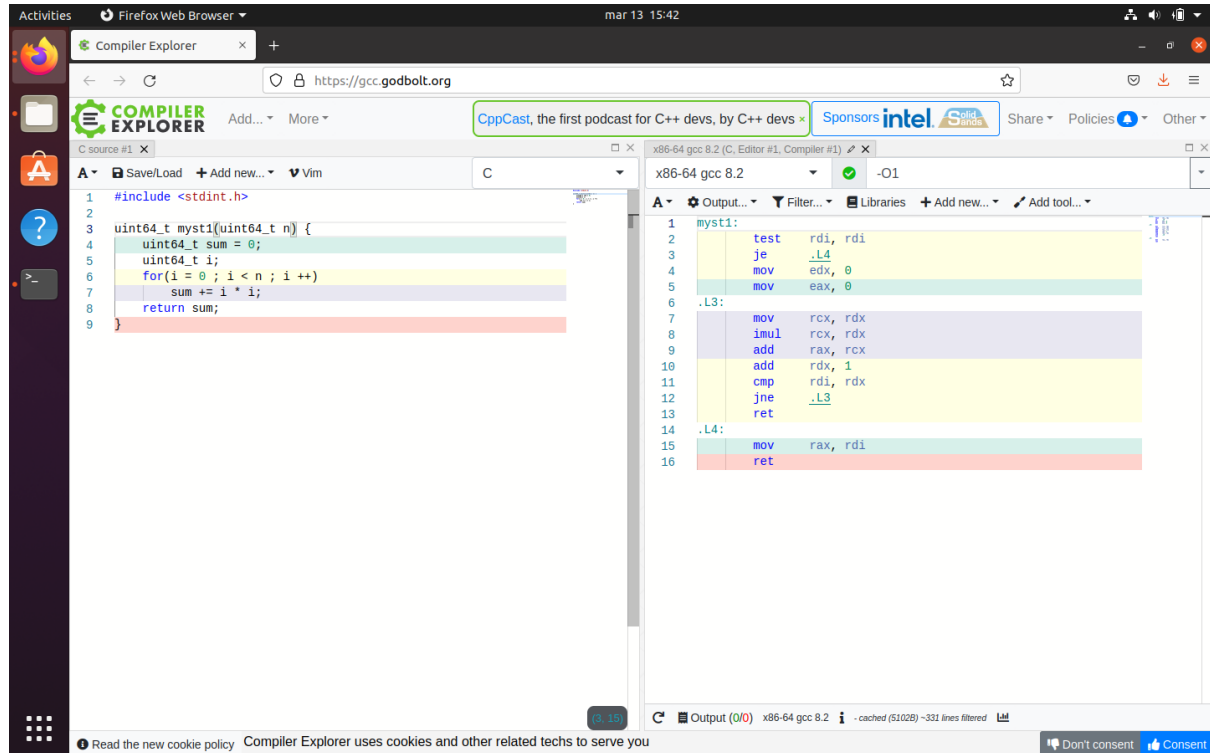


Task 1: Assembly analysis

- 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.



The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed in a file named 'task11.c':

```
1 #include <stdint.h>
2
3 uint64_t myst1(uint64_t n) {
4     uint64_t sum = 0;
5     uint64_t i;
6     for(i = 0; i < n; i++)
7         sum += i * i;
8     return sum;
9 }
```

On the right, the assembly output for 'x86-64 gcc 8.2' is shown, with optimization level '-O1'. The assembly code is as follows:

```
1 myst1:
2     test    rdi, rdi
3     je      .L4
4     mov     edx, 0
5     mov     eax, 0
6
7 .L3:
8     mov     rcx, rdx
9     imul    rcx, rdx
10    add     rax, rcx
11    add     rdx, 1
12    cmp     rdi, rdx
13    jne     .L3
14    ret
15
16 .L4:
17    mov     rax, rdi
18    ret
```

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.

The screenshot shows the Compiler Explorer interface with the following C code in the left pane:

```

1 #include <stdint.h>
2
3 uint64_t myst2(char* a)
4 {
5     uint64_t i=0;
6     while (a[i] != '\0')
7     {
8         i++;
9     }
10    return i;
11 }
12

```

The right pane shows the generated assembly code for x86-64 gcc 8.2 at -O1 optimization level:

```

1 myst2:
2     cmp     BYTE PTR [rdi], 0
3     je      .L4
4     mov     rax, 0
5     .L3:
6     add     rax, 1
7     cmp     BYTE PTR [rdi+rax], 0
8     jne     .L3
9     .L4:
10    mov     rax, 0
11    ret
12

```

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.

The screenshot shows the Compiler Explorer interface with the following C code in the left pane:

```

1 #include <stdint.h>
2
3 uint64_t myst3(uint64_t a, uint64_t b, uint64_t c, uint64_t d, uint64_t e, uint64_t f)
4 {
5     return a - b + c - d + e - f;
6 }
7

```

The right pane shows the generated assembly code for x86-64 gcc 8.2 at -O1 optimization level:

```

1 myst3:
2     sub     r8, r9
3     add     r8, rdx
4     sub     r8, rcx
5     lea     rax, [r8+rdi]
6     sub     rax, rsi
7     ret

```

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.

The screenshot shows the Compiler Explorer interface with the following C code on the left:

```

1 #include <stdint.h>
2
3 uint64_t myst4(uint64_t n)
4 {
5     if (n > 1) {
6         return myst4(n-1) + myst4(n-2);
7     }
8     else {
9         return n;
10    }
11 }
12

```

The assembly output on the right (compiled with x86-64 gcc 8.2, -O1) is:

```

1 myst4:
2     push    rbp
3     push    rbx
4     sub     rsp, 8
5     mov     rdx, rdi
6     cmp     rdi, 1
7     ja      .L2
8     .L2:
9     mov     rax, rbx
10    add     rsp, 8
11    pop     rbx
12    pop     rbp
13    ret
14
15 .L4:
16    lea     rdi, [rdi-1]
17    call    myst4
18    mov     rbp, rax
19    lea     rdi, [rbx-2]
20    call    myst4
21    lea     rax, [rbp+8+rax]
22    jmp     .L2

```

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).

The screenshot shows the Compiler Explorer interface with the following C code on the left:

```

1 #include <stdint.h>
2
3 uint64_t myst5(uint64_t n)
4 {
5     if (n <= 1) {
6         return 0;
7     }
8
9     uint64_t aux = 2;
10
11    while (aux * aux <= n) {
12        if (n % aux == 0) {
13            return 0;
14        }
15        aux++;
16    }
17
18    return 1;
19 }
20

```

The assembly output on the right (compiled with x86-64 gcc 8.2, -O2) is:

```

1 myst5:
2     xor     edx, edx
3     cmp     rdi, 1
4     jbe     .L1
5     cmp     rdi, 3
6     jbe     .L6
7     test    rdi, 1
8     je      .L1
9     mov     ecx, 2
10    jmp     .L3
11
12 .L4:
13    mov     rax, rdi
14    xor     edx, edx
15    div     rcx
16    test    rdx, rdx
17    je      .L3
18
19 .L3:
20    add     rcx, 1
21    mov     rax, rcx
22    imul    rax, rcx
23    cmp     rax, rdi
24    jbe     .L4
25
26 .L6:
27    mov     edx, 1
28
29 .L1:
30    mov     rax, rdx
31    ret

```

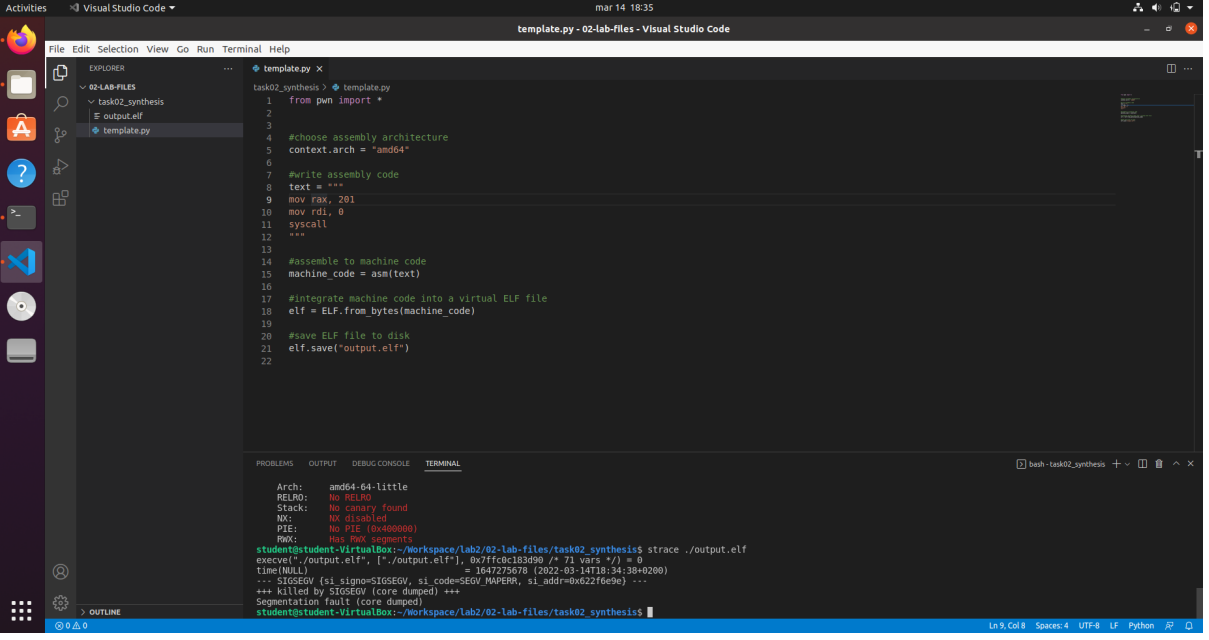
An appropriate name for the function may be “test_prime” (as it tests if its argument is a prime number or not). As a short explanation of what the assembly code does, I will explain what the code under each label does: L6 (return 1 -> the number received is prime), L8

(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.



The screenshot shows the Visual Studio Code interface with the file explorer on the left displaying the project structure: 02-LAB-FILES, task02_synthesis, output.elf, and template.py. The main editor shows the task2.py script, which is a Python script that generates an ELF file from assembly code. The script includes comments for each step: choosing architecture, writing assembly code, assembling to machine code, and integrating into a virtual ELF file. The terminal at the bottom shows the execution of the script and the use of strace to trace the system calls, including a successful call to gettimeofday.

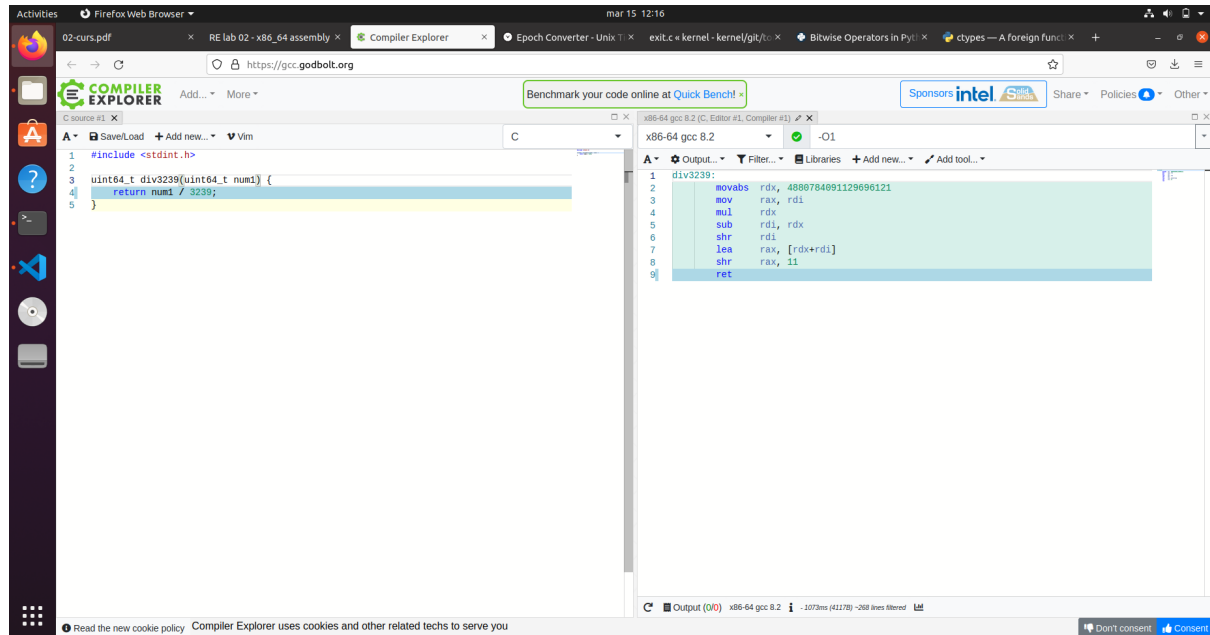
```
task02_synthesis > template.py
1 from pwn import *
2
3
4 #choose assembly architecture
5 context.arch = "amd64"
6
7 #write assembly code
8 text = """
9 mov rax, 201
10 mov rdi, 0
11 syscall
12 """
13
14 #assemble to machine code
15 machine_code = asm(text)
16
17 #integrate machine code into a virtual ELF file
18 elf = ELF.from_bytes(machine_code)
19
20 #save ELF file to disk
21 elf.save("output.elf")
22
```

```
Arch: amd64-64-little
RELRO: No RELRO
Stack: No canary found
NX: NX disabled
PIE: No PIE (0x400000)
Rwx: Has Rwx segments
student@student-VirtualBox:~/Workspace/Lab2/02-Lab-files/task02_synthesis$ strace ./output.elf
execve("./output.elf", ["/output.elf"], 0x7ffc163d90 /* ? */ var=?) = 0
time(NULL) = 1647275678 (2022-03-14T18:34:38+0200)
--- SIGSEGV (si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x02276e9e) ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
student@student-VirtualBox:~/Workspace/Lab2/02-Lab-files/task02_synthesis$
```

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 $K=64+\text{ceil}(\log(2,D))$ (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: $M*N/(2^K) = (M*N/(2^{64})) / (2^{\text{ceil}(\log(2,D))}) = (N + (M-(2^{64}))*N/(2^{64})) / (2^{\text{ceil}(\log(2,D))})$. 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 $\text{ceil}(\log(2,D))$ 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: $M \cdot N / (2^K) = ((N - Q) / 2 + Q) / (2^{\text{floor}(\log(2,D))})$.

6. Now, in our case, we have $M \cdot 2^{64} = 4880784091129696121$, $K = 64 + 1 + 11 = 76$. 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.