COA ASSIGNMENT

3. Write an ALP to accept the 64 bit numbers from user, perform arithmetic operations on them, and display the result. i) Addition ii) Division

CODE :

```
;-------------------------------------------
;   MACROS
;-------------------------------------------

%macro PRINT 2
   mov rax, 1
   mov rdi, 1
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

%macro READ 2
   mov rax, 0
   mov rdi, 0
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

%macro EXIT 0
   mov rax, 60
   xor rdi, rdi
   syscall
%endmacro
```

```
;------------------------------------------
section .data
msg1    db "Enter first number: ",0
len1    equ $ - msg1

msg2    db "Enter second number: ",0
len2    equ $ - msg2

msg_add db "Addition Result: ",0
len_add equ $ - msg_add

msg_div db "Division Result: ",0
len_div equ $ - msg_div

msg_rem db "Remainder: ",0
len_rem equ $ - msg_rem

newline db 10

;------------------------------------------
section .bss
num1   resq 1
num2   resq 1
result resq 1
buffer resb 20

;------------------------------------------
section .text
global _start

_start:

    ;--- Read First Number ---
    PRINT msg1, len1
    READ buffer, 20
    call string_to_int
```

```
mov [num1], rax

;--- Read Second Number ---
PRINT msg2, len2
READ buffer, 20
call string_to_int
mov [num2], rax

;============================================================
; ADDITION
;============================================================
mov rax, [num1]
add rax, [num2]
mov [result], rax

PRINT msg_add, len_add
mov rax, [result]
call print_number
PRINT newline, 1

;============================================================
; DIVISION
;============================================================
mov rax, [num1]
xor rdx, rdx
mov rbx, [num2]
div rbx             ; quotient = RAX, remainder = RDX
mov [result], rax
push rdx            ; Save remainder

PRINT msg_div, len_div
mov rax, [result]
call print_number
PRINT newline, 1

PRINT msg_rem, len_rem
pop rax
```

```
    call print_number
    PRINT newline, 1

    EXIT

;============================================================
; Convert ASCII string in buffer → integer in RAX
;============================================================
string_to_int:
    xor rax, rax
    xor rcx, rcx
.loop:
    movzx rbx, byte [buffer + rcx]
    cmp rbx, 10
    je .done
    cmp rbx, 0
    je .done

    sub rbx, '0'
    imul rax, 10
    add rax, rbx
    inc rcx
    jmp .loop
.done:
    ret

;============================================================
; Print unsigned integer in RAX
;============================================================
print_number:
    push rax
    push rbx
    push rcx
    push rdx

    mov rcx, 0
    mov rbx, 10
```

```
.divide_loop:
    xor rdx, rdx
    div rbx
    add dl, '0'
    push rdx
    inc rcx
    test rax, rax
    jnz .divide_loop

.print_loop:
    pop rax
    mov [buffer], al
    push rcx
    PRINT buffer, 1
    pop rcx
    loop .print_loop

    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret
```

4.Write an ALP to accept the 64 bit numbers from user, perform arithmetic operations on them, and display the result.i) Multiplication ii) Subtraction

```
;=========================================
==================
;                    MACROS
;=========================================
==================

%macro PRINT 2
   mov rax, 1
   mov rdi, 1
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

%macro READ 2
   mov rax, 0
   mov rdi, 0
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

%macro EXIT 0
   mov rax, 60
   xor rdi, rdi
```

```
    syscall
%endmacro


;=============================================
====================
section .data
msg1    db "Enter first number: ",0
len1    equ $ - msg1

msg2    db "Enter second number: ",0
len2    equ $ - msg2

msg_mul db "Multiplication Result: ",0
len_mul equ $ - msg_mul

msg_sub db "Subtraction Result: ",0
len_sub equ $ - msg_sub

newline db 10


;=============================================
====================
section .bss
```

```
num1   resq 1
num2   resq 1
result resq 1
buffer resb 20


;=======================================================
===================
section .text
global _start

_start:

    ;----------------------------
    ; READ FIRST NUMBER
    ;----------------------------
    PRINT msg1, len1
    READ buffer, 20
    call string_to_int
    mov [num1], rax

    ;----------------------------
    ; READ SECOND NUMBER
    ;----------------------------
    PRINT msg2, len2
```

```
    READ buffer, 20
    call string_to_int
    mov [num2], rax



;==============================================================
    ; MULTIPLICATION

;==============================================================
    mov rax, [num1]
    mov rbx, [num2]
    imul rax, rbx
    mov [result], rax

    PRINT msg_mul, len_mul
    mov rax, [result]
    call print_number
    PRINT newline, 1
```

```
;===============================================
================
    ; SUBTRACTION

;===============================================
================
    mov rax, [num1]
    sub rax, [num2]
    mov [result], rax

    PRINT msg_sub, len_sub
    mov rax, [result]
    call print_number
    PRINT newline, 1


    EXIT


;===============================================
=====================
; Convert ASCII → Integer
;===============================================
=====================
```

```
string_to_int:
    xor rax, rax
    xor rcx, rcx
.loop:
    movzx rbx, byte [buffer + rcx]
    cmp rbx, 10
    je .done
    cmp rbx, 0
    je .done

    sub rbx, '0'
    imul rax, 10
    add rax, rbx
    inc rcx
    jmp .loop
.done:
    ret


;================================================
=====================
; Print integer in RAX
;================================================
=====================
print_number:
```

```
    push rax
    push rbx
    push rcx
    push rdx

    mov rcx, 0
    mov rbx, 10

.divide_loop:
    xor rdx, rdx
    div rbx
    add dl, '0'
    push rdx
    inc rcx
    test rax, rax
    jnz .divide_loop

.print_loop:
    pop rax
    mov [buffer], al
    push rcx
    PRINT buffer, 1
    pop rcx
    loop .print_loop
```

```
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret
```

5.

CODE:

```
%macro READ 2
mov rax, 0 ; sys_read
mov rdi, 0 ; stdin
mov rsi, %1 ; buffer
mov rdx, %2 ; length
syscall
%endmacro
%macro WRITE 2
mov rax, 1 ; sys_write
mov rdi, 1 ; stdout
mov rsi, %1 ; buffer
mov rdx, %2 ; length
syscall
%endmacro
section .data
msg1 db "Enter the HEX number (up to 16 digits): ", 10

len1 equ $-msg1
```

```
msg2 db "The BCD Equivalent: ", 10
len2 equ $-msg2
newline db 10

section .bss
char_buff resb 17 ; Buffer for hex input
len resq 1 ; Length of input
ans resq 1 ; Store converted number
char resb 1 ; Single character buffer
cnt resb 1 ; Digit counter

section .text

global _start

_start:

; Display prompt
WRITE msg1, len1
; Read hex number

READ char_buff, 17
dec rax ; Subtract newline
mov [len], rax ; Store length
; Convert hex string to binary
mov rcx, [len]
call hex_to_binary
mov [ans], rax ; Store result
```

```
; Display result message
WRITE msg2, len2
; Convert binary to BCD and display
mov rax, [ans]
call binary_to_bcd
; Print newline
WRITE newline, 1
; Exit
mov rax, 60
mov rdi, 0
syscall
; Convert hex string to binary
; Input: rcx = length, char_buff = hex string
; Output: rax = binary value
hex_to_binary:
push rbx
push rcx
push rdx
push rsi
mov rsi, char_buff
mov rax, 0 ; Result accumulator
.loop:
movzx rbx, byte [rsi] ; Get character

; Check if 0-9
cmp bl,'9'
jbe .is_digit
;
```

```
 Check if A-F
cmp bl,'F'
jbe .is_upper
; Must be a-f
sub bl,'a'
add bl, 10
jmp .add_nibble

.is_upper:
sub bl,'A'
add bl, 10
jmp .add_nibble

.is_digit:
sub bl,'0'

.add_nibble:
shl rax, 4 ; Shift left 4 bits
add rax, rbx ; Add the nibble
inc rsi
dec rcx
jnz .loop
pop rsi
pop rdx
pop rcx
pop rbx
ret
; Convert binary to BCD and display
```

```
; Input: rax = number to convert
binary_to_bcd:
push rax
push rbx
push rcx
push rdx
mov rcx, 0 ; Digit counter
; Extract digits by dividing by 10
.divide_loop:
mov rdx, 0
mov rbx, 10
div rbx ; rax = quotient, rdx = remainder (digit)
push rdx ; Save digit on stack
inc rcx ; Count digits
cmp rax, 0
jne .divide_loop

; Store digit count
mov byte [cnt], cl
; Print digits from stack
.print_loop:
pop rbx ; Get digit
add bl,'0' ; Convert to ASCII

mov byte [char], bl
WRITE char, 1 ; Print digit
dec byte [cnt]
jnz .print_loop
```

```
pop rdx
pop rcx
pop rbx
pop rax
ret
```

6 .Write an ALP to convert 64 bit BCD number into its equivalent HEX number

CODE :

```
%macro READ 2
    mov rax, 0          ; sys_read
    mov rdi, 0          ; stdin
    mov rsi, %1         ; buffer
    mov rdx, %2         ; length
    syscall
%endmacro

%macro WRITE 2
    mov rax, 1          ; sys_write
    mov rdi, 1          ; stdout
    mov rsi, %1         ; buffer
    mov rdx, %2         ; length
    syscall
%endmacro

section .data
```

```
    msg1 db "Enter the BCD number (decimal digits): ", 10
    len1 equ $-msg1

    msg2 db "The HEX Equivalent: ", 10
    len2 equ $-msg2

    newline db 10

section .bss
    char_buff resb 20   ; Buffer for BCD input
    hex_buff resb 17    ; Buffer for hex output
    len resq 1          ; Length of input
    ans resq 1          ; Store converted number

section .text
    global _start

_start:
    ; Display prompt
    WRITE msg1, len1

    ; Read BCD number (decimal input)
    READ char_buff, 20
    dec rax             ; Subtract newline
    mov [len], rax      ; Store length

    ; Convert BCD (decimal string) to binary
    call bcd_to_binary
    mov [ans], rax      ; Store result

    ; Display result message
    WRITE msg2, len2

    ; Convert binary to hex and display
    mov rax, [ans]
```

```
    call binary_to_hex

    ; Print newline
    WRITE newline, 1

    ; Exit
    mov rax, 60
    mov rdi, 0
    syscall

; Convert BCD (decimal string) to binary
; Input: char_buff = decimal string, [len] = length
; Output: rax = binary value
bcd_to_binary:
    push rbx
    push rcx
    push rdx
    push rsi

    mov rsi, char_buff
    mov rcx, [len]
    mov rax, 0          ; Result accumulator
    mov rbx, 10

.loop:
    ; Multiply current result by 10
    mul rbx             ; rax = rax * 10

    ; Get next digit
    movzx rdx, byte [rsi]
    sub dl, '0'         ; Convert ASCII to digit

    ; Add digit to result
    add rax, rdx
```

```
    inc rsi
    dec rcx
    jnz .loop

    pop rsi
    pop rdx
    pop rcx
    pop rbx
    ret

; Convert binary to hex string and display
; Input: rax = number to convert
binary_to_hex:
    push rax
    push rbx
    push rcx
    push rdx
    push rsi

    mov rsi, hex_buff
    mov rcx, 16         ; 16 hex digits for 64-bit number
    mov rbx, rax        ; Number to convert

.loop:
    rol rbx, 4          ; Rotate left 4 bits to get next nibble
    mov dl, bl
    and dl, 0x0F        ; Mask to get lower 4 bits

    ; Convert nibble to hex character
    cmp dl, 9
    jbe .is_digit

    ; A-F
    add dl, 'A' - 10
    jmp .store
```

```
.is_digit:
    add dl, '0'

.store:
    mov byte [rsi], dl
    inc rsi
    dec rcx
    jnz .loop

    ; Find first non-zero digit to avoid leading zeros
    mov rsi, hex_buff
    mov rcx, 16

.skip_zeros:
    cmp byte [rsi], '0'
    jne .print_start
    inc rsi
    dec rcx
    jnz .skip_zeros

    ; If all zeros, print one zero
    mov byte [hex_buff], '0'
    WRITE hex_buff, 1
    jmp .done

.print_start:
    ; Print from first non-zero digit
    WRITE rsi, rcx

.done:
    pop rsi
    pop rdx
    pop rcx
    pop rbx
```

```
    pop rax
    ret
```

7. Write an ALP to perform multiplication of two 64-bit hexadecimal numbers using successive addition

CODE :

```
%macro READ 2
    mov rax, 0          ; sys_read
    mov rdi, 0          ; stdin
    mov rsi, %1         ; buffer
    mov rdx, %2         ; length
    syscall
%endmacro

%macro WRITE 2
    mov rax, 1          ; sys_write
    mov rdi, 1          ; stdout
    mov rsi, %1         ; buffer
    mov rdx, %2         ; length
    syscall
%endmacro
```

```
section .data
    msg1 db "Enter first HEX number: ", 10
    len1 equ $-msg1

    msg2 db "Enter second HEX number: ", 10
    len2 equ $-msg2

    msg3 db "Multiplication Result (HEX): ", 10
    len3 equ $-msg3

    msg4 db "Multiplication Result (Decimal): ", 10
    len4 equ $-msg4

    newline db 10

section .bss
    char_buff resb 17   ; Buffer for hex input
    hex_buff resb 17    ; Buffer for hex output
    len resq 1          ; Length of input
    num1 resq 1         ; First number
    num2 resq 1         ; Second number
    result resq 1       ; Result of multiplication

section .text
    global _start

_start:
    ; Get first hex number
    WRITE msg1, len1
    READ char_buff, 17
    dec rax
    mov [len], rax
    call hex_to_binary
    mov [num1], rax
```

```
    ; Get second hex number
    WRITE msg2, len2
    READ char_buff, 17
    dec rax
    mov [len], rax
    call hex_to_binary
    mov [num2], rax

    ; Perform multiplication using successive addition
    call multiply_successive_add
    mov [result], rax

    ; Display result in hexadecimal
    WRITE msg3, len3
    mov rax, [result]
    call display_hex
    WRITE newline, 1

    ; Display result in decimal
    WRITE msg4, len4
    mov rax, [result]
    call display_decimal
    WRITE newline, 1

    ; Exit
    mov rax, 60
    mov rdi, 0
    syscall

; Convert hex string to binary
; Input: char_buff = hex string, [len] = length
; Output: rax = binary value
hex_to_binary:
    push rbx
    push rcx
```

```
    push rdx
    push rsi

    mov rsi, char_buff
    mov rcx, [len]
    mov rax, 0          ; Result accumulator

.loop:
    movzx rbx, byte [rsi]

    ; Check if 0-9
    cmp bl, '9'
    jbe .is_digit

    ; Check if A-F
    cmp bl, 'F'
    jbe .is_upper

    ; Must be a-f
    sub bl, 'a'
    add bl, 10
    jmp .add_nibble

.is_upper:
    sub bl, 'A'
    add bl, 10
    jmp .add_nibble

.is_digit:
    sub bl, '0'

.add_nibble:
    shl rax, 4          ; Shift left 4 bits
    add rax, rbx        ; Add the nibble
```

```
    inc rsi
    dec rcx
    jnz .loop

    pop rsi
    pop rdx
    pop rcx
    pop rbx
    ret

; Multiply using successive addition
; Input: [num1], [num2]
; Output: rax = num1 * num2
multiply_successive_add:
    push rbx
    push rcx

    mov rax, 0          ; Result = 0
    mov rbx, [num1]     ; Number to add repeatedly
    mov rcx, [num2]     ; Counter (how many times to add)

    ; Check if counter is 0
    cmp rcx, 0
    je .done

.add_loop:
    add rax, rbx        ; Add num1 to result
    dec rcx             ; Decrease counter
    jnz .add_loop       ; Continue if counter not zero

.done:
    pop rcx
    pop rbx
    ret
```

```
; Display number in hexadecimal
; Input: rax = number to display
display_hex:
    push rax
    push rbx
    push rcx
    push rdx
    push rsi

    mov rsi, hex_buff
    mov rcx, 16        ; 16 hex digits
    mov rbx, rax

.loop:
    rol rbx, 4         ; Rotate left 4 bits
    mov dl, bl
    and dl, 0x0F       ; Mask lower 4 bits

    ; Convert to hex character
    cmp dl, 9
    jbe .is_digit
    add dl, 'A' - 10
    jmp .store

.is_digit:
    add dl, '0'

.store:
    mov byte [rsi], dl
    inc rsi
    dec rcx
    jnz .loop

    ; Skip leading zeros
    mov rsi, hex_buff
```

```
    mov rcx, 16

.skip_zeros:
    cmp byte [rsi], '0'
    jne .print
    inc rsi
    dec rcx
    cmp rcx, 1          ; Keep at least one digit
    jne .skip_zeros

.print:
    WRITE rsi, rcx

    pop rsi
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret

; Display number in decimal
; Input: rax = number to display
display_decimal:
    push rax
    push rbx
    push rcx
    push rdx

    mov rcx, 0          ; Digit counter
    mov rbx, 10

    ; Check if number is 0
    cmp rax, 0
    jne .divide_loop
```

```
    ; Print single '0'
    mov byte [hex_buff], '0'
    WRITE hex_buff, 1
    jmp .done

.divide_loop:
    mov rdx, 0
    div rbx             ; Divide by 10
    add dl, '0'         ; Convert remainder to ASCII
    push rdx            ; Save digit
    inc rcx
    cmp rax, 0
    jne .divide_loop

.print_loop:
    pop rax
    mov byte [hex_buff], al
    push rcx
    WRITE hex_buff, 1
    pop rcx
    loop .print_loop

.done:
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret
```

8 .Write an ALP to perform multiplication of two 64-bit hexadecimal numbers using add and shift method.

CODE :

```
%macro READ 2
    mov rax, 0         ; sys_read
    mov rdi, 0         ; stdin
    mov rsi, %1        ; buffer
    mov rdx, %2        ; length
    syscall
%endmacro

%macro WRITE 2
    mov rax, 1         ; sys_write
    mov rdi, 1         ; stdout
    mov rsi, %1        ; buffer
    mov rdx, %2        ; length
    syscall
%endmacro

section .data
    msg1 db "Enter first HEX number: ", 10
    len1 equ $-msg1

    msg2 db "Enter second HEX number: ", 10
    len2 equ $-msg2

    msg3 db "Multiplication Result (HEX): ", 10
    len3 equ $-msg3
```

```
    msg4 db "Multiplication Result (Decimal): ", 10
    len4 equ $-msg4

    newline db 10

section .bss
    char_buff resb 17   ; Buffer for hex input
    hex_buff resb 17    ; Buffer for hex output
    len resq 1          ; Length of input
    num1 resq 1         ; First number (multiplicand)
    num2 resq 1         ; Second number (multiplier)
    result resq 1       ; Result of multiplication

section .text
    global _start

_start:
    ; Get first hex number
    WRITE msg1, len1
    READ char_buff, 17
    dec rax
    mov [len], rax
    call hex_to_binary
    mov [num1], rax

    ; Get second hex number
    WRITE msg2, len2
    READ char_buff, 17
```

```
    dec rax
    mov [len], rax
    call hex_to_binary
    mov [num2], rax

    ; Perform multiplication using add and shift method
    call multiply_add_shift
    mov [result], rax

    ; Display result in hexadecimal
    WRITE msg3, len3
    mov rax, [result]
    call display_hex
    WRITE newline, 1

    ; Display result in decimal
    WRITE msg4, len4
    mov rax, [result]
    call display_decimal
    WRITE newline, 1

    ; Exit
    mov rax, 60
    mov rdi, 0
    syscall

; Convert hex string to binary
; Input: char_buff = hex string, [len] = length
```

```
; Output: rax = binary value
hex_to_binary:
    push rbx
    push rcx
    push rdx
    push rsi

    mov rsi, char_buff
    mov rcx, [len]
    mov rax, 0          ; Result accumulator

.loop:
    movzx rbx, byte [rsi]

    ; Check if 0-9
    cmp bl, '9'
    jbe .is_digit

    ; Check if A-F
    cmp bl, 'F'
    jbe .is_upper

    ; Must be a-f
    sub bl, 'a'
    add bl, 10
    jmp .add_nibble

.is_upper:
```

```
    sub bl, 'A'
    add bl, 10
    jmp .add_nibble

.is_digit:
    sub bl, '0'

.add_nibble:
    shl rax, 4          ; Shift left 4 bits
    add rax, rbx        ; Add the nibble

    inc rsi
    dec rcx
    jnz .loop

    pop rsi
    pop rdx
    pop rcx
    pop rbx
    ret

; Multiply using add and shift method (Booth's algorithm
concept)
; Algorithm: Check each bit of multiplier, if bit is 1, add
shifted multiplicand
; Input: [num1] = multiplicand, [num2] = multiplier
; Output: rax = num1 * num2
multiply_add_shift:
```

```
    push rbx
    push rcx
    push rdx

    mov rax, 0          ; Result = 0
    mov rbx, [num1]     ; Multiplicand
    mov rcx, [num2]     ; Multiplier
    mov rdx, rbx        ; Copy of multiplicand for shifting

    ; Check if multiplier is 0
    cmp rcx, 0
    je .done

.shift_loop:
    ; Check if least significant bit of multiplier is 1
    test rcx, 1
    jz .skip_add        ; If bit is 0, skip addition

    ; Add shifted multiplicand to result
    add rax, rdx

.skip_add:
    ; Shift multiplicand left (multiply by 2)
    shl rdx, 1

    ; Shift multiplier right (divide by 2)
    shr rcx, 1
```

```
    ; Continue if multiplier is not zero
    jnz .shift_loop

.done:
    pop rdx
    pop rcx
    pop rbx
    ret

; Display number in hexadecimal
; Input: rax = number to display
display_hex:
    push rax
    push rbx
    push rcx
    push rdx
    push rsi

    mov rsi, hex_buff
    mov rcx, 16        ; 16 hex digits
    mov rbx, rax

.loop:
    rol rbx, 4         ; Rotate left 4 bits
    mov dl, bl
    and dl, 0x0F       ; Mask lower 4 bits

    ; Convert to hex character
```

```
    cmp dl, 9
    jbe .is_digit
    add dl, 'A' - 10
    jmp .store

.is_digit:
    add dl, '0'

.store:
    mov byte [rsi], dl
    inc rsi
    dec rcx
    jnz .loop

    ; Skip leading zeros
    mov rsi, hex_buff
    mov rcx, 16

.skip_zeros:
    cmp byte [rsi], '0'
    jne .print
    inc rsi
    dec rcx
    cmp rcx, 1          ; Keep at least one digit
    jne .skip_zeros

.print:
    WRITE rsi, rcx
```

```
    pop rsi
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret

; Display number in decimal
; Input: rax = number to display
display_decimal:
    push rax
    push rbx
    push rcx
    push rdx

    mov rcx, 0          ; Digit counter
    mov rbx, 10

    ; Check if number is 0
    cmp rax, 0
    jne .divide_loop

    ; Print single '0'
    mov byte [hex_buff], '0'
    WRITE hex_buff, 1
    jmp .done
```

```
.divide_loop:
    mov rdx, 0
    div rbx             ; Divide by 10
    add dl, '0'         ; Convert remainder to ASCII
    push rdx            ; Save digit
    inc rcx
    cmp rax, 0
    jne .divide_loop

.print_loop:
    pop rax
    mov byte [hex_buff], al
    push rcx
    WRITE hex_buff, 1
    pop rcx
    loop .print_loop

.done:
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret
```

9. Write a menu driven ALP to implement various mentioned string operations. i)String length ii) String Concatination iii) String palindrome

CODE :

```
; 64-bit Linux Assembly Program for String Operations

%macro read 2
   mov rax, 0
   mov rdi, 0
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

%macro write 2
   mov rax, 1
   mov rdi, 1
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

section .data
   msg_menu db 10, "===== STRING OPERATIONS =====",10
          db "1. String Length",10
          db "2. String Concatenation",10
          db "3. String Palindrome",10
          db "4. Exit",10
          db "Enter your choice (1-4): "
   len_menu equ $ - msg_menu

   msg_input1 db 10,"Enter String1: "
   len_input1 equ $ - msg_input1

   msg_input2 db 10,"Enter String2: "
   len_input2 equ $ - msg_input2

   msg_length db 10,"Length of string: "
   len_length equ $ - msg_length
```

```
    msg_concat db 10,"Concatenated string: "
    len_concat equ $ - msg_concat

    msg_palin_ymses db 10,"String is PALINDROME",10
    len_palin_yes equ $ - msg_palin_yes

    msg_palin_no db 10,"String is NOT palindrome",10
    len_palin_no equ $ - msg_palin_no

    msg_invalid db 10,"Invalid choice!",10
    len_invalid equ $ - msg_invalid

    msg_exit db 10,"Exiting program...",10
    len_exit equ $ - msg_exit

    newline db 10

section .bss
    string1 resb 50
    string2 resb 50
    string3 resb 100
    l1 resq 1
    l2 resq 1
    l3 resq 1
    choice resb 2
    buff resb 16

section .text
global _start

_start:
printmenu:
    write msg_menu, len_menu
    read choice, 2

    cmp byte[choice], '1'
```

```
    je strlen
    cmp byte[choice], '2'
    je strconcat
    cmp byte[choice], '3'
    je strpalindrome
    cmp byte[choice], '4'
    je exit

    write msg_invalid, len_invalid
    jmp printmenu

; ==========================================
; 1. String Length
; ==========================================
strlen:
    write msg_input1, len_input1
    read string1, 50
    dec rax
    mov [l1], rax

    write msg_length, len_length
    mov rbx, [l1]
    call display

    jmp printmenu

; ==========================================
; 2. String Concatenation
; ==========================================
strconcat:
    write msg_input1, len_input1
    read string1, 50
    dec rax
    mov [l1], rax

    write msg_input2, len_input2
    read string2, 50
```

```
    dec rax
    mov [l2], rax

    ; Copy string1 to string3
    mov rsi, string1
    mov rdi, string3
    mov rcx, [l1]
    cld
    rep movsb

    ; Copy string2 after string1
    mov rsi, string2
    mov rcx, [l2]
    rep movsb

    ; Calculate total length
    mov rbx, [l1]
    add rbx, [l2]
    mov [l3], rbx

    write msg_concat, len_concat
    write string3, [l3]
    write newline, 1

    jmp printmenu

; ==========================================
; 3. Palindrome Check
; ==========================================
strpalindrome:
    write msg_input1, len_input1
    read string1, 50
    dec rax
    mov [l1], rax

    ; Reverse string1 into string3
    mov rsi, string1
```

```
    add rsi, [l1]
    dec rsi
    mov rdi, string3
    mov rcx, [l1]

pal_loop:
    mov dl, [rsi]
    mov [rdi], dl
    dec rsi
    inc rdi
    loop pal_loop

    ; Compare original and reversed
    mov rsi, string1
    mov rdi, string3
    mov rcx, [l1]
    cld
    repe cmpsb
    jne notpal

    write msg_palin_yes, len_palin_yes
    jmp printmenu

notpal:
    write msg_palin_no, len_palin_no
    jmp printmenu

; ==========================================
; Exit Program
; ==========================================
exit:
    write msg_exit, len_exit
    mov rax, 60
    xor rdi, rdi
    syscall

; ==========================================
```

```
; Display number in hexadecimal
; ===========================================
display:
    mov rsi, buff
    mov rcx, 16

disp_loop:
    rol rbx, 4
    mov dl, bl
    and dl, 0Fh
    cmp dl, 9
    jbe add30
    add dl, 7

add30:
    add dl, 30h
    mov [rsi], dl
    inc rsi
    loop disp_loop

    write buff, 16
    write newline, 1
    ret
```

10 .Write a menu driven ALP to implement various mentioned string operations. i)String length ii) String Compare iii) String Copy

CODE :

```
; ================================================
; STRING OPERATIONS: Length, Compare, Copy
```

```
; 64-bit Linux Assembly Program
; ==================================================

%macro read 2
   mov rax, 0
   mov rdi, 0
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

%macro write 2
   mov rax, 1
   mov rdi, 1
   mov rsi, %1
   mov rdx, %2
   syscall
%endmacro

section .data
   msg_menu db 10, "===== STRING OPERATIONS MENU =====",10
          db "1. String Length",10
          db "2. String Compare",10
          db "3. String Copy",10
          db "4. Exit",10
          db "Enter your choice (1-4): "
   len_menu equ $ - msg_menu

   msg_input1 db 10,"Enter String1: "
   len_input1 equ $ - msg_input1

   msg_input2 db 10,"Enter String2: "
   len_input2 equ $ - msg_input2

   msg_length db 10,"Length of string: "
   len_length equ $ - msg_length
```

```
    msg_equal db 10,"Strings are EQUAL",10
    len_equal equ $ - msg_equal

    msg_not_equal db 10,"Strings are NOT EQUAL",10
    len_not_equal equ $ - msg_not_equal

    msg_copied db 10,"String copied successfully!",10
    msg_display db "Copied string: "
    len_copied equ $ - msg_copied

    msg_invalid db 10,"Invalid choice! Please enter 1-4.",10
    len_invalid equ $ - msg_invalid

    msg_exit db 10,"Exiting program...",10
    len_exit equ $ - msg_exit

    newline db 10

section .bss
    string1 resb 50
    string2 resb 50
    string3 resb 50
    l1 resq 1
    l2 resq 1
    choice resb 2
    buff resb 16

section .text
global _start

_start:
printmenu:
    write msg_menu, len_menu
    read choice, 2

    cmp byte[choice], '1'
    je strlen
```

```
    cmp byte[choice], '2'
    je strcmp
    cmp byte[choice], '3'
    je strcpy
    cmp byte[choice], '4'
    je exit

    write msg_invalid, len_invalid
    jmp printmenu

; ==========================================
; 1. String Length
; ==========================================
strlen:
    write msg_input1, len_input1
    read string1, 50
    dec rax
    mov [l1], rax

    write msg_length, len_length
    mov rbx, [l1]
    call display

    jmp printmenu

; ==========================================
; 2. String Compare
; ==========================================
strcmp:
    ; Get first string
    write msg_input1, len_input1
    read string1, 50
    dec rax
    mov [l1], rax

    ; Get second string
    write msg_input2, len_input2
```

```
    read string2, 50
    dec rax
    mov [l2], rax

    ; First check if lengths are equal
    mov rbx, [l1]
    cmp rbx, [l2]
    jne not_equal

    ; Compare byte by byte
    mov rsi, string1
    mov rdi, string2
    mov rcx, [l1]
    cld
    repe cmpsb
    jne not_equal

    ; Strings are equal
    write msg_equal, len_equal
    jmp printmenu

not_equal:
    write msg_not_equal, len_not_equal
    jmp printmenu

; ==========================================
; 3. String Copy
; ==========================================
strcpy:
    ; Get source string
    write msg_input1, len_input1
    read string1, 50
    dec rax
    mov [l1], rax

    ; Copy string1 to string3
    mov rsi, string1
```

```
    mov rdi, string3
    mov rcx, [l1]
    cld
    rep movsb

    ; Display success message and copied string
    write msg_copied, len_copied
    write string3, [l1]
    write newline, 1

    jmp printmenu

; ==========================================
; Exit Program
; ==========================================
exit:
    write msg_exit, len_exit
    mov rax, 60
    xor rdi, rdi
    syscall

; ==========================================
; Display number in hexadecimal
; ==========================================
display:
    mov rsi, buff
    mov rcx, 16

disp_loop:
    rol rbx, 4
    mov dl, bl
    and dl, 0Fh
    cmp dl, 9
    jbe add30
    add dl, 7

add30:
```

```
    add dl, 30h
    mov [rsi], dl
    inc rsi
    loop disp_loop

    write buff, 16
    write newline, 1
    ret
```

11. Write a menu driven ALP to implement various mentioned string operations. i)String length ii) String Reverseiii) String palindrome

CODE :

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

%macro write 2
    mov rax, 1
    mov rdi, 1
    mov rsi, %1
    mov rdx, %2
    syscall
%endmacro

section .data
    msg_menu db 10, "===== STRING OPERATIONS MENU =====",10
            db "1. String Length",10
            db "2. String Reverse",10
            db "3. String Palindrome",10
            db "4. Exit",10
            db "Enter your choice (1-4): "
```

```
    len_menu equ $ - msg_menu

    msg_input db 10,"Enter string: "
    len_input equ $ - msg_input

    msg_length db 10,"Length of string: "
    len_length equ $ - msg_length

    msg_reversed db 10,"Reversed string: "
    len_reversed equ $ - msg_reversed

    msg_palin_yes db 10,"String is PALINDROME",10
    len_palin_yes equ $ - msg_palin_yes

    msg_palin_no db 10,"String is NOT PALINDROME",10
    len_palin_no equ $ - msg_palin_no

    msg_invalid db 10,"Invalid choice! Please enter 1-4.",10
    len_invalid equ $ - msg_invalid

    msg_exit db 10,"Exiting program...",10
    len_exit equ $ - msg_exit

    newline db 10

section .bss
    string1 resb 50
    string2 resb 50
    l1 resq 1
    choice resb 2
    buff resb 16

section .text
global _start

_start:
printmenu:
```

```
    write msg_menu, len_menu
    read choice, 2

    cmp byte[choice], '1'
    je strlen
    cmp byte[choice], '2'
    je strreverse
    cmp byte[choice], '3'
    je strpalindrome
    cmp byte[choice], '4'
    je exit

    write msg_invalid, len_invalid
    jmp printmenu

; ==========================================
; 1. String Length
; ==========================================
strlen:
    write msg_input, len_input
    read string1, 50
    dec rax
    mov [l1], rax

    write msg_length, len_length
    mov rbx, [l1]
    call display

    jmp printmenu

; ==========================================
; 2. String Reverse
; ==========================================
strreverse:
    ; Get input string
    write msg_input, len_input
    read string1, 50
```

```
    dec rax
    mov [l1], rax

    ; Reverse the string
    mov rsi, string1
    add rsi, [l1]
    dec rsi              ; RSI points to last character
    mov rdi, string2     ; RDI points to destination
    mov rcx, [l1]        ; Counter = length

reverse_loop:
    mov dl, [rsi]        ; Get character from end
    mov [rdi], dl        ; Put it at beginning of string2
    dec rsi              ; Move backward in string1
    inc rdi              ; Move forward in string2
    loop reverse_loop    ; Repeat

    ; Display reversed string
    write msg_reversed, len_reversed
    write string2, [l1]
    write newline, 1

    jmp printmenu

; ==========================================
; 3. String Palindrome
; ==========================================
strpalindrome:
    ; Get input string
    write msg_input, len_input
    read string1, 50
    dec rax
    mov [l1], rax

    ; Reverse string1 into string2
    mov rsi, string1
    add rsi, [l1]
```

```
    dec rsi
    mov rdi, string2
    mov rcx, [l1]

palin_reverse:
    mov dl, [rsi]
    mov [rdi], dl
    dec rsi
    inc rdi
    loop palin_reverse

    ; Compare original and reversed
    mov rsi, string1
    mov rdi, string2
    mov rcx, [l1]
    cld
    repe cmpsb
    jne not_palindrome

    ; It's a palindrome
    write msg_palin_yes, len_palin_yes
    jmp printmenu

not_palindrome:
    write msg_palin_no, len_palin_no
    jmp printmenu

; ==========================================
; Exit Program
; ==========================================
exit:
    write msg_exit, len_exit
    mov rax, 60
    xor rdi, rdi
    syscall

; ==========================================
```

```
; Display number in hexadecimal
; ==========================================
display:
    mov rsi, buff
    mov rcx, 16

disp_loop:
    rol rbx, 4
    mov dl, bl
    and dl, 0Fh
    cmp dl, 9
    jbe add30
    add dl, 7

add30:
    add dl, 30h
    mov [rsi], dl
    inc rsi
    loop disp_loop

    write buff, 16
    write newline, 1
    ret
```

12 .Write an ALP to find the roots of the quadratic equation for all the possible cases.

CODE :

```
%macro READ 2
```

```
    mov rax, 0          ; sys_read
    mov rdi, 0          ; stdin
    mov rsi, %1         ; buffer
    mov rdx, %2         ; length
    syscall
%endmacro

%macro WRITE 2
    mov rax, 1          ; sys_write
    mov rdi, 1          ; stdout
    mov rsi, %1         ; buffer
    mov rdx, %2         ; length
    syscall
%endmacro

section .data
    title db "===== QUADRATIC EQUATION SOLVER
=====", 10
          db "Format: ax^2 + bx + c = 0", 10, 10
    title_len equ $-title

    msg_a db "Enter coefficient a: "
    len_a equ $-msg_a

    msg_b db "Enter coefficient b: "
    len_b equ $-msg_b

    msg_c db "Enter coefficient c: "
```

```
   len_c equ $-msg_c

   msg_discriminant db 10, "Discriminant (b^2 - 4ac) = "
   len_discriminant equ $-msg_discriminant

   msg_real_equal db 10, "CASE: Real and Equal Roots", 10
                  db "Root1 = Root2 = "
   len_real_equal equ $-msg_real_equal

   msg_real_distinct db 10, "CASE: Real and Distinct Roots", 10
   len_real_distinct equ $-msg_real_distinct

   msg_root1 db "Root1 = "
   len_root1 equ $-msg_root1

   msg_root2 db "Root2 = "
   len_root2 equ $-msg_root2

   msg_imaginary db 10, "CASE: Imaginary Roots", 10
                 db "Root1 = "
   len_imaginary equ $-msg_imaginary

   msg_plus db " + "
   len_plus equ $-msg_plus

   msg_minus db " - "
```

```
    len_minus equ $-msg_minus

    msg_i db "i", 10
    len_i equ $-msg_i

    msg_root2_imag db "Root2 = "
    len_root2_imag equ $-msg_root2_imag

    msg_not_quadratic db 10, "ERROR: 'a' cannot be zero!
Not a quadratic equation.", 10
    len_not_quadratic equ $-msg_not_quadratic

    newline db 10
    minus_sign db "-"

section .bss
    buffer resb 20      ; Input buffer
    a resq 1            ; Coefficient a
    b resq 1            ; Coefficient b
    c resq 1            ; Coefficient c
    discriminant resq 1 ; b^2 - 4ac
    temp resq 1         ; Temporary storage
    root resq 1         ; Root value
    digit_buff resb 20  ; For printing numbers

section .text
    global _start
```

```
_start:
    ; Display title
    WRITE title, title_len

    ; Input coefficient a
    WRITE msg_a, len_a
    READ buffer, 20
    call string_to_int
    mov [a], rax

    ; Check if a = 0
    cmp rax, 0
    je not_quadratic

    ; Input coefficient b
    WRITE msg_b, len_b
    READ buffer, 20
    call string_to_int
    mov [b], rax

    ; Input coefficient c
    WRITE msg_c, len_c
    READ buffer, 20
    call string_to_int
    mov [c], rax

    ; Calculate discriminant: D = b^2 - 4ac
    mov rax, [b]
```

```
    imul rax, rax          ; rax = b^2
    mov [temp], rax

    mov rax, 4
    imul rax, [a]
    imul rax, [c]          ; rax = 4ac

    mov rbx, [temp]
    sub rbx, rax           ; rbx = b^2 - 4ac
    mov [discriminant], rbx

    ; Display discriminant
    WRITE msg_discriminant, len_discriminant
    mov rax, [discriminant]
    call print_signed_number
    WRITE newline, 1

    ; Check discriminant cases
    mov rax, [discriminant]
    cmp rax, 0
    je equal_roots         ; D = 0
    jl imaginary_roots     ; D < 0
    jmp distinct_roots     ; D > 0

not_quadratic:
    WRITE msg_not_quadratic, len_not_quadratic
    jmp exit_program
```

```
; Case 1: D = 0, Real and Equal Roots
; Root = -b / 2a
equal_roots:
    WRITE msg_real_equal, len_real_equal

    ; Calculate -b
    mov rax, [b]
    neg rax

    ; Divide by 2a
    mov rbx, [a]
    imul rbx, 2
    cqo                     ; Sign extend rax to rdx:rax
    idiv rbx

    call print_signed_number
    WRITE newline, 1
    jmp exit_program

; Case 2: D > 0, Real and Distinct Roots
; Root1 = (-b + sqrt(D)) / 2a
; Root2 = (-b - sqrt(D)) / 2a
distinct_roots:
    WRITE msg_real_distinct, len_real_distinct

    ; Calculate sqrt(D) using approximation
    mov rax, [discriminant]
    call integer_sqrt
```

```
mov [temp], rax        ; Store sqrt(D)

; Calculate Root1 = (-b + sqrt(D)) / 2a
mov rax, [b]
neg rax                ; -b
add rax, [temp]        ; -b + sqrt(D)

mov rbx, [a]
imul rbx, 2            ; 2a
cqo
idiv rbx

WRITE msg_root1, len_root1
call print_signed_number
WRITE newline, 1

; Calculate Root2 = (-b - sqrt(D)) / 2a
mov rax, [b]
neg rax                ; -b
sub rax, [temp]        ; -b - sqrt(D)

mov rbx, [a]
imul rbx, 2            ; 2a
cqo
idiv rbx

WRITE msg_root2, len_root2
call print_signed_number
```

```
    WRITE newline, 1
    jmp exit_program

; Case 3: D < 0, Imaginary Roots
; Root1 = -b/2a + (sqrt(-D)/2a)i
; Root2 = -b/2a - (sqrt(-D)/2a)i
imaginary_roots:
    WRITE msg_imaginary, len_imaginary

    ; Calculate real part: -b/2a
    mov rax, [b]
    neg rax
    mov rbx, [a]
    imul rbx, 2
    cqo
    idiv rbx
    mov [temp], rax        ; Store real part

    ; Calculate imaginary part: sqrt(-D)/2a
    mov rax, [discriminant]
    neg rax                ; Make positive
    call integer_sqrt

    mov rbx, [a]
    imul rbx, 2
    cqo
    idiv rbx
    mov [root], rax        ; Store imaginary part
```

```
    ; Print Root1 = real + imag*i
    mov rax, [temp]
    call print_signed_number
    WRITE msg_plus, len_plus
    mov rax, [root]
    call print_number
    WRITE msg_i, len_i

    ; Print Root2 = real - imag*i
    WRITE msg_root2_imag, len_root2_imag
    mov rax, [temp]
    call print_signed_number
    WRITE msg_minus, len_minus
    mov rax, [root]
    call print_number
    WRITE msg_i, len_i

    jmp exit_program

exit_program:
    WRITE newline, 1
    mov rax, 60
    mov rdi, 0
    syscall

; Function: Calculate integer square root (Newton's method)
```

```
; Input: rax = number
; Output: rax = sqrt(number)
integer_sqrt:
    push rbx
    push rcx
    push rdx

    cmp rax, 0
    je .done

    mov rbx, rax           ; x = number
    mov rax, 1             ; Initial guess

.newton_loop:
    mov rcx, rax           ; Save current guess

    ; new_guess = (guess + number/guess) / 2
    mov rax, rbx
    xor rdx, rdx
    div rcx                ; rax = number / guess
    add rax, rcx           ; rax = guess + number/guess
    shr rax, 1             ; rax = (guess + number/guess) / 2

    ; Check convergence
    sub rcx, rax
    cmp rcx, 1
    jg .newton_loop
    cmp rcx, -1
```

```
    jl .newton_loop

.done:
    pop rdx
    pop rcx
    pop rbx
    ret

; Convert string to signed integer
string_to_int:
    push rbx
    push rcx
    push rsi

    xor rax, rax
    xor rcx, rcx
    mov rsi, buffer
    xor rbx, rbx            ; Sign flag (0 = positive, 1 = negative)

    ; Check for negative sign
    movzx rdx, byte [rsi]
    cmp rdx, '-'
    jne .loop
    mov rbx, 1
    inc rsi

.loop:
    movzx rdx, byte [rsi + rcx]
```

```
    cmp rdx, 10
    je .done
    cmp rdx, 0
    je .done

    sub rdx, '0'
    imul rax, 10
    add rax, rdx
    inc rcx
    jmp .loop

.done:
    cmp rbx, 1
    jne .positive
    neg rax

.positive:
    pop rsi
    pop rcx
    pop rbx
    ret

; Print signed number
print_signed_number:
    push rax

    cmp rax, 0
    jge .positive
```

```
    ; Negative number
    WRITE minus_sign, 1
    neg rax

.positive:
    call print_number
    pop rax
    ret

; Print unsigned number
print_number:
    push rax
    push rbx
    push rcx
    push rdx

    mov rcx, 0
    mov rbx, 10

    cmp rax, 0
    jne .divide_loop

    mov byte [digit_buff], '0'
    WRITE digit_buff, 1
    jmp .done

.divide_loop:
```

```
    xor rdx, rdx
    div rbx
    add dl, '0'
    push rdx
    inc rcx
    cmp rax, 0
    jne .divide_loop

.print_loop:
    pop rax
    mov byte [digit_buff], al
    push rcx
    WRITE digit_buff, 1
    pop rcx
    loop .print_loop

.done:
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret
```