To INT Or To UINT

About Me:

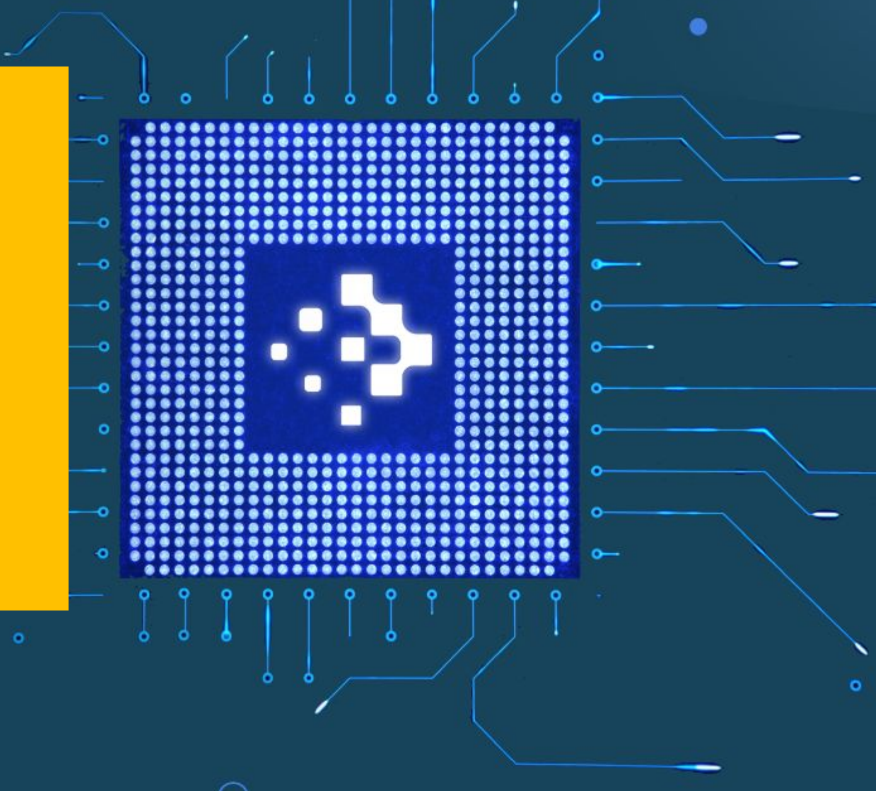
SPEEDATA

alex.dathskovsky@speedata.io

www.linkedin.com/in/alexdathskovsky

www.cppnext.com

https://www.youtube.com/@cppnext-alexd



To INT Or To UINT

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

> *"There are far too many integer types, there are far too lenient rules for mixing them together, and it's a major bug source, which is why I'm saying stay as simple as you can, use {signed} integers till you really need something else."*
>
> ~Bjarne Stroustrup

https://graphitemaster.github.io/aau/

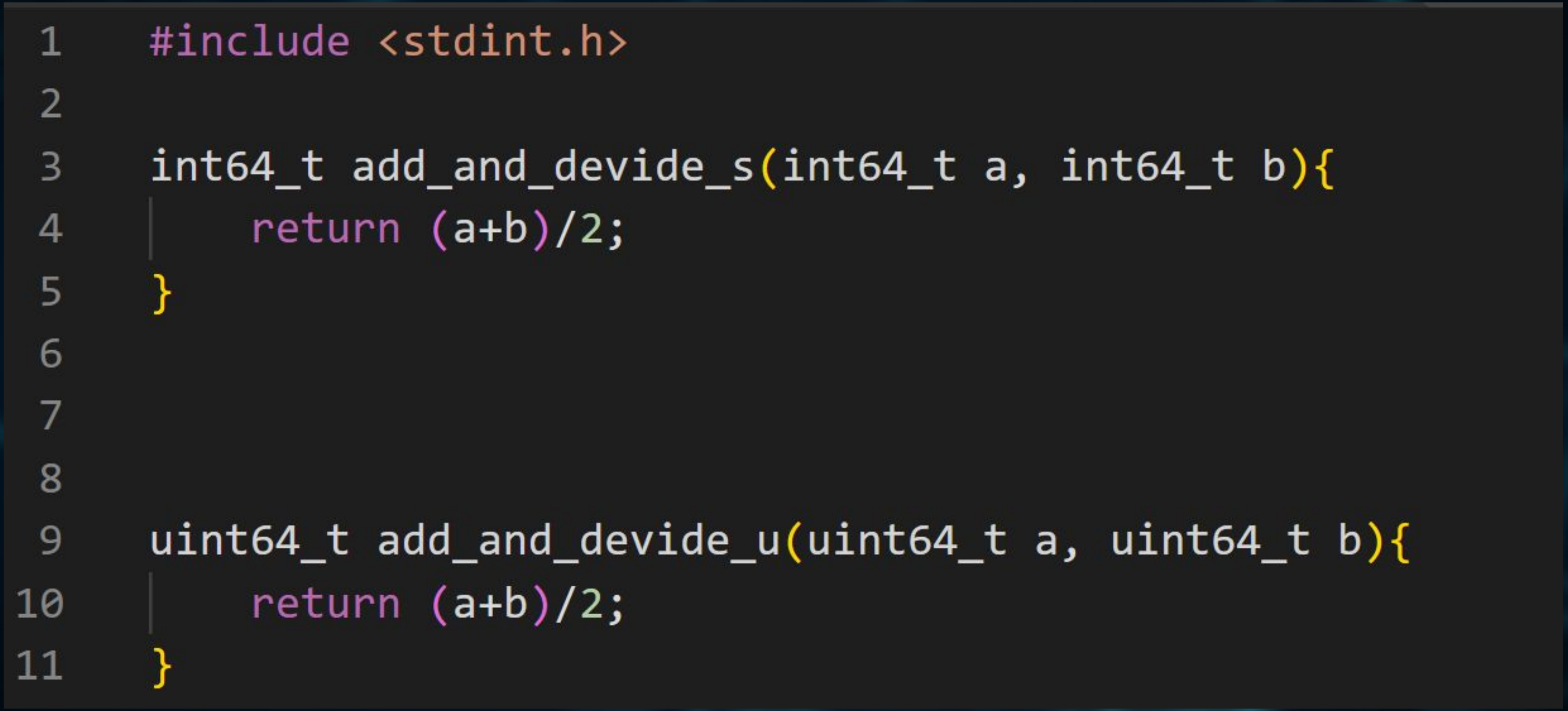
“The need for signed integer arithmetic is often misplaced as most integers never represent negative values within a program. The indexing of arrays and iteration count of a loop reflects this concept as well. There should be a propensity to use unsigned integers more often than signed, yet despite this, most coders incorrectly choses to use signed integers almost exclusively.”

~ Dale Weiler

https://graphitemaster.github.io/aau/

Disclaimer : X86 machines only in this talk

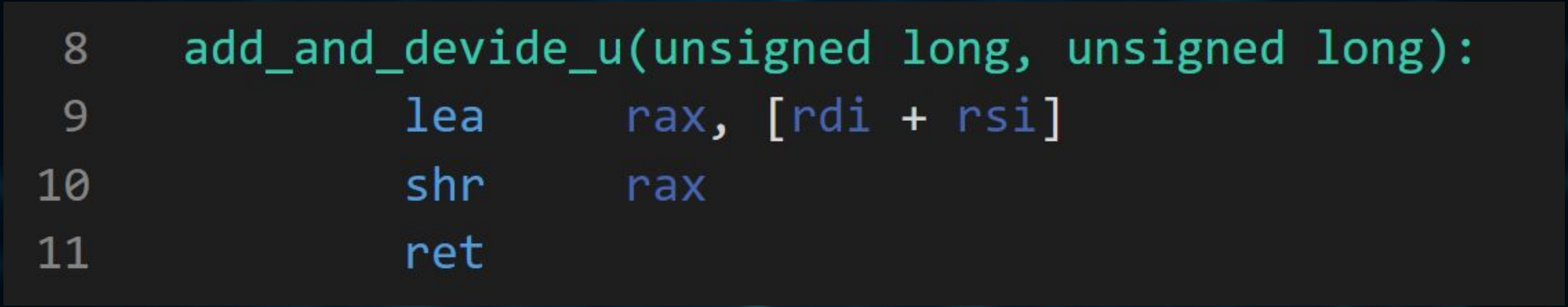
SIMPLE EXAMPLE:



```c
#include <stdint.h>

int64_t add_and_devide_s(int64_t a, int64_t b){
    return (a+b)/2;
}



uint64_t add_and_devide_u(uint64_t a, uint64_t b){
    return (a+b)/2;
}
```

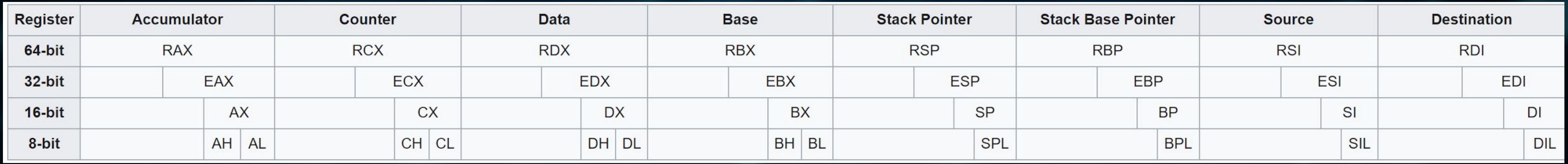
SIMPLE EXAMPLE: UNSIGNED VERSION

SIMPLE EXAMPLE: UNSIGNED VERSION



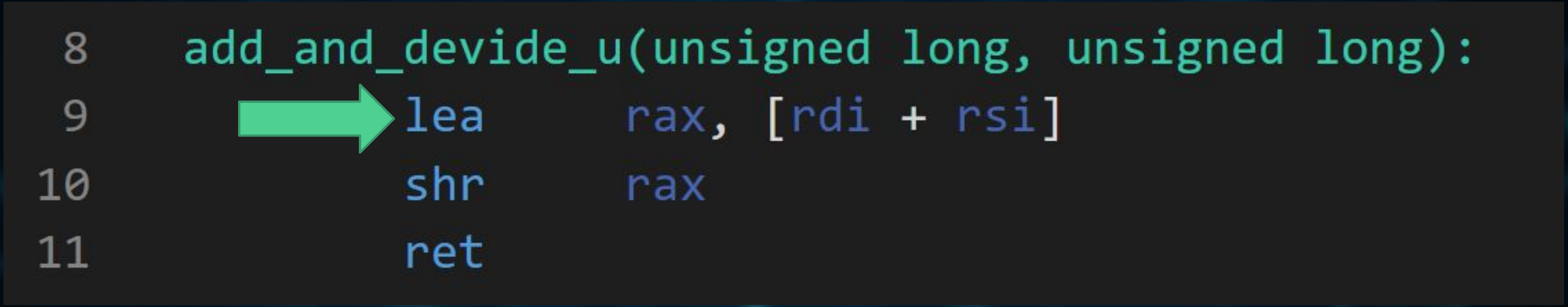
```
add_and_devide_u(unsigned long, unsigned long):
        lea     rax, [rdi + rsi]
        shr     rax
        ret
```

SIMPLE EXAMPLE: SOME ASSEMBLY



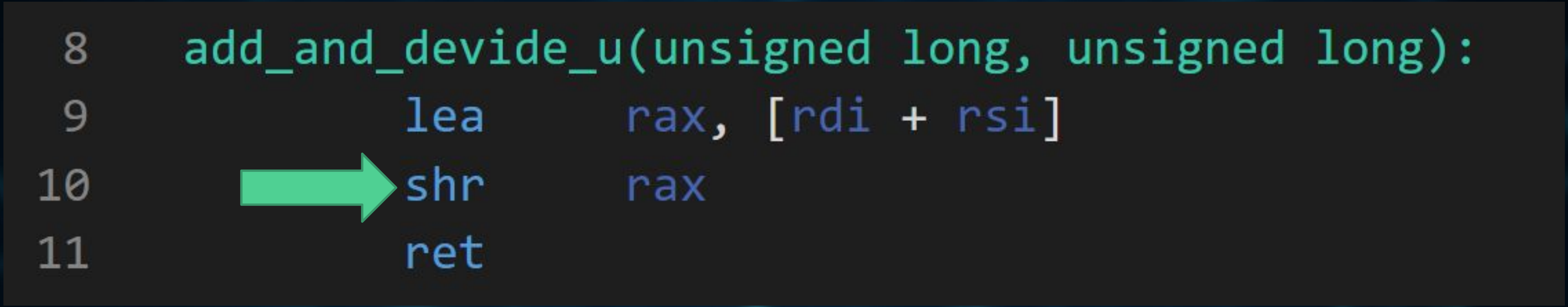
Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH AL	CH CL	DH DL	BH BL	SPL	BPL	SIL	DIL

SIMPLE EXAMPLE: SOME ASSEMBLY



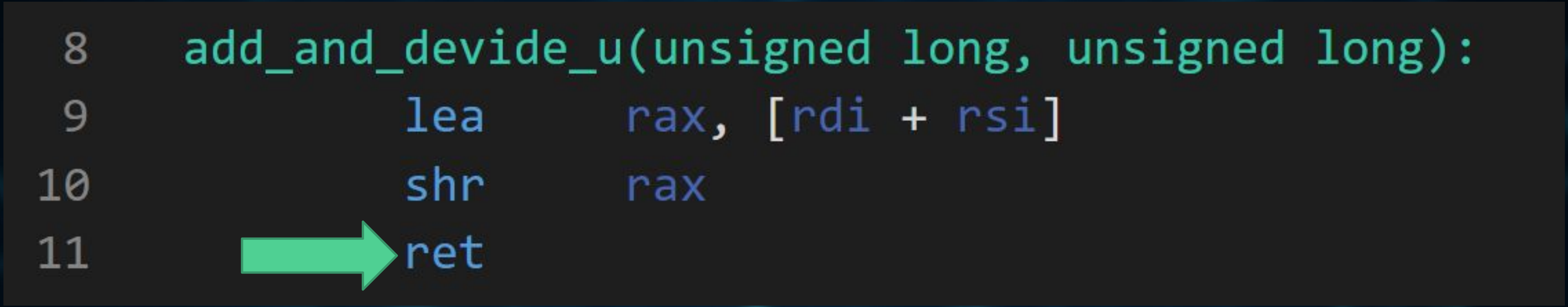
```
add_and_devide_u(unsigned long, unsigned long):
        lea     rax, [rdi + rsi]
        shr     rax
        ret
```

SIMPLE EXAMPLE: SOME ASSEMBLY



```
add_and_devide_u(unsigned long, unsigned long):
        lea     rax, [rdi + rsi]
        shr     rax
        ret
```

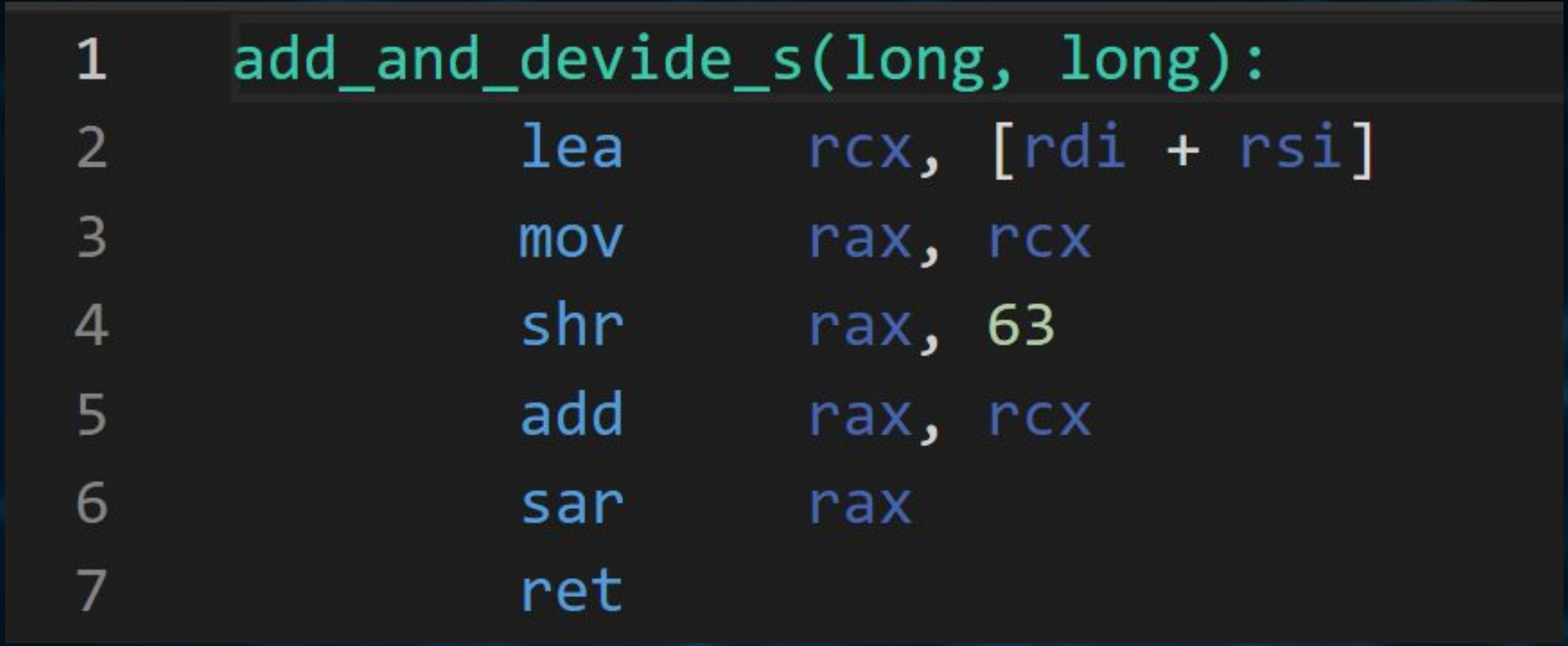
SIMPLE EXAMPLE: SOME ASSEMBLY



```
add_and_devide_u(unsigned long, unsigned long):
        lea     rax, [rdi + rsi]
        shr     rax
        ret
```

SIMPLE EXAMPLE: SIGNED VERSION

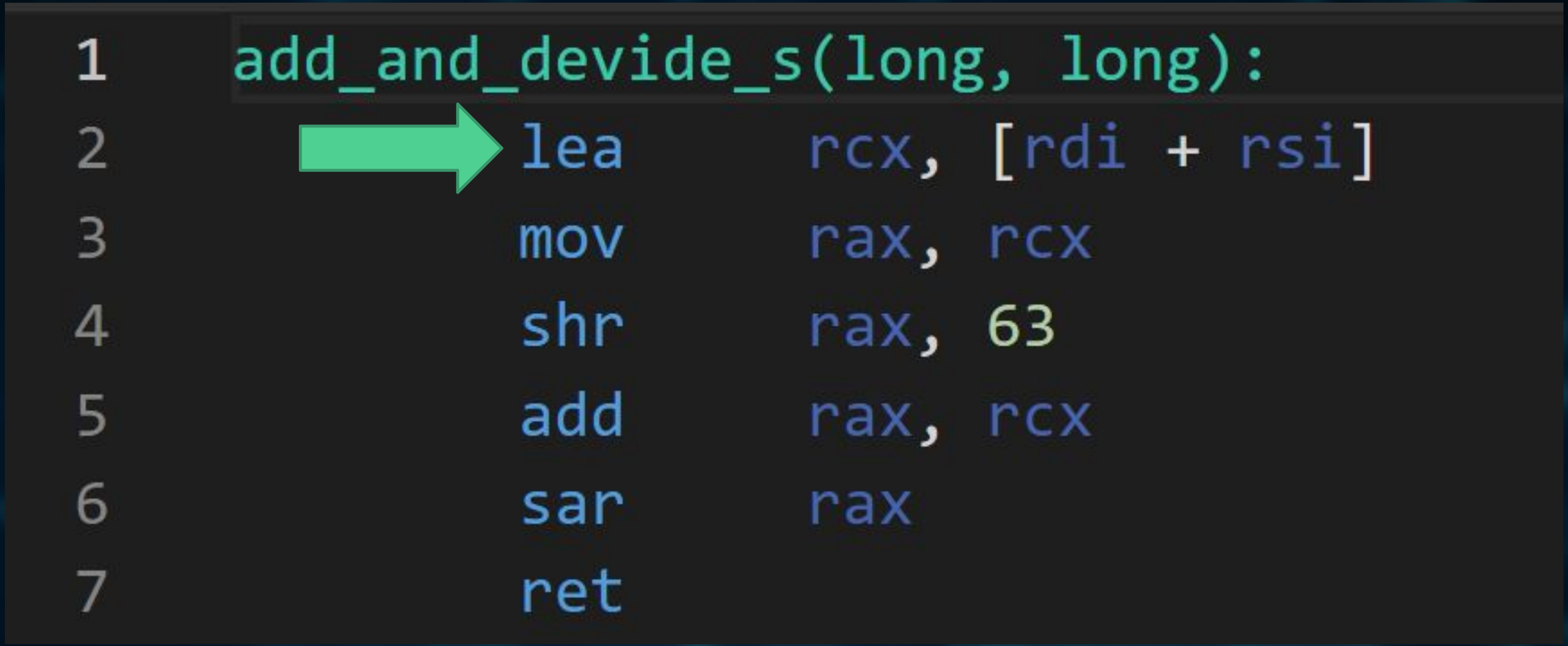
SIMPLE EXAMPLE: SIGNED VERSION



```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

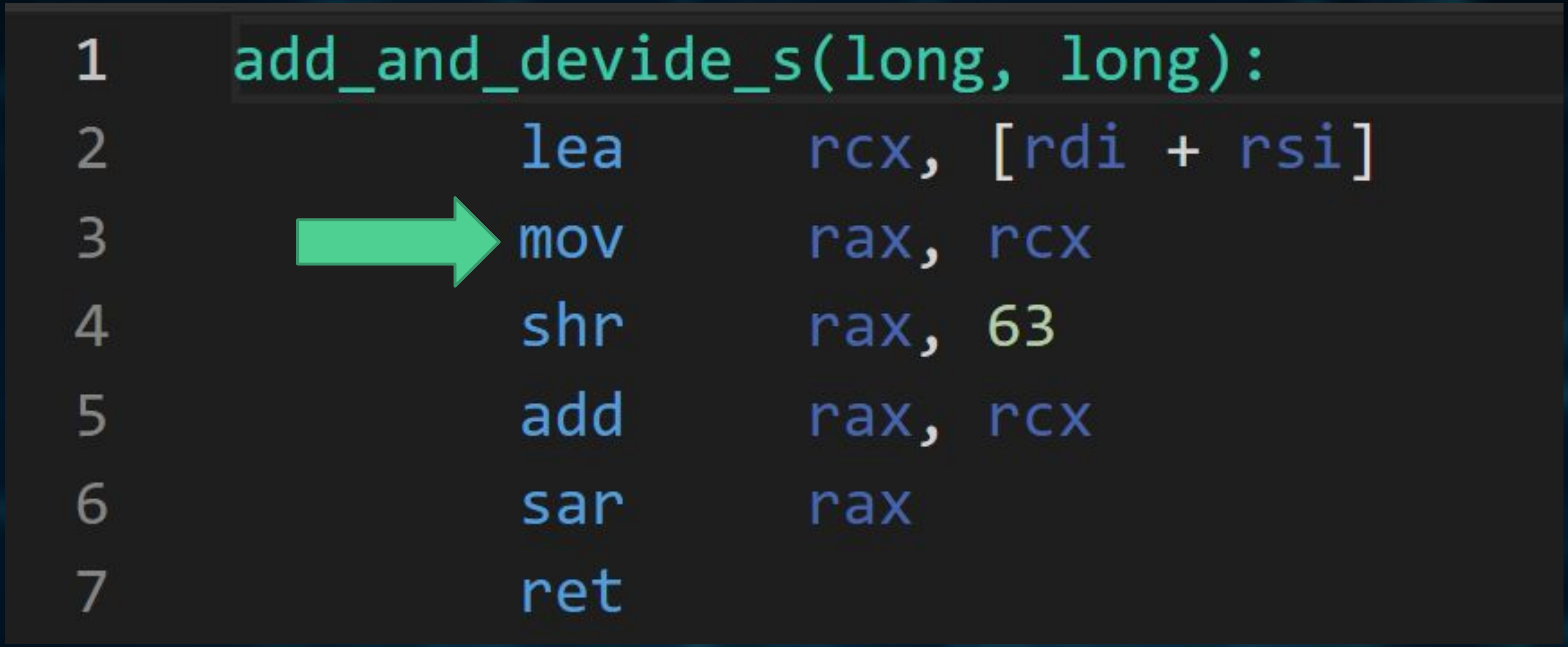
Surprise
☺️

SIMPLE EXAMPLE: SIGNED VERSION (ASSEMBLY)



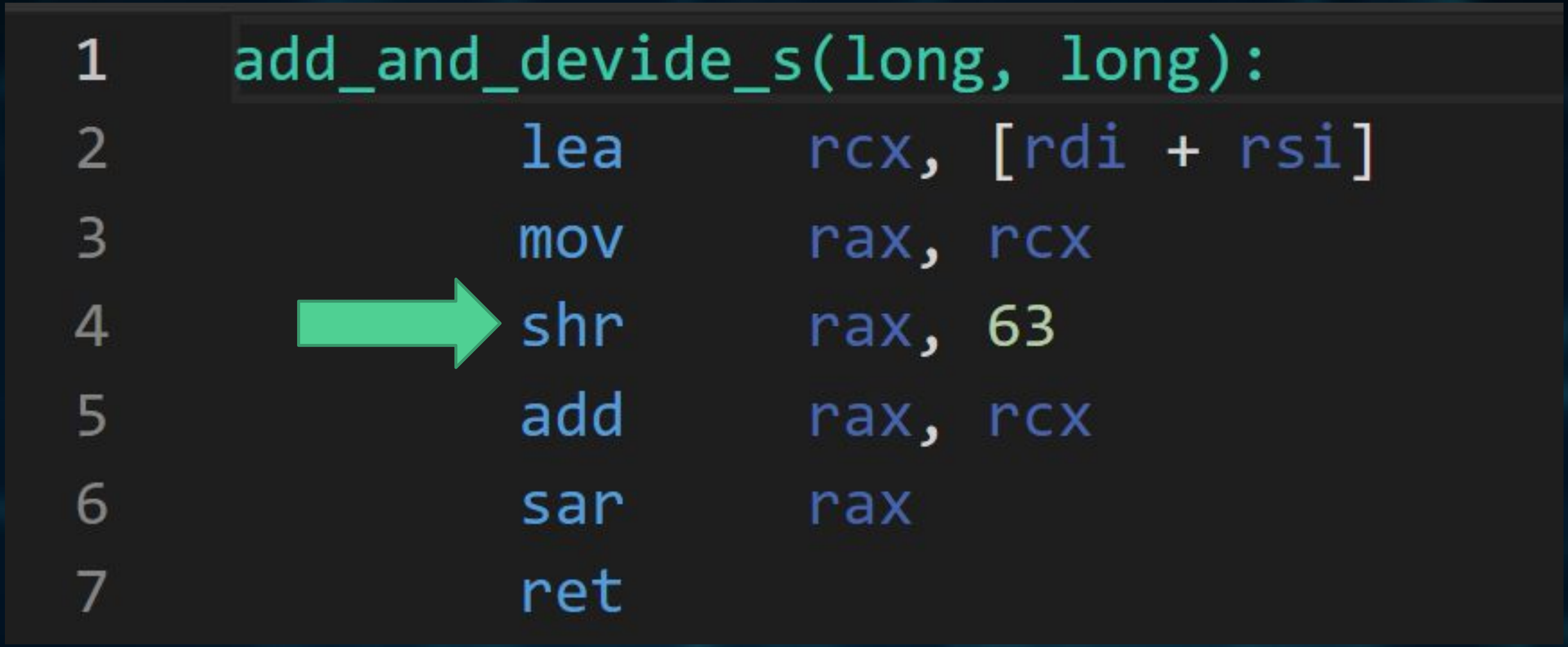
```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

SIMPLE EXAMPLE: SIGNED VERSION (ASSEMBLY)



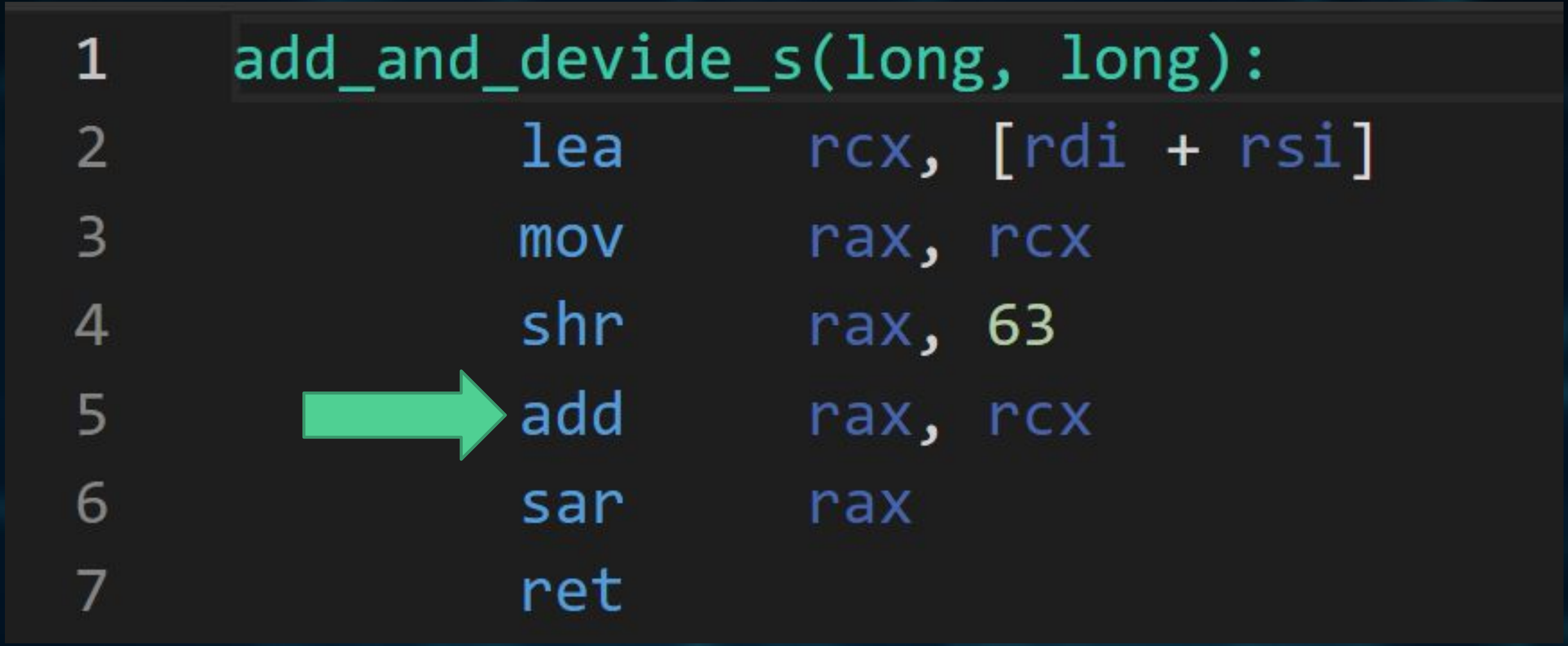
```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

SIMPLE EXAMPLE: SIGNED VERSION (ASSEMBLY)



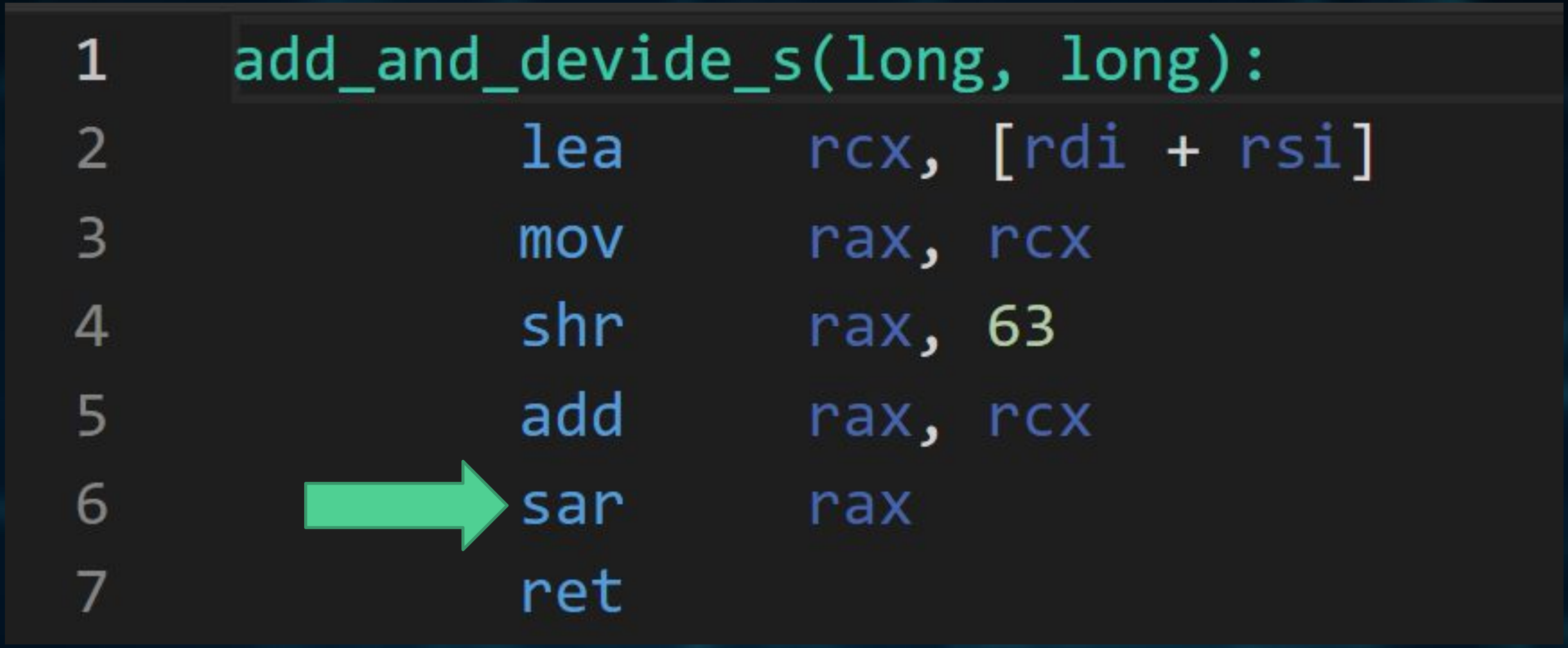
```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

SIMPLE EXAMPLE: SIGNED VERSION (ASSEMBLY)



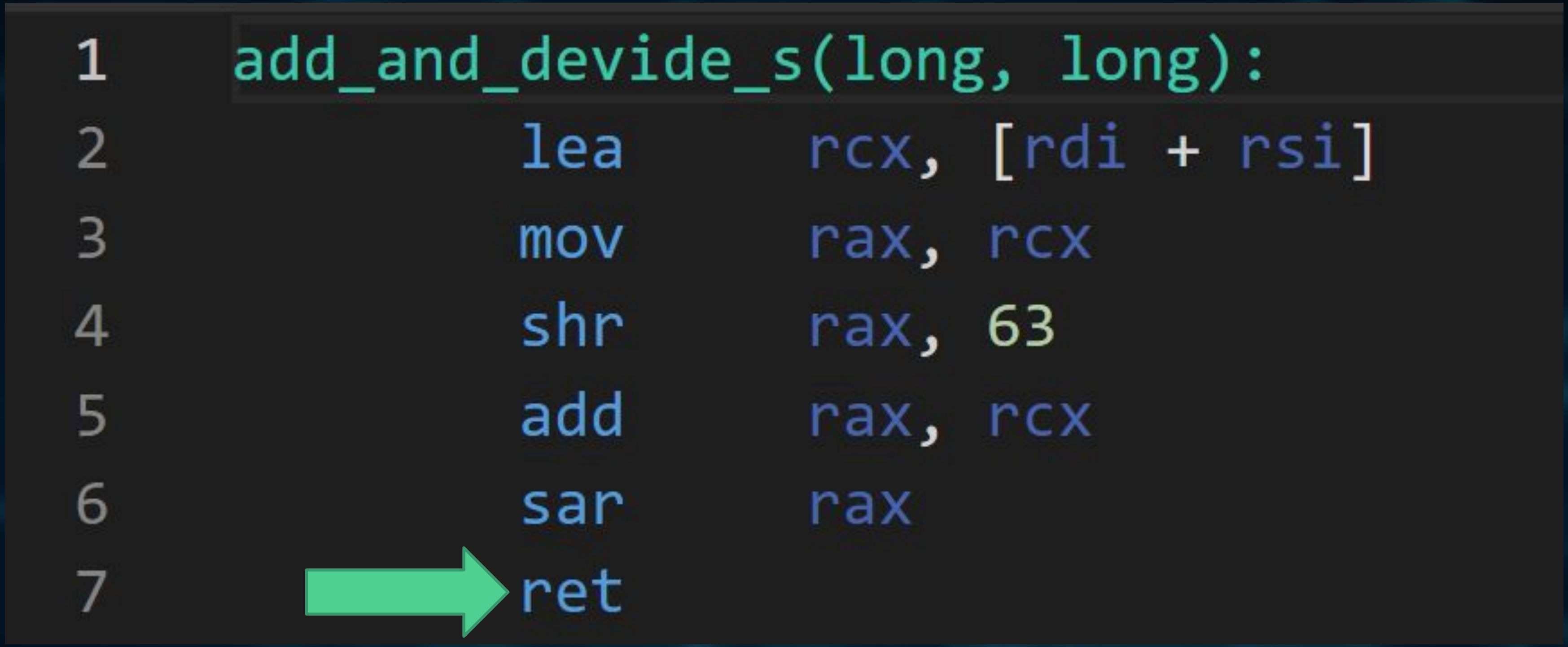
```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

SIMPLE EXAMPLE: SIGNED VERSION (ASSEMBLY)



```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

SIMPLE EXAMPLE: SIGNED VERSION (ASSEMBLY)



```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

WHY DID THIS HAPPEN?

WHY DID THIS HAPPEN:

- In the current example we are dividing the number by two and so, division by two is just like shifting the number right, as all numbers are represented in the binary form. Therefor using n/2 is equal to n>>1.

WHY DID THIS HAPPEN:

- In the current example we are dividing the number by two and so, division by two is just like shifting the number right, as all numbers are represented in the binary form. Therefor using n/2 is equal to n>>1.
- Each instruction that is fetched from the memory is pushed into a pipeline, one of the steps in the pipeline is execution, execution may be piped as well.

WHY DID THIS HAPPEN:

- In the current example we are dividing the number by two and so, division by two is just like shifting the number right, as all numbers are represented in the binary form. Therefor using n/2 is equal to n>>1.
- Each instruction that is fetched from the memory is pushed into a pipeline, one of the steps in the pipeline is execution, execution may be piped as well.
- Each execution has its own unit and there is a limited number of execution units. (depends on the **CPU**)

WHY DID THIS HAPPEN:

- In the current example we are dividing the number by two and so, division by two is just like shifting the number right, as all numbers are represented in the binary form. Therefor using n/2 is equal to n>>1.
- Each instruction that is fetched from the memory is pushed into a pipeline, one of the steps in the pipeline is execution, execution may be piped as well.
- Each execution has its own unit and there is a limited number of execution units. (depends on the **CPU**)
- Each instruction has its own latency

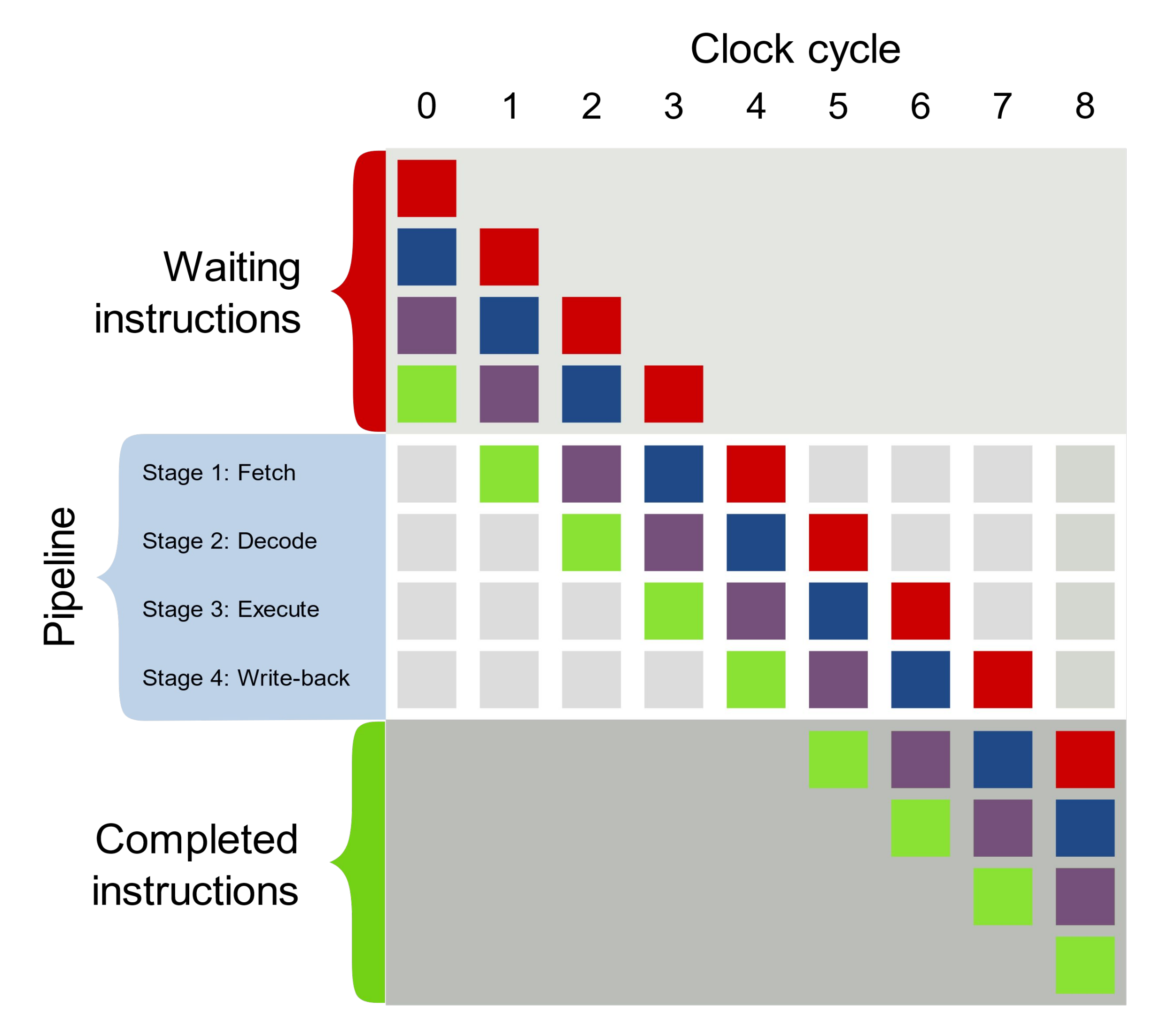
WHAT IS LATENCY?

- The number of cycles it takes to compute an instruction
    - Errors, misalignment, and cache misses might increase the cycles count
    - NAN’s and INFS do not increase the cycles

WHAT IS LATENCY: EXAMPLES

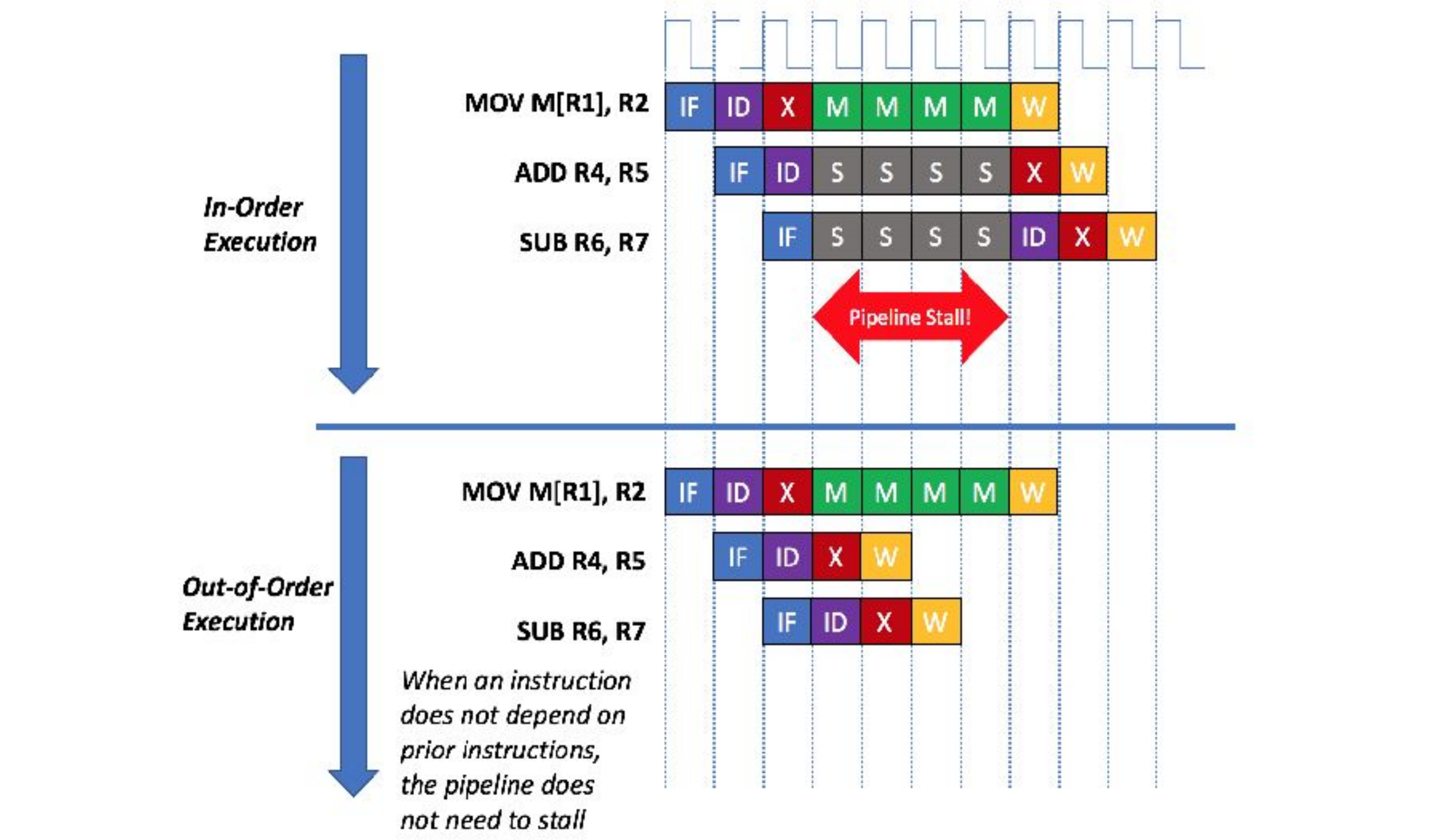
- ADD – 1 cycle (not piped)
- IMUL – 3 cycles
- DIV – at least x20 time slower than imul (depending on the architecture)

WHAT IS LATENCY: PIPES



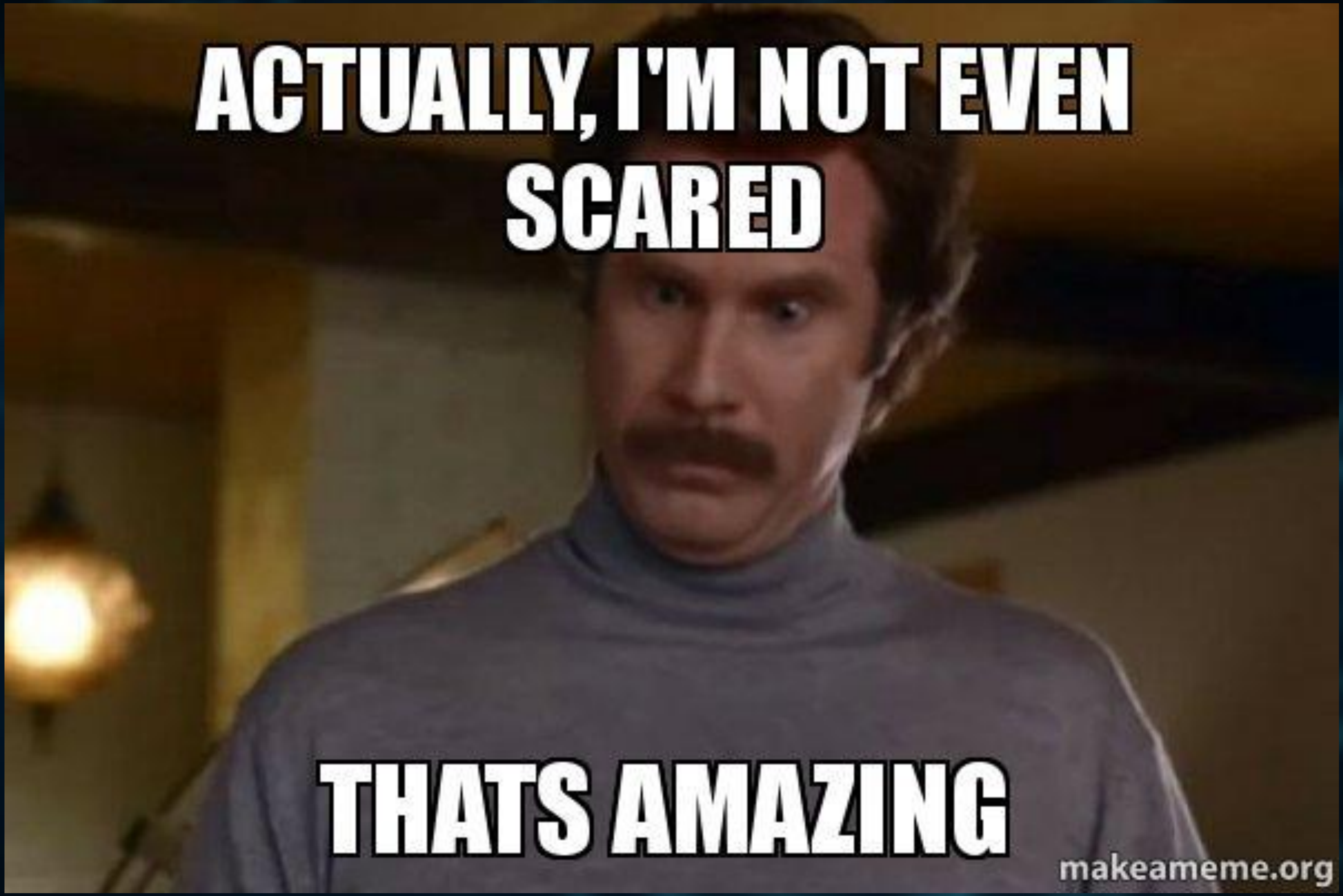
Source: https://en.wikipedia.org/wiki/Instruction_pipelining

WHAT IS LATENCY: PIPES

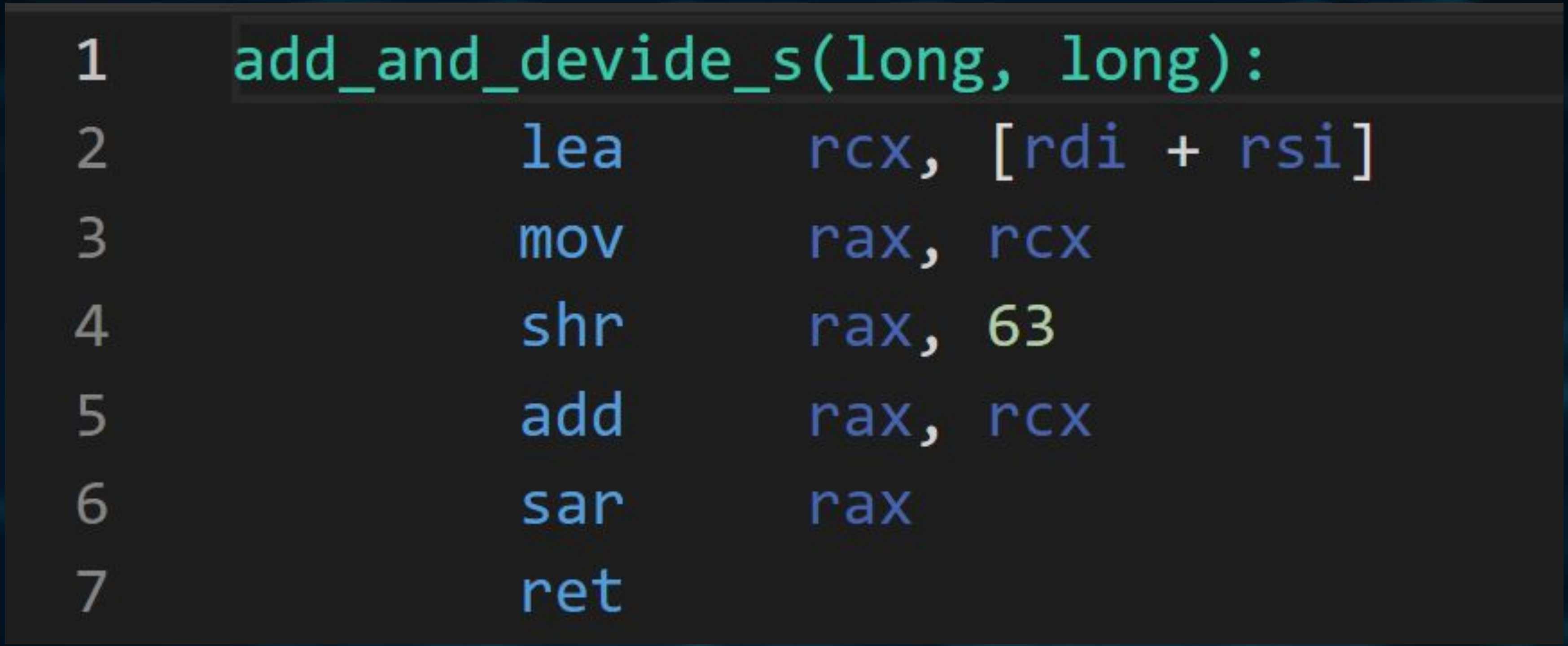


When an instruction does not depend on prior instructions, the pipeline does not need to stall

Source:
https://www.semanticscholar.org/paper/RISC-V-Reward:-Building-Out-of-Order-Processors-in-Zekany-Tan/f7f6d27f334604c3c85f0b8d21d2a9b4df22a983


ACTUALLY, I'M NOT EVEN SCARED
THATS AMAZING
makeameme.org

BACK TO OUR EXAMPLE: WHAT HAPPENED?



```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

SIGNED VS UNSIGNED INTEGERS

SIGNED VS UNSIGNED INTEGERS

- **Unsigned Integers**:
  - Stored using Modulo 2 representation
  - Support only positive numbers
  - Overflow is well-defined
  - The Range of a *64-bit* **unsigned integer** is *0* to *18,446,744,073,709,551,615 (2^n)*

SIGNED VS UNSIGNED INTEGERS

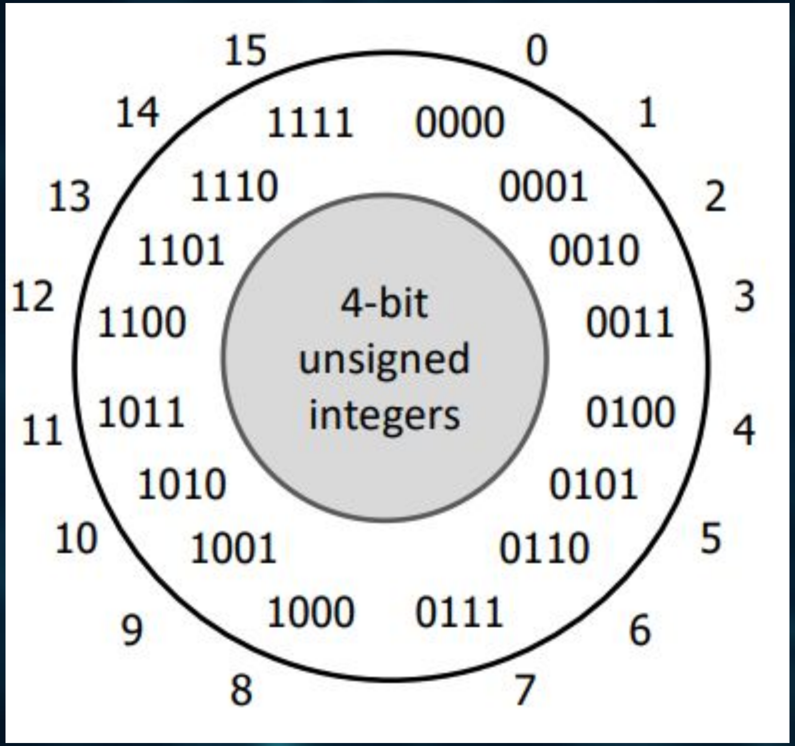
- **Unsigned Integers**: representation

Bits:	1	0	1	1	0
Wight:	2^4	2^3	2^2	2^1	2^0

Actual: $1*2^4+0*2^3+1*2^2+1*2^1+0*2^0 = 1*16+0*8+1*4+1*2+0*1 = 22$

SIGNED VS UNSIGNED INTEGERS

- **Unsigned Integers**: Overflow



15 + 2 = 1

1111 0010 = 0001

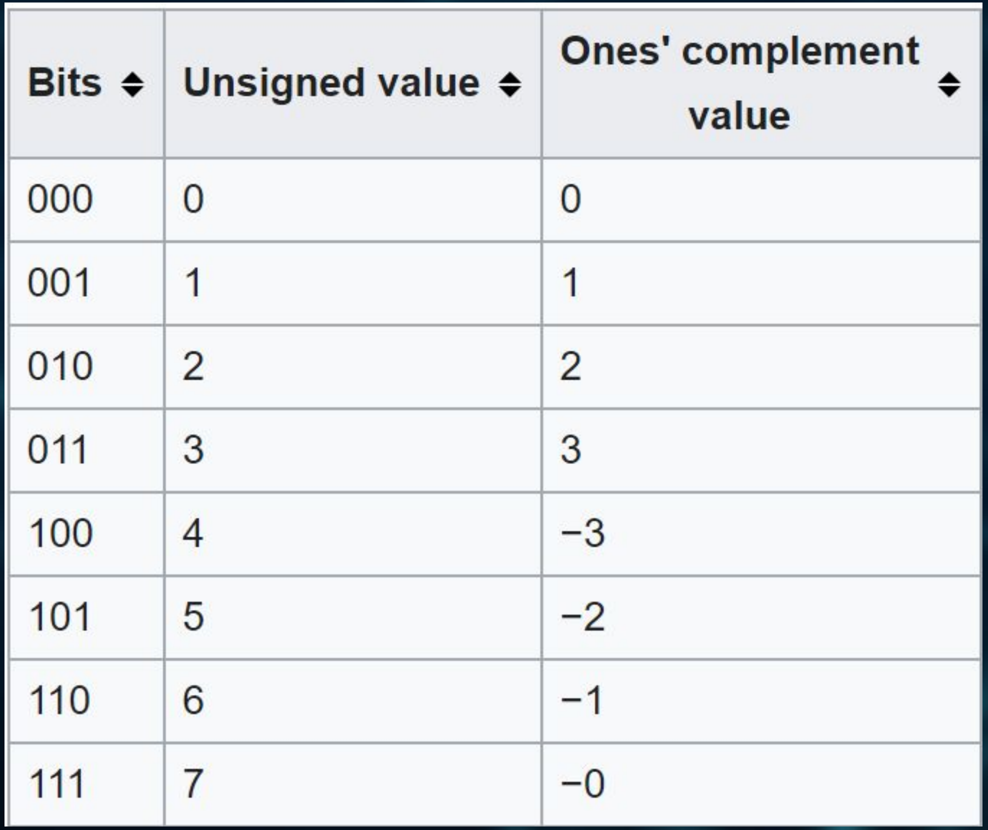
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SIGNED VS UNSIGNED INTEGERS

- **Signed Integers:**
  - support negative numbers
  - Stored using:
    - Sign and magnitude
    - One’s complement
    - Two’s complement
  - overflow is considered undefined behavior
  - The range of a *64-bit* **signed integer** is *-9,223,372,036,854,775,808* to *9,223,372,036,854,775,807*

SIGNED VS UNSIGNED INTEGERS

- One’s Complement
  - Representing negative numbers by inverting all bits



Bits	Unsigned value	Ones' complement value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	−3
101	5	−2
110	6	−1
111	7	−0

Source: https://en.wikipedia.org/wiki/Ones%27_complement

SIGNED VS UNSIGNED INTEGERS

- One’s Complement
  - Example: 4 bit number

F = 1111 (unsigned)

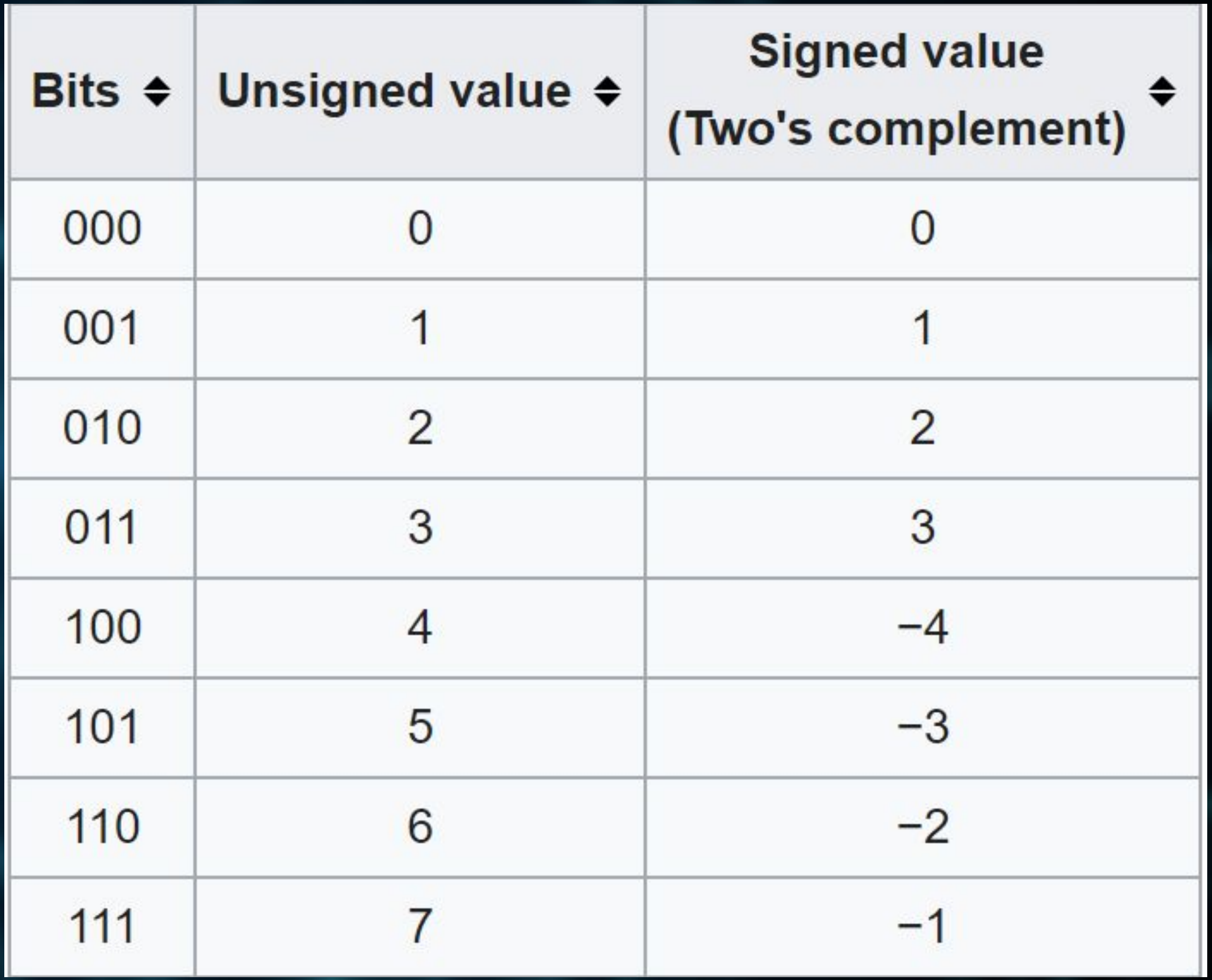
↓

0 0 0 0 = -0

Source: https://en.wikipedia.org/wiki/Ones%27_complement

SIGNED VS UNSIGNED INTEGERS

- Two’s Complement
  - Start with a positive number
  - Invert all bits
  - Add 1 and ignore overflows

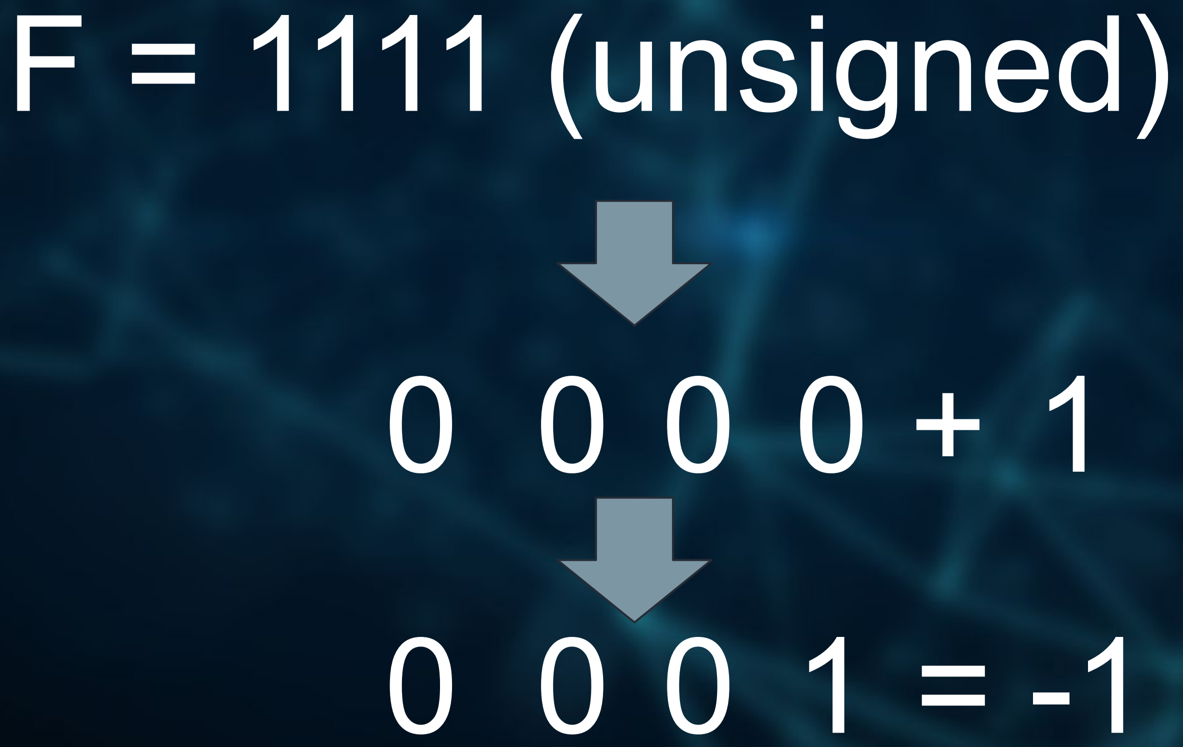


Bits	Unsigned value	Signed value (Two's complement)
000	0	0
001	1	1
010	2	2
011	3	3
100	4	−4
101	5	−3
110	6	−2
111	7	−1

Source: https://en.wikipedia.org/wiki/Two%27s_complement

SIGNED VS UNSIGNED INTEGERS

- Two’s Complement
    - Example: 4 bit number



F = 1111 (unsigned)

0 0 0 0 + 1

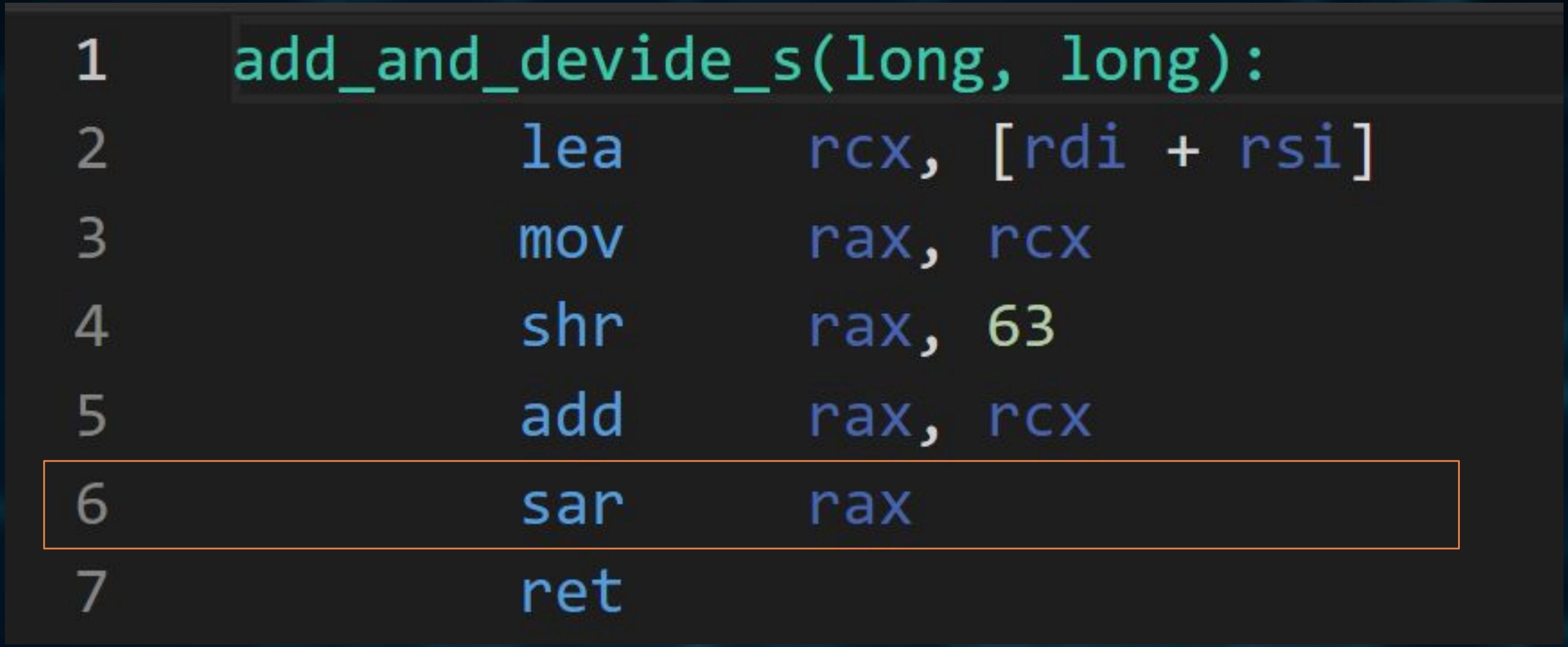
0 0 0 1 = -1

Source: https://en.wikipedia.org/wiki/Ones%27_complement

SIGNED VS UNSIGNED INTEGERS

- Positive numbers are represented in the same way for signed and unsigned
- Since C++20 negative numbers are represented only with Two’s complement

BACK TO OUR EXAMPLE: WHAT HAPPENED?



```
add_and_devide_s(long, long):
        lea     rcx, [rdi + rsi]
        mov     rax, rcx
        shr     rax, 63
        add     rax, rcx
        sar     rax
        ret
```

DIFFERENCE BETWEEN SHR AND SAR

DIFFERENCE BETWEEN SHR AND SAR

SHR: Logical right shift means shifting the bits to the right and **MSB** becomes 0.

Example: shr **1** 0 1 1 0 1 1 1 = **0** 1 0 1 1 0 1 1

DIFFERENCE BETWEEN SHR AND SAR

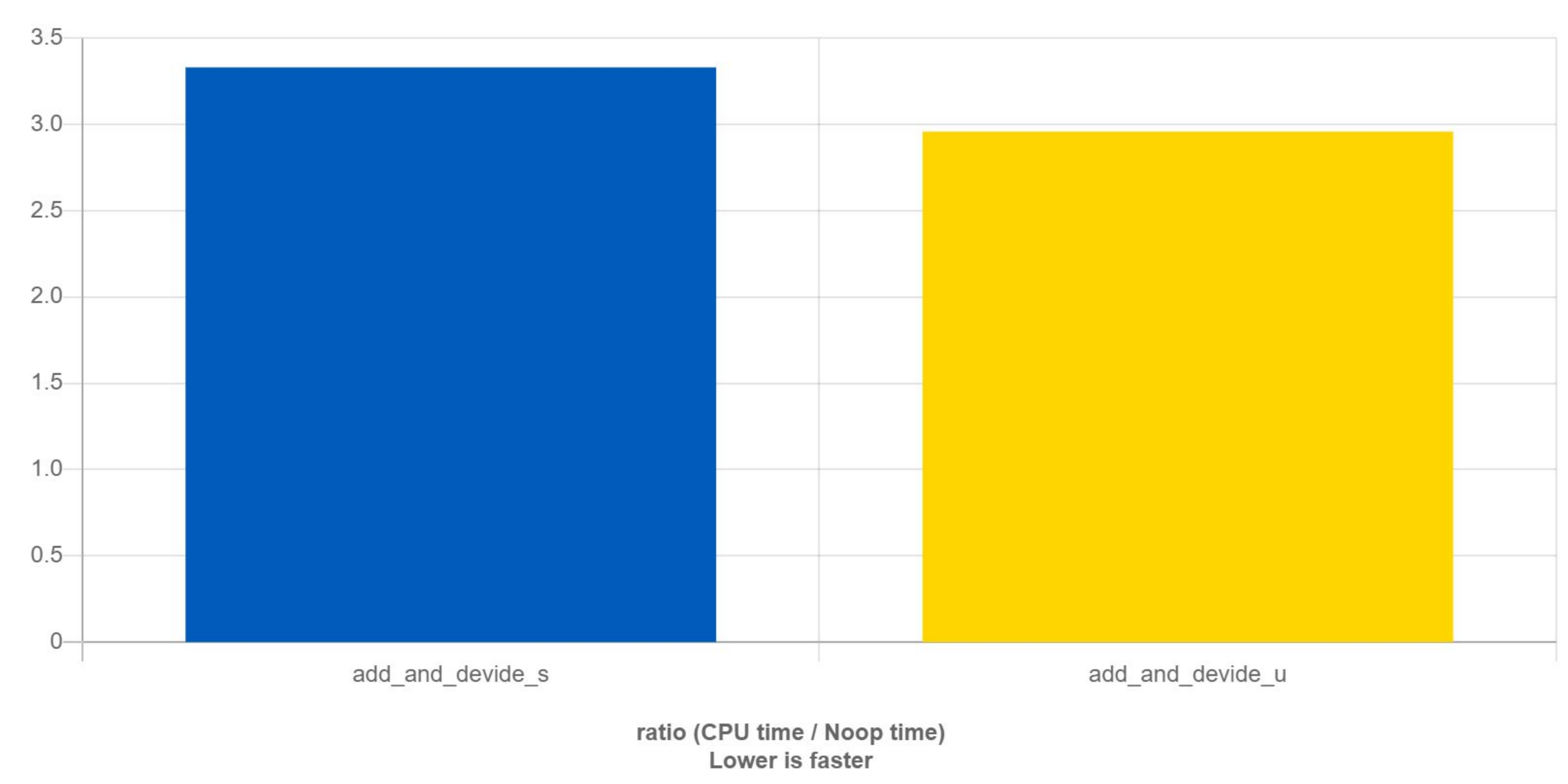
SAR: Arithmetic right shift means shifting the bits to the right and **MSB** bit is same as in the original number.

Example: sar **1** 0 1 1 0 1 0 1 = **1** 1 0 1 1 0 1 0.

BACK TO OUR EXAMPLE: WHAT HAPPENED?

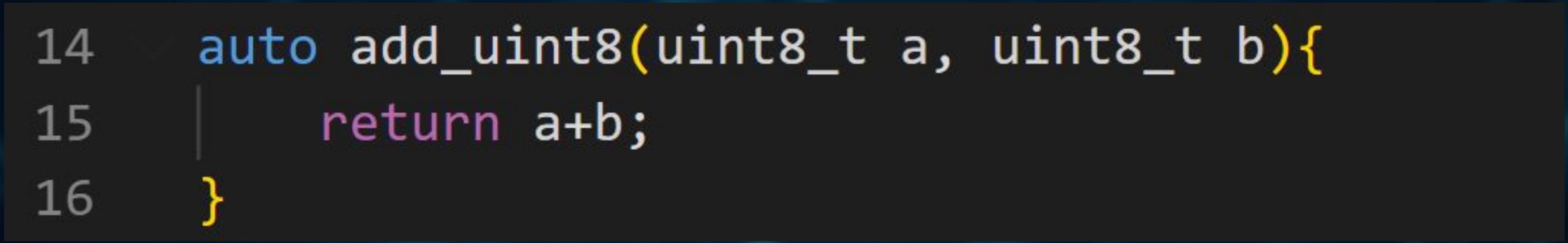
```
add_
                    i]
```

BACK TO OUR EXAMPLE: PERFORMANCE



SIGNED AND UNSIGNED PITFALLS

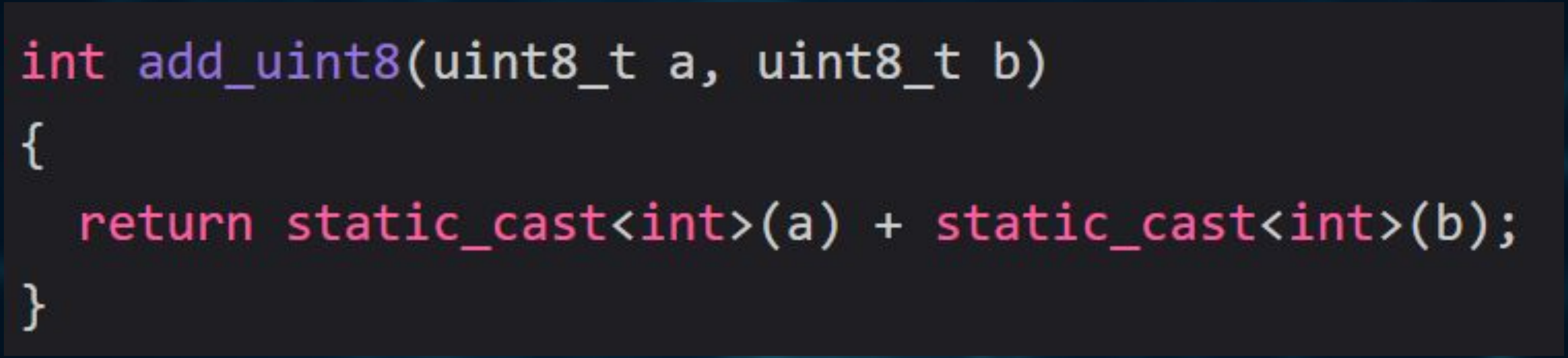
SIGNED AND UNSIGNED PITFALLS:



```
auto add_uint8(uint8_t a, uint8_t b){
    return a+b;
}
```

- what will the result be if we call add_uint8(255u, 1u)?

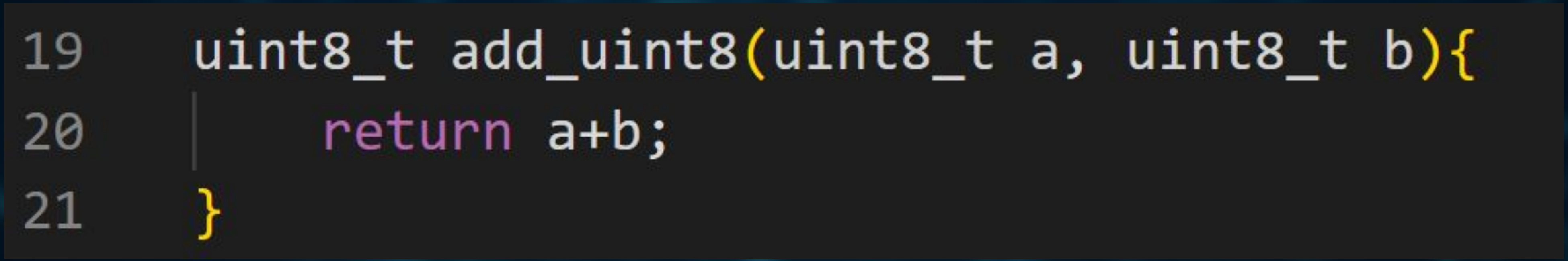
SURPRISE AGAIN: INTEGER PROMOTION



```
int add_uint8(uint8_t a, uint8_t b)
{
  return static_cast<int>(a) + static_cast<int>(b);
}
```

256

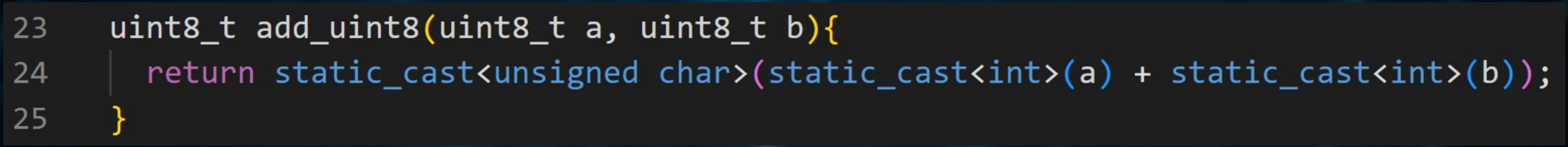
SIGNED AND UNSIGNED PITFALLS:



```
uint8_t add_uint8(uint8_t a, uint8_t b){
    return a+b;
}
```

- what will be the result if we will call add_uint8(255u, 1u) ?

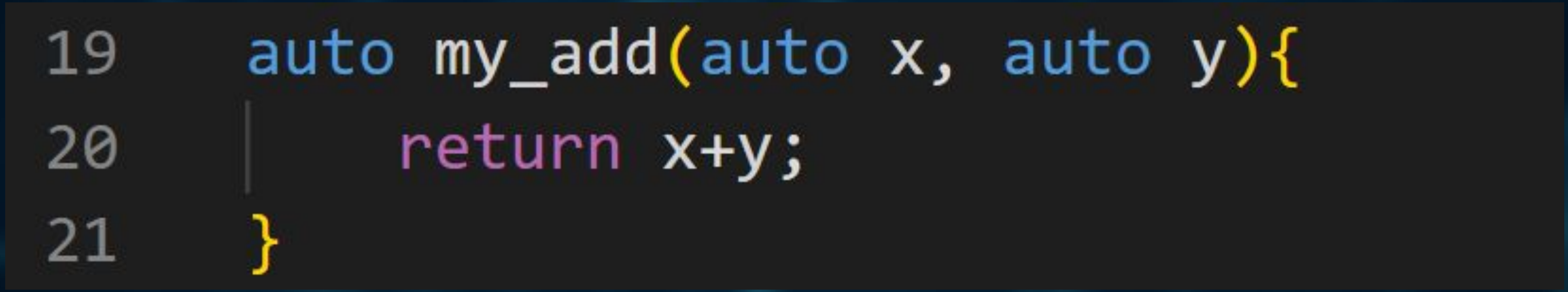
INTEGER NARROWING



```
uint8_t add_uint8(uint8_t a, uint8_t b){
  return static_cast<unsigned char>(static_cast<int>(a) + static_cast<int>(b));
}
```

0

SURPRISE AGAIN: WAIT THAT’S NOT ALL

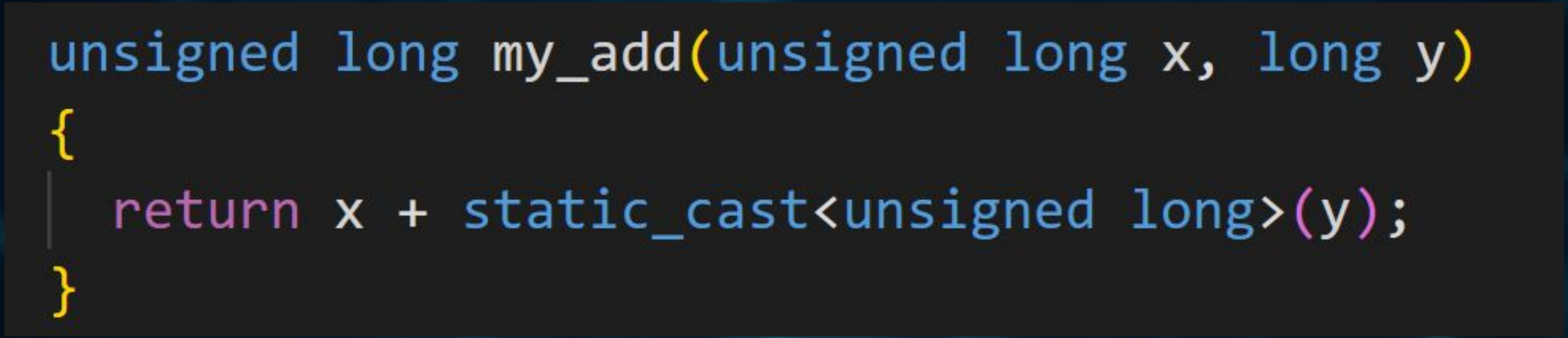


```
auto my_add(auto x, auto y){
    return x+y;
}
```

- what will be the result if we will call

  my_add(uint64_t(1), int64_t(-2)) ?

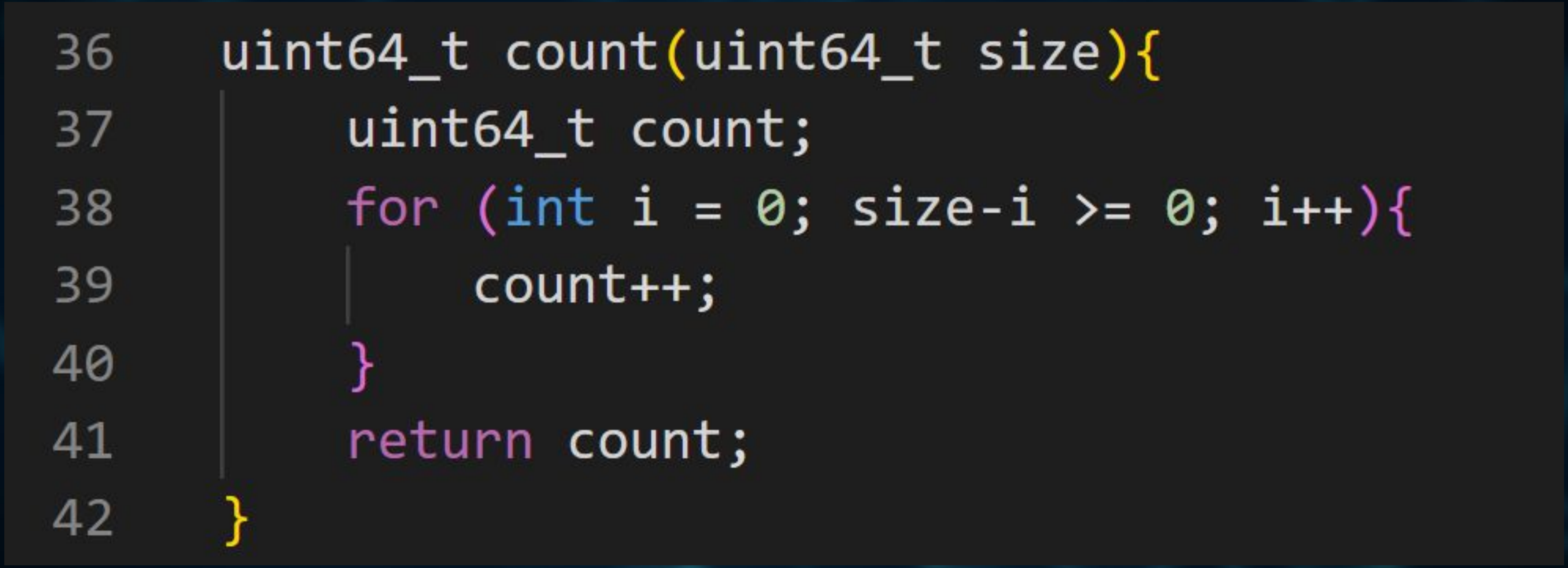
SURPRISE AGAIN: MORE INTEGER PROMOTION



```
unsigned long my_add(unsigned long x, long y)
{
  return x + static_cast<unsigned long>(y);
}
```

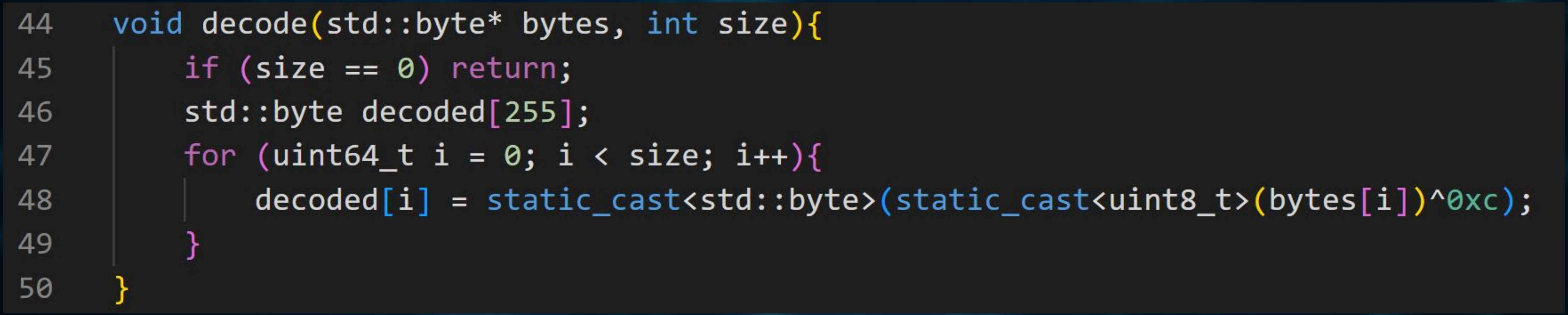
18446744073709551615

MIXING INTEGER TYPES MAY CAUSE HORRIBLE BUGS



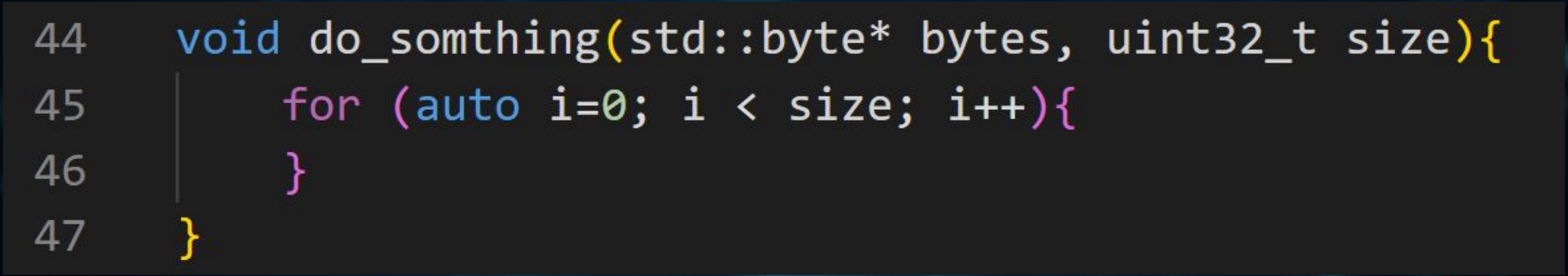
```
uint64_t count(uint64_t size){
    uint64_t count;
    for (int i = 0; size-i >= 0; i++){
        count++;
    }
    return count;
}
```

MIXING INTEGER TYPES MAY CAUSE HORRIBLE BUGS



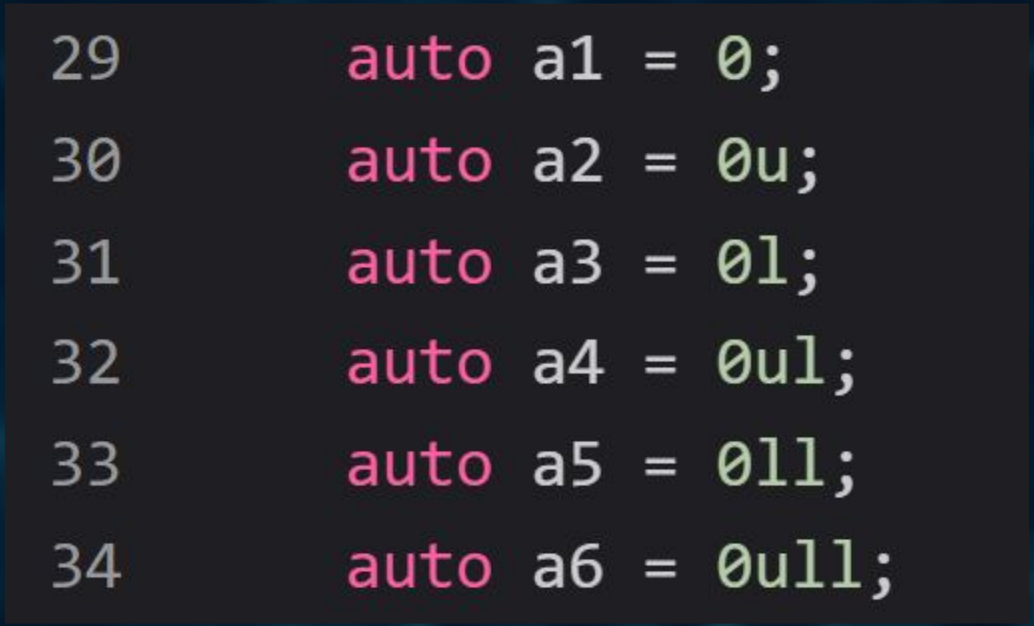
```cpp
void decode(std::byte* bytes, int size){
    if (size == 0) return;
    std::byte decoded[255];
    for (uint64_t i = 0; i < size; i++){
        decoded[i] = static_cast<std::byte>(static_cast<uint8_t>(bytes[i])^0xc);
    }
}
```

BEWARE OF THIS PATTERN



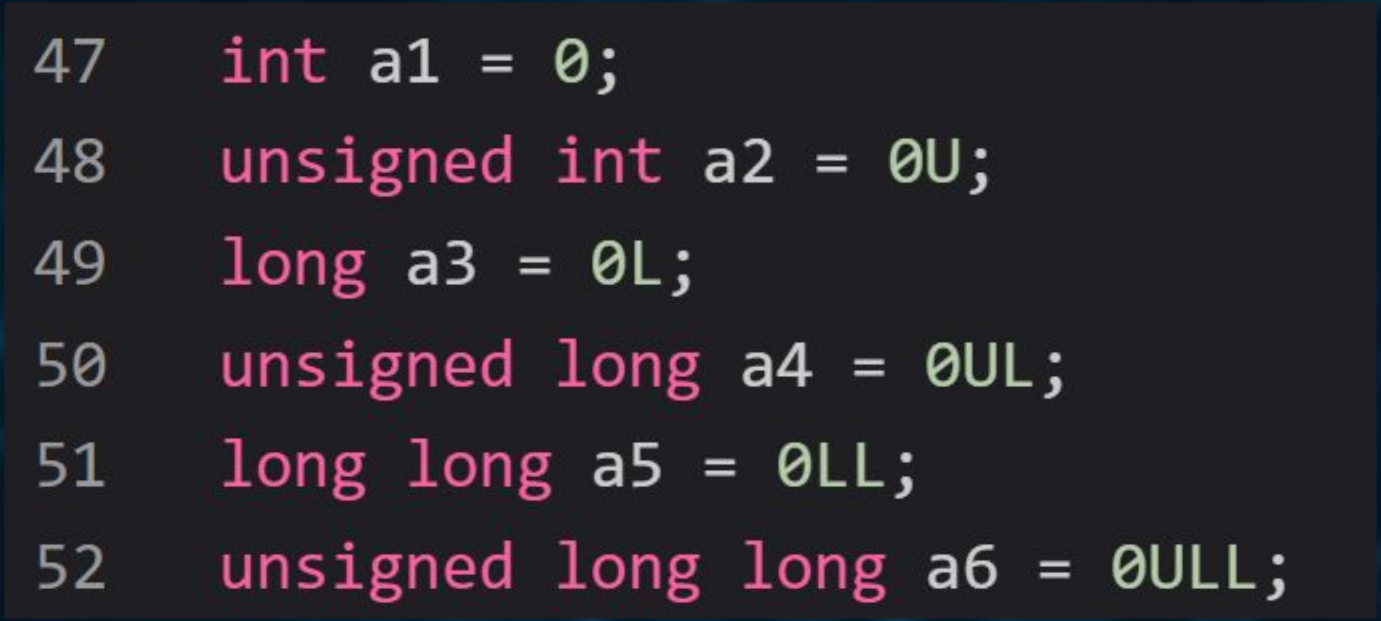
```
void do_somthing(std::byte* bytes, uint32_t size){
    for (auto i=0; i < size; i++){
    }
}
```

AUTO IS GREAT: USE IT DON’T ABUSE IT



```cpp
        auto a1 = 0;
        auto a2 = 0u;
        auto a3 = 0l;
        auto a4 = 0ul;
        auto a5 = 0ll;
        auto a6 = 0ull;
```

AUTO IS GREAT: USE IT DON'T ABUSE IT



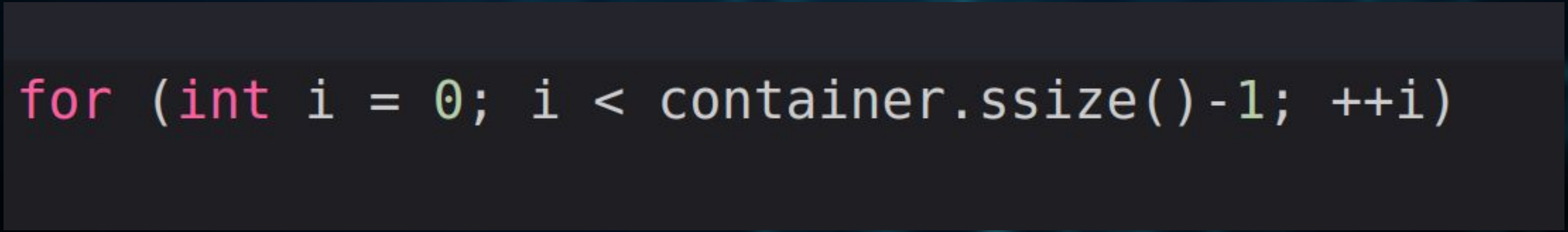
```
int a1 = 0;
unsigned int a2 = 0U;
long a3 = 0L;
unsigned long a4 = 0UL;
long long a5 = 0LL;
unsigned long long a6 = 0ULL;
```

SIZE_T AND SSIZE_T

- **size_t:**
  - Unsigned integer
  - Used for size operations
  - Defined in cstddef
  - size_t limit is SIZE_MAX
  - Introduced in C89 to eliminate portability problems

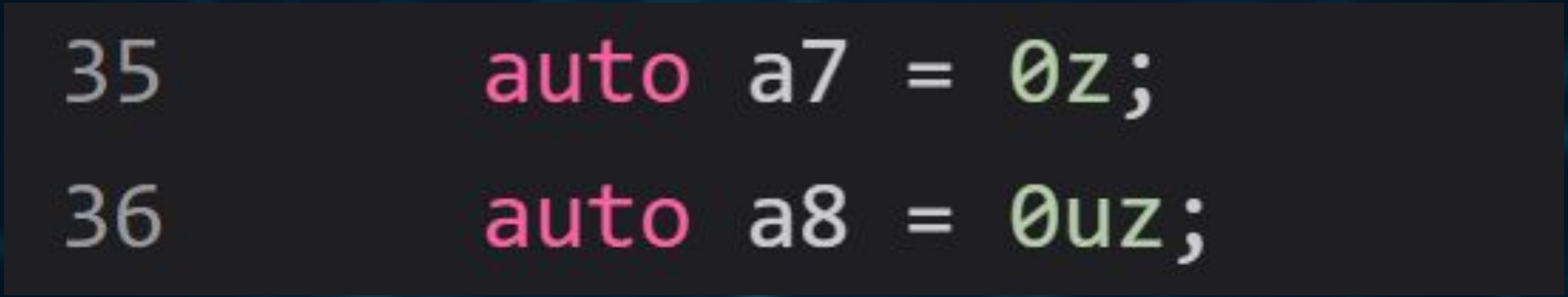
SIZE_T AND SSIZE_T

- **ssize_t:**
  - Signed version of size_t
  - Defined by POSIX.1-2017
  - Represent at least the range [-1,{SSIZE_MAX}].



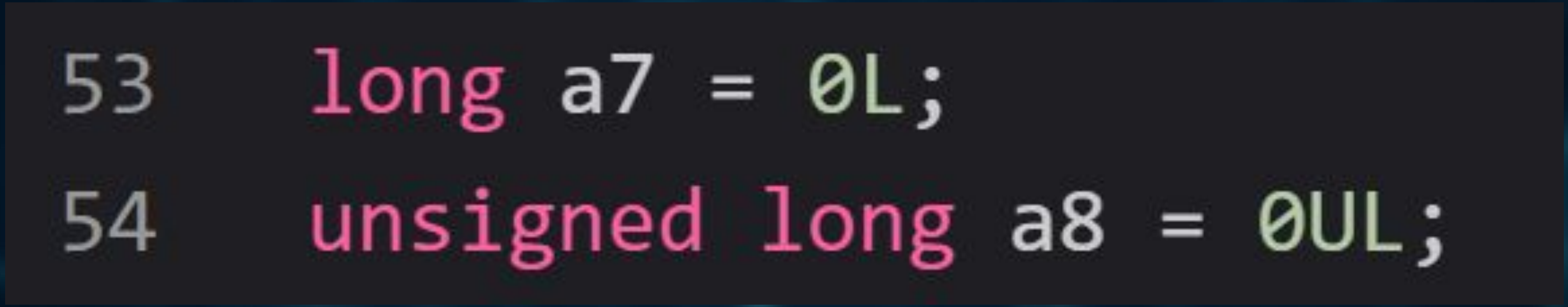
```
for (int i = 0; i < container.ssize()-1; ++i)
```

AUTO IS GREAT: USE IT DON’T ABUSE IT (C++23 ADDITIONS)



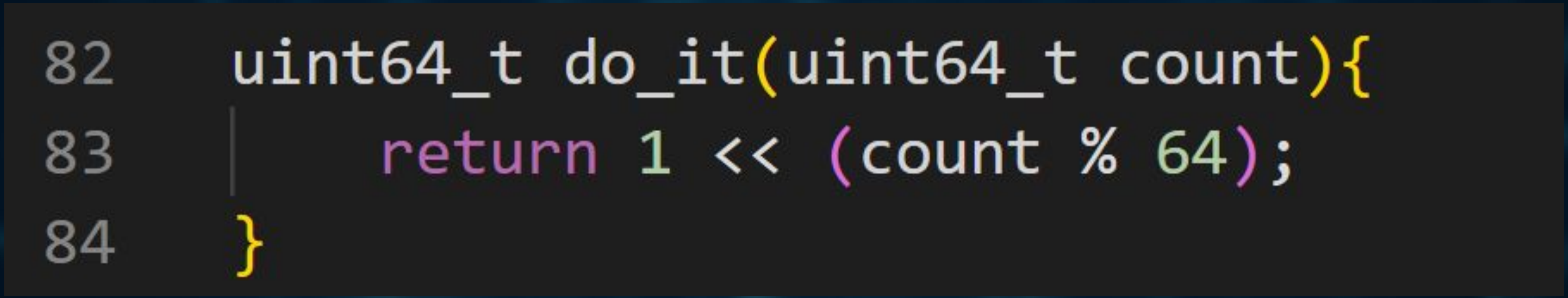
```
    auto a7 = 0z;
    auto a8 = 0uz;
```

AUTO IS GREAT: USE IT DON’T ABUSE IT (C++23 ADDITIONS)



```
long a7 = 0L;
unsigned long a8 = 0UL;
```

POP QUIZ ☺



```
uint64_t do_it(uint64_t count){
    return 1 << (count % 64);
}
```

MORE COMPLEX EXAMPLE: ARITHMETIC SERIES

MORE COMPLEX EXAMPLE: ARITHMETIC SERIES

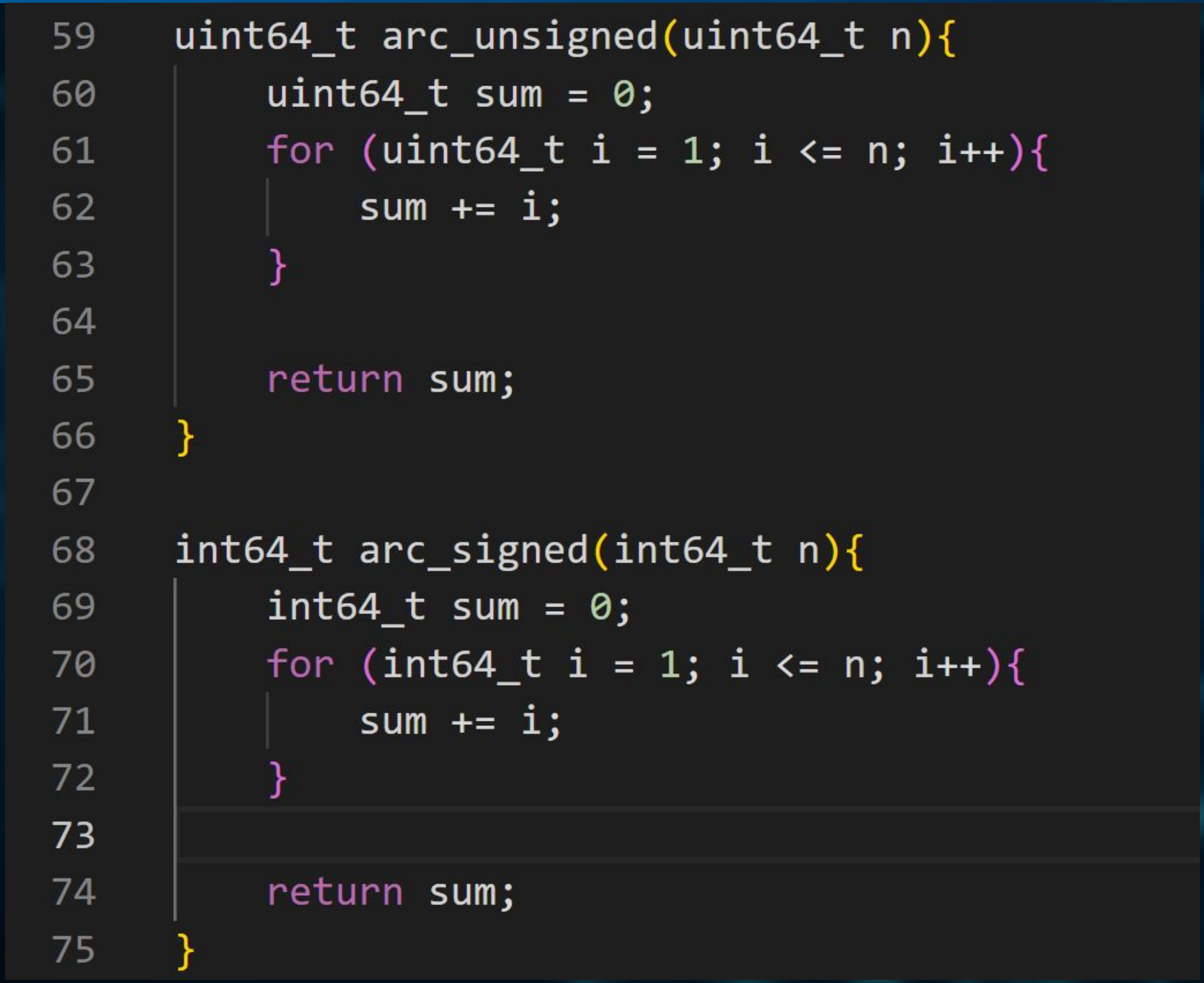
- Series of numbers where the difference between any two sequential numbers is constant.

For example, *1,2,3,4,5,6,7,8,9,10,…,n* is an **Arithmetic Series** where the difference between any two sequential numbers is *1*.

MORE COMPLEX EXAMPLE: ARITHMETIC SERIES

$$\sum_{k=1}^{n} a_k = \frac{n(a_1 + \ a_n)}{2}$$

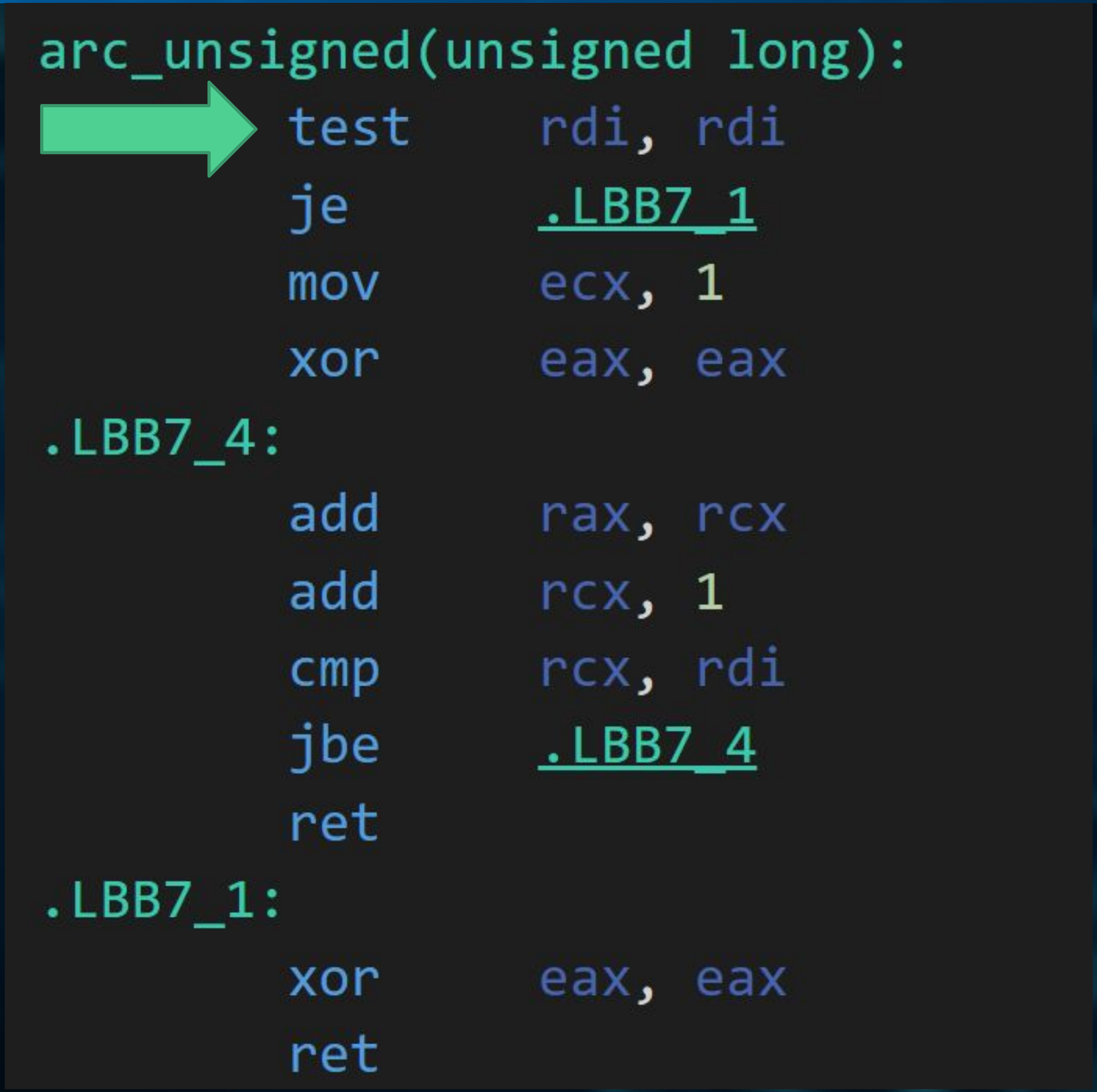
MORE COMPLEX EXAMPLE: ARITHMETIC SERIES



```
uint64_t arc_unsigned(uint64_t n){
    uint64_t sum = 0;
    for (uint64_t i = 1; i <= n; i++){
        sum += i;
    }

    return sum;
}

int64_t arc_signed(int64_t n){
    int64_t sum = 0;
    for (int64_t i = 1; i <= n; i++){
        sum += i;
    }

    return sum;
}
```

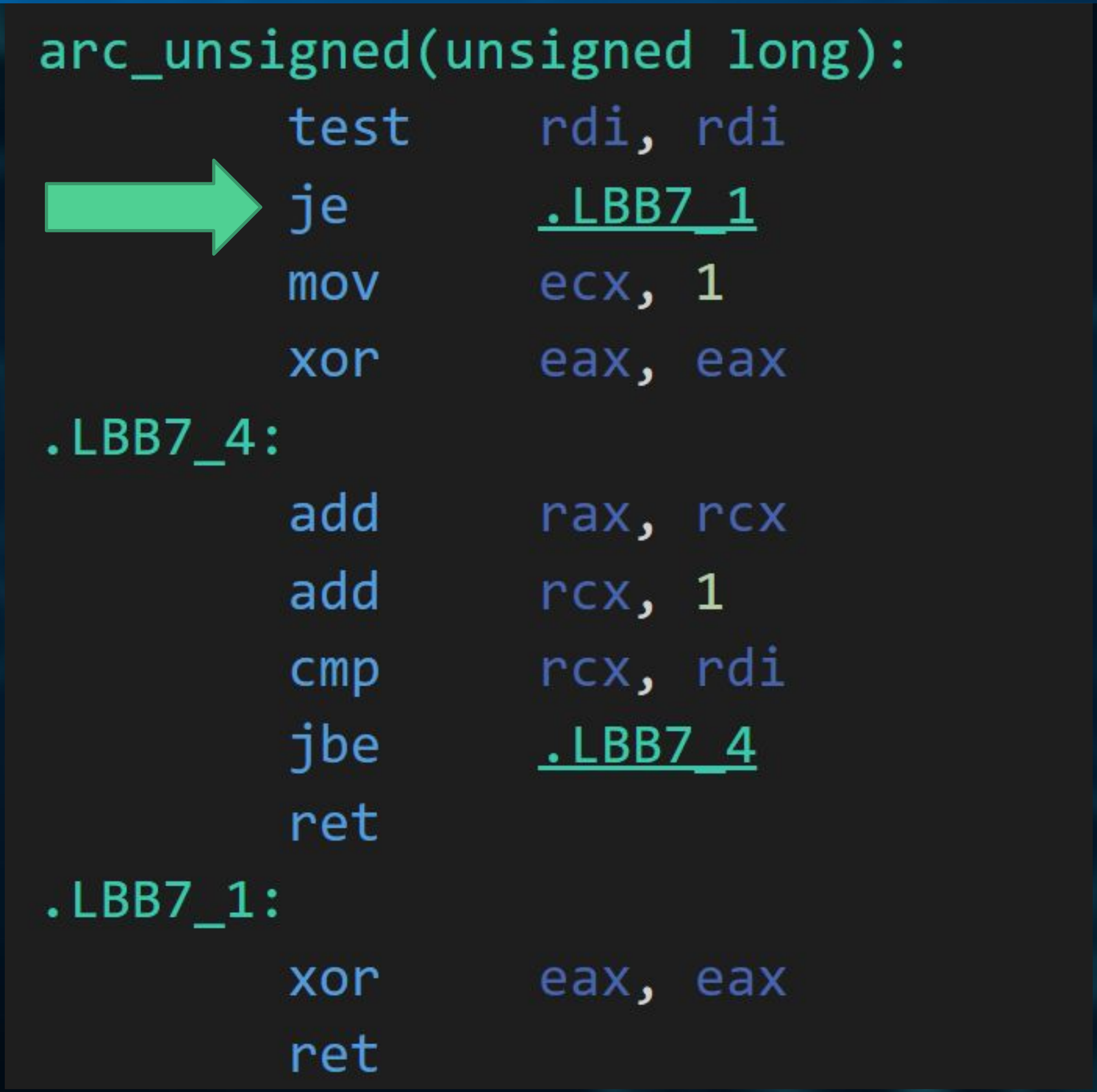
MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



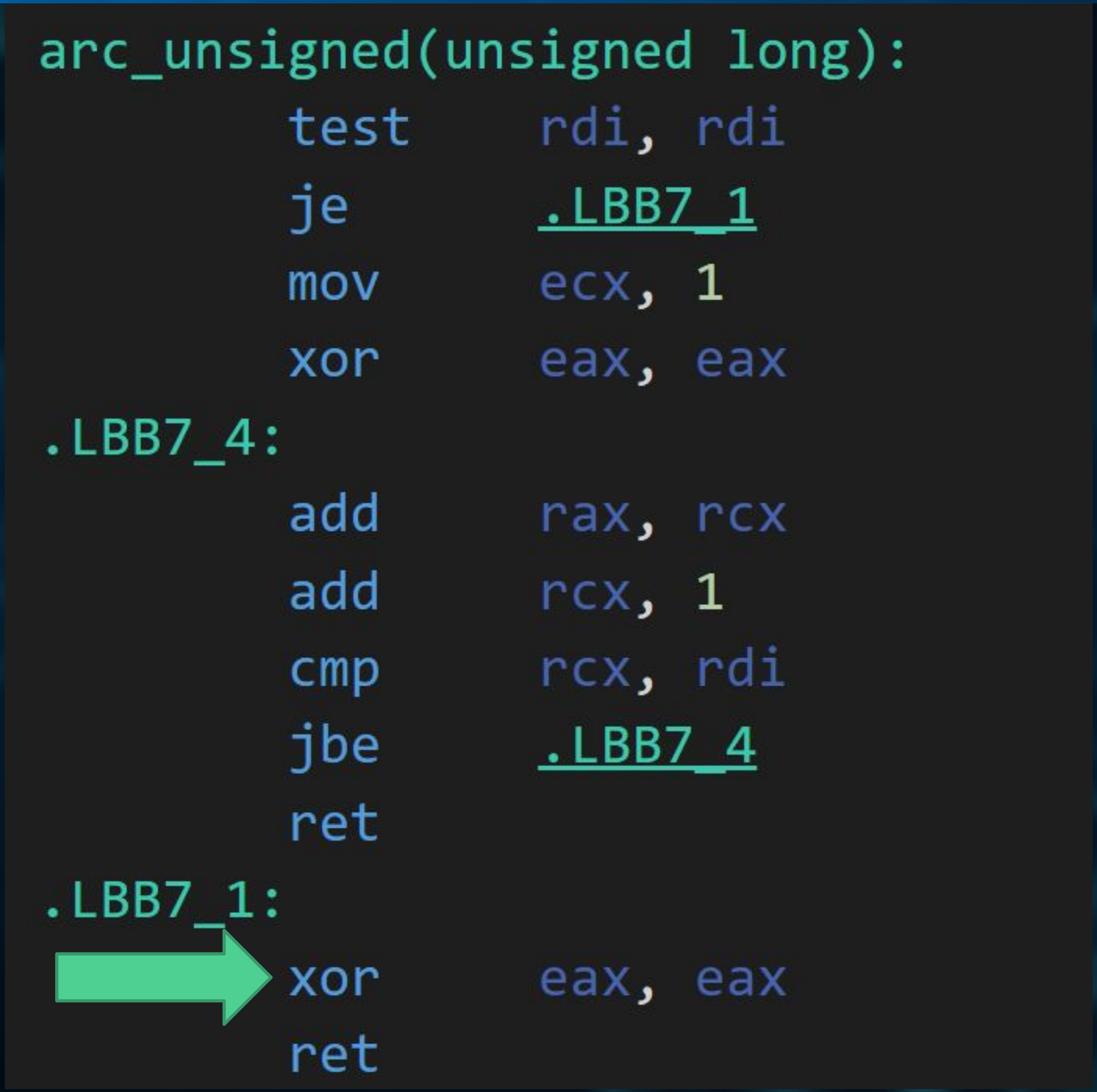
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



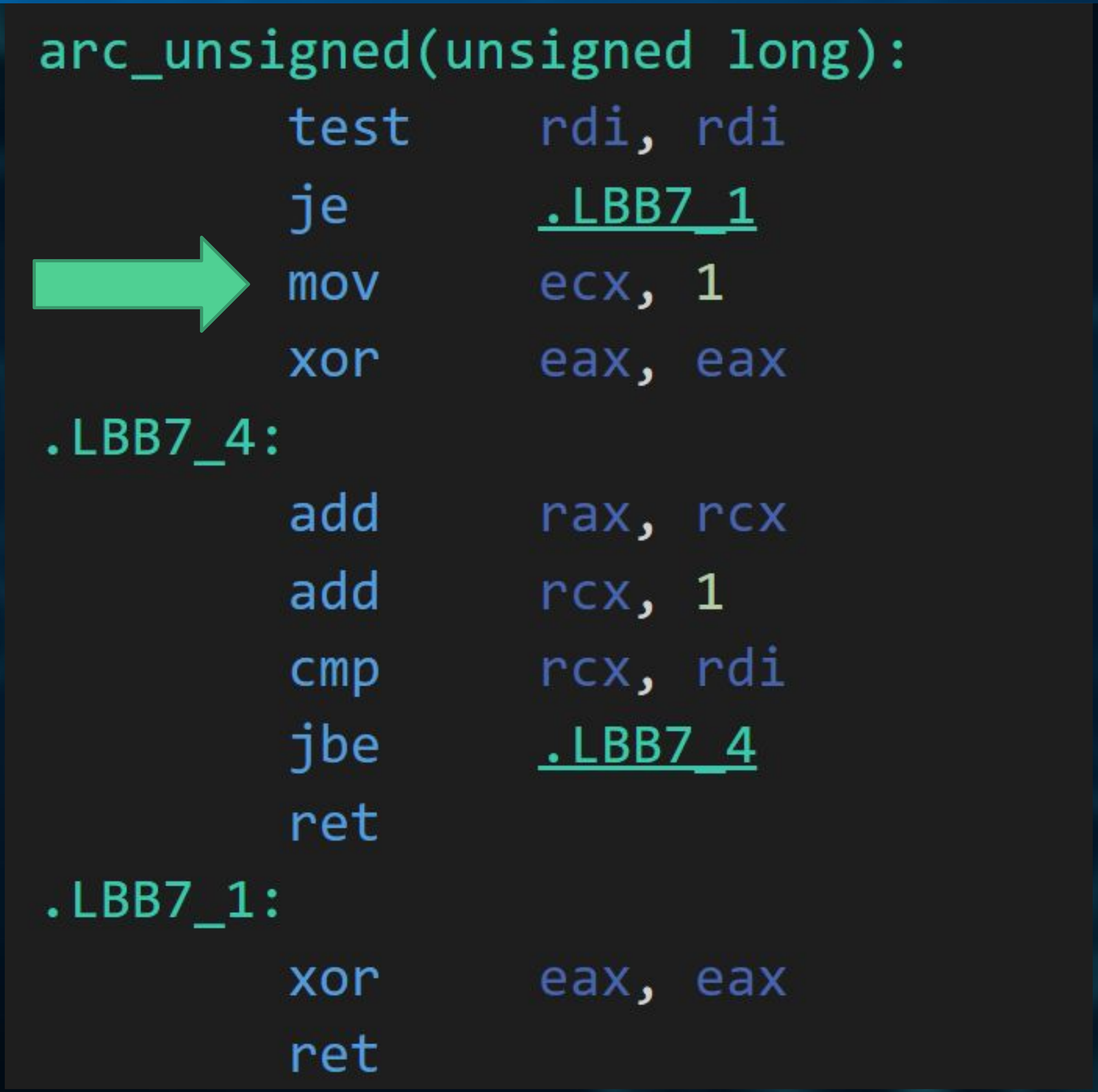
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



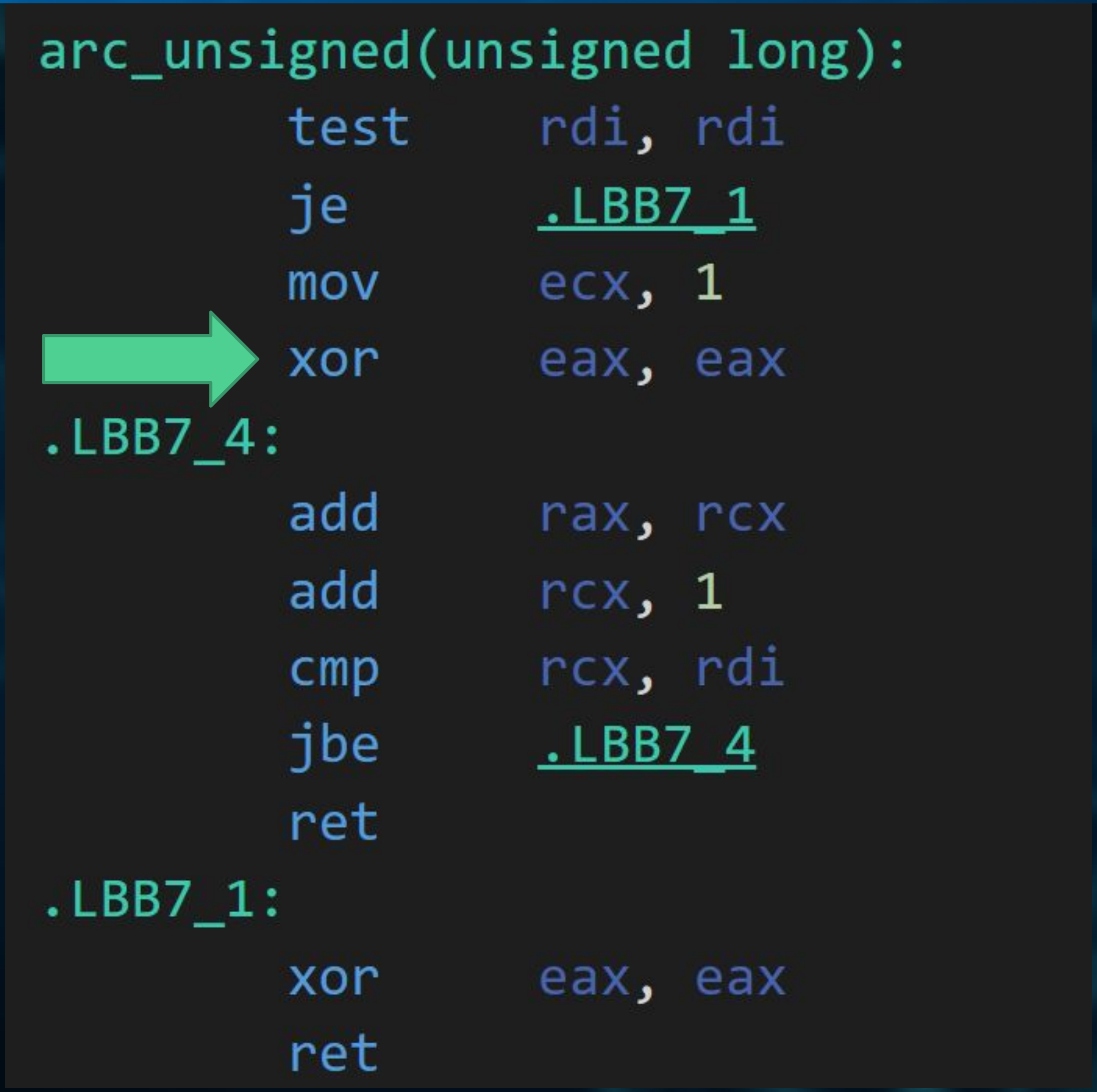
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



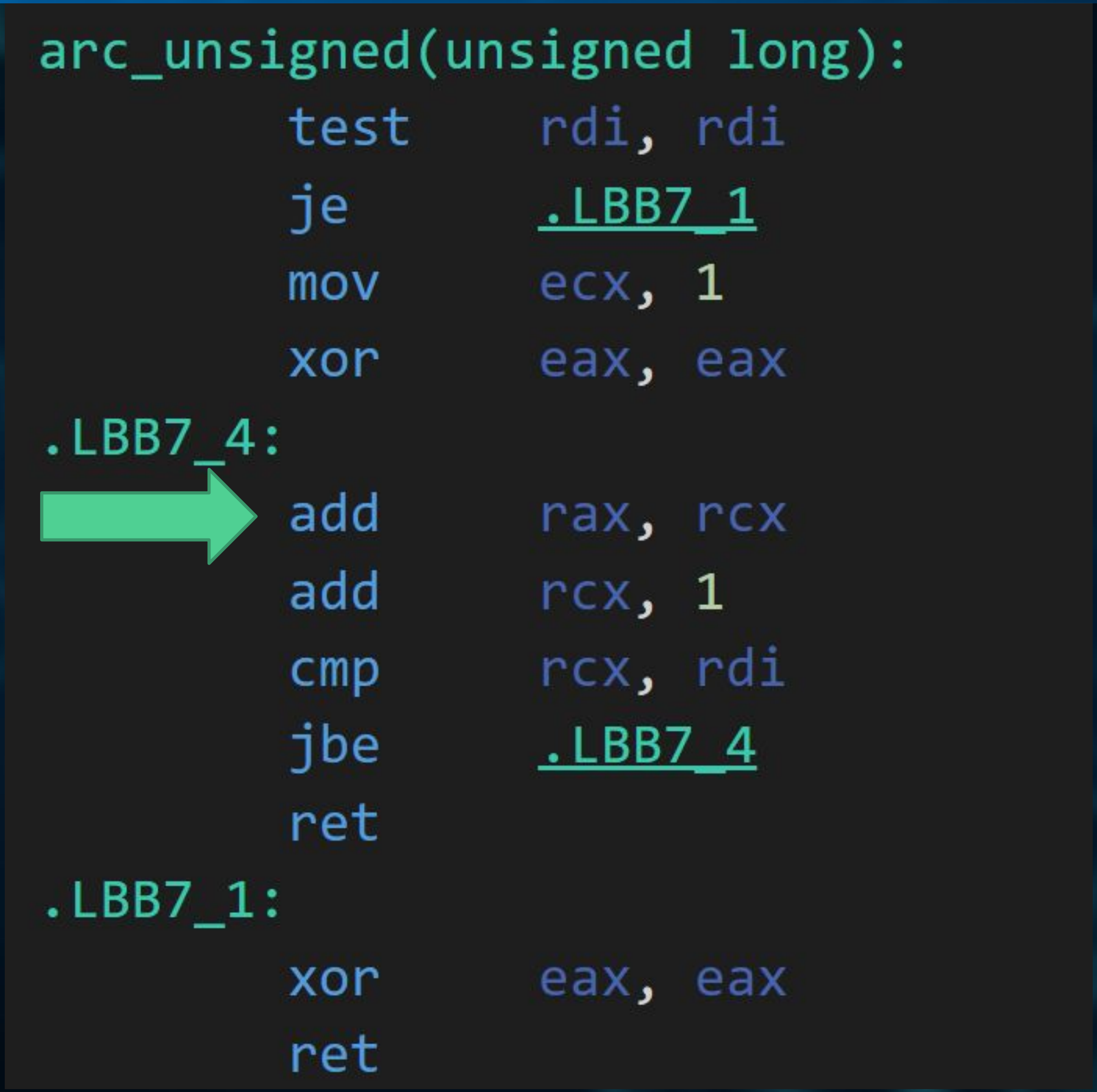
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



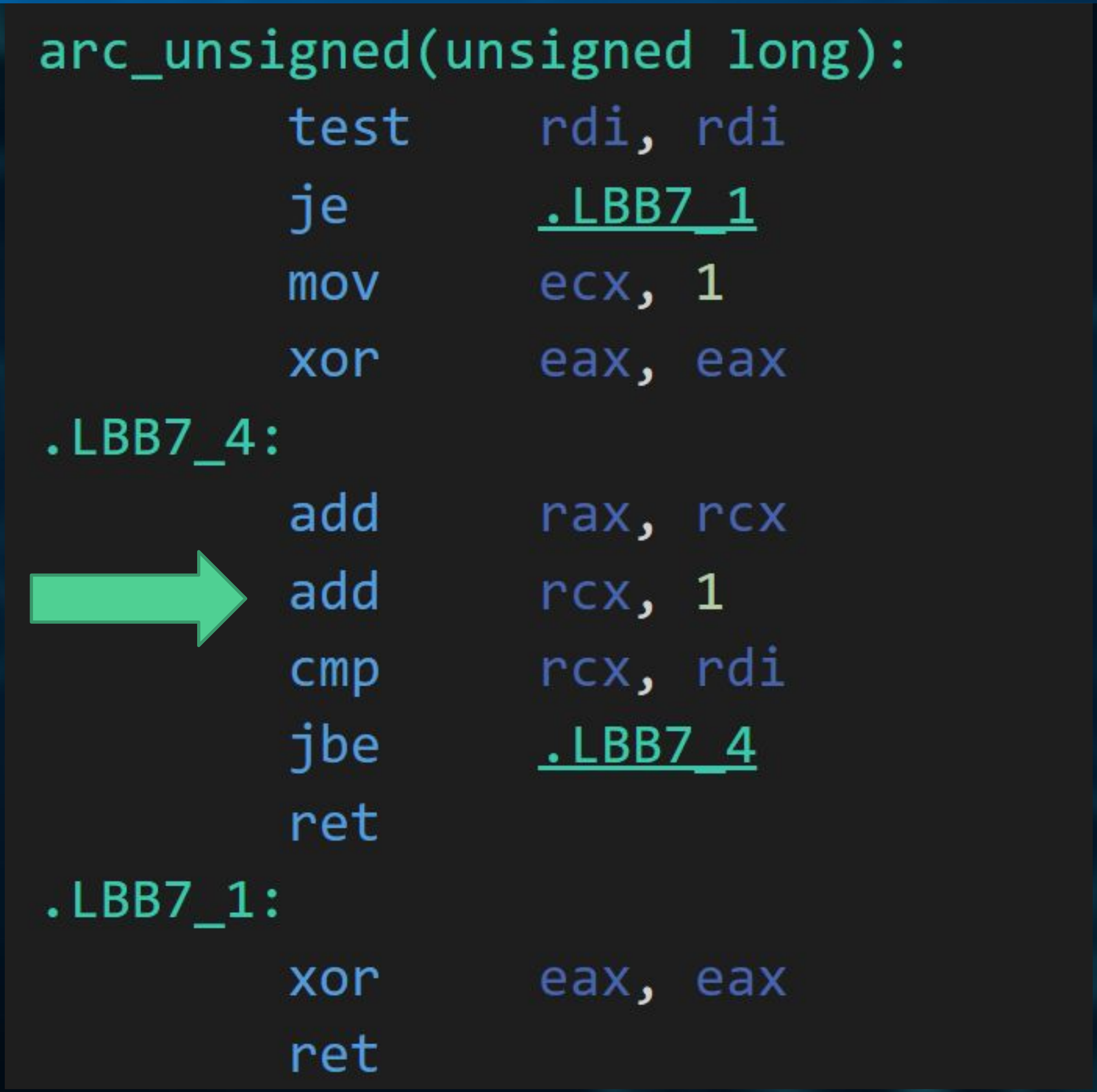
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



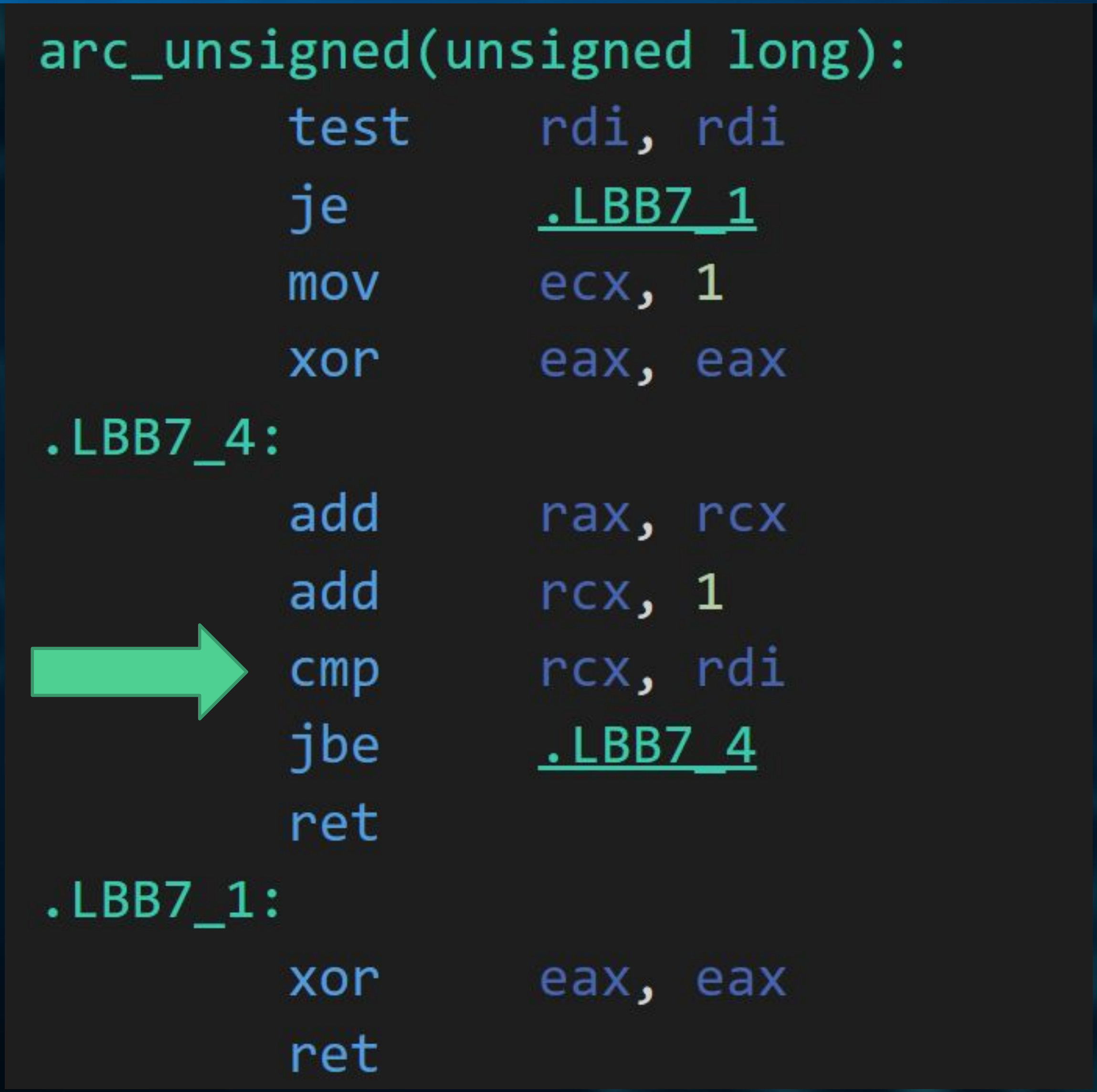
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



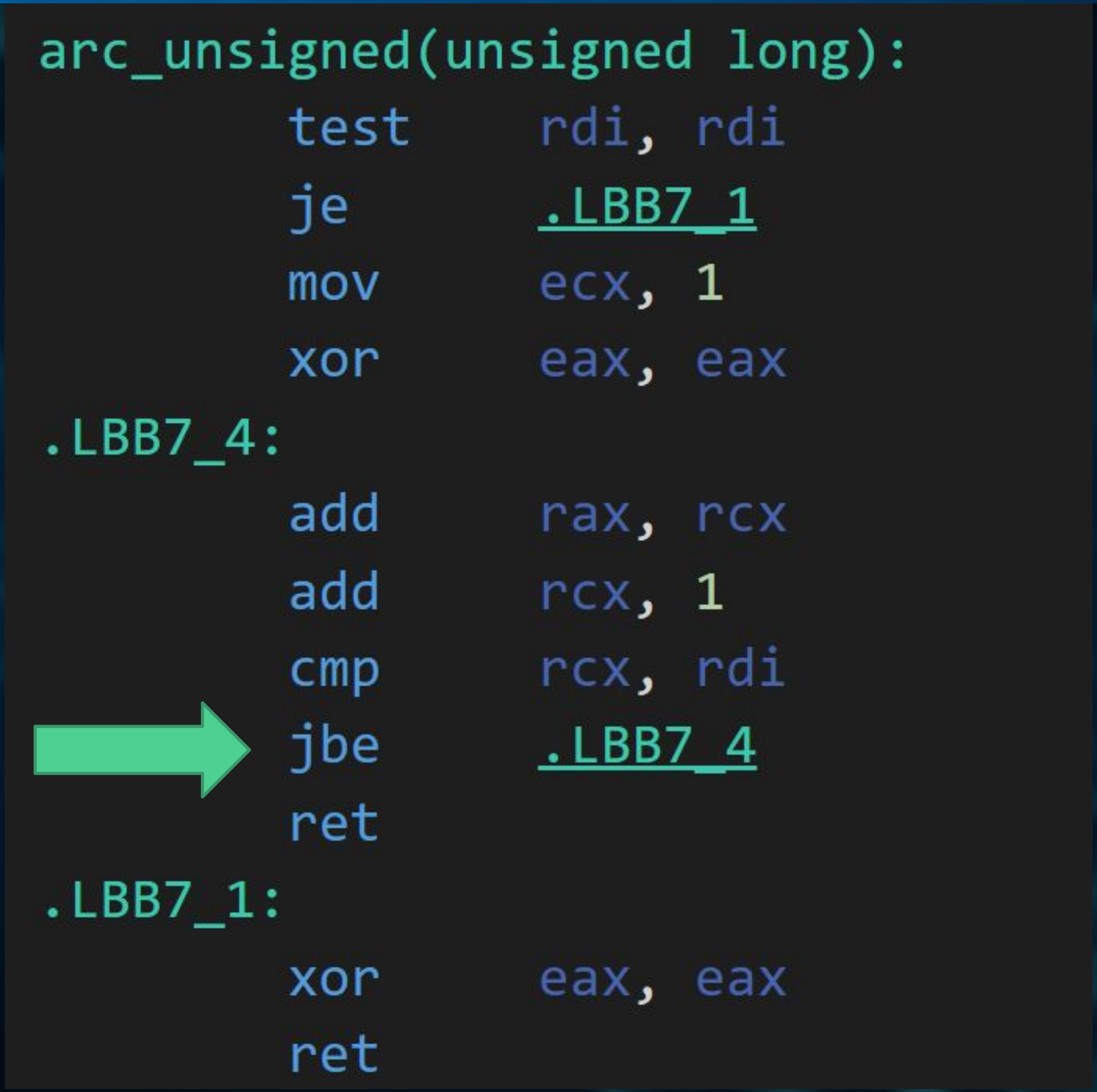
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



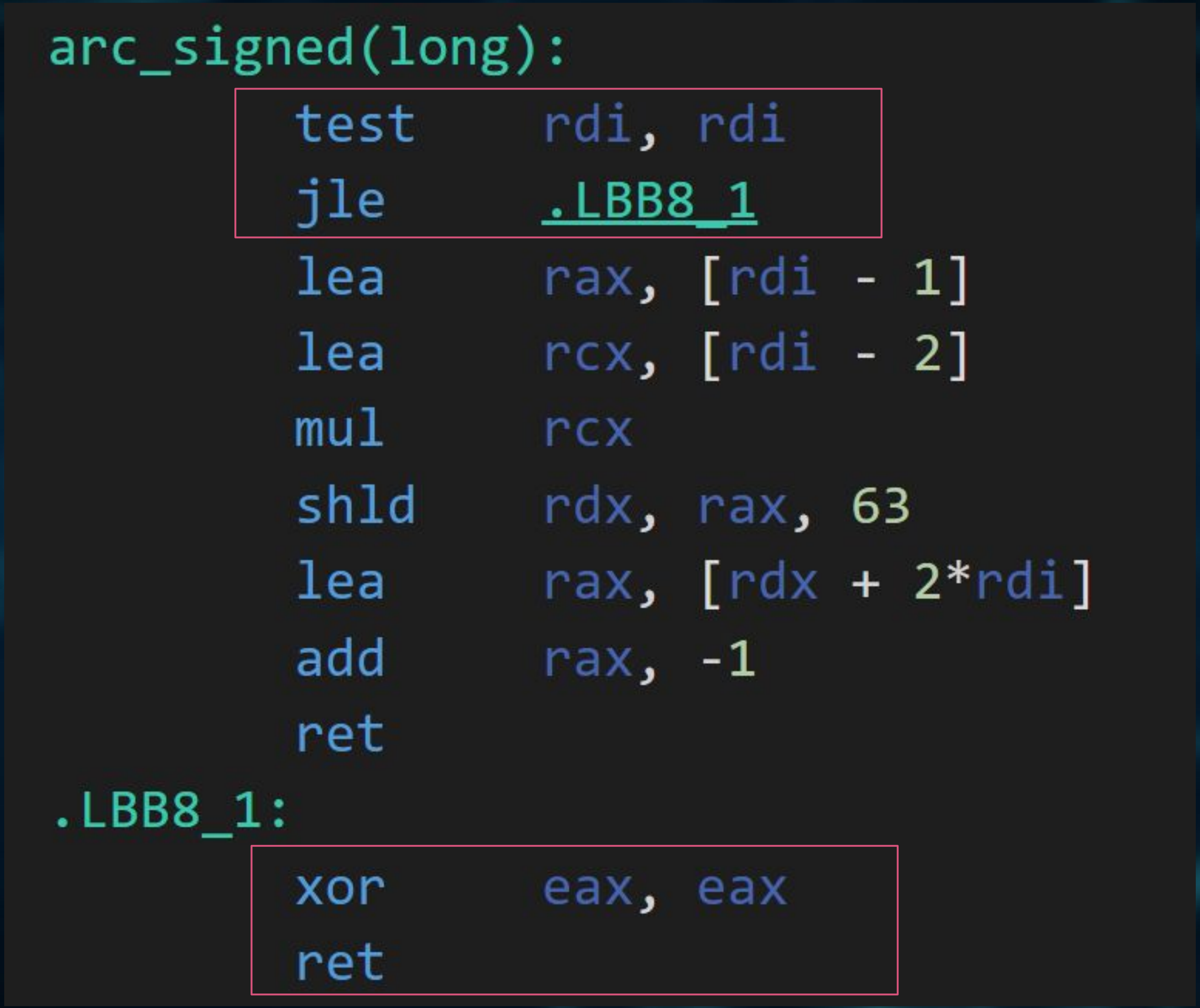
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY



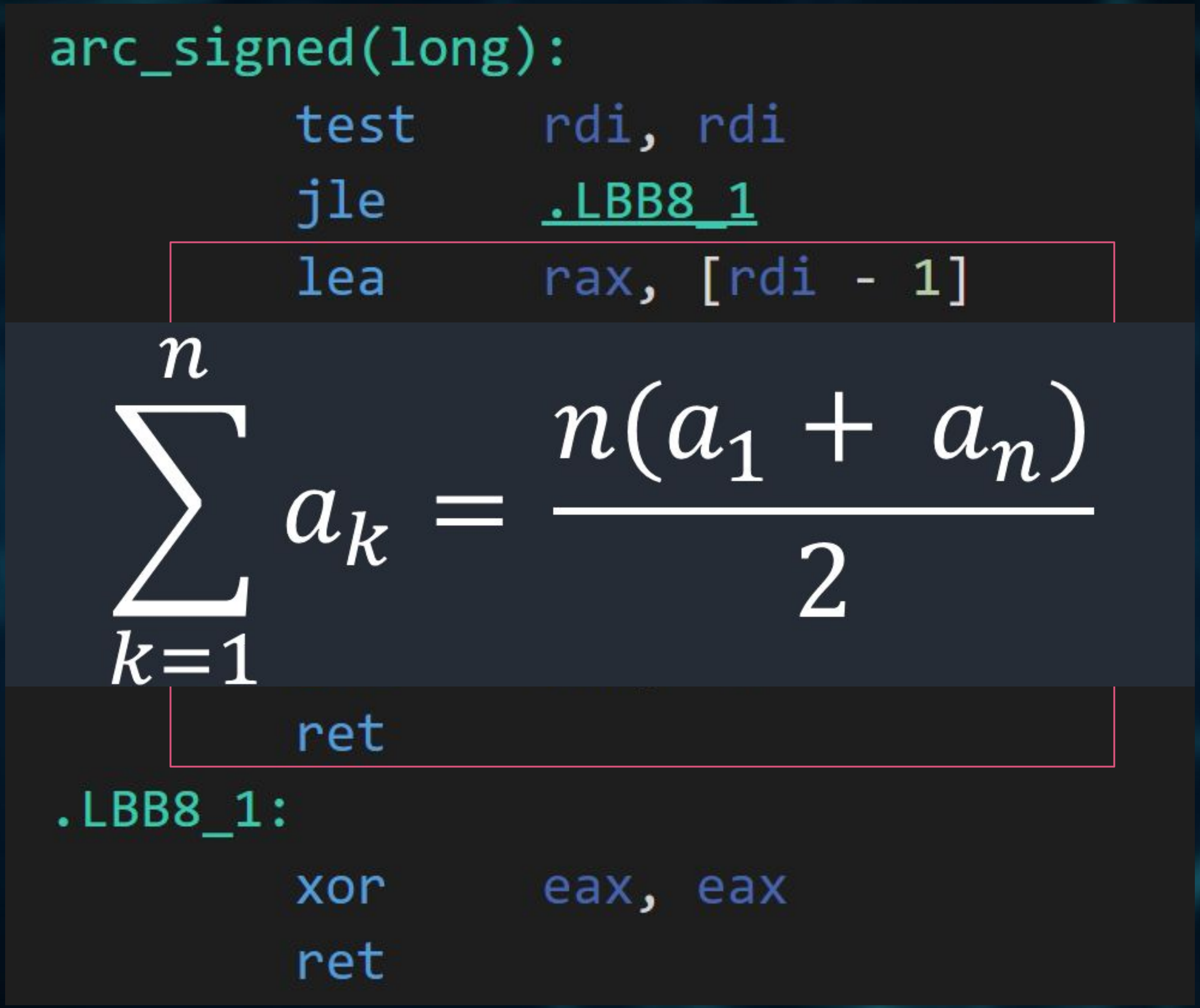
```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB7_1
        mov     ecx, 1
        xor     eax, eax
.LBB7_4:
        add     rax, rcx
        add     rcx, 1
        cmp     rcx, rdi
        jbe     .LBB7_4
        ret
.LBB7_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: SIGNED ASSEMBLY



```
arc_signed(long):
        test    rdi, rdi
        jle     .LBB8_1
        lea     rax, [rdi - 1]
        lea     rcx, [rdi - 2]
        mul     rcx
        shld    rdx, rax, 63
        lea     rax, [rdx + 2*rdi]
        add     rax, -1
        ret
.LBB8_1:
        xor     eax, eax
        ret
```

MORE COMPLEX EXAMPLE: UNSIGNED ASSEMBLY

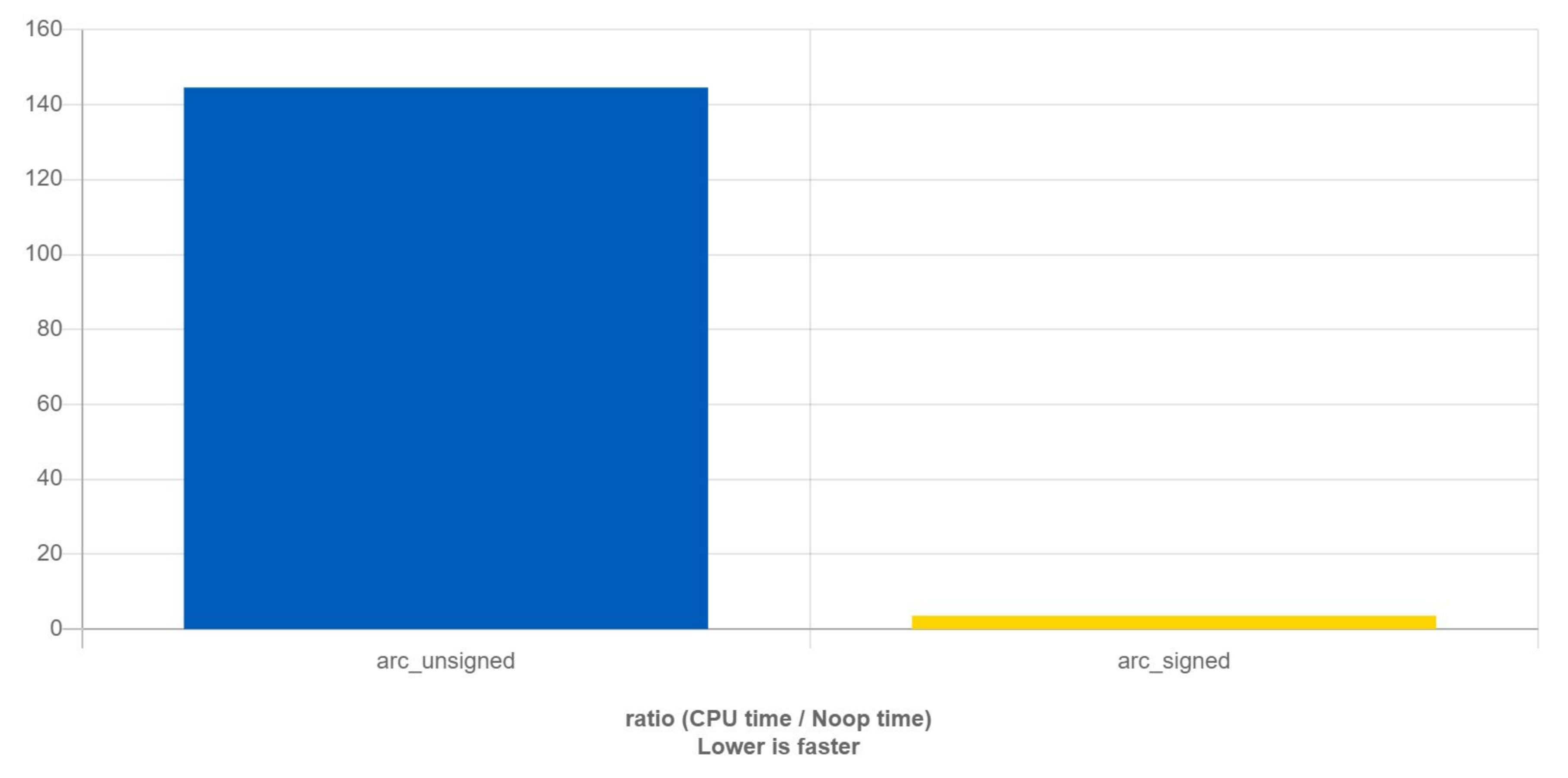


```
arc_signed(long):
        test    rdi, rdi
        jle     .LBB8_1
        lea     rax, [rdi - 1]
```

$$\sum_{k=1}^{n} a_k = \frac{n(a_1 + a_n)}{2}$$

```
        ret
.LBB8_1:
        xor     eax, eax
        ret
```

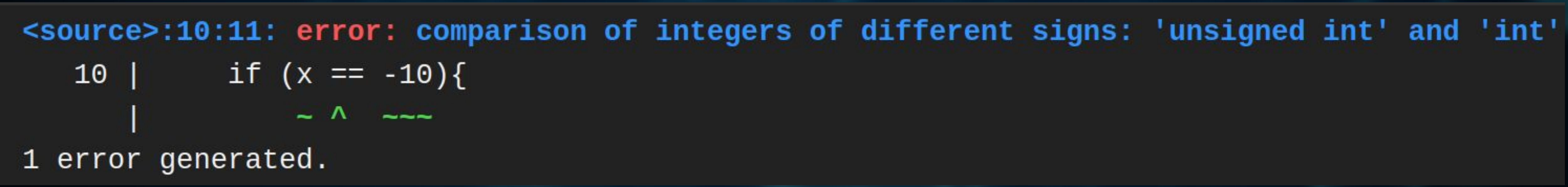
MORE COMPLEX EXAMPLE: PERFORMANCE



MORE COMPLEX EXAMPLE: WHY?

WHAT TO DO?

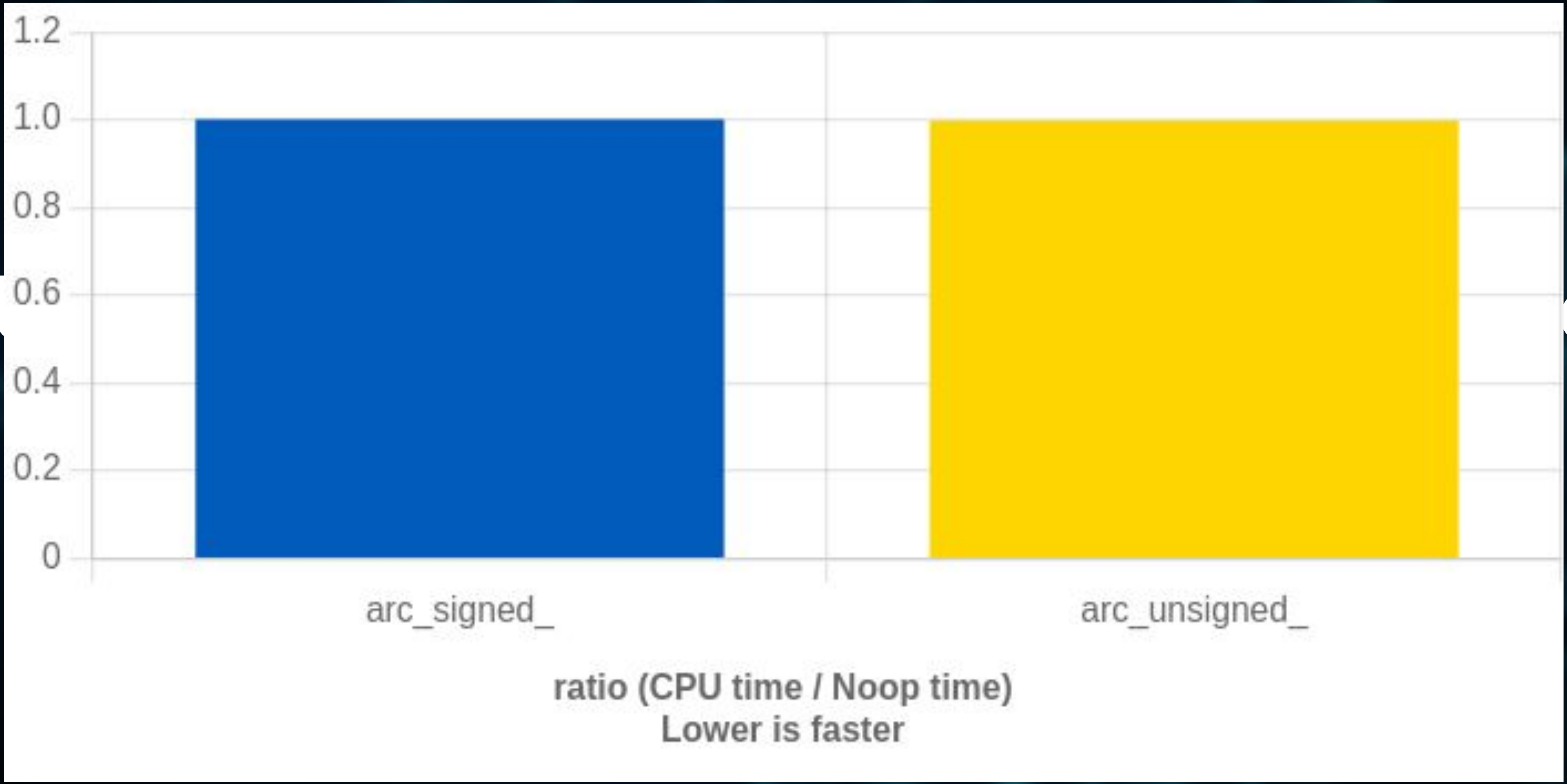
WHAT TO DO?



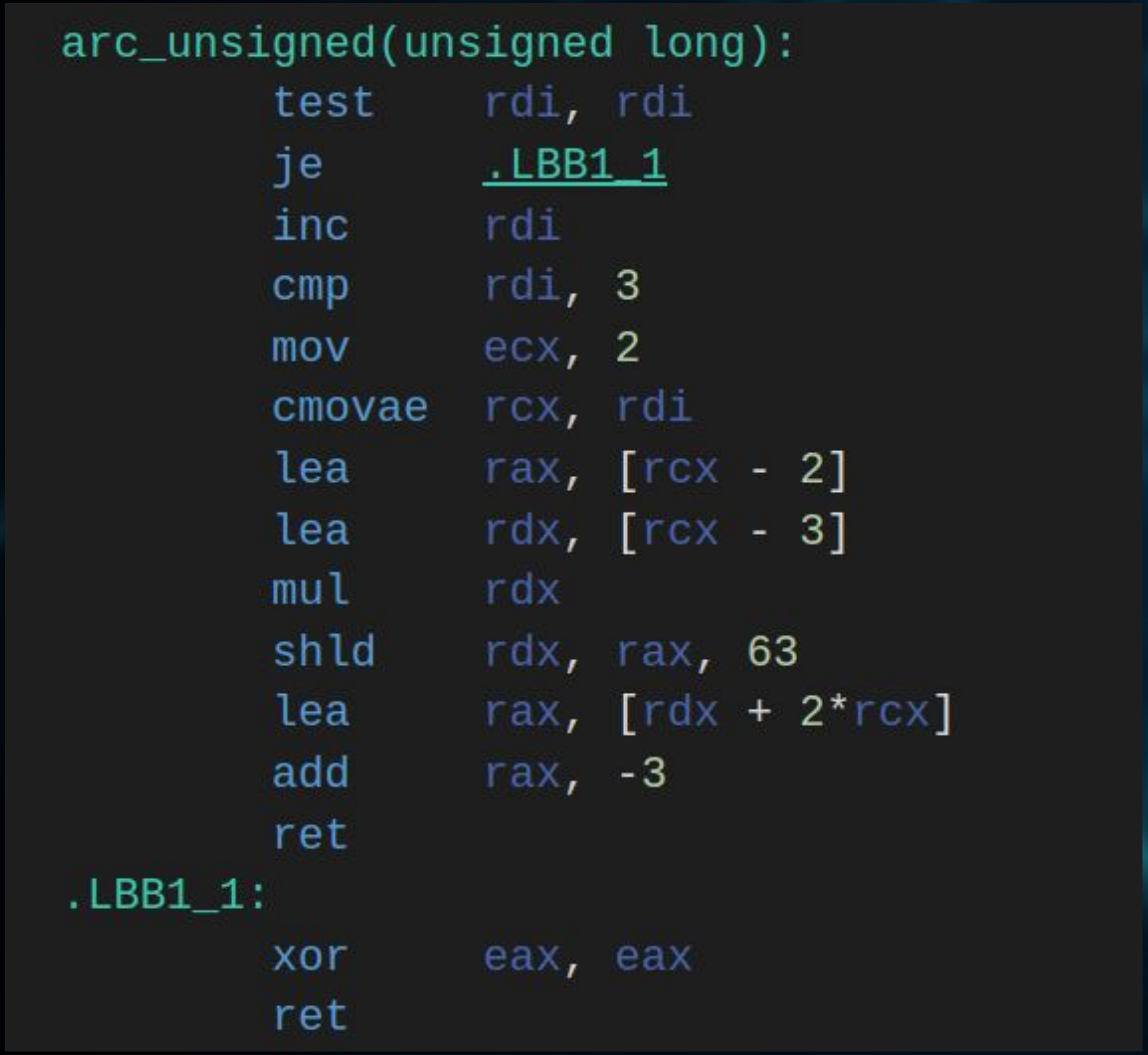
```
<source>:10:11: error: comparison of integers of different signs: 'unsigned int' and 'int'
   10 |     if (x == -10){
      |         ~ ^  ~~~
1 error generated.
```

WHAT TO DO?

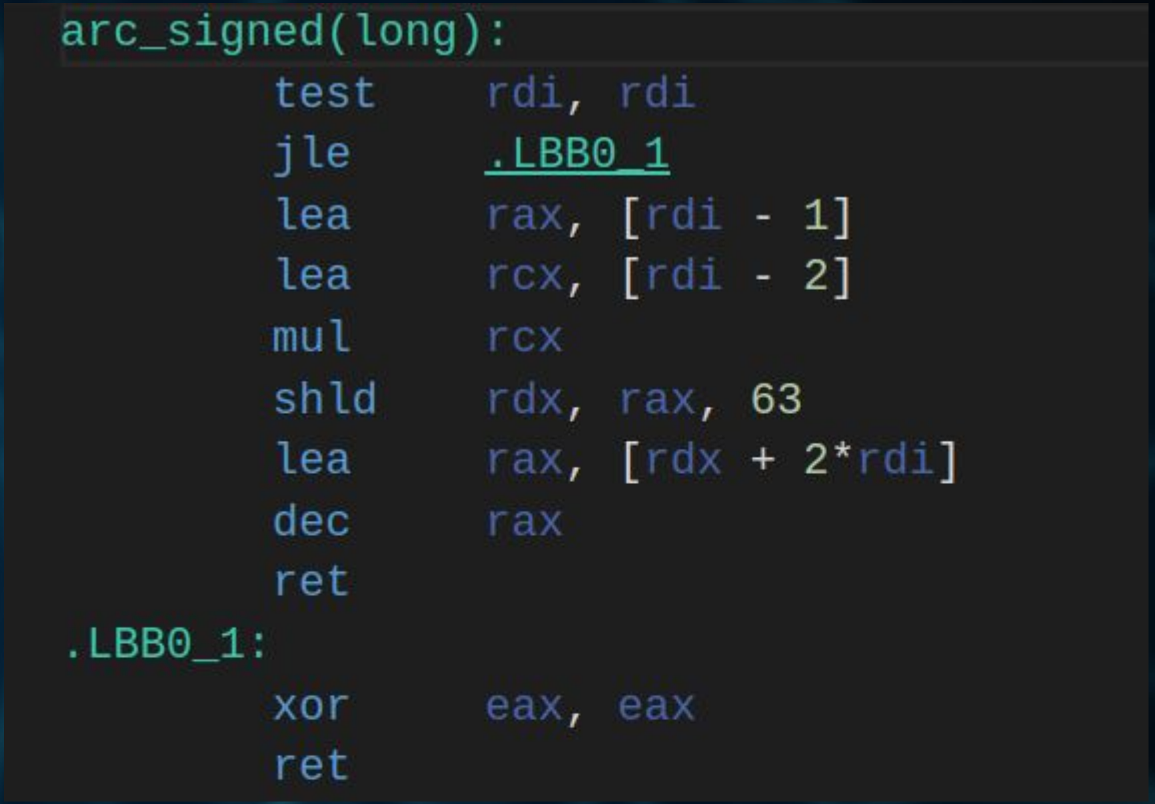
- U er!



WHAT TO DO?



```
arc_unsigned(unsigned long):
        test    rdi, rdi
        je      .LBB1_1
        inc     rdi
        cmp     rdi, 3
        mov     ecx, 2
        cmovae  rcx, rdi
        lea     rax, [rcx - 2]
        lea     rdx, [rcx - 3]
        mul     rdx
        shld    rdx, rax, 63
        lea     rax, [rdx + 2*rcx]
        add     rax, -3
        ret
.LBB1_1:
        xor     eax, eax
        ret
```



```
arc_signed(long):
        test    rdi, rdi
        jle     .LBB0_1
        lea     rax, [rdi - 1]
        lea     rcx, [rdi - 2]
        mul     rcx
        shld    rdx, rax, 63
        lea     rax, [rdx + 2*rdi]
        dec     rax
        ret
.LBB0_1:
        xor     eax, eax
        ret
```

WHAT TO DO?

- Use Sanitizers

```
-fsanitize=signed-integer-overflow
-fsanitize=unsigned-integer-overflow
```

WHAT TO DO?

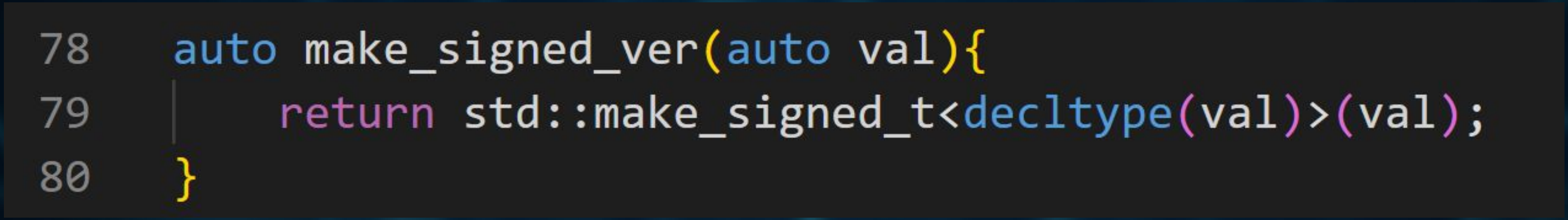
- Use special types for better performance
  - `int_fastN_t, uint_fastN_t`

WHAT TO DO?

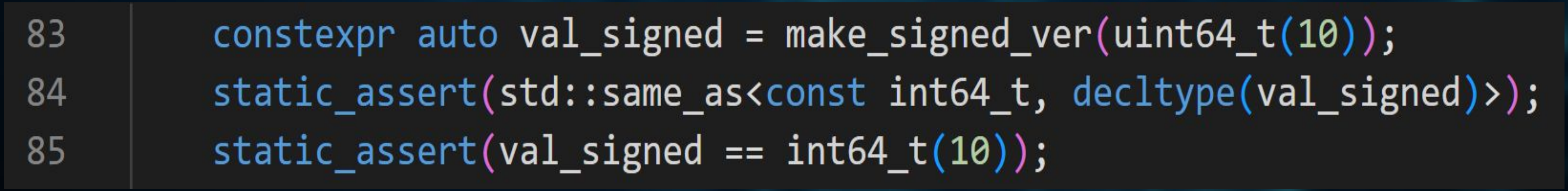
- Know your CPU

WHAT TO DO?

- Use special helpers from the standard
  - MAKE_SIGNED
  - MAKE_UNSIGNED



```
auto make_signed_ver(auto val){
    return std::make_signed_t<decltype(val)>(val);
}
```

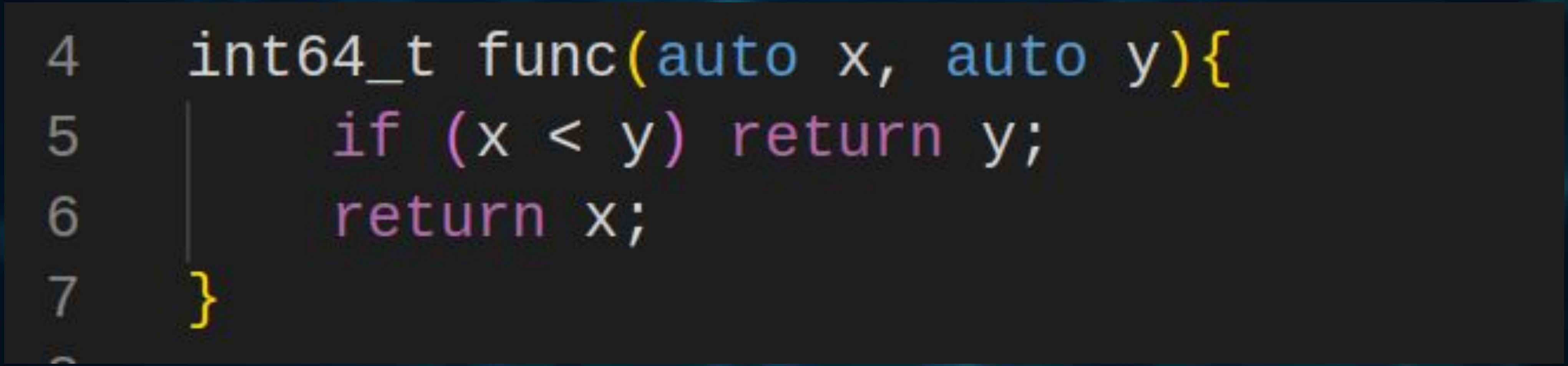


```
    constexpr auto val_signed = make_signed_ver(uint64_t(10));
    static_assert(std::same_as<const int64_t, decltype(val_signed)>);
    static_assert(val_signed == int64_t(10));
```

WHAT TO DO?

- Use C++20 safe comparators
  - `std::cmp_equal: ==`
  - `std::cmp_not_equal: !=`
  - `std::cmp_less: <`
  - `std::cmp_less_equal: <=`
  - `std::cmp_greater: >`
  - `std::cmp_greater_equal: >=`

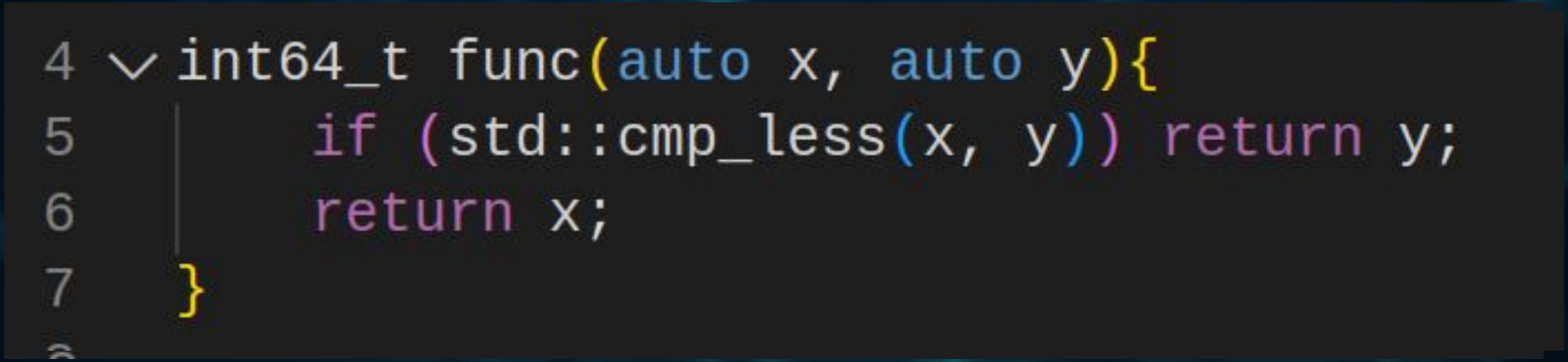
WHAT TO DO?



```
int64_t func(auto x, auto y){
    if (x < y) return y;
    return x;
}
```

func(-10, 20ul) —> -10

WHAT TO DO?



```
int64_t func(auto x, auto y){
        if (std::cmp_less(x, y)) return y;
        return x;
}
```

func(-10, 20ul) —> 20

WHAT TO DO?

- Avoid using auto when not sure about the type

WHAT TO DO?

- Avoid using auto when not sure about the type
- Use concrete types when possible!

WHAT TO DO?

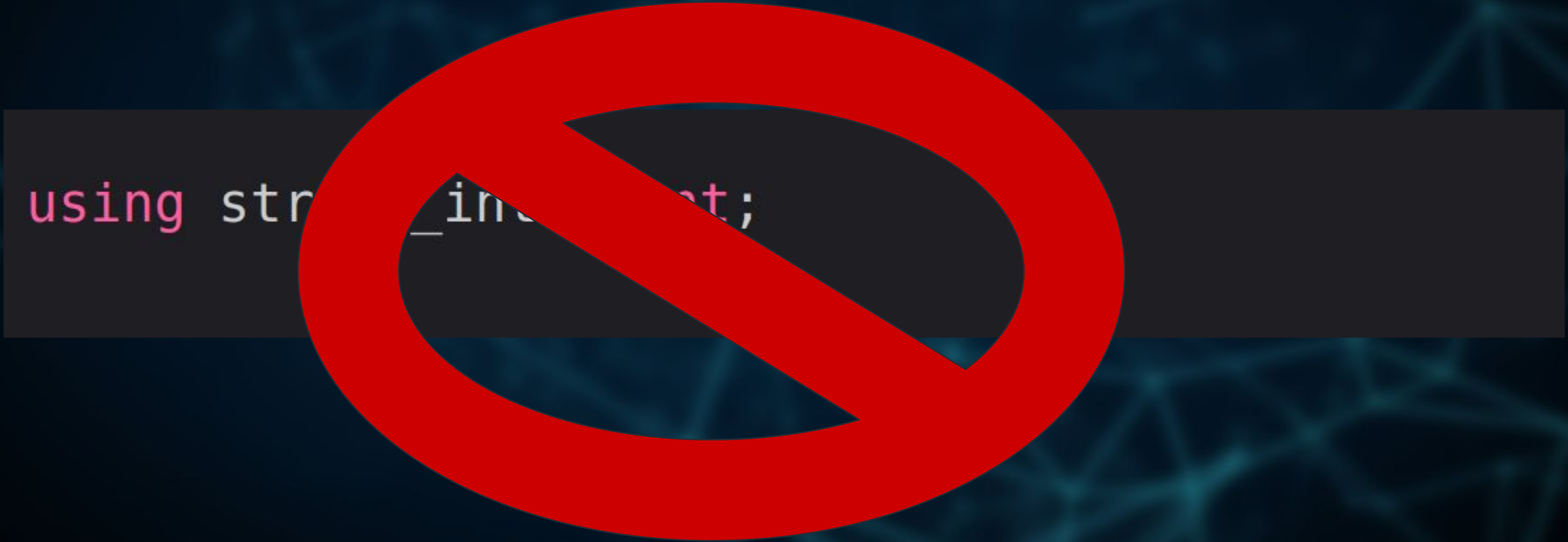
- Avoid using auto when not sure about the type
- Use concrete types when possible!
- Use modern loops as much as you can

WHAT TO DO?

- Avoid using auto when not sure about the type
- Use concrete types when possible!
- Use modern loops as much as you can
- Use strong types.

WHAT TO DO?

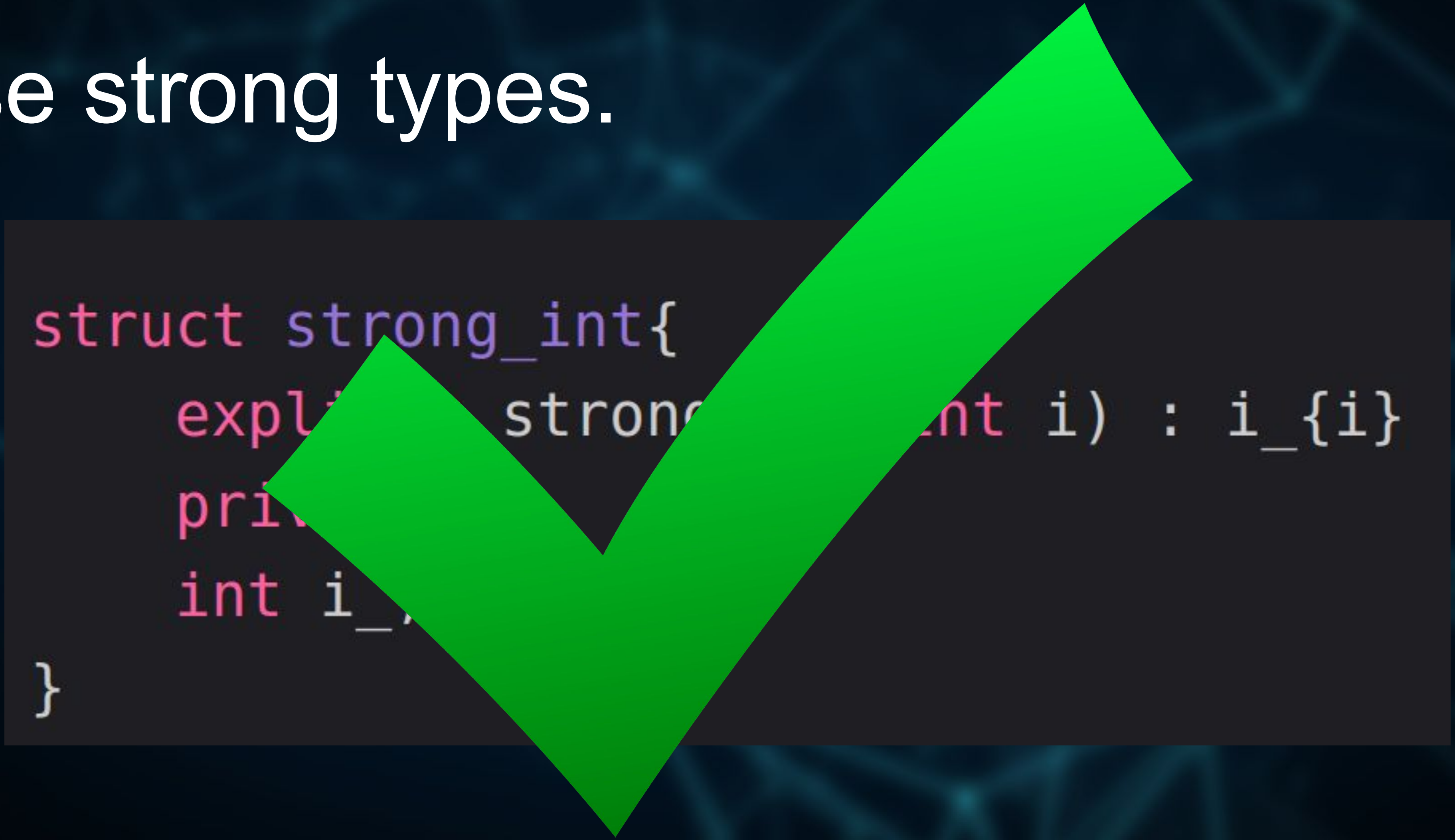
- Use strong types.



```
using str    _in     t;
```

WHAT TO DO?

- Use strong types.



```
struct strong_int{
    expl      strong      nt i) : i_{i}
    pri
    int i_
}
```

QUESTIONS

?

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

THANK YOU FOR LISTENING

ALEX DATHSKOVSKY

+97254-7685001

ALEX.DATHSKOVSKY@SPEEDATA.IO

WWW.LINKEDIN.COM/IN/ALEXDATHSKOVSKY

Link to presented code: https://godbolt.org/z/W6zvzMzv7