

Assignment 2

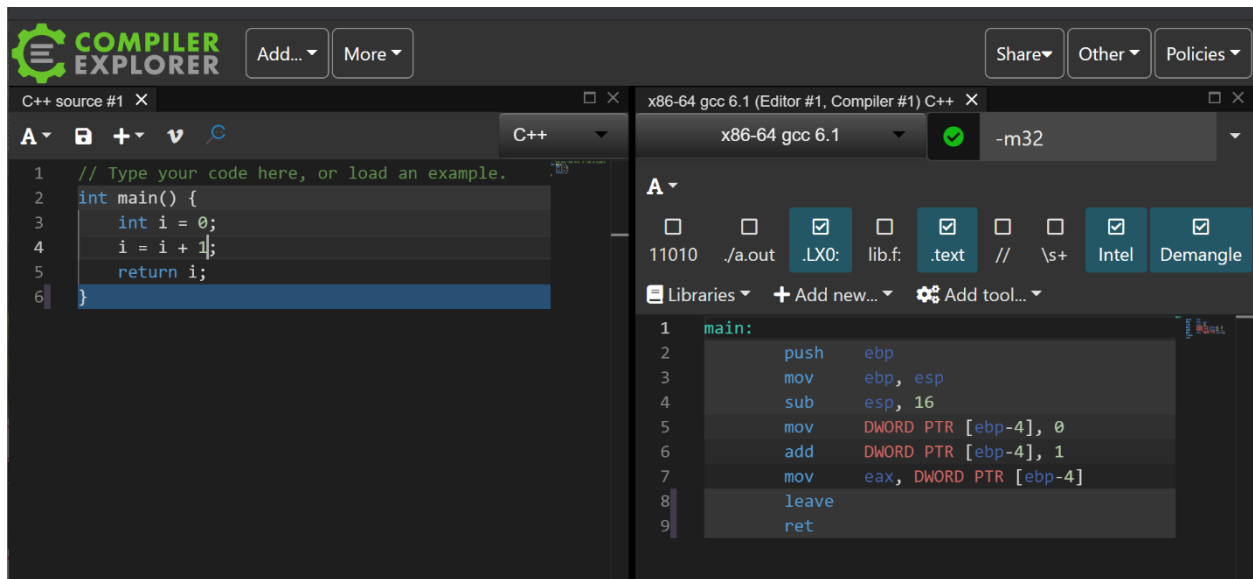
Problem 1.

The subroutine has 3 arguments which are $\text{ebp} + 8$, $\text{ebp} + 12$, and $\text{ebp} + 16$.

Problem 2.

- Increment operation:

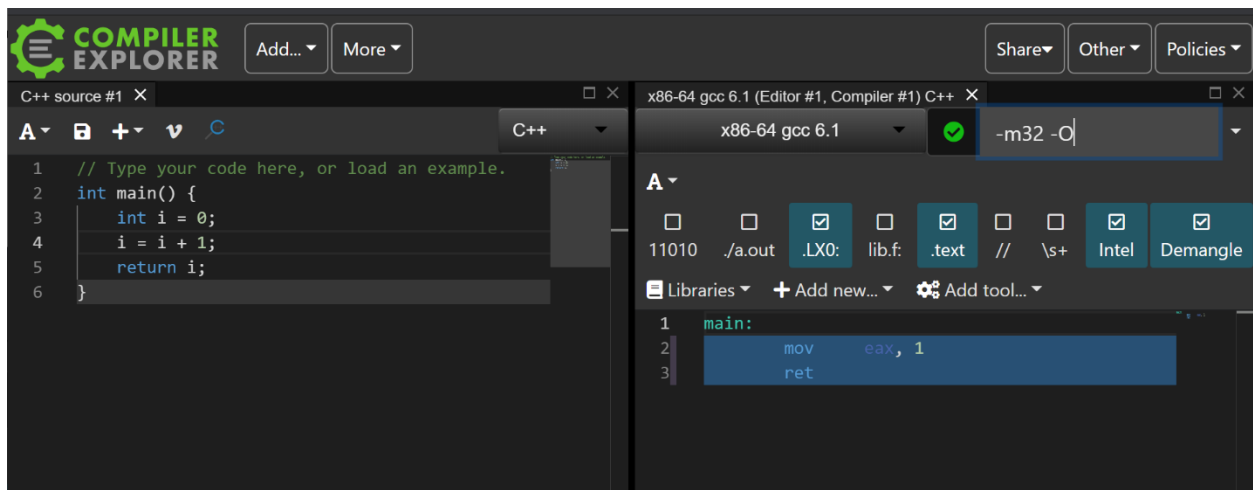
$i = i + 1$ unoptimized



The screenshot shows the Compiler Explorer interface with the C++ source code `int main() { int i = 0; i = i + 1; return i; }` on the left. The right pane shows the assembly output for x86-64 gcc 6.1 with the `-m32` flag. The assembly code is as follows:

```
1  main:
2      push    ebp
3      mov     ebp, esp
4      sub     esp, 16
5      mov     DWORD PTR [ebp-4], 0
6      add     DWORD PTR [ebp-4], 1
7      mov     eax, DWORD PTR [ebp-4]
8      leave
9      ret
```

$i = i + 1$ optimized



The screenshot shows the Compiler Explorer interface with the same C++ source code. The right pane shows the assembly output for x86-64 gcc 6.1 with the `-m32 -O` flag. The assembly code is significantly shorter:

```
1  main:
2      mov     eax, 1
3      ret
```

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

The screenshot shows the Compiler Explorer interface with the C++ source code in the left pane and the assembly output in the right pane. The source code is:

```
1 // Type your code here, or load an example.
2 int main() {
3     int i = 0;
4     i++;
5     return i;
6 }
```

The assembly output for the unoptimized version (gcc 6.1, -m32) is:

```
1 main:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     mov     DWORD PTR [ebp-4], 0
6     add     DWORD PTR [ebp-4], 1
7     mov     eax, DWORD PTR [ebp-4]
8     leave
9     ret
```

i++ optimized

The screenshot shows the Compiler Explorer interface with the same C++ source code in the left pane and the assembly output in the right pane. The source code is:

```
1 // Type your code here, or load an example.
2 int main() {
3     int i = 0;
4     i++;
5     return i;
6 }
```

The assembly output for the optimized version (gcc 6.1, -m32 -O) is:

```
1 main:
2     mov     eax, 1
3     ret
```

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:

The screenshot shows the Compiler Explorer interface with the following details:

- Source Code (C++ source #1):**

```

1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     int y = 1;
5     int z = (x < y) ? 2 : 3;
6     return z;
7 }

```
- Compiler (x86-64 gcc 6.1):**
 - Architecture: x86-64 gcc 6.1
 - Options: -m32
 - Libraries: .a.out, .LX0, lib.f, .text, //, \s+, Intel, Demangle
- Assembly Output (main:):**

```

1 main:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     mov     DWORD PTR [ebp-4], 0
6     mov     DWORD PTR [ebp-8], 1
7     mov     eax, DWORD PTR [ebp-4]
8     cmp     eax, DWORD PTR [ebp-8]
9     jge     .L2
10    mov     eax, 2
11    jmp     .L3
12 .L2:
13    mov     eax, 3
14 .L3:
15    mov     DWORD PTR [ebp-12], eax
16    mov     eax, DWORD PTR [ebp-12]
17    leave
18    ret

```

Ternary operator optimized:

The screenshot shows the Compiler Explorer interface with the following details:

- Source Code (C++ source #1):**

```

1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     int y = 1;
5     int z = (x < y) ? 2 : 3;
6     return z;
7 }

```
- Compiler (x86-64 gcc 6.1):**
 - Architecture: x86-64 gcc 6.1
 - Options: -m32 -O
 - Libraries: .a.out, .LX0, lib.f, .text, //, \s+, Intel, Demangle
- Assembly Output (main:):**

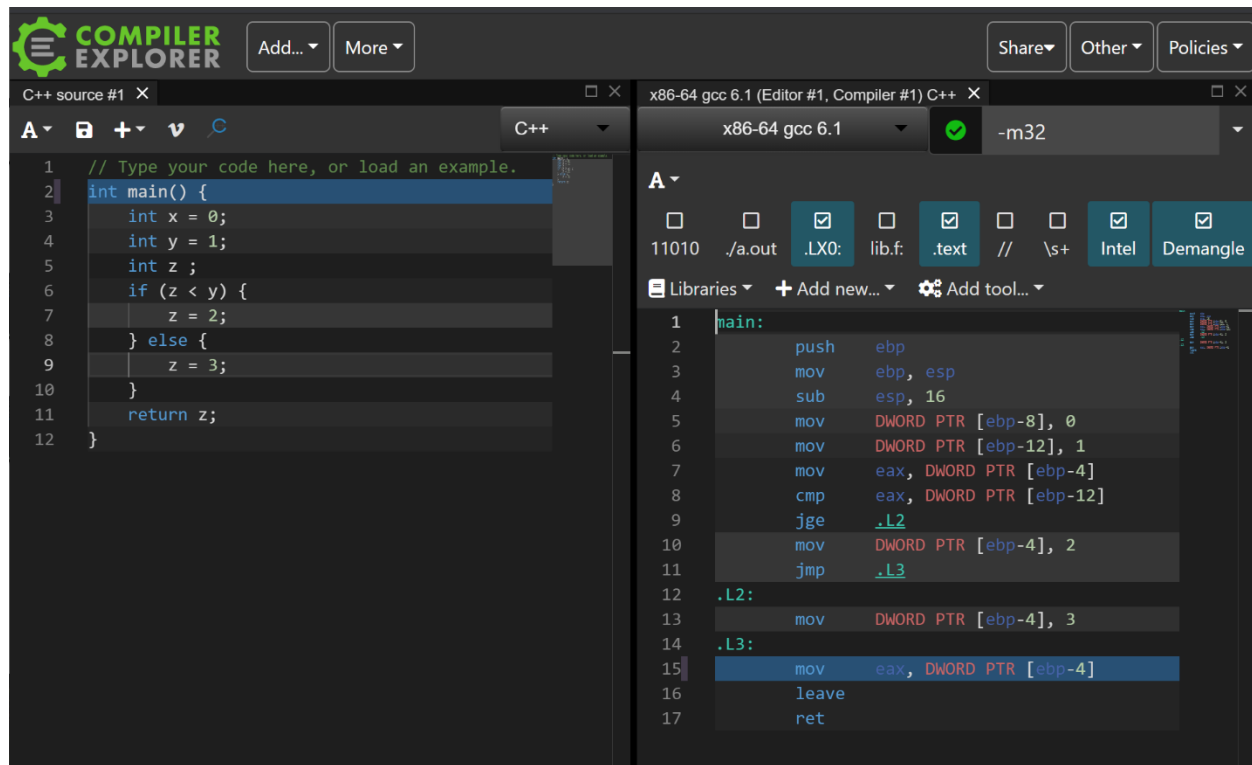
```

1 main:
2     mov     eax, 2
3     ret

```

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:



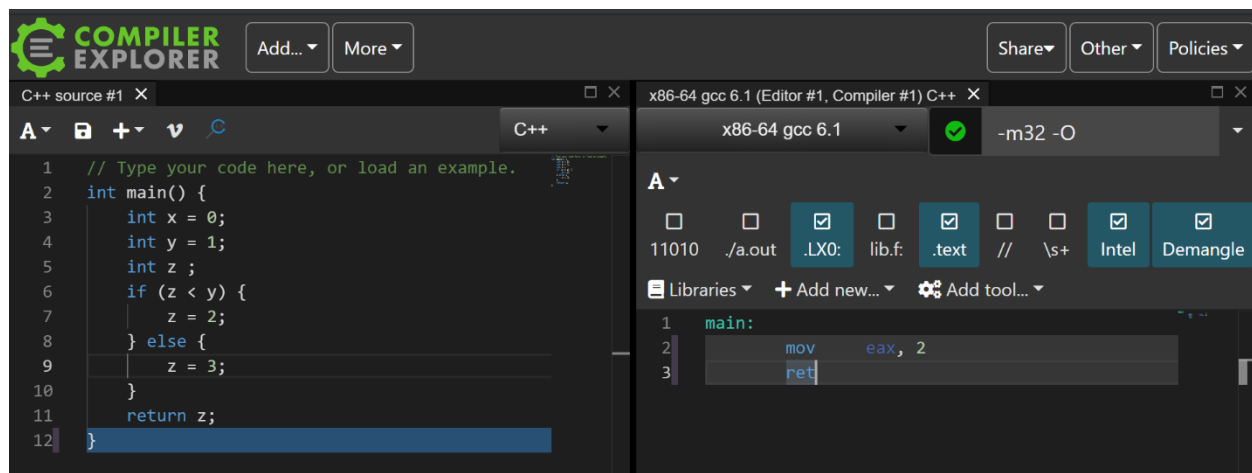
The screenshot shows the Compiler Explorer interface with the following details:

- Source Code (C++):**

```
1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     int y = 1;
5     int z;
6     if (z < y) {
7         z = 2;
8     } else {
9         z = 3;
10    }
11    return z;
12 }
```
- Compiler:** x86-64 gcc 6.1 (Editor #1, Compiler #1) C++
- Options:** x86-64 gcc 6.1, -m32
- Assembly Output:**

```
1 main:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     mov     DWORD PTR [ebp-8], 0
6     mov     DWORD PTR [ebp-12], 1
7     mov     eax, DWORD PTR [ebp-4]
8     cmp     eax, DWORD PTR [ebp-12]
9     jge     .L2
10    mov     DWORD PTR [ebp-4], 2
11    jmp     .L3
12 .L2:
13    mov     DWORD PTR [ebp-4], 3
14 .L3:
15    mov     eax, DWORD PTR [ebp-4]
16    leave
17    ret
```

If – else optimized:



The screenshot shows the Compiler Explorer interface with the following details:

- Source Code (C++):**

```
1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     int y = 1;
5     int z;
6     if (z < y) {
7         z = 2;
8     } else {
9         z = 3;
10    }
11    return z;
12 }
```
- Compiler:** x86-64 gcc 6.1 (Editor #1, Compiler #1) C++
- Options:** x86-64 gcc 6.1, -m32 -O
- Assembly Output:**

```
1 main:
2     mov     eax, 2
3     ret
```

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:

The screenshot shows the Compiler Explorer interface with the following details:

- Source Code (C++ source #1):**

```

1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     while (x < 2) {
5         x++;
6     }
7     return x;
8 }

```
- Compiler (x86-64 gcc 6.1):**
 - Architecture: x86-64 gcc 6.1
 - Options: -m32
 - Libraries: .a.out, .LX0, lib.f, .text, //, \s+, Intel, Demangle
- Assembly Output:**

```

1 main:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     mov     DWORD PTR [ebp-4], 0
6
7     .L3:
8     cmp     DWORD PTR [ebp-4], 1
9     jg      .L2
10    add     DWORD PTR [ebp-4], 1
11    jmp     .L3
12
13    .L2:
14    mov     eax, DWORD PTR [ebp-4]
15    leave
16    ret

```

While optimized:

The screenshot shows the Compiler Explorer interface with the following details:

- Source Code (C++ source #1):**

```

1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     while (x < 2) {
5         x++;
6     }
7     return x;
8 }

```
- Compiler (x86-64 gcc 6.1):**
 - Architecture: x86-64 gcc 6.1
 - Options: -m32 -O
 - Libraries: .a.out, .LX0, lib.f, .text, //, \s+, Intel, Demangle
- Assembly Output:**

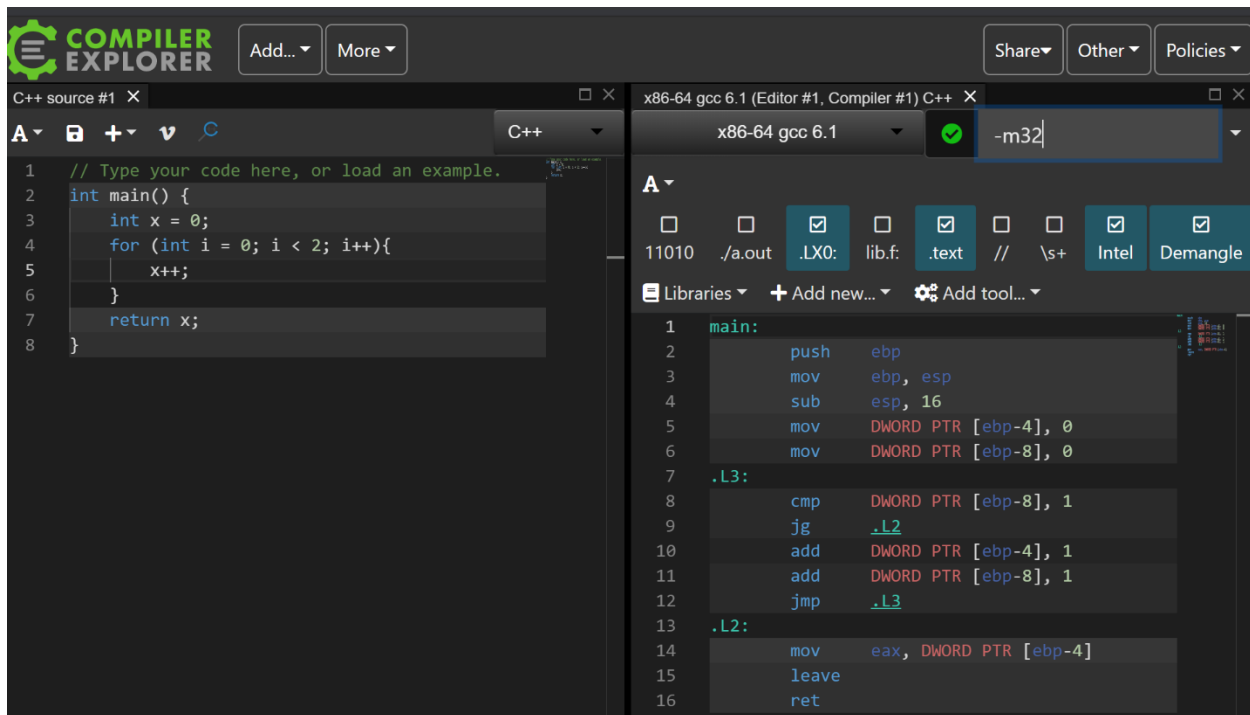
```

1 main:
2     mov     eax, 2
3     ret

```

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:



The screenshot shows the Compiler Explorer interface. The source code is a C++ program that initializes a variable `x` to 0 and enters a `for` loop that increments `x` twice. The compiler is set to `x86-64 gcc 6.1` with the `-m32` flag. The assembly output shows the generated machine code for the unoptimized version, which includes stack frame setup, a loop body with increments, and a return statement.

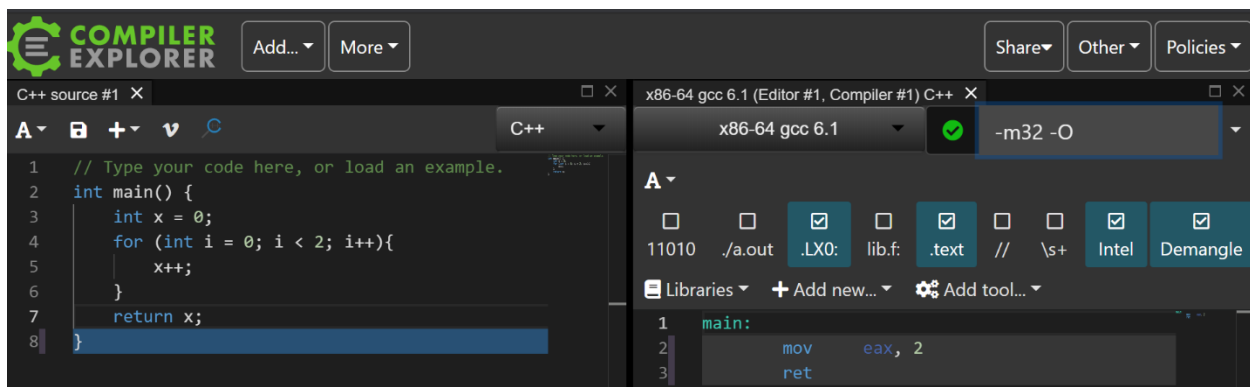
```
1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     for (int i = 0; i < 2; i++){
5         x++;
6     }
7     return x;
8 }
```

Compiler: x86-64 gcc 6.1 (Editor #1, Compiler #1) C++ X
Flags: -m32

Assembly output:

```
1 main:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     mov     DWORD PTR [ebp-4], 0
6     mov     DWORD PTR [ebp-8], 0
7 .L3:
8     cmp     DWORD PTR [ebp-8], 1
9     jg      .L2
10    add     DWORD PTR [ebp-4], 1
11    add     DWORD PTR [ebp-8], 1
12    jmp     .L3
13 .L2:
14    mov     eax, DWORD PTR [ebp-4]
15    leave
16    ret
```

For optimized:



The screenshot shows the Compiler Explorer interface with the same C++ source code as the previous example. The compiler is set to `x86-64 gcc 6.1` with the `-m32 -O` flag. The assembly output shows the optimized version, which is significantly simpler, consisting of a single `mov` instruction to set `eax` to 2, followed by a `ret` instruction.

```
1 // Type your code here, or load an example.
2 int main() {
3     int x = 0;
4     for (int i = 0; i < 2; i++){
5         x++;
6     }
7     return x;
8 }
```

Compiler: x86-64 gcc 6.1 (Editor #1, Compiler #1) C++ X
Flags: -m32 -O

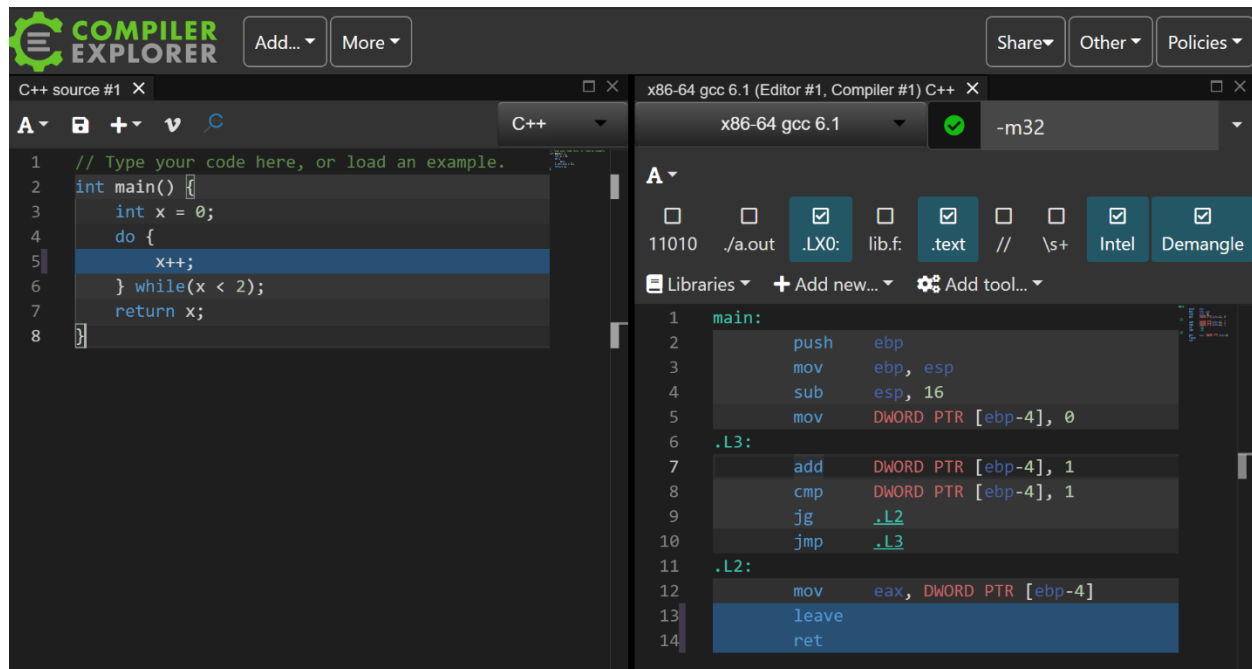
Assembly output:

```
1 main:
2     mov     eax, 2
3     ret
```

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:

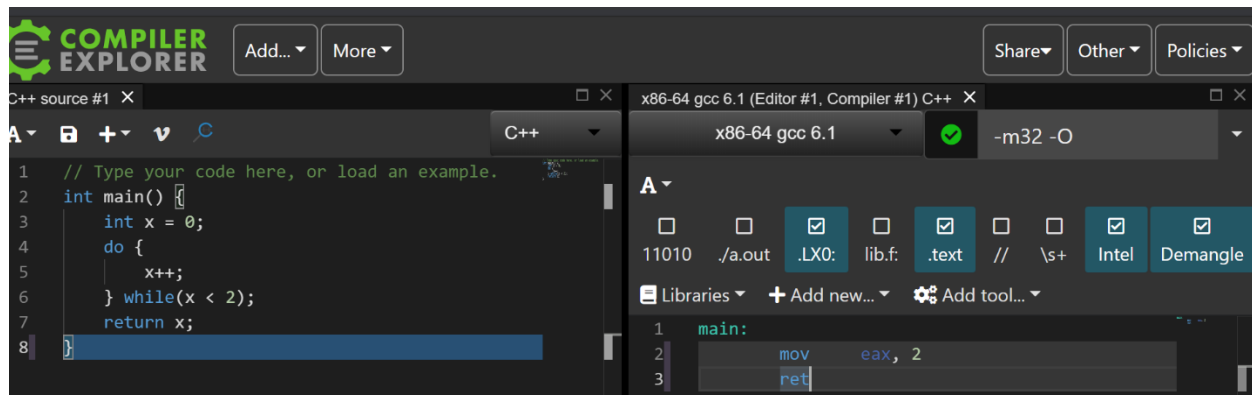


The screenshot shows the Compiler Explorer interface with the following details:

- Language:** C++
- Compiler:** x86-64 gcc 6.1
- Options:** -m32
- Assembly Output:**

```
1 main:
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 16
5     mov     DWORD PTR [ebp-4], 0
6
7     .L3:
8         add     DWORD PTR [ebp-4], 1
9         cmp     DWORD PTR [ebp-4], 1
10        jg      .L2
11        jmp     .L3
12
13     .L2:
14        mov     eax, DWORD PTR [ebp-4]
15        leave
16        ret
```

Do-while optimized:



The screenshot shows the Compiler Explorer interface with the following details:

- Language:** C++
- Compiler:** x86-64 gcc 6.1
- Options:** -m32 -O
- Assembly Output:**

```
1 main:
2     mov     eax, 2
3     ret
```

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 of 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 $\text{arg2} * 9000$ in arg1 which is 81000. It then sets arg2 equal to $9000 - \text{arg1}$ which is -72000. The program then returns $\text{arg1} + \text{arg2}$ 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 arg2 = atoi(argv[2]);

    Z1fPiS(arg1, arg2);

    return arg1 + arg2;
}
```