Vision and Mission of the CSE Department

Vision

The department of Computer Science Engineering aims to impart **progressive education** with **state-of-art curriculum** and generate **socially sensitized** engineers, **innovators, and entrepreneurs** for **sustainable development.**

Dept Vision Keywords	Dept Mission Phrases
progressive education	M1, M2
state-of-art curriculum	M1, M2
socially sensitivity	M3, M4
innovation, and entrepreneurship	M2, M3
sustainable development	M3

Mission

- To pioneer the education in Computer Science and Engineering with dynamic and industry ready curriculum.
- To evolve as a Centre of Excellence that contributes significantly to multi- disciplinary research, innovation and entrepreneurship in collaboration with industry and academia.
- To encourage students and faculties by engaging in multi-disciplinary problem solving and creating sustainable solutions for social well-being.
- To nurture students with ethics and values through holistic education for developing responsible leadership.

Mission-PO Mapping

a. Knowledge base	Demonstrated competence in university level mathematics, natural sciences, engineering fundamentals, and specialized engineering knowledge appropriate to the program.	dynamic and industry ready curriculum(M1) Centre of excellence(M2)
b. Problem analysis	An ability to use appropriate knowledge and skills to identify, formulate, analyze, and solve complex engineering problems in order to reach substantiated conclusions.	contributes significantly to multi-disciplinary research(M2) inter-disciplinary problem solving and creating sustainable solutions (M3)
c. Investigation	An ability to conduct investigations of complex problems by methods that includeappropriate experiments, analysisand interpretation of data, and synthesis of information in order to reach valid conclusions.	contributes significantly to multi-disciplinary research(M2)
d. Design	An ability to design solutions for complex, open-ended engineering problems and to design systems, components or processes that meet specified needs with appropriate attention to health and safety risks, applicable standards, and economic, environmental, cultural and societal considerations.	contributes significantly to multi-disciplinary research(M2) engaging in inter-disciplinary problem solving and creating sustainable solutions(M3)
e. Use of engineering tools	An ability to create, select, apply, adapt, and extend appropriate techniques, resources, and modern engineering tools to a range of engineering activities, from simple to complex, with an understanding of the associated limitations.	Computer Science and Engineering with dynamic and industry ready curriculum(M1)

f. Individual and team work	An ability to work effectively as a member and leader in teams, preferably in a multi-disciplinary setting.	holistic education for developing responsible leadership(M4)
g. Communication skills	An ability to communicate complex engineering concepts within the profession and with society at large. Such ability includes reading, writing, speaking and listening, and the ability to comprehend and write effective reports and design documentation, and to give and effectively respond to clear instructions.	holistic education for developing responsible leadership(M4)
g. Professionalism	An understanding of the roles and responsibilities of the professional engineer in society, especially the primary role of protection of the public and the public interest.	holistic education for developing responsible leadership(M4) creating sustainable solutions for social well-being(M3)
i. Impact of engineering on society and the environment	An ability to analyze social and environmental aspects of engineering activities. Such ability includes an understanding of the interactions that engineering has with the economic, social, health, safety, legal, and cultural aspects of society, the uncertainties in the prediction of such interactions; and the concepts of sustainable design and development and environmental stewardship.	inter-disciplinary problem solving and creating sustainable solutions for social well-being(M3)
j. Ethics and Equity	An ability to apply professional ethics, accountability and equity.	ethics and values through holistic education for developing responsible leadership. (M4)

k. Economics and project management	An ability to appropriately incorporate economics and business practices including project, risk, and change management into the practice of engineering and to understand their limitations.	sustainable solutions for social well-being(M3) inter-disciplinary problem solving(M3)
l. Life-long learning	An ability to identify and to address their own educational needs in a changing world in wayssufficient to maintain their competence and to allow them to contribute to the advancement of knowledge.	dynamic and industry ready curriculum(M1) multi-disciplinary research, innovation and entrepreneurship(M2) creating sustainable solutions(M3) holistic education for developing responsible leadership(M4)

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1

To apply the acquired knowledge of Computer Science & Engineering to build sustainable solutions to real-world problems in society.

PEO2

Attain the ability to adapt quickly to new environments and technologies, assimilate new information, and work in multi-disciplinary areas with a strong focus on higher studies, innovation and entrepreneurship.

PEO3

To practice collaborative learning and team spirit as professionals with project-based learning and technical activities.

PEO4

To be able to effectively present, communicate and exhibit professionalism with ethical values and empathy for needs of society.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1

Apply fundamental knowledge in Computer Science and Engineering with advanced learning in specialized domains of Data Science, Artificial Intelligence, Networking, Security and high-performance computing using standard practices in Engineering and Sciences.

PSO2

Apply standard industry-based practices and techniques to provide solutions to real world problems.

MIT Art, Design and Technology, Pune
Department of Computer Engineering

Name: _____ **Date of Performance**: _ _ /_ _/20_ _

Class: S.Y. Computer **Date of Completion:** _ _ /_ _/20_ _

Div: _ _ _, **Batch**: _ _ _ _ _

Roll No: _ _ _ _ _

Experiment No. 0

TITLE: Understand of the Programming Toolchain

SUB-EXPERIMENT No: 0.1

AIM: Installation of Ubuntu and establishing toolchain of NASM.

OBJECTIVE:

1. To study software development tools required, and commands to use them.
2. To study the steps of installing Linux-based OS.
3. To study different Linux commands.

Tasks to Perform:

1. Download VMWare Player
2. Download UBUNTU18.04 LTS OS ISO
3. Install VMWare
    a. Give a name to OS
    b. Select Folder to Install OS
    c. Set Space & RAM size for OS
    d. Keep other settings as default
4. Select UBUNTU18.04 LTS OS ISO to install on virtual space

5. While installing OS: set user as: user & pwd: user
6. Once done with OS installation, just run the virtual machine.

- In Ubuntu OS: just verify internet connectivity

7. Now, open the terminal with (cntrl+shift+T)

STEPS TO INSTALL NASM in UBUNTU - LINUX OS

a. sudo apt-get update
b. sudo apt-get -y install nasm

STEPS TO RUN NASM in UBUNTU - LINUX OS - 32 BIT

- Following steps for compiling and linking the above program:
- Type the above code using a text editor and save it as hello.asm.
- Make sure that you are in the same directory as where you saved hello.asm.
- To assemble the program, type nasm -f elf hello.asm hello.lst
- If there is any error, you will be prompted about that at this stage.
- Otherwise an object file of your program named hello.o will be created.
- ld -o hello hello.o To link the object file and create an executable
- To link the object file and create an executable file named hello, type ld -m elf_i386 -s -o hello hello.o
- Execute the program by typing ./hello If you have done everything correctly, it will display **Hello, world!** on the screen.

STEPS TO RUN NASM in UBUNTU - LINUX OS - 64 BIT

- Following steps for compiling and linking the above program:
- Type the above code using a text editor and save it as hello.asm.
- Make sure that you are in the same directory as where you saved hello.asm.
- To assemble the program, type nasm -f elf64 hello.asm
- If there is any error, you will be prompted about that at this stage.
- Otherwise an object file of your program named hello.o will be created.
- ld -o hello hello.o To link the object file and create an executable
- To link the object file and create an executable file named hello, type ld -m elf64 -s -o hello hello.o
- Execute the program by typing ./hello If you have done everything correctly, it will display **Hello, world!** on the screen.

Experimentation Work:

1. Students can perform the installation on different OS platform
2. Change the size of RAM / ROM see the performance variations
3. Read the Shell Commands and note down few important one's.

CONCLUSION: In conclusion, the experiment involved the installation of Ubuntu and NASM, as well as a thorough evaluation of their respective toolchains. Throughout the process, we successfully executed the necessary steps to install Ubuntu, ensuring a stable and functional operating system for further experimentation. Additionally, we installed NASM and verified its toolchain, which proved to be a powerful and versatile assembler for x86 and x86-64 architectures.

Experiment No. 0

TITLE: Understand of the Programming Toolchain

SUB-EXPERIMENT No: 0.2

AIM: Understanding of Assembly Programming basics.

OBJECTIVE:

1. To study basics of assembly language programming i.e. Software development tools required, commands to use them, format of ALP, assembler directives.
2. To study steps in assembly language programming.
3. To execute simple assembly programs

Theory:

The assembly programming language is a low-level language which is developed by using mnemonics. The microcontroller or microprocessor can understand only the binary language like 0's or 1's therefore the assembler convert the assembly language to binary language and store it the memory to perform the tasks. Before writing the program the embedded designers must have sufficient knowledge on particular hardware of the controller or processor, so first we required to know hardware of 8086 processor.

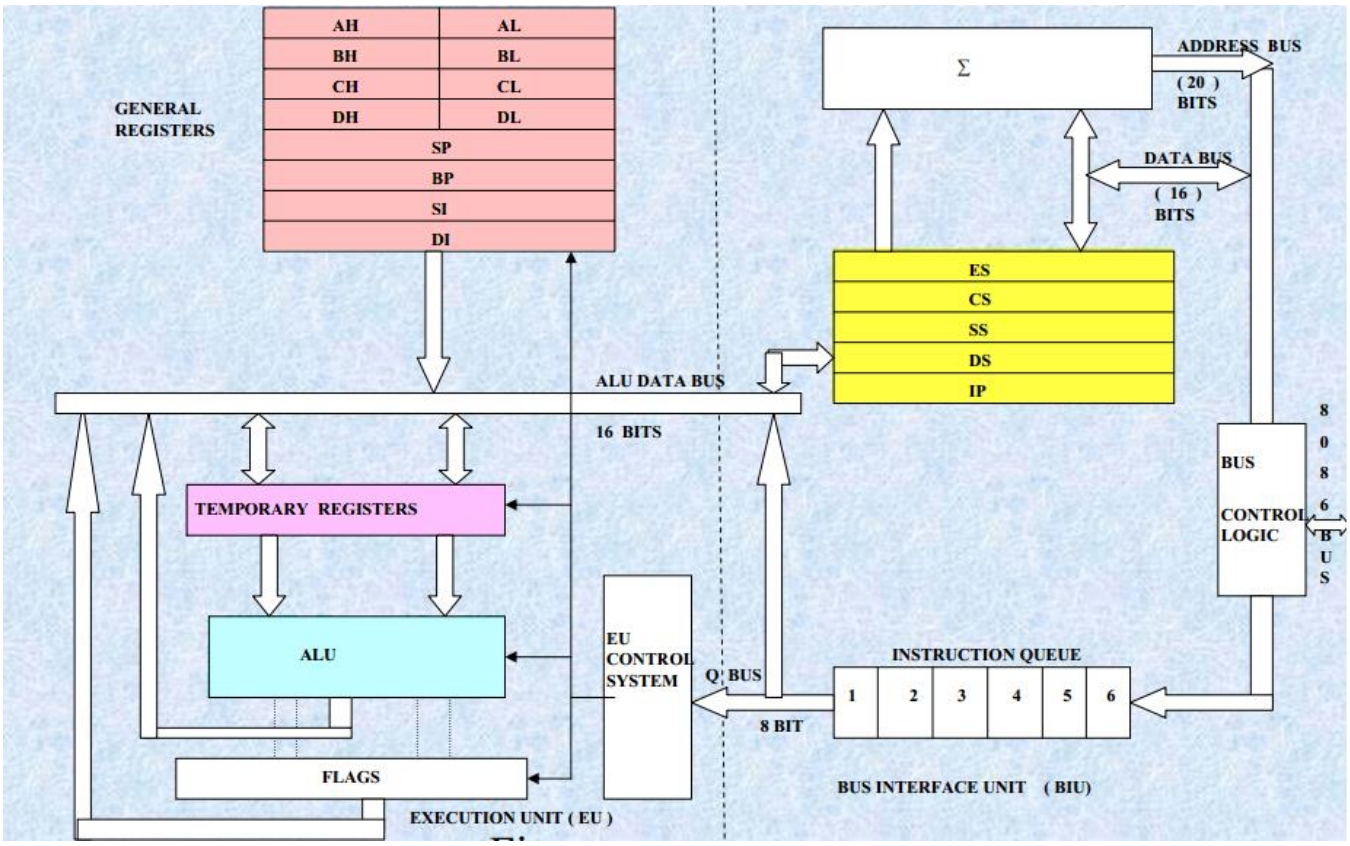
General purpose registers: The 8086 CPU has consisted 8-general purpose registers and each register has its own name as shown in the figure such as AX, BX, CX, DX, SI,DI, BP, SP . These all are 16-bit registers where four registers are divided into two parts such as AX, BX, CX, and DX which is mainly used to keep the numbers.

Special purpose registers: The 8086 CPU has consisted 2- special function registers such as IP and flag registers. The IP register point to the current executing instruction and always works to gather with the CS segment register. The main function of flag registers is to modify the CPU operations after mechanical functions are completed and we cannot access directly

The 8086 processor operates in a segmented memory model, where memory is divided into different segments, each with its own starting address and size.

- **Code Segment (CS):** The code segment contains the program instructions. It is used to fetch instructions during program execution.
- **Data Segment (DS):** The data segment holds data and variables used by the program. It is utilized for reading and writing data.
- **Stack Segment (SS):** The stack segment is responsible for managing the program stack. It stores local variables, function call information, and return addresses.

- **Extra Segment (ES):** The extra segment is an additional segment that can be used for storing additional data or accessing data in memory beyond the data segment.



Segment registers: The 8086 CPU has consisted 4- segment registers such as CS, DS, ES, SS which is mainly used for possible to store any data in the segment registers and we can access a block of memory using segment registers.

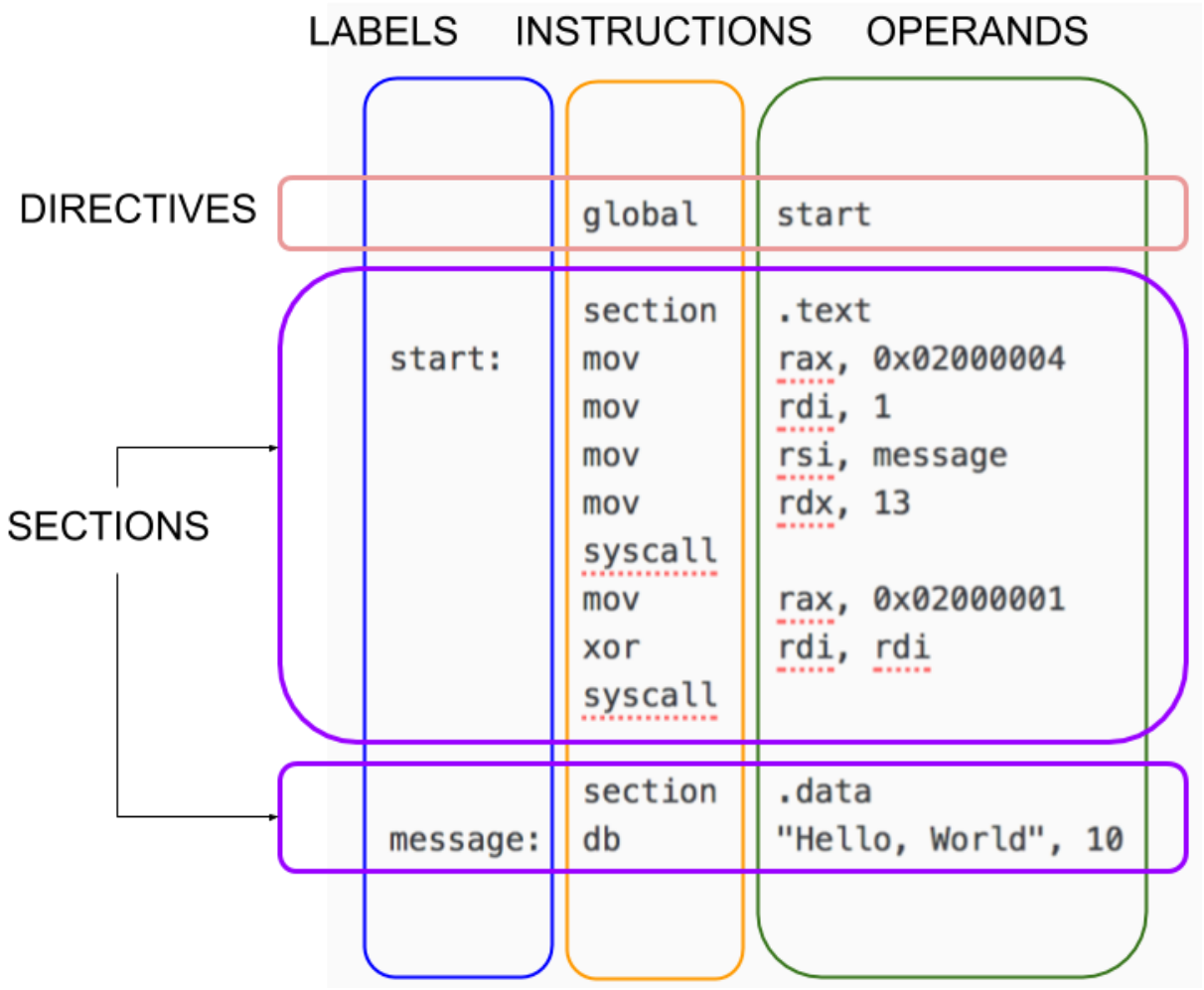
The assembly language programming 8086 has some rules such as

- The assembly-level programming 8086 code must written in upper-case letters
- The labels must be followed by a colon, for example: label:
- All labels and symbols must begin with a letter
- All comments are typed in lowercase
- The last line of the program must be ended with the END directive

8086 processors have two other instructions to access the data, such as WORD PTR – for word (two bytes), BYTE PTR – for byte.

- Op-code: A single instruction is called as an op-code that can be executed by the CPU. Here the 'MOV' instruction is called as an op-code.
- Operands: A single piece data are called operands that can be operated by the op-code. Example, subtraction operation is performed by the operands that are subtracted by the operand.
- **Syntax: SUB b, c**

Structure of a NASM Program



Experimentation Work:

1. Student can perform simple programs using NASM

Sample Code : Hello World

```
section .data
msg db 'Hello, world!'          ;string to be printed
len: equ  $ - msg               ;length of the string
section .text
      global _start             ;must be declared for linker (ld)
_start:                         ;tells linker entry point
mov eax,4                       ;invoke sys.write
mov ebx,1                       ;write (stdout)
mov edx,len                     ;message length
mov ecx,msg                     ;message to write
mov ebx,1                       ;file descriptor (stdout)
mov eax,4                       ;system call number (sys_write)
 int 0x80                       ;call kernel
mov eax,1                       ;system call number (sys_exit)
mov ebx,0                       ;returns zero if no errors
int 0x80                        ;call kernel
```

CONCLUSION: In conclusion, the experiment involved the installation of Ubuntu and NASM, as well as a thorough evaluation of their respective toolchains. Throughout the process, we successfully executed the necessary steps to install Ubuntu, ensuring a stable and functional operating system for further experimentation. Additionally, we installed NASM and verified its toolchain, which proved to be a powerful and versatile assembler for x86 and x86-64 architectures.

Experiment No. 0

TITLE: Understand of the Programming Toolchain

SUB-EXPERIMENT No: 0.3

AIM: Understanding of System calls.

OBJECTIVE:

1. To study basics of assembly language programming i.e. Software development tools required, commands to use them, format of ALP, assembler directives.
2. To study steps in assembly language programming
3. To study different System calls

Theory:

In NASM (Netwide Assembler), system calls are used to interact with the operating system and perform various tasks such as file operations, input/output operations, process management, and more. System calls provide a standardized interface for user programs to communicate with the underlying operating system kernel.

The specific system calls available in NASM may vary depending on the target operating system, as different operating systems have their own set of system calls. However, there are some commonly used system calls that are supported by many operating systems. Here are a few examples:

- **exit:** Terminates the current process and returns an exit status to the operating system.
- **read:** Reads data from a file descriptor into a buffer.
- **write:** Writes data from a buffer to a file descriptor.
- **open:** Opens a file and returns a file descriptor.
- **close:** Closes a file descriptor.
- **fork:** Creates a new process by duplicating the existing process.
- **execve:** Executes a new program, replacing the current process.
- **wait:** Waits for a child process to terminate and retrieves its termination status.

To invoke a system call in NASM, you typically need to set up the appropriate registers with the required parameters and then use a specific instruction, such as int 0x80 (for x86) or syscall (for x86-64), to trigger the system call.

- **Preparing the registers:** Before making a system call, the necessary information is usually loaded into specific registers. This includes the system call number and any required parameters.

- **Invoking the system call:** Once the registers are prepared, the assembly code triggers the system call using an appropriate instruction. For **x86** architecture, the int **0x80** instruction is commonly used, while the **syscall** instruction is used for **x86-64** architecture.
- **Handling the system call result**: After the system call is executed, the result is often returned in a register, typically the **eax** register. It is crucial to handle this result properly to determine the success or failure of the system call.

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

```
mov     edx,4           ; message length
mov     ecx,msg         ; message to write
mov     ebx,1           ; file descriptor (stdout)
mov     eax,4           ; system call number (sys_write)
int     0x80            ; call kernel
```

All the syscalls are listed in */usr/include/asm/unistd.h*, together with their numbers (the value to put in EAX before you call int 80h).

The following table shows some of the system calls used in this tutorial −

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Experimentation Work:

1. Student can perform simple programs with system call using NASM

CONCLUSION:

In conclusion, the experiment focused on understanding system calls in NASM assembly, which serve as a vital mechanism for interacting with the underlying operating system. Through this experiment, we gained valuable insights into the process of invoking system calls and utilizing operating system services within our assembly programs.

Experiment No. 0

TITLE: To understand of the Programming Toolchain

SUB-EXPERIMENT No: 0.4

AIM: Understanding of Debugging Steps(using gdb) for NASM.

OBJECTIVE:

1. Familiarity with GDB
2. To study Step-by-step debugging.
3. To learn mechanism of breakpoints & check the results.
4. To learn how to check Register and memory values.

Theory:

To debug NASM assembly programs using GDB (GNU Debugger), you can follow these steps:

1. Assemble and link your assembly program: Use NASM to assemble your assembly code and link it to create an executable file. For example, if your assembly file is named "program.asm," you can use the following commands to assemble and link it:

nasm -f elf32 program.asm -o program.o

ld -m elf_i386 program.o -o program

2. Start GDB: Launch GDB by running the following command in the terminal, specifying your executable file as an argument: ***gdb program***

3. Set breakpoints: Use the break command in GDB to set breakpoints at specific locations in your assembly code. For example, to set a breakpoint at the beginning of the program, use:

break _start

4. **Run the program:** Start the execution of your program by entering the run command in GDB. Execution will stop at the breakpoints you have set.

5. **Debugging:** Once execution is paused at a breakpoint, you can use various commands in GDB to debug your assembly program. Here are some commonly used commands:

6. **stepi (or si):** Execute the current instruction and stop at the next instruction. This command allows you to step through the code line by line.

7. **nexti (or ni):** Similar to stepi, but it does not enter into subroutines. It executes the current instruction and moves to the next instruction at the same level of the program.

8. **info registers**: Display the contents of registers. This command provides information about the values stored in the registers.

9. **x/<format> <address>**: Examine memory at a specific address. The <format> specifies the display format (such as x for hexadecimal) and <address> specifies the memory location you want to examine.

10. **Continue execution**: To continue execution after a breakpoint or to let the program run until completion, use the continue command in GDB.

11. Analyze program state: Use GDB commands such as info registers and x/<format> <address> to inspect the state of registers and memory at different points in your program. This information can help you identify bugs or unexpected behavior.

Important Steps to remember

nasm -f elf -l add.lst add.asm	assemble file
ld -m elf_i386 add.o -o add	invoke loader
./add	execute the output
USE OF GDB	
gdb ./add	load exe in gdb
break _start	set break point
run	run the program
set disassembly-flavor intel	convert into intel mode
disassemble _start	to see the disassemble flavor
nexti	move next step
info registers	to see the status of registers
q	quit the program
layout asm	To open the Assembly View
layout regs	To open the Registers View in run-time

Experimentation Work:

1. Student can execute simple registers value checking / addition programs and check the gdb functionality
2. Students can compare the register updating
3. Students can see the list and object files to verify the execution.

CONCLUSION:

Through the experiment, we verified the correctness and reliability of our assembly program by observing its execution in GDB. By leveraging GDB's debugging capabilities, we gained confidence in the program's functionality and were able to address any issues that arose during the debugging process.

MIT Art, Design and Technology, Pune

Department of Computer Engineering

Name: _____ **Date of Performance**: _ _ /_ _/20_ _

Class: S.Y. Computer **Date of Completion:** _ _ /_ _/20_ _

Div--------, **Batch**: _ _ _ _ _

Roll No: _ _ _ _ **Sign of Teacher----------**

EXPERIMENT No: 01

TITLE: Arithmetic Operations (such as Addition, Subtraction Multiplication, and Division) of 8-bit hex numbers

AIM:

1. To study and execute the Debug Commands.
2. Write an 8086 assembly language program (ALP) to add, subtract, multiply, and divide 8-bit / 16-bit numbers stored in memory & store the result (16-bit / 32-bit) in memory.

OBJECTIVE:

1. To study basics of assembly language programming i.e. Software development tools required, commands to use them, the format of ALP, and assembler directives.
2. To study steps in assembly language programming.
3. To study gdb-debug to execute the program and to check the results.
4. To learn how to perform various arithmetic operations in Assembly Language.

ALGORITHM 1: Addition of Two 8bit Hex Number

1. **Initialize Data Segment**

- Load the data segment into the DS register.

2. **Load the 8-bit Numbers**

- Load the first number into the AL register.
- Load the second number into the BL register.

3.**Add the Numbers**

- Add the value in the BL register to the AL register.

4. **Store the Result**

- Store the result in a memory location.

5. **Exit Program**

- Use a Linux system call to exit the program.

THEORY:

Initializing the Data Segment

We usually need to initialize the data segment in an 8086 assembly program running under DOS. However, in a Linux environment, the initialization of segments is handled by the operating system, so we typically don't need to explicitly set the segment registers.

Loading the 8-bit Numbers

AL (Accumulator Low): In the 8086 assembly language, the AL register is the lower 8-bit part of the AX register. It is used for arithmetic, logic, and data transfer operations.

BL (Base Low): The BL register is the lower 8-bit part of the BX register.

To load the 8-bit numbers into the registers:

mov al, num1: This instruction moves the value stored at the memory location labeled num1 into the AL register.

mov bl, num2: This instruction moves the value stored at the memory location labeled num2 into the BL register.

Adding the Numbers

ADD Instruction: The add instruction performs an addition of two operands. Here, add al, bl adds the value in BL to the value in AL and stores the result in AL.

Storing the Result

mov [result], al: This instruction moves the value in the AL register (which is the result of the addition) to the memory location labeled result.

Exiting the Program

In a Linux environment, system calls are used to interact with the operating system kernel. To exit the program, we use the sys_exit system call.

System Call Number: Each system call has a unique number. For sys_exit, the number is 1.

Exit Code: The exit code is typically passed in the EBX register. An exit code of 0 usually indicates a successful termination.

To make a system call in Linux:

Load the system call number into the EAX register.

Load the exit code into the EBX register.

Invoke the system call using the int 0x80 instruction, which triggers a software interrupt and transfers control to the kernel.

PROGRAM TEMPLATE:

```
section .data
    num1 db 0x1A          ; First 8-bit number (example: 1Ah)
    num2 db 0x2B          ; Second 8-bit number (example: 2Bh)
    result db 0           ; Memory location to store the result

section .bss
section .text
    global _start         ; Entry point for the program

_start:
    ; Initialize Data Segment
    mov eax, num1         ; Load first 8-bit number into AL (EAX lower byte)
    mov ebx, num2         ; Load second 8-bit number into BL (EBX lower byte)

    ; Add the Numbers
    add al, bl            ; Add BL to AL, result in AL
```

```
; Store the Result
mov [result], al       ; Store the result in memory

; Exit Program using Linux system call
mov eax, 1             ; syscall number for exit (sys_exit)
xor ebx, ebx           ; exit code 0
int 0x80               ; invoke syscall
```

ALGORITHM 2: for subtraction of two 8 bit hex number

1. **Initialize Data Segment**
   - Load the data segment into the DS register.
2. **Load the 8-bit Numbers**
   - Load the first number into the AL register.
   - Load the second number into the BL register.
3. **Subtract the Numbers**
   - Subtract the value in the BL register from the AL register.
4. **Store the Result**
   - Store the result in a memory location.
5. **Exit Program**
   - Use an appropriate system call or interrupt to exit the program.

THEORY:

1. Initializing the Data Segment

In assembly language, data is often stored in a segment called the data segment. The data segment register (DS) needs to be initialized to point to the data segment. However, in environments like Linux, segment registers are often managed by the operating system, so this step might be implicitly handled.

2. Loading the 8-bit Numbers

To perform arithmetic operations, the operands need to be loaded into registers. The 8086 has several general-purpose registers, including `AL` (the lower byte of `AX`) and `BL` (the lower byte of `BX`), which can be used for 8-bit operations.

3. Performing the Subtraction

The `SUB` instruction is used to subtract one value from another. In this case, the value in the `BL` register is subtracted from the value in the `AL` register, and the result is stored in `AL`.

4. Storing the Result

After the subtraction, the result in `AL` is stored in a predefined memory location.

5. Exiting the Program

To properly exit the program, a system call or interrupt is used. In DOS, the interrupt `int 21h` is typically used, while in Linux, a system call using `int 0x80` is used.

PROGRAM TEMPLATE:

```
section .data
    num1 db 0x3C          ; First 8-bit number (example: 3Ch)
    num2 db 0x1A          ; Second 8-bit number (example: 1Ah)
    result db 0           ; Memory location to store the result

section .bss

section .text
    global _start         ; Entry point for the program

_start:
    ; Load the 8-bit numbers
    mov al, num1          ; Load first 8-bit number into AL
    mov bl, num2          ; Load second 8-bit number into BL

    ; Subtract the numbers
    sub al, bl            ; Subtract BL from AL, result in AL

    ; Store the result
    mov [result], al      ; Store the result in memory
```

```
; Exit program using Linux system call
mov eax, 1          ; syscall number for exit (sys_exit)
xor ebx, ebx        ; exit code 0
int 0x80            ; invoke syscall
```

ALGORITHM 3: for Multiplication of two 8-bit hex number

1. **Initialize Data Segment**
   - Load the data segment into the DS register.
2. **Load the 8-bit Numbers**
   - Load the first number into the AL register.
   - Load the second number into the BL register.
3. **Multiply the Numbers**
   - Use the MUL instruction to multiply the value in the AL register by the value in the BL register. The result is stored in the AX register.
4. **Store the Result**
   - Store the result in a memory location.
5. **Exit Program**
   - Use an appropriate system call or interrupt to exit the program.

Theory:

- **Data Segment Initialization**:
  - The data segment is implicitly initialized in the Linux environment.
- **Load the 8-bit Numbers**:

- mov al, num1: This instruction moves the value stored at the memory location labeled num1 into the AL register.
- mov bl, num2: This instruction moves the value stored at the memory location labeled num2 into the BL register.

 Multiplication:

- mul bl: The mul instruction multiplies the value in the AL register by the value in the BL register. The result of the multiplication is stored in the AX register. Since the result of multiplying two 8-bit numbers can be up to 16 bits, the AX register is used to store the full result.

 Store the Result:

- mov [result], ax: This instruction moves the 16-bit value in the AX register to the memory location labeled result.

 Exit Program:

- mov eax, 1: Loads the system call number for sys_exit into EAX.
- xor ebx, ebx: Clears the EBX register, setting the exit code to 0.
- int 0x80: Invokes the Linux system call to exit the program.

PROGRAM TEMPLATE:

```
section .data
    num1 db 0x0A          ; First 8-bit number (example: 0Ah)
    num2 db 0x05          ; Second 8-bit number (example: 05h)
    result dw 0           ; Memory location to store the result (16-bit)

section .bss

section .text
    global _start         ; Entry point for the program

_start:
    ; Load the 8-bit numbers
    mov al, num1          ; Load first 8-bit number into AL
```

```
mov bl, num2          ; Load second 8-bit number into BL

; Multiply the numbers
mul bl                ; Multiply AL by BL, result in AX

; Store the result
mov [result], ax      ; Store the result in memory (AX is 16-bit)

; Exit program using Linux system call
mov eax, 1            ; syscall number for exit (sys_exit)
xor ebx, ebx          ; exit code 0
int 0x80              ; invoke syscall
```

ALGORITHM 4: for Division of two 8-bit hex number

1. **Initialize Data Segment**

- Load the data segment into the DS register.

2. Load the 8-bit Numbers

- Load the dividend into the AL register.
- Load the divisor into the BL register.

3. **Clear the AH Register**

- Clear the AH register since the DIV instruction uses AX (AH) for division.

4. **Divide the Numbers**

- Use the DIV instruction to divide the value in the AL register by the value in the BL register. The quotient is stored in AL and the remainder in AH.

5. **Store the Result**

- Store the quotient in a memory location.
- Store the remainder in a memory location.

6. **Exit Program**

- Use an appropriate system call or interrupt to exit the program.

THEORY:

1. Initializing the Data Segment

In assembly language, data is often stored in a segment called the data segment. The data segment register (DS) needs to be initialized to point to the data segment. However, in environments like Linux, segment registers are often managed by the operating system, so this step might be implicitly handled.

2. Loading the 8-bit Numbers

To perform arithmetic operations, the operands need to be loaded into registers. The 8086 has several general-purpose registers, including `AL` (the lower byte of `AX`) and `BL` (the lower byte of `BX`), which can be used for 8-bit operations.

3. Preparing for Division

The `DIV` instruction is used for unsigned division. When dividing 8-bit numbers, the dividend should be placed in the `AX` register, which is treated as a 16-bit register (with the high byte in `AH` and the low byte in `AL`). The divisor is placed in another 8-bit register, such as `BL`.

4. Performing the Division

The `DIV` instruction divides the 16-bit value in `AX` by the 8-bit value in the specified register (e.g., `BL`). The quotient of the division is stored in `AL`, and the remainder is stored in `AH`.

5. Storing the Result

After the division, the quotient in `AL` and the remainder in `AH` need to be saved in predefined memory locations.

6. Exiting the Program

To properly exit the program, a system call or interrupt is used. In DOS, the interrupt `int 21h` is typically used, while in Linux, a system call using `int 0x80` is used.

Program template

```
section .data
    dividend db 0x1A       ; Dividend (example: 1Ah)
    divisor  db 0x05       ; Divisor (example: 05h)
    quotient db 0          ; Memory location to store the quotient
    remainder db 0         ; Memory location to store the remainder

section .bss

section .text
    global _start          ; Entry point for the program

_start:
    ; Load the 8-bit numbers
    mov al, [dividend]     ; Load dividend into AL
    mov bl, [divisor]      ; Load divisor into BL

    ; Clear AH register
    xor ah, ah             ; Clear AH to ensure AH:AL is the dividend

    ; Divide the numbers
    div bl                 ; Divide AX by BL, quotient in AL, remainder in AH

    ; Store the result
    mov [quotient], al     ; Store the quotient in memory
    mov [remainder], ah    ; Store the remainder in memory

    ; Exit program using Linux system call
    mov eax, 1             ; syscall number for exit (sys_exit)
    xor ebx, ebx           ; exit code 0
    int 0x80               ; invoke syscall
```

RESULT/OUTPUT:

CONCLUSION:

This program demonstrates how to perform arithmetic operations on two 8-bit hexadecimal numbers using 8086 assembly language and exit the program using a Linux system call. The key steps involve loading the numbers into registers, performing the addition, and various arithmetic operations (subtraction, division, and multiplication) storing the result, and invoking a system call to exit the program.

MIT Art, Design and Technology, Pune

Department of Computer Engineering

Name: _____**Date of Performance**: _ _ /_ _/20_ _

Class: S.Y. Computer **Date of Completion:** _ _ /_ _/20_ _

Div--------, **Batch**: _ _ _ _ _

Roll No: _ _ _ _ **Sign of Teacher----------**

EXPERIMENT No: 02

TITLE: 8086 assembly language program (ALP) to add an array of n numbers, 8-bit and 16-bit numbers.

AIM: Write an 8086 assembly language program (ALP) to add an array of n numbers, 8-bit and 16-bit numbers stored in memory & store the result (16-bit I 32-bit) in memory.

OBJECTIVE:

1. Add an array of 8-bit numbers and store the 16-bit result in memory.
2. Add an array of 16-bit numbers and store the 32-bit result in memory.

Algorithm for Adding an Array of 8-Bit Numbers

1. Initialize Data Segment

- Set up the DS register to point to the data segment.

2. Load the Array Address

- Load the base address of the array into a register.

3. Initialize Counters

- Initialize a counter to keep track of the number of elements.
- Initialize the `AX` register to zero for accumulating the sum.

4. Loop Through Array

- Read each element from the array.
- Add the element to the accumulator (`AL`).

- Update the sum in the `AX` register.
- Decrement the counter.
- Repeat until all elements are processed.

5. Store the Result

- Store the 16-bit result (from `AX`) in a memory location.

6. Exit Program

- Use an appropriate system call or interrupt to exit the program.

Algorithm for Adding an Array of 16-Bit Numbers

1. Initialize Data Segment

- Set up the DS register to point to the data segment.

2. Load the Array Address

- Load the base address of the array into a register.

3. Initialize Counters

- Initialize a counter to keep track of the number of elements.
- Initialize the EAX register to zero for accumulating the sum (32-bit).

4. Loop Through Array

- Read each 16-bit element from the array.
- Add the element to the accumulator (EAX).
- Update the sum in the EAX register.
- Decrement the counter.
- Repeat until all elements are processed.

5. Store the Result

- Store the 32-bit result (from EAX) in a memory location.

6. Exit Program

- Use an appropriate system call or interrupt to exit the program.

THEORY:

1. Initialize Data Segment

The data segment (DS) is a segment register that points to the segment where variables and data are stored. For simplicity, in modern operating systems like Linux, this initialization is often managed by the environment, but in DOS environments, you might explicitly set up the segment registers.

2. Load the Array Address

The array of 8-bit numbers needs to be loaded into a register. This register will be used to access each element of the array during the addition process.

3. Initialize Counters

Counters help keep track of the number of elements to be processed. The sum will be accumulated in a 16-bit register (AX).

4. Loop Through the Array

A loop iterates over each element of the array:

- **Read** the current 8-bit element from the array.
- **Add** this element to the accumulator (AX).
- **Update** the sum.
- **Repeat** the process for all elements in the array.

5. Store the Result

After completing the loop, the final sum (which is in the AX register) is stored in a predefined memory location.

6. Exit the Program

The program terminates using a system call or interrupt. For Linux, this is done with the int 0x80 instruction for the sys_exit system call.

PROGRAM TEMPLATE: Program for 8 bit Araay addition

```
section .data
    array db 0x01, 0x02, 0x03, 0x04  ; Array of 8-bit numbers
    n db 4                    ; Number of elements in the array
    sum dw 0                  ; Memory location to store the 16-bit sum

section .text
    global _start             ; Entry point for the program

_start:
    ; Initialize data segment (not needed in Linux but required for DOS)
    ; mov ax, 0x0800          ; Example data segment initialization for DOS
    ; mov ds, ax

    ; Initialize pointers and counters
    lea si, [array]           ; Load the address of the array into SI
    mov cl, [n]               ; Load the number of elements into CL
    xor ax, ax                ; Clear AX register to store the sum

    ; Loop through the array
loop_start:
    lodsb                     ; Load byte from DS:SI into AL and increment SI
    add ax, al                ; Add AL to AX, sum in AX
    loop loop_start           ; Decrement CL and loop if CL != 0

    ; Store the result
    mov [sum], ax             ; Store the 16-bit sum in memory

    ; Exit program using Linux system call
    mov eax, 1                ; syscall number for exit (sys_exit)
    xor ebx, ebx              ; exit code 0
    int 0x80                  ; invoke syscall
```

PROGRAM TEMPLATE: Program for 16-bit Araay addition

```
section .data
    array dw 0x1234, 0x5678, 0x9ABC, 0xDEF0 ; Array of 16-bit numbers
    n db 4                          ; Number of elements in the array
    sum dd 0                        ; Memory location to store the 32-bit sum

section .text
    global _start                   ; Entry point for the program

_start:
    ; Initialize data segment (not needed in Linux but required for DOS)
    ; mov ax, 0x0800                ; Example data segment initialization for DOS
    ; mov ds, ax

    ; Initialize pointers and counters
    lea si, [array]                 ; Load the address of the array into SI
    mov cl, [n]                     ; Load the number of elements into CL
    xor eax, eax                    ; Clear EAX register to store the sum (32-bit)

    ; Loop through the array
loop_start:
    lodsw                           ; Load word from DS:SI into AX and increment SI
    add eax, eax                    ; Add the 16-bit value to EAX (sum in EAX)
    loop loop_start                 ; Decrement CL and loop if CL != 0

    ; Store the result
    mov [sum], eax                  ; Store the 32-bit sum in memory

    ; Exit program using Linux system call
    mov eax, 1                      ; syscall number for exit (sys_exit)
    xor ebx, ebx                    ; exit code 0
    int 0x80                        ; invoke syscall
```

RESULT/OUTPUT:

CONCLUSION: This 8086 assembly language program performs the addition of an array of 8-bit numbers and stores the 16-bit result in memory. The algorithm and code demonstrate basic assembly language constructs such as loops, arithmetic operations, and memory management. The key elements involve initializing the data segment, iterating through the array, performing addition, and storing the final result.

MIT Art, Design and Technology, Pune

Department of Computer Engineering

Name: _____**Date of Performance**: _ _ /_ _/20_ _

Class: S.Y. Computer **Date of Completion:** _ _ /_ _/20_ _

Div--------, **Batch**: _ _ _ _ _

Roll No: _ _ _ _ **Sign of Teacher----------**

EXPERIMENT No: 03

TITLE: An 8086 Assembly Language program for Hex to BCD conversion

AIM: Write an 8086 assembly language program (ALP) to convert a 1-digit Hex number into its equivalent BCD number. Make your program user-friendly to accept the input from users & display results on the screen. Display proper strings to prompt the user while receiving the input and displaying the result.

OBJECTIVE:

1. Illustrate the conversion process from hexadecimal to BCD format.
2. Demonstrate fundamental assembly language programming techniques, including data manipulation and control flow.
3. Implement a validation mechanism to ensure correct user input and handle errors.
4. Perform arithmetic operations required for the conversion from hex to BCD.
5. Explain how different data representations are used and converted in assembly language programming.
6. Illustrate basic memory management techniques including storing and retrieving data.

ALGORITHM:

1. **Initialize Data Segment:**
   - Set up the data segment for storing strings and results.
2. **Display Prompt for Input:**
   - Print a message prompting the user to enter a 1-digit hexadecimal number.
3. **Accept User Input:**
   - Read a single character input from the user, which represents the hexadecimal number.

4. **Convert Hex to BCD:**
    - Check if the input is a valid hexadecimal digit (0-9, A-F).
    - Convert the character to its numeric value.
    - Convert the numeric value to its BCD equivalent.
5. **Display Result:**
    - Print a message displaying the BCD result.
6. **Exit Program:**
    - Use the exit system call to terminate the program.

PROGRAM TEMPLATE:

```
section .data
    input_prompt db "Enter a 1-digit hexadecimal number: ", 0
    invalid_input db "Invalid input! Please enter a valid hex digit (0-9, A-F).", 10, 0
    result_prompt db "The BCD equivalent is: ", 0
    hex_input db 0                ; To store the input hex digit
    bcd_result db 0               ; To store the BCD result
    result_display db "BCD: 00", 10, 0  ; To display the result

section .bss
    ; Uninitialized data

section .text
    global _start                 ; Entry point for the program

_start:
    ; Display input prompt
    mov eax, 4                    ; syscall number for sys_write
    mov ebx, 1                    ; file descriptor 1 (stdout)
    lea ecx, [input_prompt]       ; address of input prompt
    mov edx, 34                   ; length of input prompt
    int 0x80                      ; invoke syscall
```

```
    ; Read user input
    mov eax, 3                  ; syscall number for sys_read
    mov ebx, 0                  ; file descriptor 0 (stdin)
    lea ecx, [hex_input]        ; address of input buffer
    mov edx, 1                  ; number of bytes to read
    int 0x80                    ; invoke syscall
    sub byte [hex_input], '0'   ; Convert ASCII to number

    ; Validate input (0-9, A-F)
    cmp byte [hex_input], 9
    jbe valid_hex
    sub byte [hex_input], 7     ; Convert 'A'-'F' to 10-15
    cmp byte [hex_input], 15
    ja invalid_hex
    jmp convert_hex

invalid_hex:
    ; Display invalid input message
    mov eax, 4                  ; syscall number for sys_write
    mov ebx, 1                  ; file descriptor 1 (stdout)
    lea ecx, [invalid_input]    ; address of invalid input message
    mov edx, 47                 ; length of invalid input message
    int 0x80                    ; invoke syscall
    jmp _start                  ; Restart input

valid_hex:
    ; Continue with valid hex input

convert_hex:
    ; Convert hex to BCD
    mov al, [hex_input]         ; Load input hex digit
    mov ah, 0                   ; Clear high byte
    aam                         ; Convert to BCD
    mov [bcd_result], ax        ; Store BCD result
```

```
; Display result prompt
mov eax, 4                  ; syscall number for sys_write
mov ebx, 1                  ; file descriptor 1 (stdout)
lea ecx, [result_prompt]    ; address of result prompt
mov edx, 23                 ; length of result prompt
int 0x80                    ; invoke syscall

; Display result
mov al, [bcd_result]
aam                         ; Adjust AX for display
add ax, '00'                ; Convert to ASCII
mov [result_display + 5], ah    ; Store tens digit
mov [result_display + 6], al    ; Store ones digit
mov eax, 4                  ; syscall number for sys_write
mov ebx, 1                  ; file descriptor 1 (stdout)
lea ecx, [result_display]   ; address of result display
mov edx, 9                  ; length of result display
int 0x80                    ; invoke syscall

; Exit program
mov eax, 1                  ; syscall number for sys_exit
xor ebx, ebx                ; exit code 0
int 0x80                    ; invoke syscall
```

THEORY:

1. Data Segment Initialization

- **Input Prompt:** A string to prompt the user to enter a 1-digit hexadecimal number.
- **Invalid Input Message:** A string to notify the user when an invalid input is entered.
- **Result Prompt:** A string to precede the display of the conversion result.
- **Result Display:** A string template to display the BCD result.
- **Variables:** Storage for the hex input digit and the BCD result.

2. Display Input Prompt

- **System Call Used:** sys_write (system call number 4).
- **Parameters:**
  - **File Descriptor:** 1 (stdout) for writing to the terminal.
  - **Buffer:** Address of the input prompt string.
  - **Count:** Number of bytes to write (length of the prompt string).
- **Operation:** The prompt is displayed on the screen using the int 0x80 interrupt to invoke the system call.

3. Accept User Input

Objective: Use a Linux system call to read a single character input from the user.

- **System Call Used:** sys_read (system call number 3).
- **Parameters:**
  - **File Descriptor:** 0 (stdin) for reading from the terminal.
  - **Buffer:** Address where the input will be stored.
  - **Count:** Number of bytes to read (1 byte for a single character).
- **Operation:** The user input is read and stored in the designated buffer using the int 0x80 interrupt to invoke the system call.
- **ASCII to Numeric Conversion:** The ASCII value of the input is converted to its numeric equivalent by subtracting the ASCII code for '0'.

4. Validate Input

Objective: Ensure the input character is a valid hexadecimal digit (0-9 or A-F).

- **Check Numeric Range (0-9):**
  - If the numeric value of the input is between 0 and 9, it is valid.
- **Check Alphabetic Range (A-F):**
  - If the input is not a digit, it is adjusted for uppercase letters 'A'-'F' by subtracting 7 from the ASCII value (since 'A' is ASCII 65 and should map to 10).
  - If the adjusted value is between 10 and 15, it is valid.
- **Invalid Input Handling:**

- If the input is outside these ranges, an invalid input message is displayed using the sys_write system call, and the program prompts the user again.

***5*. Convert Hex to BCD**

- **Steps for conversion**
  - **Load Hex Input:** Load the validated hex input into a register.
  - **Clear High Byte:** Clear the high byte of the register to prepare for the conversion.
  - **AAM Instruction:** Use the ASCII Adjust after Multiply (AAM) instruction to convert the hex digit to its BCD equivalent.
  - **Store BCD Result:** Store the resulting BCD value in a designated variable.

6. Display Conversion Result

- **Display Result Prompt:**
  - **System Call Used:** sys_write.
  - **Parameters:**
    - **File Descriptor:** 1 (stdout).
    - **Buffer:** Address of the result prompt string.
    - **Count:** Number of bytes to write (length of the prompt string).
  - **Operation:** The result prompt is displayed on the screen.
- **Format and Display BCD Result:**
  - **Load BCD Result:** Load the BCD result from memory.
  - **Adjust for Display:** Use the AAM instruction again to prepare the result for ASCII display.
  - **Convert to ASCII:** Convert the numeric BCD digits to their ASCII equivalents by adding the ASCII code for '0'.
  - **Update Result String:** Update the result display string with the converted ASCII digits.
  - **System Call Used:** sys_write.
  - **Parameters:**
    - **File Descriptor:** 1 (stdout).
    - **Buffer:** Address of the result display string.
    - **Count:** Number of bytes to write (length of the result string).
  - **Operation:** The BCD result is displayed on the screen.

7. Exit Program

- **System Call Used:** sys_exit (system call number 1).
- **Parameters:**
  - **Exit Code:** 0 for successful termination.
- **Operation:** The program is terminated using the int 0x80 interrupt to invoke the system call.

Hexadecimal to BCD Conversion Example

Example: Convert the hex number 2A to BCD.

Step-by-Step Conversion

1. **Separate the Hexadecimal Digits:**
   - The hex number 2A has two digits: 2 and A.
2. **Convert Each Hex Digit to Decimal:**
   - Hex digit 2 in decimal is 2.
   - Hex digit A in decimal is 10.
3. **Combine the Decimal Values:**
   - We need to combine the decimal values of the two hex digits into a single number.
   - The hex number 2A can be thought of as (2 * 16^1) + (10 * 16^0) in decimal, which is 32 + 10 = 42.
4. **Convert the Decimal Number to BCD:**
   - The decimal number 42 needs to be converted to its BCD representation.
   - In BCD, each decimal digit is represented separately in binary:
     - The decimal number 4 in BCD is 0100.
     - The decimal number 2 in BCD is 0010.
   - Combining these BCD values, 42 in BCD is 0100 0010.

Converting a 1-Digit Hex Number to BCD in 8086 Assembly Language

Example: Convert the hex number A (which is a single digit) to BCD.

Conversion Steps

1. **Input Hex Digit:**
    - Hex digit: A.
2. **Convert Hex Digit to Decimal:**
    - Hex A in decimal is 10.
3. **Convert Decimal to BCD:**
    - Decimal 10 in BCD is 0001 0000.

RESULT/OUTPUT:

CONCLUSION: This program converts a 1-digit hex number to its BCD equivalent by reading input from the user, validating the input, performing the conversion using the AAM instruction, and displaying the result. It uses Linux system calls for input and output operations, making the program user-friendly and interactive.

MIT Art, Design and Technology, Pune

Department of Computer Engineering

Name: _____**Date of Performance**: _ _ /_ _/20_ _

Class: S.Y. Computer **Date of Completion:** _ _ /_ _/20_ _

Div--------, **Batch**: _ _ _ _ _

Roll No: _ _ _ _ **Sign of Teacher----------**

EXPERIMENT No: 04

TITLE: Write 8086 ALP to perform various String Operations

AIM: Write menu-driven 8086 assembly language program (ALP) to perform string operations

1. Calculation of string length.
2. Concatenation of two strings.
3. Comparison of two strings.
4. Copy one string to another.

OBJECTIVE:

1. Compute the length of a null-terminated string.
2. Concatenate two null-terminated strings.
3. Compare two null-terminated strings.
4. Copy a null-terminated string from one location to another.

ALGORITHM:

Algorithm and 8086 ALP for String Operations

a) Calculation of String Length

Algorithm:

1. **Initialize Registers:**
    - Set the SI register to point to the start of the string.
    - Set the CX register to zero to hold the length count.
2. **Loop Through the String:**
    - Load the current character from the string into AL.
    - Check if the character is the null terminator (0x00).
    - If not, increment the length counter (CX).
    - Move to the next character in the string.
    - Repeat until the null terminator is found.
3. **Store the Result:**
    - After the loop ends, the CX register contains the length of the string.

PROGRAM TEMPLATE

```
; Calculate the length of a string

section .data
    string1 db "Hello, World!", 0 ; Null-terminated string

section .text
    global _start

_start:
    ; Initialize registers
    lea si, [string1]  ; Point SI to the start of the string
    xor cx, cx         ; Clear CX to use it as a counter

length_loop:
    lodsb              ; Load byte at DS:SI into AL and increment SI
    cmp al, 0          ; Check if the byte is the null terminator
    je done_length     ; If it is, jump to the done_length label
    inc cx             ; Increment the length counter
    jmp length_loop    ; Repeat the loop

done_length:
```

```
; Here CX contains the length of the string
; Exit the program
mov eax, 1          ; sys_exit
xor ebx, ebx        ; Status code 0
int 0x80            ; Invoke system call
```

b) Concatenation of Two Strings

Objective: Concatenate two null-terminated strings.

Algorithm:

1. **Initialize Registers:**
   - Set SI to point to the first string.
   - Set DI to point to the end of the first string (find the null terminator).
   - Set DS and ES to point to the memory locations of the source and destination strings, respectively.
2. **Find End of the First String:**
   - Loop through the first string until the null terminator is found.
3. **Concatenate Strings:**
   - Copy characters from the second string to the end of the first string until a null terminator is encountered.
4. **Store the Result:**
   - The concatenated string will be in the memory location of the first string.

PROGRAM TEMPLATE

```
; Concatenate two strings

section .data
   string1 db "Hello, ", 0       ; First string
   string2 db "World!", 0        ; Second string

section .text
   global _start
```

```
_start:
    ; Initialize registers
    lea si, [string1]  ; Point SI to the start of the first string
    lea di, [string2]  ; Point DI to the start of the second string

find_end:
    lodsb              ; Load byte at DS:SI into AL and increment SI
    cmp al, 0          ; Check if the byte is the null terminator
    je start_concat    ; If it is, jump to the start_concat label
    jmp find_end       ; Otherwise, keep searching

start_concat:
    lea si, [string2]  ; Point SI to the start of the second string

concat_loop:
    lodsb              ; Load byte at DS:SI into AL and increment SI
    stosb              ; Store byte in AL at ES:DI and increment DI
    cmp al, 0          ; Check if the byte is the null terminator
    je done_concat     ; If it is, jump to the done_concat label
    jmp concat_loop    ; Otherwise, keep copying

done_concat:
    ; Strings are concatenated
    ; Exit the program
    mov eax, 1         ; sys_exit
    xor ebx, ebx       ; Status code 0
    int 0x80           ; Invoke system call
```

c) Comparison of Two Strings

Objective: Compare two null-terminated strings.

Algorithm:

1. **Initialize Registers:**

- Set SI and DI to point to the start of the two strings to be compared.

2. **Compare Characters:**
   - Compare the characters from both strings byte-by-byte.
   - If characters differ, jump to the result indicating which string is greater or smaller.
   - If characters are the same, continue to the next characters.
   - If null terminator is reached for both strings, they are equal.

```
; Compare two strings

section .data
    string1 db "Hello", 0   ; First string
    string2 db "Hello", 0   ; Second string
section .text
    global _start

_start:
    ; Initialize registers
    lea si, [string1]  ; Point SI to the start of the first string
    lea di, [string2]  ; Point DI to the start of the second string

compare_loop:
    lodsb           ; Load byte at DS:SI into AL and increment SI
    scasb           ; Compare byte in AL with byte at ES:DI
    jne strings_different ; Jump if not equal
    cmp al, 0       ; Check if we reached the end of both strings
    je strings_equal   ; If equal, the strings are equal
    jmp compare_loop   ; Otherwise, continue comparing

strings_different:
    ; Strings are different
    ; Exit the program
    mov eax, 1      ; sys_exit
    mov ebx, 1      ; Status code 1 (not equal)
    int 0x80        ; Invoke system call
```

```
strings_equal:
    ; Strings are equal
    ; Exit the program
    mov eax, 1          ; sys_exit
    xor ebx, ebx        ; Status code 0 (equal)
    int 0x80            ; Invoke system call
```

d) Copy One String to Another

Objective: Copy a null-terminated string from one location to another.

Algorithm:

1. **Initialize Registers:**
    - Set SI to point to the source string.
    - Set DI to point to the destination string.
2. **Copy Characters:**
    - Copy characters from the source string to the destination string.
    - Continue until the null terminator is reached.
3. **Store the Result:**
    - The destination string will be a copy of the source string.

```
; Copy one string to another

section .data

    source_string db "Hello, World!", 0  ; Source string

    dest_string db 20 dup (0)            ; Destination string with space for copy

section .text

    global _start

_start:
```

```
    ; Initialize registers
    lea si, [source_string]  ; Point SI to the source string
    lea di, [dest_string]   ; Point DI to the destination string

copy_loop:
    lodsb                ; Load byte at DS:SI into AL and increment SI
    stosb                ; Store byte in AL at ES:DI and increment DI
    cmp al, 0            ; Check if the byte is the null terminator
    je done_copy         ; If it is, jump to the done_copy label
    jmp copy_loop        ; Otherwise, continue copying

done_copy:
    ; String copy is complete
    ; Exit the program
    mov eax, 1           ; sys_exit
    xor ebx, ebx         ; Status code 0
    int 0x80             ; Invoke system call
```

THEORY:

In computing, a string is typically represented as a sequence of characters ending with a null terminator (0x00). The length of the string is the number of characters before this null terminator.

Concepts and Instructions:

- **Registers:** Use SI (Source Index) to point to the beginning of the string and CX (Count Register) to keep track of the length.
- **Instructions:**
  - **LODSB**: Load Byte String. It moves the byte pointed to by DS:SI into the AL register and increments SI.
  - **CMP**: Compare. It compares the value in AL with 0 to check for the null terminator.
  - **JE**: Jump if Equal. It jumps to a label if the comparison is true (i.e., if the null terminator is found).
  - **INC**: Increment. It increments the CX register to count the characters.

Explanation with Example:

Suppose we have the string "HELLO":

- **SI** points to the start of "HELLO".
- **LODSB** loads 'H' into AL, checks for null terminator, increments SI, and increments CX if not null.
- This process continues until the null terminator is encountered.

2. Concatenation of Two Strings

Theory:

String concatenation involves appending the characters of the second string to the end of the first string. This operation modifies the first string to contain both strings followed by a new null terminator.

Concepts and Instructions:

- **Registers:** Use SI to point to the first string and DI for the second string.
- **Instructions:**
  - **LODSB**: Load Byte String.
  - **STOSB**: Store Byte String. It moves the byte from AL to ES:DI and increments DI.

- **CMP**: Compare. Used to check for the null terminator in the source string.
- **JE**: Jump if Equal. Used to end the loop when the null terminator is encountered.

Explanation with Example:

To concatenate "HELLO" with "WORLD":

1. **Find the End of the First String:** Loop through "HELLO" to find the null terminator.
2. **Concatenate Second String:** Start copying "WORLD" from the end of "HELLO" and continue until the null terminator of "WORLD".

3. Comparison of Two Strings

Objective:

To compare two null-terminated strings to check if they are equal or if one is greater than or less than the other.

Theory:

String comparison involves comparing characters from both strings one by one. If the characters are the same, the comparison continues; if they differ, the result of the comparison is determined.

Concepts and Instructions:

- **Registers:** Use SI for the first string and DI for the second string.
- **Instructions:**
  - **LODSB**: Load Byte String.
  - **SCASB**: Scan String Byte. Compares the byte in AL with the byte at ES:DI.
  - **CMP**: Compare.
  - **JE**: Jump if Equal.
  - **JNE**: Jump if Not Equal.

Explanation with Example:

To compare "HELLO" with "HELLO":

- **Compare Each Character:** Compare the ASCII values of each character in the strings.

- **Check for Equality:** If all characters match and the null terminators are also equal, the strings are identical.
- **Result:** The program sets an appropriate status code based on whether the strings are equal or not.

4. Copying One String to Another

Objective:

To copy a null-terminated string from one location to another.

Theory:

String copying involves transferring each character from the source string to the destination string until the null terminator is encountered.

Concepts and Instructions:

- **Registers:** Use SI for the source string and DI for the destination string.
- **Instructions:**
  - **LODSB**: Load Byte String.
  - **STOSB**: Store Byte String. Stores the byte in AL to ES:DI and increments DI.
  - **CMP**: Compare.
  - **JE**: Jump if Equal.

Explanation with Example:

To copy "HELLO" to another location:

- **Copy Characters:** Move each character from "HELLO" to the destination string.
- **End at Null Terminator:** Continue copying until the null terminator is encountered.

RESULT/OUTPUT:

CONCLUSION: