Conditional Loops

===

LOOPZ and LOOPE Instructions

===

The **LOOPZ (Loop if Zero)** and **LOOPE (Loop if Equal)** instructions are conditional loop instructions used in assembly language programming.

They both share the same opcode and have identical behavior based on the condition of the Zero Flag (ZF). Here's a simplified explanation:

ECX (or RCX in 64-bit mode) is the loop counter register. ECX is decremented by 1 in each iteration of the loop.

The **loop continues if** ECX is greater than 0 and the Zero Flag (ZF) is set (indicating the result of the previous operation was zero).

If the condition is met, the program jumps to the specified destination label. If the condition is not met (ECX not greater than 0 or ZF not set), the loop exits, and control proceeds to the next instruction.

For example, the following code snippet will sum the elements of an array using the LOOPZ instruction:

```
; sum the elements of the array `array`
mov ecx, array_size
loopz sum_loop

; sum_loop:
add eax, [array + ecx - 1]
dec ecx
jnz sum_loop
```

In this example, the loop counter register ECX is initialized with the size of the array.

The LOOPZ instruction then decrements ECX and adds the element at array + ECX - 1 to the accumulator register EAX. If ECX is still greater than 0 and the Zero flag is set, the loop will jump back to the sum_loop label.

Otherwise, execution will fall through to the next instruction, which will be the end of the loop.

LOOPZ (Loop if Zero): This instruction is used to create a loop with an additional condition. The condition is that the Zero Flag (ZF) must be set for the loop to continue. Here's the syntax:

```
LOOPZ destination
```

These instructions do not affect any other status flags. LOOPE is essentially the same as LOOPZ, and they **share the same opcode.** They both have the same behavior and conditions as described above.

These instructions are often used for implementing loops in assembly language, where you want to repeat a block of code a specific number of times while a certain condition is met (in this case, the Zero Flag being set). Loop a specific number of times based on the value in the loop counter (ECX) and a condition (Zero Flag set).

The LOOPZ and LOOPE instructions can be used in a variety of other ways as well. For example, they can be used to implement nested loops, to search for a value in an array, or to reverse a string.

==

LOOPNZ (Loop if Not Zero)
LOOPNE (Loop if Not Equal) Instructions

==

The **LOOPNZ (Loop if Not Zero)** and **LOOPNE (Loop if Not Equal)** instructions are used in assembly language programming to create loops that repeat a block of code while a certain condition is met.

These instructions are quite similar and often interchangeable, as they share the same opcode and perform similar tasks.

The LOOPNZ instruction continues looping while two conditions are met: The **unsigned value** of the **ECX register** is **greater than zero** after being decremented. The **Zero Flag (ZF) is clear.** The syntax for LOOPNZ is:

```
LOOPNZ destination
```

Here's how it works:

- Decrement ECX by 1.
- If ECX > 0 and ZF = 0 (i.e., the Zero Flag is clear), jump to the specified destination label.
- If ECX becomes zero or ZF is set, the loop terminates, and control passes to the next instruction.

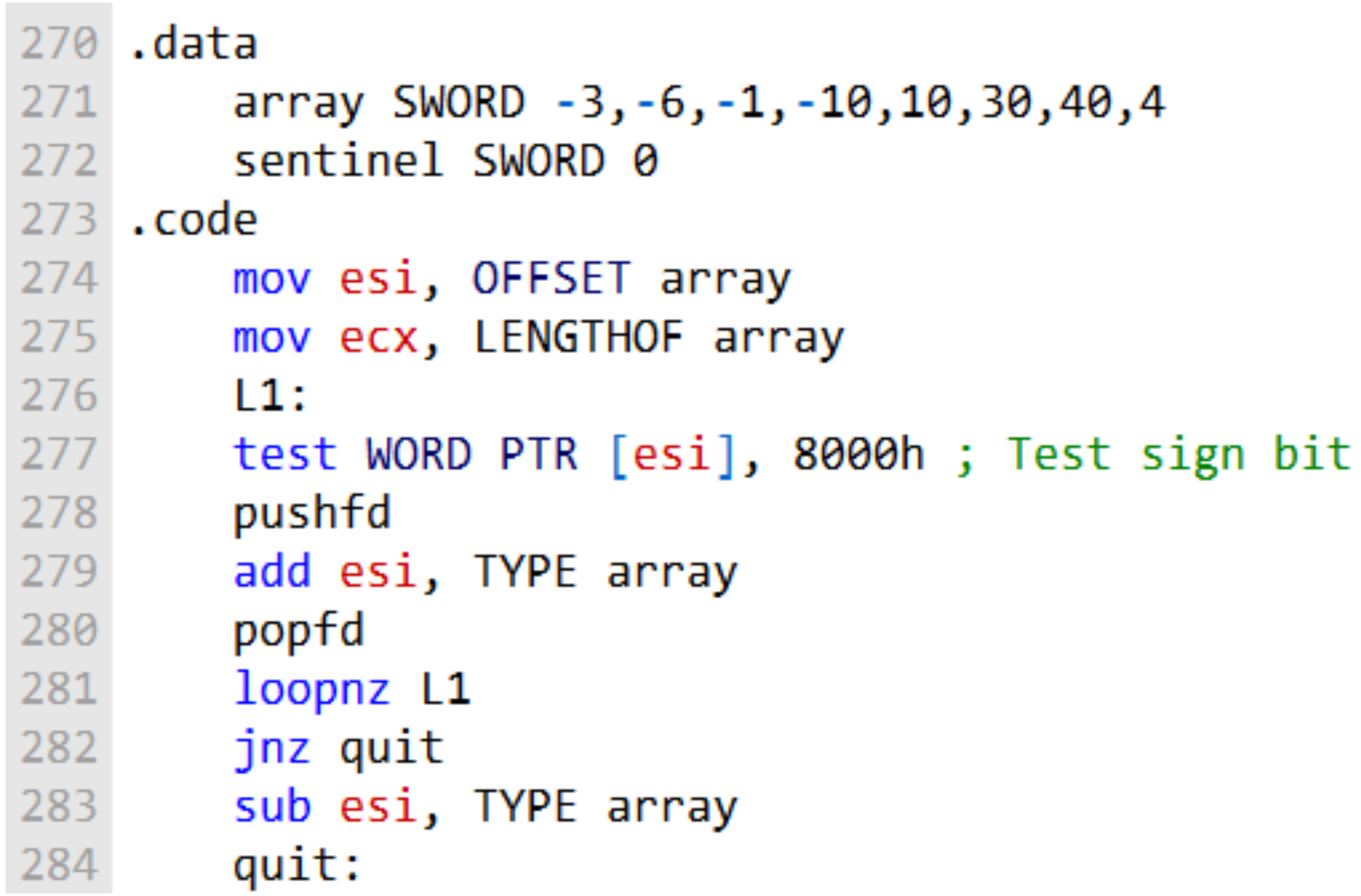
The **LOOPNE instruction** is equivalent to LOOPNZ in terms of functionality and shares the same opcode.

It also performs the following tasks:

- Decrement ECX by 1.
- If ECX > 0 and ZF = 0 (i.e., the Zero Flag is clear), jump to the specified destination label.
- If ECX becomes zero or ZF is set, the loop terminates, and control passes to the next instruction.

In essence, both LOOPNZ and LOOPNE create loops that continue while a counter (usually stored in ECX) is not zero and the Zero Flag is not set. They are often used for iterating through arrays or data structures until a specific condition is met.

Here's an example code excerpt that uses LOOPNZ to scan an array until a non-negative number is found:



```
.data
    array SWORD -3,-6,-1,-10,10,30,40,4
    sentinel SWORD 0
.code
    mov esi, OFFSET array
    mov ecx, LENGTHOF array
    L1:
    test WORD PTR [esi], 8000h ; Test sign bit
    pushfd
    add esi, TYPE array
    popfd
    loopnz L1
    jnz quit
    sub esi, TYPE array
    quit:
```

This code iterates through the array, testing the sign bit of each element, and continues the loop until a nonnegative value is found. If none is found, it stops when ECX becomes zero and jumps to the quit label, where ESI points to the sentinel value (0) located after the array.

Now, let's break down the code inside the .code section:

mov esi, OFFSET array: This instruction initializes the ESI register with the memory address of the array variable, effectively pointing to the beginning of the array.

mov ecx, LENGTHOF array: It loads the ECX register with the length of the array, which is the number of elements in the array.

L1:: This is a label for the beginning of a loop.

test WORD PTR [esi], 8000h: The test instruction checks the sign bit of the current array element by bitwise ANDing it with 8000h (hexadecimal representation of a signed word with the sign bit set).

pushfd: It pushes the processor flags onto the stack. This is done to save the state of the Zero Flag (ZF) because the add instruction that follows modifies the flags.

add esi, TYPE array: ESI is incremented by the size of a single array element (TYPE array), effectively moving to the next position in the array.

popfd: The flags saved by pushfd are popped from the stack, restoring the previous state.

loopnz L1: The LOOPNZ instruction decrements ECX by 1 and checks if ECX > 0 and ZF = 0. If both conditions are met, it jumps to the L1 label, continuing the loop. Otherwise, if ECX becomes zero or ZF is set, the loop terminates.

jnz quit: If the loop completes without finding a nonnegative value, it jumps to the quit label.

sub esi, TYPE array: If a nonnegative value is found, ESI is left pointing to that value.

The code efficiently iterates through the array, testing each element's sign bit. If it finds a nonnegative value, ESI points to that value; otherwise, it points to the sentinel value (0) after the array. This logic allows you to handle different cases depending on the outcome of the loop.

(True/False): The LOOPE instruction jumps to a label when (and only when) the Zero flag is clear.

True. The LOOPE (Loop If Equal) instruction jumps to a label when the Zero flag (ZF) is clear and the ECX register is greater than zero.

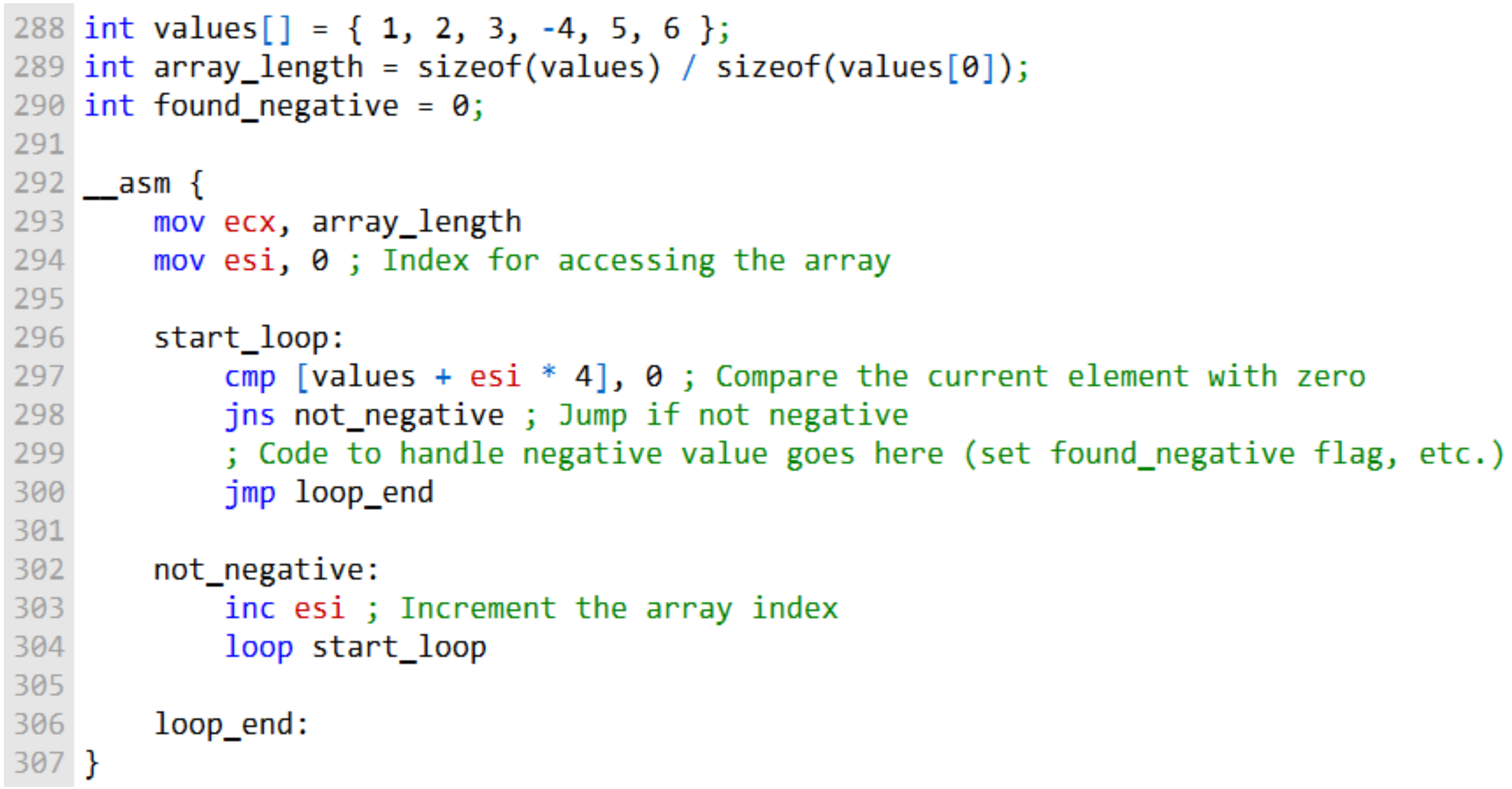
(True/False): In 32-bit mode, the LOOPNZ instruction jumps to a label when ECX is greater than zero and the Zero flag is clear.

True. The LOOPNZ (Loop If Not Zero) instruction in 32-bit mode jumps to a label when the ECX register is greater than zero and the Zero flag (ZF) is clear.

(True/False): The destination label of a LOOPZ instruction must be no farther than ±128 or ±127 bytes from the instruction immediately following LOOPZ.

False. The destination label of a LOOPZ instruction must be no farther than ±128 bytes from the instruction immediately following LOOPZ. This is because the offset for a short jump (like the one used by LOOPZ) is limited to a signed 8-bit value, which covers a range of -128 to +127 bytes.
Modify the LOOPNZ example in Section 6.4.2 so that it scans for the first negative value in the array. Change the array initializers so they begin with positive values.

To modify the LOOPNZ example to scan for the first negative value in the array, you can change the array initialization to start with positive values. Here's an example in C:

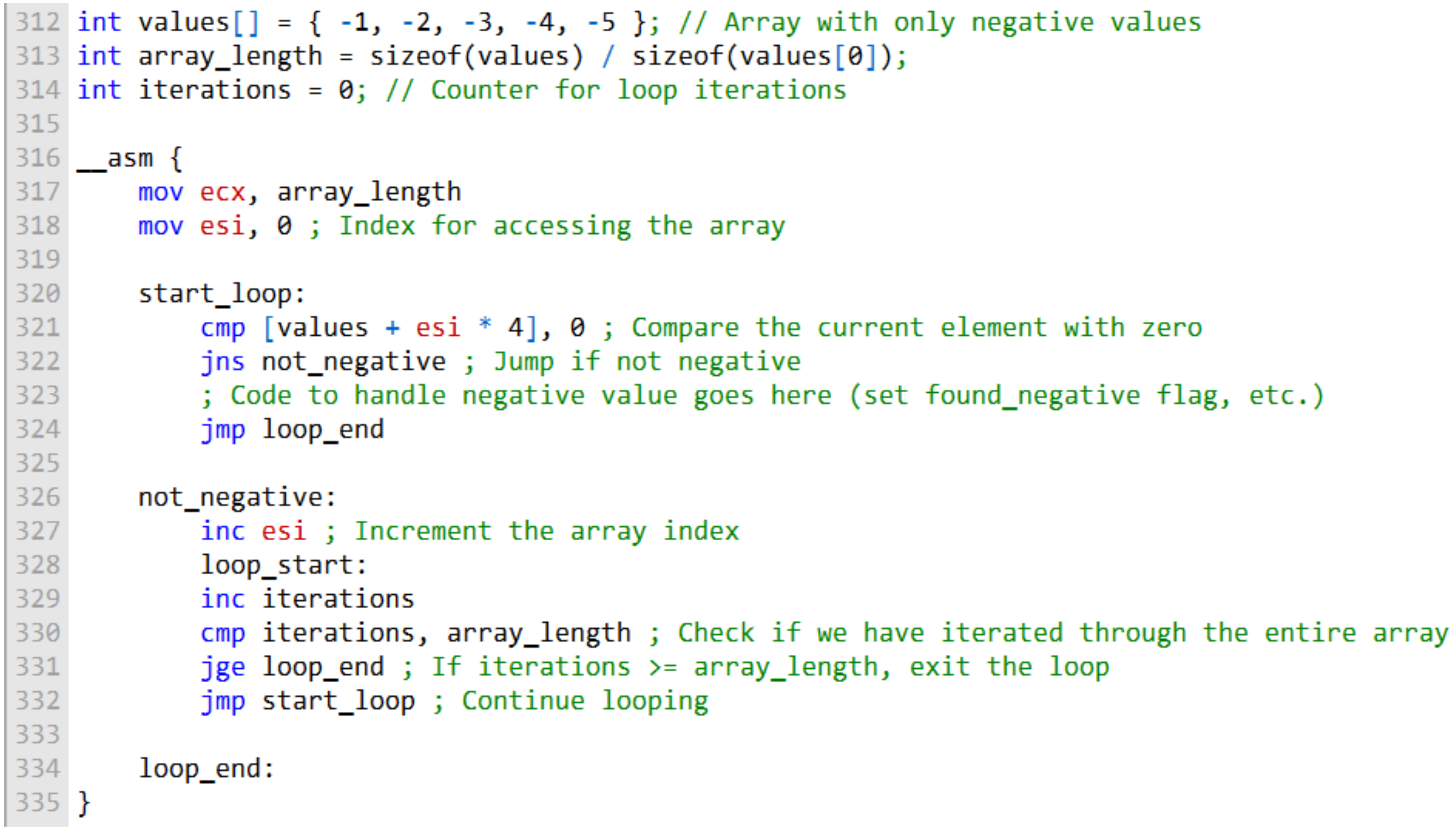


```
int values[] = { 1, 2, 3, -4, 5, 6 };
int array_length = sizeof(values) / sizeof(values[0]);
int found_negative = 0;

__asm {
    mov ecx, array_length
    mov esi, 0 ; Index for accessing the array

    start_loop:
        cmp [values + esi * 4], 0 ; Compare the current element with zero
        jns not_negative ; Jump if not negative
        ; Code to handle negative value goes here (set found_negative flag, etc.)
        jmp loop_end

    not_negative:
        inc esi ; Increment the array index
        loop start_loop

    loop_end:
}
```

Challenge: The LOOPNZ example in Section 6.4.2 relies on a sentinel value to handle the possibility that a positive value might not be found. What might happen if you removed the sentinel?

If you remove the sentinel value from the LOOPNZ example, it means that there is no clear indicator to stop the loop when a positive value is not found in the array. Without a sentinel value, the loop would continue running indefinitely, potentially causing an infinite loop in your program. To avoid this, it's essential to have some mechanism, like a sentinel value or a counter, to terminate the loop when a specific condition is not met.

Certainly, here's an example of how you might modify the LOOPNZ code to remove the sentinel value and handle the case where a positive value might not be found. In this modified code, we use a counter to limit the number of iterations:



```
int values[] = { -1, -2, -3, -4, -5 }; // Array with only negative values
int array_length = sizeof(values) / sizeof(values[0]);
int iterations = 0; // Counter for loop iterations

__asm {
    mov ecx, array_length
    mov esi, 0 ; Index for accessing the array

    start_loop:
        cmp [values + esi * 4], 0 ; Compare the current element with zero
        jns not_negative ; Jump if not negative
        ; Code to handle negative value goes here (set found_negative flag, etc.)
        jmp loop_end

    not_negative:
        inc esi ; Increment the array index
        loop_start:
        inc iterations
        cmp iterations, array_length ; Check if we have iterated through the entire array
        jge loop_end ; If iterations >= array_length, exit the loop
        jmp start_loop ; Continue looping

    loop_end:
}
```

In this code, we've introduced an iterations counter, which increments with each iteration of the loop. If iterations becomes greater than or equal to array_length, it means we've iterated through the entire array without finding a positive value, and we exit the loop. This prevents an infinite loop from occurring when there are no positive values in the array.