Muhammad Ammar Janjuha

BCS213104

Assignment 4

Coal

Question No.1

Part(a):

The operations on num1 and num2, you would follow these steps:

Extended Left Shifting: This operation shifts the bits of the operand to the left. The leftmost bit is lost and a zero replaces the rightmost bit. To shift num1 by 2 bits to the left, you would use the shl instruction:

Code:

```
mov eax, num1
shl eax, 2
```

This operation effectively multiplies num1 by 2^2 (or 4). The result is stored in the eax register 1.

Extended Right Shifting: This operation shifts the bits of the operand to the right. The rightmost bit is lost and the leftmost bit is filled with the sign bit (for signed numbers). To shift num1 by 2 bits to the right, you would use the shr instruction:

Code:

```
mov eax, num1
shr eax, 2
```

This operation effectively divides num1 by 2^2 (or 4). The result is stored in the eax register

Extended Addition: This operation adds the operands together. To add num1 and num2, you would use the add instruction:

Code:

```
mov eax, num1
add eax, num2
```

The sum of num1 and num2 is stored in the eax register 2.

Extended Subtraction: This operation subtracts the second operand from the first. To subtract num2 from num1, you would use the sub instruction:

Code:

```
mov eax, num1
sub eax, num2
```

The result of num1 - num2 is stored in the eax register 2.

Part(b):

In the context of assembly language, the stack serves several key purposes:

Storing Local Variables: Each time a function is called, its local variables need a place to live. This is where the stack comes in. When a function is called, it pushes its parameters onto the stack. Inside the function, any local variables are also allocated on the stack. When the function returns, it cleans up after itself by popping off the local variables and parameters it pushed onto the stack.

Maintaining Control Flow Information: The stack is used to keep track of where control should return to after a function call. This is done using a mechanism known as "call stack". Each time a function is called, its return address is pushed onto the stack. After the function finishes executing, it pops this address off the stack and jumps back to that location, resuming execution where it left off .

Implementing Function Calls: The stack is crucial for implementing recursive functions. In recursion, a function calls itself, which means it needs a way to remember where it was so it can continue where it left off when the recursive call returns. This is achieved by pushing the current state (including the return address) onto the stack before making the recursive call, and then popping it off when returning from the call.

Error Handling: The stack plays a role in error handling. When an exception occurs, the system can use the stack to unwind the call stack and clean up resources, such as deallocating memory and closing files, that were allocated during the function calls leading up to the exception.

Part(c):

The Stack Pointer (SP) plays a crucial role in managing the stack in 8088 assembly language. It serves as a pointer to the top of the stack, indicating where the next value will be placed or retrieved from the stack. Here's how it works:

Pointing to the Top of the Stack: The SP always points to the last item that has been added to the stack. This is because the stack grows downwards in memory, meaning new items are added at lower memory addresses. Thus, the SP is updated every time an item is pushed onto or popped off the stack.

Managing Memory Allocation: By keeping track of the top of the stack, the SP allows efficient management of memory allocation. When a function wants to allocate space for local variables, it knows exactly where to start based on the current position of the SP.

Controlling Function Calls: The SP is also used to manage function calls. When a function is called, its return address is pushed onto the stack, updating the SP. When the function returns, it pops the return address off the stack, again updating the SP.

Error Handling: In case of errors or exceptions, the SP can be used to clean up the stack. By knowing the current position of the SP, the system can easily remove all values above it, effectively unwinding the stack.

Part(d):

Subroutines in 8088 assembly language are blocks of instructions that perform a specific task. They are used to encapsulate repeated or complex tasks within a program, making the code more modular, easier to understand, and more efficient.

Here's how subroutines work:

Definition: A subroutine is defined once in the program, and can be called from multiple places. This avoids duplicating code, which saves memory and makes maintenance easier.

Calling a Subroutine: To call a subroutine, the CALL instruction is used. This instruction saves the return address (the location in the program to return to after the subroutine finishes) and jumps to the start of the subroutine. The subroutine is said to be the "callee", and the part of the program that initiated the call is the "caller".

Returning from a Subroutine: Once a subroutine has finished executing, it returns to the caller by using the RET instruction. This instruction pops the return address from the stack and jumps back to that location in the program.

Subroutines play a significant role in organizing 8088 assembly programs:

Modularity: By breaking down a program into smaller, self-contained subroutines, each performing a specific task, the code becomes more modular and easier to understand.

Reusability: Subroutines can be called multiple times within a program, promoting code reuse.

Abstraction: Subroutines hide the details of their implementation from the rest of the program. This allows programmers to focus on the overall structure of the program, rather than the specifics of individual tasks.

Control Flow: Subroutines allow for complex control flow structures, such as nested subroutines and recursive calls, which would be difficult to implement without them.

Part(e):

In 8088 assembly language, parameters can be passed to a subroutine through registers, main memory, or the stack. The method used depends on the complexity of the data structures involved and the specific requirements of the program.

Passing Parameters Through Registers: Simple subroutines often pass parameters through registers. This is especially true if there are plenty of registers available. For example, the ARM architecture uses this method extensively 1.

Passing Parameters Through Main Memory: Some subroutines, particularly those working on large, single-occurrence data structures, can sometimes pass parameters directly through main memory. However, this method is less commonly used due to potential issues with data consistency and synchronization 1.

Passing Parameters Through the Stack: The most general method for passing parameters is through the stack. This method supports the largest number of cases, including large and small data structures, nested subroutine calls, and recursion. The caller pushes the parameters onto the stack, then calls the subroutine. The subroutine can then access the parameters from the stack. When the subroutine returns, it pops the parameters off the stack.

An example of passing parameters through the stack in 8088 assembly:

```
caller:
push param1 ; Push parameter onto the stack
push param2 ; Push another parameter onto the stack
call subroutine ; Call the subroutine

subroutine:
push bp ; Save old base pointer
mov bp, sp ; Use the current stack pointer as new base pointer
mov ax, [bp+4] ; Access the first parameter
mov bx, [bp+6] ; Access the second parameter
; ... do your stuff, use the stack all you want,
; just make sure that by when we get here push/pop have balanced out
pop bp ; Restore old base pointer
ret ; Return, popping the extra 4 bytes of the arguments in the process
```

Part(f):

To invert bits 4 and 8 in the binary number 0110110110 using a logical operator and a mask in 8086 assembly language, you can use the xor instruction. The xor instruction performs a bitwise exclusive OR operation, which effectively inverts the bits of the operands.

First, define the binary number and the mask. The binary number is 0110110110, and the mask should have 1s at the 4th and 8th positions (from right to left, starting from index 0). So, the mask is 0001000100.

Then, perform the xor operation between the binary number and the mask. This will invert the 4th and 8th bits.

Here's how you can do it in 8086 assembly language:

Code:

```
mov ax, 0b0110110110 ; Load the binary number into the AX register
mov cx, 0b0001000100 ; Load the mask into the CX register
xor ax, cx           ; Perform the XOR operation
```

After these instructions, the ax register will contain the binary number with the 4th and 8th bits inverted.

Question No.2

Part(a)

Code:

```
section .data
  prompt db 'Enter a character: ', 0
  buffer db 10 dup(0) ; Buffer to store the input characters

section .bss
  stack resb 10 ; Reserve space for the stack

section .text
  global _start
```

```
_start:
    ; Read 10 characters from the user
    mov ecx, 10
read_loop:
    mov edx, prompt
    mov eax, 4
    syscall ; Print the prompt

    mov edx, 1
    mov eax, 3
    syscall ; Read a character

    dec ecx
    jz print_loop ; If we've read 10 characters, jump to print_loop

    ; Push the character onto the stack
    mov [stack + ecx], al
    jmp read_loop

print_loop:
    ; Pop a character from the stack
    mov al, [stack + ecx]

    ; Print the character
    mov edx, 1
```

```
   mov eax, 1
   syscall

   ; Move to the next character
   inc ecx
   jmp print_loop

   ; Exit the program
   mov eax, 60
   xor edi, edi
   syscall
```

Part(b)

Code:

```
section .data
   arr dw 3, 2, 5, 7, 8, 6, 4, 1, 9, 0 ; Array of integers
   len equ $-arr ; Length of the array

section .text
   global _start

_start:
   ; Find the maximum value in the array
   call max_value
```

```
    ; Print the maximum value
    mov [numStr], ax
    mov ecx, numStr
    mov edx, 2
    mov ebx, 1
    mov eax, 4
    int 0x80

    ; Exit the program
    mov eax, 1
    xor ebx, ebx
    int 0x80

max_value:
    ; Initialize the max value with the first element of the array
    mov ax, [arr]
    mov bx, 1

next_element:
    ; Get the next element from the array
    mov cx, [arr + bx * 2]

    ; Compare the current max value with the current element
    cmp ax, cx
```

```
    jge increment_index

    ; If the current element is greater than the max value, update the max value
    mov ax, cx

increment_index:
    ; Move to the next element in the array
    inc bx
    cmp bx, len
    jl next_element

    ret

section .data
    numStr db 'Max value: %d', 0xA, 0 ; String to print the maximum value
```