter in 6H mode
6. Initialize CX with zero (X=0 i.e. angle)
7. Push CX
8. Call get\_sine procedure
9. Move CX in BX
10. Call vector\_to\_memory procedure
11. Move AX in DI (Offset for pixel)
12. Move color of pixel (0-255) in AL
13. Put pixel directly in video RAM by Moving AL at the address given by ES:DI
14. Pop CX
15. Increment CX
16. Compare CX with 360 (right boarder of screen) If not equal then go to step vii else continue
17. Display message “Sine Wave:” by passing 09h in AH, address of message in DX and call int 21h
18. For Exit, Move 4CH in AH and then call interrupt 21 H
19. Stop

**Algorithm for Plotting Cosine wave:**

1. Start
2. Initialize x=0.0, xadd=3.0, one80=180.0,count=360,x1=0,ycursor=0 in data segment.
3. Initialize DS (needed for .exe-program) mov ax,@data and mov ds, ax
4. Move address of video memory 0A000H in AX and the Move AX in ES
5. Select Desired video mode Here we selected mode 6H (640x200 256 color graphics (MCGA,VGA)). For this Move AX=6H [or AH=00 Set Video mode) and AL=6H (Video mode)] and then call int 10H BIOS interrupt to enter in 16H mode
6. Initialize CX with zero (X=0 i.e. angle)
7. Push CX
8. Call get\_cos procedure
9. Move CX in BX
10. Call vector\_to\_memory procedure
11. Move AX in DI (Offset for pixel)
12. Move color of pixel (0-255) in AL
13. Put pixel directly in video RAM by Moving AL at the address given by ES:DI
14. Pop CX
15. Increment CX
16. Compare CX with 360 (right boarder of screen) If not equal then go to step vii else continue
17. Display message “Cos Wave:” by passing 09h in AH, address of message in DX and call int 21h
18. For Exit, Move 4CH in AH and then call interrupt 21 H Stop

**Expected Input:**

Angle values

**Expected Output:**

Display Sine wave and Cos wave.

**Conclusion:**

80387 Assembly Level Program to plot Sine Wave, Cosine Wave is assembled and executed successfully.

**Assignment and Oral Questions:**

Q1.**Explain the following instruction with one example of each:**

FMUL:

FSIN:

FLDPI:

FDIV:

FCOS:

FBSTP:

Q2. **Explain the following Interrupt with example**

**INT 10h** / **AH = 0**

**INT 10h / AH=0CH**

**INT 21h** / **AH=9** - output of a string at **DS:DX**. String must be terminated by

**Assignment No: 11**

**Problem Statement:**

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.

**Theory:**

**Mean:-**

The mean or average of a set of data represents the characteristic nature or central tendency of those numbers.This is the figure you would expect to occur most often over the long run.

For example, for the numbers 4, 5, and 6

mean (average) = (4+5+6)/3 = 5.

**Variance:**

The variance is a measure of how spread out those numbers are among each other. The range between the highest and lowest numbers would indicate variation. Generally, the smaller the range the lower is the calculated variance; the larger the range the higher will be the calculated variance.

The variance is calculated as the average squared difference between each number and the mean. For example, for the numbers 4, 5, and 6

variance = ((4-5)² + (5-5)² + (6-5)²) / 3 = 0.67

**Standard Deviation:**

The standard deviation is the square root of the variance.

standard deviation = square root of (0.67)= 0.8

**Program instruction:**

* + 1. FADDP
    2. FDIVP
    3. FMULP
    4. FSUB
    5. FLD
    6. FST
    7. FIST
    8. FISTP
    9. FRNDINT
    10. FSQRT

**Algorithm:**

**Expected Input:**

Array of floating point values

**Expected Output:**

From given values it calculates Mean, Variance and standard deviation

**Conclusion:**

The i) Mean ii) Variance iii) Standard Deviation are caluculated for the given floating point values using 80387 instructions .

**Assignment and Oral Questions:**

Q1 Explain following instruction with example in detail

* + 1. FADDP
    2. FDIVP
    3. FMULP
    4. FSUB
    5. FLD
    6. FST
    7. FIST
    8. FISTP
    9. FRNDINT
    10. FSQRT

Q2 Explain 80387 Data Type

Q3. Explain and draw Instruction Queue, Data Registers of 80387 in detail.

Q4 Draw Interfacing diagram of 80387 with 80386

**Assignment No: 12**

**Problem Statement:**

Write a TSR to generate the pattern of the frequency tones by reading the Real Time Clock (RTC). The duration of the each tone is solely decided by the programmer.

**Theory:**

**Terminate but Stay Resident (TSR):**

Terminate execution of the currently executing program , passing a return code to the parent process, but reserves part or all of the program’s memory so that it will not be overlaid by the next transient program to be loaded.

**Transient Program**

* Programs that run under MS-DOS comes into two basic flavours.

1. .COM Programs
2. .EXE Programs

These programs also referred as transient programs (Short term, Temporary)

* Transient program “Owns” the memory block it has never been allocated and has nearly total control of the system’s resources while executing.
* When program terminates either because of aborted by the OS or completed work and systematically performed final exit back to MS-DOS , then the memory block is freed (so called as transient) and can be used by the next program .

**Resident Program**

* Programs are reside on memory while other programs are running and we can activate their service through special keystroke.
* We can load the resident programs after DOS is loaded and before activating other normal processing programs .
* They are almost always .COM programs and are also known as **Terminate and Stay Resident** (TSR) Programs.

**TSR Programs**

* These programs start when you first turn on your computer and stay in the memory. Ready for use, even if they are not active on your screen.
* They appear to exit, but remain in the memory to perform tasks in the background.
* TSR programs typically include Clock, Screen Savers, Calculators.
* When running another program in DOS you can press the preset keyboard key or combination of keys and the TSR programs will “POP-UP” into view.
* Most of TSR programs did not have option for unloading themselves from memory, so loading TSR meant it would stay until a reboot

**Structure of TSR**

* TSR Program is divided into Three sections.

1. Initialization Routine
2. Data Area
3. Resident Routine

**Initialization Routine**

* Steps to be performed in the initialization routine

1. Get Interrupt Vector
2. Set Interrupt Vector
3. Terminate and stay Resident

Values of Interrupt

Timer Interrupt (INT 08H)

Keyboard Interrupt (INT 09H**)**

**Interrupts and functions**

Int 21H Function 35H (53): Get interrupt vector

Int 21H Function 25H (37): Set interrupt vector

Int 21H Function 31H (49): Terminate and stay resident

**Data Area**

* We know that before replacing the entry in interrupt vector table, we have stored the original address. To store this 32 bit address (16 bit segment + 16 bit offset) we have reserve 32 bit space in the data area.
* The data area may include additional data definitions as per program requirements**.**

**Resident Routine**

* Whenever the TSR is activated by any of the interrupt, the program control is first transferred to the resident program and after execution of resident program, the program control is transferred to the original ISR of the requested interrupt.
* In this process, the resident program uses CPU registers and may change their contents, which is not desired and execution of ISR may give wrong results because of changed contents of registers.
* Therefore it is necessary to preserve the contents of registers. To execute desired task.
* Restore original contents of CPU registers .
* Finally resident program should have unconditional jump instruction with the address of original interrupt service routine.

**Expected Output:**

Program reads the system Real Time Clock (RTC) and it will display on the screen.

**Conclusion:**

Terminate and stay resident Program reads the system Real Time Clock (RTC) and it will display on the screen.

**Assignment and Oral Questions:**

1. Explain the sections of TSR program
2. Explain the following functions and Interrupts of TSR program

Int 21H Function 35H (53)

Int 21H Function 25H (37)

Int 21H Function 31H (49)