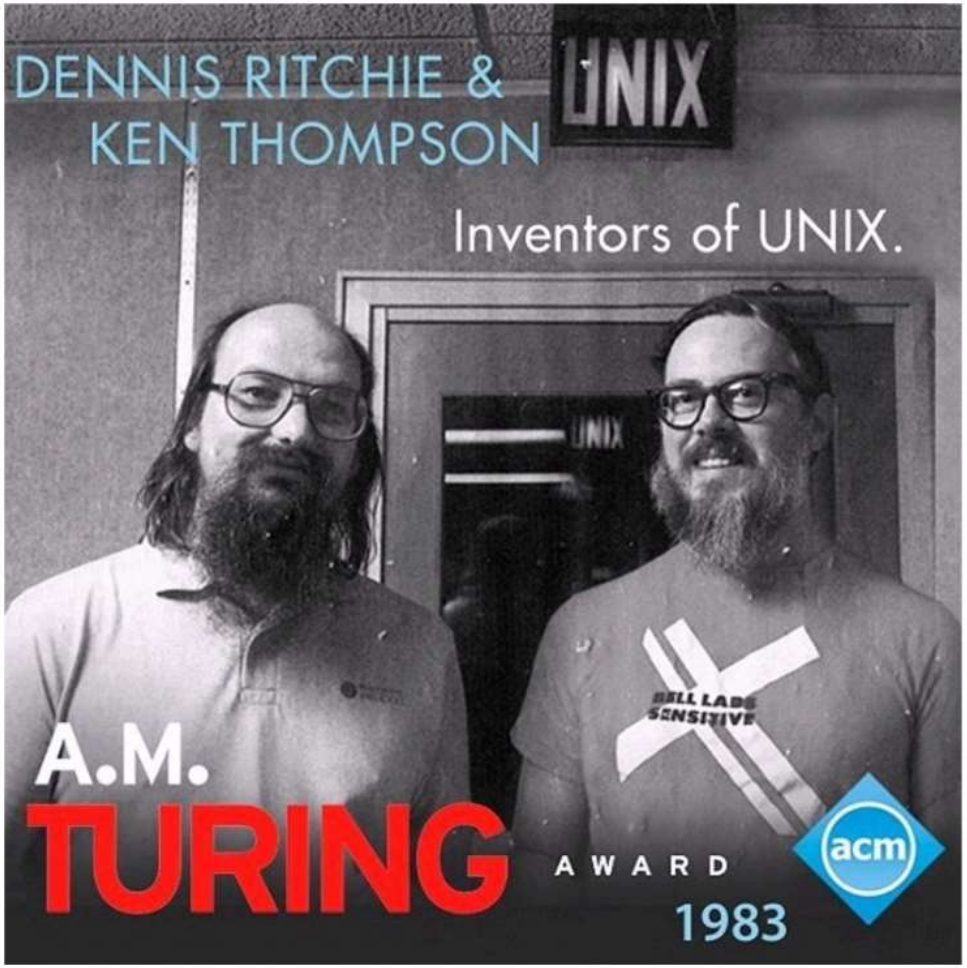
Operating Systems & C

Lecture 5: C

Willard Rafnsson
IT University of Copenhagen

ITU CPH

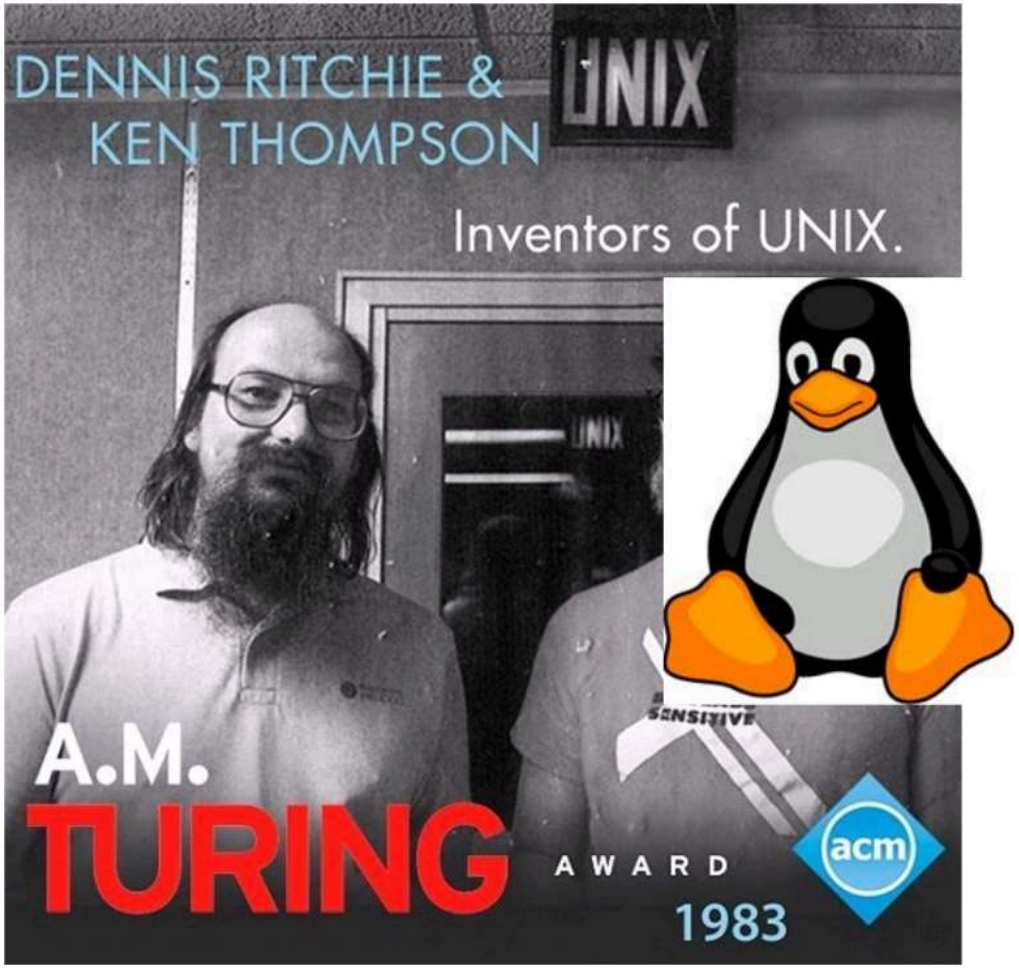
C: history & importance



ACM citation:

- Success of UNIX stems from its **tasteful selection** of a **few key ideas** and their **elegant implementation**. The model of the Unix system led a generation of software designers to new ways of thinking about programming. The genius of Unix is its framework, which **enables programmers to stand on the work of others**.
- **Ken Thompson** also **created an interpretive language called B**, based on BCPL, which he used to re-implement the non-kernel parts of Unix. **Ritchie added types** to B, and later created a compiler for **the C language**. Thompson and Ritchie rewrote most of Unix in C in 1973; made further development and porting to other platforms much easier.

C: history & importance



ACM citation:

- Success of UNIX stems from its **tasteful selection** of a **few key ideas** and their **elegant implementation**. The model of the Unix system led a generation of software designers to new ways of thinking about

e genius of Unix is its
h **enables programmers to**
k of others.
lso **created an interpretive**
3, based on BCPL, which he
nent the non-kernel parts
dded types to B, and later
created a compiler for **the C language**.
Thompson and Ritchie rewrote most of Unix in C in 1973; made further development and porting to other platforms much easier.

C: key features

imperative language

statically typed (albeit permissively)

minimal language:

- **efficient** mapping to assembly code (this lecture)
- all interesting stuff done by libraries (even printf)

minimal run-time support:

- **explicit** memory management (pointers, malloc)
- **explicit** threads programming (next lecture)
- **efficient** mapping to assembly code (this lecture)

Programming Language Concepts,
Ch. 7 (esp. 7.5)
"Programs as Data" course

C language itself very minimal.
you must be acquainted w/ C libraries.

C: spirit

(a) trust the programmer.
(b) do not prevent the programmer from doing what needs to be done.
(c) keep the language small and simple.
(d) provide only one way to do an operation.
(e) make it fast, even if not guaranteed to be portable.
(f) make support for safety & security demonstrable

C: why learn it?

"C has the power of assembly language and the convenience of … assembly language. [...]

C is **quirky**, **flawed**, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments. "

- D. Ritchie http://csapp.cs.cmu.edu/3e/docs/chistory.html

C is a mess. syntactic sugar on top of assembly (Linus Torvalds quote)
why learn C: go-to language for systems programming.
you care about *security*, *performance*, *resource utilization*? learn C.
high-level PLs abstract away many issues.
learn a PL each semester. this semester: C.

UNIVERSITY OF COPENHAGEN

ITU CPH

C *primer*, not a lecture/tutorial

we assume you are comfortable with another imperative programming language.
so we assume familiarity with:

- expressions, operators, numbers, characters, strings, arrays,
- statements, blocks, conditionals, loops, variables, scope,
- functions, parameters, modules, libraries, …

in other words, that you can easily grasp what is going on over there →

UNIVERSITY OF COPENHAGEN

```c
#include <stdio.h>
#define N 20

int fibs[N];

void computefibs() {
  fibs[0] = 0;
  fibs[1] = 1;
  for ( int i = 2; i < N; i++ ) {
    fibs[i] = fibs[i-1] + fibs[i-2];
  }
}

void printfibs() {
  for ( int i = 0; i < N; i++ ) {
    printf("%d\n", fibs[i]);
  }
}

int main() {
  computefibs();
  printfibs();
  return 0;
}
```

-UU-:----F1 **fibs.c** All (1,

ITU CPH

C: Types

ITU CPH

Type specifiers: numeric

ble 2. C Numeric Types

Type name	Usual size	Values stored	How to declare
char	1 byte	integers	`char x;`
short	2 bytes	signed integers	`short x;`
int	4 bytes	signed integers	`int x;`
long	4 or 8 bytes	signed integers	`long x;`
long long	8 bytes	signed integers	`long long x;`
float	4 bytes	signed real numbers	`float x;`
double	8 bytes	signed real numbers	`double x;`

also provides *unsigned* versions of the integer numeric types (`char` , `short` , `int` , `long` , and `long long`). To declare riable as unsigned, add the keyword `unsigned` before the type name. For example:

```
int x;          // x is a signed int variable
unsigned int y; // y is an unsigned int variable
```

ITU CPH

C pointers

that's really all.
small level of abstraction on top of `mov`:

"A pointer is a variable that contains [an] address."
- K & R

useful for:

- explicit memory management (data placement)
- share data w/o copies
- indirection

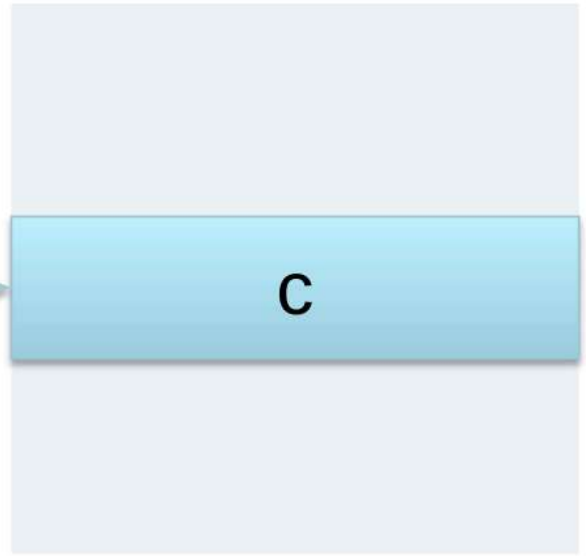
(i.e. "give me the next byte")
(great power over memory access)
(great responsibility; might not have access

Notation

p is a char pointer. It contains an address (of a char variable).

```
char *p;
char c;
p = &c;
```

(char *) &c



address-of

Q: how many bytes (or bits) to represent a pointer?

can also write char* p;
Linux: next to var n

Exercise

write type declarations for the following variables:

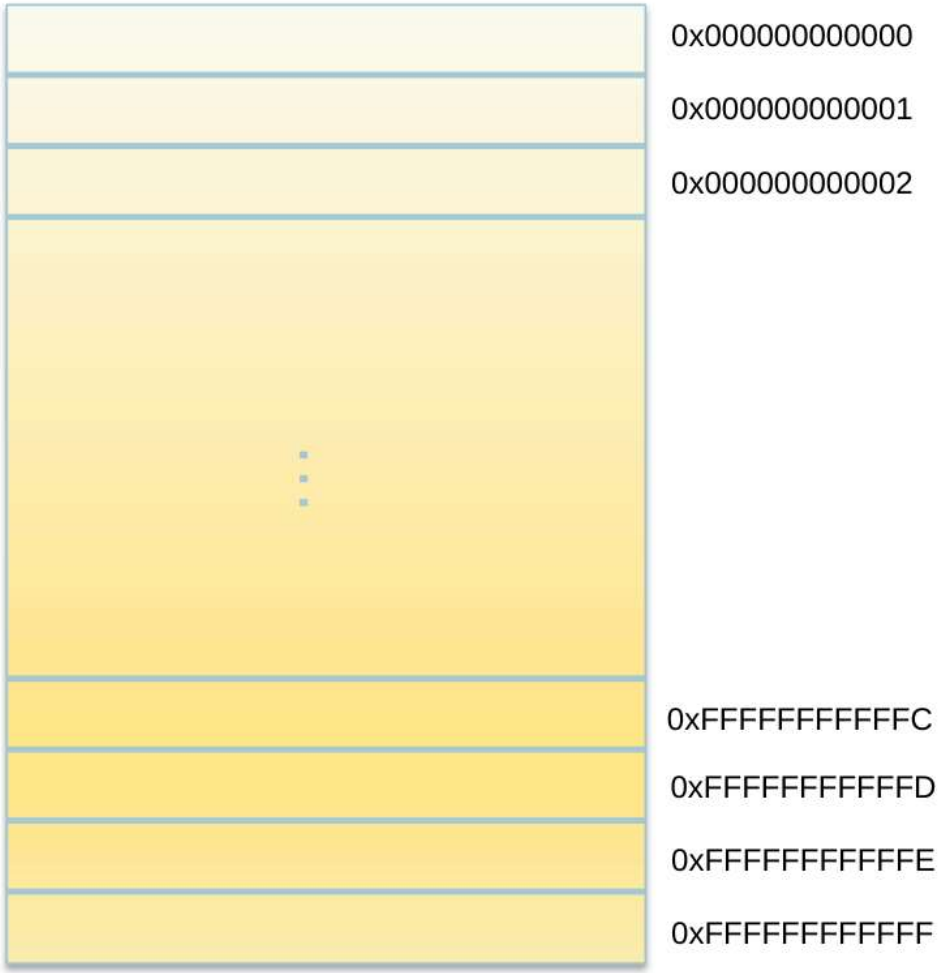
- q : a pointer to an integer pointer
- t : a pointer to a byte
- u : a pointer to a byte array

Q: spend 1-2 mins on this

How does main memory work?

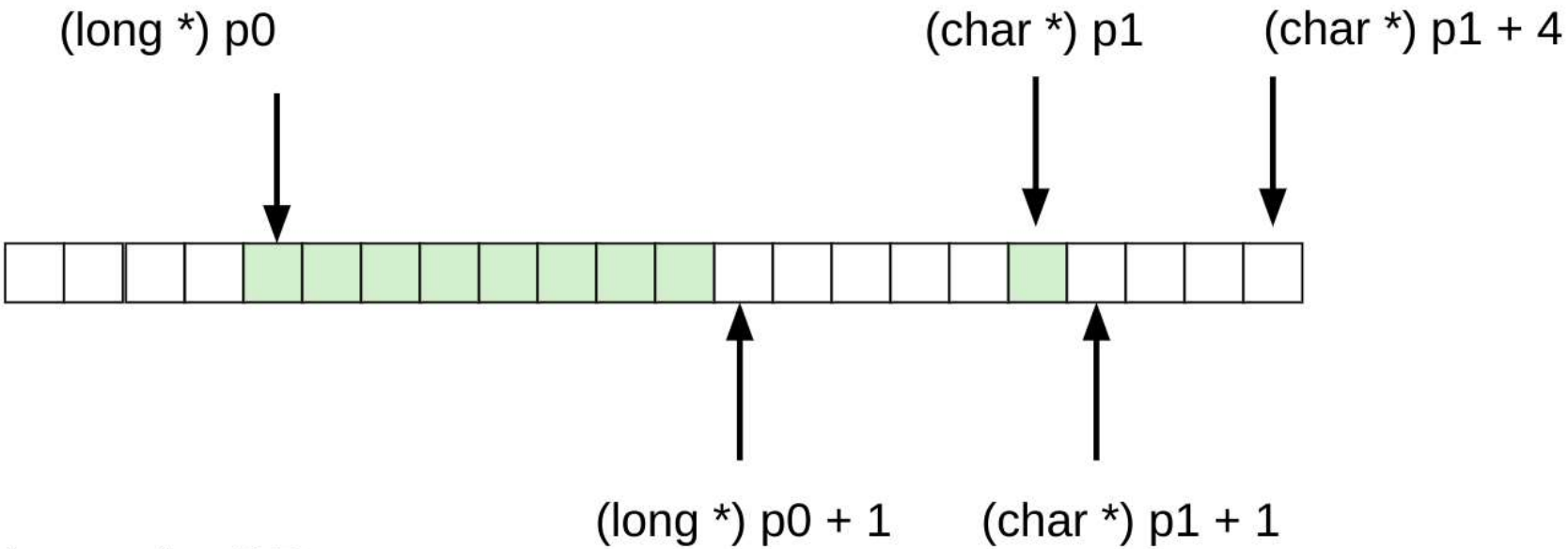
- **sequence of bytes**.
  1 byte = 8 bits
- each byte has unique **address**.
- address space is **linear**.

technology:
-DRAM, SRAM: transient
-3D Xpoint: persistent



Pointer Arithmetic

recall: 1 cell is 1 byte.
memory is byte-addressable.

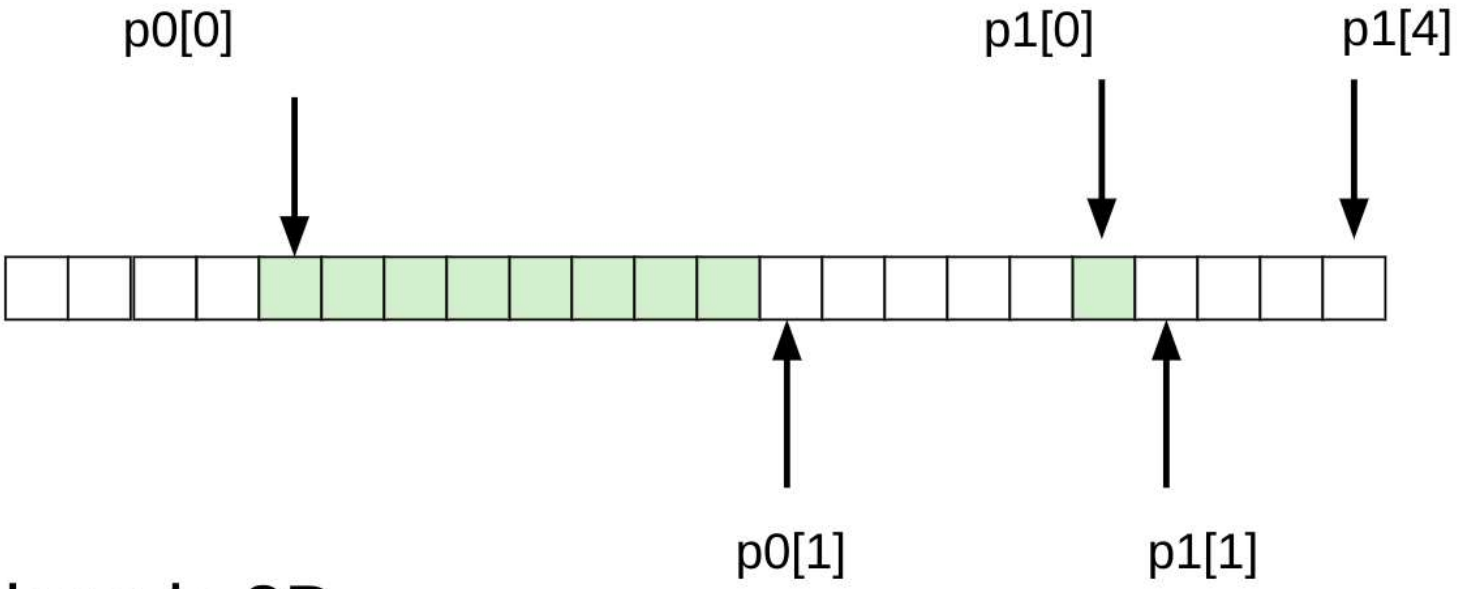


A long is 8B
A char is 1B

you can do arith. on addr.
how far you skip, depends on type of pointer.

Pointer Arithmetic

equivalent to this;
arrays are just pointers
(and a little more)



A long is 8B
A char is 1B

Notation:
a[i] is equivalent to *(a+i)

UNIVERSITY OF COPENHAGEN

16. ITU CPH

Array Allocation

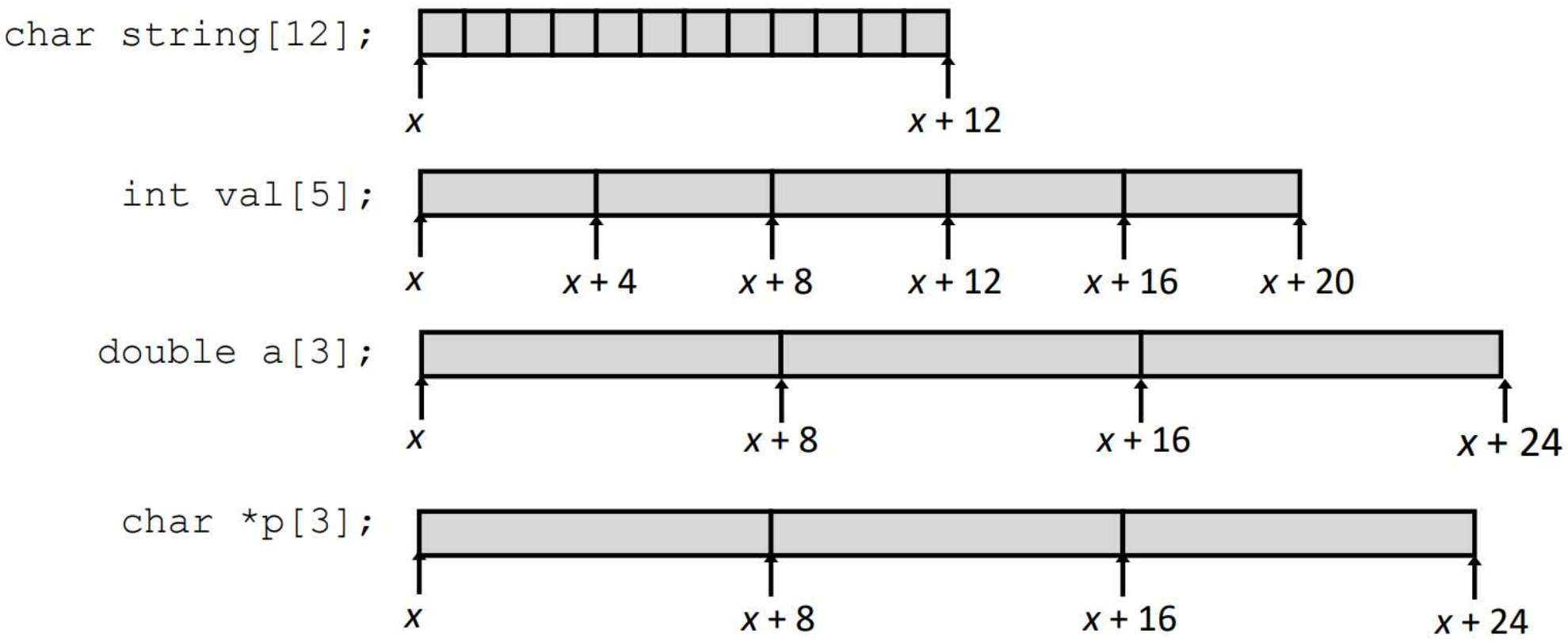
Basic Principle

T **`A[`***L***`];`**

A is an Array of data type *T* and length *L*

Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes

Array Allocation



Array Access

int **A[5] = {0, 1, 2, 3, 4};**

Array of data type *int* and length *5*

Identifier **A** can be used as a pointer to array element 0: Type *int**

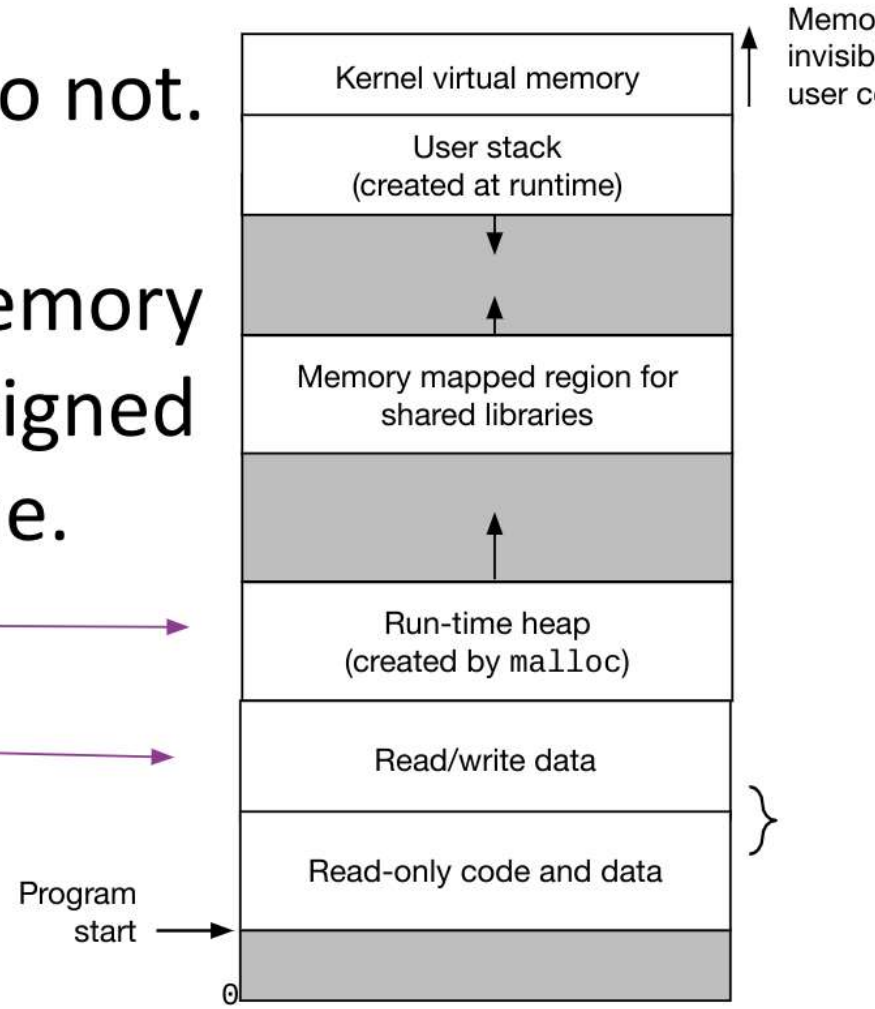
Reference	Type	Value
val[4]	**int**	**4**
val	**int ***	*x*
val+1	**int ***	*x* + 4
&val[2]	**int ***	*x* + 8
val[5]	**int**	??
***(val+1)**	**int**	**1**
val + *i*	**int ***	*x* + 4 *i*

Pointers are NOT arrays

(1) arrays have `size`; pointers do not.

(2) arrays assigned address in memory at compile time; pointers are assigned an address in memory at run time.

```
int* a;
int b[10];
a = b;
```



a now points to &b[0], `size` info lost

must specify array size, so compiler can allocate space at compile time

Restrictions

You can't have:

- A function that returns a function

  Never foo()()
- A function that returns an array

  Never foo()[]
- An array of function

  Never foo[]()

(type system doesn't allow it,
despite conceptually making sense)

Instead...

But you *can* have

- A function returning a pointer to a function

  *fun() ()
- A function returning a pointer to an array

  *fun()[]
- An array of function pointers

  *foo[]()

(due to restrictions in C type system,
we use pointers as an indirection-level)

Type specifiers: struct

a **record**

struct: a bunch of data items grouped together (in memory)

```
struct tag {
    type_1 identifier_1;
    type_2 identifier_2;
    ...
    type_N identifier_N;
} ;
struct tag variable_name;
```

Type specifier: struct

data items accessed through dot operator.
when using *pointer to struct*,
data items accessed through arrow operator.

```
/* struct that points to the next struct */
struct node_tag {
    int datum;
    struct node_tag *next;
};
struct node_tag a,b;
a.next = &b;
a.next->next=NULL;
```

foo->bar
shorthand for
(*foo).bar

shorthand for
(*(a.next)).next

Type specifier: struct

giving names to bits inside a struct.

structs can have bit fields, unnamed fields, and word-aligned fields.

naming its bits

```
/* process ID info */
struct pid_tag {
    unsigned short int inactive :1;
    unsigned short int :1;              /* 1 bit of padding */
    unsigned short int refcount :6;
    unsigned short int :8;              /* pad to short length *
    short pid_id;
    struct pid_tag *link;
};
```

Type specifier: struct

```c
#include <stdio.h>
#include <string.h>

struct {
   unsigned int age : 3;
} Age;

int main( ) {

   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );

   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );

   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );

   return 0;
}
```

```
phbo@parallels-vm ~/C/C/Lectures> ./l3ex2
Sizeof(Age): 4
Age.age: 4
Age.age: 7
Age.age: 0
```

UNIVERSITY OF COPENHAGEN

Q: if I assign 10, and print, what gets printed?

16. ITU CPH

Type specifier: struct, the beauty of

```
struct s_tag { int a[100]; };
struct s_tag orange, lime, lemon;
struct s_tag twofold (struct s_tag s) {
    int j;
    for (j=0;j<100;j++) s.a[j] *= 2;
    return s;
}

main() {
    int i;
    for (i=0;i<100;i++) lime.a[i] = 1;
    lemon = twofold(lime);
    orange = lemon; /* assigns entire struct */
}
```

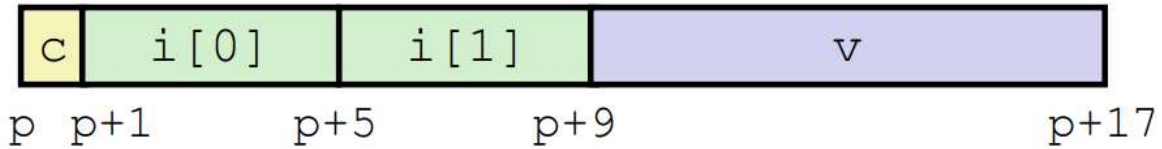
array as input

function cannot return an array,
but
function can return a struct.
(cf. returning a pointer)

this will **copy** the entire structure!
(if that's not what you want, then use pointers)

Structures & Alignment

Unaligned Data

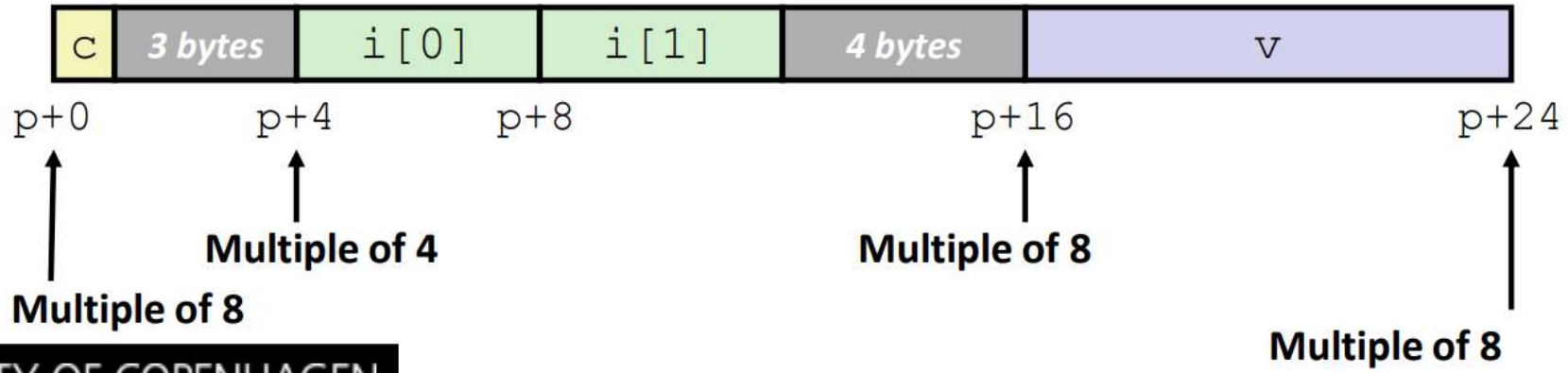



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Aligned Data

Primitive data type requires ***K*** bytes

Address must be multiple of ***K***



UNIVERSITY OF COPENHAGEN

ITU CPH

Specific Cases of Alignment (x86-64)

- 1 byte: **`char`**, …

  no restrictions on address
- 2 bytes: **`short`**, …

  lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, …

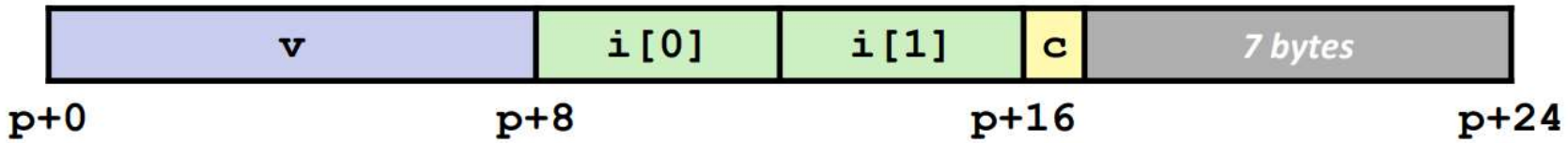
  lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, **`char *`**, …

  lowest 3 bits of address must be 000_2

Meeting Overall Alignment Requirement

For largest alignment requirement K
Overall structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



UNIVERSITY OF COPENHAGEN

ITU CPH

Type specifier: union

union: like struct, except:
the storage for the individual members is overlaid:
only one member at a time can be stored there.

```
union bits32_tag {
    int whole; /* a 4B value */
    struct {char c0,c1,c2,c3;} byte; /* 4 * 1B values */
}
```

example use: TCP frames

Type specifier: enum

enums (enumerated types) just a way of associating a series of names w/ a series of integer values.

```
enum sizes { small=7, medium, large=10, humongous };
```

8 (7+1)

11

`sizes.small` etc. are **constants**

Type specifier: void

void is the type of a function that does not return a result.

`unit` type
(but cannot use it as freely as other types,
hence this wording)

void *

void * defines a pointer to data of unspecified type.

Type qualifier: const

const qualifies a read-only variable; one that cannot be a left value in assignment (except variable declaration).

const.c

```
1 int main()
2 {
3         const int i;
4         i = 12;
5
6         return 0;
7 }
```

```
 % make const
cc     const.c   -o const
const.c:4:4: error: cannot assign to variable 'i' with const-qualified type 'const int'
        i = 12;
        ~ ^
const.c:3:12: note: variable 'i' declared const here
        const int i;
        ~~~~~~~~~~^
1 error generated.
make: *** [const] Error 1
```

combination of const and *: "I am giving you a pointer to this thing, but you may not change the pointer."

```
int * const p;
// p cannot be left value in an assignment
```

but you may overwrite what's at the other end of the pointer.

UNIVERSITY OF COPENHAGEN

ITU CPH

Type qualifier: const

(note: cannot have)

```
const int limit;
limit = 10;
```

```
const int limit = 10;
const int * limitp = &limit;
int i=27;
limitp = &i;
```

pointer to constants. can point to
her constants (i.e. something that
annot be on lhs of assignment.

```
int limit = 10;
int * const limitp = &limit;
int i=27;
limitp = &i;
```

constant pointer. cannot point to other things.
(**W:** but can overwrite pointee; e.g. upda

ITU CPH

Type qualifier: volatile

volatile qualifies a variable that might be modified outside the program. (by another thread, device, etc.).

(program reads it twice ⇏ same value read)

```
struct devregs{
   unsigned short volatile csr;
   unsigned short const volatile data;
};
```

can be initialized outside your program.

Type conversions

```
(type) expr      e.g.
(int)'y'
```

Explicit:

a value of one type is explicitly cast to another type.
bits do not change. higher bits that do not fit truncated.

Implicit:

1. a value of one type is assigned to a variable of a different type
2. an operator converts the type of its operands
3. **a value is passed as argument to a function, or when a value is returned from a function**

Unsigned and signed

Same bit level representation, different interpretations

If there is mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*

Pointer conversions, Rules

- A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage.
- A pointer to an object can be converted to a pointer to an object whose type requires less or equally strict storage alignment, and back again without change.
- A pointer to void can be converted to or from a pointer to any type, without restriction or loss of information. If the result is converted back to the original type, the original pointer is recovered.
- If a pointer is converted to another pointer with the same type but having different or additional qualifiers, the new pointer is the same as the old except for restrictions imposed by the new qualifier.

Pointer conversions, Rules

A pointer value can also be converted to an integral value.
The conversion path depends on the size of the pointer and the size of the integral type:

- If the size of the pointer is greater than or equal to the size of the integral type,
  the pointer behaves like an unsigned value. It cannot be converted to a floating value.
- If the pointer is smaller than the integral type,
  the pointer is first converted to a pointer with the same size as the integral type,
  then converted to the integral type.

Conversely, an integral type can be converted to a pointer type
according to the following rules:

- If the integral type is the same size as the pointer type,
  the conversion simply causes the integral value to be treated as a pointer
  (an unsigned integer).
- If the size of the integral type is different from the size of the pointer type,
  the integral type is first extended or truncated to fit the size of the pointer.
  It is then treated as a pointer value.

Pointer conversions, Example (common use)

Name

malloc, free, calloc, realloc - allocate and free dynamic memory

Synopsis

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

```
/* j is a pointer to an array of 20 char */
char (*j)[20];
j = (char (*)[20]) malloc( 20 );
```

C: Declarations & Definitions

ITU CPH

Declaration and definition

Definition: specifies
what a function does or where a variable is stored.

Declaration: describes type/name of variable/function.
No space is allocated.

Variables and functions are defined exactly once,
but may be declared several times.

W: think of dec as "intent",
and def as "enact"

def fun: what it does
def var: how it's stored
dec: just a signature.

dec: x exists,
def: dec + allocate mem for x

UNIVERSITY OF COPENHAGEN

ITU CPH

Scope of variables

A variable defined in a function is local to that function. It is an **automatic** variable. It does not retain its value across function calls (lives in stack frame).

A variable defined outside any function is an *external* variable. It is a **global** variable.

Before a global variable can be accessed in other files, it must be declared with the `extern` prefix.

A global variable does not need to be declared in the file where it is defined.

typically, these are collected in a header file

UNIVERSITY OF COPENHAGEN ITU CPH

Scope of variables

The scope of a global variable can be restricted to the file where it is defined with the `static` prefix.

`static` and `extern` are mutually exclusive.

An automatic variable can retain its value across calls to a function when it is defined with `static`.

Recap so far

What is an automatic variable?

A: local to function

How is a global variable defined when it can be accessed by all C files contributing to an executable?

A: in files where not def, it must be dec w/ `extern`

How is a global variable defined when it can only be accessed from the C file where it is defined?

A: `static`

How is a global variable declared outside the file where it is defined?

A: `extern`

Example

function definition; we are specifying what function does

local to function, but retains values across calls.

scope.c

```c
#include<stdio.h>
int fun()
{
        static int count = 0;
        count++;
        return count;
}

int main()
{
        printf("%d ", fun());
        printf("%d ", fun());
        return 0;
}
```

```
 % make scope
cc      scope.c   -o scope
 % ./scope
1 2
```

Example

scope2.c

```c
extern int var;
int main(void)
{
        var = 10;
        return 0;
}
```

```
 % make scope2
cc     scope2.c   -o scope2
Undefined symbols for architecture x86_64:
  "_var", referenced from:
      _main in scope2-4b8294.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
make: *** [scope2] Error 1
```

solution:
define it in a header file. (next)

Example

int var implicitly defined (to 0)

scope2.c

```
#include "scope2.h"

extern int var;
int main(void)
{
        var = 10;
        return 0;
}
```

scope2.c scope2.h

```
int var;
```

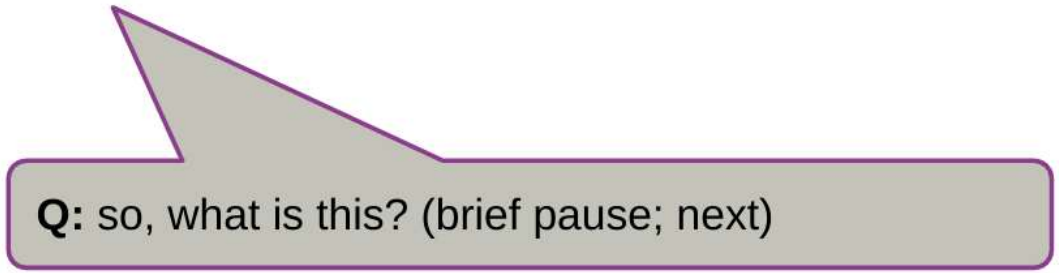
```
% make scope2
cc     scope2.c   -o scope2
```

C Declarations, what they mean

1. Declarations are read by starting with the name (of the variable, function or type)
2. The following precedence rules apply:
   A. Parentheses grouping together part of the declaration
   B. The postfix operators
      - Parenthesis indicating a function
      - Square brackets indicating an array
   C. The prefix operator
      - * denoting a pointer to
3. If a const or volatile is next to a type specifier it qualifies it, otherwise const or volatile applies to the * on its immediate left

Unscrambling C declarations, Example

```
char* const *(*next)();
```



Unscrambling C declarations, Example

```
char* const *(*next)();
```

Next is (1)
a pointer to (2A)
a function returning (2B)
a pointer to (2C)
a constant pointer to (3)
char

that's one expressive type system, from the 70s!

C: Gotcha-s

ITU CPH

Security, Revisited

```
13 #define MSIZE 528
14
15 void getstuff() {
16     char mybuf[MSIZE];
17     copy_from_kernel(mybuf, -MSIZE);
18     . . .
19 }
```

l2ex1.c+

```
1  /* Kernel memory region holding user-accessible data */
2  #define KSIZE 1024
3  char kbuf[KSIZE];
4
5  /* Copy at most maxlen bytes from kernel region to user buffer */
6  int copy_from_kernel(void *user_dest, int maxlen) {
7      /* Byte count len is minimum of buffer size and maxlen */
8      int len = KSIZE < maxlen ? KSIZE : maxlen;
9      memcpy(user_dest, kbuf, len);
10     return len;
11 }
```

memcpy

```
MEMCPY(3)                        Linux Programmer's Manual                       MEMCPY(3)

NAME
       memcpy - copy memory area

SYNOPSIS
       #include <string.h>

       void *memcpy(void *dest, const void *src, size_t n);

DESCRIPTION
       The  memcpy()  function  copies  n  bytes from memory area src to memory area dest.  The memory areas must not
       overlap.  Use memmove(3) if the memory areas do overlap.

RETURN VALUE
       The memcpy() function returns a pointer to dest.
```

From GNU glibc manual /Appendix A – Language Features /Important Data Types:

ata Type: **size_t**

This is an unsigned integer type used to represent the sizes of objects. The result of the `sizeof` operator is of this type, and functions such as `malloc` (see Unconstrained Allocation) and `memcpy` (see Copying S and Arrays) accept arguments of this type to specify object sizes. On systems using the GNU C Library, this will be `unsigned int` or `unsigned long int`.

Usage Note: `size_t` is the preferred way to declare any arguments or variables that hold the size of an object.

UNIVERSITY OF COPENHAGEN

09. ITU CPH 29

Security - Woops

```
12
13 #define MSIZE 528
14
15 void getstuff() {
16     char mybuf[MSIZE];
17     copy_from_kernel(mybuf, -MSIZE);
18     . . .
19 }
```

ex1.c+

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

-528 in two's complement:

0xFFFFFDF0

Reinterpreted as unsigned within memcpy: 4294966768 (decimal)

Why do we care?

Always be mindful/careful of unsigned integers!

Undefined Behavior; What happens?

"[In C], anything at all can happen; the standard imposes no requirements. [The compiler decides what should happen.] The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended."

"In a *safe* programming language [like Java], errors are trapped as they happen [(e.g. via. exceptions)]. [...]
In an *unsafe* programming language, errors are not trapped."

- John Regehr

https://blog.regehr.org/archives/213
http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

Exercise

Consider the following code:

le3ex1.c

```c
#include <stdio.h>

int main() {
    int *ptr;
    *ptr = 20;
    printf("%d\n", *ptr);
    return 0;
}
```

What is wrong?

Exercise

Consider the following code:

le3ex1.c

```c
#include <stdio.h>

int main() {
    int *ptr;
    *ptr = 20;
    printf("%d\n", *ptr);
    return 0;
}
```

not initialized (indeterminate pointer); where to store the 20?

What is wrong?

(compiler *should* complain, but compiler *could* do anything)

Exercise

```
phbo@mac610891 ~/D/C/BOSC-E19> cc -Wall l3ex1.c -o l3ex1
l3ex1.c:5:3: warning: variable 'ptr' is uninitialized when used here [-Wuninitialized]
        *ptr = 20;
         ^~~
l3ex1.c:4:10: note: initialize the variable 'ptr' to silence this warning
        int *ptr;
                ^
                 = NULL
1 warning generated.
```

Consider the

l3ex1.c

```c
#include <stdio.h>

int main() {
    int *ptr;
    *ptr = 20;
    printf("%d\n", *ptr);
    return 0;
}
```

gcc decided to initialize to null. other compilers, or later versions of gcc, might do different. (segfault, write to device memory, …)

```
(gdb) p ptr
$2 = (int *) 0x0
(gdb) p &ptr
$3 = (int **) 0x7fffffffe3a8
```

What is wrong?

UNIVERSITY OF COPENHAGEN

16. ITU CPH

Initialize your Pointers!

Pointers are valid, null or indeterminate.

A pointer is null when assigned 0

Null pointers evaluate to false in logical expressions

Dereferencing indeterminate pointers leads to **undefined behaviour**

Always initialize pointers!

Overloading, of keywords

From expert C programming

Symbol	Meaning
`static`	Inside a function, *retains its value between calls* At the function level, *visible only in this file* [1]
`extern`	Applied to a function definition, *has global scope* (and is redundant) Applied to a variable, *defined elsewhere*
`void`	As the return type of a function, *doesn't return a value* In a pointer declaration, the type of a generic pointer In a parameter list, *takes no parameters*
*	The multiplication operator

can be used for different things. (just need to be aware of the overloading)

Overloading, of symbols

From expert C programming

lol

lol

lol

lol

*	The multiplication operator Applied to a pointer, indirection In a declaration, a pointer
&	Bitwise `AND` operator Address-of operator
= ==	Assignment operator Comparison operator
<= <<=	Less-than-or-equal-to operator Compound shift-left assignment operator
< <	Less-than operator Left delimiter in `#include` directive
()	Enclose formal parameters in a function definition Make a function call Provide expression precedence Convert (cast) a value to a different type Define a macro with arguments Make a macro call with arguments Enclose the operand of the `sizeof` operator when it is a typename

Precedence rules (operators)

a mess.

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13[note 1]	?:	Ternary conditional[note 2]	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

From http://en.cppreference.com/w/c/language/operator_precedence

UNIVERSITY OF COPENH

16. ITU CPH

Precedence, when in doubt...

"Some operators have the wrong precedence"

Kernighan and Ritchie.

Precedence problem	Expression	What People Expect	What They Actually Get
. is higher than * the p->f op was made to smooth over this	`*p.f`	the f field of what p points to `(*p).f`	take the f offset from p, use it as a pointer `*(p.f)`
[] is higher than *	`int *ap[]`	ap is a ptr to array of ints `int(*ap)[]`	ap is an array of ptrs-to-int `int *(ap[])`
function () higher than *	`int *fp()`	fp is a ptr to function returning int `int (*fp) ()`	fp is a function returning ptr-to-int `int *(fp())`
== and != higher precedence than bitwise operators	`(val&mask != 0)`	`(val&mask) !=0`	`val & (mask !=0)`
== and != higher precedence than assignment	`c=getchar( )!=EOF`	`(c=getchar()) != EOF`	`c=(getchar() !=EOF)`
arithmetic higher precedence than shift	`msb<<4 + lsb`	`(msb<<4)+lsb`	`msb<<(4+lsb)`
, has lowest precedence of all operators	`i = 1,2;`	`i= (1,2);`	`(i=1), 2;`

when in doubt:
use parentheses!
circumvents
precedence
rules

UNIVERSITY OF COPENHAGEN

ITU CPH

source: expert C programming

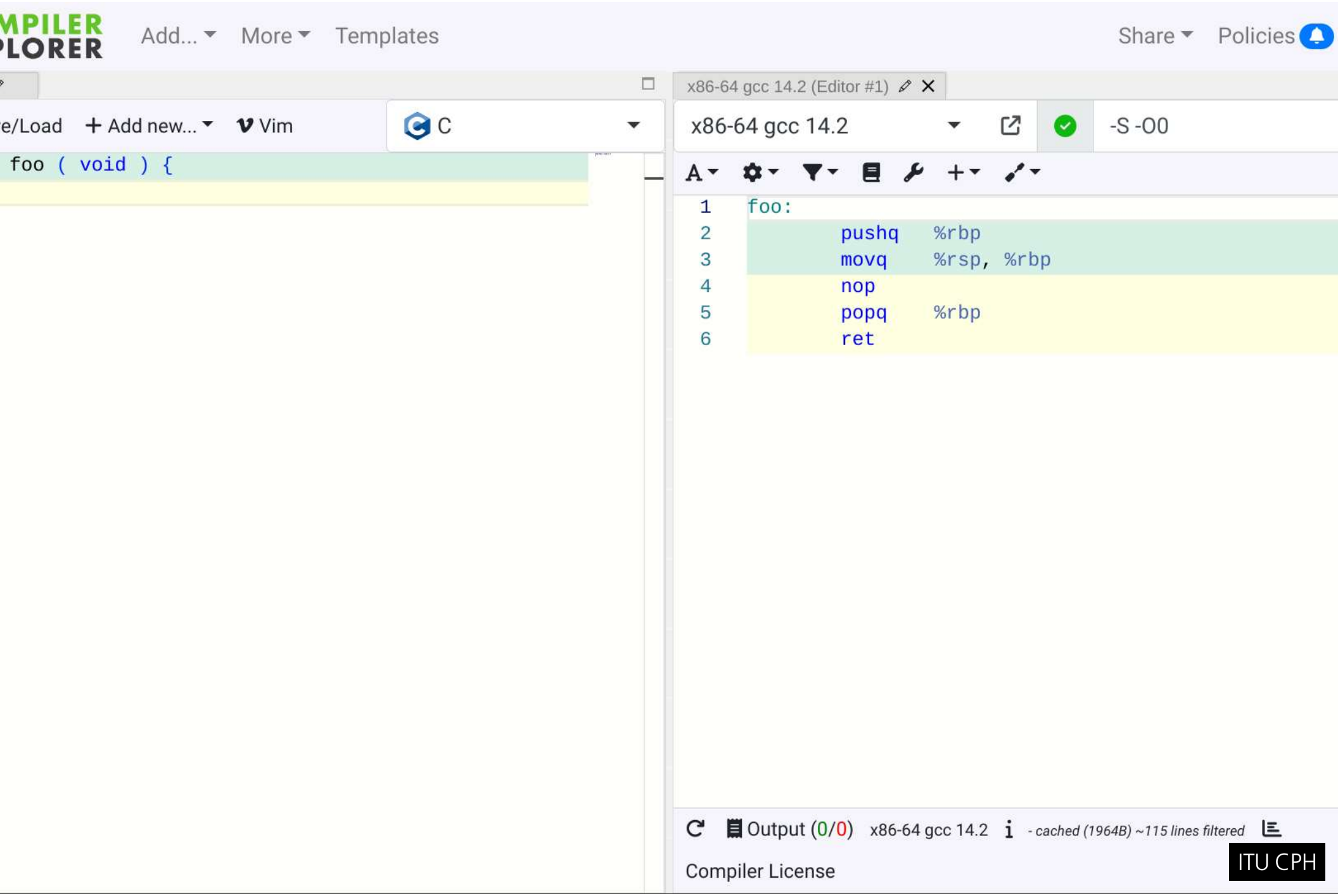
C: Take-Aways

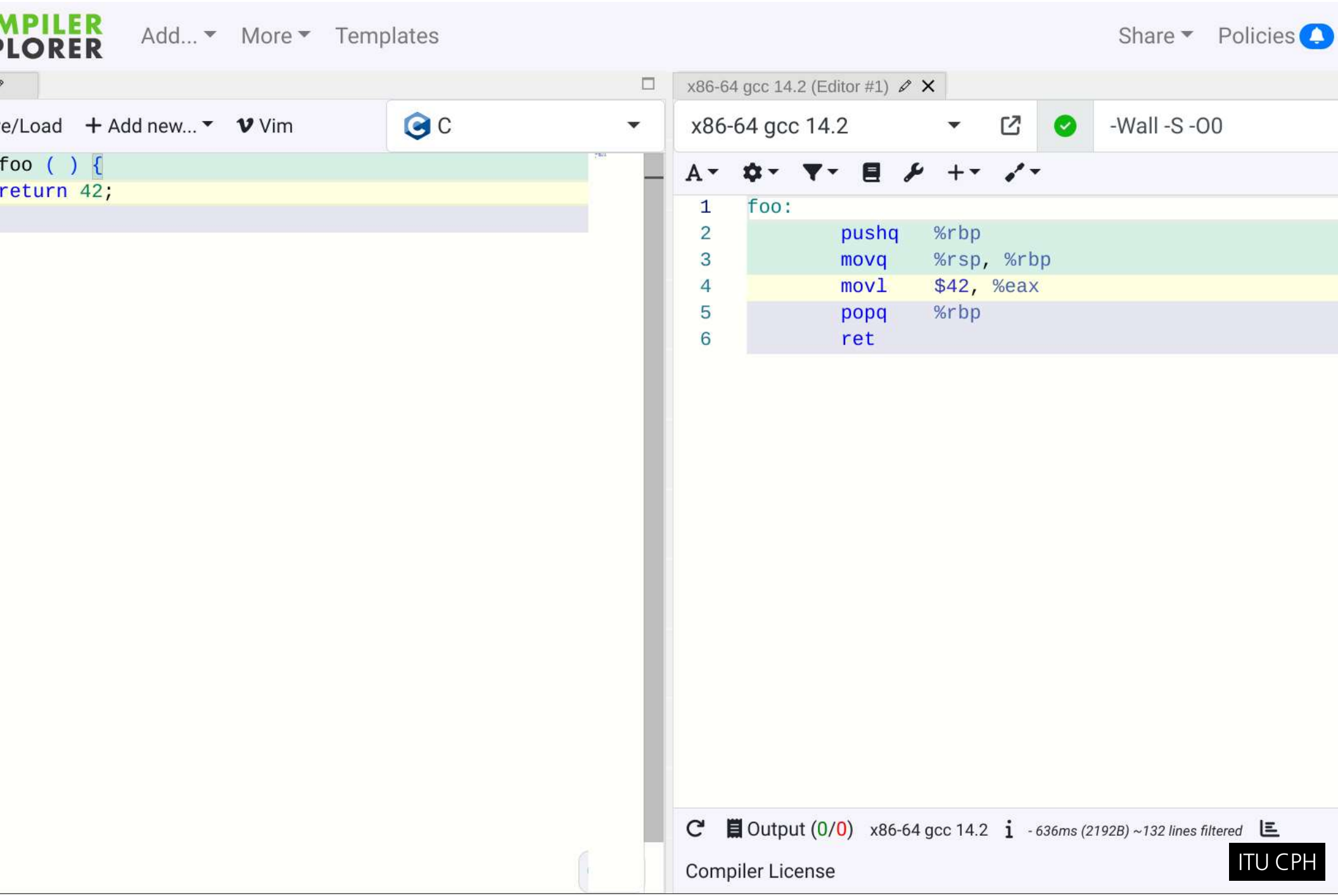
You should remember:

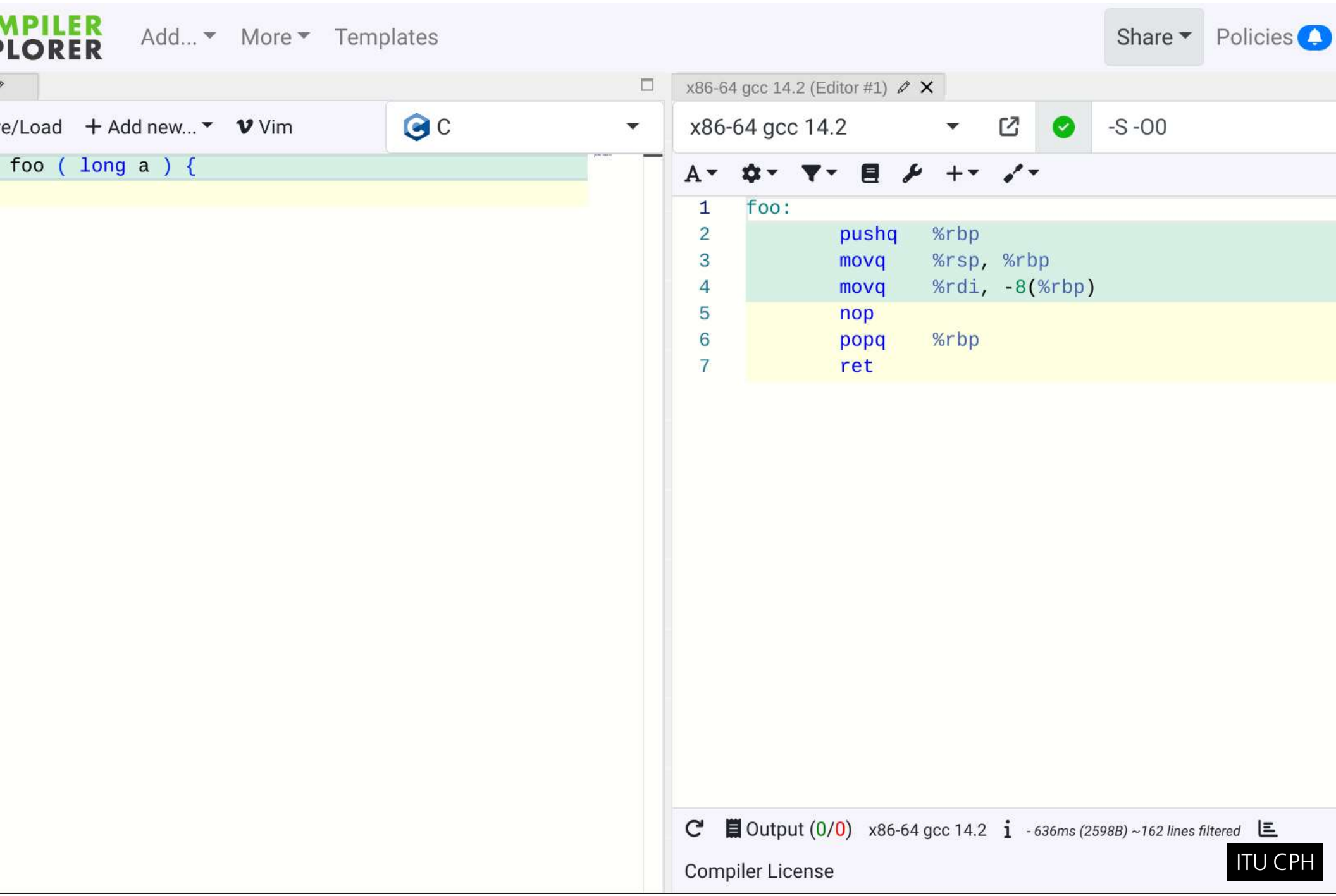
1. A pointer is a variable that contains the address of a variable
2. The nature of structs
3. The meaning of const and volatile
4. When type conversions takes place
5. The difference between type specifier and qualifier
6. The difference between declaration and definition
7. The scope of variables (automatic / global)
8. What happens when signed and unsigned are mixed
9. Beware operator precedence
10. Use cdecl when in doubt about a declaration

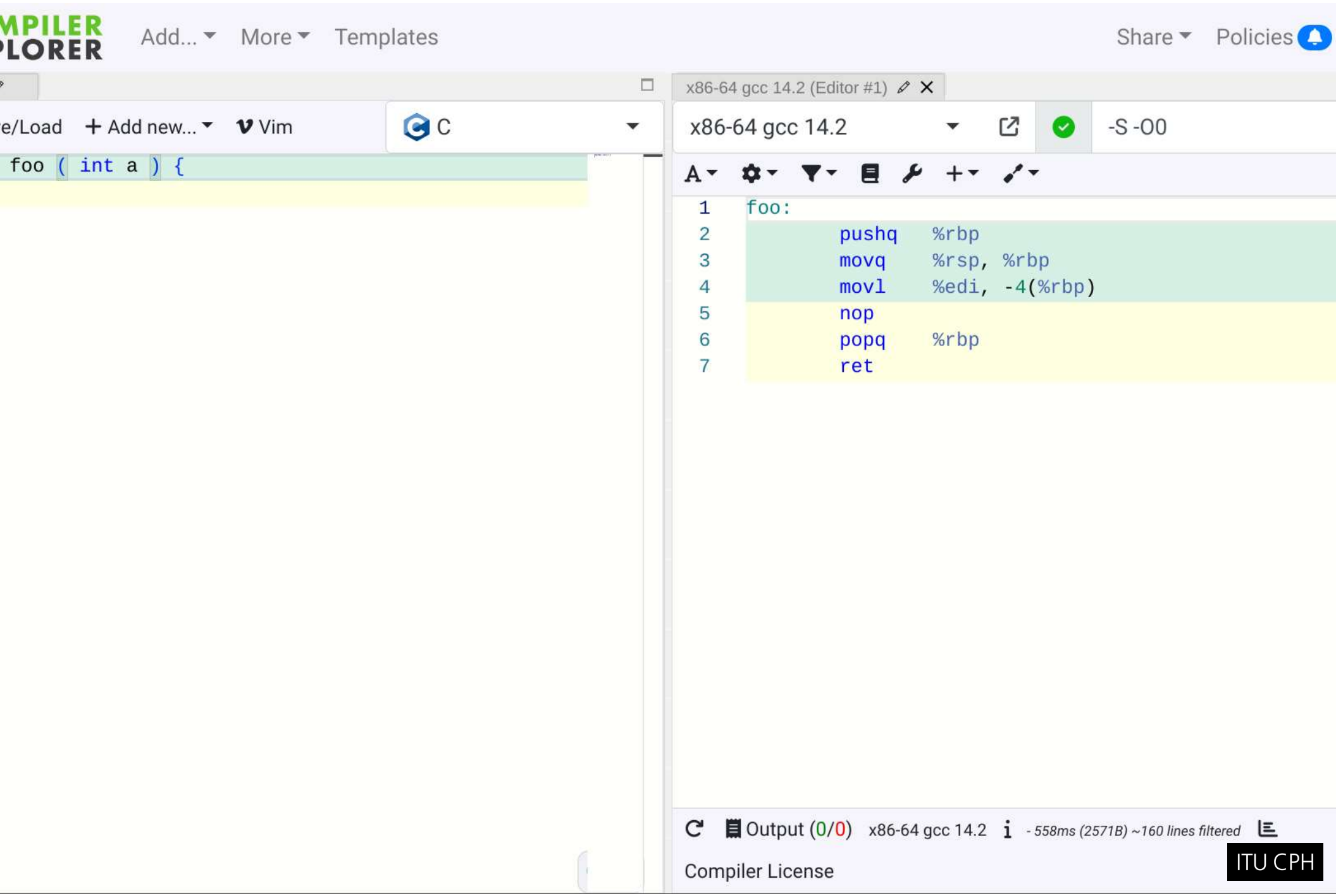
C to Assembly

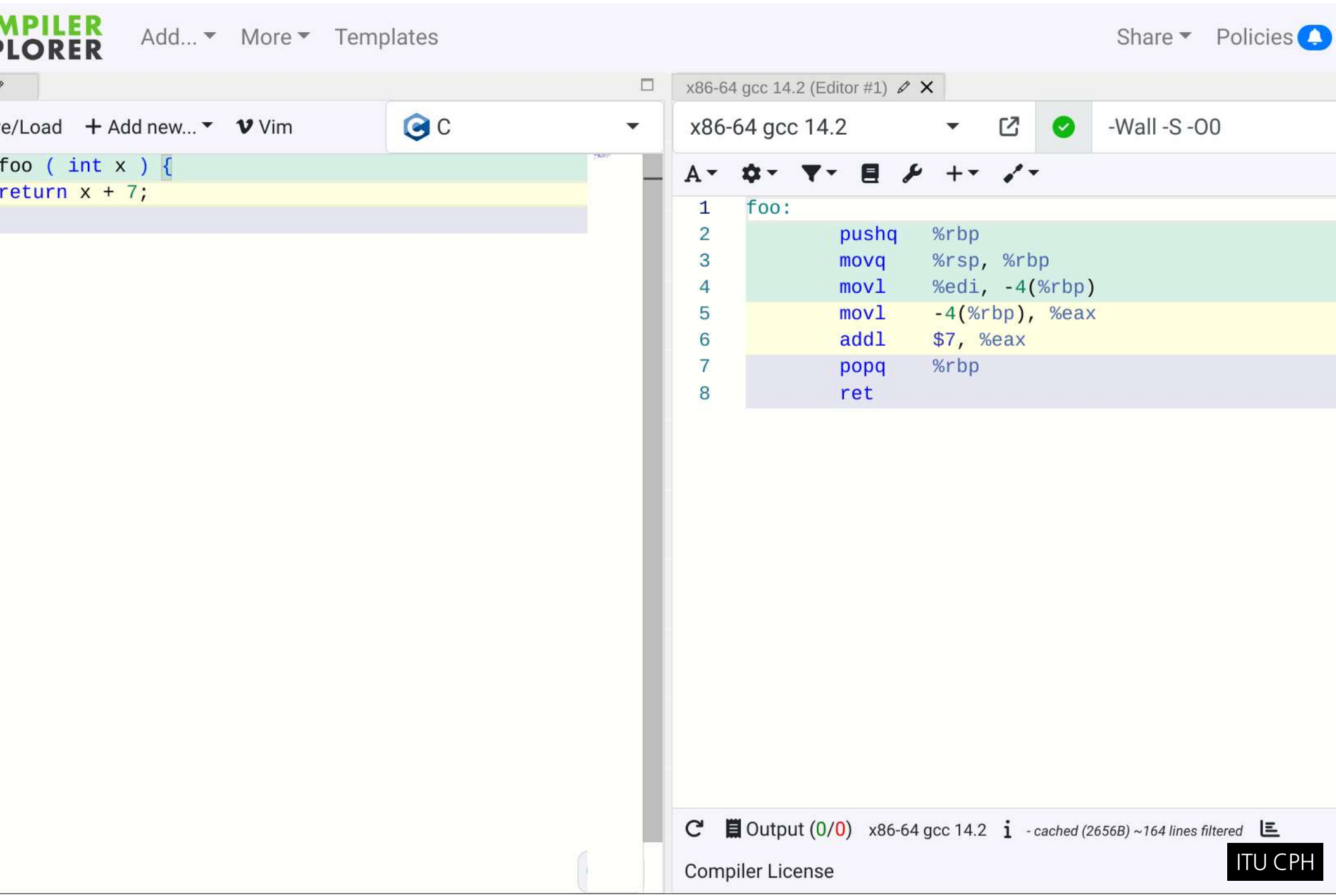
ITU CPH

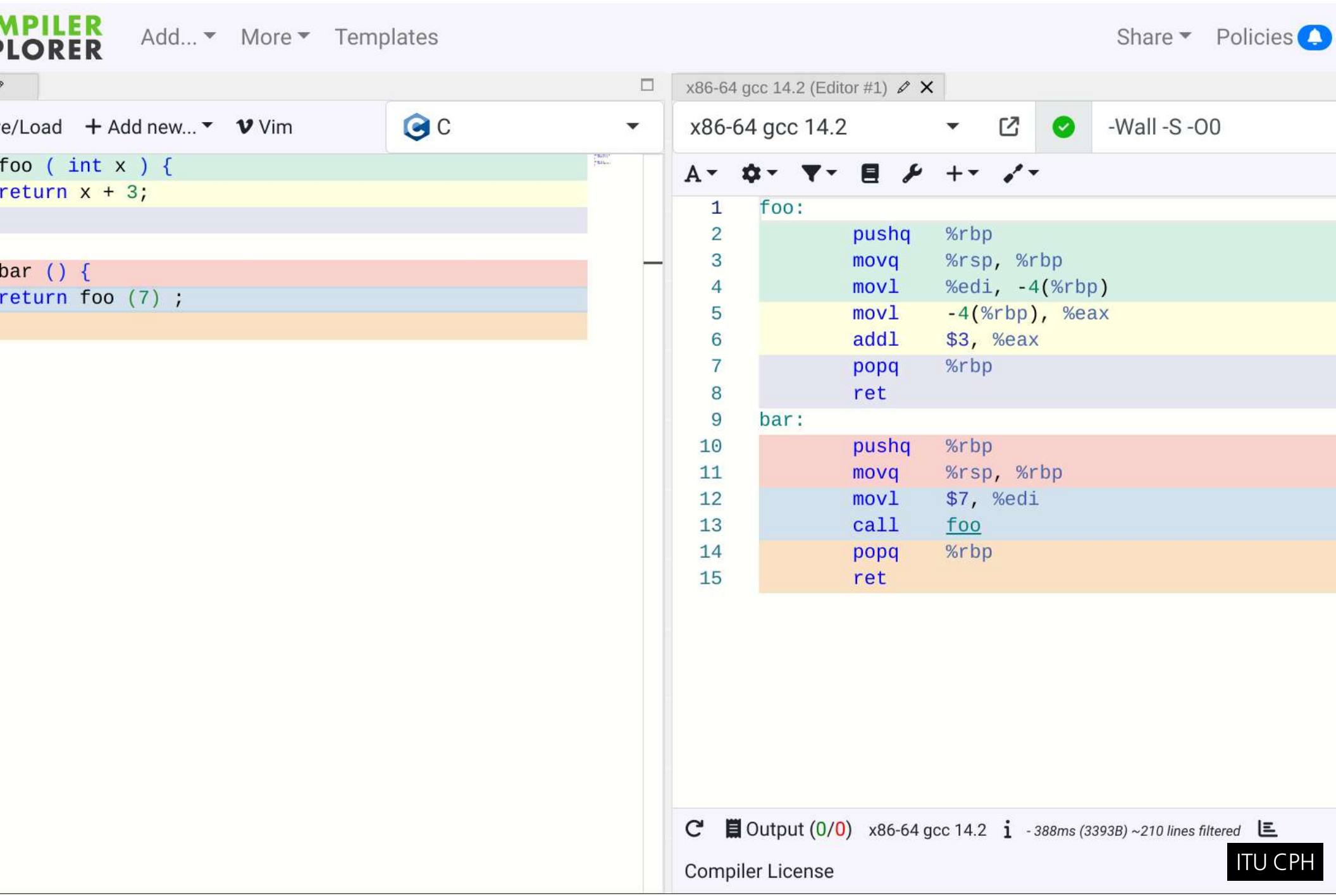

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (void) {
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 nop
5 popq %rbp
6 ret
Output (0/0)
x86-64 gcc 14.2
- cached (1964B) ~115 lines filtered
Compiler License
ITU CPH

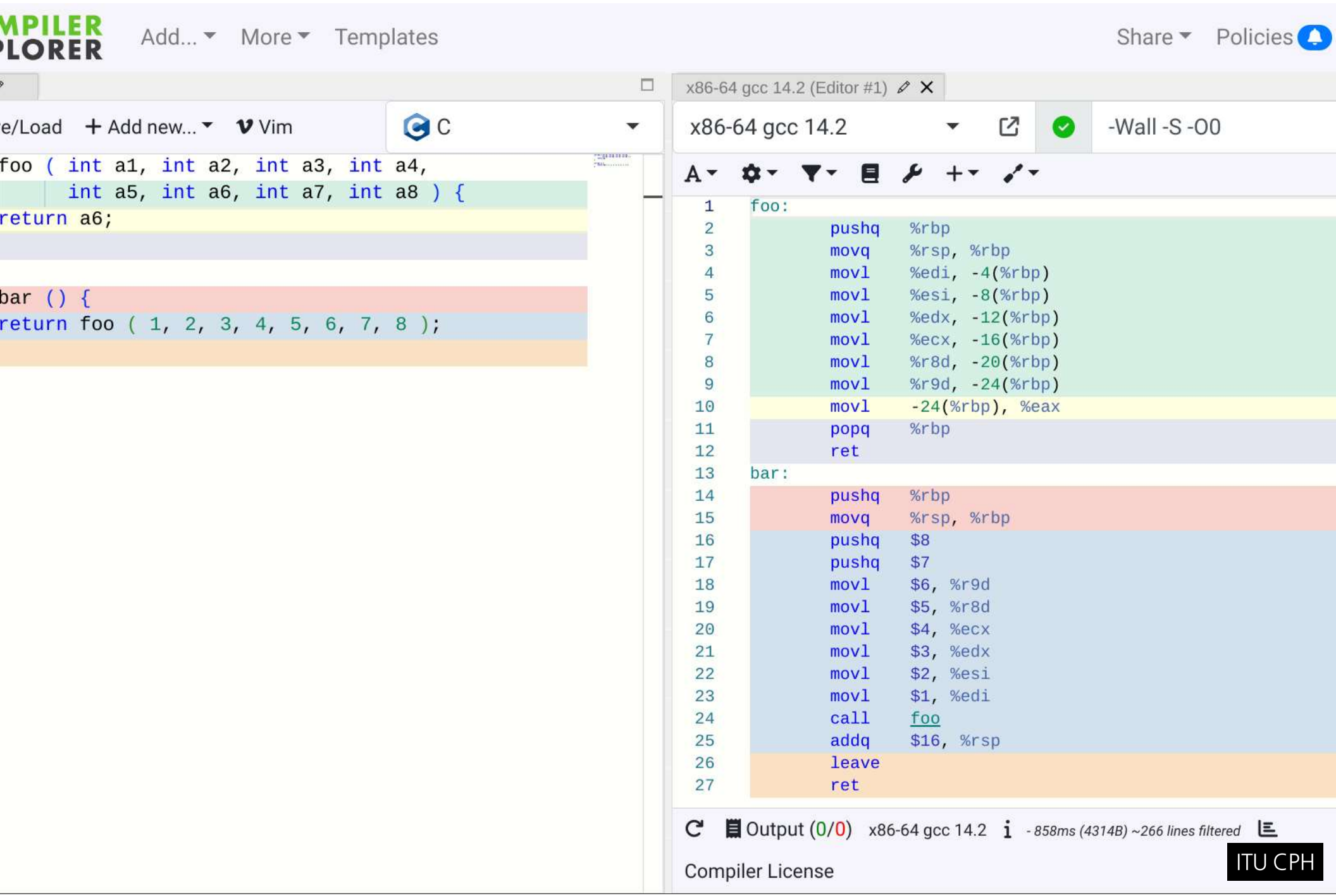

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo ( ) {
return 42;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl $42, %eax
5 popq %rbp
6 ret
Output (0/0)
x86-64 gcc 14.2
- 636ms (2192B) ~132 lines filtered
Compiler License
ITU CPH

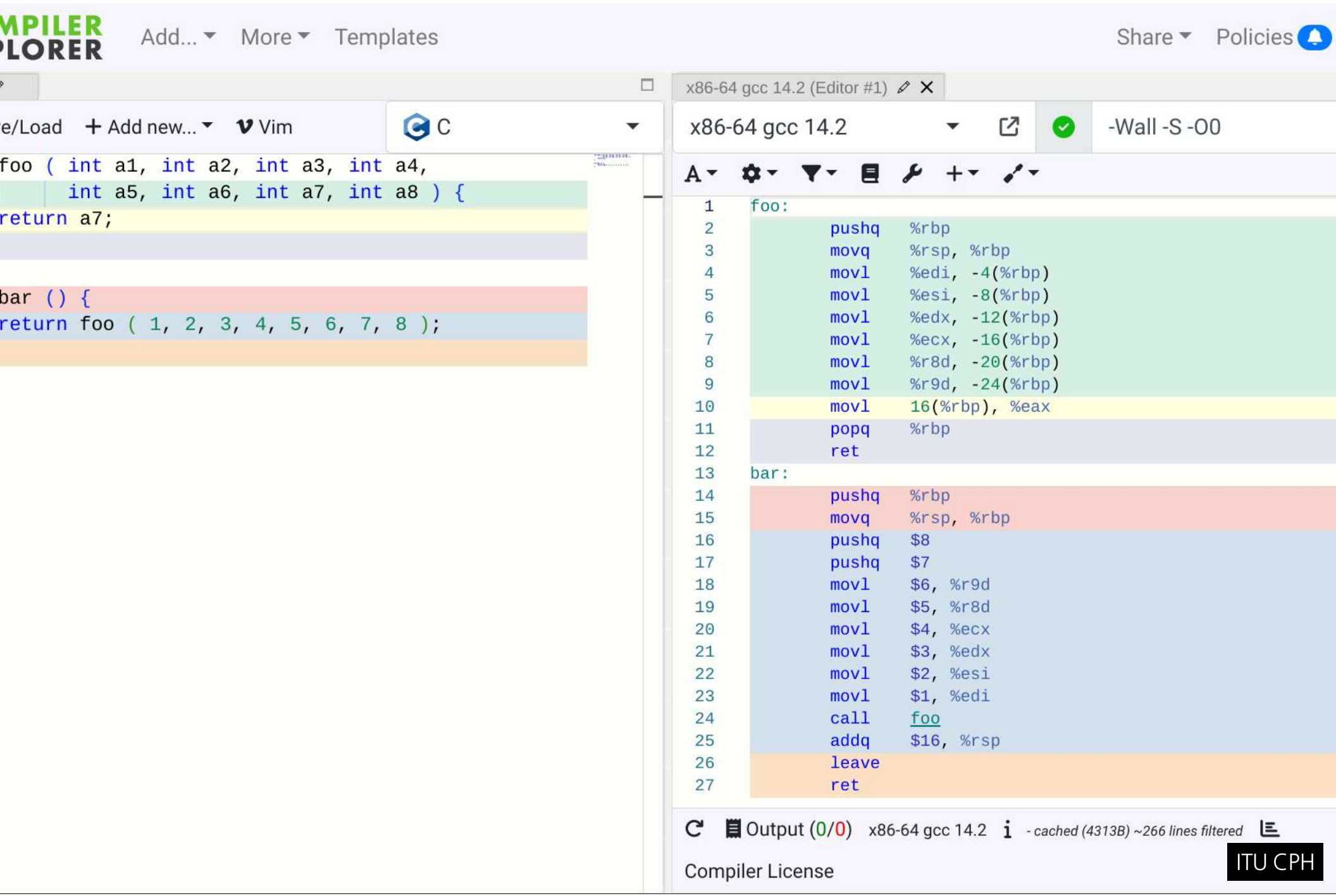

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (long a) {
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 nop
6 popq %rbp
7 ret
Output (0/0)
x86-64 gcc 14.2
- 636ms (2598B) ~162 lines filtered
Compiler License
ITU CPH

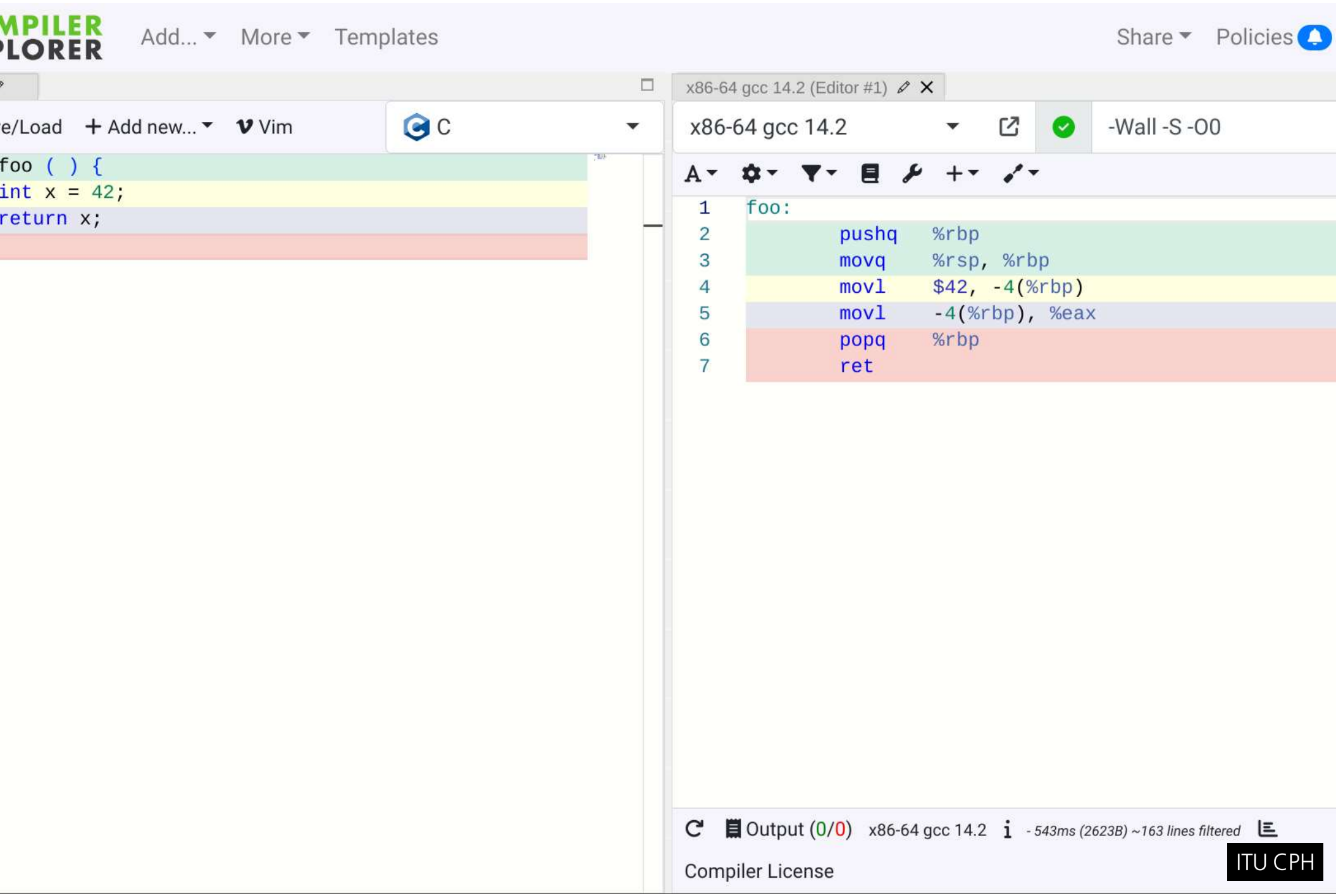

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int a) {
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -4(%rbp)
5 nop
6 popq %rbp
7 ret
Output (0/0)
x86-64 gcc 14.2
- 558ms (2571B) ~160 lines filtered
Compiler License
ITU CPH

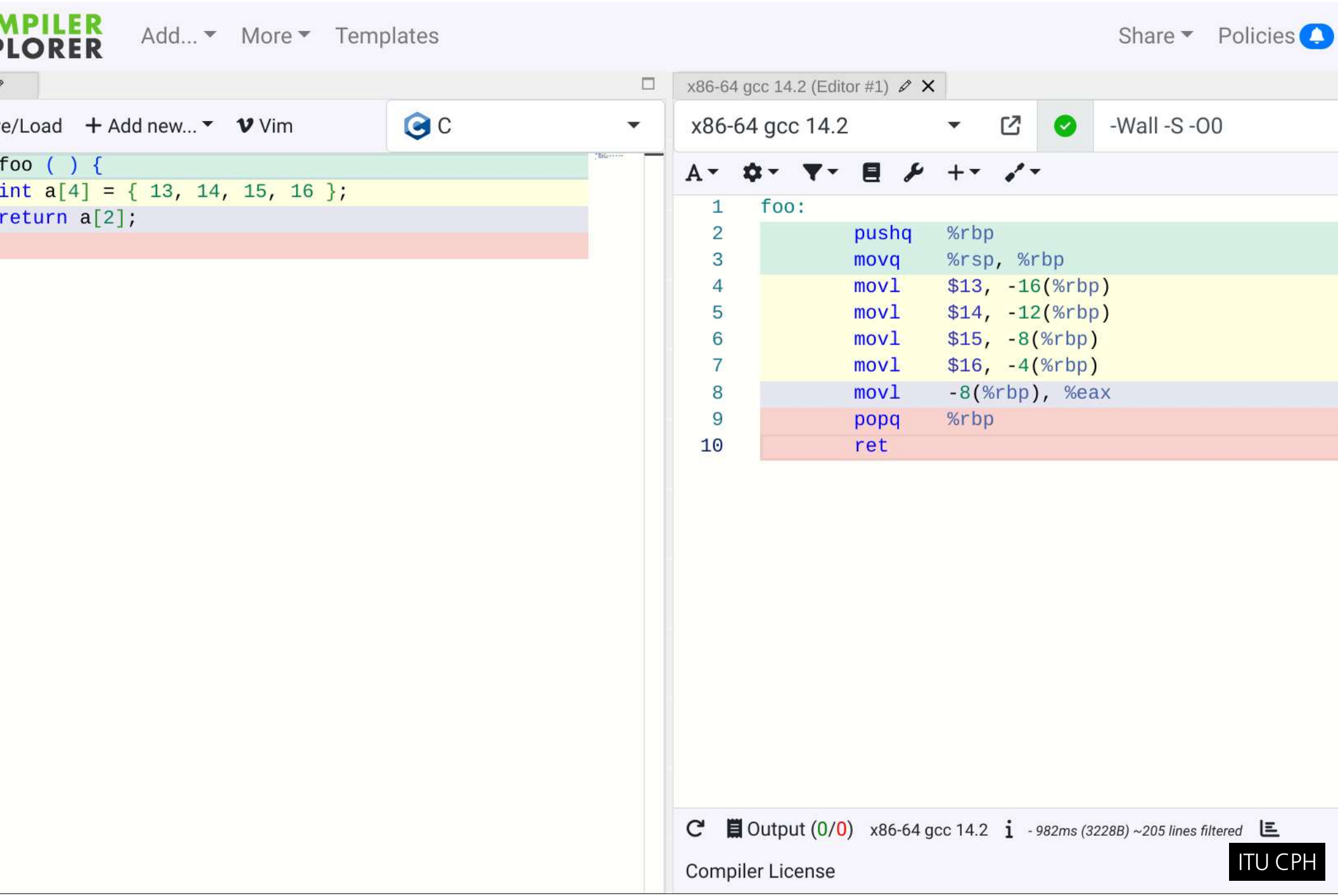

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int x) {
return x + 7;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -4(%rbp)
5 movl -4(%rbp), %eax
6 addl $7, %eax
7 popq %rbp
8 ret
Output (0/0)
x86-64 gcc 14.2
- cached (2656B) ~164 lines filtered
Compiler License
ITU CPH

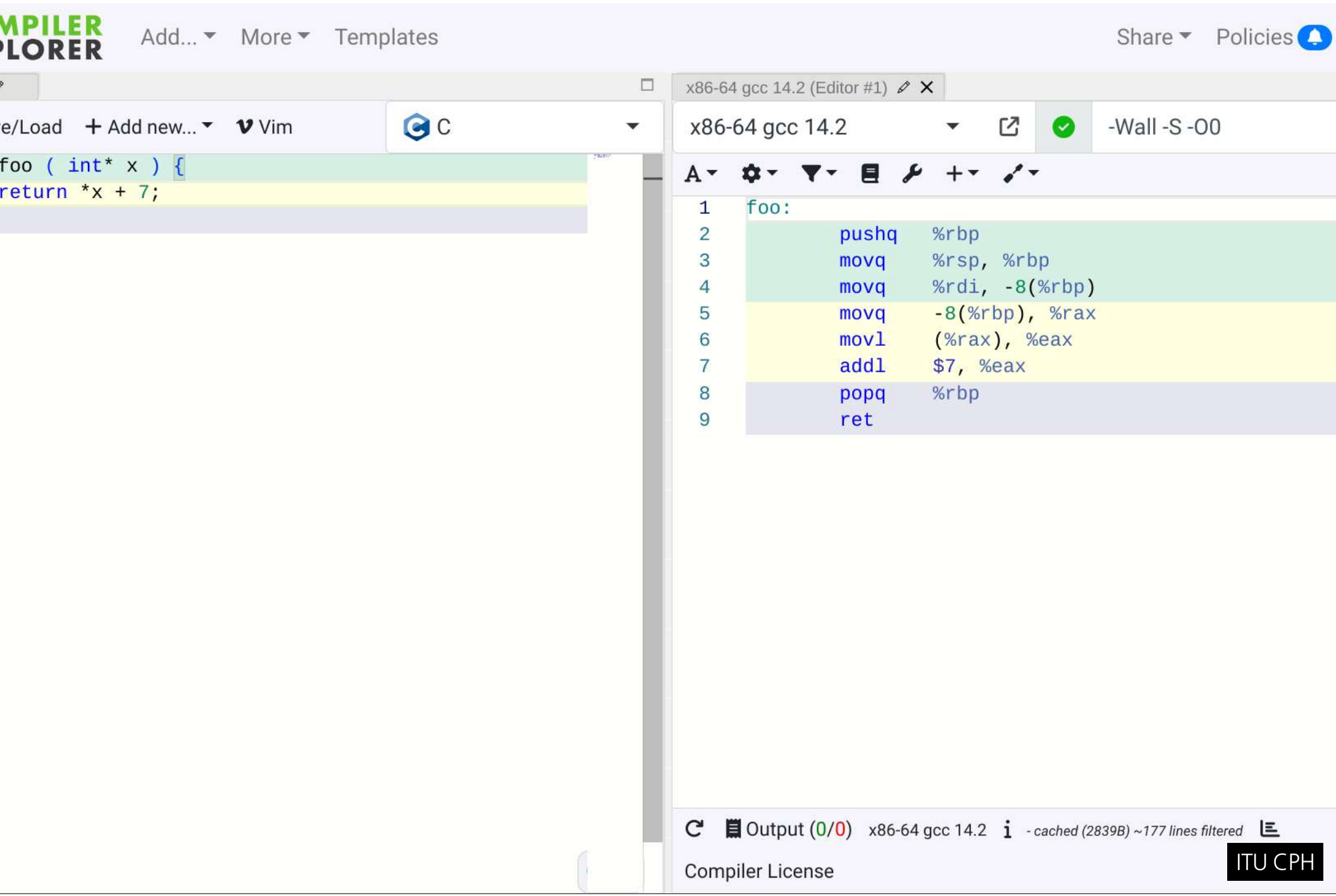

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int x) {
return x + 3;
bar () {
return foo (7) ;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -4(%rbp)
5 movl -4(%rbp), %eax
6 addl $3, %eax
7 popq %rbp
8 ret
9 bar:
10 pushq %rbp
11 movq %rsp, %rbp
12 movl $7, %edi
13 call foo
14 popq %rbp
15 ret
Output (0/0)
x86-64 gcc 14.2
- 388ms (3393B) ~210 lines filtered
Compiler License
ITU CPH

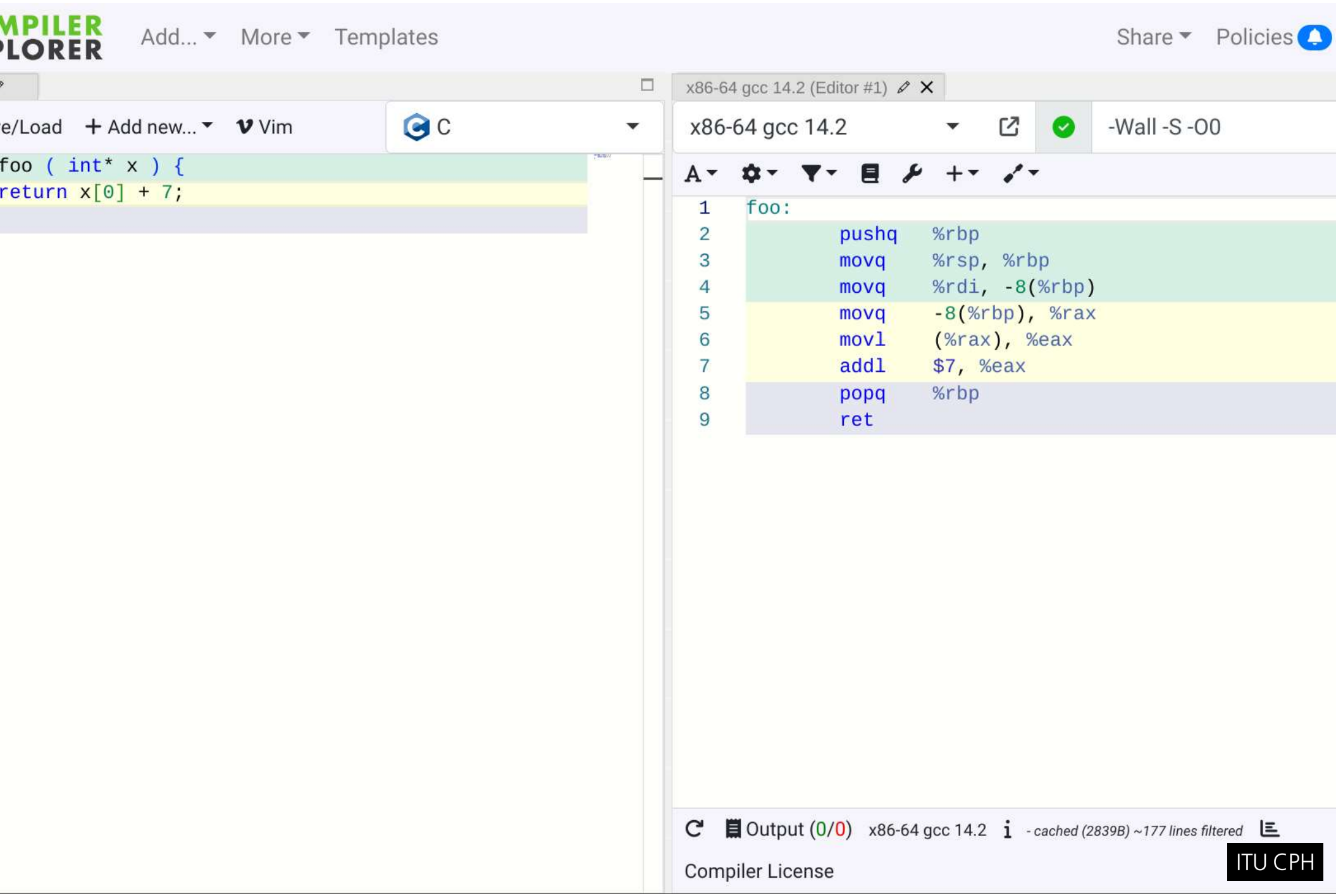

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
foo (int a1, int a2, int a3, int a4,
int a5, int a6, int a7, int a8) {
return a6;
bar () {
return foo (1, 2, 3, 4, 5, 6, 7, 8);
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -4(%rbp)
5 movl %esi, -8(%rbp)
6 movl %edx, -12(%rbp)
7 movl %ecx, -16(%rbp)
8 movl %r8d, -20(%rbp)
9 movl %r9d, -24(%rbp)
10 movl -24(%rbp), %eax
11 popq %rbp
12 ret
13 bar:
14 pushq %rbp
15 movq %rsp, %rbp
16 pushq $8
17 pushq $7
18 movl $6, %r9d
19 movl $5, %r8d
20 movl $4, %ecx
21 movl $3, %edx
22 movl $2, %esi
23 movl $1, %edi
24 call foo
25 addq $16, %rsp
26 leave
27 ret
Output (0/0) x86-64 gcc 14.2 - 858ms (4314B) ~266 lines filtered
Compiler License
ITU CPH

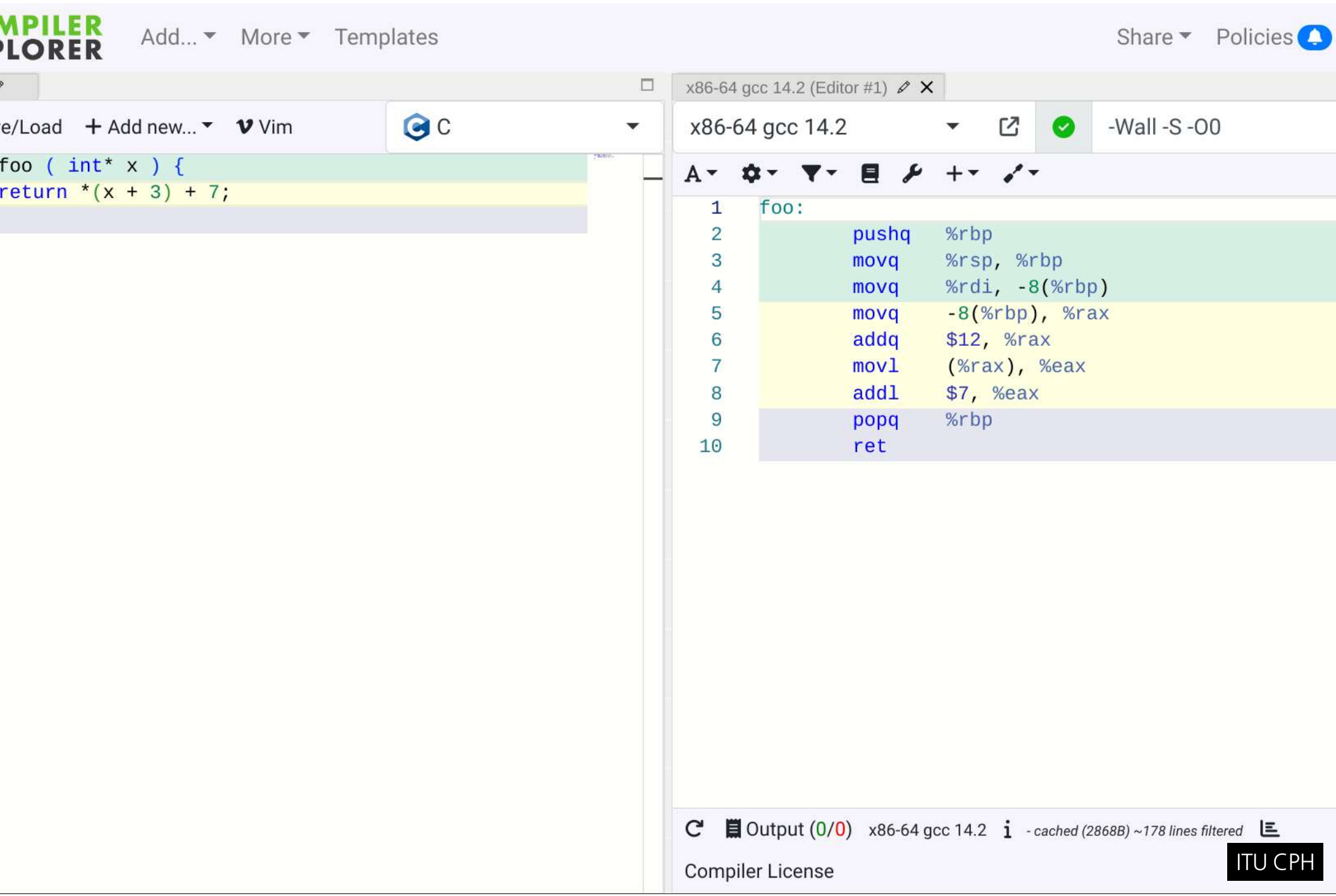

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int a1, int a2, int a3, int a4,
int a5, int a6, int a7, int a8) {
return a7;
bar () {
return foo (1, 2, 3, 4, 5, 6, 7, 8);
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -4(%rbp)
5 movl %esi, -8(%rbp)
6 movl %edx, -12(%rbp)
7 movl %ecx, -16(%rbp)
8 movl %r8d, -20(%rbp)
9 movl %r9d, -24(%rbp)
10 movl 16(%rbp), %eax
11 popq %rbp
12 ret
13 bar:
14 pushq %rbp
15 movq %rsp, %rbp
16 pushq $8
17 pushq $7
18 movl $6, %r9d
19 movl $5, %r8d
20 movl $4, %ecx
21 movl $3, %edx
22 movl $2, %esi
23 movl $1, %edi
24 call foo
25 addq $16, %rsp
26 leave
27 ret
Output (0/0) x86-64 gcc 14.2
- cached (4313B) ~266 lines filtered
Compiler License
ITU CPH

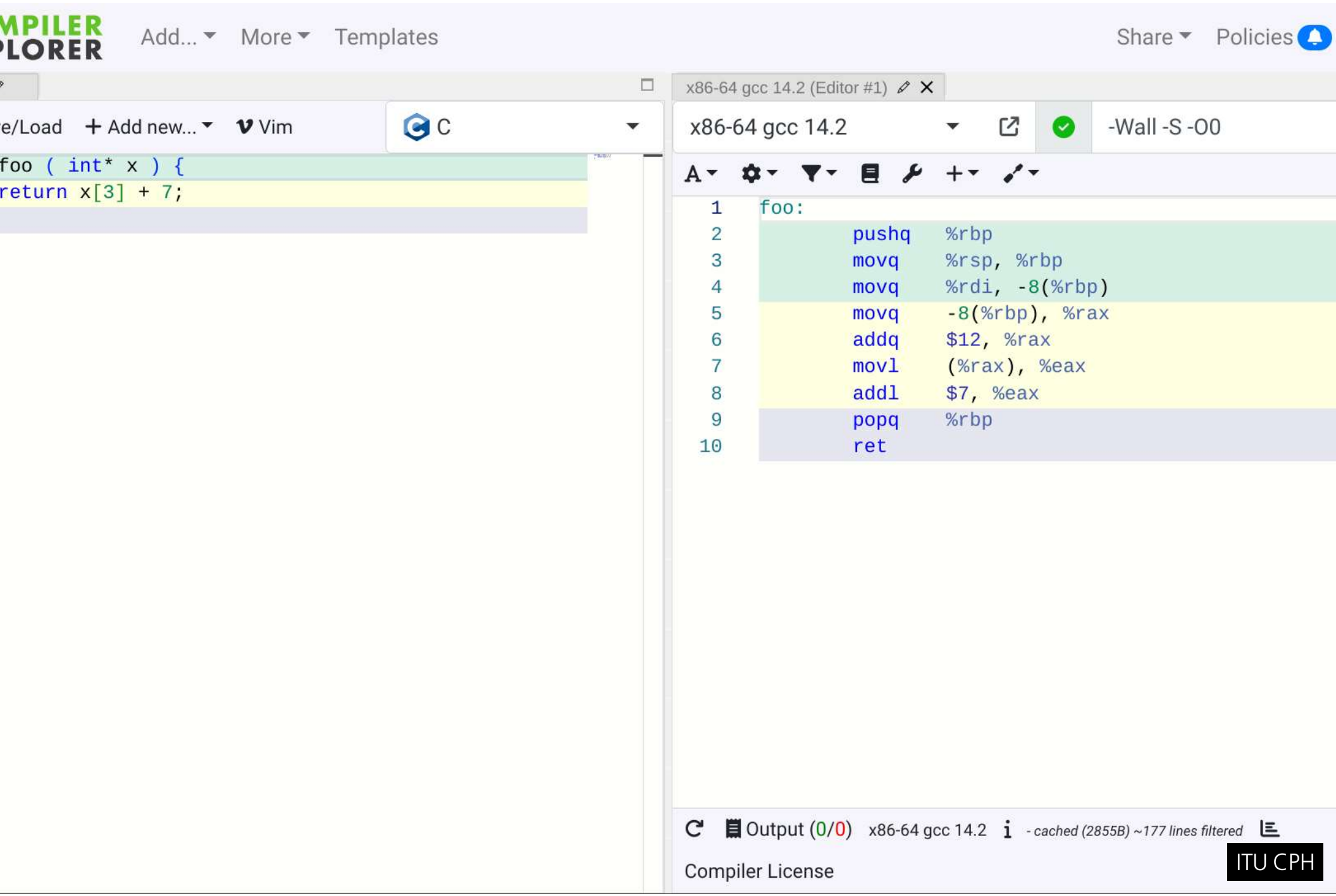

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo ( ) {
int x = 42;
return x;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl $42, -4(%rbp)
5 movl -4(%rbp), %eax
6 popq %rbp
7 ret
Output (0/0)
x86-64 gcc 14.2
- 543ms (2623B) ~163 lines filtered
Compiler License
ITU CPH

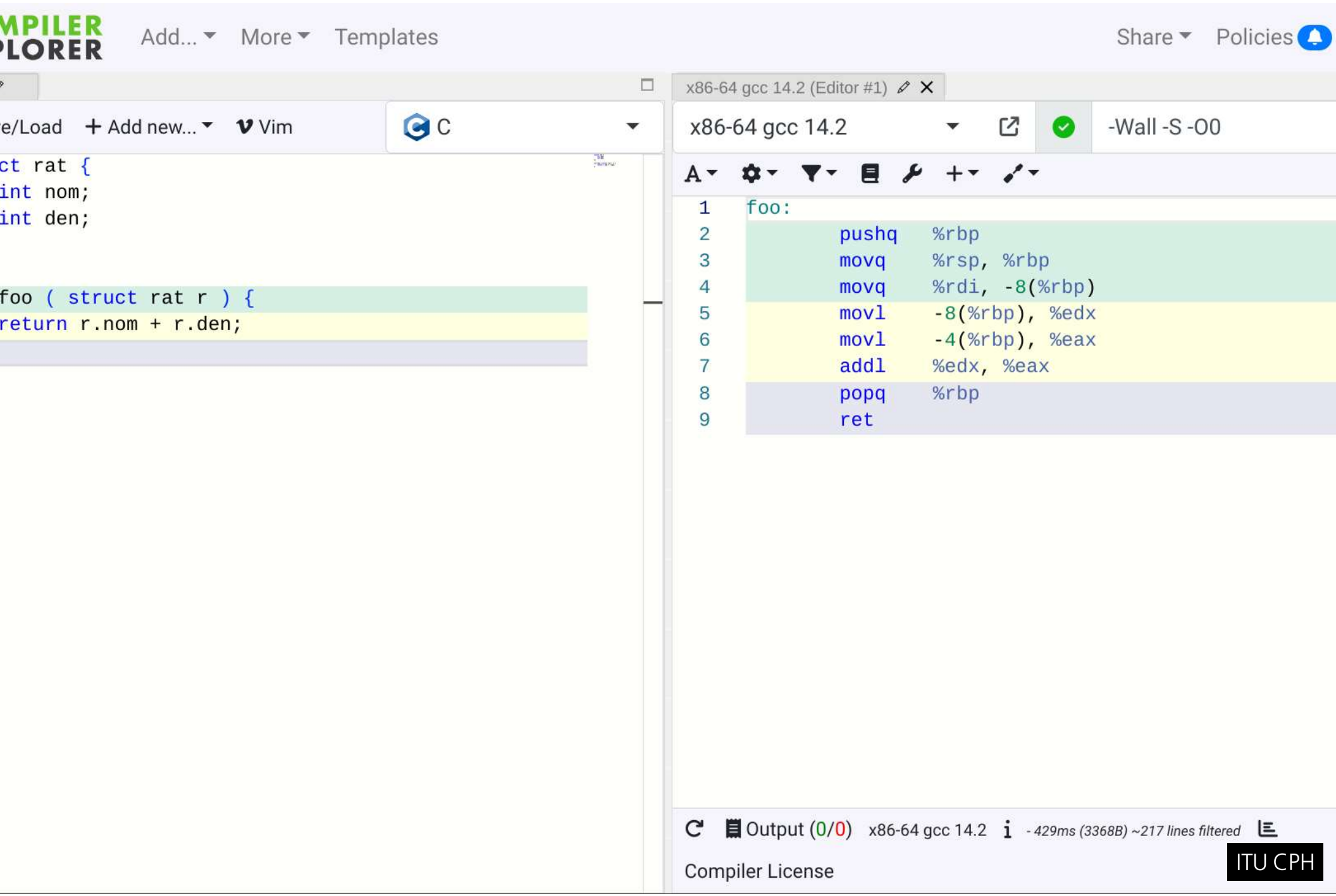

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo ( ) {
int a[4] = { 13, 14, 15, 16 };
return a[2];
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl $13, -16(%rbp)
5 movl $14, -12(%rbp)
6 movl $15, -8(%rbp)
7 movl $16, -4(%rbp)
8 movl -8(%rbp), %eax
9 popq %rbp
10 ret
Output (0/0)
x86-64 gcc 14.2
- 982ms (3228B) ~205 lines filtered
Compiler License
ITU CPH

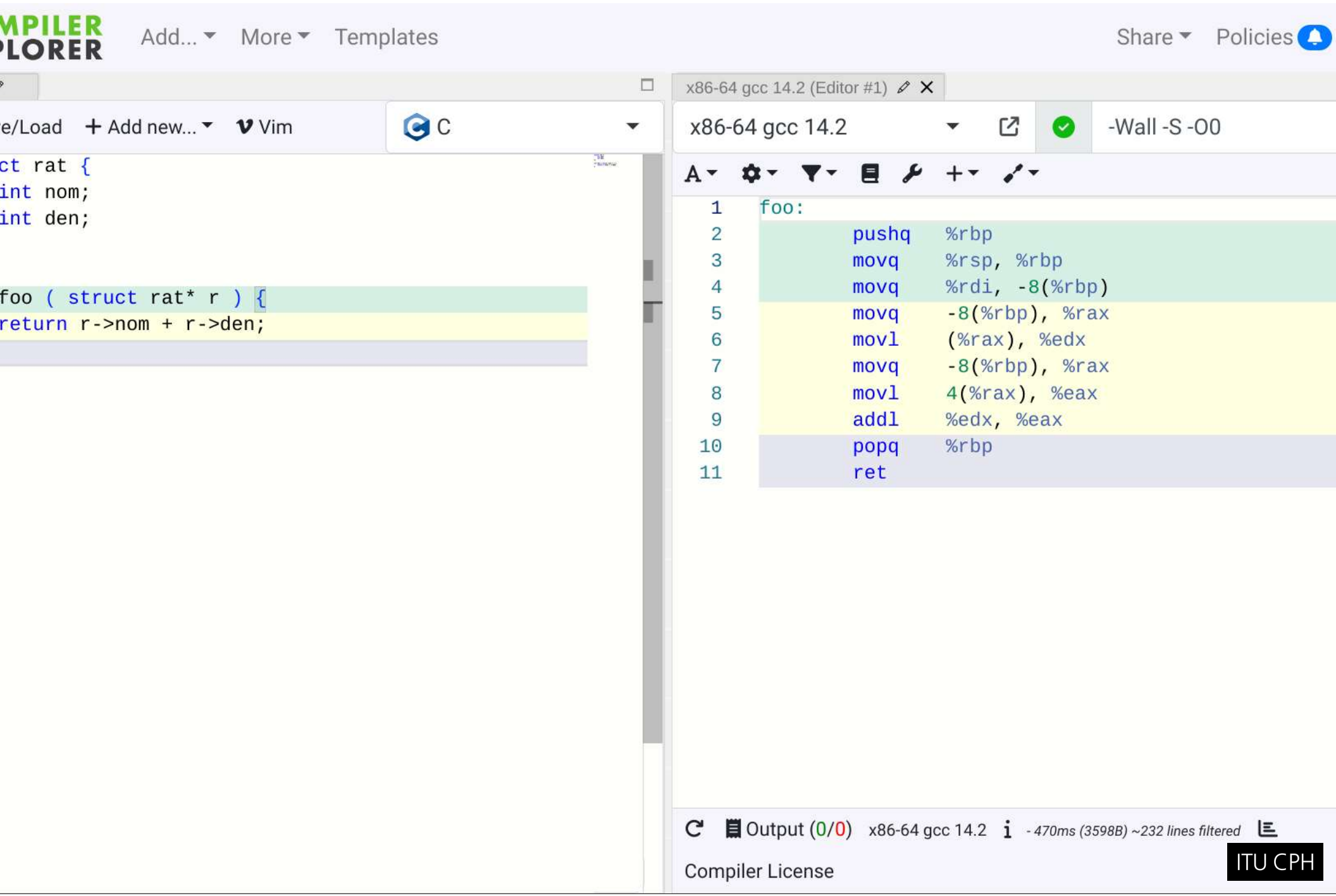

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int* x) {
return *x + 7;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 movq -8(%rbp), %rax
6 movl (%rax), %eax
7 addl $7, %eax
8 popq %rbp
9 ret
Output (0/0)
x86-64 gcc 14.2
- cached (2839B) ~177 lines filtered
Compiler License
ITU CPH

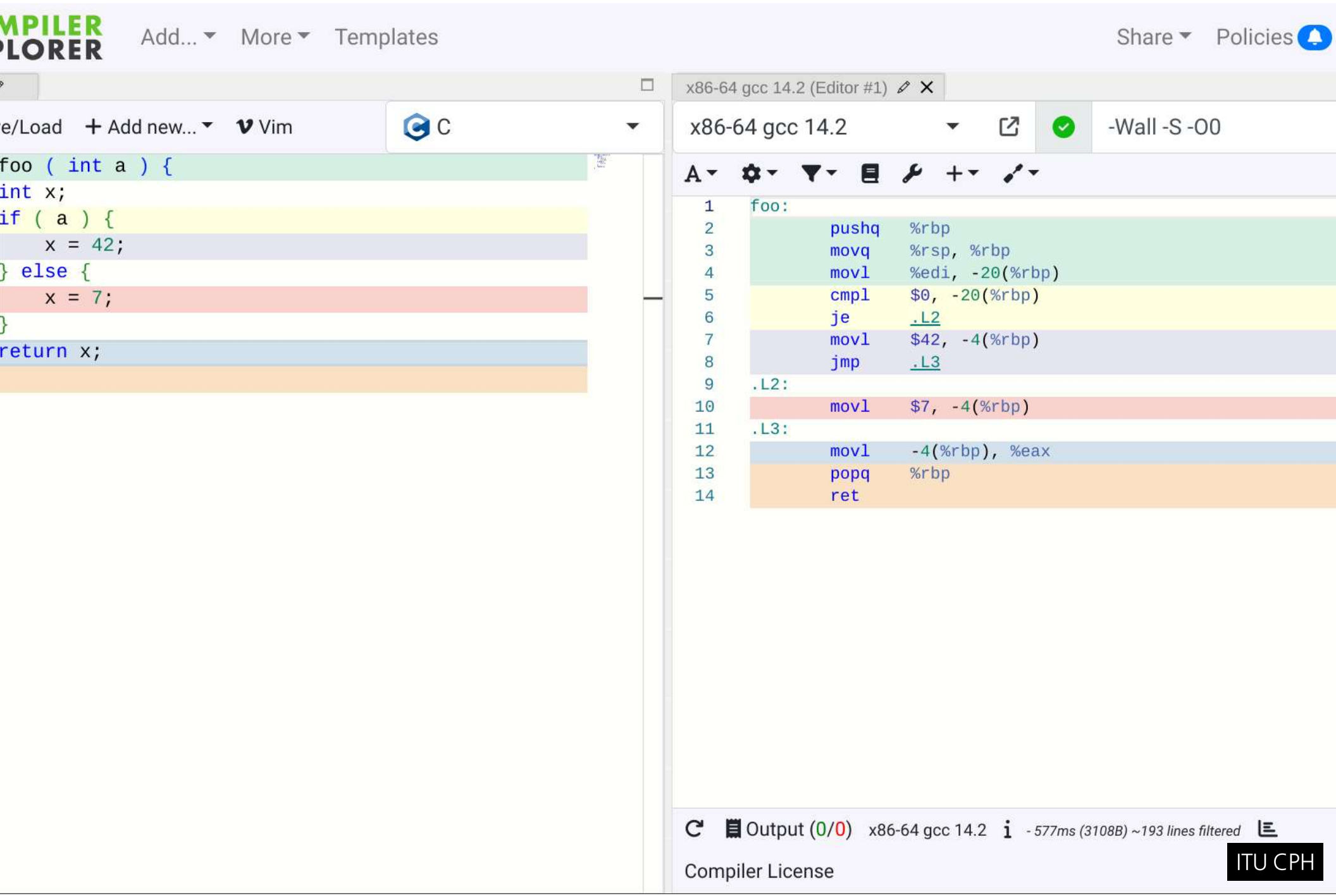

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int* x) {
return x[0] + 7;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 movq -8(%rbp), %rax
6 movl (%rax), %eax
7 addl $7, %eax
8 popq %rbp
9 ret
Output (0/0)
x86-64 gcc 14.2
- cached (2839B) ~177 lines filtered
Compiler License
ITU CPH

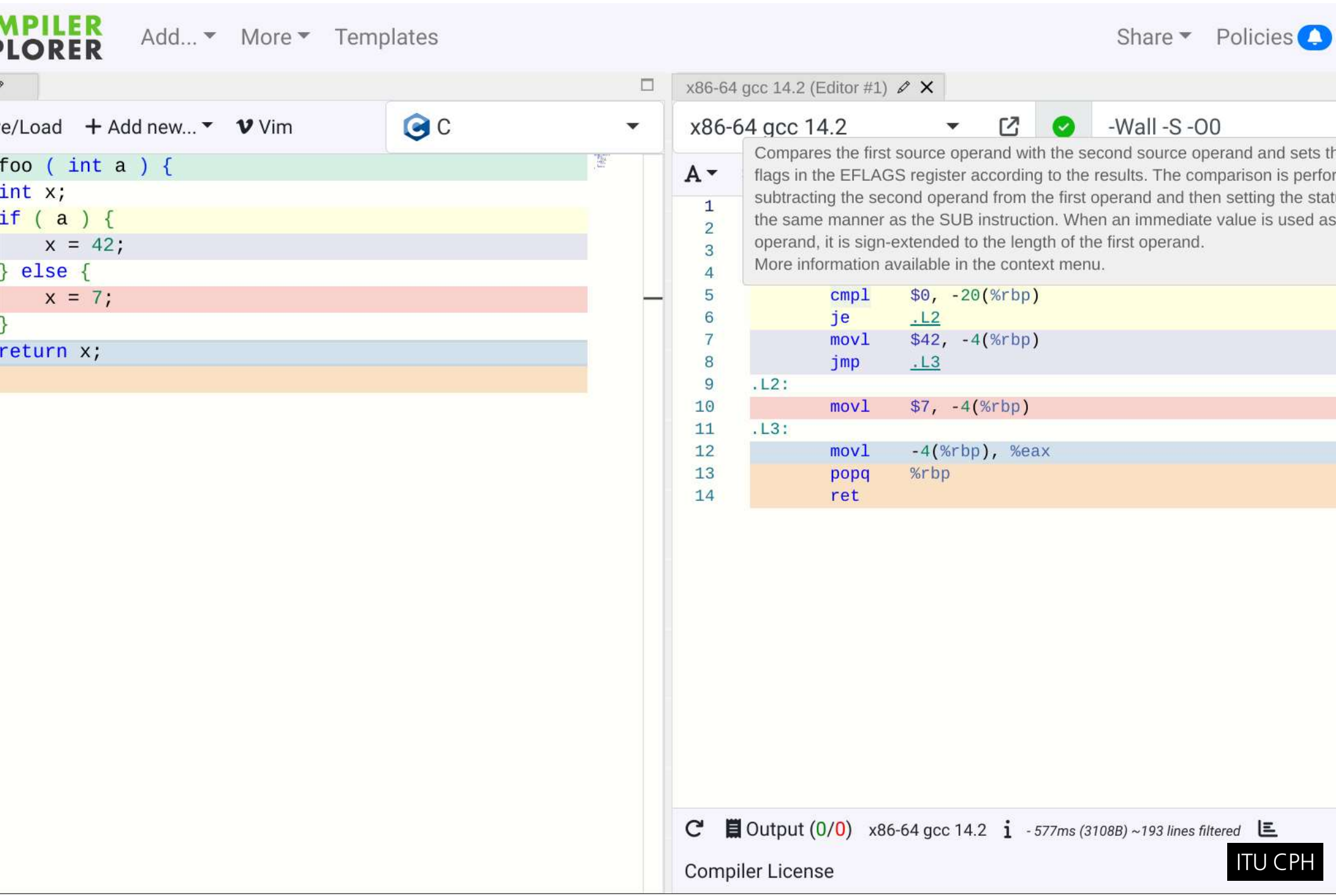

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int* x) {
return *(x + 3) + 7;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 movq -8(%rbp), %rax
6 addq $12, %rax
7 movl (%rax), %eax
8 addl $7, %eax
9 popq %rbp
10 ret
Output (0/0)
x86-64 gcc 14.2
- cached (2868B) ~178 lines filtered
Compiler License
ITU CPH

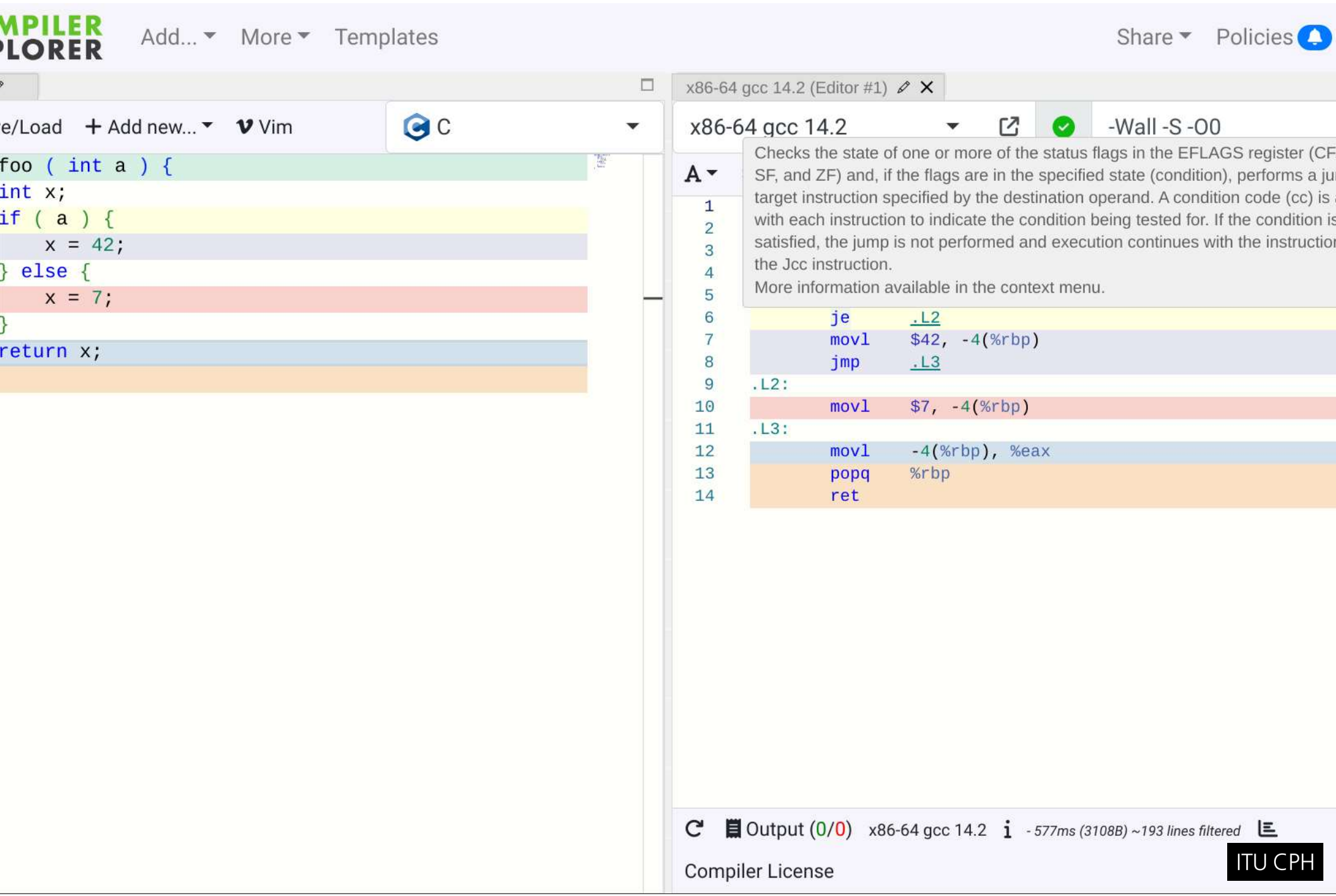

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int* x) {
return x[3] + 7;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 movq -8(%rbp), %rax
6 addq $12, %rax
7 movl (%rax), %eax
8 addl $7, %eax
9 popq %rbp
10 ret
Output (0/0)
x86-64 gcc 14.2
- cached (2855B) ~177 lines filtered
Compiler License
ITU CPH

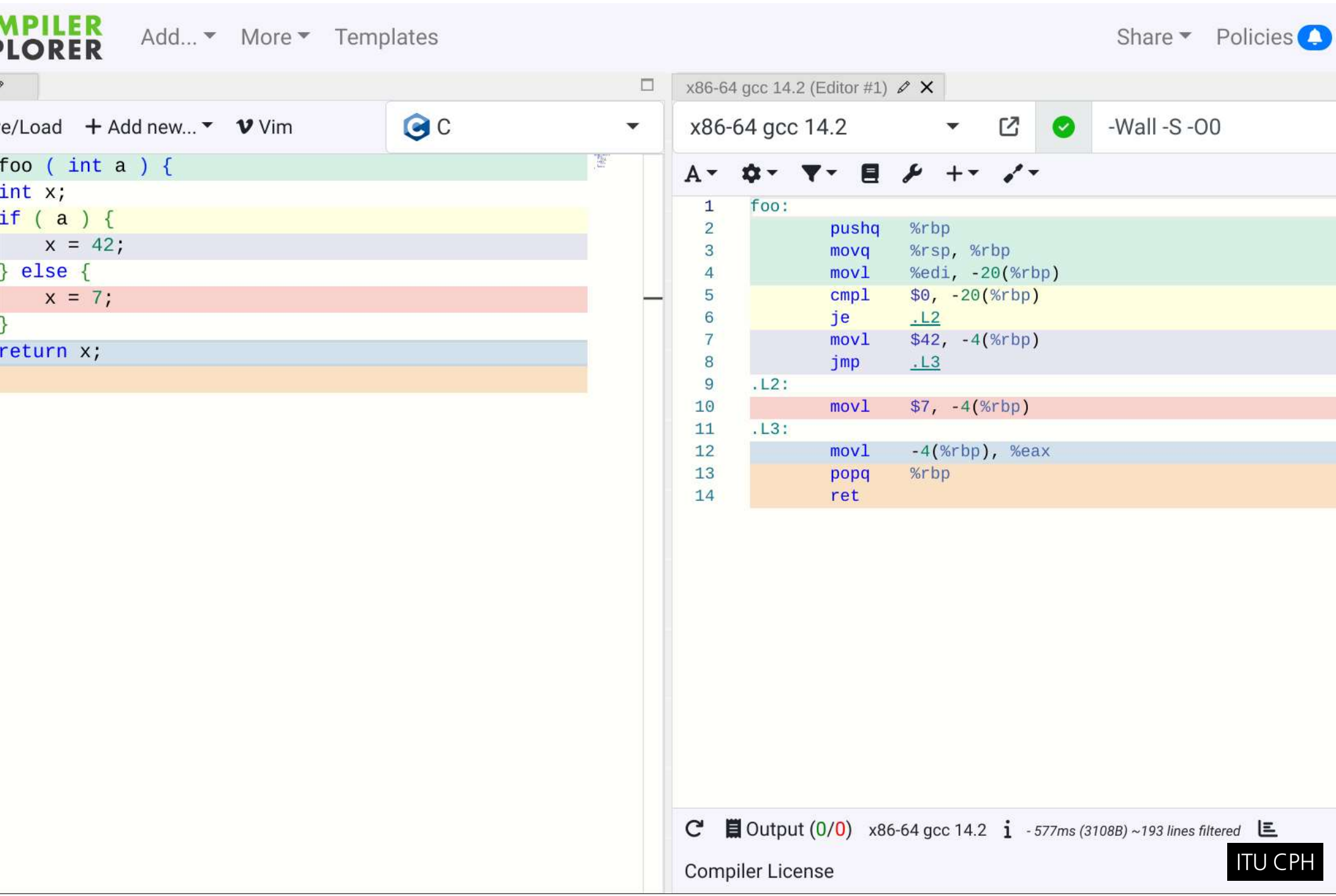

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
ct rat {
int nom;
int den;
foo (struct rat r) {
return r.nom + r.den;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 movl -8(%rbp), %edx
6 movl -4(%rbp), %eax
7 addl %edx, %eax
8 popq %rbp
9 ret
Output (0/0)
x86-64 gcc 14.2
- 429ms (3368B) ~217 lines filtered
Compiler License
ITU CPH

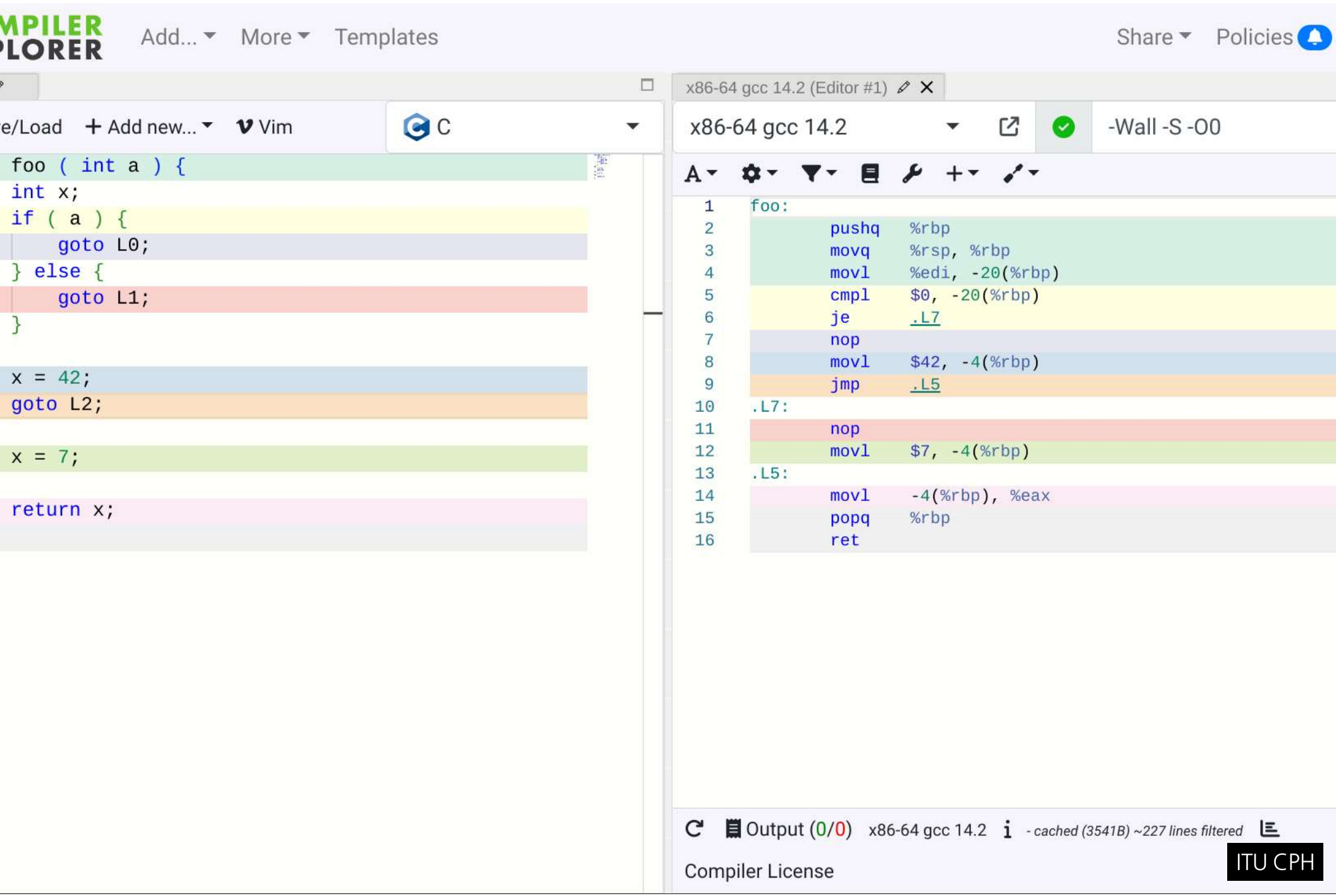

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
ct rat {
int nom;
int den;
foo (struct rat* r) {
return r->nom + r->den;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movq %rdi, -8(%rbp)
5 movq -8(%rbp), %rax
6 movl (%rax), %edx
7 movq -8(%rbp), %rax
8 movl 4(%rax), %eax
9 addl %edx, %eax
10 popq %rbp
11 ret
Output (0/0)
x86-64 gcc 14.2
- 470ms (3598B) ~232 lines filtered
Compiler License
ITU CPH

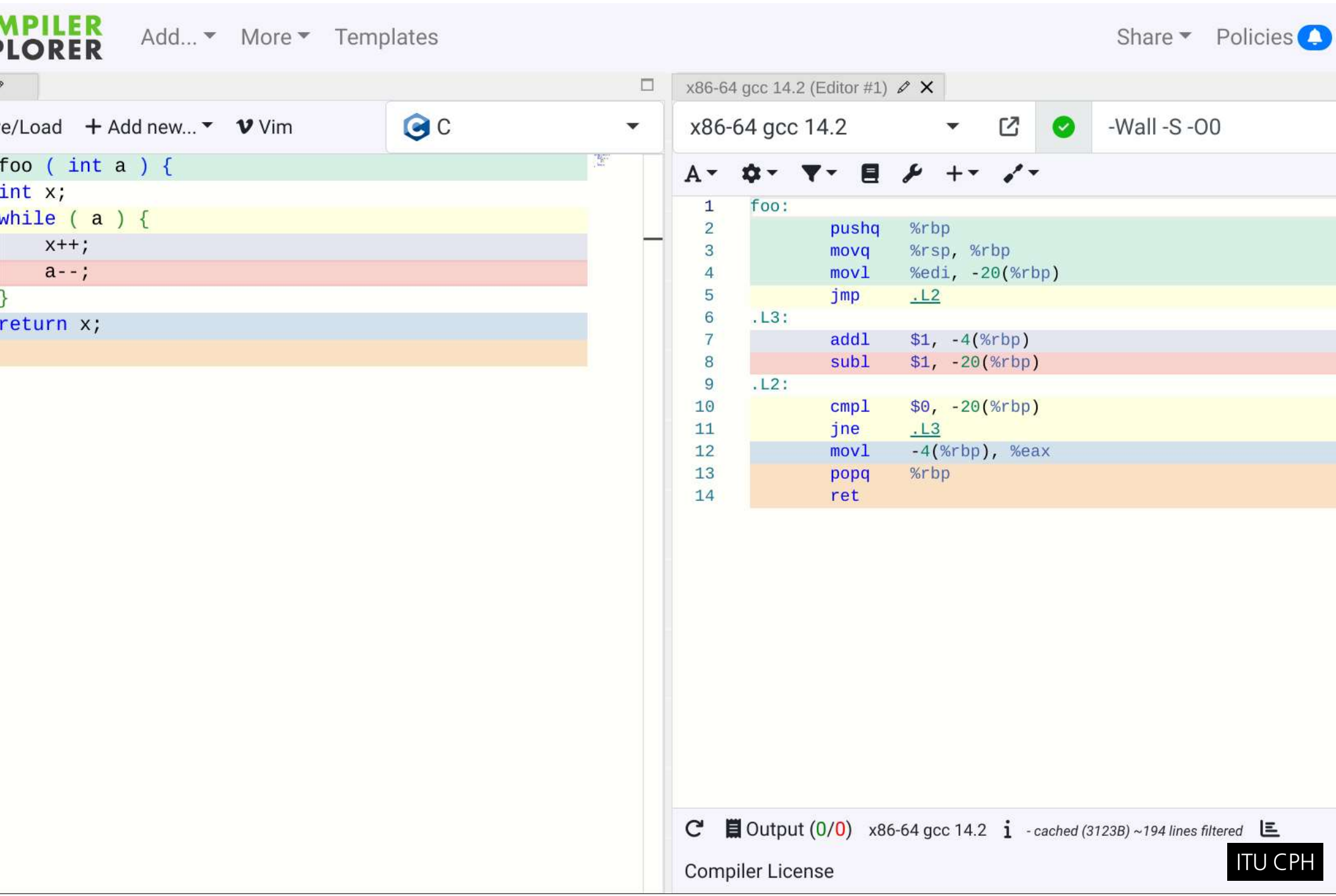

MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
foo (int a) {
int x;
if (a) {
x = 42;
} else {
x = 7;
}
return x;
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -20(%rbp)
5 cmpl $0, -20(%rbp)
6 je .L2
7 movl $42, -4(%rbp)
8 jmp .L3
9 .L2:
10 movl $7, -4(%rbp)
11 .L3:
12 movl -4(%rbp), %eax
13 popq %rbp
14 ret
Output (0/0)
x86-64 gcc 14.2
- 577ms (3108B) ~193 lines filtered
Compiler License
ITU CPH


MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
A
foo (int a) {
int x;
if (a) {
x = 42;
} else {
x = 7;
}
return x;
Compares the first source operand with the second source operand and sets th
flags in the EFLAGS register according to the results. The comparison is perfor
subtracting the second operand from the first operand and then setting the statu
the same manner as the SUB instruction. When an immediate value is used as
operand, it is sign-extended to the length of the first operand.
More information available in the context menu.
1
2
3
4
5 cmpl $0, -20(%rbp)
6 je .L2
7 movl $42, -4(%rbp)
8 jmp .L3
9 .L2:
10 movl $7, -4(%rbp)
11 .L3:
12 movl -4(%rbp), %eax
13 popq %rbp
14 ret
Output (0/0)
x86-64 gcc 14.2
- 577ms (3108B) ~193 lines filtered
Compiler License
ITU CPH


MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
foo (int a) {
int x;
if (a) {
x = 42;
} else {
x = 7;
}
return x;
Checks the state of one or more of the status flags in the EFLAGS register (CF
SF, and ZF) and, if the flags are in the specified state (condition), performs a ju
target instruction specified by the destination operand. A condition code (cc) is
with each instruction to indicate the condition being tested for. If the condition is
satisfied, the jump is not performed and execution continues with the instructio
the Jcc instruction.
More information available in the context menu.
1 je .L2
2 movl $42, -4(%rbp)
3 jmp .L3
4 .L2:
5 movl $7, -4(%rbp)
6 .L3:
7 movl -4(%rbp), %eax
8 popq %rbp
9 ret
10
11
12
13
14
Output (0/0)
x86-64 gcc 14.2
- 577ms (3108B) ~193 lines filtered
Compiler License
ITU CPH


MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
foo (int a) {
int x;
if (a) {
x = 42;
} else {
x = 7;
}
return x;
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -20(%rbp)
5 cmpl $0, -20(%rbp)
6 je .L2
7 movl $42, -4(%rbp)
8 jmp .L3
9 .L2:
10 movl $7, -4(%rbp)
11 .L3:
12 movl -4(%rbp), %eax
13 popq %rbp
14 ret
Output (0/0)
x86-64 gcc 14.2
- 577ms (3108B) ~193 lines filtered
Compiler License
ITU CPH


MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
foo (int a) {
int x;
if (a) {
goto L0;
} else {
goto L1;
}
x = 42;
goto L2;
x = 7;
return x;
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -20(%rbp)
5 cmpl $0, -20(%rbp)
6 je .L7
7 nop
8 movl $42, -4(%rbp)
9 jmp .L5
10 .L7:
11 nop
12 movl $7, -4(%rbp)
13 .L5:
14 movl -4(%rbp), %eax
15 popq %rbp
16 ret
Output (0/0)
x86-64 gcc 14.2
- cached (3541B) ~227 lines filtered
Compiler License
ITU CPH


MPILER
PLORER
Add...
More
Templates
Share
Policies
e/Load
Add new...
Vim
C
foo (int a) {
int x;
while (a) {
x++;
a--;
}
return x;
x86-64 gcc 14.2 (Editor #1)
x86-64 gcc 14.2
-Wall -S -O0
1 foo:
2 pushq %rbp
3 movq %rsp, %rbp
4 movl %edi, -20(%rbp)
5 jmp .L2
6 .L3:
7 addl $1, -4(%rbp)
8 subl $1, -20(%rbp)
9 .L2:
10 cmpl $0, -20(%rbp)
11 jne .L3
12 movl -4(%rbp), %eax
13 popq %rbp
14 ret
Output (0/0)
x86-64 gcc 14.2
- cached (3123B) ~194 lines filtered
Compiler License
ITU CPH

C: Optimizations, Compiler

ITU CPH

Performance Realities

big-O, etc.

There's more to performance than asymptotic complexity

- Constant factors matter too!

order of magnitude

  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs are **compiled** and **executed**
  - How modern **processors** + **memory** systems operate
  - How to measure program **performance** and identify **bottlenecks**
  - How to improve performance without destroying code **modularity** and **generality**

Optimizing Compilers

your task: implement alg. efficiently. compiler is your friend here! ("black box" to DS. SWU study them)

- provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

some optimizations explained in a bit

(also re-order instructions)

- do not (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
- have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

gcc: -O1, -O2, -O3, … performance vs. compile time & size of binary.

Optimizing Compilers, Limitations

compiler is conservative

- Operate under fundamental constraint
  - Must **not** cause any **change** in program **behavior**
  - Except, possibly when program making use of nonstandard language features
    - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Dead Code Elimination

despite this, compiler can help.

```
int junk ( int n ) {
  int k = 0;
  for (int i = 0; i <= n; i++){
    k += i;
  }
  return 4;
}
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Code Motion

- Reduce frequency with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop

i : row index

n : row length

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
     a[n*i+j] = b[j];
}
```

→

```
    long j;
    long ni = n*i;
    for (j = 0; j < n; j++)
     a[ni+j] = b[j];
```

Q: spot another (code motion) optimization?

Code Motion

- Reduce frequency with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop

i : row index **n** : row length

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
     a[n*i+j] = b[j];
}
```



```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
     *rowp++ = b[j];
```

Strength Reduction

shift (1 cycle) is way more efficient than multiplication (3 cycles)

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x --> x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction

    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

(think of multiplication as a sequence of ad

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```

```
imulq%rcx, %rsi  # i*n
addq %rdx, %rsi  # i*n+j
movq %rsi, %rax  # i*n+j
subq %rcx, %rax  # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

C: Optimizations, Manual

ITU CPH

Convert to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Q: `lower` is quite slow. why?

Calling Strlen

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Q: `lower` is quite slow. why?

- strlen performance
  - Only way to determine length of string is to scan its entire length, looking for null charact
- Overall performance, string of length N
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Improving Performance

```
void lower2(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

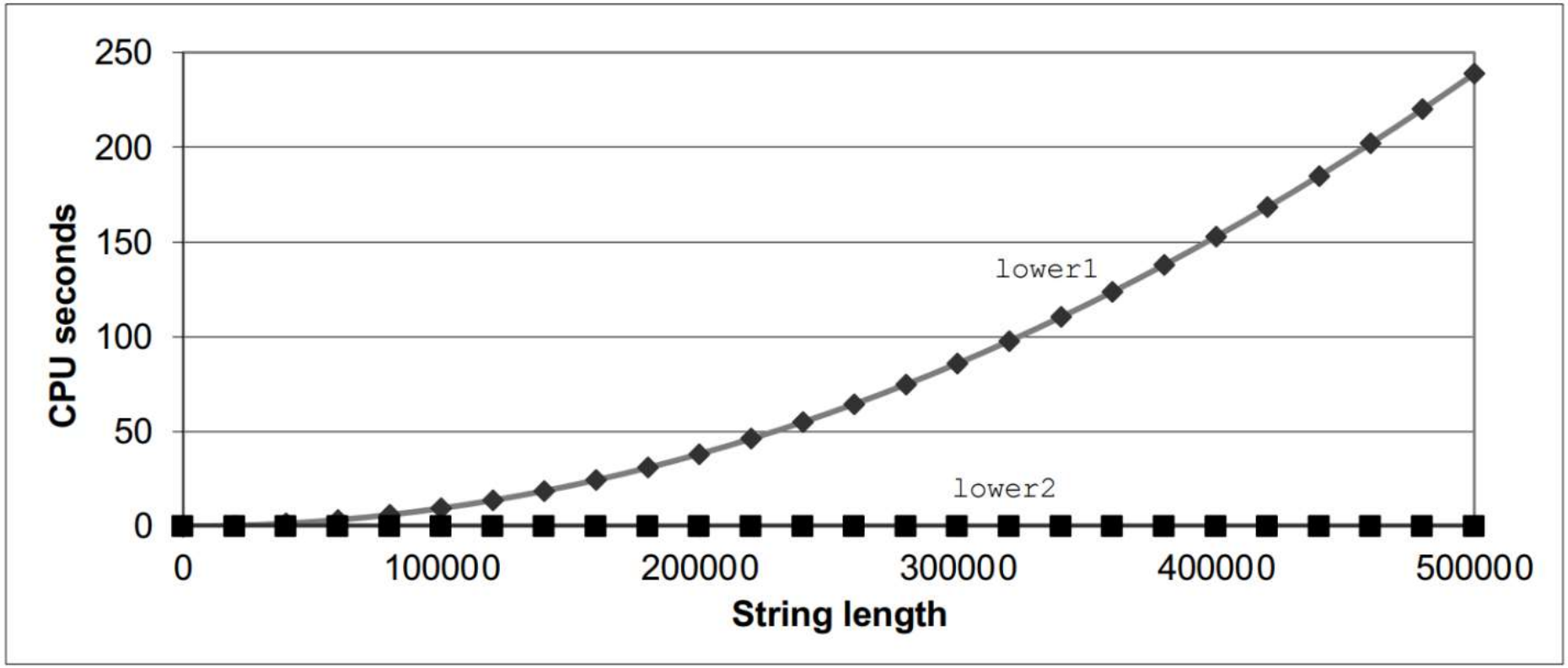
Move call to `strlen` outside of loop

Since result does not change from one iteration to another

Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



Optimization Blocker: Procedure Calls

Here is what an optimizing compiler does to this code:

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Optimization Blocker: Procedure Calls

- *Compiler won't move* `strlen` *out of inner loop. Why won't it?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- Remedies:
  - Use of inline functions
    - GCC does this with –O1

      Within single file
  - Do your own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    lencnt += length;
    return length;
}
```

Unnecessary movs?

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0     # FP load
        addsd   (%rdi), %xmm0            # FP add
        movsd   %xmm0, (%rsi,%rax,8)     # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

why doesn't the compiler kee
the intermediate results in a
register, and write register to
mem when done?
(would be 100x fas

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

Value of B:

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

Q: first suppose we had
`double B[3] = {42,42,42};`
what are final values in B?

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

ITU CPH

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

Value of B:

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

Q: first suppose we had
`double B[3] = {42,42,42};`
what are final values in B?

`final: [3, 28, 224]`

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

ITU CPH

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

Value of B:

```
init:  [4, 8, 16]
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

Q: first suppose we had
`double B[3] = {42,42,42};`
what are final values in B?

```
final: [3, 28, 224]
```

Q: now, for B as defined on left, what are intermediate & final B?

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

ITU CPH

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

we just updated part of A

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

Q: first suppose we had
`double B[3] = {42,42,42};`
what are final values in B?

```
final: [3, 28, 224]
```

Q: now, for B as defined on left, what are intermediate & final B?

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

ITU CPH

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:  [4, 8, 16]
```

we just updated part of A

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

reading & writing to same row

Q: first suppose we had
`double B[3] = {42,42,42};`
what are final values in B?

```
final: [3, 28, 224]
```

Q: now, for B as defined on left, what are intermediate & final B?

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

ITU CPH

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     b[i] = 0;
     for (j = 0; j < n; j++)
         b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double *B = A+3;

sum_rows1(A, B, 3);
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

we just updated part of A

reading & writing to same row

Q: first suppose we had
`double B[3] = {42,42,42};`
what are final values in B?

```
final: [3, 28, 224]
```

Q: now, for B as defined on left, what are intermediate & final B?

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

UNIVERSITY OF COPENHAGEN

ITU CPH

Optimization Blocker: Memory Aliasing

Aliasing

Two **different** memory **references** specify **single location**

Easy to have happen in C

- Since allowed to do address arithmetic
- Direct access to storage structures

Get in habit of introducing **local variables**

- **Accumulating within loops**
- Your way of telling compiler not to check for aliasing

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     double val = 0;
     for (j = 0; j < n; j++)
         val += a[i*n + j];
         b[i] = val;
    }
}
```

Value of B:

```
init:  [4, 8, 16]
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0  # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

no mov instruction;
100x faster.

Now `val` cannot be an alias for cells in `a`. (in inner loop)

No need to store intermediate results

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     double val = 0;
     for (j = 0; j < n; j++)
         val += a[i*n + j];
         b[i] = val;
    }
}
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0  # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

no mov instruction; 100x faster.

Now `val` cannot be an alias for cells in `a`. (in inner loop)

No need to store intermediate results

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     double val = 0;
     for (j = 0; j < n; j++)
         val += a[i*n + j];
         b[i] = val;
    }
}
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 27, 16]
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0  # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

no mov instruction;
100x faster.

Now `val` cannot be an alias for cells in `a`. (in inner loop)

No need to store intermediate results

Removing Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
     double val = 0;
     for (j = 0; j < n; j++)
         val += a[i*n + j];
         b[i] = val;
    }
}
```

Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 27, 16]
```

```
i = 2: [3, 22, 224]
```

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0  # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

no mov instruction;
100x faster.

Now `val` cannot be an alias for cells in `a`. (in inner loop)

No need to store intermediate results

C: Optimizations, Example

ITU CPH

Exploiting Instruction-Level Parallelism

CPUs are in fact no
“1 instruction at a
time” interpreters as
we introduced them

- Need general understanding of modern processor design

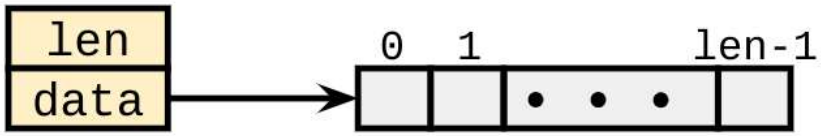
  Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement

  Compilers often cannot make these transformations

  Lack of associativity and distributivity in floating-point arithmetic

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



intuition holds if `data` in a vec points to array of length `len`

Data Types

Use different declarations for `data_t`

- `int`
- `long`
- `float`
- `double`

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

intuition holds if `val` just points to a data_t not an array

UNIVERSITY OF COPENHAGEN

ITU CPH

Benchmark Computation

```
typedef vec* vec_ptr;
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

in benchmark: compute
sum or **product**
of vector elements

Data Types

Use different declarations for `data_t`

- `int`
- `long`
- `float`
- `double`

Operations

Use different definitions of `OP` and `IDENT`

- `+ / 0`
- `* / 1`

Benchmark Performance

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

twice as fast

OF COPENHAGEN

ITU CPH

Basic Optimizations

let's apply the optimizations tha
we've learned about so far.
(help the compiler help us)

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

rder of magnitude
n top of previous
nprovement!
his really pays off!)

Eliminates sources of overhead in loop

UNIVERSITY OF COPENHAGEN

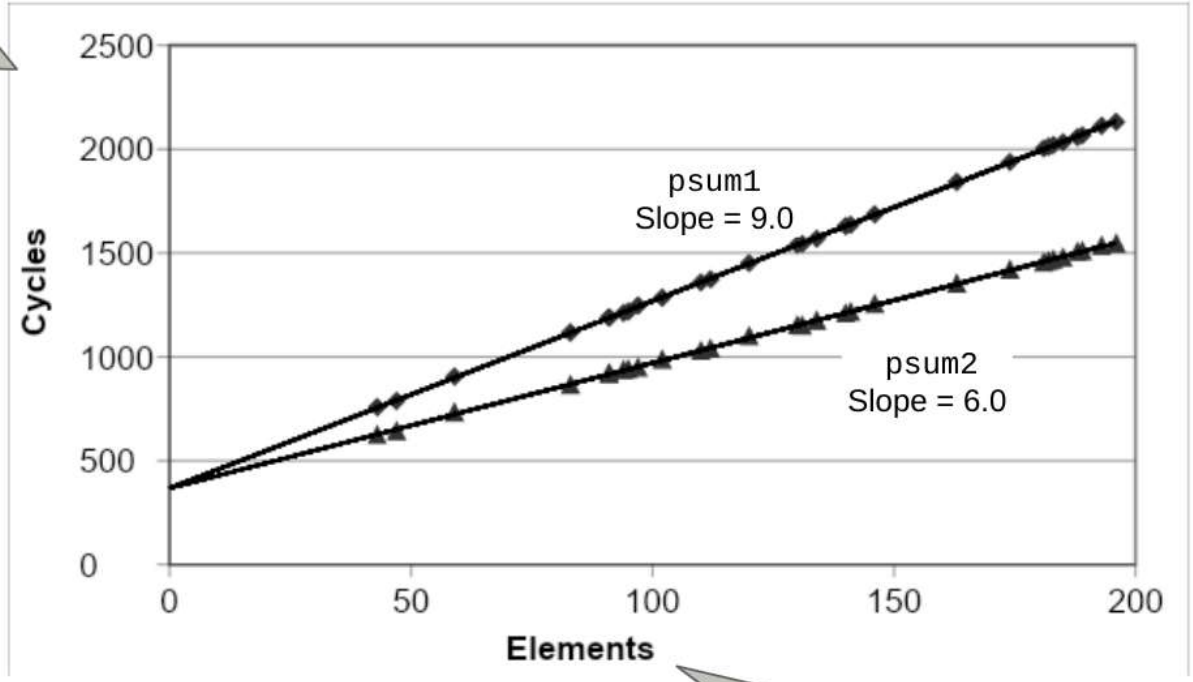
if we can do more than 1 op at a time,
then we can go below “1 op per element in sequen

ITU CPH

Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- Length = n
- In our case: CPE = cycles per OP
- T = CPE*n + Overhead
  - CPE is slope of line

example of how to present such a benchmark result:



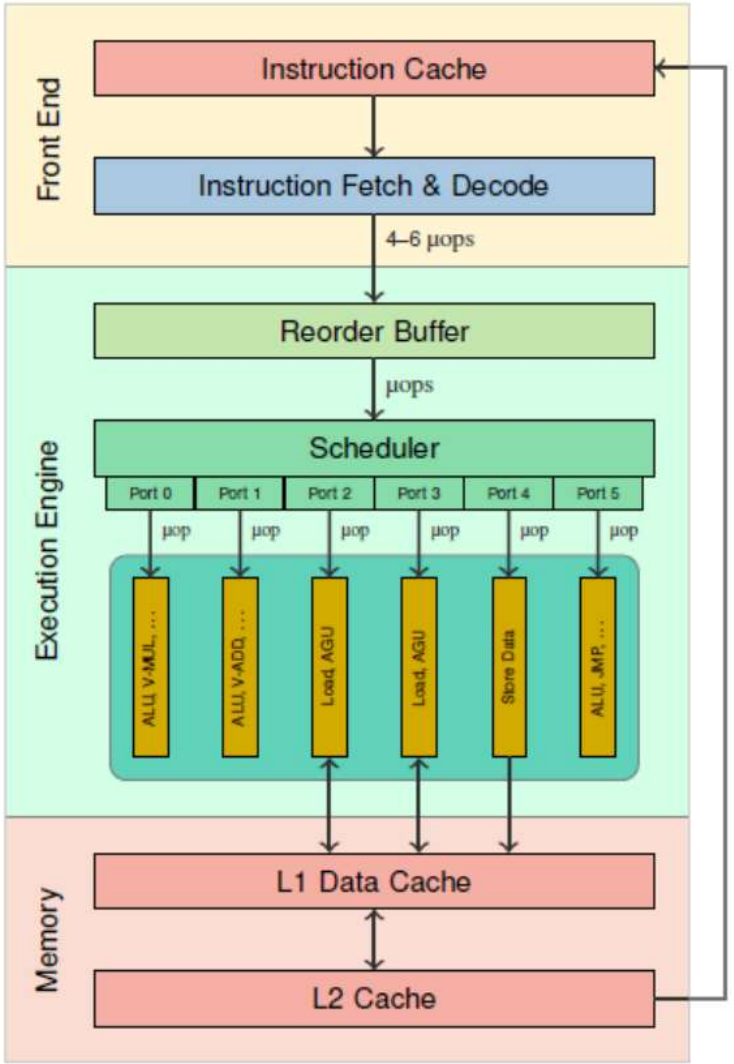
CPU time (cycles spent)

length of vector

UNIVERSITY OF COPENHAGEN

ITU CPH

Modern CPU Design



front end
accesses L2 cache.

back end
accesses L1 cache, and if necessary L2 cache.

execution engine
reorders & schedules instructions (to different ports.
ports = how many micro-operations **at the same time**. each instruction is 1+ micro-operation. recent version of ARM processor has 4 ports)

each port is pipelined (stages).

Back End, linked to memory

https://uops.info/background.html#supportedMicroarc

Superscalar Processor

Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

program instructions not necessarily executed in order.

Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

Most modern CPUs are superscalar.
Intel: since Pentium (1993)

x86-64 Compilation of Combine4

Inner Loop (Case: Integer Multiply)

```
.L519:          # Loop:
     imull(%rax,%rdx,4), %ecx  # t = t * d[i]
     addq $1, %rdx   # i++
     cmpq %rdx, %rbp# Compare length:i
     jg   .L519# If >, goto Loop
```

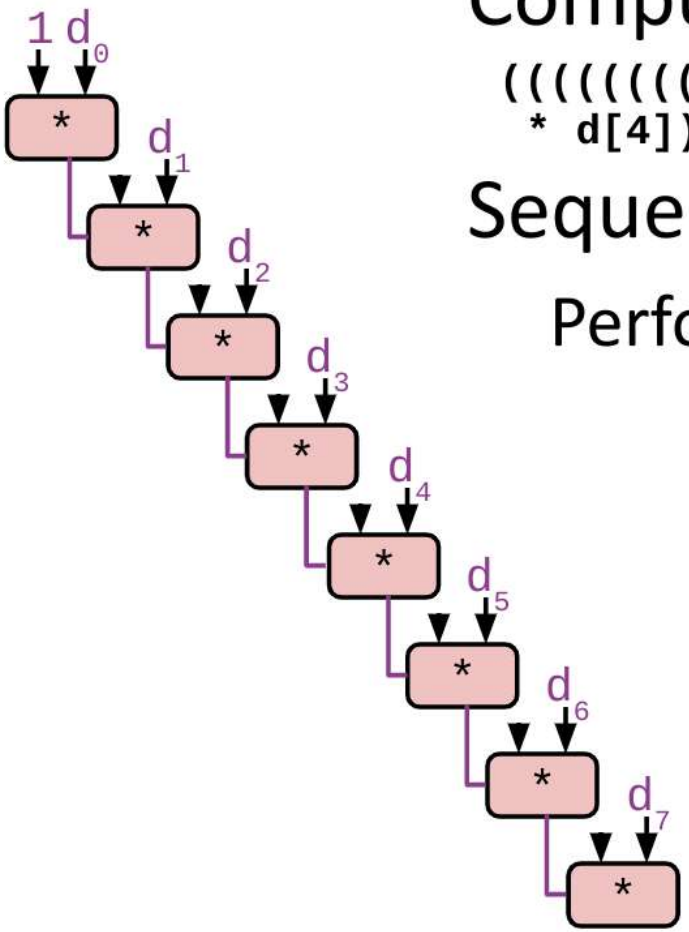
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

we are already pretty close to
theoretical bound
w/o using pipelining.
can we get closer?

UNIVERSITY OF COPENHAGEN

ITU CPH

Combine4 = Serial Computation (OP = *)



Computation (length=8)

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

Sequential dependence

Performance: determined by latency of OP

w/o pipelining: serial.
how to do better?
loop unrolli

UNIVERSITY OF COPENHAGEN ITU CPH

Loop Unrolling (2x1)

we give the CPU opportunity to run ops i parallel…

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit  = length-1;
    data_t *d   = get_vec_start(v);
    data_t x    = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
     x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
     x = x OP d[i];
    }
    *dest = x;
}
```

… by combining 2 elements at a time within 1 iteration of the loop.

Perform 2x more useful work per iteration

UNIVERSITY OF COPENHAGEN

ITU CPH

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Helps integer add

Achieves latency bound

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Helps integer add

Achieves latency bound

```
x = (x OP d[i]) OP d[i+1];
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Helps integer add

Achieves latency bound

Others don't improve. *Why?*

```
x = (x OP d[i]) OP d[i+1];
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Helps integer add

Achieves latency bound

Others don't improve. *Why?*

Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

UNIVERSITY OF COPENHAGEN

ITU CPH

Loop Unrolling with Reassociation (2x1a)

let's break sequential dependency

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
     x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
     x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

Can this change the result of the computation?

Yes, for FP. *Why?*

UNIVERSITY OF COPENHAGEN

ITU CPH

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

theoretical bound w/ parallelism taken into account (how many ports available, how many ports each instruction needs)

2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

UNIVERSITY OF COPENHAGEN

ITU CPH

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

theoretical bound w/ parallelism taken into account (how many ports available, how many ports each instruction needs)

Nearly 2x speedup for Int *, FP +, FP *

2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

theoretical bound w/ parallelism taken into account (how many ports available, how many ports each instruction needs)

Nearly 2x speedup for Int *, FP +, FP *

Reason: **Breaks sequential dependency**

2 func. units for FP *
2 func. units for load

4 func. units for int +
2 func. units for load

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

theoretical bound w/ parallelism taken into account (how many ports available, how many ports each instruction needs)

Nearly 2x speedup for Int *, FP +, FP *

Reason: **Breaks sequential dependency**

```
x = x OP (d[i] OP d[i+1]);
```

Why is that? (next slide)

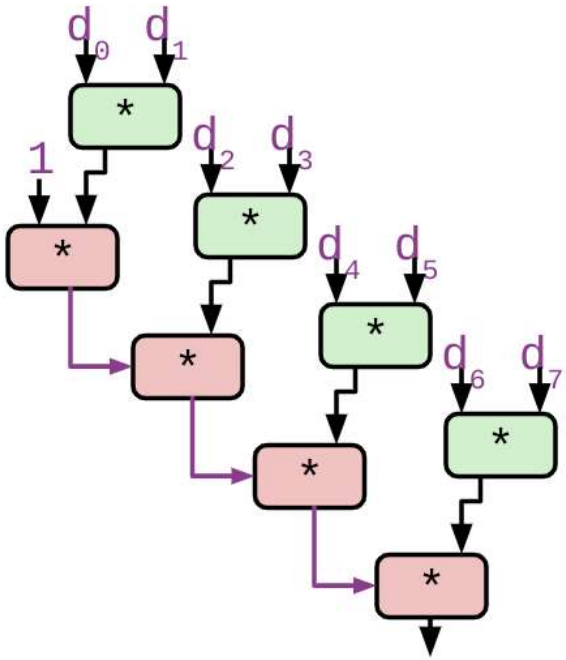
4 func. units for int +
2 func. units for load

2 func. units for FP *
2 func. units for load

Reassociated Computation

sequential dependency broken; can do more things at the same time

```
x = x OP (d[i] OP d[i+1]);
```



What changed:

Ops in the next iteration can be started early (no dependency)

Overall Performance

N elements, D cycles latency/op

(N/2+1)*D cycles:

CPE = D/2

UNIVERSITY OF COPENHAGEN

ITU CPH

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
     x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

can do even better:
two accumulators.
(2 chains instead of 1)

Different form of reassociation

UNIVERSITY OF COPENHAGEN

ITU CPH

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

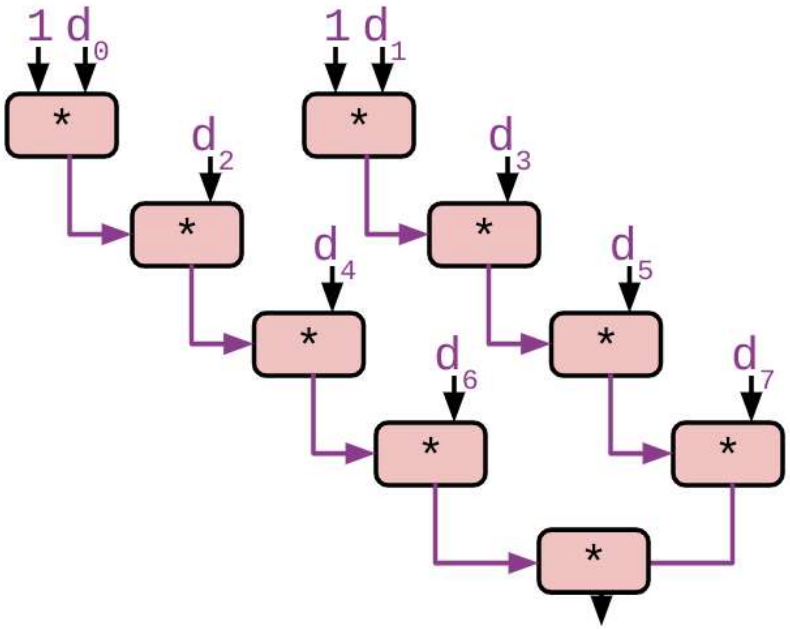
Int + makes use of two load units

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

2x speedup (over unroll2) for Int *, FP +, FP *

Separate Accumulators

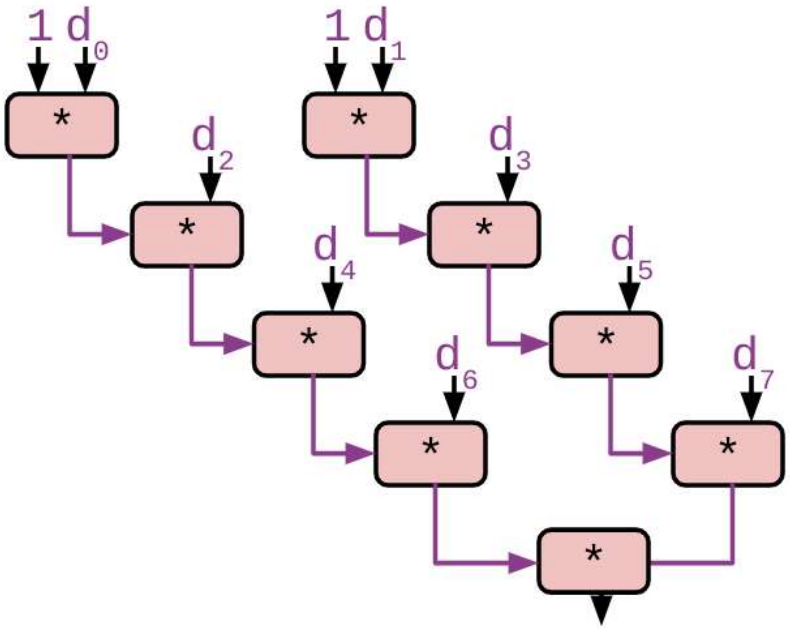
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent "streams" of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent "streams" of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

Unrolling & Accumulating

- Idea

  Can unroll to any degree L

  Can accumulate K results in parallel

  L must be multiple of K

how many ops at a time depends on the ops (how parallelizable), and numbe of ports (resources).

- Limitations

  Diminishing returns

  – Cannot go beyond throughput limitations of # ports

  Large overhead for short lengths

  – Finish off iterations sequentially

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Limited only by throughput of functional units

Up to **42X improvement** over original, unoptimized code

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

Make use of AVX Instructions

another order of magnitude improvement

Parallel operations on multiple data elements

See Web Aside OPT:SIMD on CS:APP web page

What About Branches?

Challenge

Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

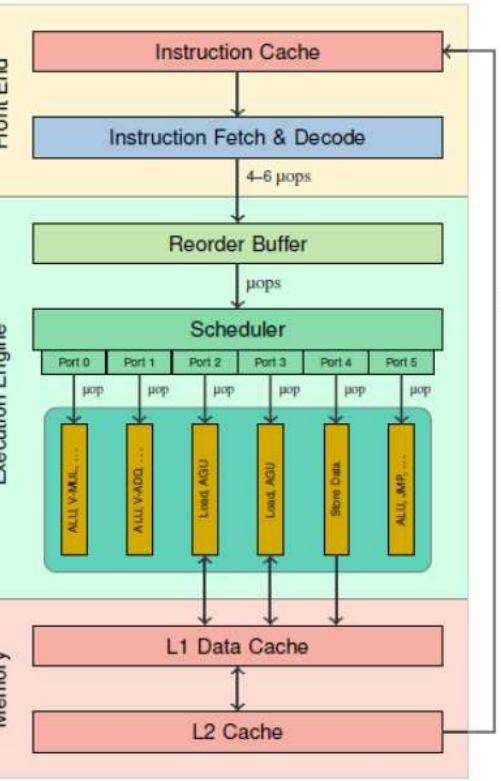
```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

 . . .

404685:  repz retq
```

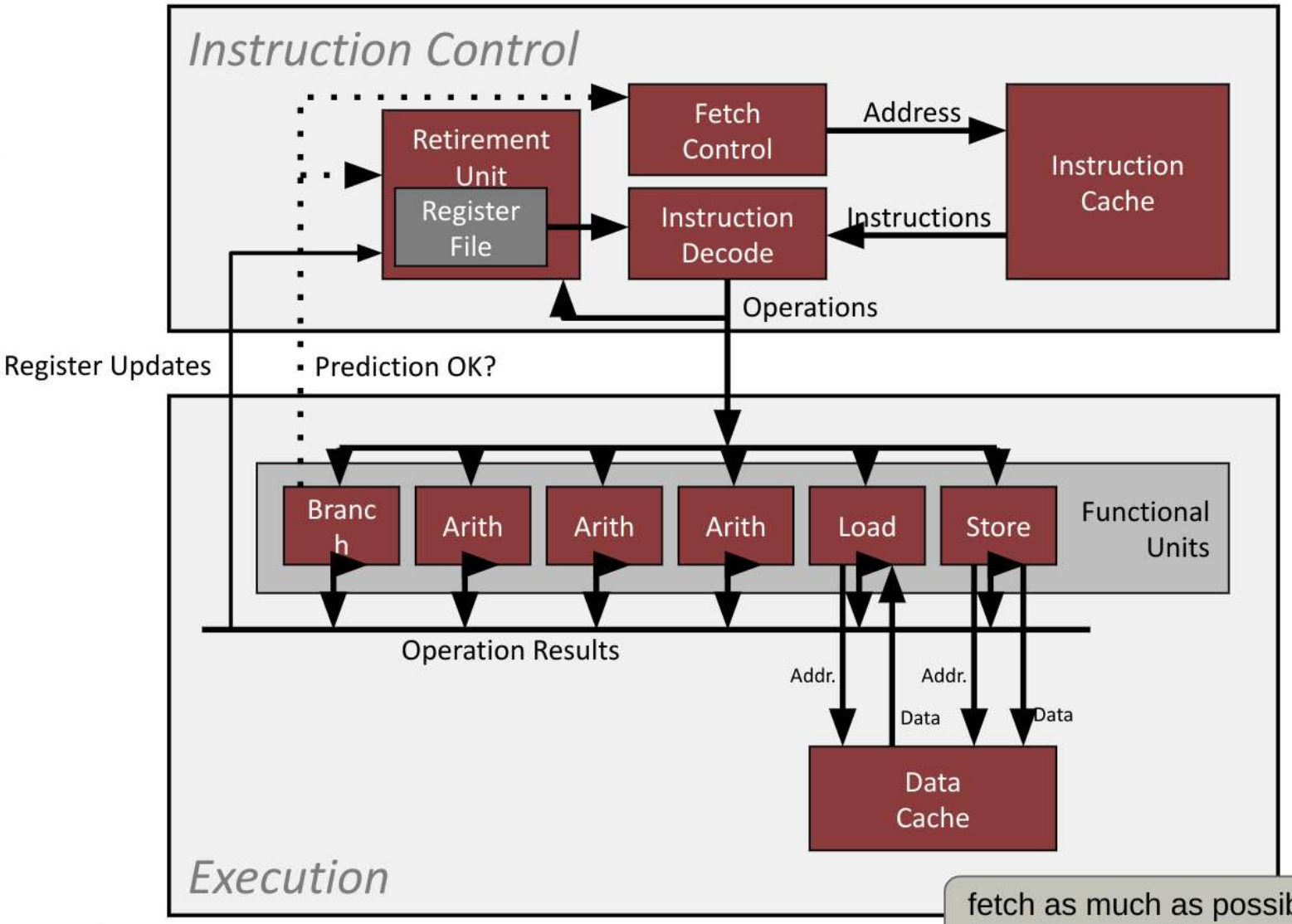
Executing

How to continue?

When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design





fetch as much as possible t
keep front end busy.

UNIVERSITY OF COPENHAGEN

ITU CPH

Branch Outcomes

"look, you are going throug a loop. so I am going to guess 'true', and prefetch"

When encounter conditional branch, cannot determine where to continue fetching

- Branch Taken: Transfer control to branch target
- Branch Not-Taken: Continue with next instruction in sequence

Cannot resolve until outcome determined by branch/integer unit

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

 . . .

404685:  repz retq
```

Branch Prediction

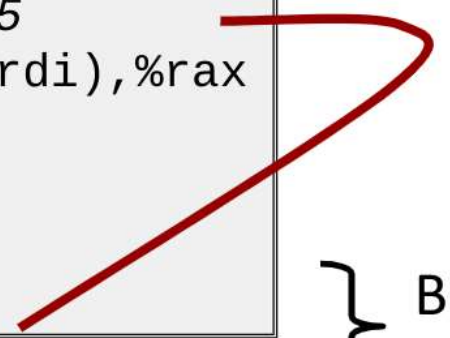
- Idea

  Guess which way branch will go

  Begin executing instructions at predicted position

  – But don't actually modify register or memory data

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

 . . .

404685:  repz retq
```



Begin Execution

Branch Prediction Through Loop

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 100

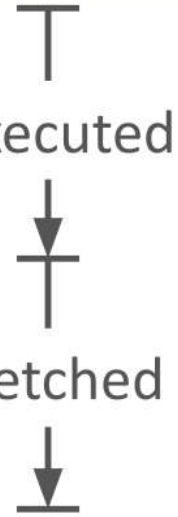
Read invalid location

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Assume
vector length = 100

branch mis-prediction
(aka. prediction failure)



UNIVERSITY OF COPENHAGEN

ITU CPH

Branch Misprediction Invalidation

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 100

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Invalidate

Assume
vector length = 100

pipeline full of invalidated instructions (mis

Branch Misprediction Recovery

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
401036:  jmp    401040
 . . .
401040:  vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

Performance Cost

- Multiple clock cycles on modern processor
- Can be a major performance limiter

Take-Aways

- Understand compiler optimizations
  ⇒ You can help the compiler do them for you.
- Don't do anything stupid

  Watch out for hidden algorithmic inefficiencies

  Write compiler-friendly code
  - Watch out for optimization blockers: procedure calls & memory references

  Look carefully at innermost loops (where most work is done)
- Tune code for machine

  Exploit instruction-level parallelism

  Avoid unpredictable branches

  Make code cache friendly (see last week => blocking)