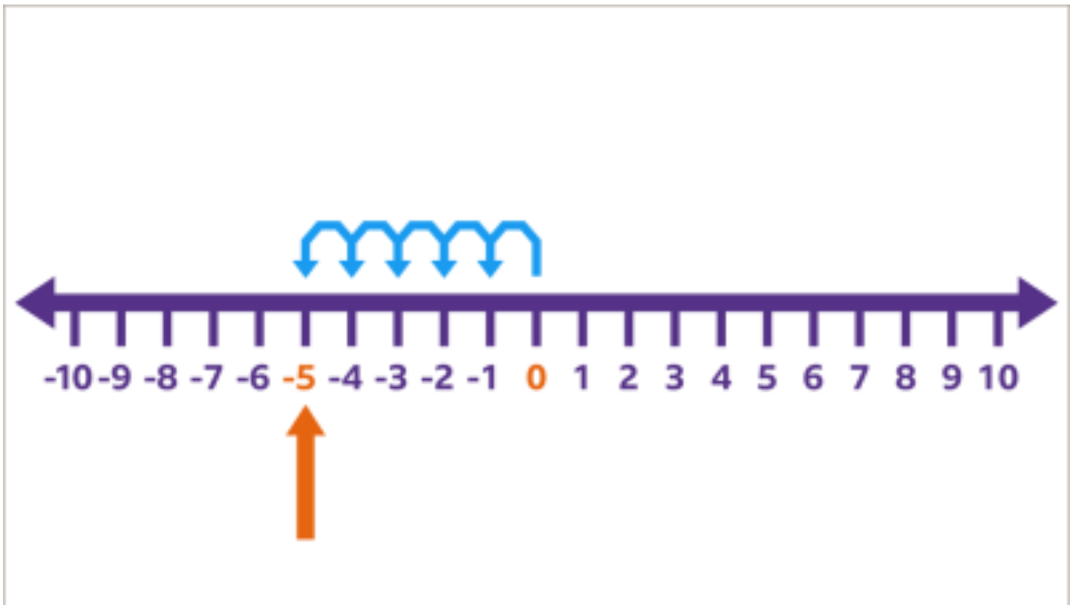
Questions for this topic

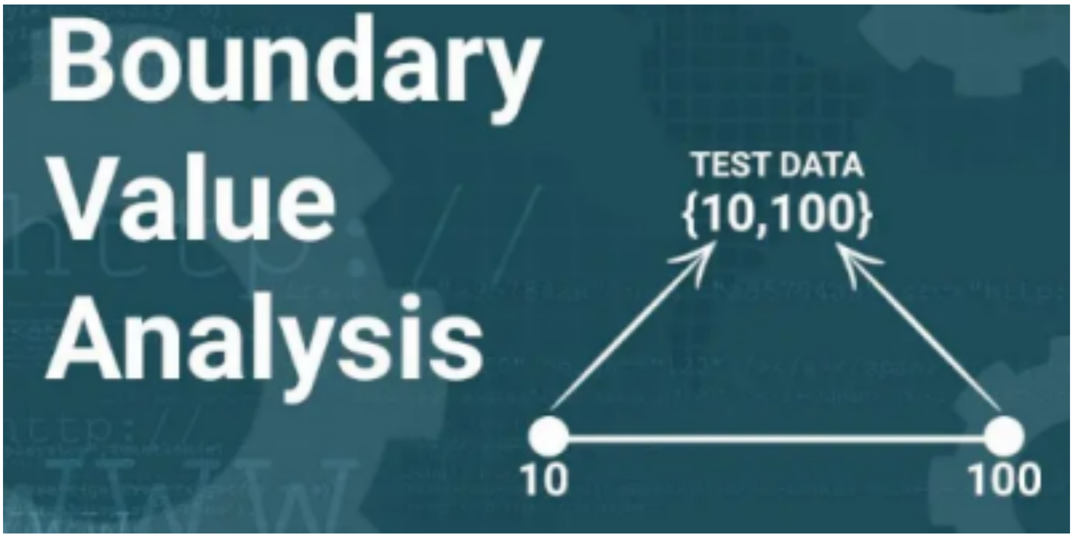
I can explain the suggestions provided for testing your code:

Using a Debugger: Debugging your code using a debugger is a crucial step in ensuring its correctness. Debuggers allow you to step through your code line by line, inspect variables, and identify issues. You can use debugging tools provided by your development environment (e.g., Visual Studio Debugger).

Testing with Negative Values: If your code deals with signed data, it's essential to include negative values in your test cases to cover all possible scenarios.

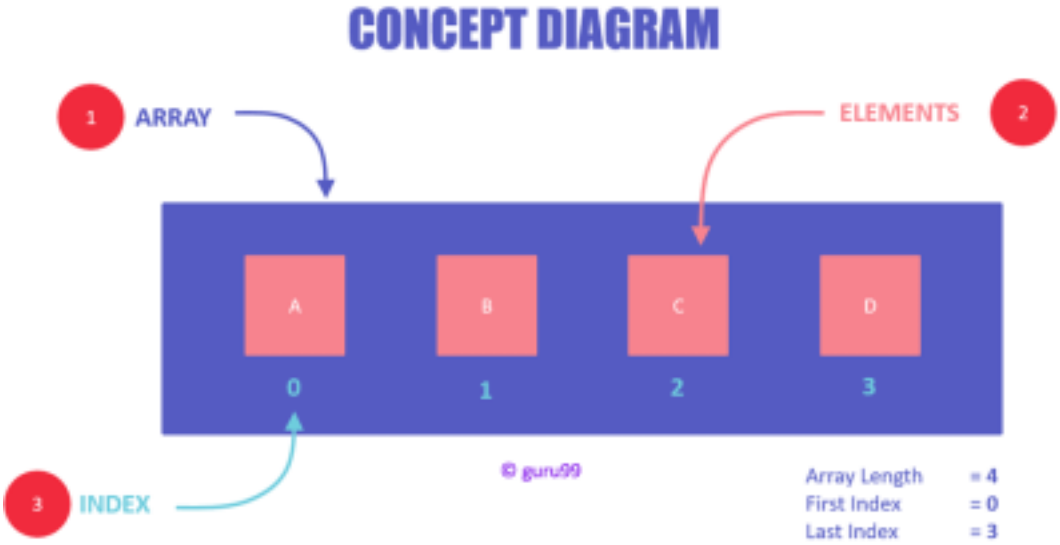


Testing at Boundaries: When a range of input values is specified, test your code with values that fall before, on, and after these boundaries. This helps verify how your code handles edge cases.



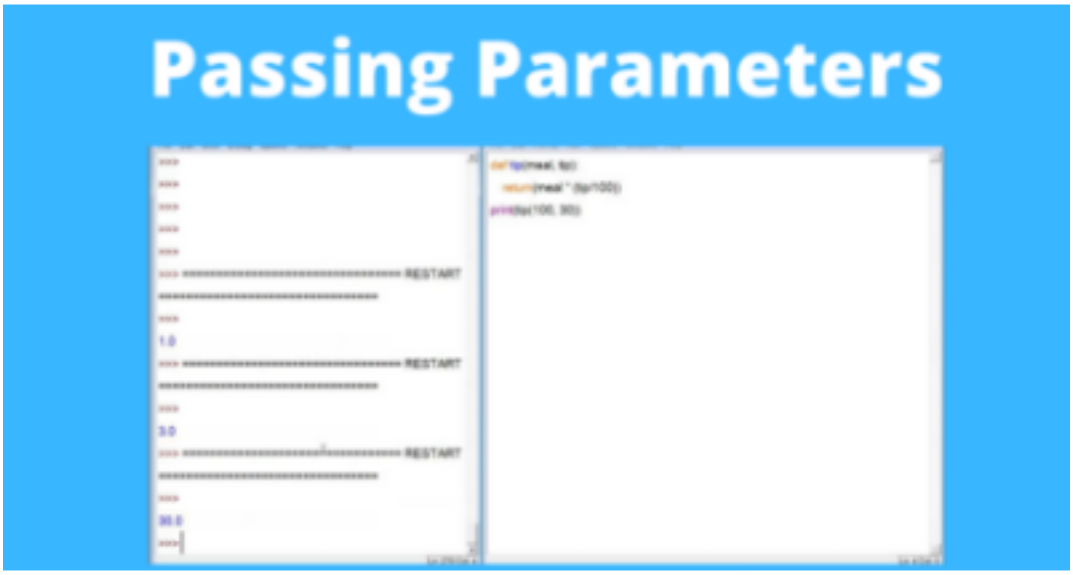
Multiple Test Cases: Create multiple test cases with different inputs and conditions. This ensures that your code is robust and can handle a variety of scenarios.

Using a Debugger for Array Operations: When working with arrays, especially when modifying them, a debugger's Memory window can be very useful. It allows you to inspect the array's contents in hexadecimal or decimal representation.



Checking Register Preservation: If you have a procedure that modifies registers, consider calling it twice in a row. This helps verify that the procedure correctly preserves register values between calls.

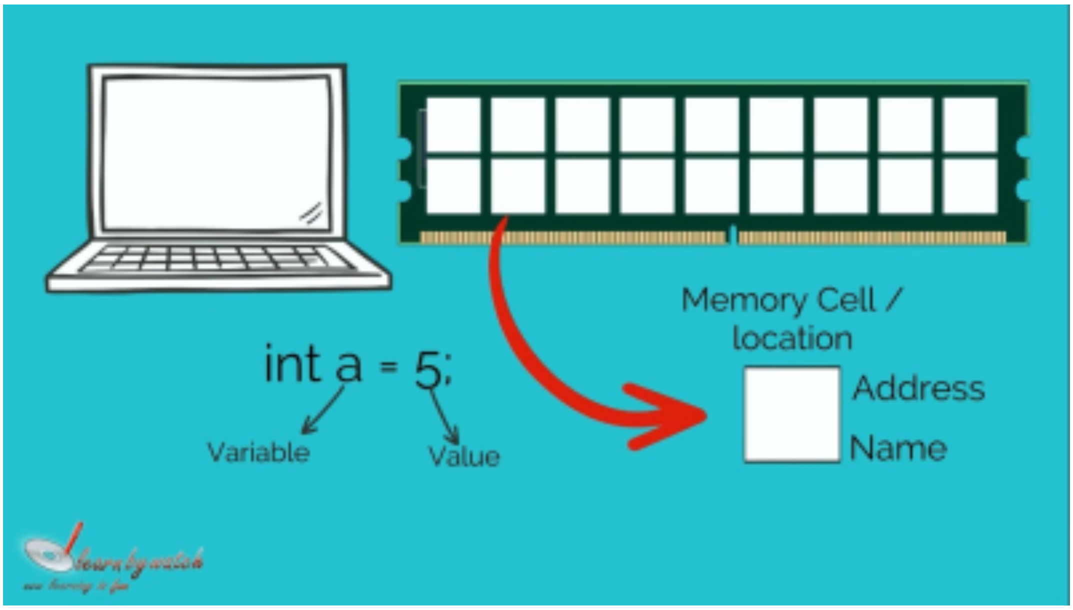
Parameter Passing for Multiple Arrays: When passing multiple arrays to a procedure, it's a good practice not to refer to arrays by name inside the procedure. Instead, set registers like ESI or EDI to the offsets of the arrays before calling the procedure. Use indirect addressing ([esi] or [edi]) inside the procedure to access array elements.



```
def tip(meal, tip):
  return(meal * (tip/100))
print(tip(100, 30))
```

```
>>>
>>>
>>>
>>>
>>>
>>> ================================ RESTART
================================
>>>
1.0
>>> ================================ RESTART
================================
>>>
3.0
>>> ================================ RESTART
================================
>>>
30.0
>>>
```

Local Variables in Procedures: If you need to create variables for use only within a procedure, you can declare them using the .data directive before the variable and the .code directive afterward. Initialize these variables within the procedure to ensure they start with the correct values when the procedure is called multiple times.



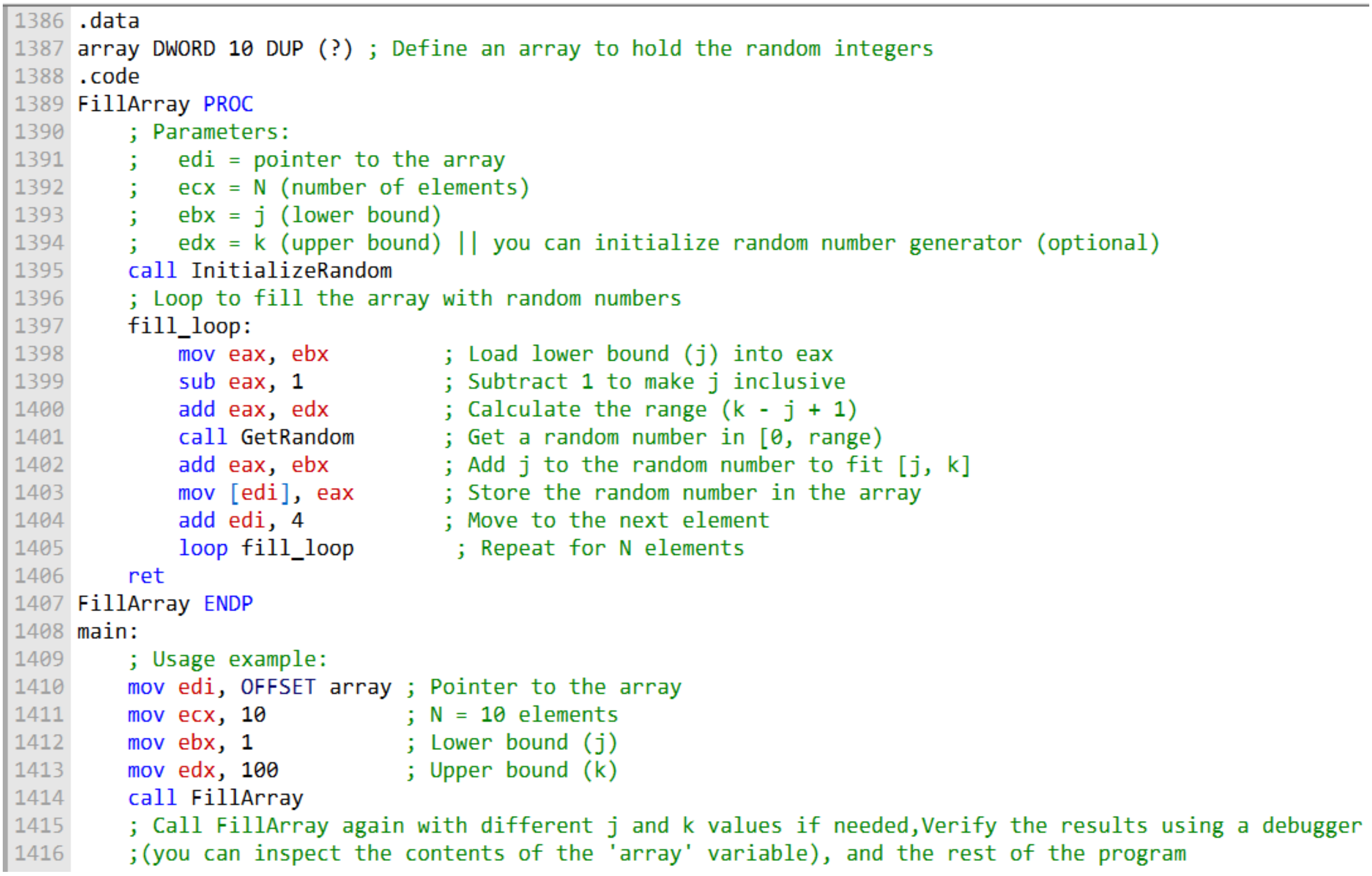
================================

Exercise 1: Filling an Array

================================

This exercise requires you to create a procedure that fills an array of doublewords with N random integers within the range [j, k]. You need to pass a pointer to the array, the value of N, and the values of j and k as parameters to the procedure. Additionally, you should preserve all register values between calls to the procedure.

Here's a sample assembly code for this exercise:



```
.data
array DWORD 10 DUP (?) ; Define an array to hold the random integers
.code
FillArray PROC
    ; Parameters:
    ;   edi = pointer to the array
    ;   ecx = N (number of elements)
    ;   ebx = j (lower bound)
    ;   edx = k (upper bound) || you can initialize random number generator (optional)
    call InitializeRandom
    ; Loop to fill the array with random numbers
    fill_loop:
        mov eax, ebx          ; Load lower bound (j) into eax
        sub eax, 1            ; Subtract 1 to make j inclusive
        add eax, edx          ; Calculate the range (k - j + 1)
        call GetRandom        ; Get a random number in [0, range)
        add eax, ebx          ; Add j to the random number to fit [j, k]
        mov [edi], eax        ; Store the random number in the array
        add edi, 4            ; Move to the next element
        loop fill_loop         ; Repeat for N elements
    ret
FillArray ENDP
main:
    ; Usage example:
    mov edi, OFFSET array ; Pointer to the array
    mov ecx, 10           ; N = 10 elements
    mov ebx, 1            ; Lower bound (j)
    mov edx, 100          ; Upper bound (k)
    call FillArray
    ; Call FillArray again with different j and k values if needed,Verify the results using a debugger
    ;(you can inspect the contents of the 'array' variable), and the rest of the program
```

This code defines a procedure called FillArray, which fills an array with random integers within the specified range. The main program demonstrates how to use this procedure with different values of j and k.

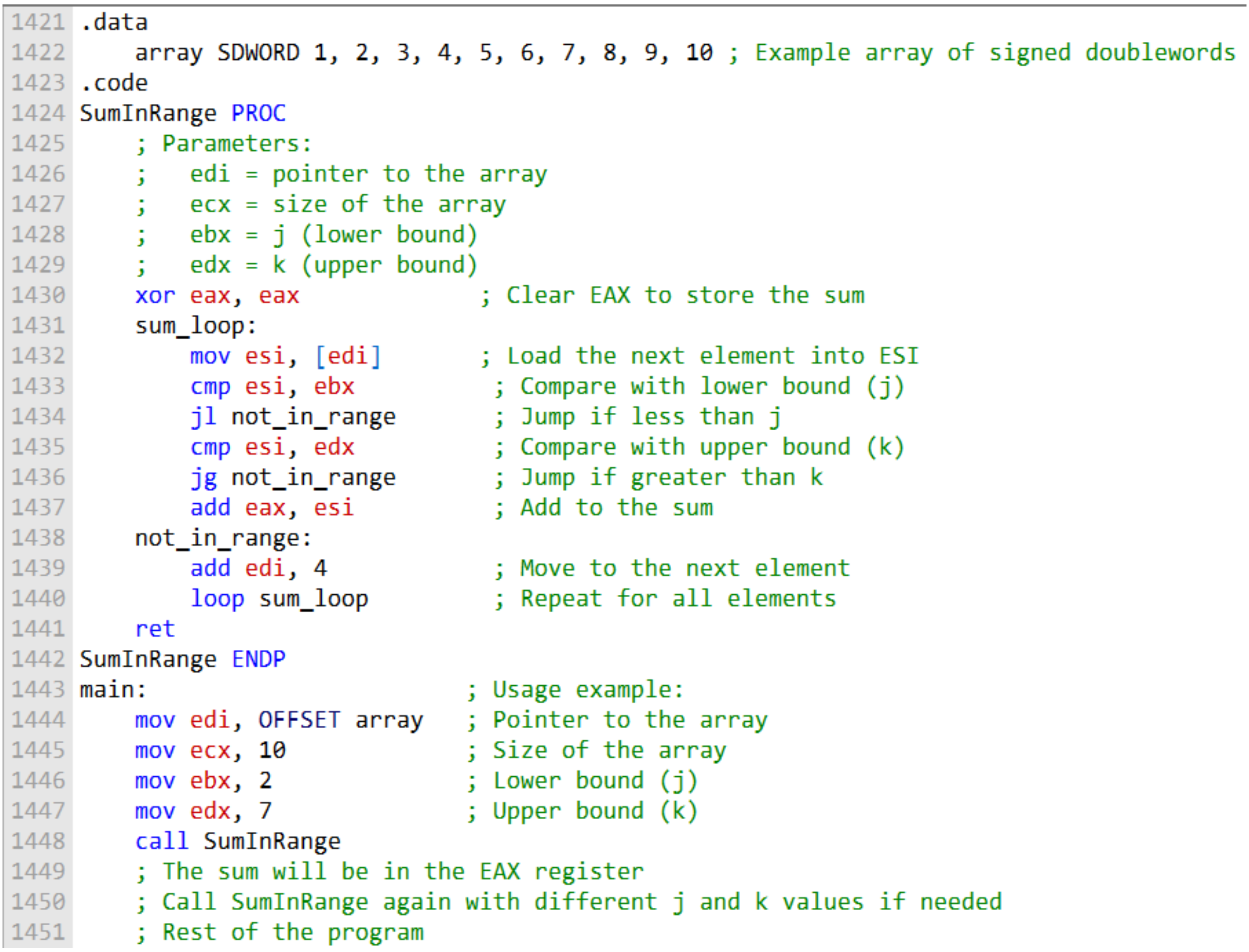
==============================

Exercise 1: Summing an Array

==============================

This exercise requires you to create a procedure that returns the sum of all array elements within the range [j, k]. You'll pass a pointer to the array, the size of the array, and the values of j and k as parameters to the procedure. The sum should be returned in the EAX register, and all other register values should be preserved between calls.

Here's a sample assembly code for this exercise:



```
.data
    array SDWORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ; Example array of signed doublewords
.code
SumInRange PROC
    ; Parameters:
    ;   edi = pointer to the array
    ;   ecx = size of the array
    ;   ebx = j (lower bound)
    ;   edx = k (upper bound)
    xor eax, eax              ; Clear EAX to store the sum
    sum_loop:
        mov esi, [edi]        ; Load the next element into ESI
        cmp esi, ebx           ; Compare with lower bound (j)
        jl not_in_range        ; Jump if less than j
        cmp esi, edx           ; Compare with upper bound (k)
        jg not_in_range        ; Jump if greater than k
        add eax, esi           ; Add to the sum
    not_in_range:
        add edi, 4             ; Move to the next element
        loop sum_loop          ; Repeat for all elements
    ret
SumInRange ENDP
main:                        ; Usage example:
    mov edi, OFFSET array    ; Pointer to the array
    mov ecx, 10              ; Size of the array
    mov ebx, 2               ; Lower bound (j)
    mov edx, 7               ; Upper bound (k)
    call SumInRange
    ; The sum will be in the EAX register
    ; Call SumInRange again with different j and k values if needed
    ; Rest of the program
```

This code defines a procedure called SumInRange, which calculates the sum of array elements within the specified range [j, k]. The main program demonstrates how to use this procedure with different values of j and k.

===================================

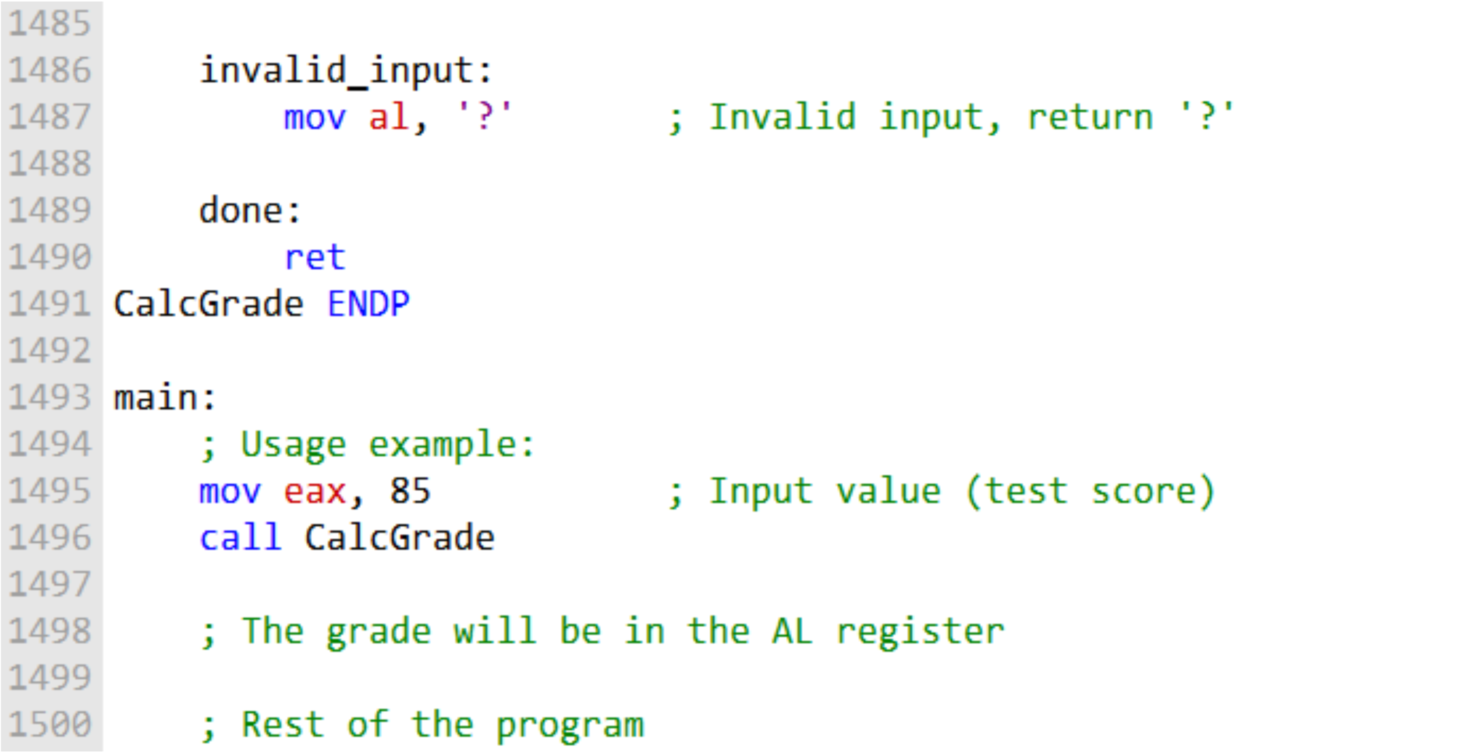
Exercise 1: TestScore Evaluation

===================================

This exercise requires you to create a procedure named CalcGrade that receives an integer value between 0 and 100 and returns a single capital letter grade in the AL register. The grade returned should be based on specified ranges.

Here's a sample assembly code for this exercise:

```
.data
    grade CHAR ? ; Variable to store the grade
.code
CalcGrade PROC
    ; Parameter:
    ;   eax = integer value between 0 and 100
    cmp eax, 0
    jl invalid_input     ; Input is less than 0, return 'F'
    cmp eax, 60
    jl grade_F           ; Input is less than 60, return 'F'
    cmp eax, 70
    jl grade_D           ; Input is less than 70, return 'D'
    cmp eax, 80
    jl grade_C           ; Input is less than 80, return 'C'
    cmp eax, 90
    jl grade_B           ; Input is less than 90, return 'B'
    grade_A:
        mov al, 'A'       ; Input is 90 or greater, return 'A'
        jmp done
    grade_B:
        mov al, 'B'       ; Input is between 80 and 89, return 'B'
        jmp done
    grade_C:
        mov al, 'C'       ; Input is between 70 and 79, return 'C'
        jmp done
    grade_D:
        mov al, 'D'       ; Input is between 60 and 69, return 'D'
        jmp done
    grade_F:
        mov al, 'F'       ; Input is between 0 and 59, return 'F'
```



```
    invalid_input:
        mov al, '?'       ; Invalid input, return '?'

    done:
        ret
CalcGrade ENDP

main:
    ; Usage example:
    mov eax, 85           ; Input value (test score)
    call CalcGrade

    ; The grade will be in the AL register

    ; Rest of the program
```

This code defines a procedure called CalcGrade, which returns a grade based on the specified ranges. The main program demonstrates how to use this procedure by passing a test score (integer value) and receiving the corresponding grade in the AL register.

==================================

Now it's time for you to do your own practice:

Exercise 4: Test Score Evaluation

Create a program that generates 10 random integers between 50 and 100 (inclusive). For each integer generated, pass it to the CalcGrade procedure, which will return a corresponding letter grade based on specified ranges. Display the integer and its corresponding letter grade. You can use the RandomRange procedure from the Irvine32 library to generate random integers.

Exercise 5: Boolean Calculator (1)

Create a program that acts as a simple boolean calculator for 32-bit integers. It displays a menu with options to perform logical operations (AND, OR, NOT, XOR) and allows the user to choose an operation. Implement this menu using Table-Driven Selection. When the user selects an operation, call a procedure to display the operation name. Implement this menu-driven program.

Exercise 6: Boolean Calculator (2)

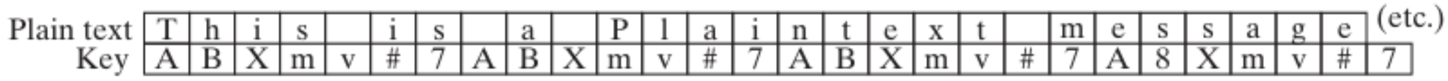
Continuing from Exercise 5, implement procedures for each of the logical operations (AND, OR, NOT, XOR). Prompt the user for inputs (hexadecimal integers) as required by the chosen operation, perform the operation, and display the result in hexadecimal.

Exercise 7: Probabilities and Colors

Write a program that randomly selects one of three colors (white, blue, green) with specific probabilities (30%, 10%, 60%). Use a loop to display 20 lines of text, each with a randomly chosen color based on the given probabilities. You can generate a random integer between 0 and 9 and use it to select colors accordingly.

Exercise 8: Message Encryption

Revise an encryption program to encrypt and decrypt a message using an encryption key consisting of multiple characters. Implement encryption and decryption by XOR-ing each character of the key against a corresponding byte in the message. Repeat the key as necessary until all plaintext bytes are translated.

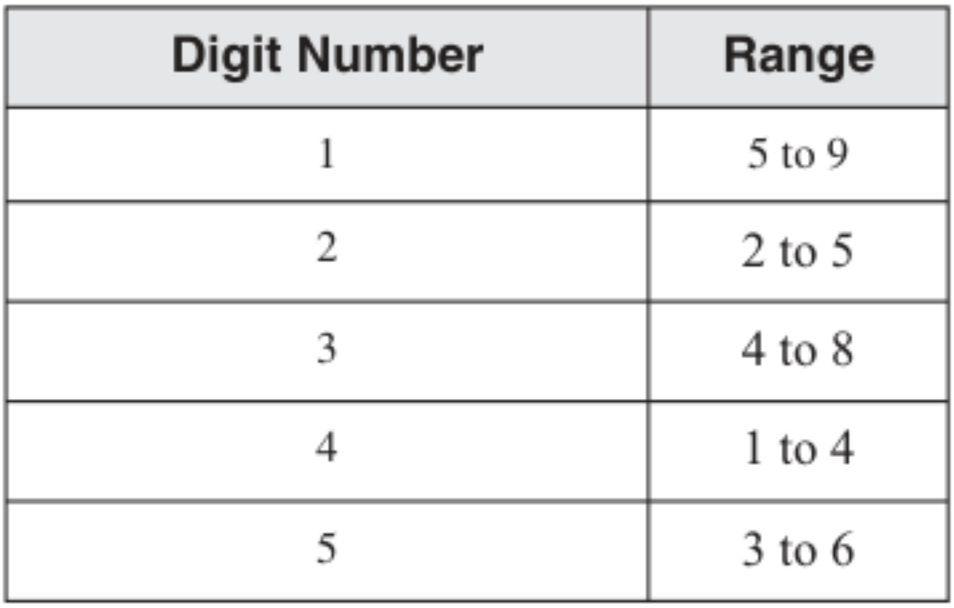


Plain text	T	h	i	s		i	s		a		P	l	a	i	n	t	e	x	t		m	e	s	s	a	g	e	(etc.)
Key	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	8	X	m	v	#	7

(The key repeats until it equals the length of the plain text...)

Exercise 9: Validating a PIN

Create a procedure called Validate_PIN that checks the validity of a 5-digit PIN based on specified digit ranges. The procedure receives a pointer to an array containing the PIN and validates each digit. If any digit is outside its valid range, return the digit's position (1 to 5) in the EAX register; otherwise, return 0. Write a test program that calls Validate_PIN with valid and invalid PINs and verifies the return values.



Digit Number	Range
1	5 to 9
2	2 to 5
3	4 to 8
4	1 to 4
5	3 to 6

Exercise 10: Parity Checking

Implement a procedure that checks the parity (even or odd) of bytes in an array. The procedure returns True (1) in EAX if the bytes have even parity and False (0) if they have odd parity. Write a test program that calls the procedure with arrays having even and odd parity and verifies the return values.