Compilers

Chapter 1: Introduction

- Lecturers:
  - Paul Kelly (p.kelly@imperial.ac.uk)
  - Jamie Willis (j.willis@imperial.ac.uk)
  - Hongxiang Fan (hongxiang.fan@imperial.ac.uk)

Acknowledgements: Naranker Dulay and Chris Hankin, who have both contributed to the course design

- This course is about a particular class of programs called *language processors*, of which the best example is a compiler.

What is a compiler?

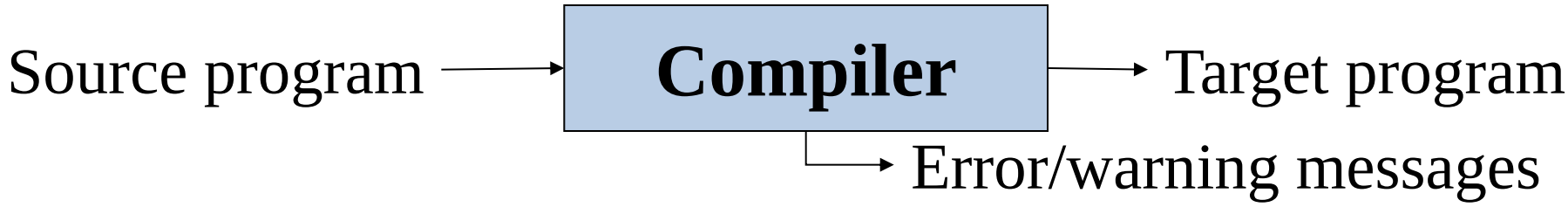
- A program which ***processes*** programs, written in some programming language.
- A program which ***writes*** programs (in some language).
- A compiler ***translates*** programs written in one language into “equivalent” programs in another language.

- This course is about a particular class of programs called *language processors*, of which the best example is a compiler.

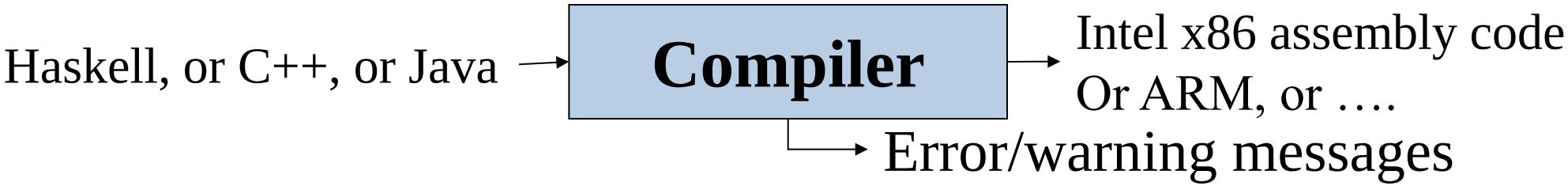
What is a compiler?

- A program which ***processes*** programs, written in some programming language.
- A program which ***writes*** programs (in some language).
- A compiler ***translates*** programs written in one language into “equivalent” programs in another language

- A tool to ***enable you to program at a higher level***, by mapping high-level concepts to low-level implementation

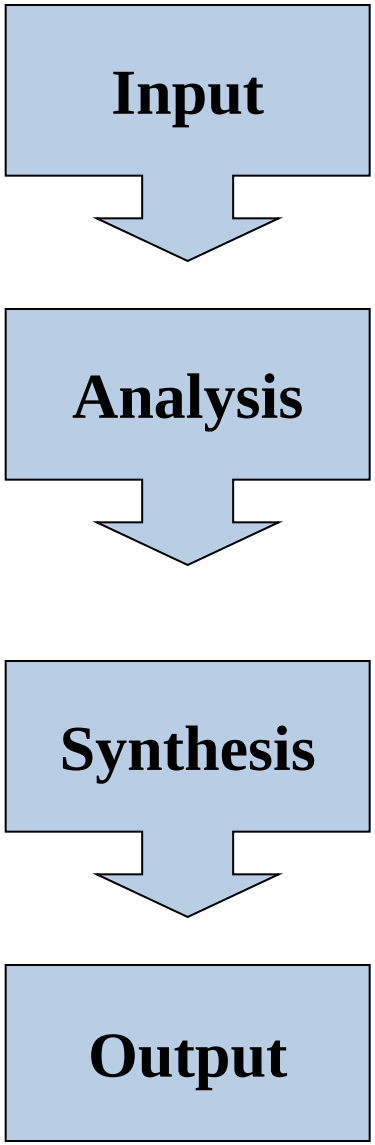


For example:



- Translates from one language into another
- **Or**: Output a low-level program which behaves as specified by the input, higher-level program.
- **That is**: Mediate between higher-level human concepts, and the word-by-word data manipulation which the machine performs.

Basic compiler structure



In some language

Construct an internal representation of the source language structure, and hence its meaning

Use this internal representation to construct target language version

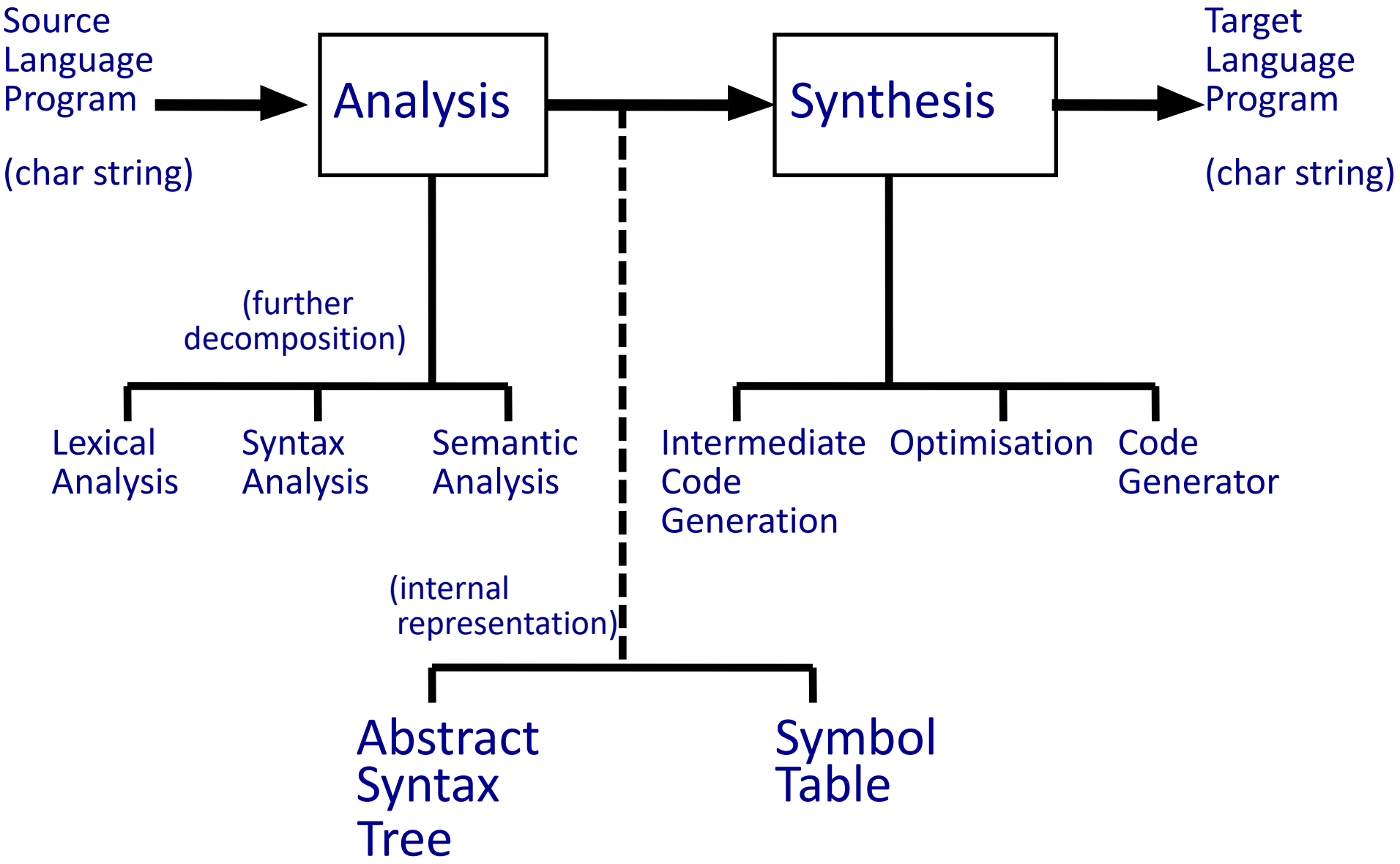
In the target language

Eg C, Java, C#

Usually start by building a tree representation, but may build graph, eg to represent control flow

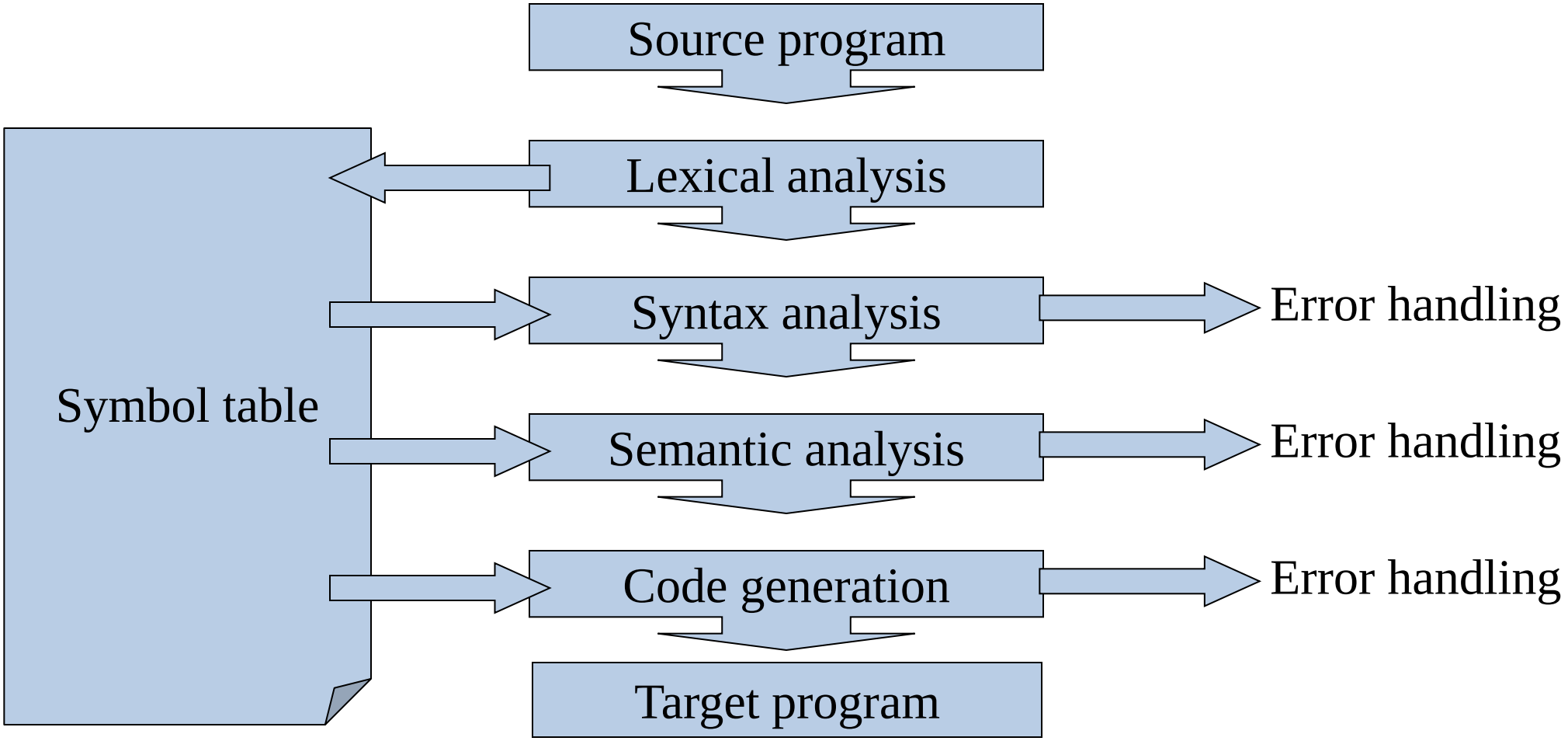
Analyse and transform the internal representation, then traverse it to produce output

Eg in Intel x86 assembler



Compiler structure in more detail

The phases of a compiler



Information from declarations is gathered in the symbol table, and is used to check how each variable is used, to reserve memory for it, and to generate code to access it

Phases of a compiler - example

- *Input file* "`test.c`":

```
int A;
int B;
test_fun()
{
   A = B+123;
}
```

- Command: "`gcc -S -O test.c`"

  (the flag "`-S`" tells the compiler to produce assembly code, "`-O`" turns optimisation on).

- *Output file* "`test.s`":

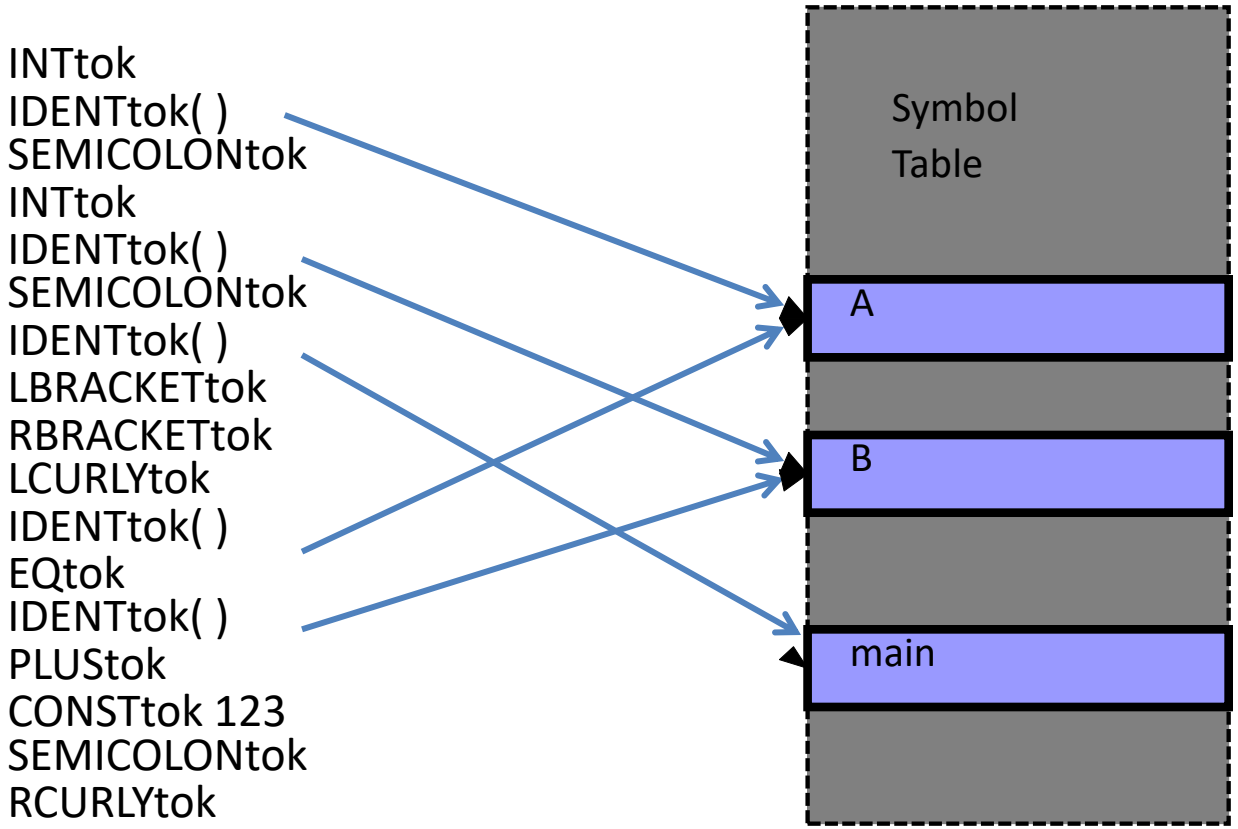
```
        .comm   _A, 4
        .comm   _B, 4
_test_fun:
        pushl %ebp
        movl %esp,%ebp
        movl _B,%eax
        addl $123,%eax
        movl %eax,_A
        movl %ebp,%esp
        popl %ebp
        ret
```

Introduction to lexical analysis

- INPUT: sequence of characters

int$_{sp}$A;$_{nl}$int$_{sp}$B;$_{nl}$main()$_{nl}${$_{nl}$A$_{sp}$=$_{sp}$B+123;$_{nl}$}$_{nl}$

- OUTPUT: sequence of tokens:



User identifiers like `A`, `B` and main are all represented by the same lexical token (**`IDENTtok`**), which includes a pointer to a symbol table record giving the actual name.

Introduction to Syntax Analysis (also known as "parsing")

- Programming languages have grammatical structure specified by grammatical rules in a notation such as BNF (Backus-Naur Form)
- Example:

```
stat →   'print' expression |
         'while' expression 'do' stat |
         expression '=' expression
```

- The function of syntax analysis is to extract the grammatical structure— to work out how the BNF rules must have been applied to yield the input program.
- The output of the syntax analyser is a data structure representing the program structure: the **Abstract Syntax Tree** (AST).

Returning to our C example:

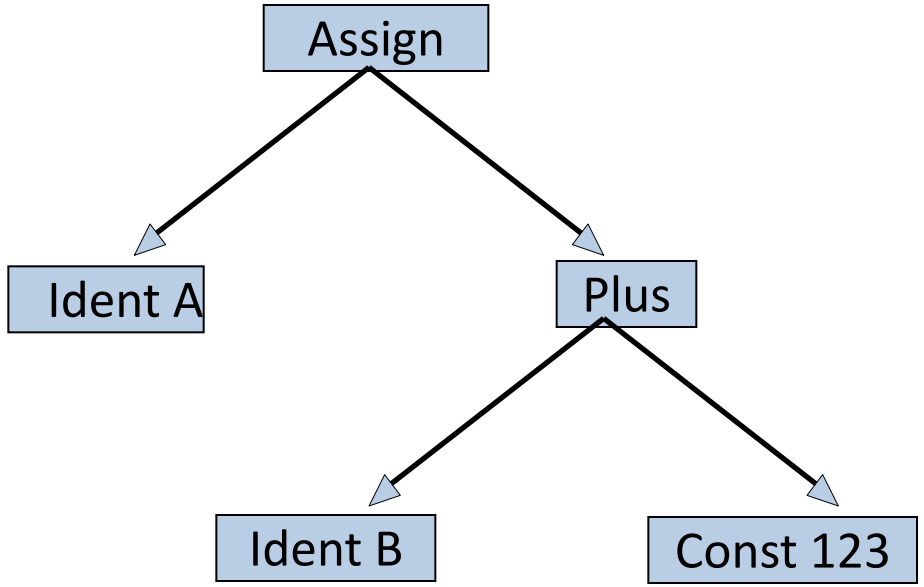
- Input characters:

  “.......A = B+123.......”

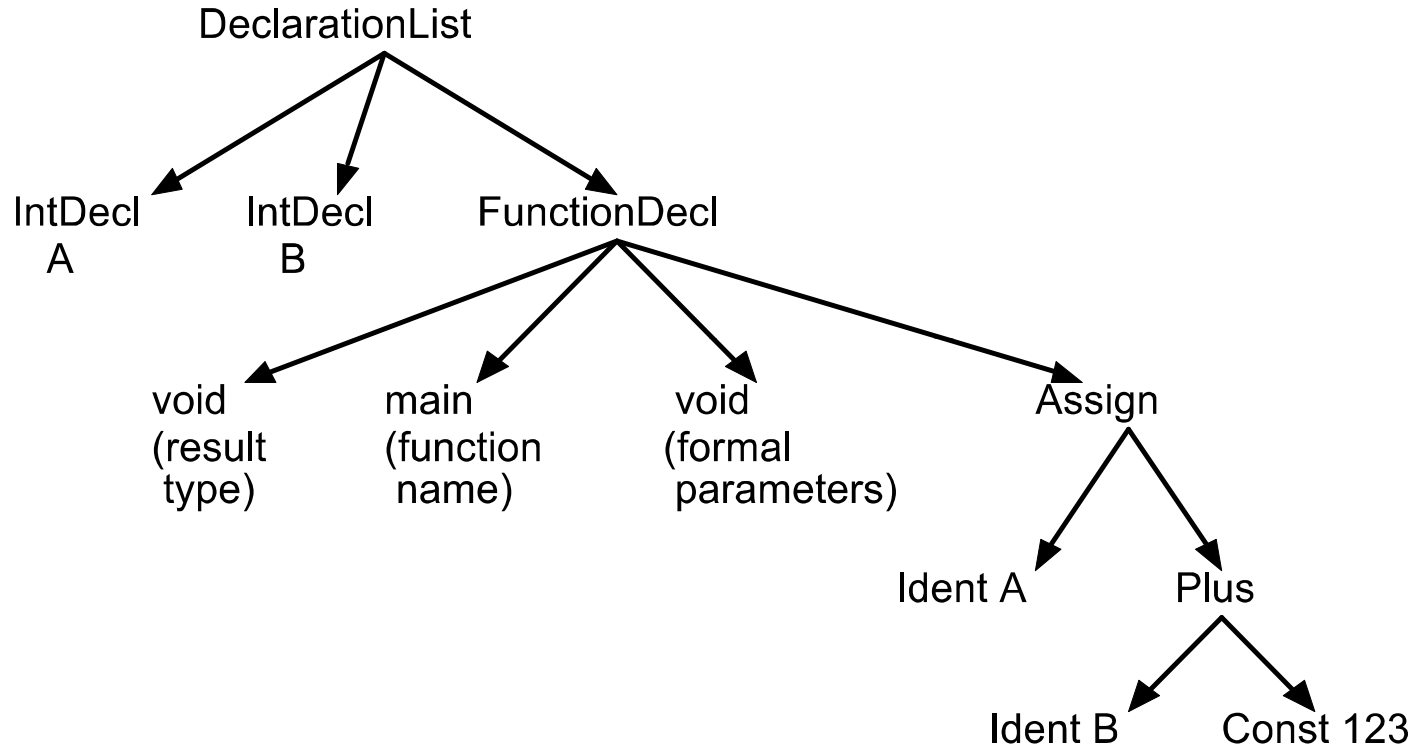
- Lexical tokens:

  [IDENTtok A, EQtok, IDENTtok B,
  PLUStok, CONSTtok 123]

- Abstract syntax tree:



AST for whole C example:



- The AST is implemented as a tree of linked objects
- The compiler writer must design the AST data structure carefully so that it is easy to build (during syntax analysis), and easy to use (e.g. during code generation).

You try: experimenting with real compilers

- Create a file "file.c" containing a simple C function
- Under Linux, type the command:

  gcc –O –S file.c

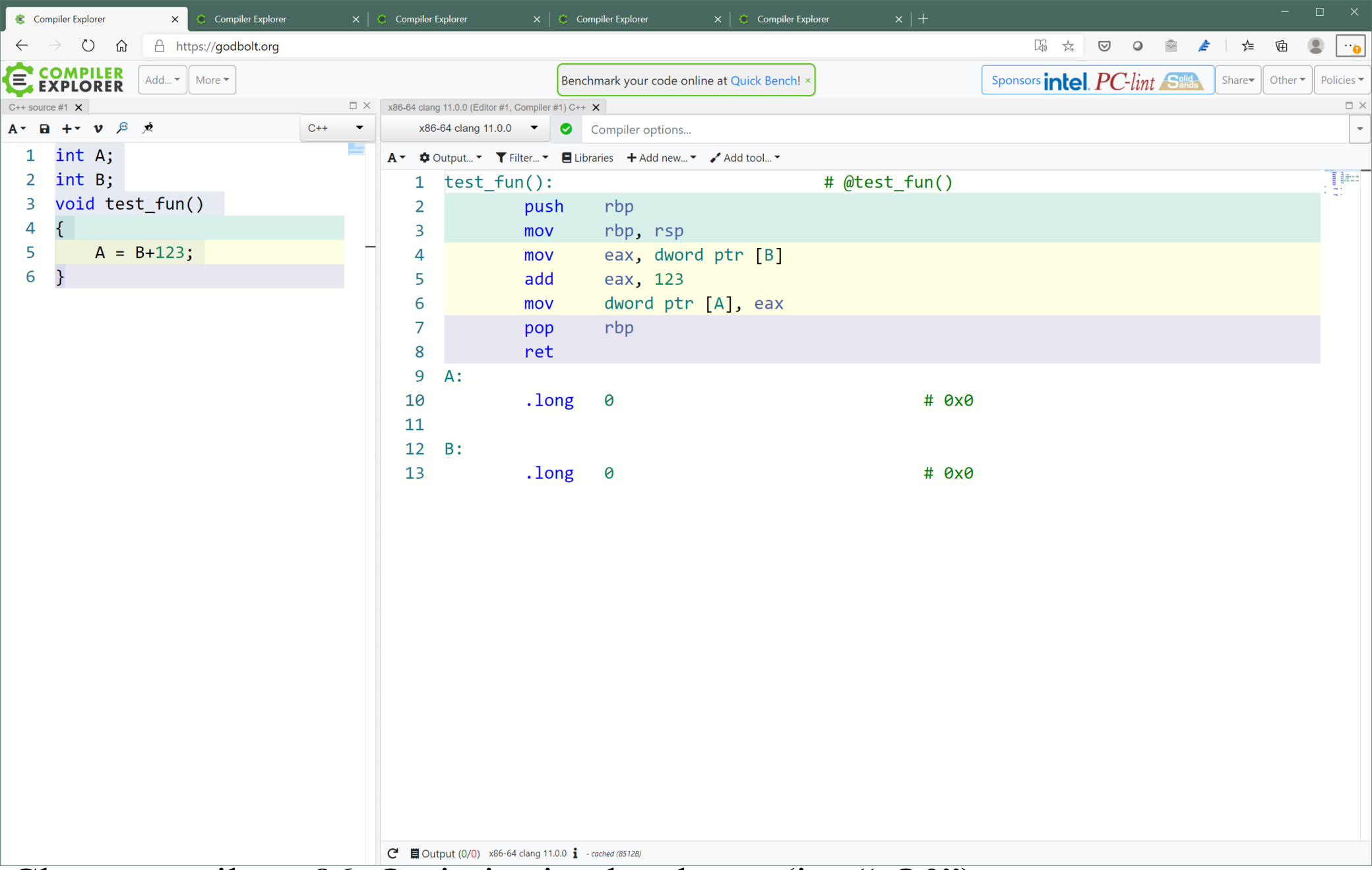
- This tells Gnu C compiler 'gcc' to produce optimised translation of the C program and leave result in "file.s"
- Examine "file.s"
- You might try

  gcc –O –S –fomit-frame-pointer file.c

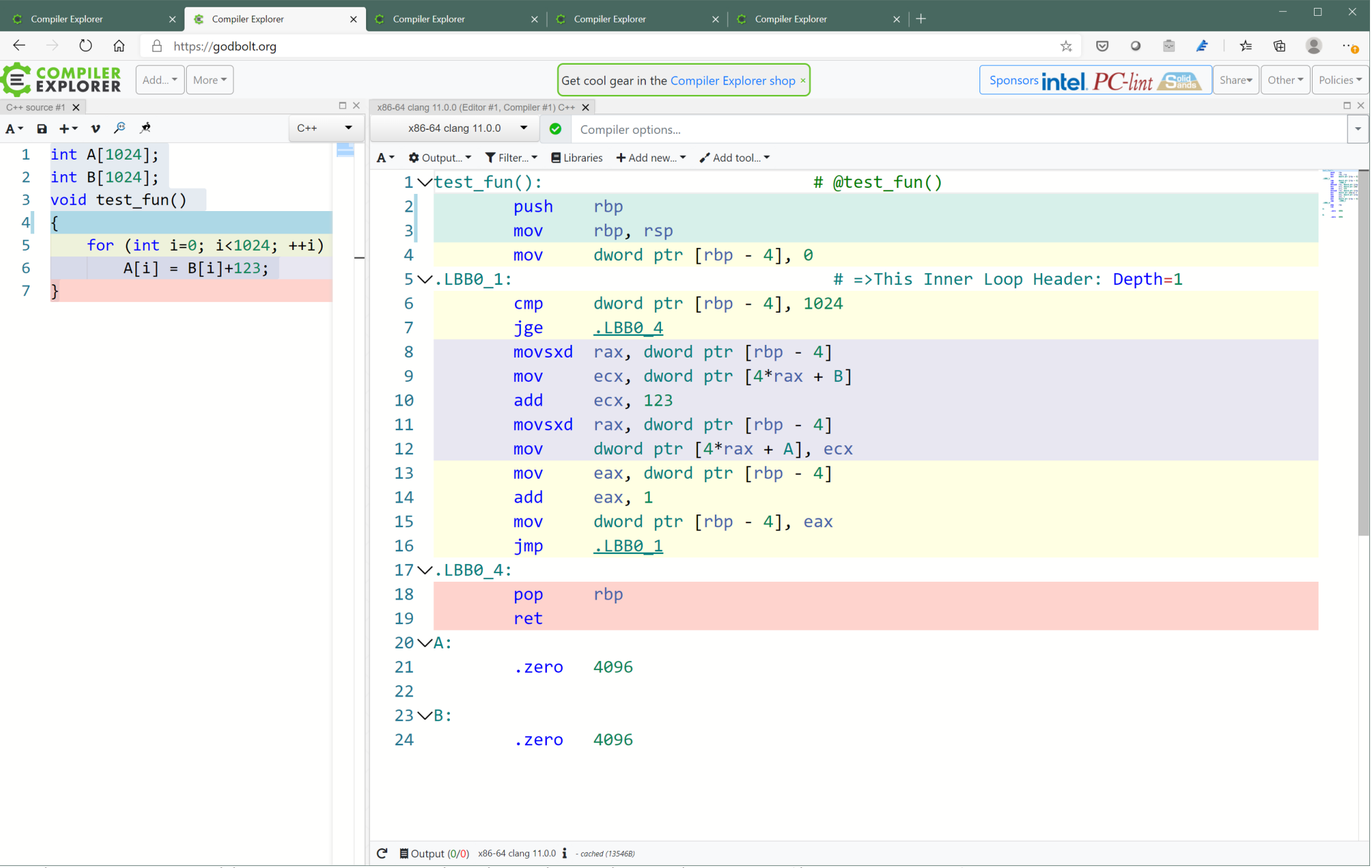
  (This simplifies the output slightly)
- On Windows using Microsoft Visual Studio, try

  cl /Fa /Ox /c test.c

  (The output is written to "test.asm")
- **Better still: http://gcc.godbolt.org/**

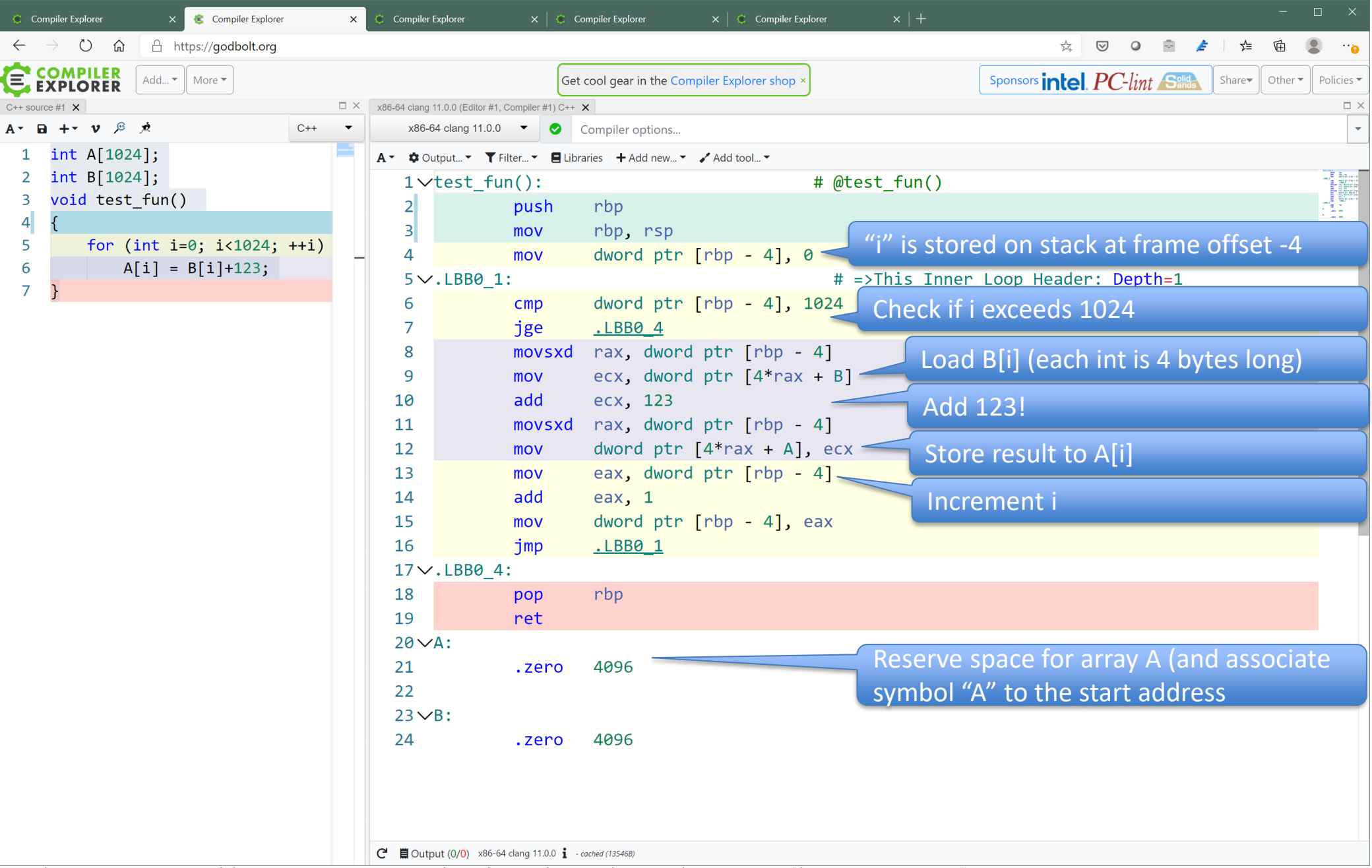


Clang compiler, x86, Optimisation level zero (i.e. “-O0”)

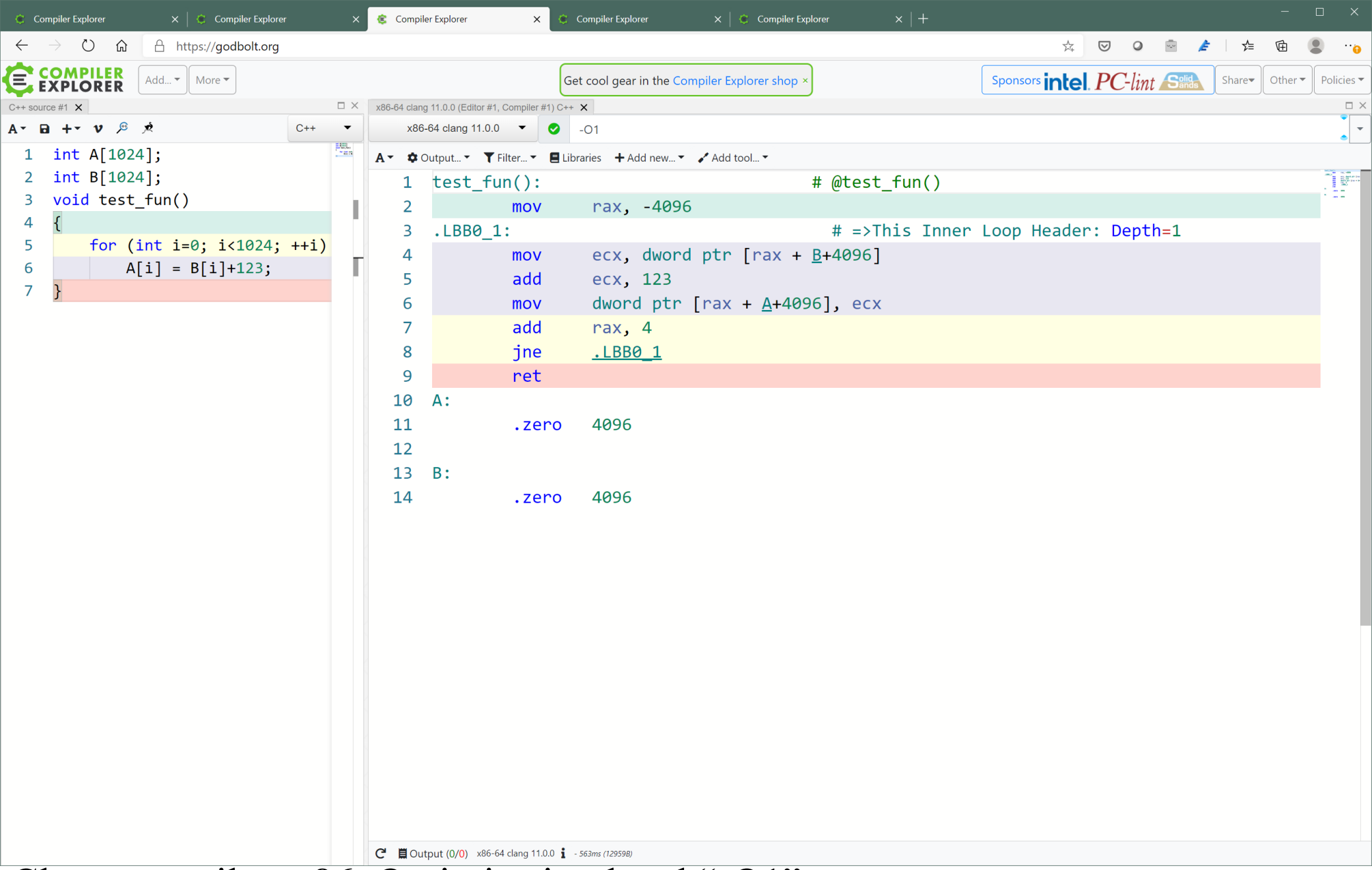


Clang compiler, x86, Optimisation level zero (i.e. “-O0”)

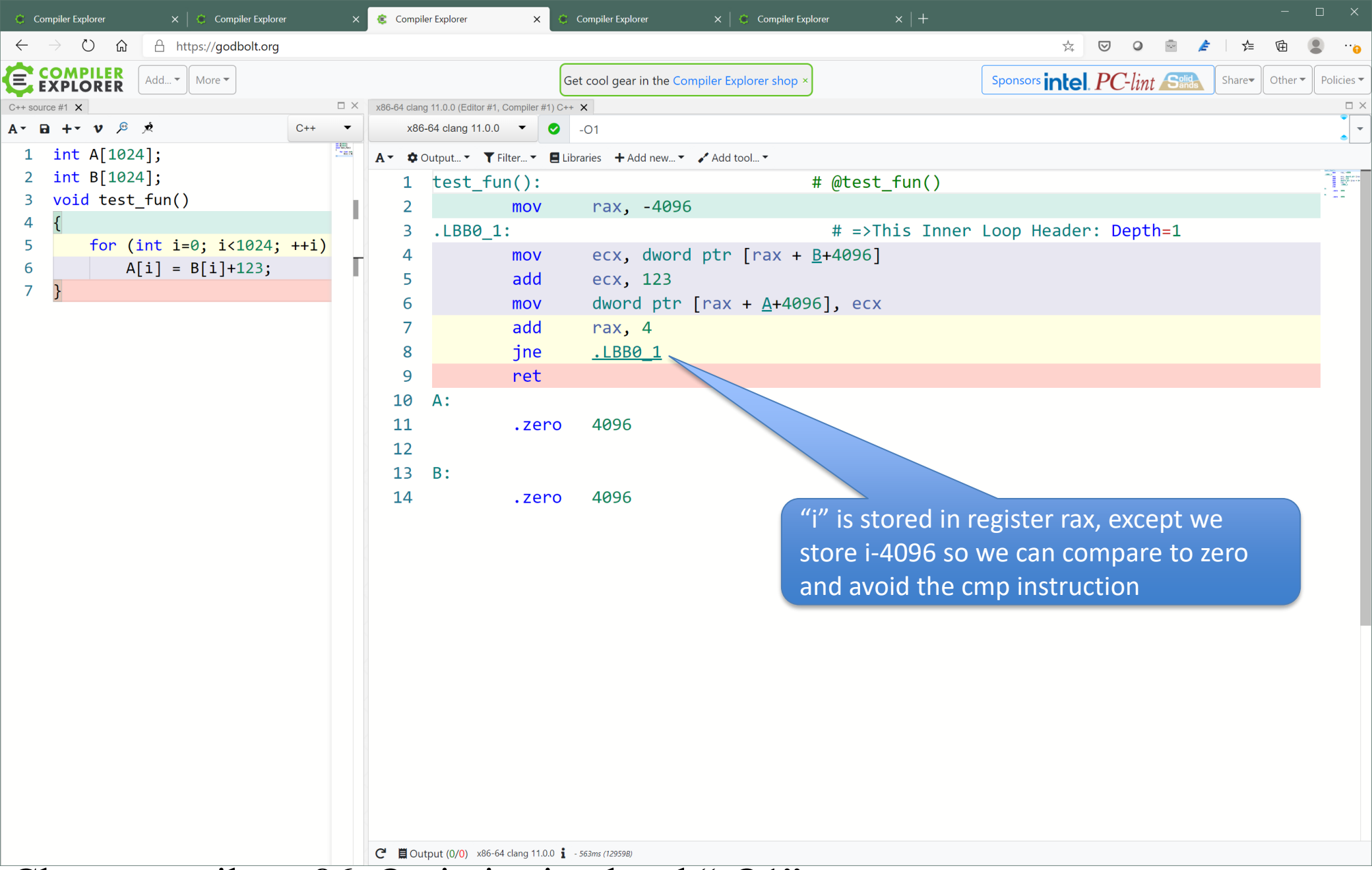
(https://godbolt.org/z/1h1drv9Y3)



Clang compiler, x86, Optimisation level zero (i.e. "-O0")

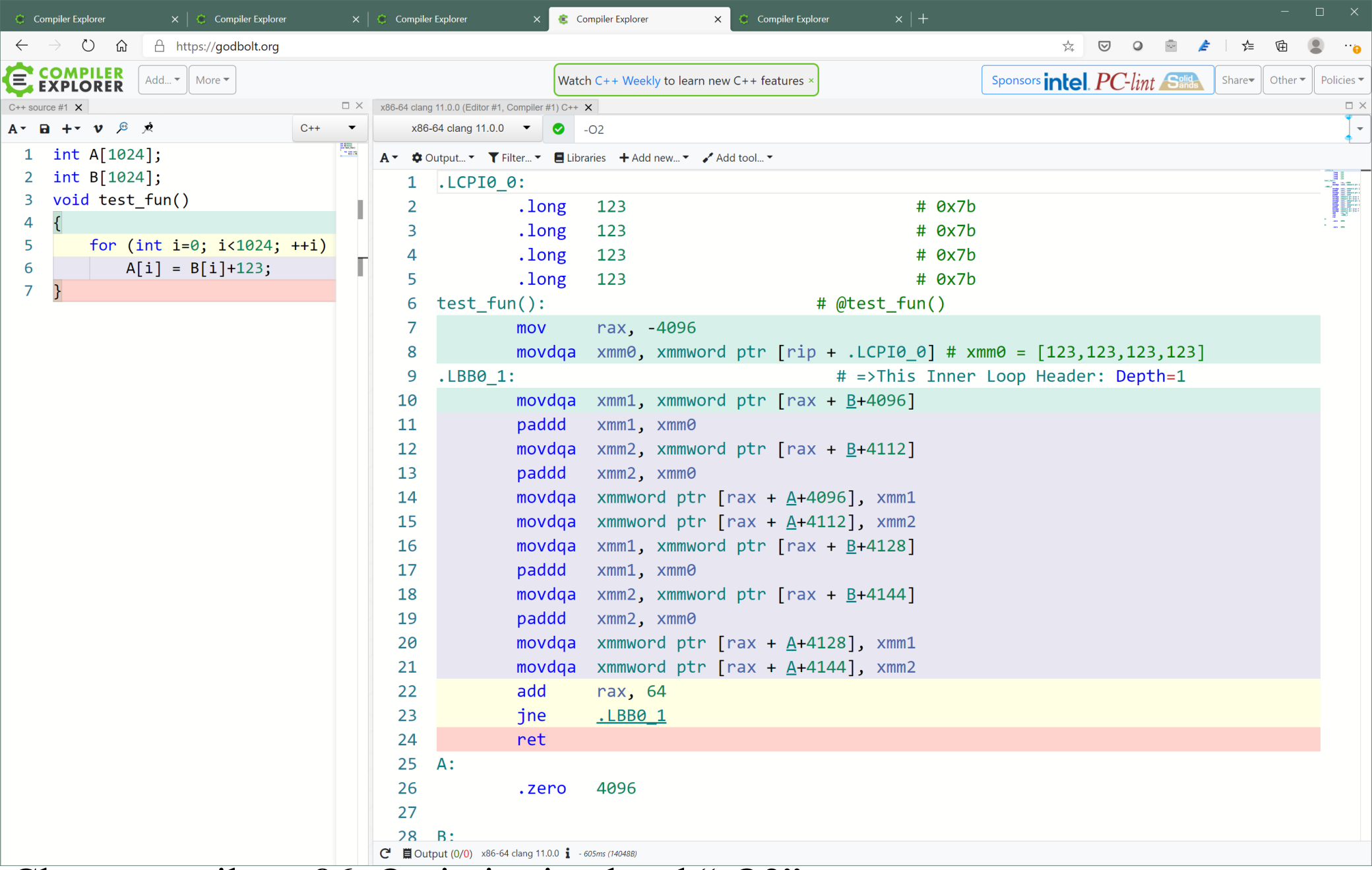


Clang compiler, x86, Optimisation level “-O1”



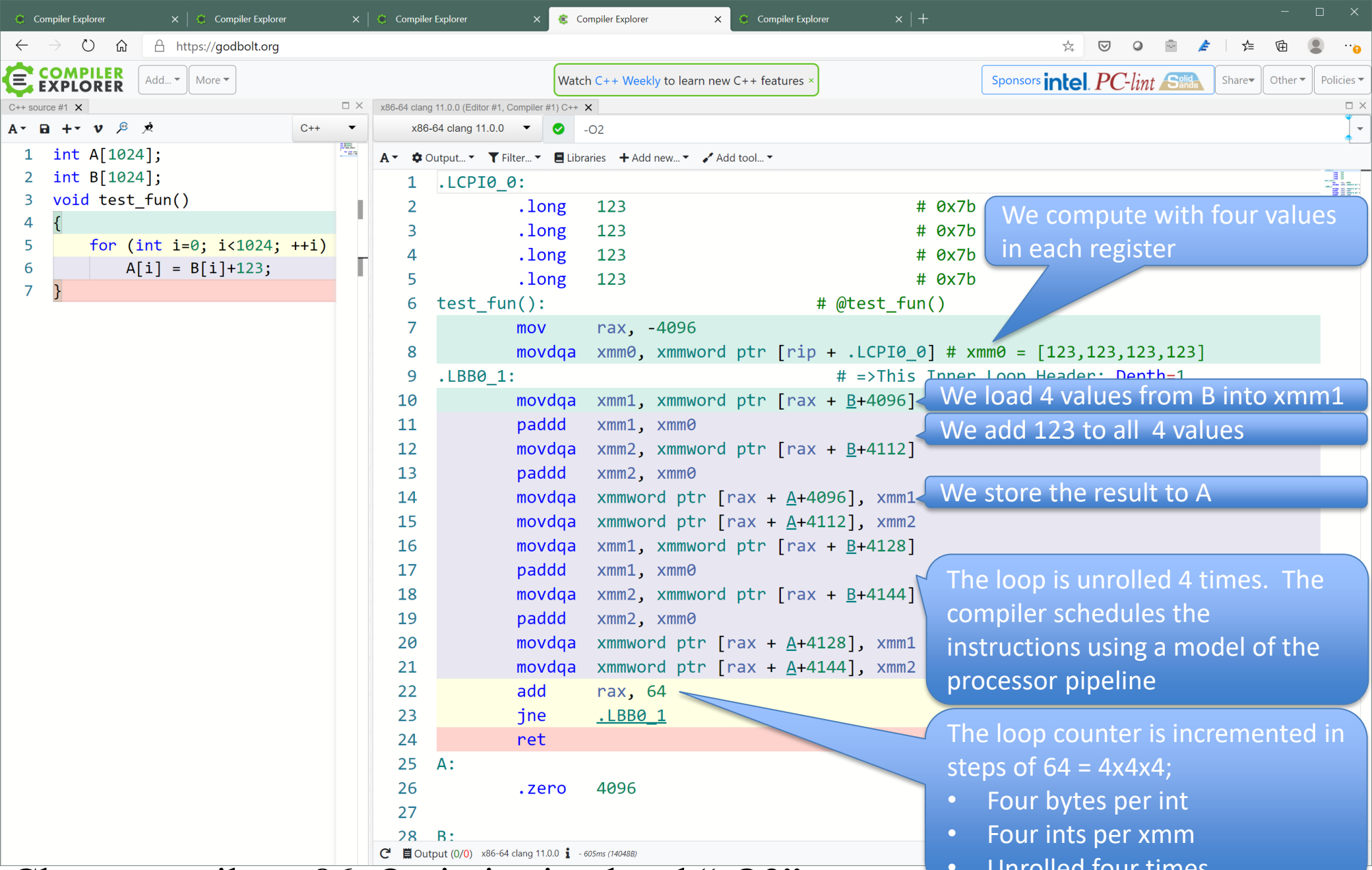
Clang compiler, x86, Optimisation level "-O1"

(https://godbolt.org/z/W5h4a9P3T)

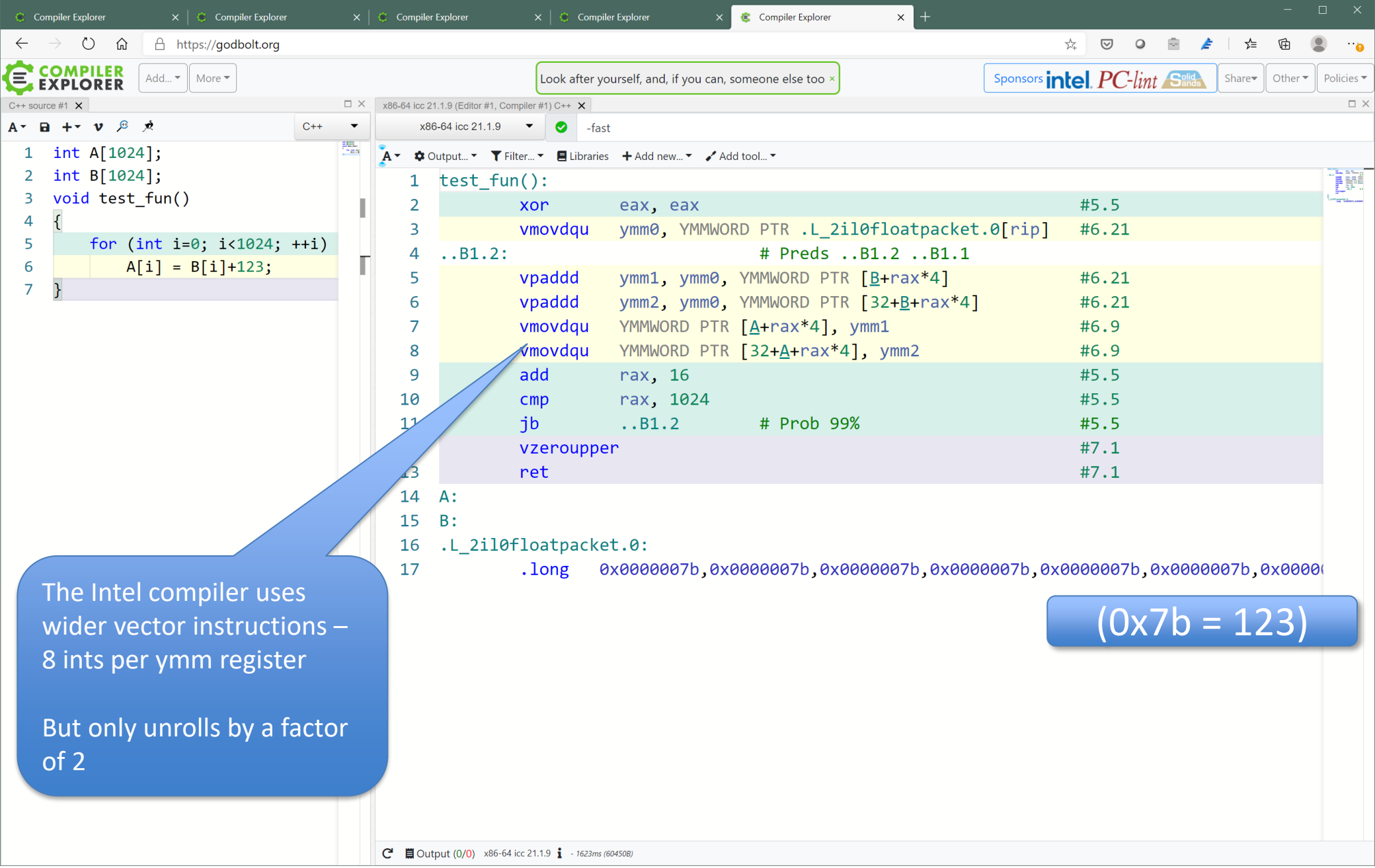


Clang compiler, x86, Optimisation level “-O2”

(https://godbolt.org/z/Y8jaGEW6b)



Clang compiler, x86, Optimisation level “-O2”



Intel compiler, x86, Optimisation level “-fast”

(https://godbolt.org/z/nxqGs7Pdf)

Compilers are just one kind of language processor:

- Really useful software tools are useful because they are programmable
- If it's programmable it must have some kind of programming language
- Programming languages are often "domain-specific" – designed for a particular application area

- Domain-specific languages – examples:
  - Tensorflow: deep learning
  - P4: network packet forwarding
  - Solidity: smart contracts
  - GLSL: shaders for 3D graphics
  - SQL: database queries
  - Verilog: digital circuit design
  - Matlab: prototyping numerical computations
  - Simulink: modelling dynamical systems
  - R: statistical data analytics
  - Prolog: logic programming
  - LaTeX: typesetting
  - The FEniCS Project's Unified Form Language: solving PDEs
  - ANTLR, Yacc: parser generation
  - TableGen: the LLVM compiler's DSL for instruction selection

- Language processors- examples:
  - FindBugs: finds Java bugs
  - PyLint: bug and quality checking for Python
  - Coverity, CodeSonar: find C++ (etc) bugs
  - BitBlaze, Coverity/B: vulnerability analysis for binaries
  - KLEE: symbolic execution engine
  - JUnit: annotation-driven unit testing
  - Mockito: mock object generation for test
  - IDEs – Intellisense in Visual Studio, etc
  - Binary-to-binary: eg x86 to ARM for Windows-on-ARM
  - Valgrind. PIN, Mambo: dynamic binary rewriting

Course structure

- Introduction to syntax analysis – and a complete toy compiler
- Code generation
- Generating better code by using registers
- Register allocation
- Optimisation and data flow analysis
- Loop optimisations

- Syntax analysis (sometimes called “parsing”)
- Semantic analysis: is the program “legal”?
- Runtime (memory) organisation: stacks and heaps

Paul

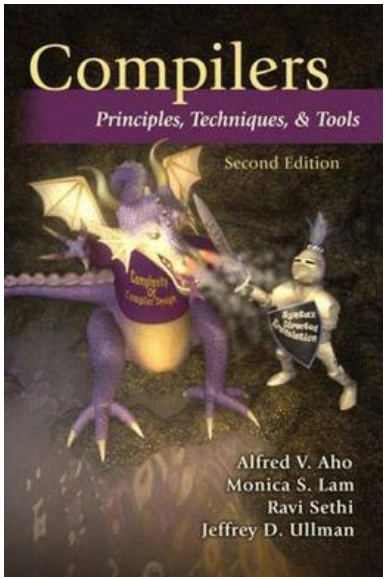
Jamie and Hongxiang

Textbook - philosophy

- There are many textbooks on compilers, some good
- The purpose of lecture course is to give you enough understanding of the area to be able to use a serious textbook effectively
- Textbook should be worth more to you *after* the course than during it!
- Choose an authoritative book that goes substantially beyond this course

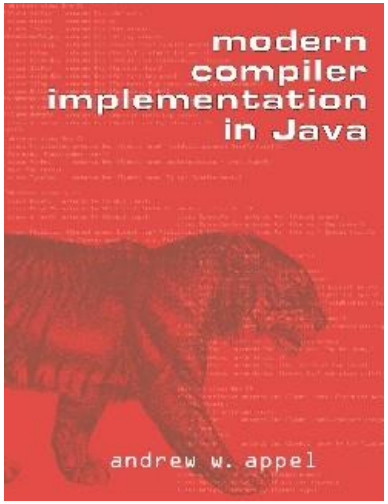
Somewhat recommended textbooks

- ***Compilers: Principles, Techniques, and Tools (second edition, 2006)*** by Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Monica Lam.
  - The new(ish) edition of the definitive book by pioneers in the subject. Often called the **Dragon book** because of the picture on the front. Compiler engineers regularly refer to "standard Dragon book stuff".



https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools

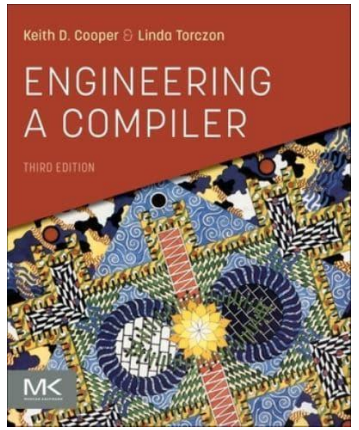
- ***Modern Compiler Implementation in Java (second edition, 2005)*** by Andrew Appel
  - Useful source for specific advice on how to build your compiler, including simple and more sophisticated techniques



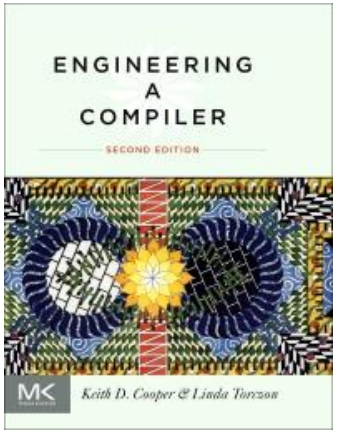
https://www.cs.princeton.edu/~appel/modern/java/

The most recommended textbook

- ***Engineering a compiler***, by Keith Cooper and Linda Torczon, Morgan Kaufmann/Elsevier (3rd edition, 2022). About £75.
  - "This book is the best compiler engineering guide ever I read." (review, Amazon.com)
  - "This book has a good introduction guiding the beginning compiler student into understanding basic concepts and gradually revealing the more intimidating stuff, but the authors took great care not to scare the beginners away and instead offers great indepth explanations into how concepts and implementation merge. Its an overall good book!" (review, Amazon.com)
  - "*First, all the algorithms are consistent with the latest research. Second, the explanations are exceptionally clear, especially compared to other recent books. Third, there's always enough extra context presented so that you understand the choices you have to make, and understand how those choices fit with the structure of your whole compiler*". (review, Jeff Kenton, comp.compilers 3/12/03)
  - "If you are a beginner "do not buy this book"" (review, Amazon.com)



https://www.elsevier.com/books/engineering-a-compiler/cooper/978-0-12-815412-0



(The 2nd edition is actually adequate for this course)

How to enjoy, learn from and pass this course:

- **Textbook** – start early, read the first couple of chapters
- Make **notes** during lectures
- **Tutorial exercises** are used to introduce new examinable material
- Tutorials are designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the **past papers - *now***
- Use the **live Q&A classes** to get feedback on your solutions
- You are **assumed to have studied the past exam papers**
- Substantial lab exercise should bring it all together
- Ask questions! Use **EdStem**!
- There are hundreds of compilers courses at universities around the world, often more advanced than this one – with slides on the web that will help with your questions

Beyond the curriculum….

- **The first compilers (for Fortran, COBOL etc) were designed to put programmers out of work**
- **But that's not what happened**

- **Why?**

- **See Jevons Paradox https://en.wikipedia.org/wiki/Jevons_paradox**

Beyond the curriculum….

- **LLMs can generate code**
- **LLM agents can automate business logic without code**

- **Will AI put programmers out of work?**

Beyond the curriculum….

- **Programs can have type errors**
- **A good type system should prevent type errors**
  - **In a language with strong, static type checking, we should have a guarantee that no type error can occur at runtime**

- **You can write a compiler in a language with strong, static type checking**
- **That is, you can write a program that generates code**

- **Can we guarantee that the code we generate contains no type errors?**

- **See MetaOCaml https://okmij.org/ftp/ML/MetaOCaml.html**