Assignment 2

1. Compute and explain real as well as virtual memory of 80386DX in real and protected

mode.

Real Mode:

In real mode, the 80386DX processor operates similarly to its predecessors, such as the 8086 and 80286 processors. It features a 20-bit address bus, which allows it to address up to 1 MB of physical memory directly. Memory accesses are performed using physical addresses, meaning that the processor interacts directly with the physical memory chips installed in the system. In real mode, there is no memory protection or multitasking support. Programs have full access to all memory locations, and there are no restrictions on memory accesses. As a result, software running in real mode can easily access any memory location and execute instructions without encountering any protection violations.

Protected Mode:

Virtual Memory:

- In protected mode, the processor can access up to 4 GB of virtual memory through paging and segmentation mechanisms. Virtual memory allows the operating system to present a larger, contiguous address space to each program, even though physical memory may be fragmented or limited in size.

- Paging divides the virtual address space into fixed-size pages, typically 4 KB in size, and maps these pages to physical memory using a page table. The Memory Management Unit (MMU) translates virtual addresses to physical addresses, enabling the processor to work with virtual memory efficiently.

- Segmentation allows for the creation of multiple memory segments with different access permissions, such as code segments, data segments, and stack segments. Each segment has its own base address, limit, and access rights, providing fine-grained control over memory access.

Memory Protection:

- Memory protection mechanisms in protected mode prevent unauthorized access to memory regions, enhancing system security and reliability. Each memory segment is associated with access rights that specify whether the segment is readable, writable, or executable, and whether access is restricted to privileged or non-privileged code.

- Access violations, such as attempts to write to read-only memory or execute data segments, are detected by the processor and result in a fault or exception, allowing the operating system to handle the error gracefully.

2. Write and explain 4 bit accept and display procedure to be used in 80386DX ALP. Explain the importance of packing and unpacking of numbers in 80386DX ALP.

section .data ; Message prompts for the user

msg\_prompt db 'Enter a 4-bit number (0-9, A-F): ', 0

msg\_display db 'You entered: ', 0

section .bss ; Variables to store user input and result

input resb 1

result resb 1

section .text

global \_start

\_start:

; Display prompt asking the user to enter a 4-bit number

mov eax, 4 ; System call for write

mov ebx, 1 ; File descriptor for stdout

mov ecx, msg\_prompt ; Address of the prompt message

mov edx, 35 ; Length of the prompt message

int 0x80 ; Call kernel to display the prompt

; Read user input

mov eax, 3 ; System call for read

mov ebx, 2 ; File descriptor for stdin

mov ecx, input ; Address to store the input

mov edx, 1 ; Number of bytes to read

int 0x80 ; Call kernel to read user input

; Display message indicating the input received

mov eax, 4 ; System call for write

mov ebx, 1 ; File descriptor for stdout

mov ecx, msg\_display ; Address of the display message

mov edx, 13 ; Length of the display message

int 0x80 ; Call kernel to display the message

; Display the input received from the user

mov eax, 4 ; System call for write

mov ebx, 1 ; File descriptor for stdout

mov ecx, input ; Address of the input

mov edx, 1 ; Length of the input

int 0x80 ; Call kernel to display the input

; Exit the program

mov eax, 1 ; System call for exit

mov ebx, 0 ; Exit code 0

int 0x80 ; Call kernel to exit the program

Packing:

Packing involves combining multiple smaller data elements into a single larger data element. For example, packing two 4-bit numbers into a single 8-bit byte. Packing allows you to optimize memory usage by reducing the number of bytes required to store data. It also simplifies data manipulation by treating multiple smaller elements as a single unit.

Unpacking:

Unpacking is the reverse process of packing, where a larger data element is split into multiple smaller data elements. Unpacking is essential for extracting individual data elements from packed data. It enables data processing on individual elements and is necessary for proper interpretation of data.

3. Write a menu driven 80386 ALP to compute 4 -bit multiplication using shift & add mechanism.

section .data

menu\_msg db 'Menu:', 10

db '1. Perform multiplication', 10

db '2. Exit', 10

db 'Enter your choice: ', 0

result\_msg db 'Result: ', 0

section .bss

choice resb 1

num1 resb 1

num2 resb 1

result resb 2

section .text

global \_start

\_start:

; Display menu options to the user

mov eax, 4 ; System call for write

mov ebx, 1 ; File descriptor for stdout

mov ecx, menu\_msg ; Address of the message to display

mov edx, 43 ; Length of the message

int 0x80 ; Call kernel to display the menu

; Get user's choice from the menu

mov eax, 3 ; System call for read

mov ebx, 2 ; File descriptor for stdin

mov ecx, choice ; Address to store the user's choice

mov edx, 1 ; Number of bytes to read

int 0x80 ; Call kernel to read the choice

; Perform operation based on the user's choice

cmp byte [choice], '1' ; Compare the user's choice with '1'

je perform\_multiplication ; Jump to perform\_multiplication if choice is '1'

cmp byte [choice], '2' ; Compare the user's choice with '2'

je exit\_program ; Jump to exit\_program if choice is '2'

perform\_multiplication:

mov ecx, 4 ; Loop counter for 4 iterations (4 bits)

movzx eax, byte [num1] ; Load the first 4-bit number into EAX

movzx ebx, byte [num2] ; Load the second 4-bit number into EBX

xor edx, edx ; Clear EDX to store the result

multiply\_loop: ; Check the least significant bit of the second number

test bl, 1

jz multiply\_continue ; If the bit is 0, skip addition

add edx, eax; Add the first number to the result

multiply\_continue: ; Shift both numbers to the right by 1 bit

shr eax, 1

rcr ebx, 1

loop multiply\_loop; Decrement the loop counter and loop if not zero

mov [result], dl; Store the result in the 'result' variable

ret

; Display the result to the user

mov eax, 4 ; System call for write

mov ebx, 1 ; File descriptor for stdout

mov ecx, result\_msg ; Address of the message to display

mov edx, 8 ; Length of the message

int 0x80 ; Call kernel to display the result

jmp main ; Jump back to the main menu

exit\_program: ; Exit the program

mov eax, 1 ; System call for exit

mov ebx, 0 ; Exit code 0

int 0x80 ; Call kernel to exit the program

4. What is the role of Task Register in multitasking? What are the instructions used to

modify and read TR?

The Task Register (TR) in the 80386DX processor plays a pivotal role in multitasking environments by facilitating efficient task switching and management. Its primary purpose is to store the Task State Segment (TSS) selector for the currently executing task. The TSS contains essential information about the state of a task, including register values, stack pointers, and privilege level. By storing the TSS selector in the TR, the processor can quickly access the relevant TSS when performing a task switch.

When multitasking, the processor needs to switch between different tasks seamlessly to ensure smooth execution of multiple programs concurrently. The TR enables this functionality by providing a quick and efficient way to identify the TSS associated with the currently running task. During a task switch, the processor loads the TSS selector from the TR and uses it to locate the corresponding TSS in memory, allowing it to retrieve the task's state and resume execution from where it left off.

Instructions such as LTR (Load Task Register) and STR (Store Task Register) are used to modify and read the TR, respectively. The LTR instruction is used to load a new TSS selector into the TR, allowing the processor to switch to a different task. This instruction takes the TSS selector as an operand and updates the TR accordingly. On the other hand, the STR instruction is used to read the current value of the TR, allowing the operating system or privileged software to retrieve information about the currently executing task.

5. With neatly labelled diagram explain software initialization process of 80386DX in V86 mode.

Reset State:

After a reset, the 80386DX starts in real-address mode. In this mode, it behaves like an enhanced 8086 processor, allowing execution of 16-bit code.

Switch to V86 Mode:

The CPU can transition from real mode to V86 mode. In V86 mode, the processor emulates an 8086 environment within a protected mode context. This allows running 16-bit programs alongside 32-bit protected mode code.

Initialization Steps in V86 Mode:

Load Segment Registers: Set up segment registers (CS, DS, ES, SS) with appropriate values for the V86 task.

Set Flags: Initialize flags (EFLAGS) to match the 8086 environment.

Enter V86 Mode: Trigger the switch to V86 mode.

Execute 8086 Code: The CPU now executes 8086 instructions.

Leave V86 Mode: When needed, exit V86 mode and return to protected mode execution.