WHILE Loops

While loop

The notes you provided for WHILE loops in assembly language are a bit confusing and unexplained, so I will try to rewrite them in a more clear and concise way.

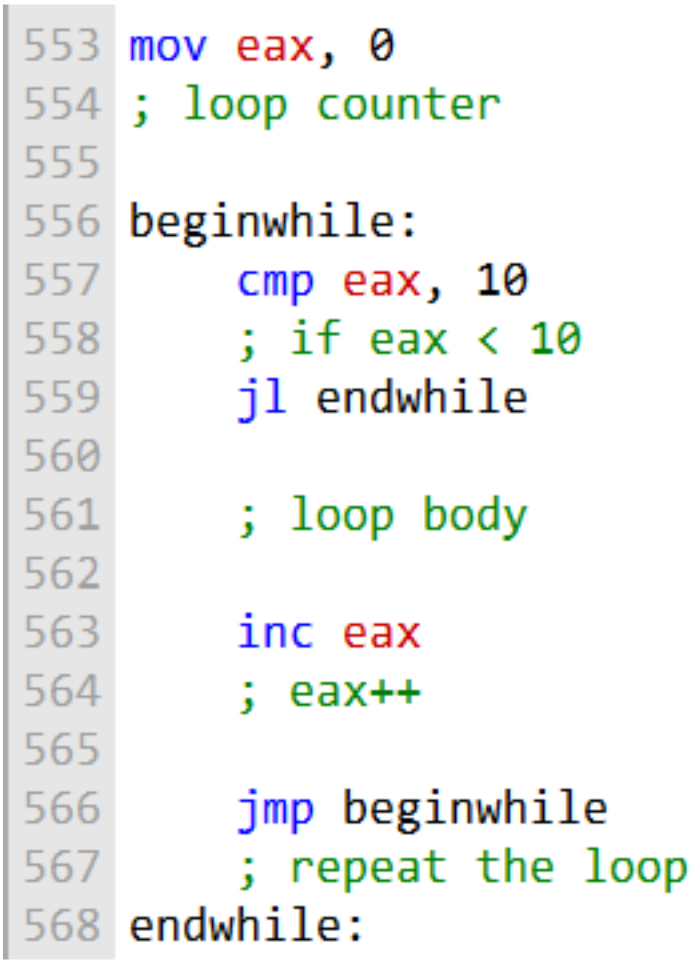
WHILE loops in assembly language work in a similar way to WHILE loops in high-level languages. The loop first checks a condition. If the condition is true, the loop body is executed.

Then, the condition is checked again. If the condition is still true, the loop body is executed again. This process continues until the condition becomes false.

To implement a WHILE loop in assembly language, you can use the following steps:

Initialize a register to store the loop condition. Check the loop condition. If the condition is false, jump to the end of the loop. Execute the loop body. Update the loop condition. Jump back to step 2.

The following assembly code shows a simple WHILE loop:



```
mov eax, 0
; loop counter

beginwhile:
    cmp eax, 10
    ; if eax < 10
    jl endwhile

    ; loop body

    inc eax
    ; eax++

    jmp beginwhile
    ; repeat the loop
endwhile:
```

This loop will print the numbers from 0 to 9 to the console.

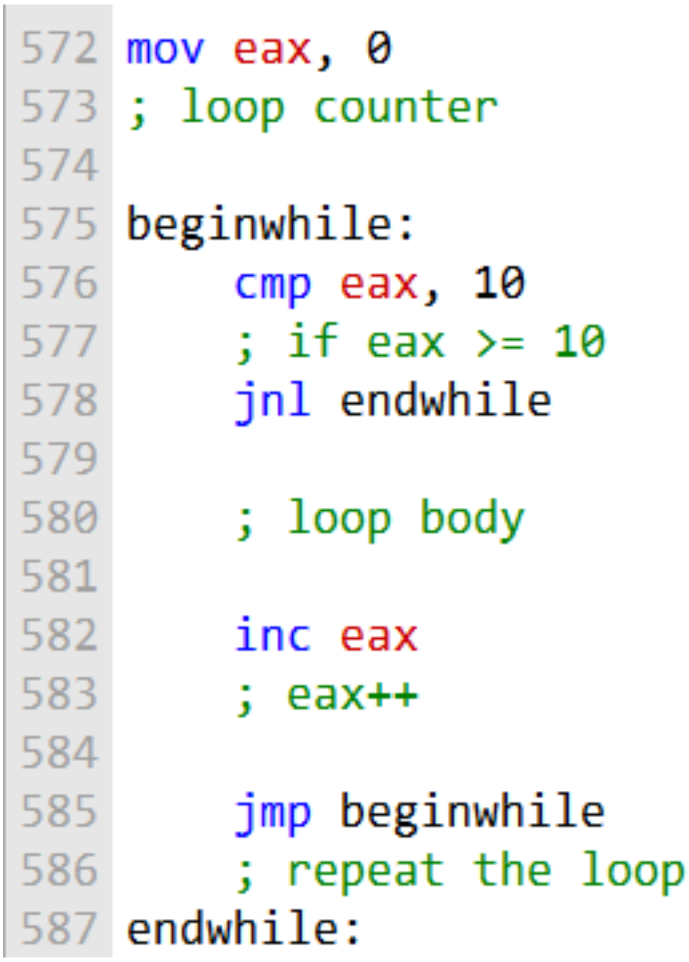
Reverse the Loop condition

As the notes you provided mention, it is often convenient to reverse the loop condition in assembly language. This means that the loop will continue to iterate as long as the condition is false.

To reverse the loop condition, you can use the jnl instruction instead of the jl instruction.

The jnl instruction jumps to the specified label if the condition is not less than or equal to zero.

For example, the following assembly code is equivalent to the previous example, but it reverses the loop condition:



```
mov eax, 0
; loop counter

beginwhile:
    cmp eax, 10
    ; if eax >= 10
    jnl endwhile

    ; loop body

    inc eax
    ; eax++

    jmp beginwhile
    ; repeat the loop
endwhile:
```

Copy and restore the loop variable

If the loop variable is used inside the loop body, you need to copy it to a register before the loop starts.

Then, you need to restore the value of the loop variable at the end of the loop.
This is necessary because assembly language is a stack-based language.

This means that all variables are stored on the stack. When a function is called, the parameters are pushed onto the stack.

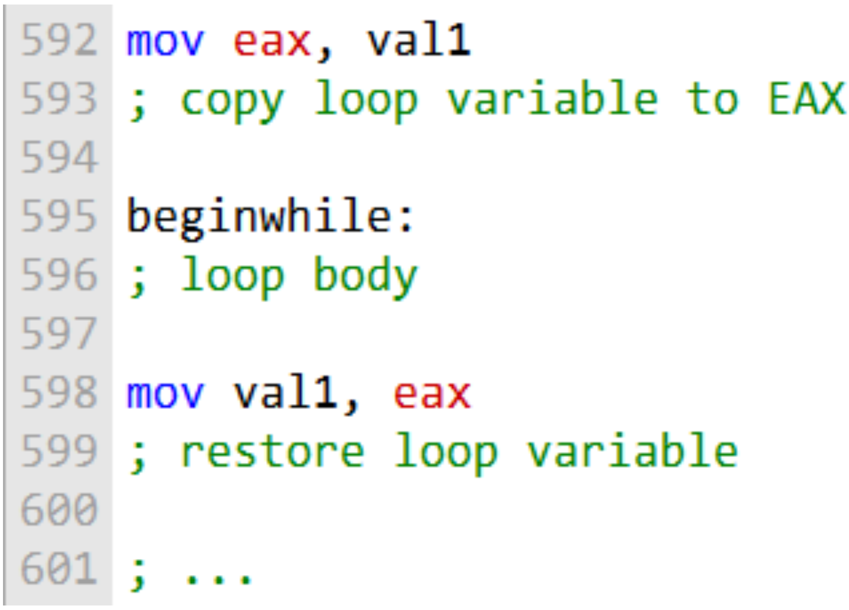
When the function returns, the parameters are popped off the stack.

If you use a loop variable inside the loop body, the loop variable will be pushed onto the stack when you call the loop body.

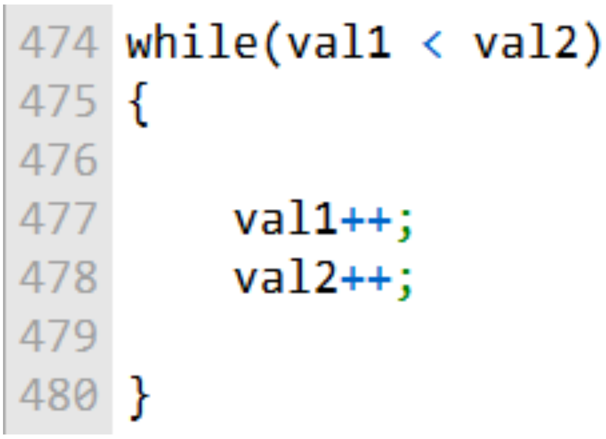
When the loop body returns, the loop variable will be popped off the stack. This means that the loop variable will be modified by the loop body.

To avoid this problem, you need to copy the loop variable to a register before the loop starts. Then, you need to restore the value of the loop variable at the end of the loop.

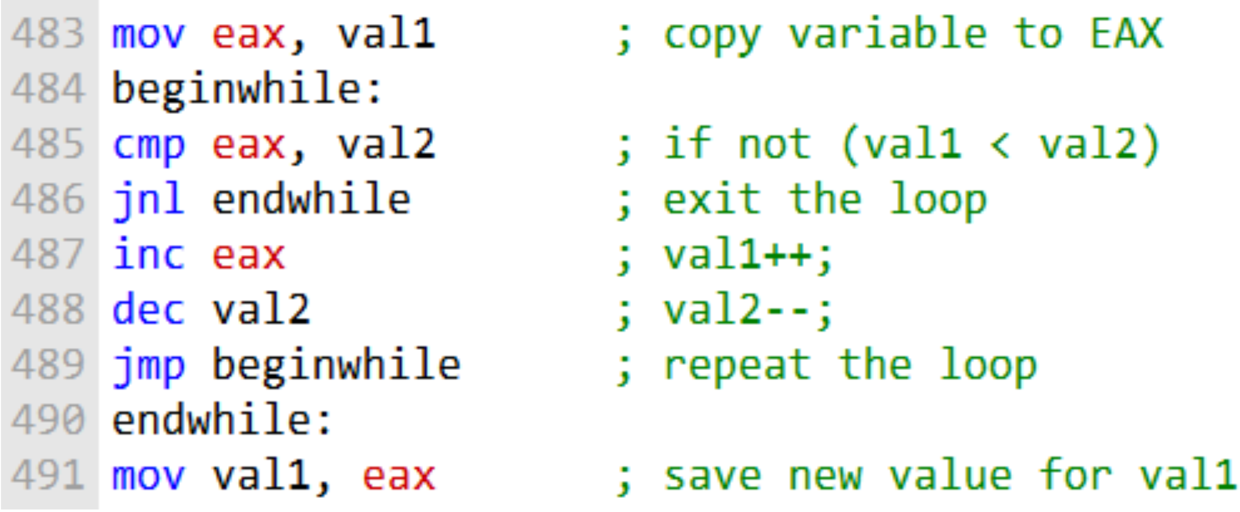
For example, the following assembly code shows how to copy and restore the loop variable:



```
mov eax, val1
; copy loop variable to EAX

beginwhile:
; loop body

mov val1, eax
; restore loop variable

; ...
```



```
while(val1 < val2)
{

    val1++;
    val2++;

}
```



```
mov eax, val1         ; copy variable to EAX
beginwhile:
cmp eax, val2         ; if not (val1 < val2)
jnl endwhile          ; exit the loop
inc eax               ; val1++;
dec val2              ; val2--;
jmp beginwhile        ; repeat the loop
endwhile:
mov val1, eax         ; save new value for val1
```

The first instruction copies the value of the variable val1 to the register eax. This is done because the loop will be operating on eax, so it is important to have a copy of val1 in a register.

The next instruction is a cmp instruction that compares the values of eax and val2. If eax is not less than val2, then the loop condition is false and the program will jump to the endwhile label.

If the loop condition is true, then the program will execute the following instructions:

Increment the value of eax by 1. This corresponds to the val1++ statement in the C++ code. Decrement the value of val2 by 1. This corresponds to the val2-- statement in the C++ code.

Jump to the beginwhile label to repeat the loop. The endwhile label is used to mark the end of the loop. When the program reaches the endwhile label, it will exit the loop and continue with the rest of the program.

The last instruction copies the value of eax to the variable val1. This is done because we need to save the new value of val1 in the variable before exiting the loop.

The JNL instruction is used to jump to the endwhile label if the loop condition is not true. This instruction is used because val1 and val2 are signed integers. If val1 is greater than val2, then the loop condition is false and we need to exit the loop.

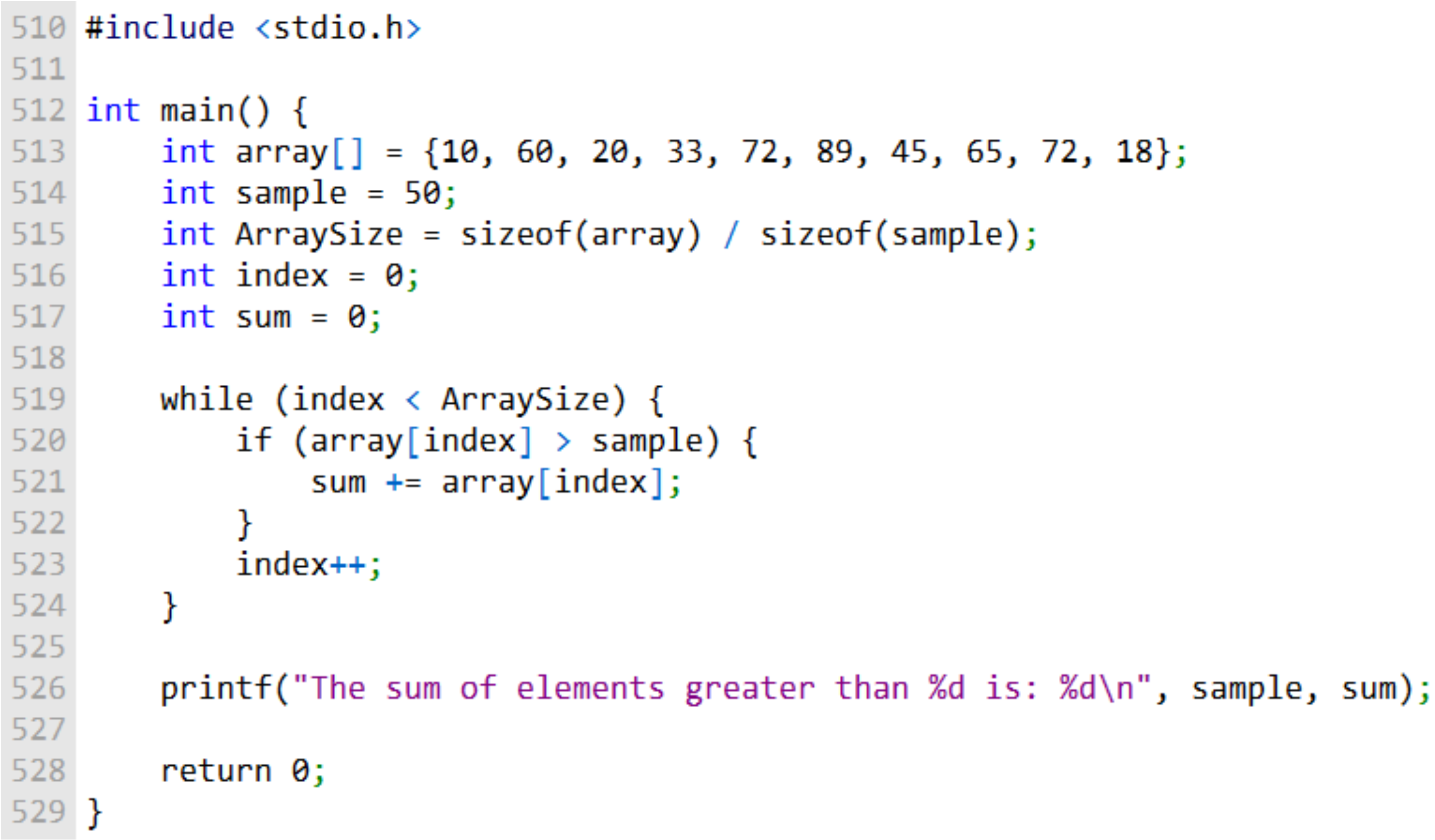
It is important to note that the eax register is used as a proxy for the variable val1 inside the loop. This means that all references to val1 must be through the eax register. This is because the loop will be operating on eax, not val1.

In this code:

The mov instruction copies the value of val1 to the EAX register. The beginwhile label marks the beginning of the loop. The cmp instruction compares the values in EAX and val2.

The jnl instruction jumps to the endwhile label if not (EAX < val2), effectively ending the loop. inc eax increments the value in EAX, representing val1++. dec val2 decrements the value in val2, representing val2--.

The jmp beginwhile instruction jumps back to the beginwhile label to repeat the loop until the condition is no longer met. Finally, the endwhile label marks the end of the loop, and mov val1, eax saves the updated value of val1 back to the variable val1.



```c
#include <stdio.h>

int main() {
    int array[] = {10, 60, 20, 33, 72, 89, 45, 65, 72, 18};
    int sample = 50;
    int ArraySize = sizeof(array) / sizeof(sample);
    int index = 0;
    int sum = 0;

    while (index < ArraySize) {
        if (array[index] > sample) {
            sum += array[index];
        }
        index++;
    }

    printf("The sum of elements greater than %d is: %d\n", sample, sum);

    return 0;
}
```

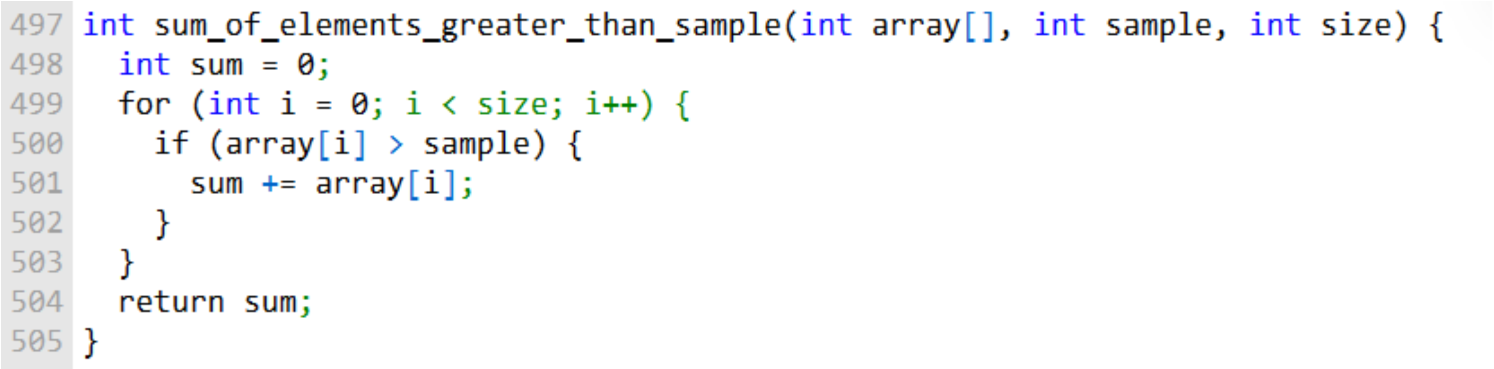
It defines an integer array array with ten elements. It initializes a variable sample with the value 50. It calculates the size of the array using sizeof(array) divided by sizeof(sample) to get the number of elements in the array, and stores it in ArraySize.

It initializes an index variable index to 0 to keep track of the current position in the array. It initializes a sum variable to 0 to accumulate the sum of elements greater than sample. The while loop iterates through the elements of the array using the index variable. For each element, it checks if it is greater than sample.

If it is, it adds that element's value to the sum. The loop continues until index is less than ArraySize. Finally, it prints the sum of elements in the array that are greater than the sample. If you have any further questions or need additional explanations, please let me know

----

The C++ code you provided is a good example of a nested IF statement inside a WHILE loop. The logic of the code is as follows:



```
int sum_of_elements_greater_than_sample(int array[], int sample, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (array[i] > sample) {
      sum += array[i];
    }
  }
  return sum;
}
```

Initialize the variables sum, sample, ArraySize, index, and sum. Enter the WHILE loop. Compare the value of index to the value of ArraySize.

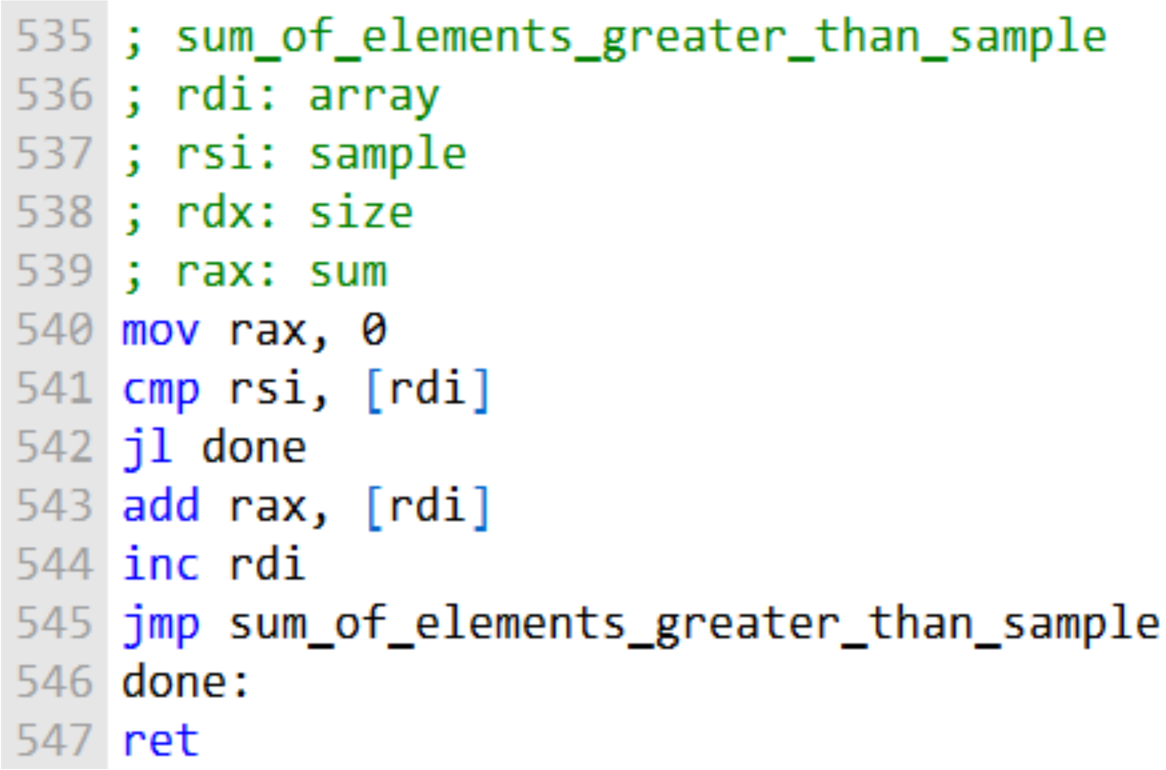
If index is less than ArraySize, proceed to step 4. Otherwise, exit the loop. Compare the value of array[index] to the value of sample.

If array[index] is greater than sample, add the value of array[index] to the variable sum. Increment the value of index.

Repeat steps 3-5 until the WHILE loop condition is no longer met. Exit the WHILE loop and store the value of sum in the variable sum.

The assembly code you provided is a good translation of the C++ code. It uses registers to store the values of the variables sum, sample, array, ArraySize, and index. It also uses labels to mark the different parts of the code flow.

Here is a brief explanation of the assembly code:



```
; sum_of_elements_greater_than_sample
; rdi: array
; rsi: sample
; rdx: size
; rax: sum
mov rax, 0
cmp rsi, [rdi]
jl done
add rax, [rdi]
inc rdi
jmp sum_of_elements_greater_than_sample
done:
ret
```

This code is more efficient because it avoids the overhead of branching.
