Assignment 11

Task 1

Snippet a

`c1` is of type `unsigned`

```
unsigned c2 = 32 * c1;
```

If we assume that the multiplication takes longer then a bit shift, we can optimize this code as follows:

```
unsigned c2 = c1 << 5;
```

If we look at the compiled assembly code, we see no difference between those two snippets when compiled with `-O3`.

Snippet b

`c1` is of type `unsigned`

```
unsigned c2 = 15 * c1;
```

Assuming bit shifting + one subtraction are faster then a multiplication, one could do this:

```
unsigned c2 = (c1 << 4) - c1;
```

Once again the corresponding assembly codes are identical.

Snippet c

`c1` is of type `unsigned`

```
unsigned c2 = 96 * c1;
```

Assuming that 2 shift operations and one addition are still faster then a single multiplication, the following optimization could be helpful:

```
unsigned c2 = (c1 << 5) + (c1 << 6);
```

The compiler **disagrees** with this optimization. The 'optimized' snippet gets replaced by the following code:

```
c_solution:
        imul    esi, edi, 96
        xor     eax, eax
        mov     edi, OFFSET FLAT:.LC2
        jmp     printf
```

which basically performs a multiplication by 96, which means that the compiler thinks that the unoptimized version is faster then the optimized one.
Additionally, if we compile the unoptimized version, the compiler does the following:

```
c:
        lea     esi, [rdi+rdi*2]
        xor     eax, eax
        mov     edi, OFFSET FLAT:.LC2
        sal     esi, 5
        jmp     printf
```

Since we are not quite tallented with x86 assembly code, we cannot explain this snippet.

Snippet d

`c1` is of type `unsigned`

```
unsigned c2 = 0.125 * c1;
```

Since the multiplication of 0.125 is equivalent to the division by 8, we can optimize this snippet as follows:

```
unsigned c2 = c1 >> 3;
```

The compiler translates this to the expected assembly code:

```
d_solution:
        mov     esi, edi
        xor     eax, eax
        mov     edi, OFFSET FLAT:.LC4
        shr     esi, 3
        jmp     printf
```

But if we look at the compiled unoptimized version, the compiler thinks this magic looking code is faster then a normal multiplication:

```
d:
        mov     edi, edi
        pxor    xmm0, xmm0
        xor     eax, eax
        cvtsi2sdq       xmm0, rdi
        mulsd   xmm0, QWORD PTR .LC3[rip]
        mov     edi, OFFSET FLAT:.LC4
        cvttsd2si       rsi, xmm0
        jmp     printf
```

Snippet e

`a` is of type `unsigned *`

```
unsigned sum_fifth = 0;
for (int i = 0; i < N / 5; ++i) {
    sum_fifth += a[5 * i];
}
```

If we want to get rid of the expensive multiplication inside the loop, we can simply transform the loop head like this:

```
unsigned sum_fifth = 0;
for (int i = 0; i < N; i+=5) {
    sum_fifth += a[i];
}
```

The compiled versions of the unoptimized and optimized snippet look identical.

Snippet f

`a` is of type `double *`

```
for (int i = 0; i < N; ++i) {
    a[i] += i / 5.3;
}
```

In this case, we want to get rid of the division with the float number `5,3` to optimize the code snippet. To do that, we could hard code the value of the constant `1/5,3` and convert it to a multiplication. But a multiplication in a for loop can be converted in to additions which is more efficient. Although `1/5,3` is an irrational number, if we implement this constant with enough decimal points, the accuracy might not be affected by these changes.

```
float h = 0.0;
for (int i = 0; i < N; ++i) {
    a[i] += h;
    h = h + 0.18867924528301886792452830188679;
}
```

The unoptimized version compiles to this:

```
        movdqa  xmm2, XMMWORD PTR .LC6[rip]
        movdqa  xmm4, XMMWORD PTR .LC7[rip]
        lea     rax, [rdi+8000]
        movapd  xmm3, XMMWORD PTR .LC8[rip]
.L17:
        pshufd  xmm0, xmm2, 238
        cvtdq2pd        xmm1, xmm2
        movupd  xmm6, XMMWORD PTR [rdi]
        add     rdi, 32
        cvtdq2pd        xmm0, xmm0
        divpd   xmm1, xmm3
        movupd  xmm5, XMMWORD PTR [rdi-16]
```

```
        paddd   xmm2, xmm4
        divpd   xmm0, xmm3
        addpd   xmm1, xmm6
        movups  XMMWORD PTR [rdi-32], xmm1
        addpd   xmm0, xmm5
        movups  XMMWORD PTR [rdi-16], xmm0
        cmp     rax, rdi
        jne     .L17
        ret
```

While the optimized version this assembly code causes:

```
        movsd   xmm2, QWORD PTR .LC10[rip]
        lea     rax, [rdi+8000]
        pxor    xmm0, xmm0
.L20:
        movsd   xmm1, QWORD PTR [rdi]
        cvtss2sd        xmm0, xmm0
        add     rdi, 8
        addsd   xmm1, xmm0
        addsd   xmm0, xmm2
        movsd   QWORD PTR [rdi-8], xmm1
        cvtsd2ss        xmm0, xmm0
        cmp     rax, rdi
        jne     .L20
        ret
```

Snippet g

`c1` is of type `float`

```
float c2 = -1 * c1;
```

To just swap the sign, the multiplication of -1 is not the best way. Better is to just put `-` before the variable you want the negation of because then a `sign-bit flip` ist executed. So most significant bit, which equals to the sign bit (IEEE 754 single-precision binary floating-point format). So in fact, only a XOR-operation is executed to negate a number.

```
float c2 = -c1;
```

The compiler does this by default so both result in the same assembly code:

```
xor     eax, eax
        ret
```