Kevin Gamage

Assignment 2

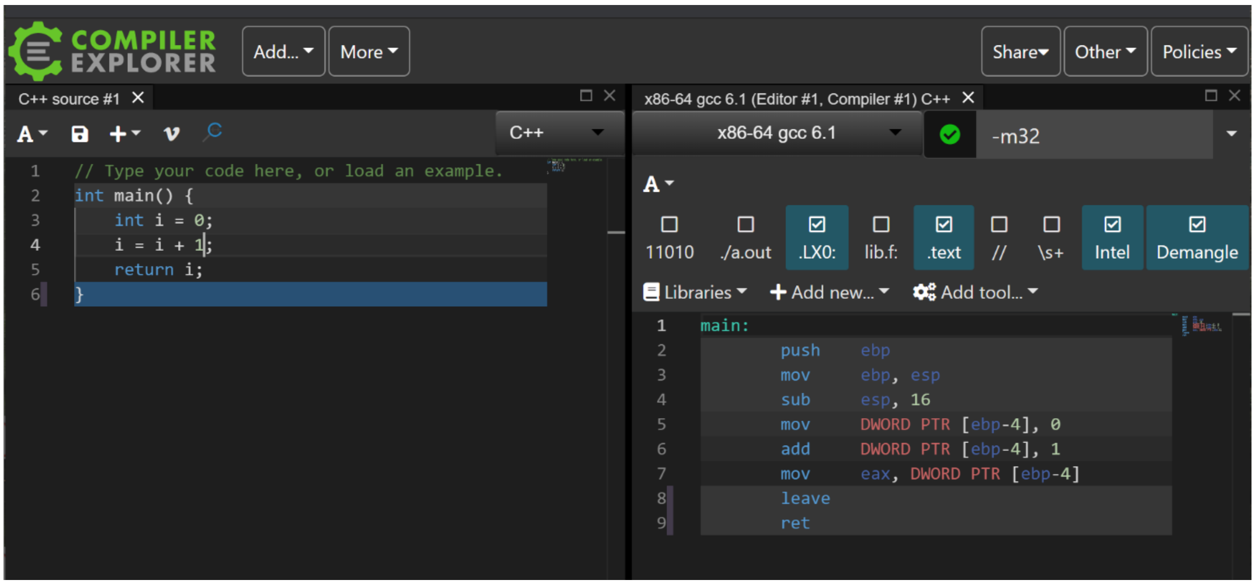
Problem 1.

The subroutine has 3 arguments which are ebp + 8, ebp + 12, and ebp + 16.

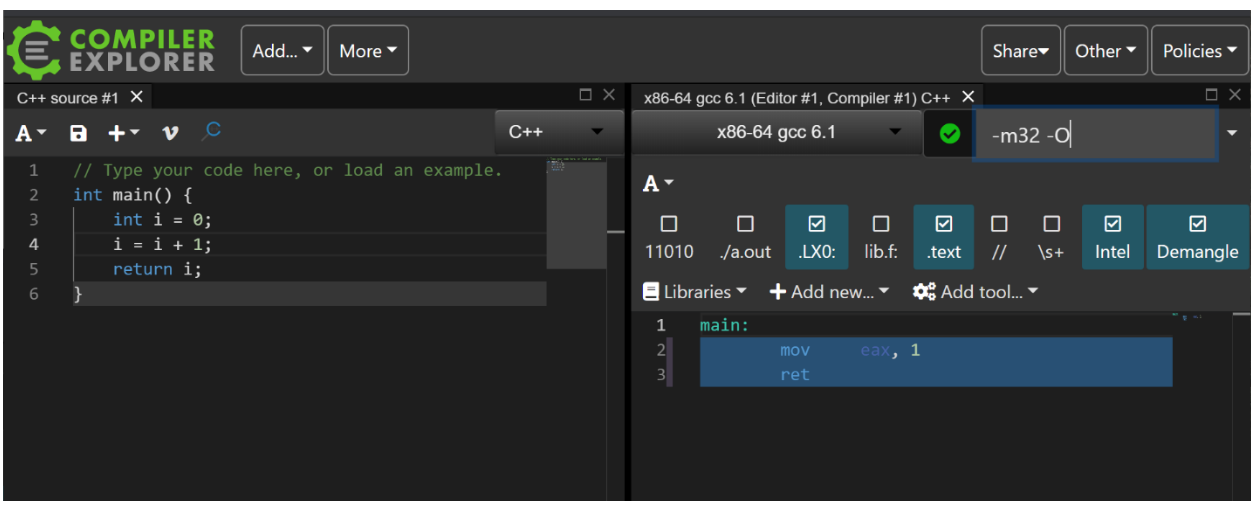
Problem 2.

- Increment operation:

i = i + 1 unoptimized

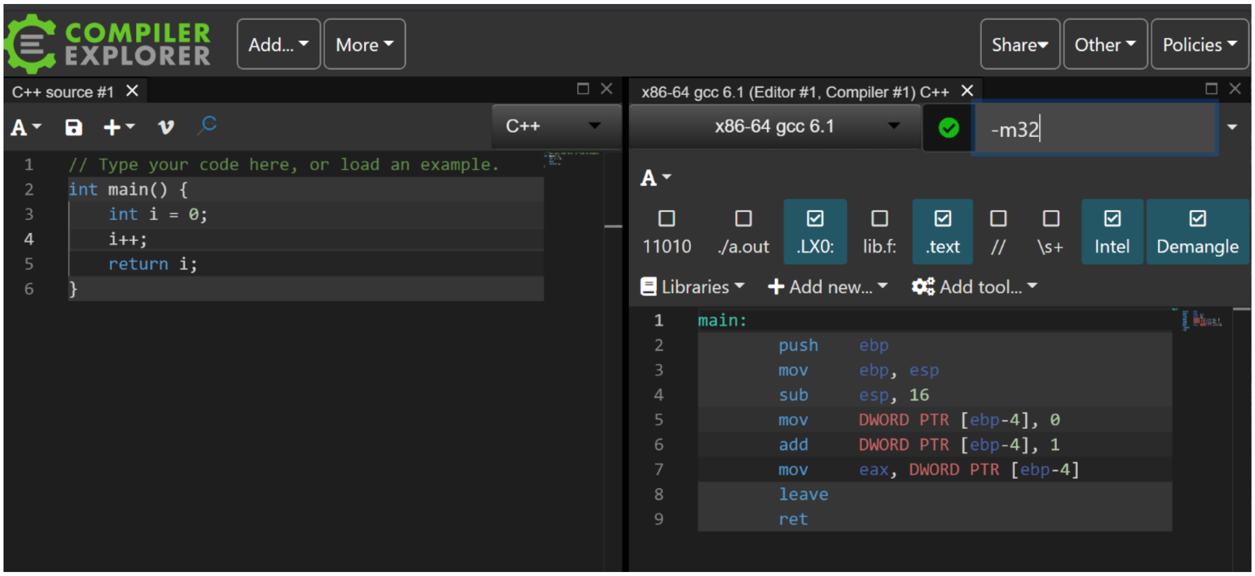


i = i + 1 optimized

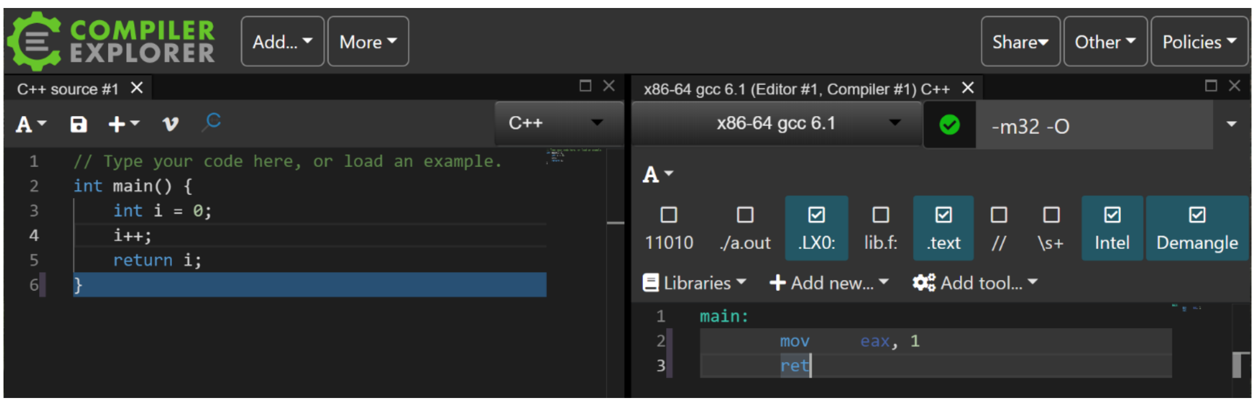


Comparing the i = i + 1 optimized version versus the unoptimized version, we see that the optimized version simply calculated the final result of what i should be and stored it in the eax register before returning. The unoptimized version stores i as a local variable, increments it, and moves its value into the eax register.

i++ unoptimized



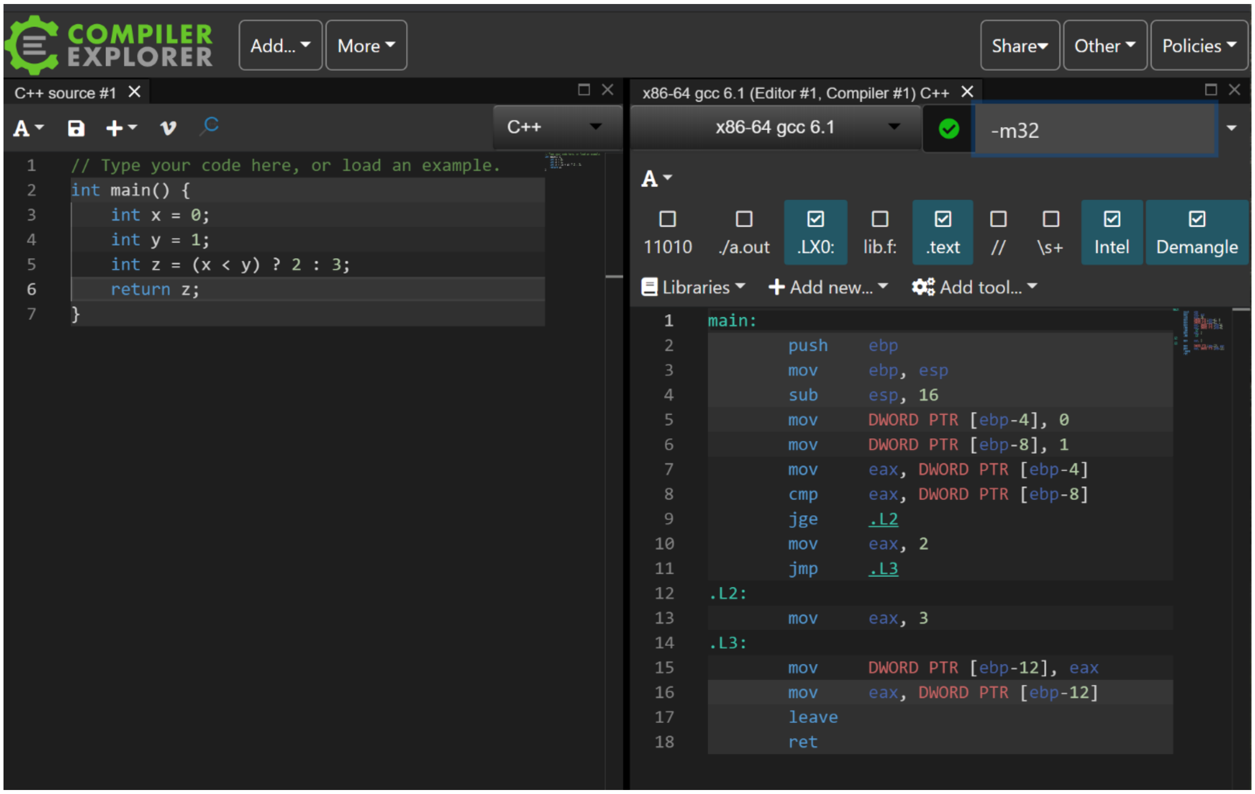
i++ optimized



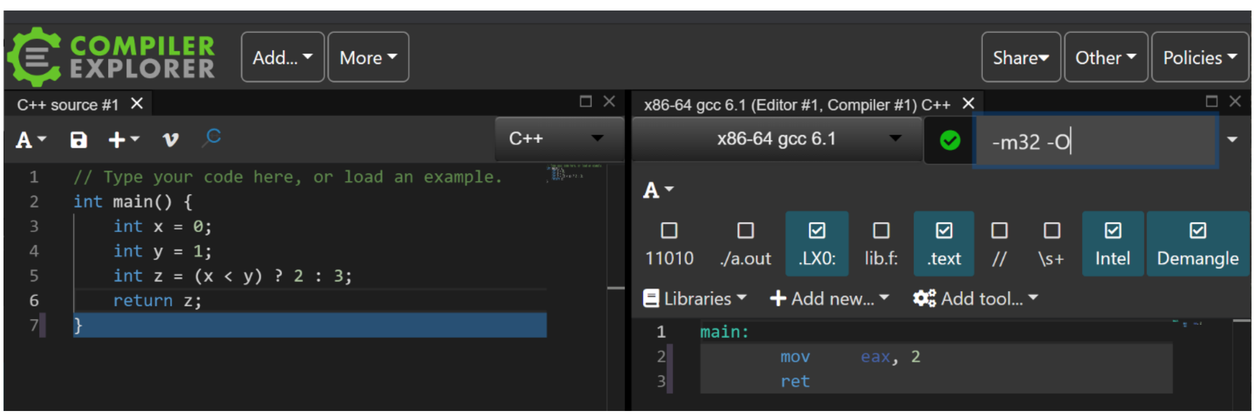
When we examine i++ optimized and its unoptimized version, we see the same result as we saw with the i = i + 1 unoptimized and optimized versions. In this case, we see that the two increment methods have the same optimized and unoptimized versions.

- Ternary, if-else

Ternary operator unoptimized:

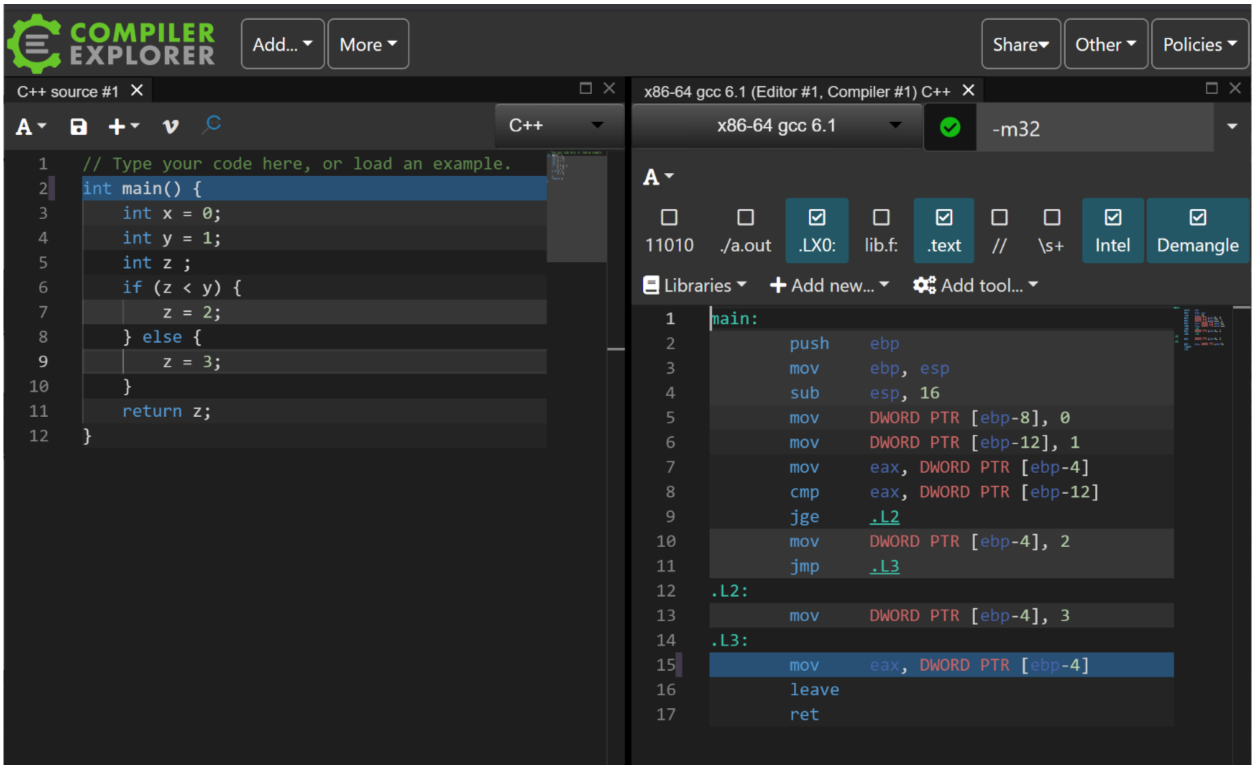


Ternary operator optimized:

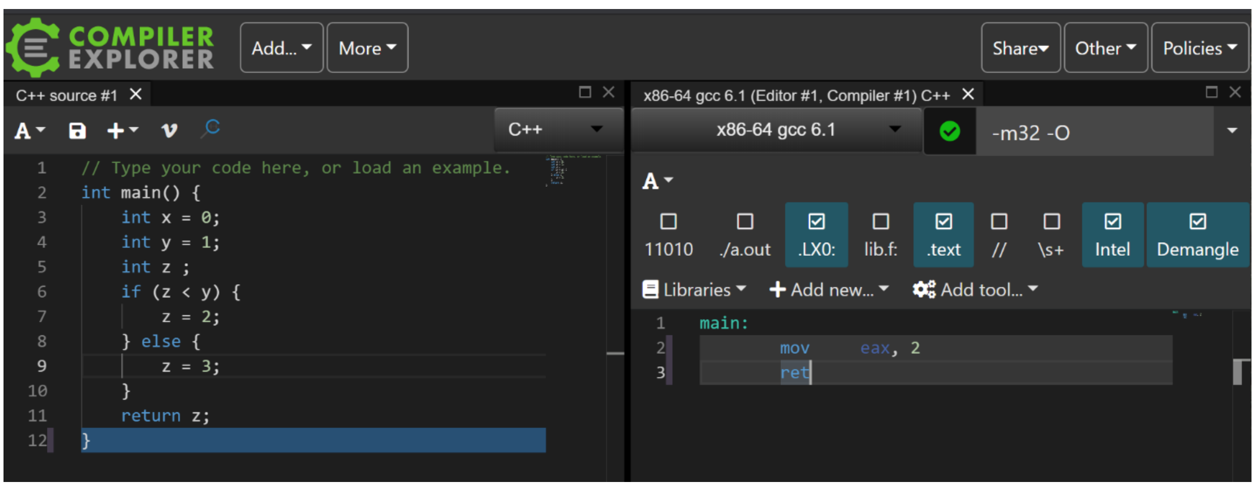


The unoptimized version of the ternary operator performs all the comparisons and assignments of values to variables in the source code. Comparing this with its optimized version, we see that the optimized version simply loads the result of z should be equal to into the eax register and returns.

If – else unoptimized:



If – else optimized:

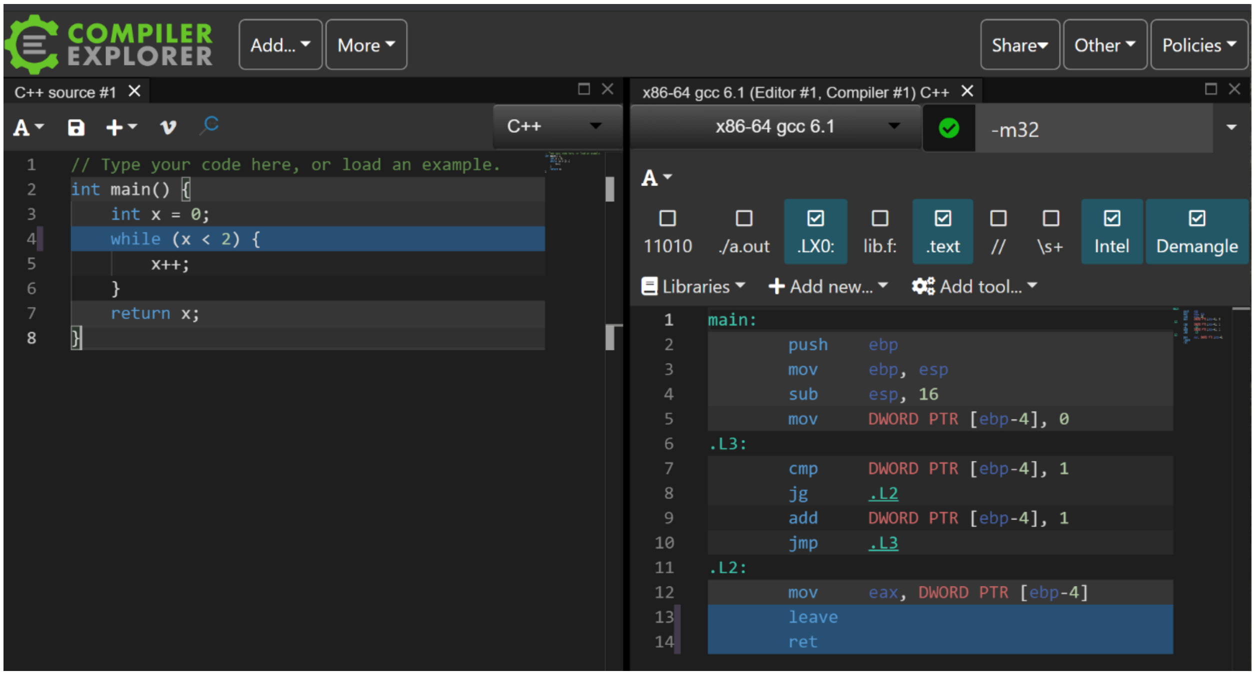


We see comparing the optimized version of the if-else with its unoptimized version that while the assembly of the unoptimized version stores and compares the values of the variables, the optimized version simply returns the value that z would be at the end of the comparisons and assignments and returns it directly.

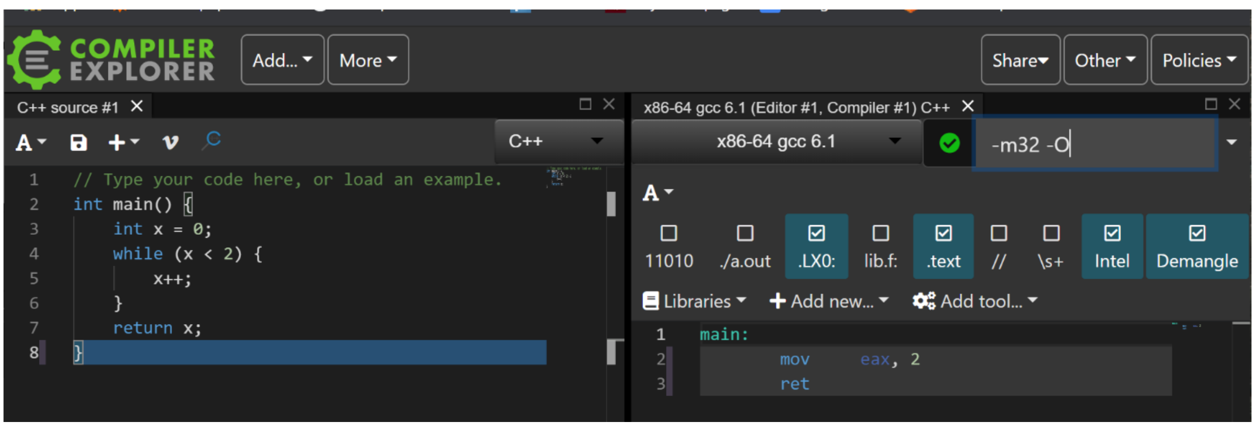
There is a difference between the ternary operator and if-else unoptimized versions. The ternary operator stores the value 2 or 3 in the eax register before assigning z the value. However, the if-else directly assigns z the value 2 or 3 in the if-else clauses. The optimized versions of the two operations appear the same.

- While, For, Do-while

While unoptimized:

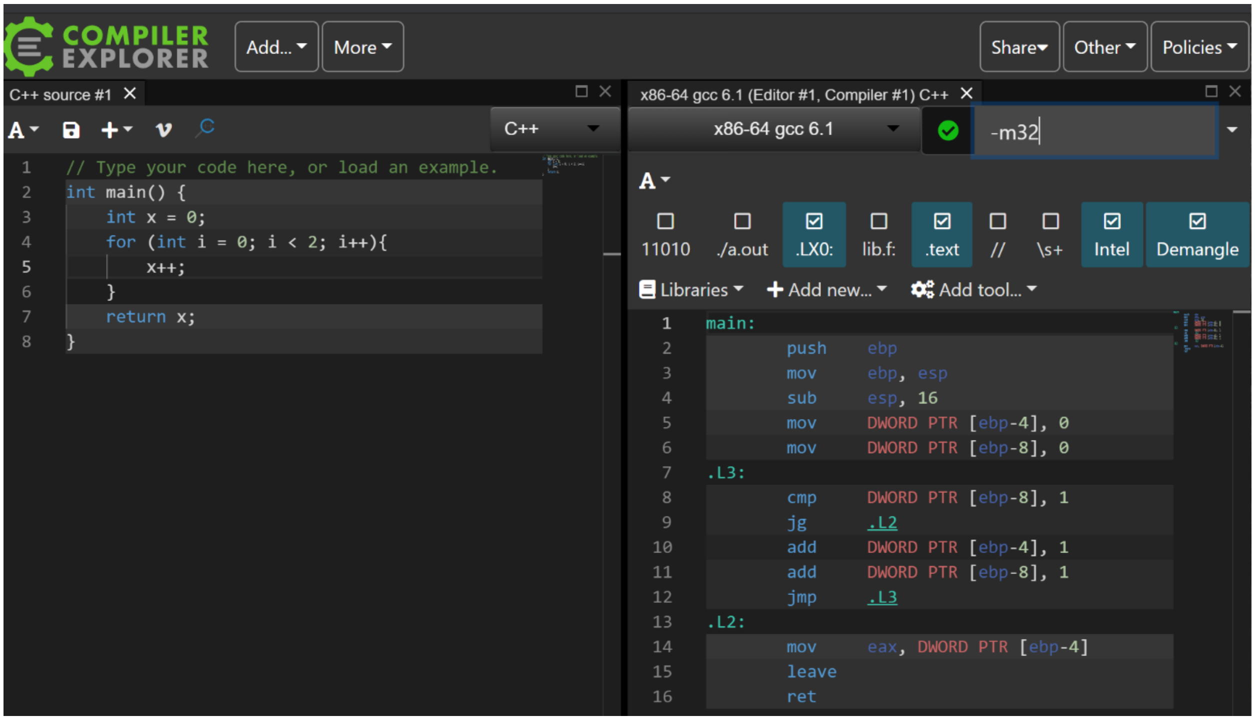


While optimized:

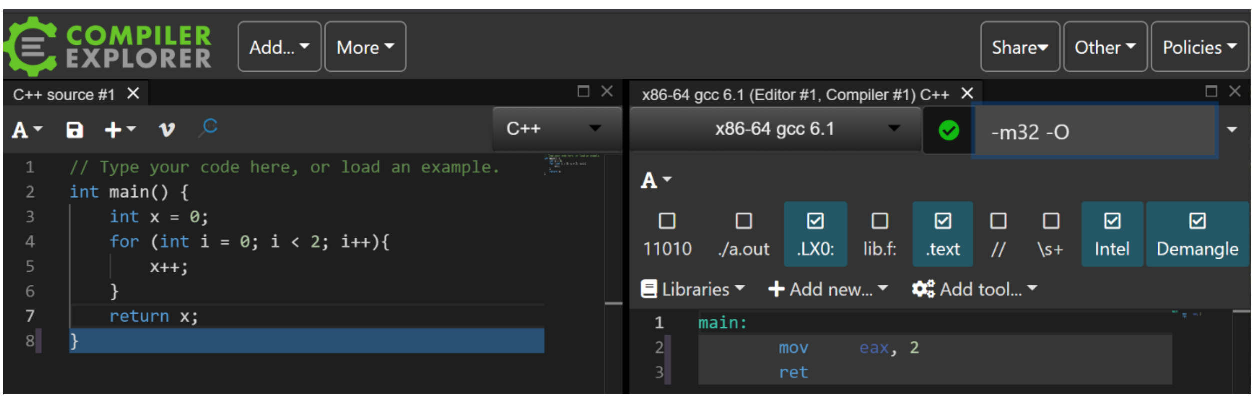


As with the previous examples, the difference between the optimized and unoptimized version of the while loop is that the optimized version contains the final result of the set of operations.

For unoptimized:



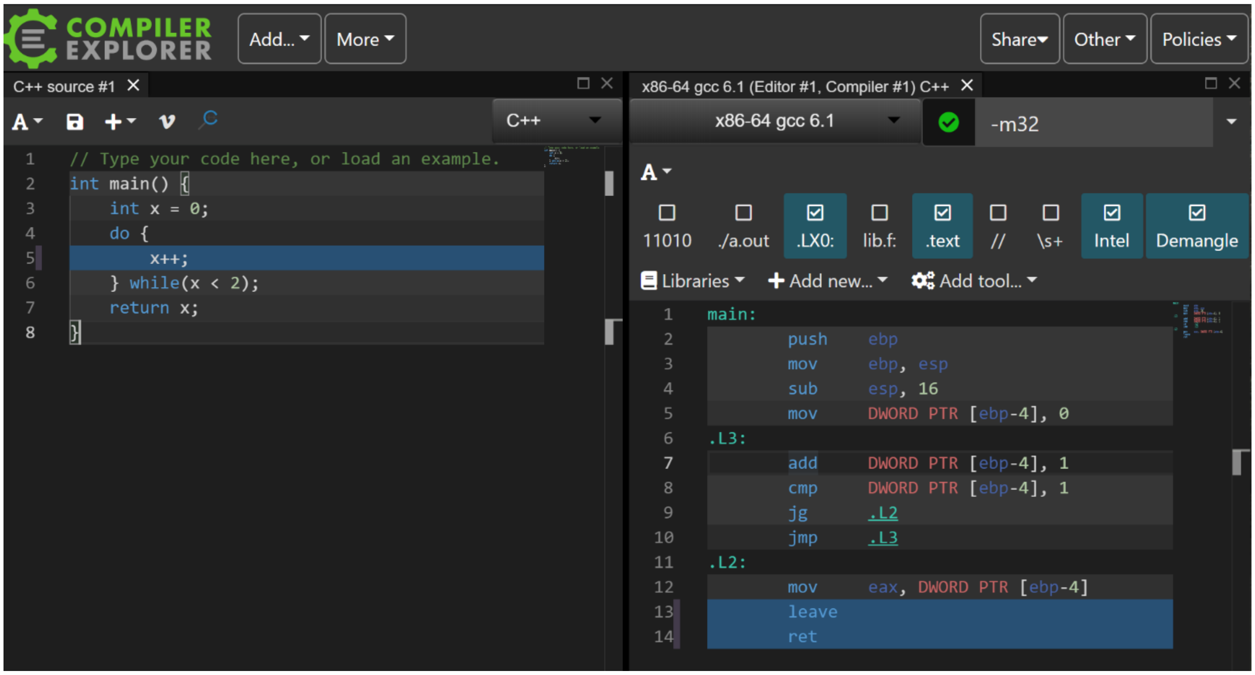
For optimized:



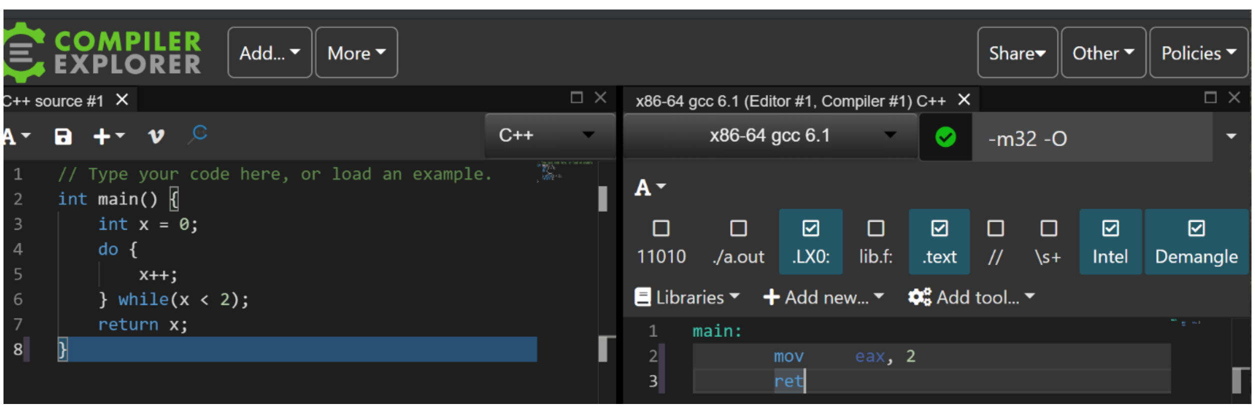
Similar to the previous examples, the difference between the optimized and unoptimized version of the for loop is that the optimized version contains the final result of the set of operations.

In the unoptimized version of the for and the while, the only difference between the two is that the for loop is also updating the i counter variable. The optimized version of both loops is the same.

Do-while unoptimized:



Do-while optimized:



The do-while optimized version contains the same assembly code as the for and while loops optimized versions.

The unoptimized version of the do-while is different from the control flow of the for and while loops in that the do-while does not immediately compare the value of the control variable. Instead it first increments it and then compares it to 1 before jumping to either the beginning or the loop or outside it.

Problem 3.

The value returned by the subroutine is (argument1 – argument2).

Problem 4.

Function Five has 3 arguments and function Six has at least 3 arguments.

Problem 5.

The function Six calls the function Five using the three arguments it was given (arg1, arg2, arg3) but passes them to function Five in the order (arg3, arg1, arg2).

Problem 6.

I was only able to find 12 combinations of the EFLAGS using the CMP operation. Since CMP behaves the same as SUB and sets the EFLAGS accordingly, I used SUB to show the value stored in the AL register after the operation executed.

AL	BL	SUB AL, BL	ZF	CF	PF	AF	SF	OF
0	0	0b00000001	1	0	1	0	0	0
1	0	0b00000001	0	0	0	0	0	0
3	0	0b00000011	0	0	1	0	0	0
0	1	0b11111111	0	1	1	1	1	0
0	3	0b11111101	0	1	0	1	1	0
0	-1	0b00000001	0	1	0	1	0	0
0	-3	0b00000011	0	1	1	1	0	0
-1	0	0b11111111	0	0	1	0	1	0
-3	0	0b11111101	0	0	0	0	1	0
-128	1	0b01111111	0	0	0	1	0	1
1	-128	0b10000001	0	1	1	0	1	1
1	-127	0b10000000	0	1	0	0	1	1

Problem 7.

For this problem, I was able to create source code of a program that was nearly identical to the assembly code. The program starts by initializing an array with the values shown in the assembly code. It then passed the reference of the array to the function Z1fPi. Z1fPi then calculated the sum of the values in the array and stored it in a short. The sum was then returned to the main function and typecast to an int. The result is 28914.

```
short Z1fPi(int array[]){
    short sum = 0;

    for(int i = 0; array[i] != 0; i++){
        sum += array[i];
    }

    return sum;
}
```

```
int main() {
    int array[] = {123434, 9000, 2243244, 34250234, 234234, 0};
    return (int)Z1fPi(array);
}
```

Problem 8.

I was able to recover the source code of this assembly code and determined that the return value is 9000. The program takes two arguments from the user, 10 and 9, and converts them to integers. It then passes the reference of the two integers to the Z1fPi function and stores arg2 * 9000 in arg1 which is 81000 . It then sets arg2 equal to 9000 – arg1 which is -72000. The program then returns arg1 + arg2 which is 9000.

```
#include <stdlib.h>

#define NINETHOUSAND 9000

void Z1fPiS(int &arg1, int &arg2){
    int x = NINETHOUSAND;
    arg1 = arg2 * NINETHOUSAND;
    arg2 = 9000 - arg1;
}

int main(int argc, char* argv[]) {
    int arg1 = atoi(argv[1]);
    int arg1 = atoi(argv[2]);

    Z1fPiS(arg1, arg2);

    return arg1 + arg2;
}
```