SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-01

Title:-

Display Accepted Numbers

Assignment Name:- Write an X86/64 ALP to accept five 64 bit Hexadecimal numbers fromuser and store them in an array and display the accepted numbers.

Objective-

- To understand the assembly language program
- To understand assembly language programming instruction set
- To understand different assembler directives with example
- To understand macro,procedure concept.

Outcome-

- Students will be able to write ALP code.
- Students will be able to understand difference between procedure and macro .

Prerequisite-

System call of Unix for Assembly language Program.

HardwareRequirement-

Desktop PC

SoftwareRequirement-

Ubuntu14.04, Assembler:

NASM version

2.10.07Linker: ld

Theory-

Introduction:-

Introduction to Assembly Language Programming:

Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities. Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction'. Processor

understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in software development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form. Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

Advantages of Assembly Language

- An understanding of assembly language provides knowledge of:
- Interface of programs with OS, processor and BIOS;
- Representation of data in memory and other external devices;
- How processor accesses and executes instruction;
- How instructions accesses and process data;
- How a program access external devices.

Other advantages of using assembly language are:

- It requires less memory and execution time;
- It allows hardware-specific complex jobs in an easier way;
- It is suitable for time-critical jobs;

ALP Step By Step:

Installing NASM:

If you select "Development Tools" while installed Linux, you may NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

- Open a Linux terminal.
- Type ***whereis nasm*** and press ENTER.
- If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will seejust *nasm:*, then you need to install NASM.

To install NASM take the following steps:

Open Terminal and run below commands:
sudo apt-get update
sudo apt-get install nasm

Assembly Basic Syntax:
An assembly program can be divided into three sections:

- The **data** section
- The **bss** section
- The **text** section

The order in which these sections fall in your program really isn't important, but by convention the .data section comes first, followed by the .bss section, and then the .text section.

The .data Section

The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution. You don't have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the program as a whole into memory. The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

The .bss Section

Not all data items need to have values before the program begins running. When you're reading data from a disk file, for example, you need to have a place for the data to go after it comes in fromdisk. Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don't say what values are to be present in the buffer. There's a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

The .text Section

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data items are defined in .text. The .text section contains symbols called *labels* that identify locations in the program code for jumps and calls, but beyond your instruction mnemonics, that's about it. All global labels must be declared in the .text section, or the labels cannot be ''seen'' outside your program by the Linux linker or the Linux loader. Let's look at the labels issue a little more closely.

Labels

A label is a sort of bookmark, describing a place in the program code and giving it a name that's easier to remember than a naked memory address. Labels are used to indicate the places where jump instructions should jump to, and they give names to callable assembly language procedures. Here are the most important things to know about labels:

- *Labels must begin with a letter, or else with an underscore, period, or question mark.* These lastthree have special meanings to the assembler, so don't use them until you know how NASM interprets them.
- *Labels must be followed by a colon when they are defined.* This is basically what tells NASMthat the identifier being defined is a label. NASM will punt if no colon is there and will not flag an error, but the colon nails it, and prevents a mistyped instruction mnemonic from being mistaken for a label. Use the colon!
- *Labels are case sensitive.* So yikes:, Yikes:, and YIKES: are three completely different labels.

Assembly Language Statements

Assembly language programs consist of three types of statements:

- Executable instructions or instructions
- Assembler directives or pseudo-ops
- Macros

Syntax of Assembly Language Statements

[label] mnemonic [operands] [;comment]

ALGORITHM:
INPUT: ARRAY
OUTPUT: ARRAY
STEP 1: Start.
STEP 2: Initialize the data segment.
STEP 3: Display msg1 "Accept array from user. "
STEP 4: Initialize counter to 05 and rbx as 00
STEP 5: Store element in array.
STEP 6: Move rdx by 17.
STEP 7: Add 17 to rbx.
STEP 8: Decrement Counter.
STEP 9: Jump to step 5 until counter value is not zero.
STEP 9: Display msg2.
STEP 10: Initialize counter to 05 and rbx as 00
STEP 11: Display element of array.
STEP 12: Move rdx by 17.
STEP 13: Add 17 to rbx.
STEP 14: Decrement Counter.
STEP 15: Jump to step 11 until counter value is not zero.
STEP 16: Stop

Conclusion:-Hence we implemented an ALP to display accepted Numbers.

Questions:-

Q.1.Explain macro with example?
Q.2 Explain procedure with example?
Q.3 Write difference between procedure and macro?

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-02

Title:-String length calculation
Assignment Name: - Write an X86/64 ALP to accept a string and to display its length.

Objective-

- To study various string instruction
- To understand how to define string in data segment.
- To calculate string length.

Outcome-

- Students will be able to write code to accept str and display string length.

Prerequisite-

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,

Assembler: NASM version 2.10.07Linker: ld

Introduction:-

Guidelines for the algorithm:

1) Initialize Data section.
2) Declare string and other required variables.
3) Accept string from user.
4) Count of entered string including enter character is available with RAX register.
5) Display length as value available in RAX.

Conclusion:-Hence we implemented an ALP to calculate string length.

Questions:-

1. Explain string instruction of 80386?
2. Explain direction flag?

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	ROLL NO.:
SEMESTER:SEM-II	YEAR:2023-24
DATE OF PERFORMANCE:	DATE OF SUBMISSION:
EXAMINED:	

Assignment No-03

Title:- Find the largest of given numbers.
Assignment Name:- An X86/64 ALP to find the largest of given Byte/Word/Dword/64-bit numbers.

Objective-

- To understand the assembly language program
- To understand 64 bit interrupt.

Outcome-

- Students will be able to write code for how to find the largest of given
- Students will be able to understand different assembly language instruction.

Prerequisite-
System call of Ūnix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,
Assembler: NASM version 2.10.07Linker

Introduction:-

Theory:

Algorithm:

1. Start
2. Initialise section .data
3. Define variable for array,large
4. Using cmp instruction find larger number from array.
5. Display largest number
6. Terminate program using system call

7. Stop

Conclusion: -Hence we implemented an ALP find the largest of given array.

Questions:-

Q.1.Explain macro used with Example?
Q.2 Explain CMP instruction?
Q.3 Draw and explain TSS segment of 80386?

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-04

Title:-Count no. of positive and negative numbers

Assignment Name: - Write an ALP to count no. of positive and negative numbers from the array.

Objective-

- To understand the assembly language program
- To understand 64 bit interrupt.

Outcome-

- Students will be able to write code for how to count positive and negative number from array
- Students will be able to understand different assembly language instruction.

Prerequisite -

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,
Assembler: NASM version 2.10.07
Linker: ld

Introduction:-

Theory:

Introduction to Assembly Language Programming:

Each personal computer has a microprocessor that manages the computer's arithmetical, logical and control activities. Each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instruction. Processor

understands only machine language instructions which are strings of 1s and 0s. However machine language is too obscure and complex for using in oftware development. So thelow level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form. Assembly language is a low-level programming language for a computer, or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.

Advantages of Assembly Language

- An understanding of assembly language provides knowledge of Interface of programs with OS, processor and BIOS.
- Representation of data in memory and other external devices.
- How processor accesses and executes instruction
- How instructions accesses and process data
- How a program access external devices.
- It requires less memory and execution time.
- It allows hardware-specific complex jobs in an easier way.
- It is suitable for time-critical jobs.

ALP Step By Step:

Installing NASM:

If you select "Development Tools" while installed Linux, you may NASM installed along with the Linux operating system and you do not need to download and install it separately. For checking whether you already have NASM installed, take the following steps:

Open a Linux terminal.
Type ***where is nasm*** and press ENTER.
If it is already installed then a line like, *nasm: /usr/bin/nasm* appears. Otherwise, you will see just *nasm:*, then you need to install NASM.

To install NASM take the following steps:

Check the netwide assembler (NASM) website for the latest version.

1. Download the Linux source archive nasm-X.XX. ta .gz, where X.XX is the NASM version number in the archive.
2. Unpack the archive into a directory, which creates a subdirectory nasm-X. XX.
3. cd to *nasm-X. XX* and type ***./configure*** . This shell script will find the best C compiler to use and set up Makefiles accordingly.
4. Type ***make*** to build the nasm and ndisasm binaries.
5. Type ***make install*** to install nasm and ndisasm in /usr/local/bin and to install the man pages.

Assembly Basic Syntax:

An assembly program can be divided into three sections:

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

The order in which these sections fall in your program really isn't important, but by convention the .data section comes first, followed by the .bss section, and then the .text section.

The .data Section

The .data section contains data definitions of initialized data items. Initialized data is data that has a value before the program begins running. These values are part of the executable file. They are loaded into memory when the executable file is loaded into memory for execution. You don't have to load them with their values, and no machine cycles are used in their creation beyond what it takes to load the program as a whole into memory.

The important thing to remember about the .data section is that the more initialized data items you define, the larger the executable file will be, and the longer it will take to load it from disk into memory when you run it.

Section .data

The .bss Section

Not all data items need to have values before the program begins running. When you're reading data from a disk file, for example, you need to have a place for the data to go after it comes in from disk. Data buffers like that are defined in the .bss section of your program. You set aside some number of bytes for a buffer and give the buffer a name, but you don't say what values are to be present in the buffer.

There's a crucial difference between data items defined in the .data section and data items defined in the .bss section: data items in the .data section add to the size of your executable file. Data items in the .bss section do not.

Section .bss

The .text Section

The actual machine instructions that make up your program go into the .text section. Ordinarily, no data items are defined in .text. The .text section contains symbols called *labels* that identify locations in the program code for jumps and calls, but beyond your instruction mnemonics, that's about it.

All global labels must be declared in the .text section, or the labels cannot be ''seen'' outside your program by the Linux linker or the Linux loader

Section .text

Assembly Language Statements

Assembly language programs consist of three types of statements:

- Executable instructions or instructions
- Assembler directives or pseudo-ops
- Macros

Syntax of Assembly Language Statements

[label]	mnemonic	[operands]	[;comment]

64 bit Interrupt

Write system Call

```
mov rax,1
mov rdi,1
mov rsi,%1
mov rdx,%2
syscall
```

Read system call

```
mov rax,0
mov rdi,0
mov rsi,%1
mov rdx,%2
syscall
```

Compiling and Linking an Assembly Program in NASM

1. Type the above code using a text editor and save it as assignment1.asm.
2. Make sure that you are in the same directory as where you saved assignment1**.asm**.
3. To assemble the program, type **nasm -f elf64 assignment1.asm**
4. If there is any error, you will be prompted about that at this stage. Otherwise an object file of your program named **assignment1.o** will be created.
5. To link the object file and create an executable file named assignment1, type **ld -o assignment assignment1.o**
6. Execute the program by typing **./assignment1**

Algorithm:

1. Start
2. Initialise section .data
3. Define variable for array,pcount,ncount
4. Count Positive and negative number using BT command.
5. Display counts
6. Terminate program using system call
6. Stop

Conclusion:- Hence we implemented an ALP to count positive and negative number from array and display count.

Questions:-

Q.1.Explain BT,JS,loop instruction with Example?
Q.2 Explain Paging in 80386?
Q.3 Draw control registers of 80386

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-05

Title:- Program to switch from real mode to protected mode and display the values of GDTR, LDTR, IDTR, TR and MSW Registers.

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

Objective-

- To understand the assembly language program
- To understand 64 bit interrupt.
- To study GDTR, LDTR and IDTR and MSW

Outcome-

- Students will be able to understand different assembly language instruction.
- Students will be able to write code for how to display the values of GDTR, LDTR, IDTR, TR and MSW Registers
- Students will be able to switch Processor Mode.

Prerequisite -

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,
Assembler: NASM version 2.10.07
Linker: ld

Introduction:-

Four registers of the 80386 locate the data structures that control segmented memory

management called as memory management registers:

1. GDTR :Global Descriptor Table Register

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

2. LDTR :Local Descriptor Table Register

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

3. IDTR Interrupt Descriptor Table Register

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

4. TR Task Register

This register points to the information needed by the processor to define the current task.,These registers store the base addresses of the descriptor tables (A descriptor table is simply a memory array of 8-byte entries that contain Descriptors and descriptor stores all the information about segment) in the linear address space and store the segment limits.

SLDT: Store Local Descriptor Table Register

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

Description: SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table. SLDT is used only in operating system software. It is notused in application programs.

Flags Affected: None

SGDT: Store Global Descriptor Table Register

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

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

Flags Affected: None

SIDT: Store Interrupt Descriptor Table Register

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

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

Flags Affected: None

ALGORITHM:

1. Display welcome message on terminal using macro disp.
2. Store most significant bit of CR0 in eax register.
3. Check the PE bit of CR0.
4. If PE=1 then display message "Processor is in Protected mode".
5. And if PE=0 then display message "Processor is in Real mode".
6. Then copies/stores the contents of GDT, IDT, LDT using sgdt, sidt, sldt instruction.
7. Display their contents using macro

Questions-

1. Explain System Address registers
2. Explain Segment selecters-LDTR and TR
3. Explain CR0 register

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-06

Title:- Non-Overlapped block data transfer.

Assignment Name: - Write X86/64 ALP to perform non-overlapped block transfer without string specific instructions Block containing data can be defined in the data segment.

Objective-

- To study various instruction related to
  a) Arithmetic operations.
  b) Data transfer operations.
  c) Branch operations.
  d) String operations.
- To understand how to define block in data segment.

Outcome-

- Students will be able to write code for block data transfer.
- Students will be able to understand different assembly language instruction.

Prerequisite -

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,
Assembler: NASM version 2.10.07
Linker: ld

Introduction:-

Guidelines for the algorithm:

NON-OVERLAPPED BLOCK DATA TRANSFER

1) Initialize Data section.
2) Define 2 arrays (5 members) for source and destination with different memory locations.
3) Initialize destination array with all zeros.
4) Take count N=5
5) Move the first element of source array to destination array.
6) Decrement count N.
7) Repeat step 5,6 till count N=0
8) Display both source and destination arrays.

Conclusion:- Hence we implemented an ALP to Non-overlapped block data transfer.

Questions:-

1. Explain Assembler directives
2. Explain E-Flag register

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-07

Title:- Overlapped block data transfer.

Assignment Name: Write X86/64 ALP to perform overlapped block transfer with string specific instructions, Block containing data can be defined in the data segment.

Objective-

- To study various instruction related to
  a) Arithmetic operations.
  b) Data transfer operations.
  c) Branch operations.
  d) String operations.
- To understand how to define block in data segment.

Outcome-

- Students will be able to write code for block data transfer.
- Students will be able to understand different assembly language instruction.

Prerequisite -

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,
Assembler: NASM version 2.10.07
Linker: ld

Introduction:-

Guidelines for the algorithm:

OVERLAPPED BLOCK DATA TRANSFER

1) Initialize Data section.
2) Define source array & destination array with 5 numbers..
3) Initialize destination array with all zeros.
4) Copy source array to destination array as it is.
5) Take index from destination array from where you want to do overlapping.
6) Find value N.
7) Move the first element of source array to index location mention in previous step to destination array.
8) Decrement count N.
9) Repeat step 7,8 till count N=0
10) Display both source and destination arrays.

Conclusion:- Hence we implemented an ALP to overlapped and nonoverlapped block data transfer.

Questions:-

3. Explain String Specific Instruction?
4. Explain Stack manipulation instructions?

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	ROLL NO.:
SEMESTER:SEM-II	YEAR:2023-24
DATE OF PERFORMANCE:	DATE OF SUBMISSION:
EXAMINED:	

Assignment No-08

Title:-Multiplication

Assignment Name: - 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.

Objective-

- To understand the different algorithm for multiplication.
- To understand how to write procedure.

Outcome-

- Students will be able to write code for doing multiplication.

Prerequisite -

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,

Assembler: NASM version 2.10.07
Linker: ld

Introduction:-

Guidelines for the algorithm:

1) Display the menu.
   Enter "1" – "ADD AND SHIFT METHOD."
   Enter "2" – "SUCCESSIVE ADDITION METHOD".
   Enter "3" – EXIT

2) Take choice from user then go to the respective subroutines.

ADD AND SHIFT METHOD

1) Initialize code and bss sections.
2) Accept multiplier and multiplicand variables in data segment.
3) Initialize product variable to zero.
4) Set count as number of bits in operand, which is 8.
5) Shift product to left by 1 bit and insert zero as LSB.
6) Transfer MSB of multiplier to carry flag by rotating it to left.
7) Check if carry flag is set or not. If yes add multiplicand to product.
8) Decrement count by 1.
9) Check count=0 else repeat step 5 through step 9 till count=0.
10) Display the final product.

SUCCESSIVE ADDITION METHOD

1) Define product=0.
2) Set count=multiplicand.
3) Add product=product + multiplier.
4) Decrement count.
5) Repeat step 3 and 4 till count=0
6) Display product variable value as final product.

Conclusion: - Hence we implemented an ALP to do multiplication.

Questions:-

1) Explain ADD and SHIFT algorithm with example?
2) Explain what is Interrupt?

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	ROLL NO.:
SEMESTER:SEM-II	YEAR:2023-24
DATE OF PERFORMANCE:	DATE OF SUBMISSION:
EXAMINED:	

Assignment No-09

Title:- Find factorial of a given integer number on a command line by using recursion

Assignment Name:-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.

Objective-

- To understand the assembly language program
- To understand the concept of recursion
- Able to Implement factorial of a integer number using recursive method

Outcome-

- Students will be able to understand different assembly language instruction.
- Students will be familiar with the format of assembly language program and able to Apply the concept of recursion to find factorial of a number

Prerequisite -

System call of Unix for Assembly language Program.

Hardware Requirement-

Desktop PC

Software Requirement-

Ubuntu 14.04,
Assembler: NASM version 2.10.07
Linker: ld

Introduction:-

THEORY:

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

second procedure, which in turn calls the first procedure.

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

```
Fact (n) = n * fact (n-1) for n > 0
```

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

Instructions needed:

1. AND-AND each bit in a byte or word with corresponding bit in another byte or word
2. INC-Increments specified byte/word by1
3. DEC-Decrements specified byte/word by1
4. JG - The command JG simply means: Jump if Greater.
5. CMP-Compares to specified bytes or words
6. MUL - The MUL (Multiply) instruction handles unsigned data
7. CALL-Transfers the control from callingprogramto procedure.
8. ADD- ADD instructions are used for performing simple addition of binary data in byte, word and doubleword size, i.e., for adding 8-bit, 16-bit or 32-bit operands, respectively.
9. RET-Return from where call is made

Algorithm:-

This algorithm use recursive approach to find factorial of N.

1. Start
2. Read: Take input N
3. Retrieve parameter and put it into Register-PUSH

4. Check for base case if n==0
5. move the first argument to %rax
6. If the number is 1, that is our base case, and we simply return.
7. multiply by the result of the last call to factorial.
8. return to the function

Conclusion:-

Questions:-

1. What is Control transfer instructions. Explain in details
2. What different conditions used to find factorial of an integer number.
3. Explain CALL, JG, ADD instructions
4. Explain Pop and Push instruction in detail.

SUBJECT: MICROPROCESSOR LAB (MPL)	
NAME:	
CLASS:SE COMP	**ROLL NO.:**
SEMESTER:SEM-II	**YEAR:2023-24**
DATE OF PERFORMANCE:	**DATE OF SUBMISSION:**
EXAMINED:	

Assignment No-10

Title:- Study Assignment

Assignment Name:-

Motherboards are complex. Break them down, component by component, and Understand how they work. Choosing a motherboard is a hugely important part of building a PC. Study- Block diagram, Processor Socket, Expansion Slots, SATA, RAM, Form Factor, BIOS, Internal Connectors, External Ports, Peripherals and Data Transfer, Display, Audio

Objective-

- To understand the Form Factor of Motherboard
- To understand different components of Motherboard

Outcome-

- Students will be able to identify components of Motherboard
- Students will be able to distinguished south bridge and north bridge and its working

Prerequisite -

Computer Fundamental.

Hardware Requirement-

Desktop PC,Motherboard,SMPS

Introduction:-

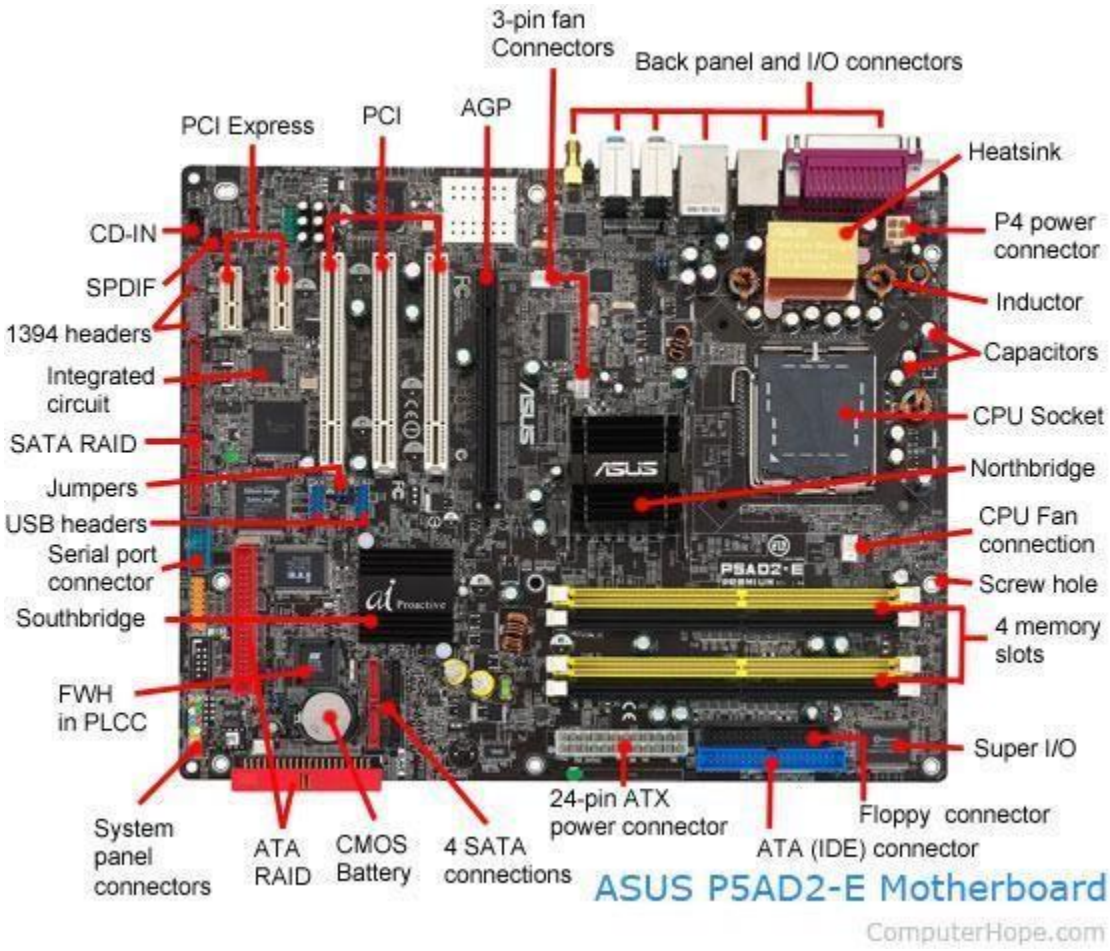
THEORY:

Motherboard overview

A motherboard provides connectivity between the hardware components of a computer, like the processor (CPU), memory (RAM), hard drive, and video card. There are multiple types of motherboards, designed to fit different types and sizes of computers.

Each type of motherboard is designed to work with specific types of processors and memory, so they don't work with every processor and type of memory. However, hard drives are mostly universal and work with the majority of motherboards, regardless of the type or brand.

Below is a picture of the ASUS P5AD2-E motherboard with labels next to each of its major components. Clicking the image directs you to a larger and more detailed version.



Where is the motherboard located?

A computer motherboard is located inside the computer case and is where most of the parts and computer peripherals connect. With tower computers, the motherboard is on the left or right side of the tower and is the biggest circuit board.

Motherboard components

Below are links to pages with more details for each of the motherboard components mentioned in the previous section. The links are listed in clockwise order starting from the top-left corner of the image. Components not labeled on the image above are found in sections later on this page.

- Expansion slots (PCI Express, PCI, and AGP)
- 3-pin case fan connectors
- Back pane connectors
- Heat sink
- 4-pin (P4) power connector

- Inductor
- Capacitor
- CPU socket
- Northbridge
- Screw hole
- Memory slot
- Super I/O
- ATA / IDE disk drive primary connection
- 24-pin ATX power supply connector
- Serial ATA connections
- Coin cell battery (CMOS backup battery)
- RAID
- System panel connectors
- FWH
- Southbridge
- Serial port connector
- USB headers
- Jumpers
- Integrated circuit
- 1394 headers
- SPDIF
- CD-IN

Older motherboard components

The following list contains links to components that are not shown in the picture above or were part of older computer motherboards.

- BIOS
- Bus
- Cache memory
- Chipset
- Diode
- Dip switches
- Electrolytic
- Floppy connection

Questions-List the different components of Motherboard