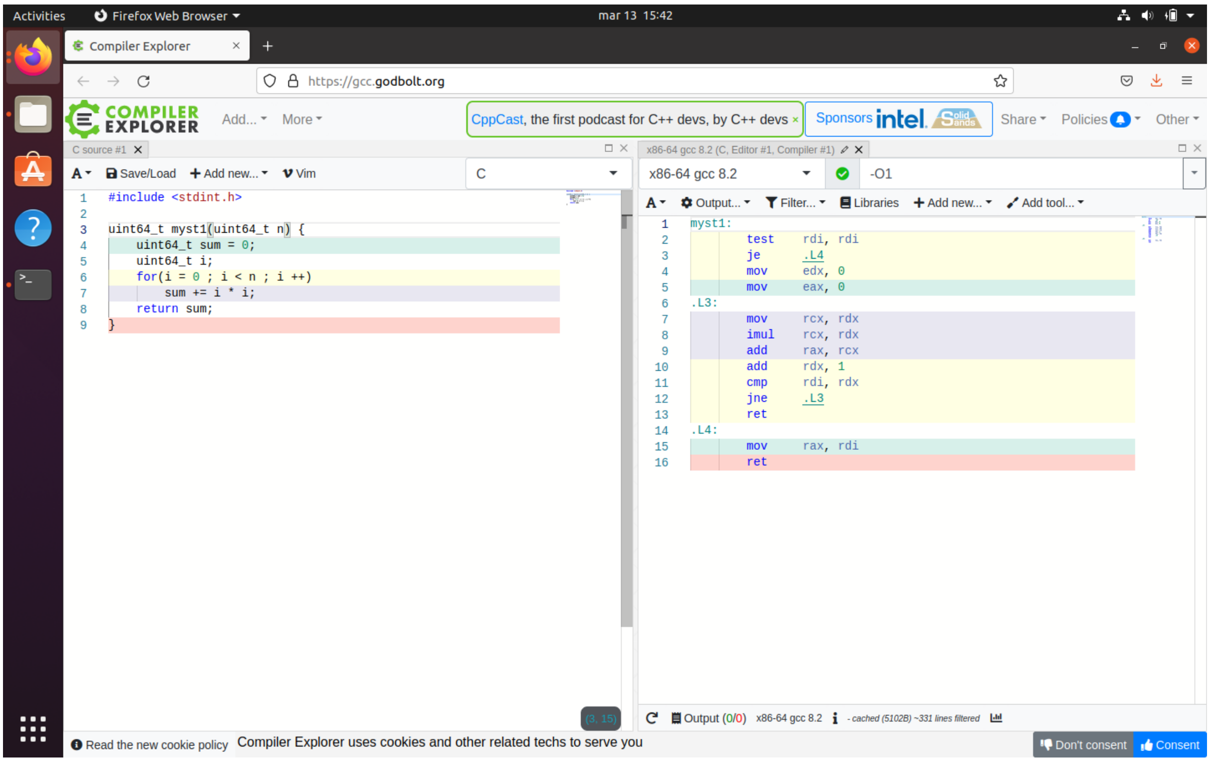
Task 1: Assembly analysis

- **a) Write the equivalent in C for this ASM snippet (1p)**

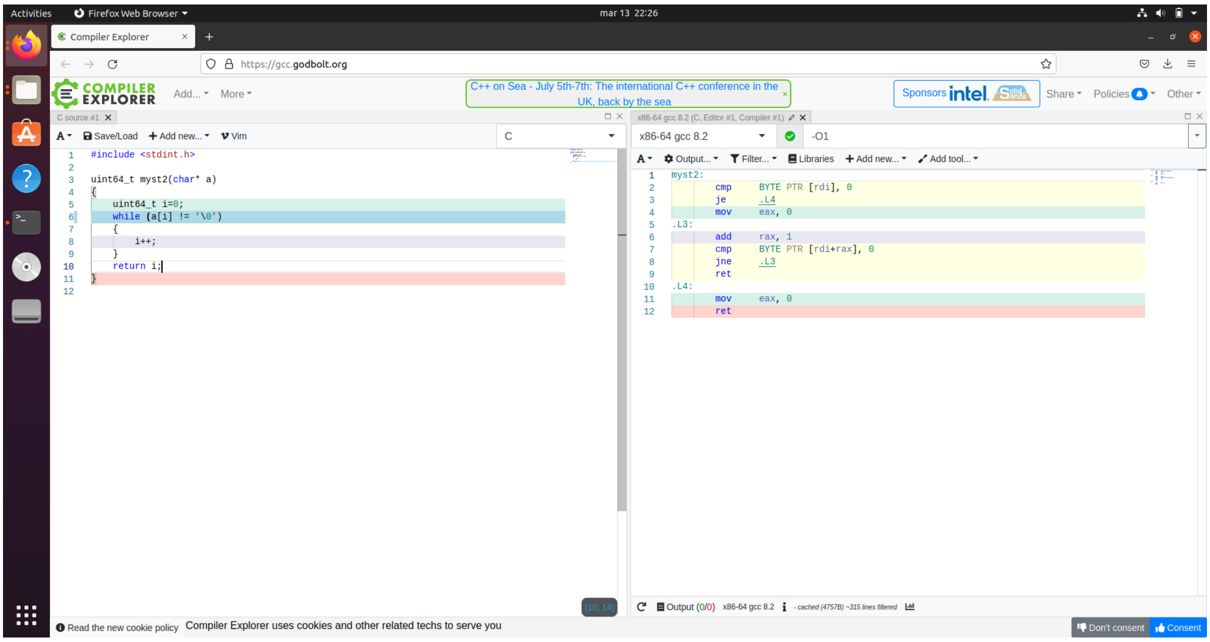
The code can be found in task11.c.
As it can be seen in the image below, the code translates into the desired assembly code.



An appropriate name for the function may be “sum_first_n_squares”.

- ## b) Write the equivalent in C for this ASM snippet (1p)

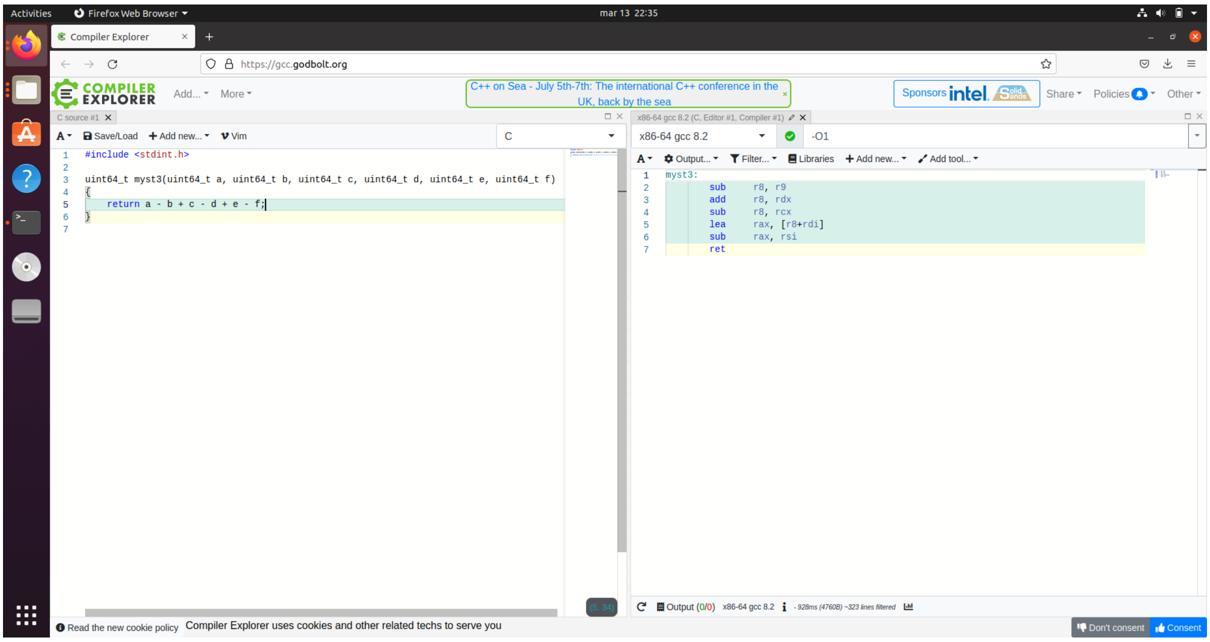
The code can be found in task12.c.
As it can be seen in the image below, the code translates into the desired assembly code.



An appropriate name for the function may be “length_string” (or “strlen”, but is already taken).

- ## c) Write the equivalent in C for this ASM snippet (1p)

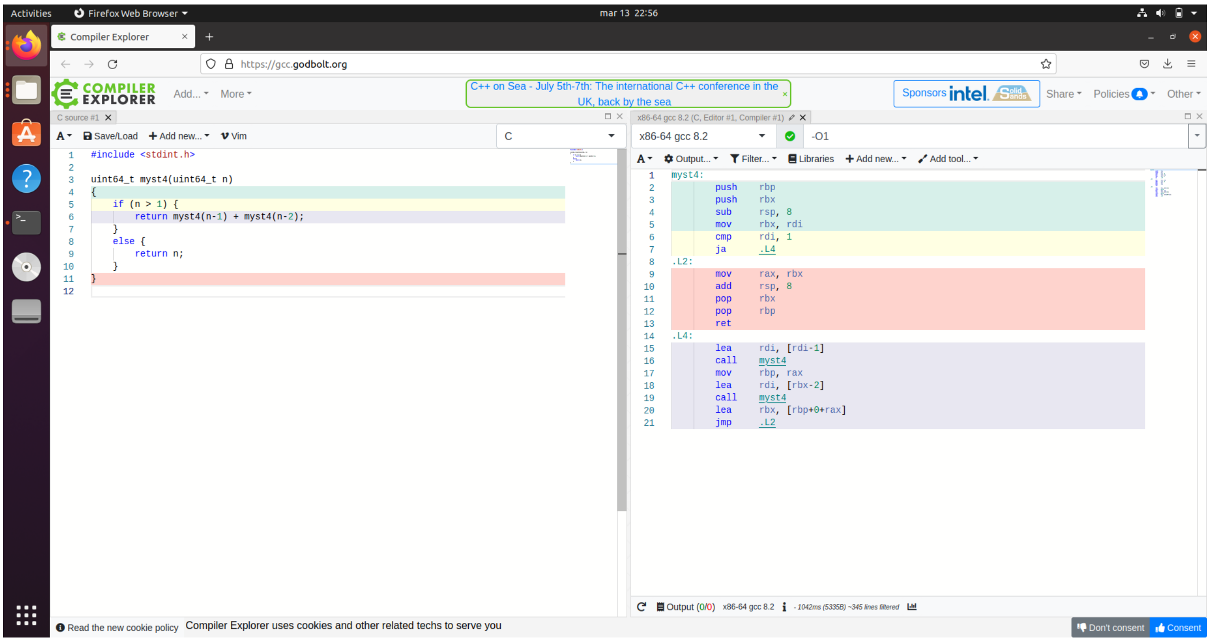
The code can be found in task13.c.
As it can be seen in the image below, the code translates into the desired assembly code.



An appropriate name for the function may be “alternate_sum6”.

- **d) Write the equivalent in C for this ASM snippet (2p)**

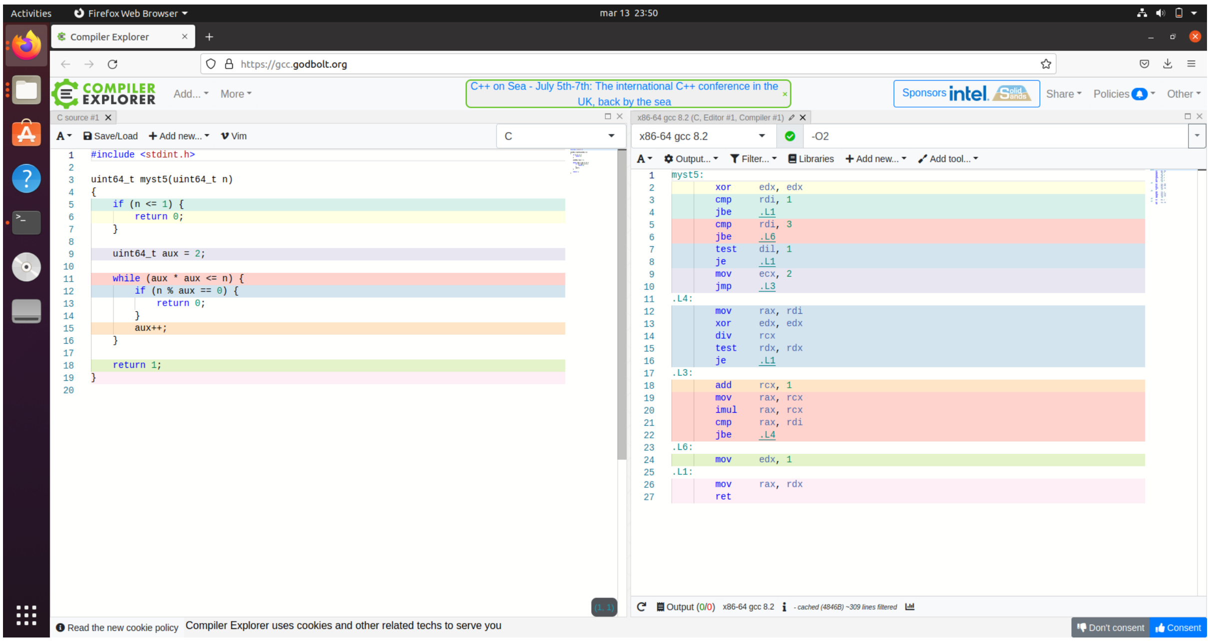
The code can be found in task14.c.
As it can be seen in the image below, the code translates into the desired assembly code.



An appropriate name for the function may be “fibonacci” (as it computes the n-th fibonacci term).

- **e) Write the equivalent in C for this ASM snippet (3p)**

The code can be found in task15.c.
As it can be seen in the image below, the translated assembly code resembles the desired assembly code (using O2 for compilation).

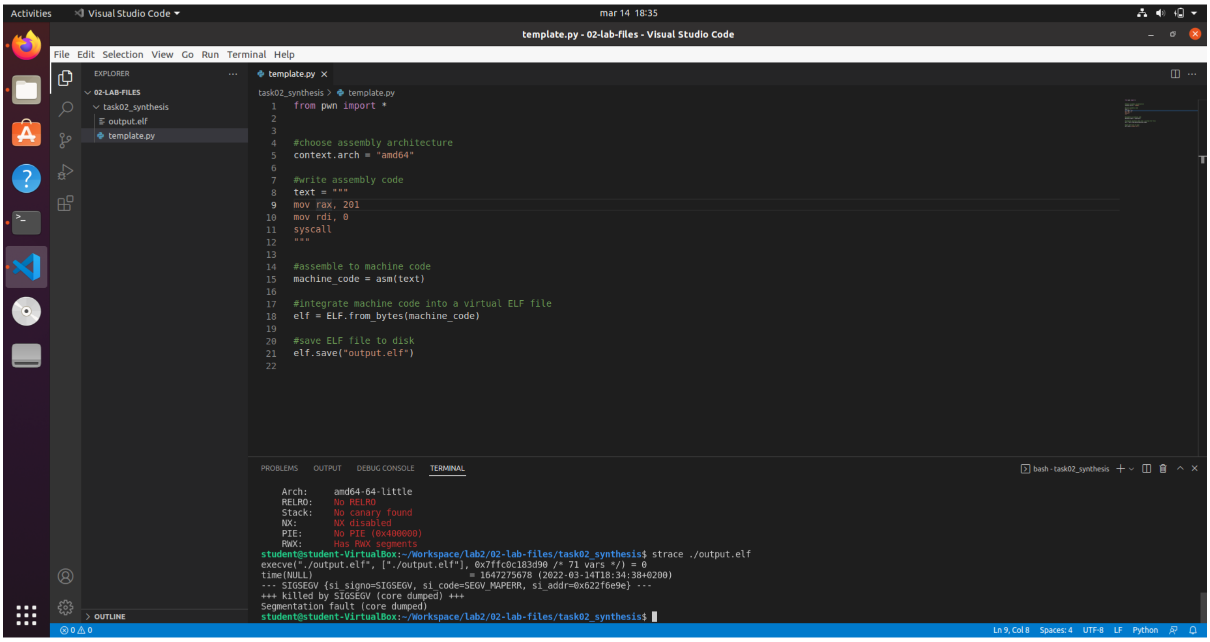


An appropriate name for the function may be “test_prime” (as it tests if its argument is a prime number or not). As a short explanation of what the assembly code does, I will explain what the code under each label does: L6 (return 1 -> the number received is prime), L8 (return 0 -> the number received is not prime), L1 (return current value from rax - the return value register), L4 (checks if the current counter divides the received number), L3 (checks if the current counter is smaller than the square root of the received number - we don’t look for divisors after that point), myst5 (handles specific edge-cases: 0, 1, 2, 3, even numbers).

Task 2: Assembly synthesis

- **Linux syscalls: get the time (3p)**

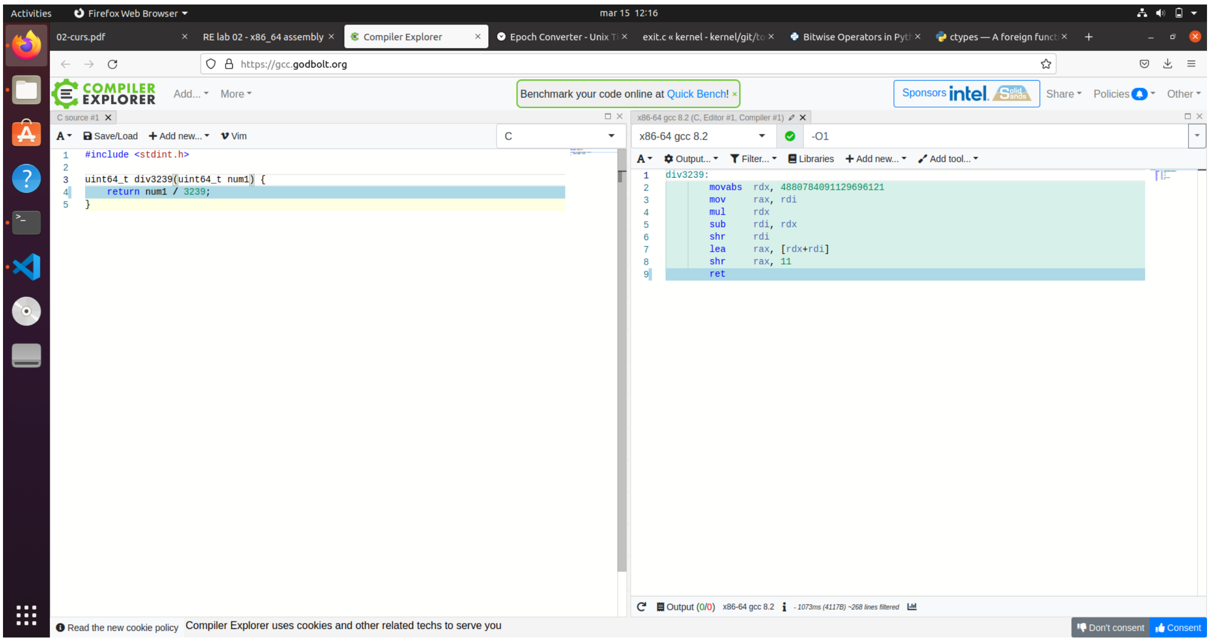
The script that contains the assembly code and that generates the ELF file can be found in task2.py. The generated ELF file is called task2.elf. As it can be seen in the following image, when we wrap the ELF file in strace, we get the correct timestamp when we make the syscall.



Task 3: Compiler magic tricks

- what the initial code was **(3p)**

I translated the assembly code into a Python implementation which can be found in task3.py. After playing with the script, feeding it different inputs, I find out that the program computes the division by 3239. I used the online compiler to confirm that, as seen in the following screenshot.



- how it works **(2p)**

To understand the optimization, I read the info from this [link](). I will explain the key points of this idea:

1. First we write the division N/D as (2^K)/D * N/(2^K), for some K.
2. We use an estimation for (2^K)/D such that we don't affect the computations (we can do that because we want to compute the quotient, not the precise division result). That estimation (also called "magic number"), denoted from now on as M, will be the ceiling of (2^K)/D, and we also have to choose K=64+ceil(log(2,D)) (64 can be replaced with the register size).
3. We now have to compute M*N/(2^K), but we can't compute M*N not even with results on 128-bits, because M is a 65-bit number. For that reason we will treat M as a 2 part number: the first part contains only one bit (2^64) and the second part contains the rest of the bits (M-2^64). So, M*N = (2^64 + M-(2^64))*N = (2^64)*N + (M-(2^64))*N.
4. To further work on 64-bit values we introduce 2^64 from 2^K into the expression. Thus, we obtain: M*N/(2^K) = (M*N/(2^64)) / (2^ceil(log(2,D))) = (N + (M-(2^64))*N/(2^64)) / (2^ceil(log(2,D))). For simplicity, we will note with Q the expression (M-(2^64))*N/(2^64) (the first 64-bits of the multiplication between the argument and the magic number).
5. The issue now is that the addition from the numerator can overflow (there are 2 64-bits registers). Since ceil(log(2,D)) is bigger than 1, we can use another 2 from the denominator and use the following equality: (A+B)/2 = (A-B)/2 + B. Applying this to

our state of the computation we obtain: M*N/(2^K) = ((N - Q) / 2 + Q) / (2^floor(log(2,D))).

6. Now, in our case, we have M-2^64 = 4880784091129696121, K = 64+1+11 = 76. The first 2 lines of code put the arguments M and N (the argument) in the registers used by the multiplication instruction. The `mul` instruction computes Q. The following 3 lines compute the nominator from the last expression described at point 5. And the last operation, before the return, computes the final division.