

Compilers 101

Intro

Course overview

1. Intro (this lecture)
2. Compiler frontend
3. Static analysis
4. Intermediate representations
5. Optimizations
6. Backend
7. Debuggers
8. Heterogeneous compilers

Contacts

Lectures: Оболенский Арсений

- E-mail: me@gooddoog.ru
- Via portal.unn.ru

Practice: Лычков Михаил

- E-mail: mikelitch@gmail.com

Practice: Кузнецов Артём

- E-mail: artyomka6192@gmail.com

GitHub organization: <https://github.com/NN-complr-tech>

Lecture materials repository: <https://github.com/NN-complr-tech/Complr-course-lectures>

What is a compiler?



First compilers

A-0, 1952

- Grace Hopper at Remington Rand
- Not a real compiler, functioned more as a loader and a linker
- Photo: https://en.wikipedia.org/wiki/Grace_Hopper



Fortran

1954-1957

- John Backus at IBM
- Fortran is still used and actively developed by ISO. The last release is Fortran 2018.
- Photo: https://en.wikipedia.org/wiki/John_Backus



Lisp

1958

- John McCarthy at MIT
- Second-oldest high-level programming language
- Photo: [https://en.wikipedia.org/wiki/John_McCarthy_\(computer_scientist\)](https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist))



C

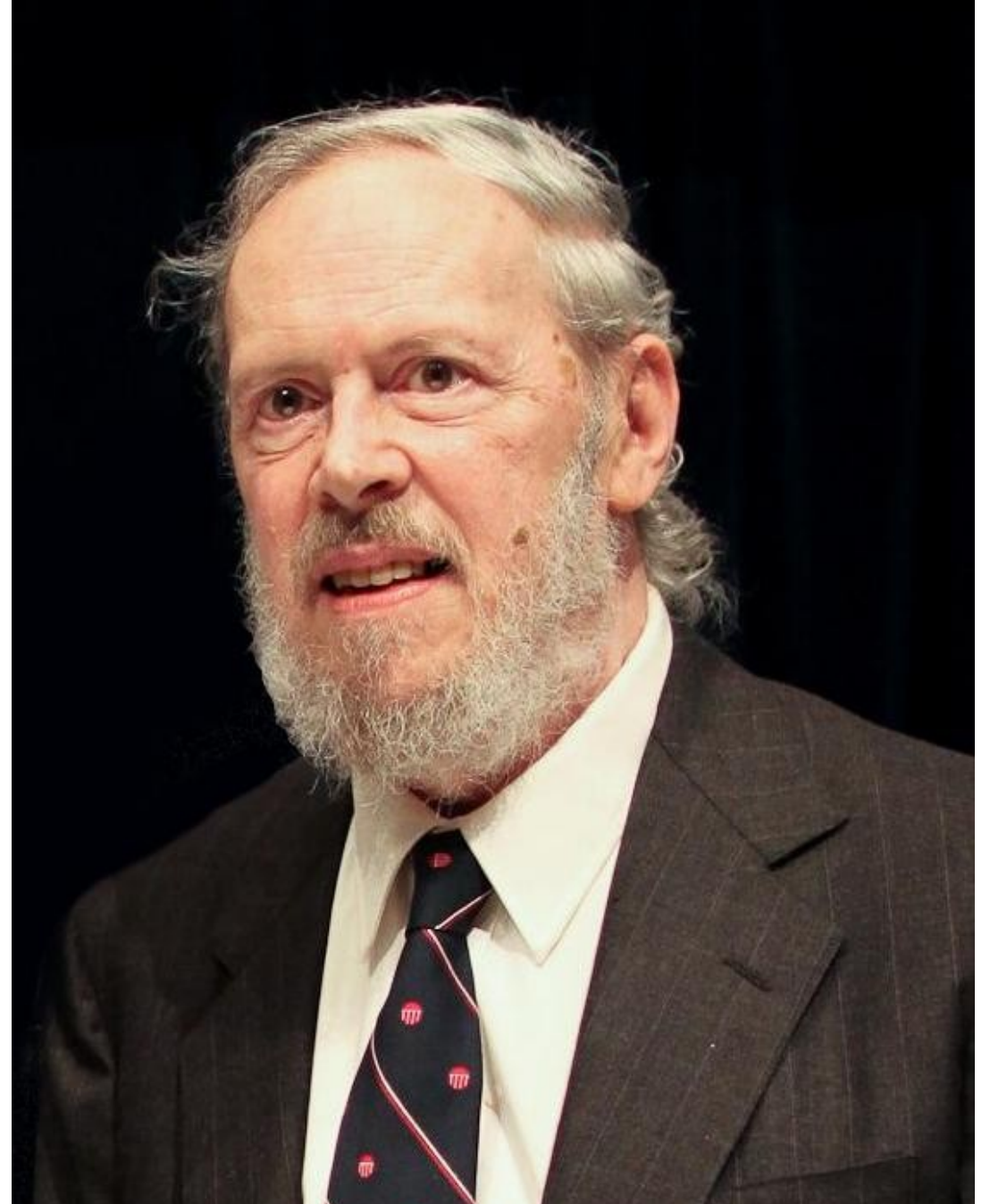
1972

Dennis Ritchie at Bell Labs

Developed the first C compiler (derived from the B language)

Became the backbone of the Unix operating system

- Photo: https://en.wikipedia.org/wiki/Dennis_Ritchie



C++

1979

Created by Bjarne Stroustrup at Bell Labs

Evolved from C to add object-oriented programming features

First commercial implementation released in 1985

- Photo: <https://www.stroustrup.com/>



Beyond software

Hardware description languages

- Verilog – 1984
- VHDL – 1987

Heterogenous calculation languages

- OpenCL – 2009
- SYCL – 2014
- Photo:
<https://www.flickr.com/photos/intelfreepress/8267616249/in/photostream/lightbox/>



OpenCL

- Purpose: Designed for creating programs that leverage the parallel computing capabilities of heterogeneous platforms.
- Cross-Platform: Supports a wide range of devices from various manufacturers, including GPUs, CPUs, DSPs, and FPGAs.
- Parallel Programming Model: Enables efficient execution of high-performance computing, graphical, and real-time applications.

```
__kernel void vector_add(__global const float* A, __global const float* B, __global float* C) {  
    ... // Get the index of the current element  
    ... int i = get_global_id(0);  
  
    ... // Perform the addition  
    ... C[i] = A[i] + B[i];  
}
```

SYCL

SYCL (pronounced "sickle") is a high-level programming model built on top of OpenCL that enables developers to write code for heterogeneous devices (such as CPUs, GPUs, DSPs, and FPGAs) using standard C++.

It abstracts away much of the boilerplate code associated with OpenCL, allowing for easier development of parallel computing applications.

```
#include <CL/sycl.hpp>

int main() {
    ... using namespace cl::sycl;
    ... queue q;

    ... // Define the size of the vectors
    ... const int size = 1024;
    ... // Create two input vectors and one output vector
    ... std::vector<float> A(size, 1.0f), B(size, 2.0f), C(size);

    ... // Create buffers from the vectors
    ... buffer<float> bufA(A.data(), range<1>(size));
    ... buffer<float> bufB(B.data(), range<1>(size));
    ... buffer<float> bufC(C.data(), range<1>(size));

    ... // Submit a command group to the queue to perform the computation
    ... q.submit([&](handler& h) {
        ... // Create accessors for the buffers
        ... auto a = bufA.get_access<access::mode::read>(h);
        ... auto b = bufB.get_access<access::mode::read>(h);
        ... auto c = bufC.get_access<access::mode::write>(h);

        ... // Define the kernel
        ... h.parallel_for(range<1>(size), [=](id<1> idx) {
            ... c[idx] = a[idx] + b[idx];
            ... });
        ... });

    ... // The destructor of the buffer c will copy the data back to C
    ... return 0;
}
```

Why study compilers?

- Write better software
 - Compiler is a tool, mastering this tool will help you write fast and reliable software
- Compilers are not only about code generation
 - Static analysis, code-to-code conversion
 - Example: Intel® DPC++ Compatibility Tool (aka SYCLomatic),
<https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html#gs.p25a6d>
- Compiler writers get paid!

Course emphasis

- Practice oriented
 - Study modern compiler tooling and infrastructure
 - Lab assignments
- Provide overview of basic algorithms and theory
- Non-goal: how to build a complete compiler optimizer
 - There are lots of links on the slides, make sure to check them out if you want to know more

The rest of this lecture

- Hardware architecture refresh
 - CPUs, GPUs, FPGAs
 - Just enough to understand the problem
- Quick compiler architecture introduction
- Overview of practical applications

CPUs

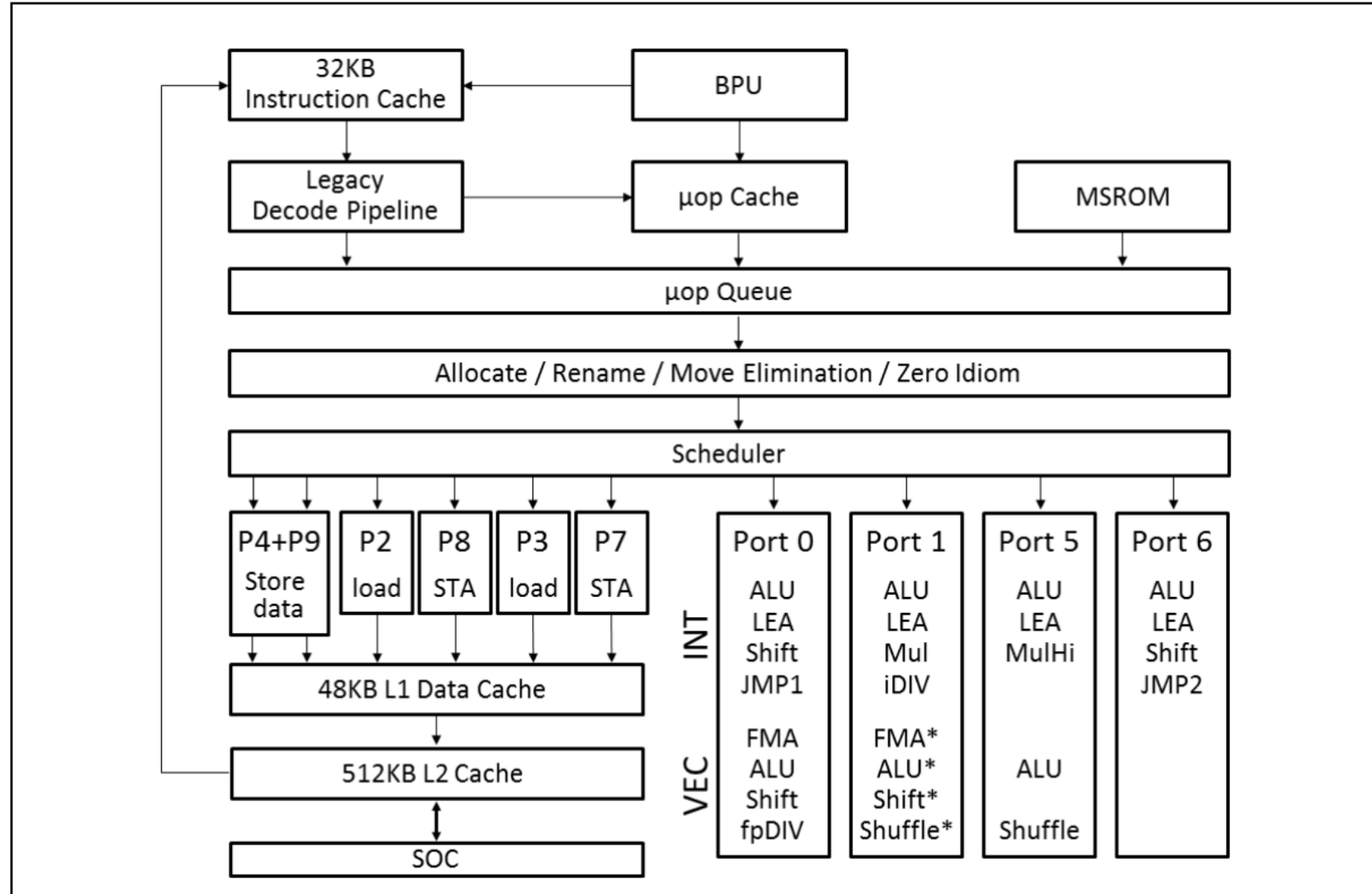


Figure 2-1. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture¹

<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>

CPU pipeline

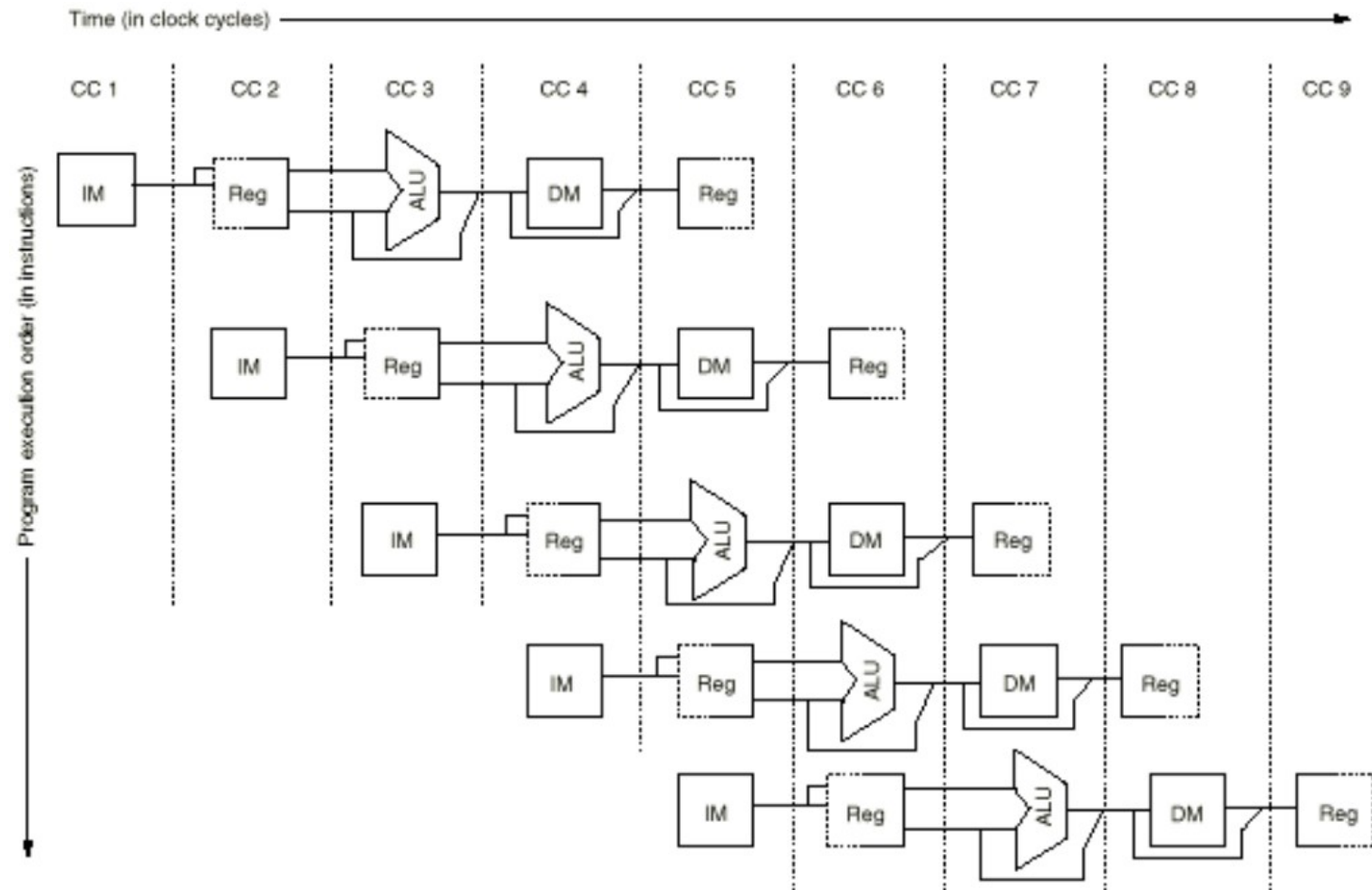
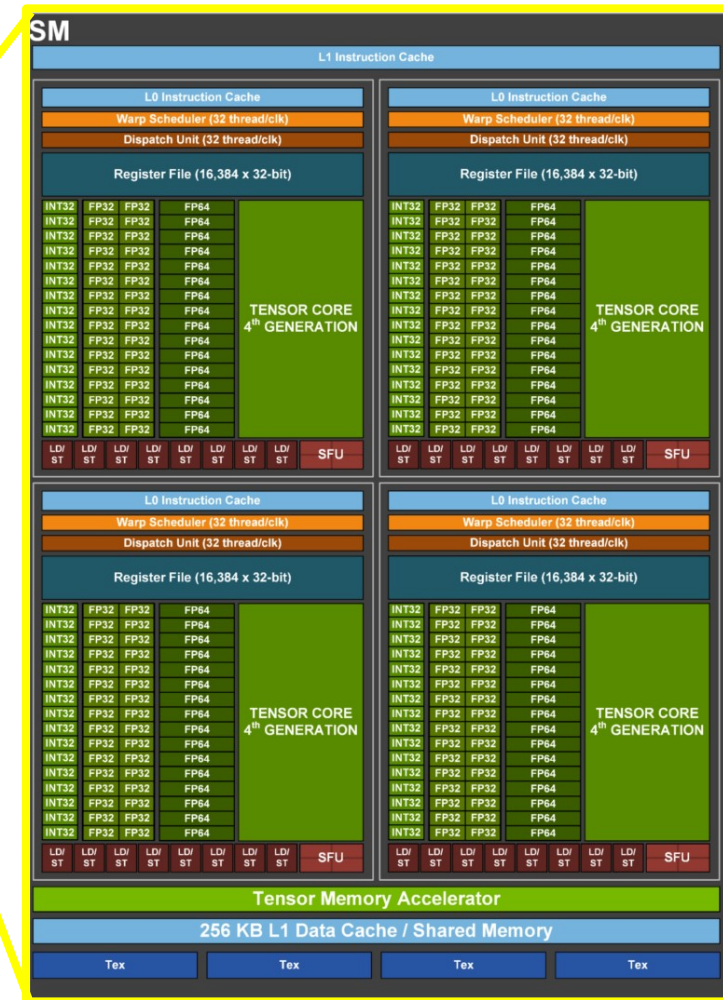
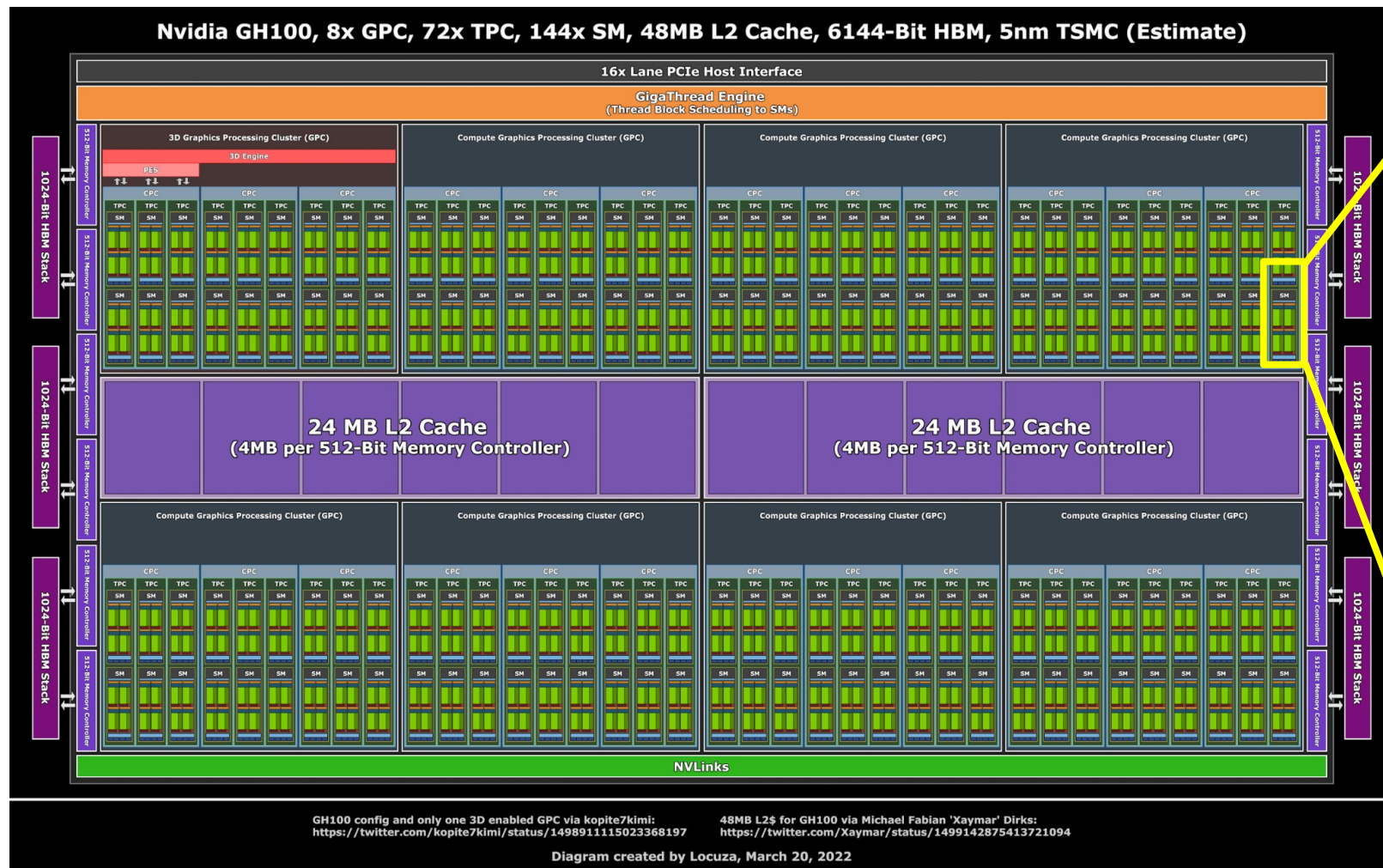


FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.

CISC vs RISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

GPUs

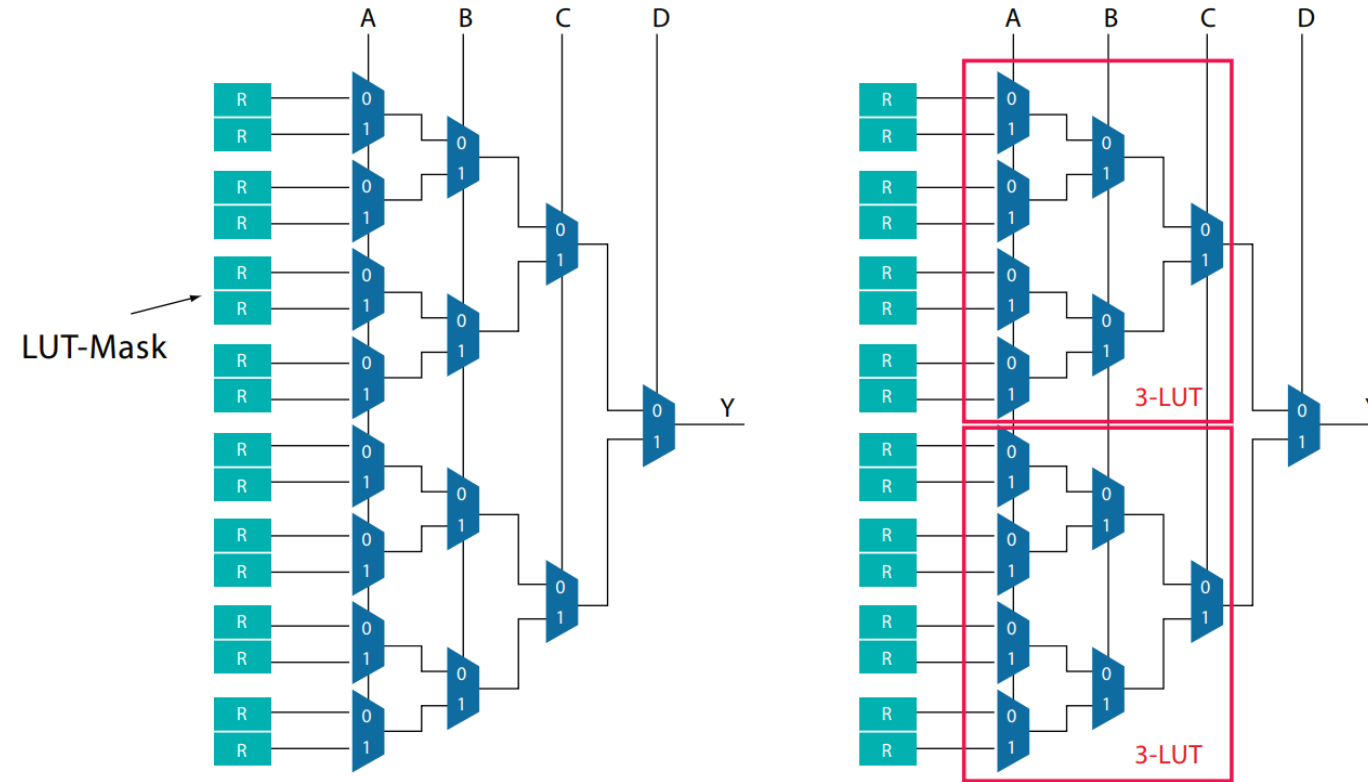


<https://videocardz.com/newz/nvidia-gh100-hopper-gpu-comes-with-48mb-of-l2-cache-only-one-gpc-has-graphics-enabled>

<https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>

FPGAs

Figure 2. Building a LUT



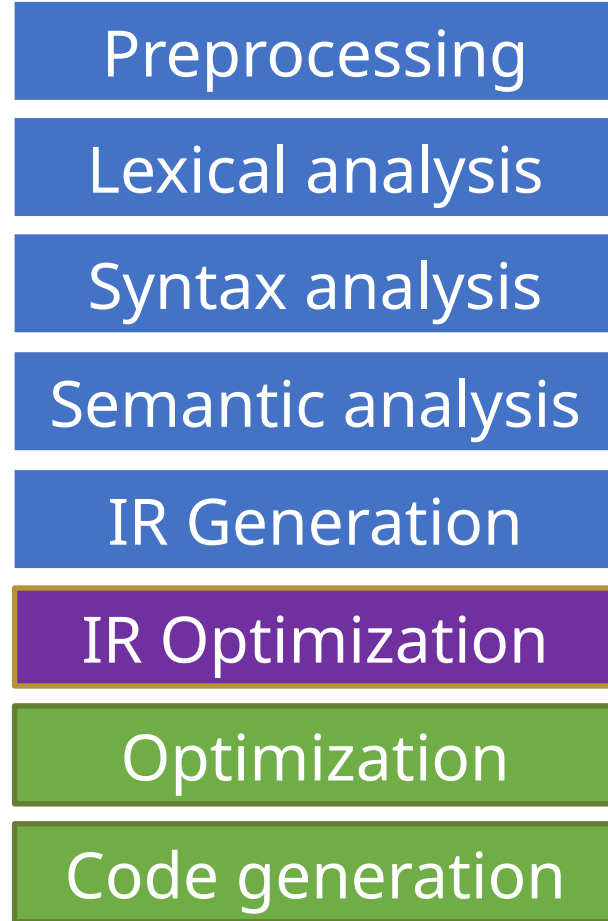
$$a'b'c'd' + abcd + abc'd' = 1000\ 0000\ 0000\ 1001 = 0x8009$$

<https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf>

Modern compilers



Compiler stages

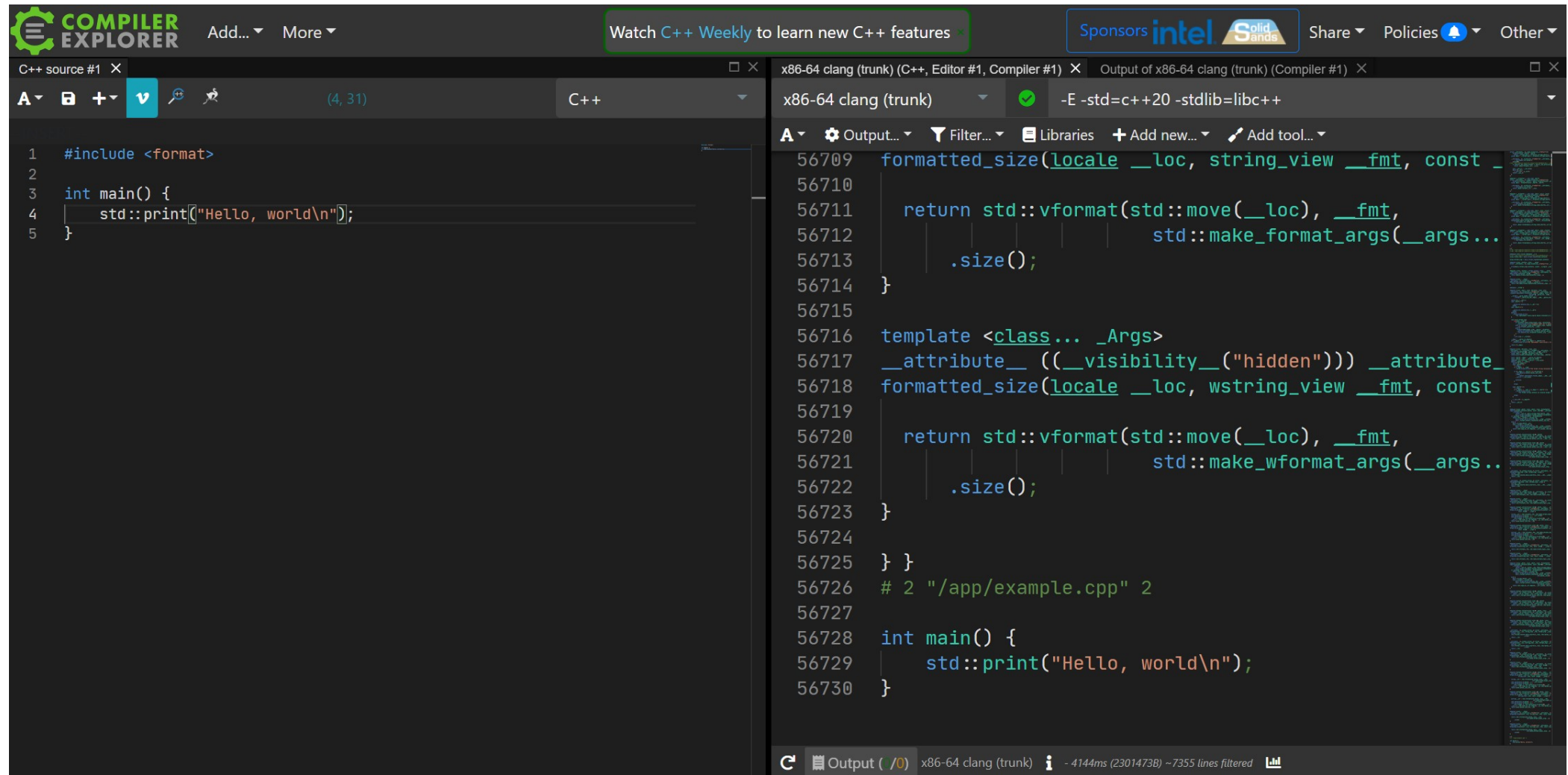


Frontend

Middle-end

Backend

Preprocessing



The screenshot displays the Compiler Explorer interface. The left pane shows the source code for a C++ program. The right pane shows the preprocessed output, which includes the original source code and the expanded content of the `<format>` header.

Source Code (Left Pane):

```
1 #include <format>
2
3 int main() {
4     std::print("Hello, world\n");
5 }
```

Compiler Options (Top Bar): x86-64 clang (trunk) (C++, Editor #1, Compiler #1) | Output of x86-64 clang (trunk) (Compiler #1) | -E -std=c++20 -stdlib=libc++

Preprocessed Output (Right Pane):

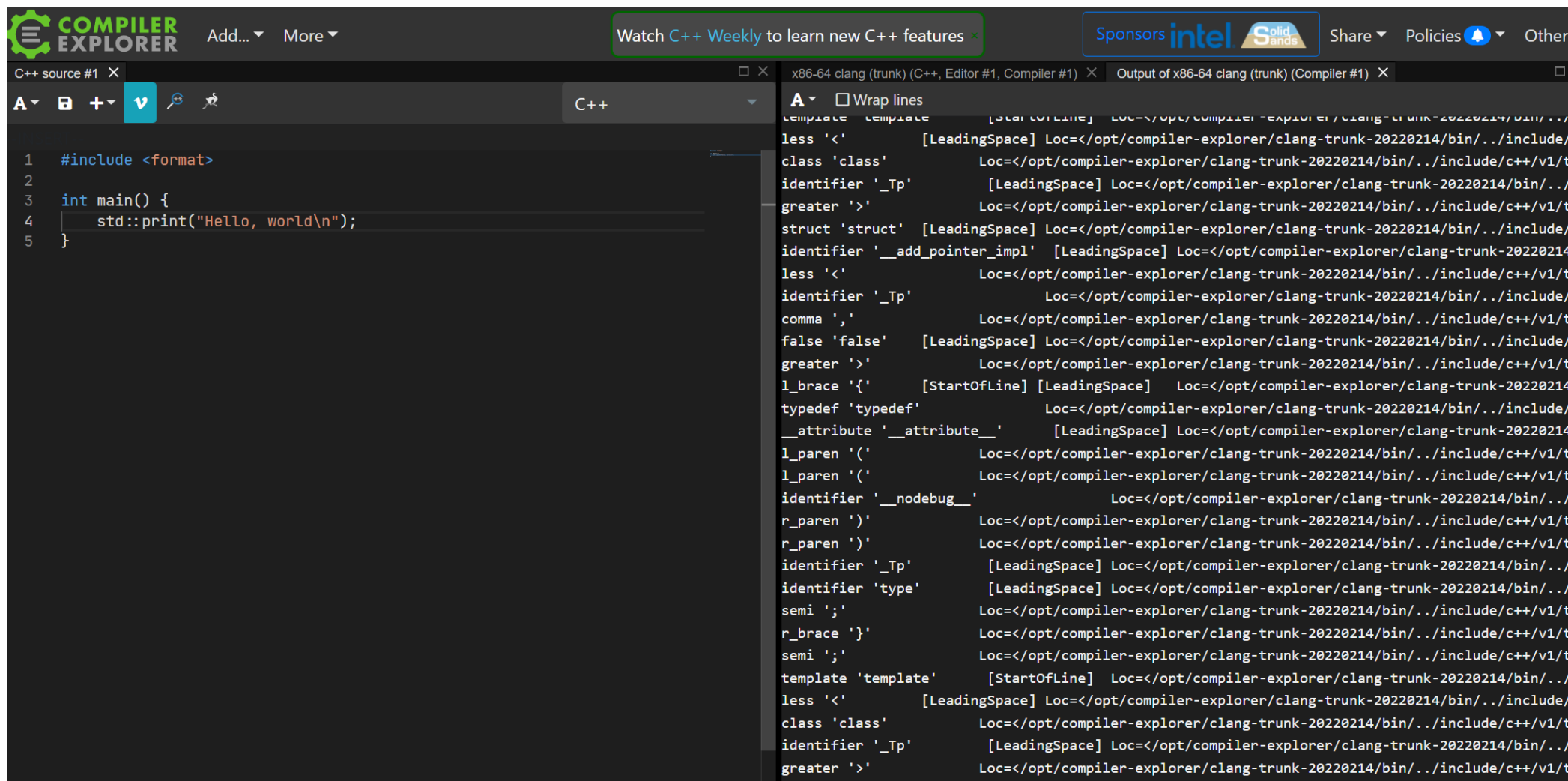
```
56709 formatted_size(locale __loc, string_view __fmt, const _
56710
56711     return std::vformat(std::move(__loc), __fmt,
56712                        std::make_format_args(__args...
56713                        .size());
56714 }
56715
56716 template <class... _Args>
56717 __attribute__((__visibility__("hidden"))) __attribute__
56718 formatted_size(locale __loc, wstring_view __fmt, const
56719
56720     return std::vformat(std::move(__loc), __fmt,
56721                        std::make_wformat_args(__args..
56722                        .size());
56723 }
56724 } }
56725 # 2 "/app/example.cpp" 2
56726
56727 int main() {
56728     std::print("Hello, world\n");
56729 }
56730 }
```

Output Summary (Bottom Bar): Output (0) | x86-64 clang (trunk) | - 4144ms (2301473B) ~7355 lines filtered



<https://godbolt.org/z/o4K7TYr5E>

Lexer



The screenshot displays the Compiler Explorer web application. The top navigation bar includes the 'COMPILER EXPLORER' logo, links to 'Add...' and 'More...', a promotional banner for 'Watch C++ Weekly', and a 'Sponsors' section featuring Intel and Solid Sands. The main interface is divided into two panes. The left pane, titled 'C++ source #1', contains a C++ program:

```
1 #include <format>
2
3 int main() {
4     std::print("Hello, world\n");
5 }
```

The right pane, titled 'x86_64 clang (trunk) (C++, Editor #1, Compiler #1)', shows the output of the compiler's lexer, which tokenizes the source code into a list of tokens and their locations. The output is as follows:

```
template 'template' [StartOfLine] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../
less '<' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c
class 'class' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
identifier '_Tp' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../i
greater '>' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
struct 'struct' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c
identifier '__add_pointer_impl' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/
less '<' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
identifier '_Tp' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c
comma ',' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
false 'false' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c
greater '>' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
l_brace '{' [StartOfLine] [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/
typedef 'typedef' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c
__attribute '__attribute__' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/
l_paren '(' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
l_paren '(' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
identifier '_nodebug_' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../i
r_paren ')' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
r_paren ')' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
identifier '_Tp' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../i
identifier 'type' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../i
semi ';' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
r_brace '}' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
semi ';' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
template 'template' [StartOfLine] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../i
less '<' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c
class 'class' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
identifier '_Tp' [LeadingSpace] Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../i
greater '>' Loc=</opt/compiler-explorer/clang-trunk-20220214/bin/.../include/c++/v1/ty
```



<https://godbolt.org/z/T6ejdsEba>

Syntax analysis

The screenshot displays the Visual Studio Code interface with two panels. The left panel shows the C++ source code for a simple program that prints "Hello, world\n". The right panel shows the Abstract Syntax Tree (AST) for the same code, generated by the x86-64 clang compiler. The AST is a tree structure representing the syntactic structure of the code, with nodes for function declarations, compound statements, expressions, and literals.

```
1  #include <format>
2
3  int main() {
4      std::format("Hello, world\n");
5  }
```

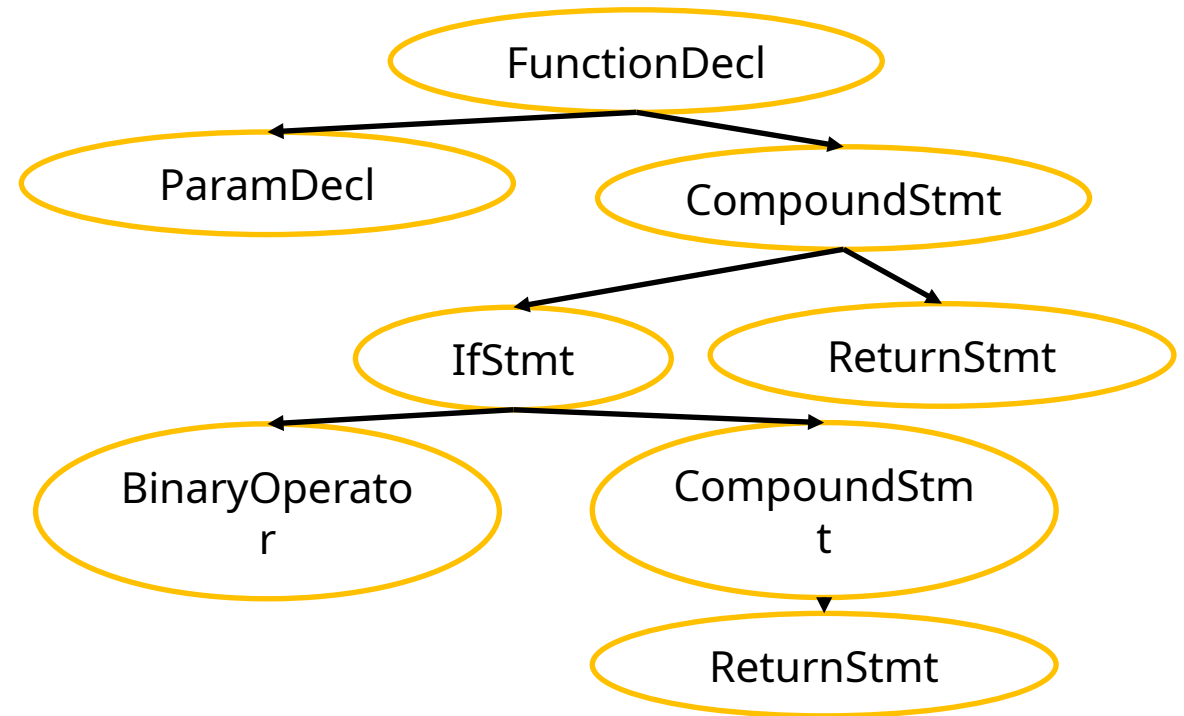
```
1  TranslationUnitDecl
2  |-FunctionDecl <line:3:1, line:5:1> line:3:5 main 'int ()'
3  |-CompoundStmt <col:12, line:5:1>
4  |   |-ExprWithCleanups <line:4:5, col:33> 'std::string': 'std::string'
5  |   |   |-CXXBindTemporaryExpr <col:5, col:33> 'std::string': 'std::string'
6  |   |   |   |-CallExpr <col:5, col:33> 'std::string': 'std::string'
7  |   |   |   |   |-ImplicitCastExpr <col:5, col:10> 'std::string (*) (std::string
8  |   |   |   |   |   |-DeclRefExpr <col:5, col:10> 'std::string (std::string_view)
9  |   |   |   |   |   |   |-ImplicitCastExpr <col:17> 'std::string_view': 'std::string_vie
10 |   |   |   |   |   |   |   |-CXXConstructExpr <col:17> 'std::string_view': 'std::string_v
11 |   |   |   |   |   |   |   |   |-ImplicitCastExpr <col:17> 'const char *' <ArrayToPointerD
12 |   |   |   |   |   |   |   |   |-StringLiteral <col:17> 'const char[14]' lvalue "Hello,
```



<https://godbolt.org/z/cz3KjMcYb>

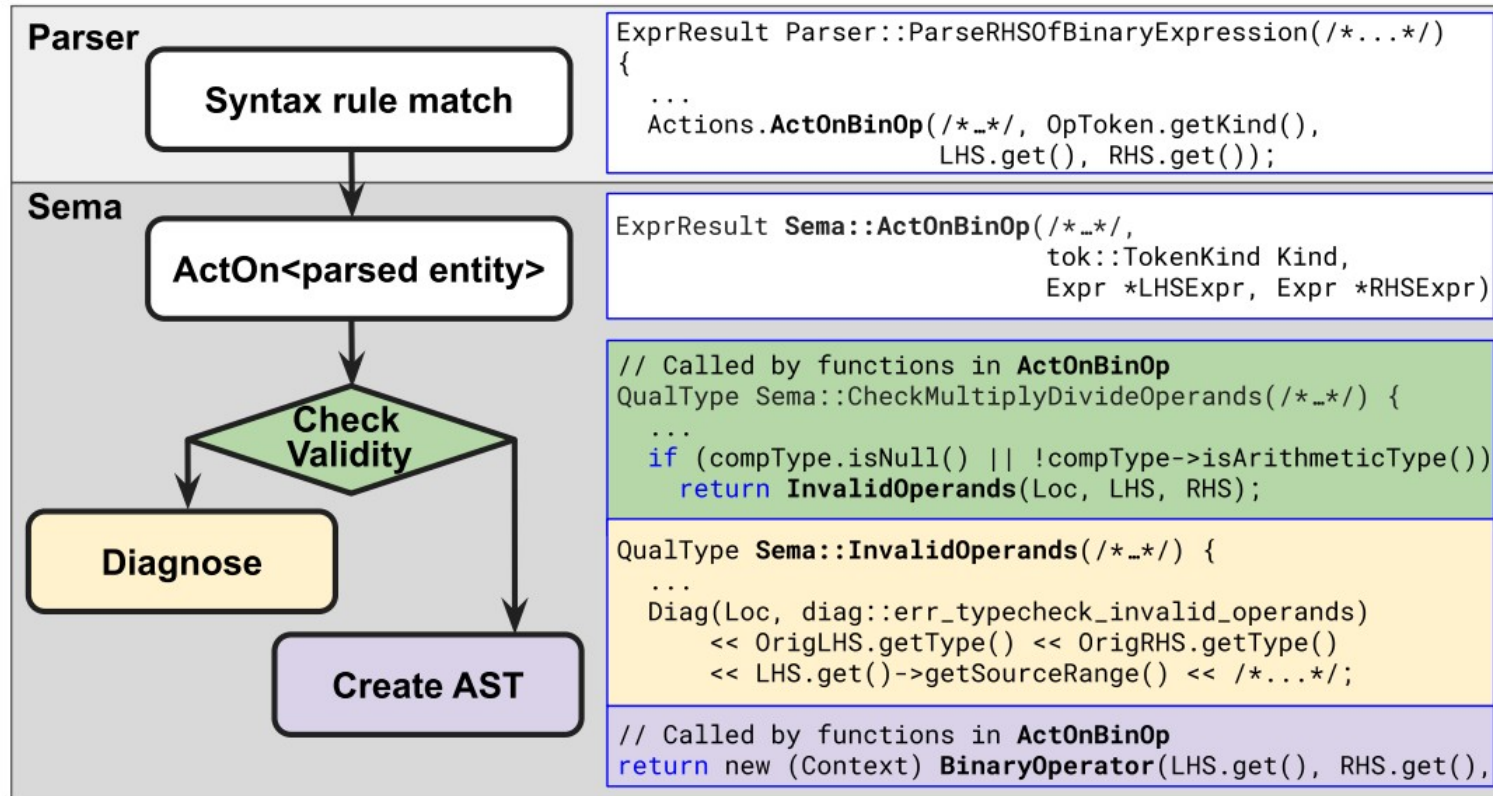
Abstract Syntax Tree

```
int foo(int i) {  
    if (i < 10) {  
        return i * 2;  
    }  
    return i;  
}
```

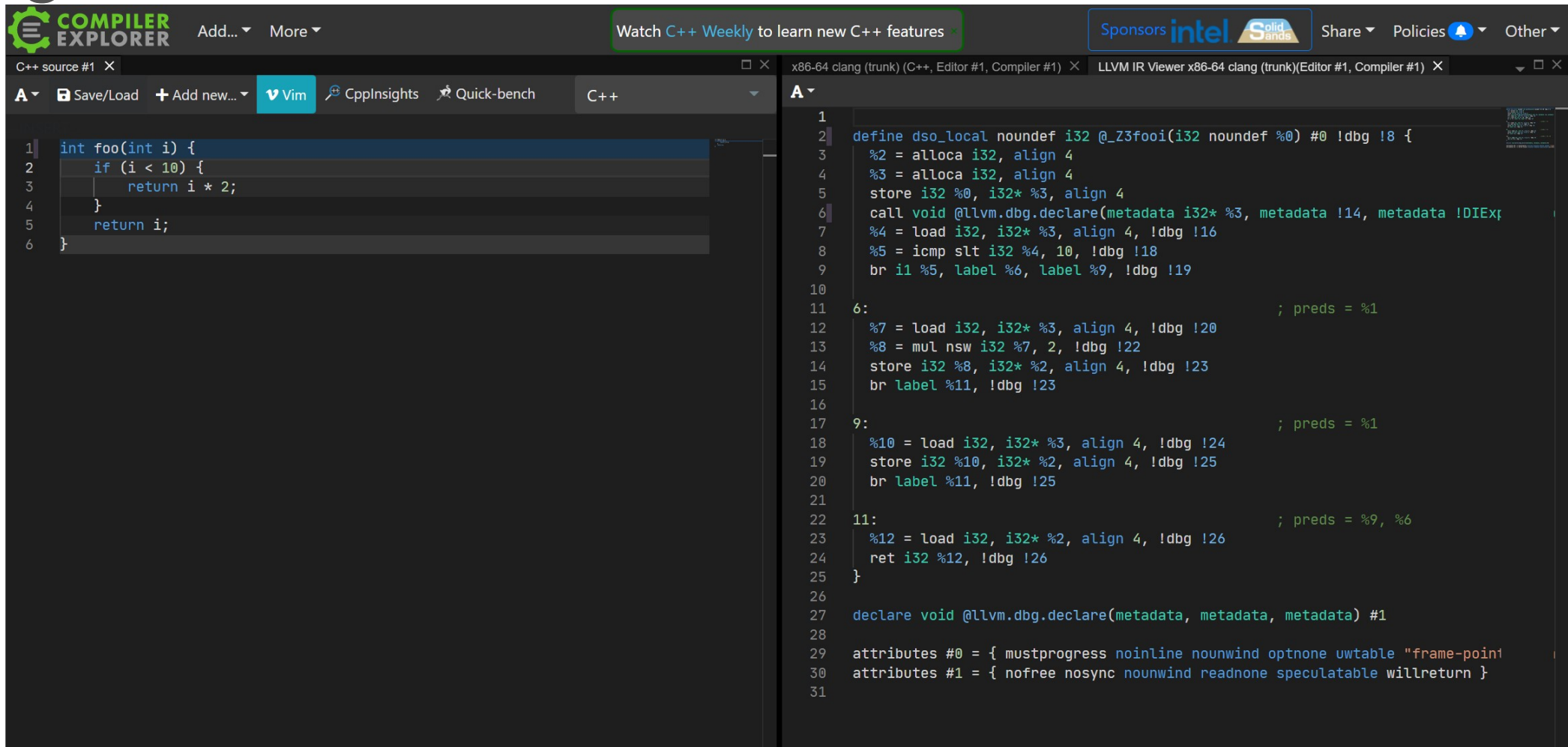


Semantic analysis

Sema Example



IR generation



The image shows a screenshot of the Compiler Explorer web application. The interface is split into two main panels. The left panel, titled 'C++ source #1', displays a C++ function named 'foo' that takes an integer 'i' and returns 'i' if 'i' is less than 10, otherwise it returns 'i * 2'. The right panel, titled 'LLVM IR Viewer x86-64 clang (trunk)', shows the corresponding LLVM Intermediate Representation (IR) for the C++ code. The IR is written in assembly-like syntax, showing memory allocation, loading, storing, and branching instructions. The top of the interface includes the 'Compiler Explorer' logo, a search bar, and various utility buttons like 'Save/Load', 'Add new...', 'Vim', 'Cpplnsights', and 'Quick-bench'. There are also links to 'Watch C++ Weekly' and 'Sponsors'.

```
1 int foo(int i) {  
2     if (i < 10) {  
3         return i * 2;  
4     }  
5     return i;  
6 }
```

```
1  
2 define dso_local noundef i32 @_Z3fooi(i32 noundef %0) #0 !dbg !8 {  
3     %2 = alloca i32, align 4  
4     %3 = alloca i32, align 4  
5     store i32 %0, i32* %3, align 4  
6     call void @llvm.dbg.declare(metadata i32* %3, metadata !14, metadata !DIExp  
7     %4 = load i32, i32* %3, align 4, !dbg !16  
8     %5 = icmp slt i32 %4, 10, !dbg !18  
9     br i1 %5, label %6, label %9, !dbg !19  
10  
11 ; preds = %1  
12     %7 = load i32, i32* %3, align 4, !dbg !20  
13     %8 = mul nsw i32 %7, 2, !dbg !22  
14     store i32 %8, i32* %2, align 4, !dbg !23  
15     br label %11, !dbg !23  
16  
17 ; preds = %1  
18     %10 = load i32, i32* %3, align 4, !dbg !24  
19     store i32 %10, i32* %2, align 4, !dbg !25  
20     br label %11, !dbg !25  
21  
22 ; preds = %9, %6  
23     %12 = load i32, i32* %2, align 4, !dbg !26  
24     ret i32 %12, !dbg !26  
25 }  
26  
27 declare void @llvm.dbg.declare(metadata, metadata, metadata) #1  
28  
29 attributes #0 = { mustprogress noline nounwind optnone uwtable "frame-point  
30 attributes #1 = { nofree nosync nounwind readnone speculatable willreturn }  
31
```



<https://godbolt.org/z/or3z3oYnq>

LLVM IR

LLVM IR is used for several key purposes within the LLVM compiler framework:

- Platform Independence
- Optimization
- Code Generation

C++

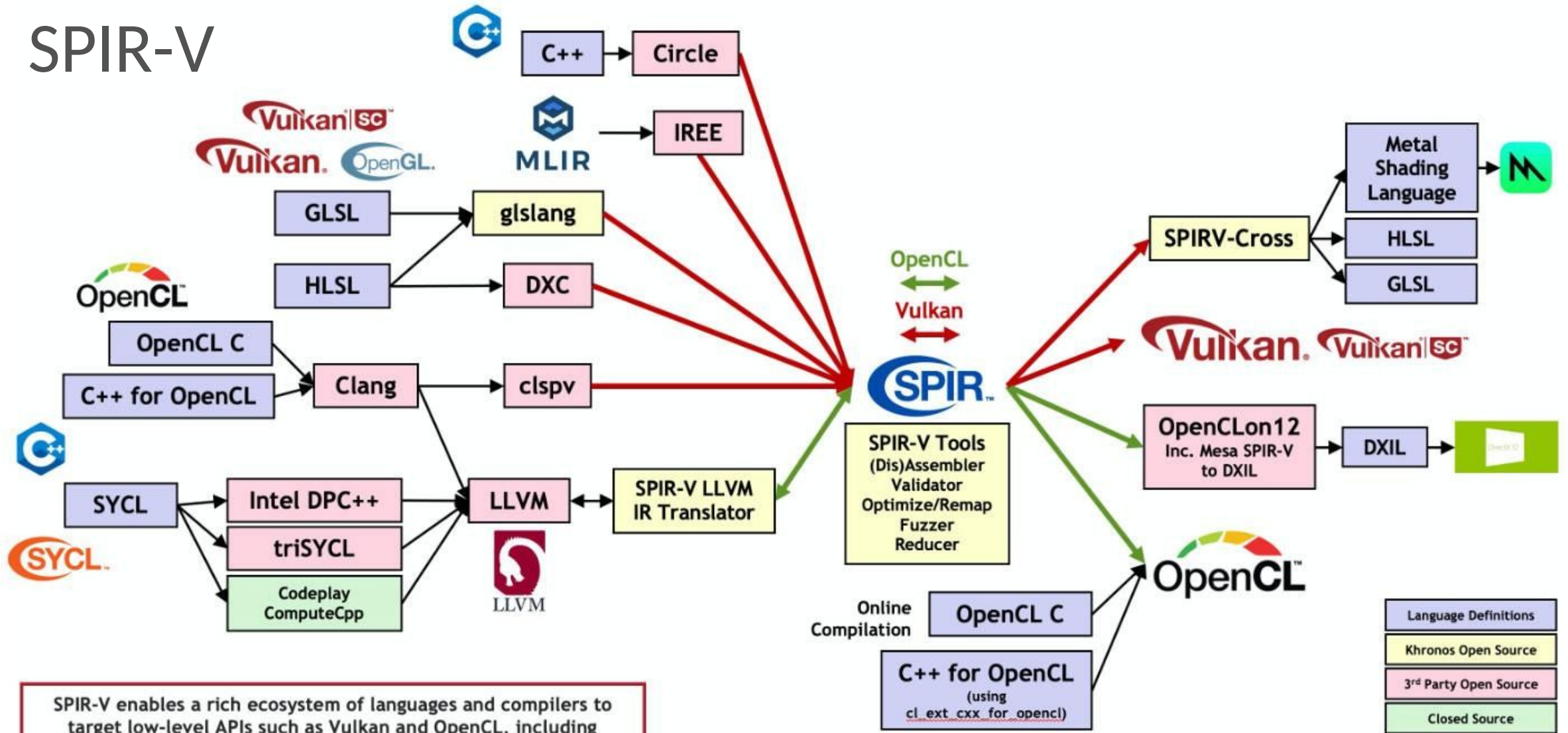
```
int add(int a, int b) {  
    ... return a + b;  
}
```

The corresponding LLVM IR for this function might look something like this:

LLVM

```
define i32 @add(i32 %a, i32 %b) {  
entry:  
    %0 = add i32 %a, %b  
    ret i32 %0  
}
```

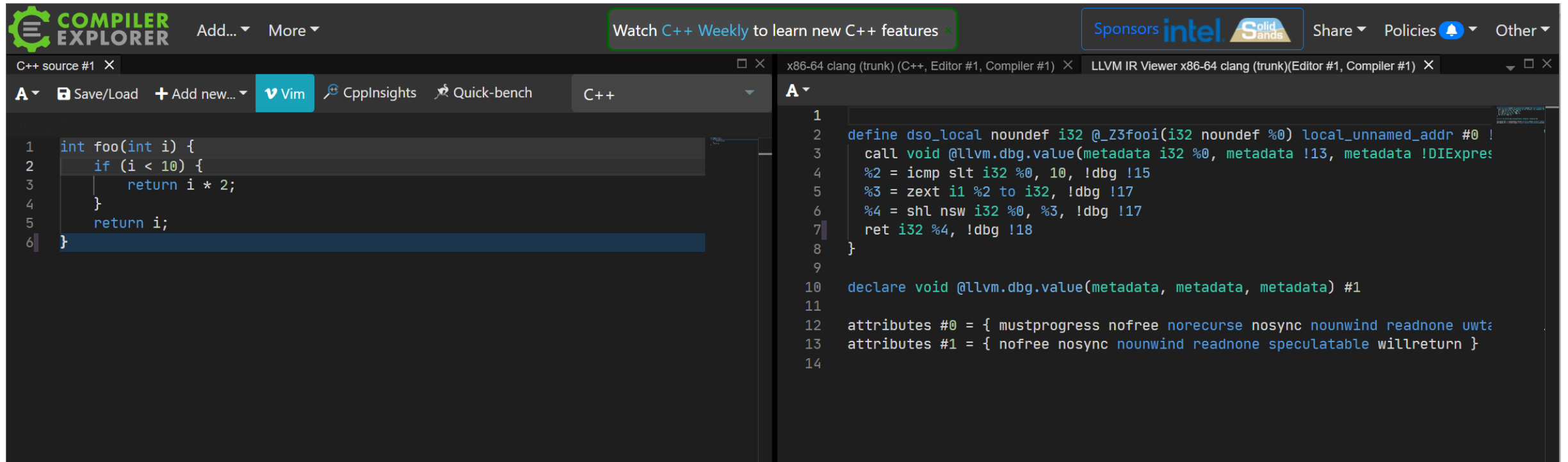
SPIR-V



SPIR-V enables a rich ecosystem of languages and compilers to target low-level APIs such as Vulkan and OpenCL, including deployment flexibility: e.g., running OpenCL kernels on Vulkan

<https://www.khronos.org/spir/>

IR optimization



The screenshot displays the Compiler Explorer interface. The left pane shows the C++ source code for a function `foo` that checks if `i < 10` and returns `i * 2` if true, otherwise `i`. The right pane shows the corresponding LLVM IR, which includes a function definition `@_Z3fooi(i32 noundef %0)` and a `declare` statement for `@llvm.dbg.value`. The IR also contains several `attributes` for the function and its arguments.

COMPILER EXPLORER Add... More ▾ Watch C++ Weekly to learn new C++ features × Sponsors intel Solid Sands Share ▾ Policies ▾ Other ▾

C++ source #1 × x86-64 clang (trunk) (C++, Editor #1, Compiler #1) × LLVM IR Viewer x86-64 clang (trunk)(Editor #1, Compiler #1) ×

Save/Load + Add new... Vim CppInsights Quick-bench C++

```
1 int foo(int i) {
2     if (i < 10) {
3         return i * 2;
4     }
5     return i;
6 }
```

```
1
2 define dso_local noundef i32 @_Z3fooi(i32 noundef %0) local_unnamed_addr #0 !
3     call void @llvm.dbg.value(metadata i32 %0, metadata !13, metadata !DIExpres
4     %2 = icmp slt i32 %0, 10, !dbg !15
5     %3 = zext i1 %2 to i32, !dbg !17
6     %4 = shl nsw i32 %0, %3, !dbg !17
7     ret i32 %4, !dbg !18
8 }
9
10 declare void @llvm.dbg.value(metadata, metadata, metadata) #1
11
12 attributes #0 = { mustprogress nofree norecurse nosync nounwind readnone uwta
13 attributes #1 = { nofree nosync nounwind readnone speculatable willreturn }
14
```



<https://godbolt.org/z/YM6Wxno37>

Code generation & optimization

The screenshot shows the Compiler Explorer interface. The left pane displays the C++ source code for a function `foo` that checks if `i < 10` and returns `i * 2` if true, otherwise `i`. The right pane shows the generated assembly code for x86-64 using clang (trunk). The assembly includes instructions for moving `edi` to `eax`, comparing `edi` with 10, setting the `cl` flag, shifting `eax` left by `cl`, and returning. Below the assembly, resource pressure metrics are shown for 10 different resources (SKXDivider, SKXFPDivider, and SKXPorts).

Resources:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-	-	2.51	1.49	0.50	0.50	-	1.50	2.50	-

Resource pressure per iteration:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
-	-	2.51	1.49	0.50	0.50	-	1.50	2.50	-

Resource pressure by instruction:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Instructions:
-	-	-	0.46	-	-	-	0.53	0.01	-	mov eax, edi
-	-	-	0.52	-	-	-	0.48	-	-	cmp edi, 10
-	-	0.50	-	-	-	-	-	0.50	-	setl cl
-	-	2.01	-	-	-	-	-	0.99	-	shl eax, cl
-	-	-	0.51	0.50	0.50	-	0.49	1.00	-	ret



<https://godbolt.org/z/n3orv3Ydj>

Linking

- Compiler operates on Translation Units
- Each translation unit produces an object file
- Linker assembles multiple object files and libraries into a program
- Not covered in this course
- A few pointers for those, who want to know more about linkers
 - [How the C++ Linker Works – YouTube](#)
 - [2017 LLVM Developers' Meeting: R. Ueyama "lld: A Fast, Simple, and Portable Linker" - YouTube](#)

Debugging

- Compilers try to preserve source code <-> binary mapping
 - They're not very successful when optimizations are involved
- Debug info is stored in a special format within a binary or in a separate file: lookup DWARF and PDB
- Debuggers get help from OS and hardware to enable breakpoints
- Will be covered in this course (stretch goal)
- More info:
 - [2016 EuroLLVM Developers' Meeting: D. Panickal & A. Warzynski "LLDB Tutorial: Adding debugger ..." - YouTube](#)
 - [2013 LLVM Developers' Meeting: "Adapting LLDB for your hardware: Remote Debugging the Hexagon DSP" - YouTube](#)

Static analysis

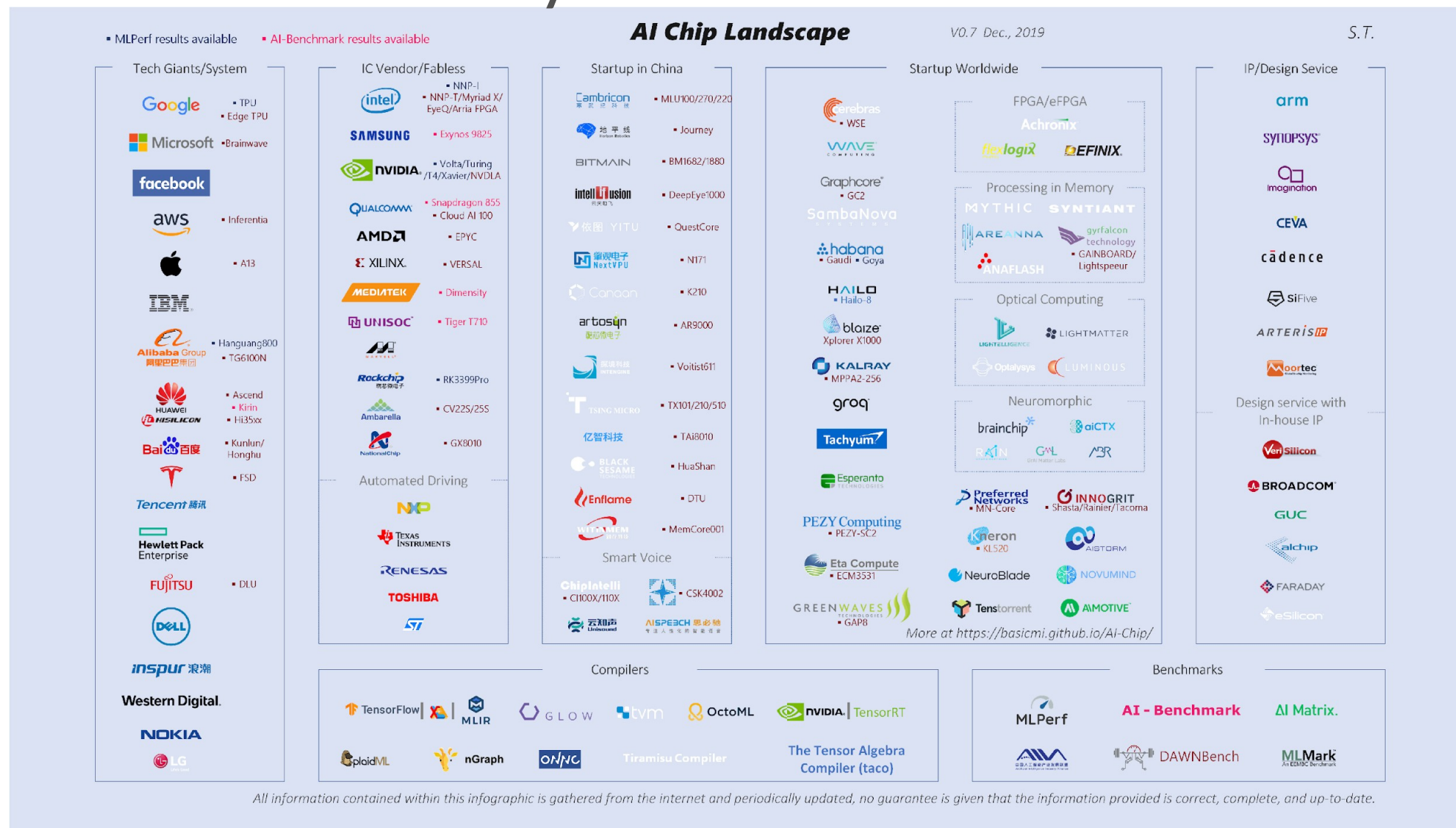
- Humans are not very good at computer programming
 - Lack of context, poor cognitive skills
- ASTs and IRs can be used to help developers find bugs
- Examples:
 - clang-tidy, PVS Studio, cppcheck, basically any IDE
 - More advanced: Rust borrow checker
- Will be covered in this course
- More info:
 - [C++ Weekly - Ep 3 Intro to clang-tidy – YouTube](#)
 - [2020 LLVM Developers' Meeting: "Using Clang-tidy for Customized Checkers and Large Scale Source..." - YouTube](#)

LLVM

- Compiler framework
- Includes LLVM IR and backends for many platforms
- Started by Chris Lattner in 2001
- Multiple frontends under LLVM umbrella: clang, flang
- Photo: <https://www.modular.ai/team>



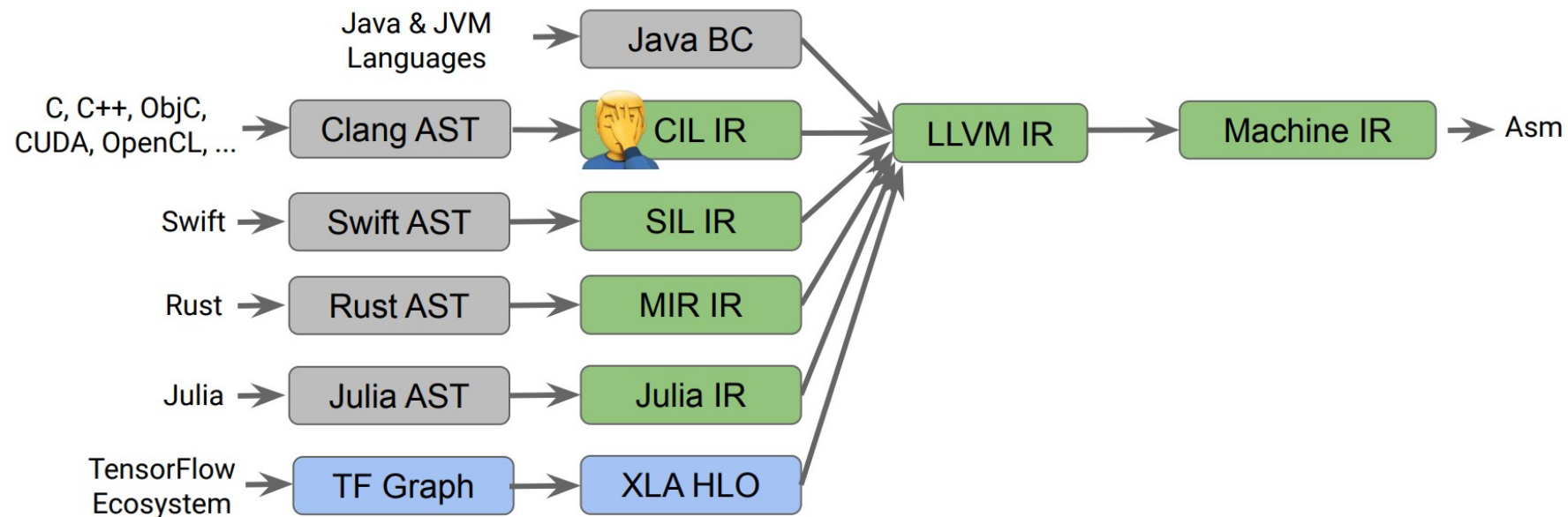
Hardware diversity



<https://basicmi.github.io/AI-Chip/>

Post-modern compilers

From Programming Languages to the TensorFlow Compiler



- Domain specific optimizations, progressive lowering
- Common LLVM platform for mid/low-level optimizing compilation in SSA form



<https://llvm-hpc-2020-workshop.github.io/presentations/llvmhpc2020-amini.pdf>

AI compilers

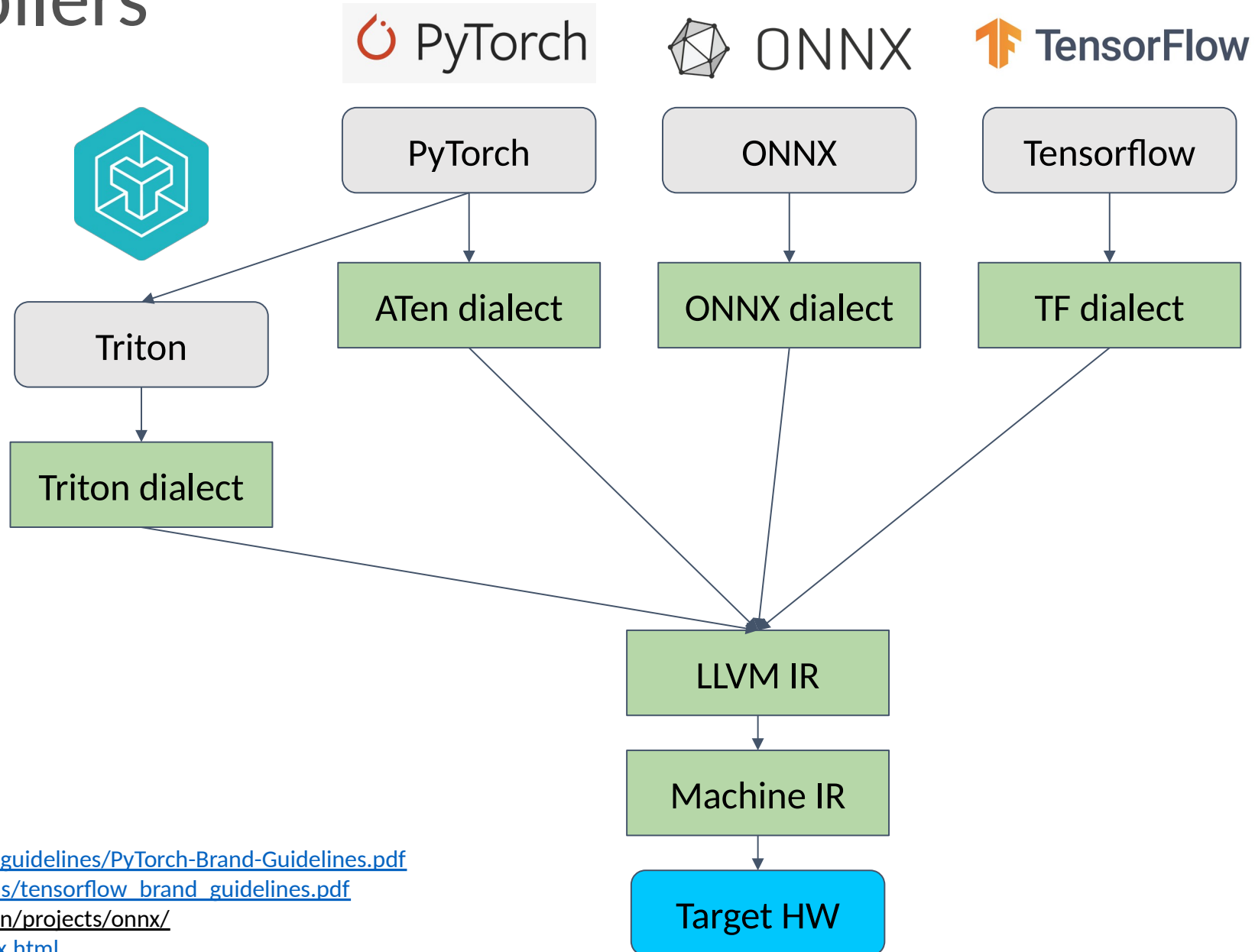


Photo:

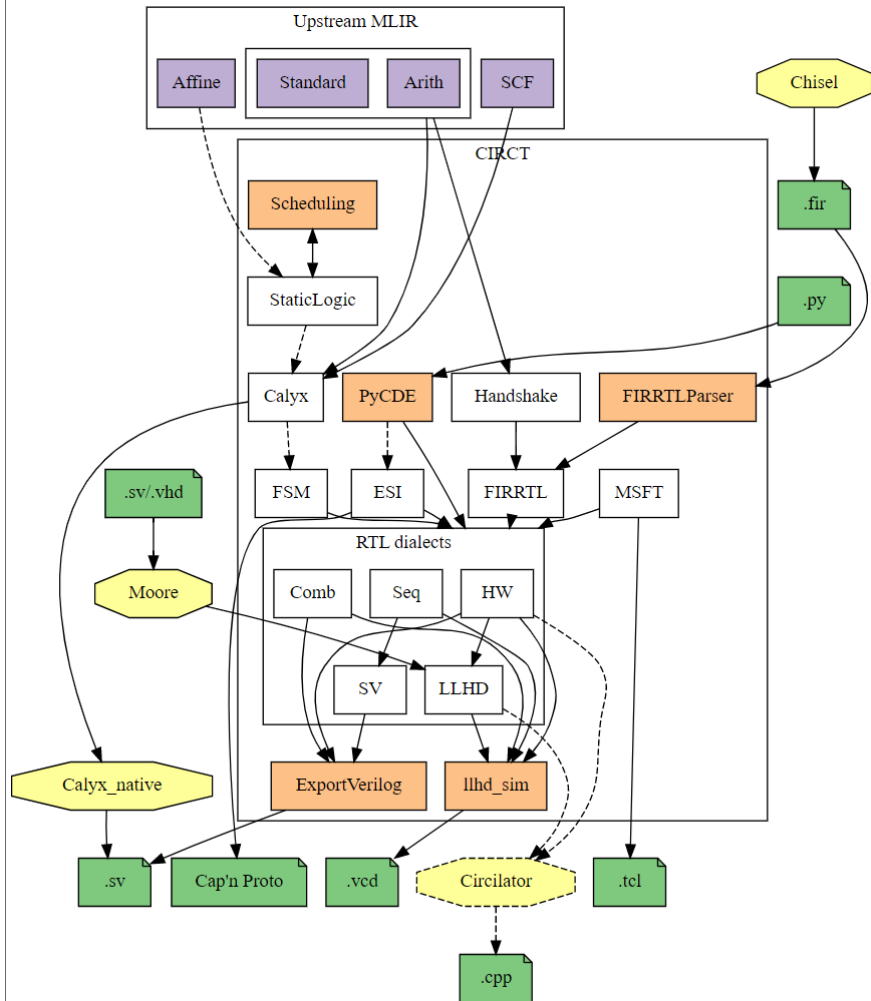
<https://pytorch.org/assets/brand-guidelines/PyTorch-Brand-Guidelines.pdf>

https://www.tensorflow.org/extras/tensorflow_brand_guidelines.pdf

<https://artwork.lfaidata.foundation/projects/onnx/>

<https://triton-lang.org/main/index.html>

Customized hardware



- “CIRCT” / Circuit IR Compilers and Tools
- Compilers help design custom hardware
- Reduce cost of development and validation
- <https://circt.llvm.org/>

Career paths

- hardware vendors
- large software companies
- AI development

In the next episode...

- Compiler frontends
 - Different frontend stages: lexers, syntax, sema
 - Clang as an example

Extra materials

- Interview with Jim Keller
 - Part 1: <https://www.youtube.com/watch?v=Nb2tebYAaOA>
 - Part 2: <https://www.youtube.com/watch?v=G4hL5Om4IJ4>
- Interview with Chris Lattner
 - Part 1: <https://www.youtube.com/watch?v=yCd3CzGSte8>
 - Part 2: <https://www.youtube.com/watch?v=nWTvXbQHwWs>
- An overview of Clang, LLVM Dev 2019 - <https://www.youtube.com/watch?v=5kkMpJpIGYU>
- Compilers: Principles, Techniques, and Tools (aka the dragon book) - <https://suif.stanford.edu/dragonbook/>
- What Has My Compiler Done for Me Lately? (Matt Godbolt) - <https://www.youtube.com/watch?v=bSkpMdDe4g4>
- Engineering a Compiler (Keith D. Cooper & Linda Torczon) - <https://github.com/lighthouseand/books/blob/master/Engineering%20A%20Compiler%202nd%20Edition%20by%20Cooper%20and%20Torczon.pdf>
- Chandler Carruth
 - Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My! - <https://www.youtube.com/watch?v=nXaxk27zwlk>
 - High Performance Code 201: Hybrid Data Structures - <https://www.youtube.com/watch?v=vElZc6zSIXM>

