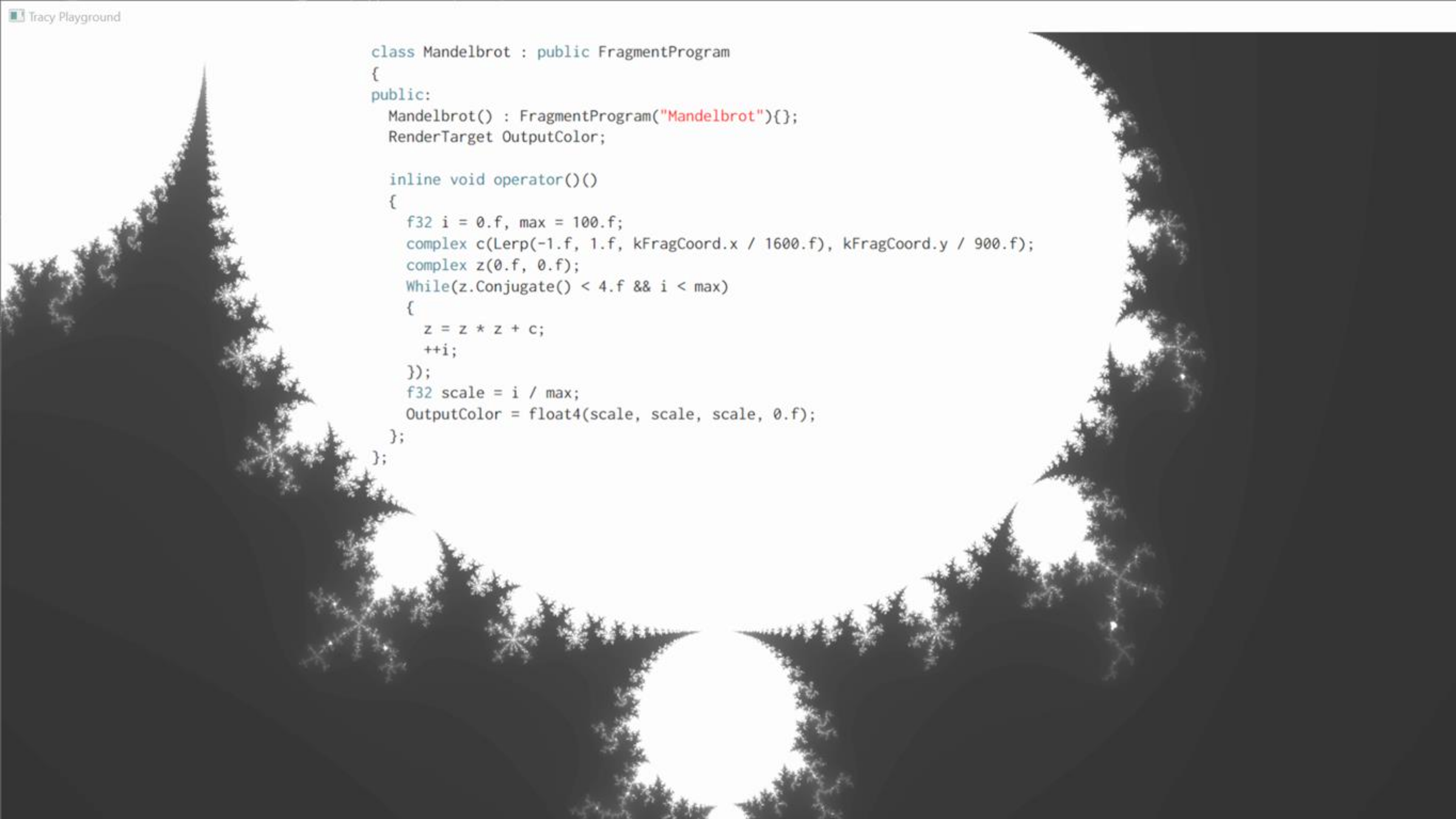
Personal Info

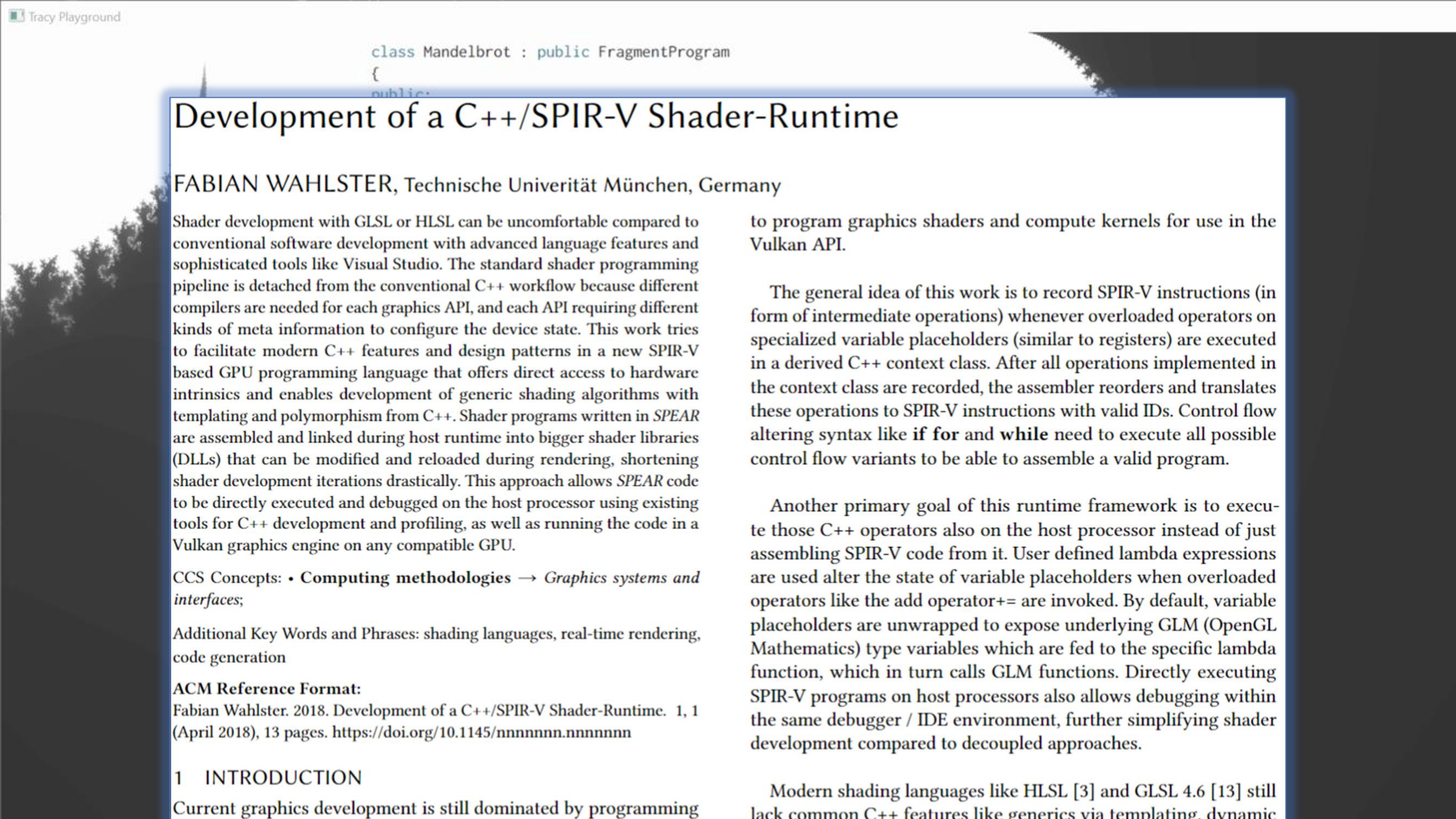
- **@SiNGUL4RiTY**
- **f.wahlster@tum.de**
- **https://github.com/razor8**

SPEAR

A C++/SPIR-V domain specific (shading) language



```
class Mandelbrot : public FragmentProgram
{
public:
  Mandelbrot() : FragmentProgram("Mandelbrot"){};
  RenderTarget OutputColor;

  inline void operator()()
  {
    f32 i = 0.f, max = 100.f;
    complex c(Lerp(-1.f, 1.f, kFragCoord.x / 1600.f), kFragCoord.y / 900.f);
    complex z(0.f, 0.f);
    While(z.Conjugate() < 4.f && i < max)
    {
      z = z * z + c;
      ++i;
    });
    f32 scale = i / max;
    OutputColor = float4(scale, scale, scale, 0.f);
  };
};
```


Tracy Playground

```
class Mandelbrot : public FragmentProgram
{
public:
```

Development of a C++/SPIR-V Shader-Runtime

FABIAN WAHLSTER, Technische Univerität München, Germany

Shader development with GLSL or HLSL can be uncomfortable compared to conventional software development with advanced language features and sophisticated tools like Visual Studio. The standard shader programming pipeline is detached from the conventional C++ workflow because different compilers are needed for each graphics API, and each API requiring different kinds of meta information to configure the device state. This work tries to facilitate modern C++ features and design patterns in a new SPIR-V based GPU programming language that offers direct access to hardware intrinsics and enables development of generic shading algorithms with templating and polymorphism from C++. Shader programs written in *SPEAR* are assembled and linked during host runtime into bigger shader libraries (DLLs) that can be modified and reloaded during rendering, shortening shader development iterations drastically. This approach allows *SPEAR* code to be directly executed and debugged on the host processor using existing tools for C++ development and profiling, as well as running the code in a Vulkan graphics engine on any compatible GPU.

CCS Concepts: • **Computing methodologies** → *Graphics systems and interfaces*;

Additional Key Words and Phrases: shading languages, real-time rendering, code generation

ACM Reference Format:
Fabian Wahlster. 2018. Development of a C++/SPIR-V Shader-Runtime. 1, 1 (April 2018), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

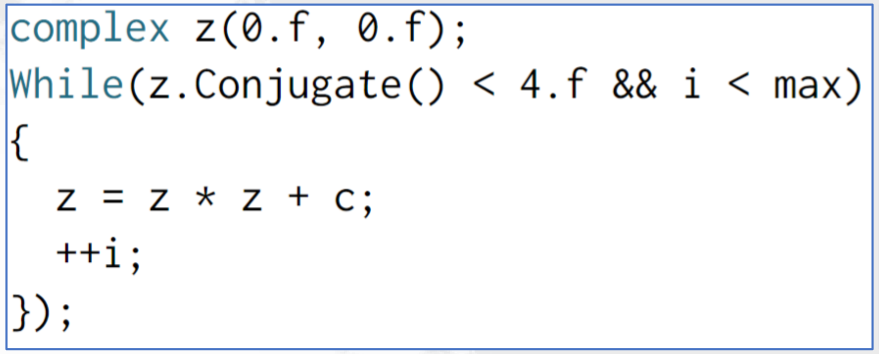
Current graphics development is still dominated by programming to program graphics shaders and compute kernels for use in the Vulkan API.

The general idea of this work is to record SPIR-V instructions (in form of intermediate operations) whenever overloaded operators on specialized variable placeholders (similar to registers) are executed in a derived C++ context class. After all operations implemented in the context class are recorded, the assembler reorders and translates these operations to SPIR-V instructions with valid IDs. Control flow altering syntax like **if for** and **while** need to execute all possible control flow variants to be able to assemble a valid program.

Another primary goal of this runtime framework is to execute those C++ operators also on the host processor instead of just assembling SPIR-V code from it. User defined lambda expressions are used alter the state of variable placeholders when overloaded operators like the add operator+= are invoked. By default, variable placeholders are unwrapped to expose underlying GLM (OpenGL Mathematics) type variables which are fed to the specific lambda function, which in turn calls GLM functions. Directly executing SPIR-V programs on host processors also allows debugging within the same debugger / IDE environment, further simplifying shader development compared to decoupled approaches.

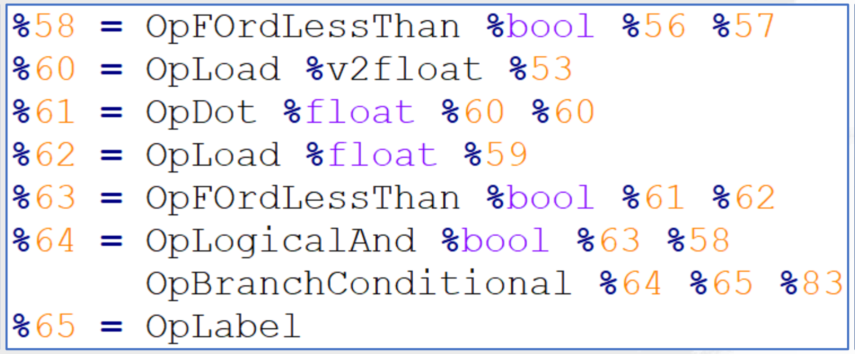
Modern shading languages like HLSL [3] and GLSL 4.6 [13] still lack common C++ features like generics via templating, dynamic

SPEAR to SPIR-V



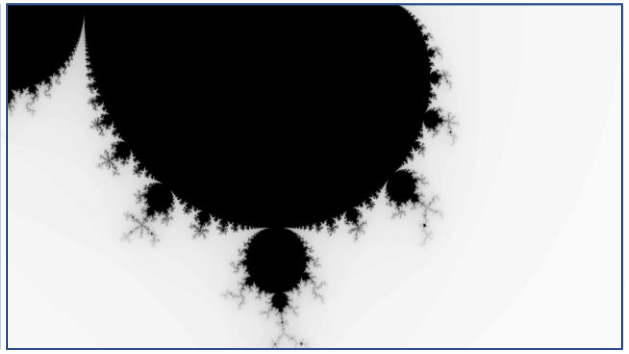
```
complex z(0.f, 0.f);
While(z.Conjugate() < 4.f && i < max)
{
  z = z * z + c;
  ++i;
});
```

SPEAR iDSL Code



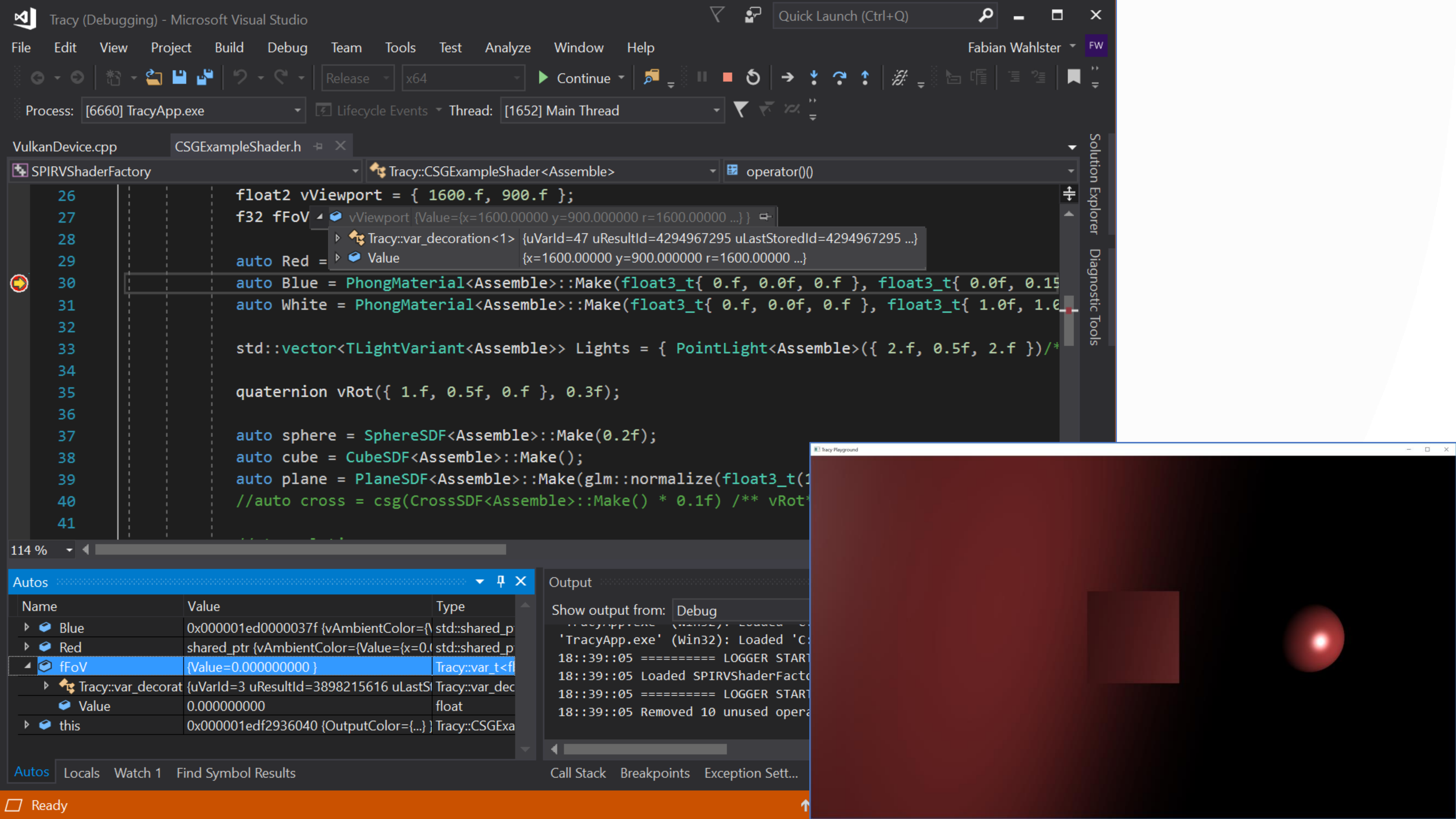
```
%58 = OpFOrdLessThan %bool %56 %57
%60 = OpLoad %v2float %53
%61 = OpDot %float %60 %60
%62 = OpLoad %float %59
%63 = OpFOrdLessThan %bool %61 %62
%64 = OpLogicalAnd %bool %63 %58
      OpBranchConditional %64 %65 %83
%65 = OpLabel
```

SPIR-V Assembly Code



Vulkan Rendering

- High-level language features
- Modularity and Extensibility
- Interchangeable Shading Libraries


Tracy (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Fabian Wahlster FW
Release x64 Continue
Process: [6660] TracyApp.exe
Lifecycle Events
Thread: [1652] Main Thread
VulkanDevice.cpp
CSGExampleShader.h
SPIRVShaderFactory
Tracy::CSGExampleShader<Assemble>
operator()()
Solution Explorer
Diagnostic Tools
114 %
Autos
Output
Show output from: Debug
Autos Locals Watch 1 Find Symbol Results
Call Stack Breakpoints Exception Sett...
Ready
Tracy Playground

```
        float2 vViewport = { 1600.f, 900.f };
        f32 fFoV
        
        auto Red =
        auto Blue = PhongMaterial<Assemble>::Make(float3_t{ 0.f, 0.0f, 0.f }, float3_t{ 0.0f, 0.15
        auto White = PhongMaterial<Assemble>::Make(float3_t{ 0.f, 0.0f, 0.f }, float3_t{ 1.0f, 1.0

        std::vector<TLightVariant<Assemble>> Lights = { PointLight<Assemble>({ 2.f, 0.5f, 2.f })/*

        quaternion vRot({ 1.f, 0.5f, 0.f }, 0.3f);

        auto sphere = SphereSDF<Assemble>::Make(0.2f);
        auto cube = CubeSDF<Assemble>::Make();
        auto plane = PlaneSDF<Assemble>::Make(glm::normalize(float3_t(1
        //auto cross = csg(CrossSDF<Assemble>::Make() * 0.1f) /** vRot*
```

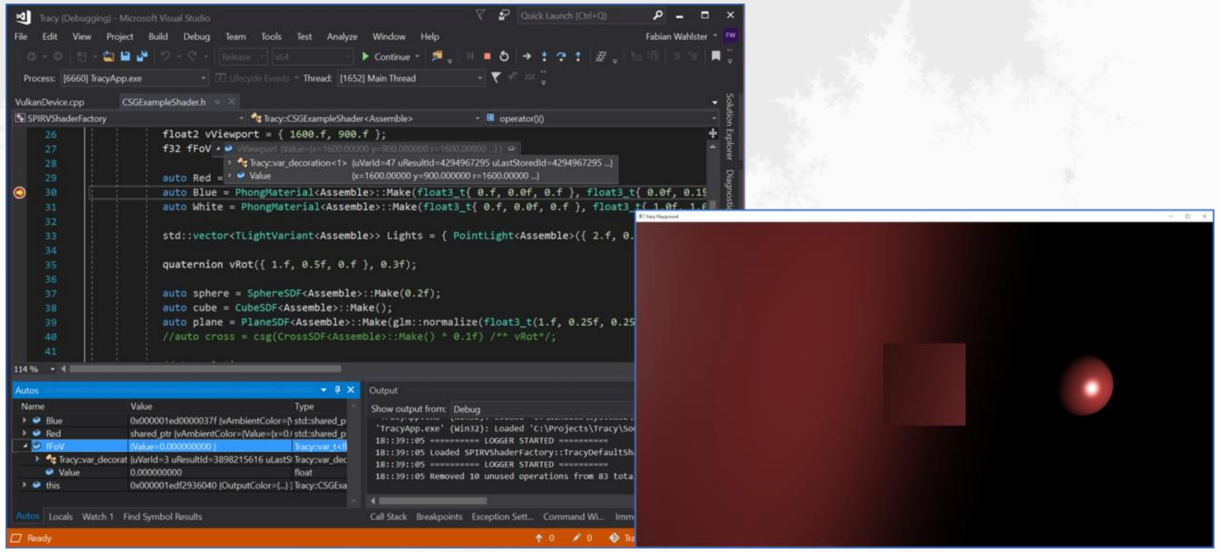
```
vViewport {Value={x=1600.00000 y=900.000000 r=1600.00000 ...} }
  Tracy::var_decoration<1>  {uVarId=47 uResultId=4294967295 uLastStoredId=4294967295 ...}
  Value                     {x=1600.00000 y=900.000000 r=1600.00000 ...}
```

Name	Value	Type
Blue	0x000001ed0000037f {vAmbientColor={\	std::shared_p
Red	shared_ptr {vAmbientColor={Value={x=0.(	std::shared_p
fFoV	{Value=0.000000000 }	Tracy::var_t<fl
Tracy::var_decorat	{uVarId=3 uResultId=3898215616 uLastSt	Tracy::var_dec
Value	0.000000000	float
this	0x000001edf2936040 {OutputColor={...} }	Tracy::CSGExa

```
'TracyApp.exe' (Win32): Loaded 'C:
18::39::05 ========== LOGGER START
18::39::05 Loaded SPIRVShaderFacto
18::39::05 ========== LOGGER START
18::39::05 Removed 10 unused opera
```

SPEAR to SPIR-V

- Write meta programs / code generators
- Debugging and profiling on host
- Write custom dispatchers to exploit CPU Arch (work stealing etc.)



```
float2 vViewport = { 1600.f, 900.f };
f32 fFoV
    vViewport (Value={x=1600.00000 y=900.000000 r=1600.00000 ...})
      Tracy::var_decoration<1>  {uVarId=47 uResultId=4294967295 uLastStoredId=4294967295 ...}
      Value                     {x=1600.00000 y=900.000000 r=1600.00000 ...}
auto Red =
auto Blue = PhongMaterial<Assemble>::Make(float3_t{ 0.f, 0.0f, 0.f }, float3_t{ 0.0f, 0.15
auto White = PhongMaterial<Assemble>::Make(float3_t{ 0.f, 0.0f, 0.f }, float3_t{ 1.0f, 1.0

std::vector<TLightVariant<Assemble>> Lights = { PointLight<Assemble>({ 2.f, 0.

quaternion vRot({ 1.f, 0.5f, 0.f }, 0.3f);

auto sphere = SphereSDF<Assemble>::Make(0.2f);
auto cube = CubeSDF<Assemble>::Make();
auto plane = PlaneSDF<Assemble>::Make(glm::normalize(float3_t(1.f, 0.25f, 0.25
//auto cross = csg(CrossSDF<Assemble>::Make() * 0.1f) /** vRot*/;
```

Name	Value	Type
Blue	0x000001ed0000037f {vAmbientColor={\	std::shared_p
Red	shared_ptr {vAmbientColor={Value={x=0.(	std::shared_p
fFoV	{Value=0.000000000 }	Tracy::var_t<fl
Tracy::var_decorat	{uVarId=3 uResultId=3898215616 uLastS	Tracy::var_dec
Value	0.000000000	float
this	0x000001edf2936040 {OutputColor={...} }	Tracy::CSGExa

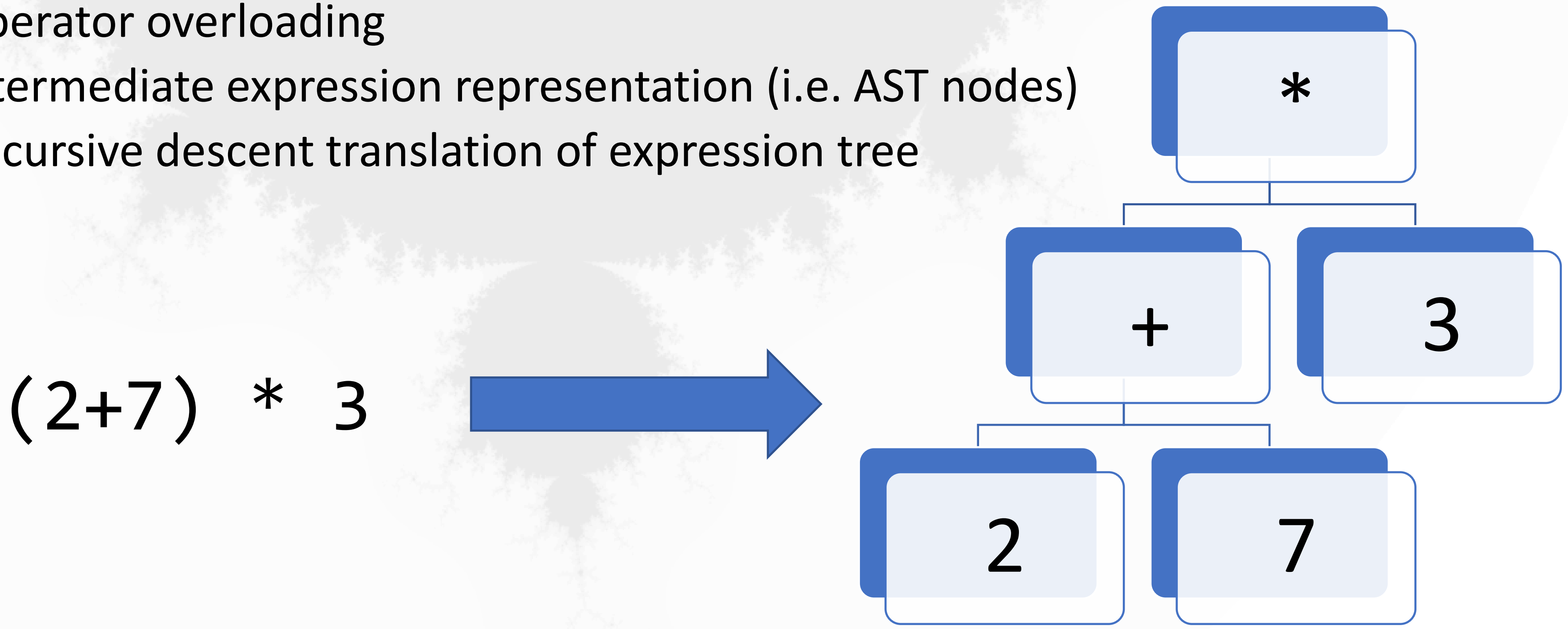
Show output from: Debug

```
'TracyApp.exe' (Win32): Loaded 'C:\Projects\Tracy\So
18::39::05 ========== LOGGER STARTED ==========
18::39::05 Loaded SPIRVShaderFactory::TracyDefaultSh
18::39::05 ========== LOGGER STARTED ==========
18::39::05 Removed 10 unused operations from 83 tota
```

Tracy (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Continue
Process: [6660] TracyApp.exe
Thread: [1652] Main Thread
VulkanDevice.cpp
CSGExampleShader.h
SPIRVShaderFactory
Tracy::CSGExampleShader<Assemble>
operator()()
114 %
Autos
Output
Autos Locals Watch 1 Find Symbol Results
Call Stack Breakpoints Exception Sett... Command Wi...
Ready

Integrated Domain Specific Language (iDSL)

- Sophisticated host language & toolset (C#, C++, Java)
- Target domain: Graphics / Shaders
  - Operator overloading
  - Intermediate expression representation (i.e. AST nodes)
  - Recursive descent translation of expression tree



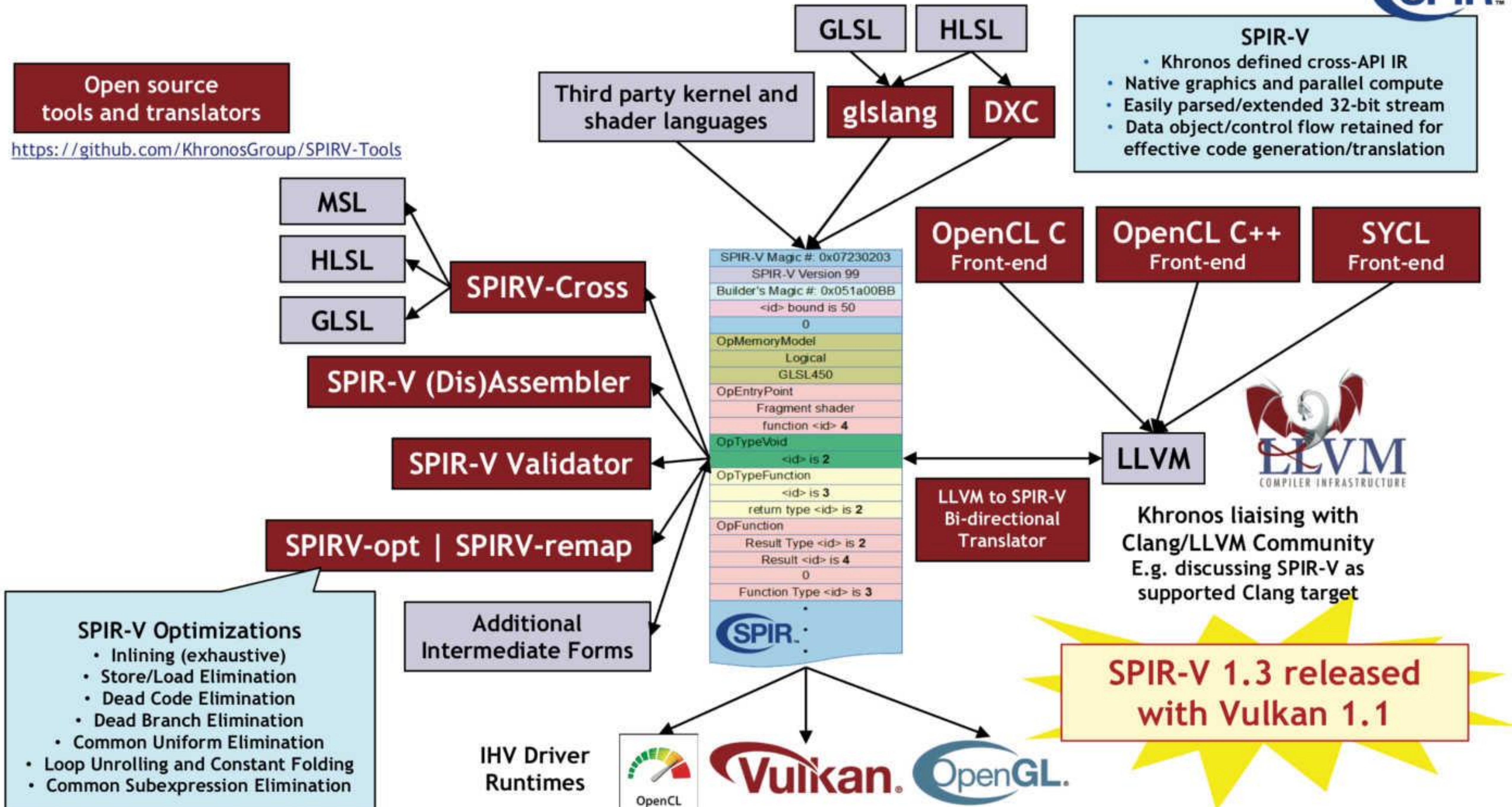
SPEAR iDSL

- Host language: **C++17**
- Target domain: **SPIR-V** with Shader Capabilites (no OpenCL kernels)
- Explicit (simplified) approach:
  - Operator overloading
  - Intermediate operation representation (Operation + Operand References)
  - Iterative resolve & translate pass

Open source tools and translators

https://github.com/KhronosGroup/SPIRV-Tools

SPIR-V
- Khronos defined cross-API IR
- Native graphics and parallel compute
- Easily parsed/extended 32-bit stream
- Data object/control flow retained for effective code generation/translation

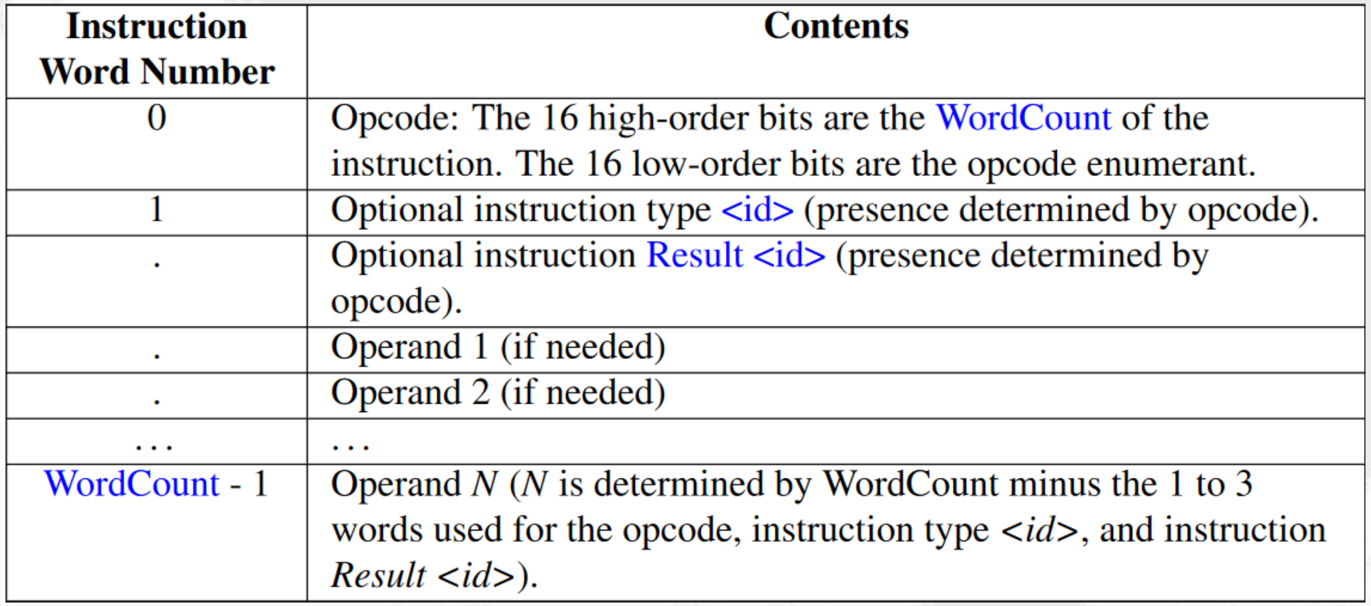


SPIR-V Optimizations
- Inlining (exhaustive)
- Store/Load Elimination
- Dead Code Elimination
- Dead Branch Elimination
- Common Uniform Elimination
- Loop Unrolling and Constant Folding
- Common Subexpression Elimination

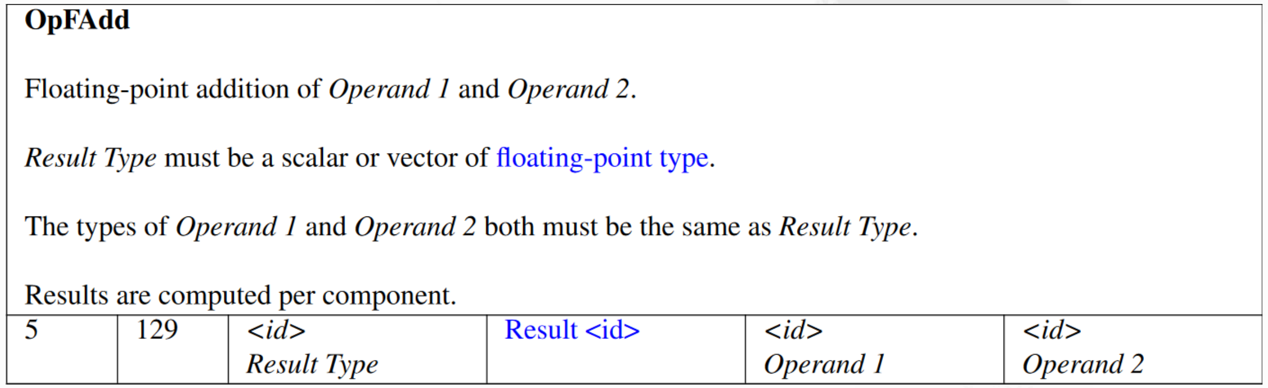
Khronos liaising with Clang/LLVM Community E.g. discussing SPIR-V as supported Clang target

SPIR-V 1.3 released with Vulkan 1.1

SPIR-V Instruction format



Instruction Word Number	Contents
0	Opcode: The 16 high-order bits are the WordCount of the instruction. The 16 low-order bits are the opcode enumerant.
1	Optional instruction type <id> (presence determined by opcode).
.	Optional instruction Result <id> (presence determined by opcode).
.	Operand 1 (if needed)
.	Operand 2 (if needed)
. . .	. . .
WordCount - 1	Operand *N* (*N* is determined by WordCount minus the 1 to 3 words used for the opcode, instruction type <*id*>, and instruction *Result* <*id*>).



OpFAdd

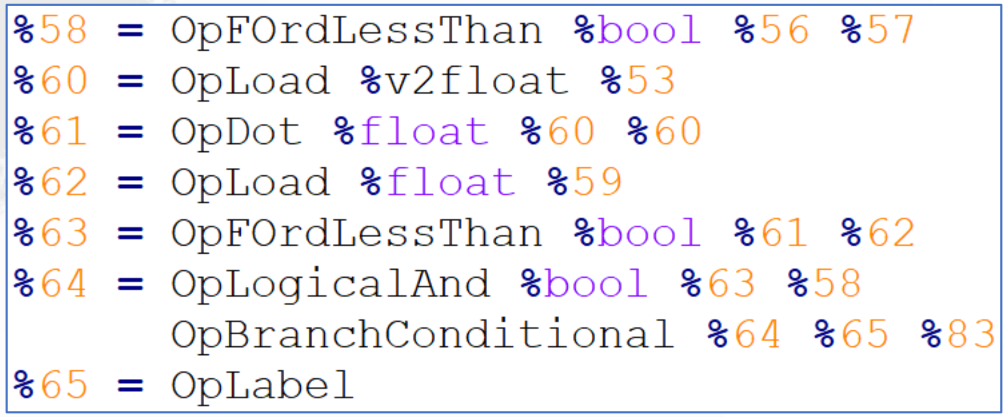
Floating-point addition of *Operand 1* and *Operand 2*.

Result Type must be a scalar or vector of floating-point type.

The types of *Operand 1* and *Operand 2* both must be the same as *Result Type*.

Results are computed per component.

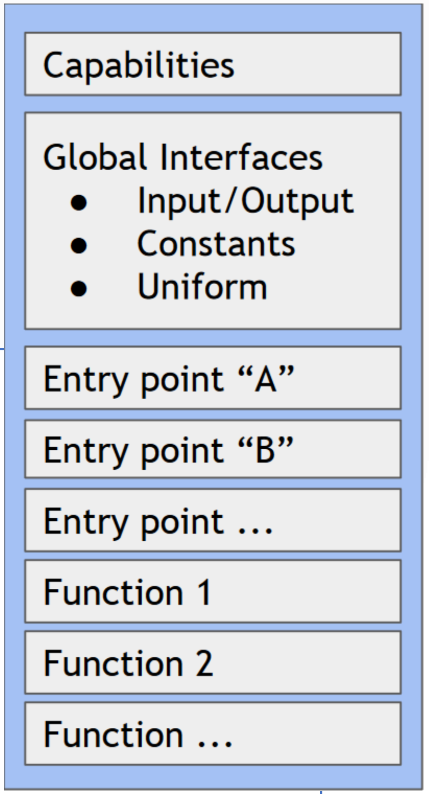
5	129	<*id*> *Result Type*	Result <id>	<*id*> *Operand 1*	<*id*> *Operand 2*



```
%58 = OpFOrdLessThan %bool %56 %57
%60 = OpLoad %v2float %53
%61 = OpDot %float %60 %60
%62 = OpLoad %float %59
%63 = OpFOrdLessThan %bool %61 %62
%64 = OpLogicalAnd %bool %63 %58
      OpBranchConditional %64 %65 %83
%65 = OpLabel
```

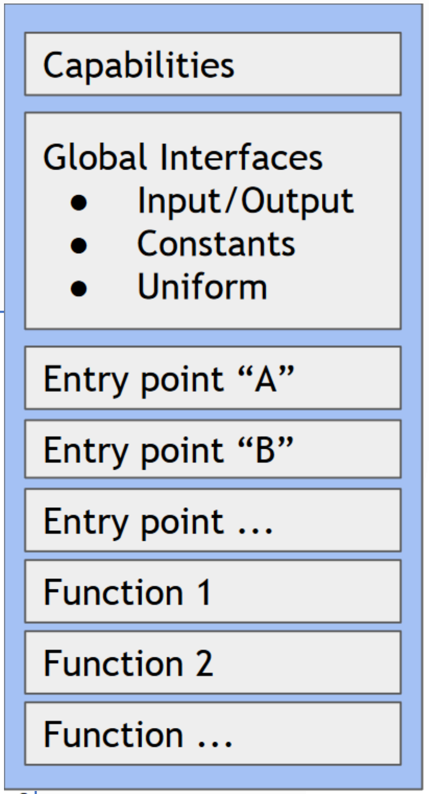
SPIR-V Module overview

```
// HEADER
; SPIR-V
; Version: 1.0
; Generator: Unknown(29817); 1
; Bound: 60
; Schema: 0

// PREAMBLE
               OpCapability Shader
               OpMemoryModel Logical GLSL450
               OpEntryPoint Vertex %28 "ScreenSpaceTriangle" %26 %gl_VertexIndex %45

// DECORATIONS
               OpDecorate %_struct_5 Block
               OpMemberDecorate %_struct_5 0 BuiltIn Position
               OpMemberDecorate %_struct_5 1 BuiltIn PointSize
               OpDecorate %gl_VertexIndex BuiltIn VertexIndex
               OpDecorate %45 Location 0
```



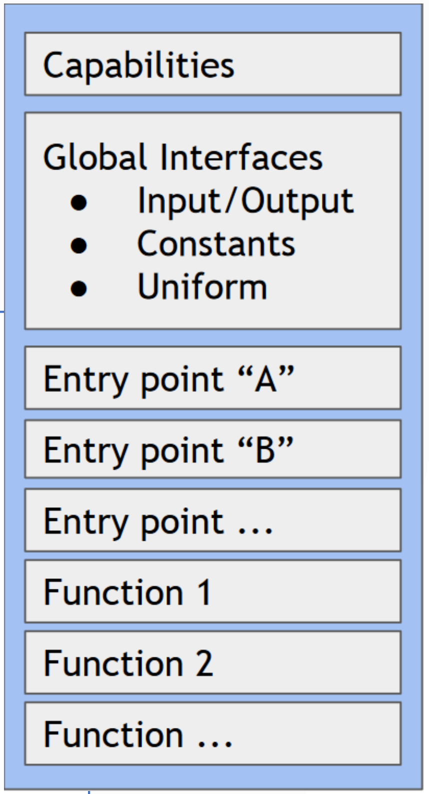
SPIR-V Module overview

```
// TYPES AND CONSTANTS
      %float = OpTypeFloat 32
    %v4float = OpTypeVector %float 4
%_ptr_Output_v4float = OpTypePointer Output %v4float
%_ptr_Output_float = OpTypePointer Output %float
  %_struct_5 = OpTypeStruct %v4float %float
%_ptr_Output__struct_5 = OpTypePointer Output %_struct_5
        %int = OpTypeInt 32 1
%_ptr_Input_int = OpTypePointer Input %int
      %int_0 = OpConstant %int 0
%_ptr_Input_v4float = OpTypePointer Input %v4float
    %float_0 = OpConstant %float 0
         %12 = OpConstantComposite %v4float %float_0 %float_0 %float_0 %float_0
%_ptr_Input_float = OpTypePointer Input %float
       %void = OpTypeVoid
         %15 = OpTypeFunction %void
      %int_1 = OpConstant %int 1
      %int_2 = OpConstant %int 2
%_ptr_Function_float = OpTypePointer Function %float
   %float_n1 = OpConstant %float -1
    %v2float = OpTypeVector %float 2
%_ptr_Output_v2float = OpTypePointer Output %v2float
    %float_1 = OpConstant %float 1
  %float_0_5 = OpConstant %float 0.5
       %uint = OpTypeInt 32 0
     %uint_0 = OpConstant %uint 0
```



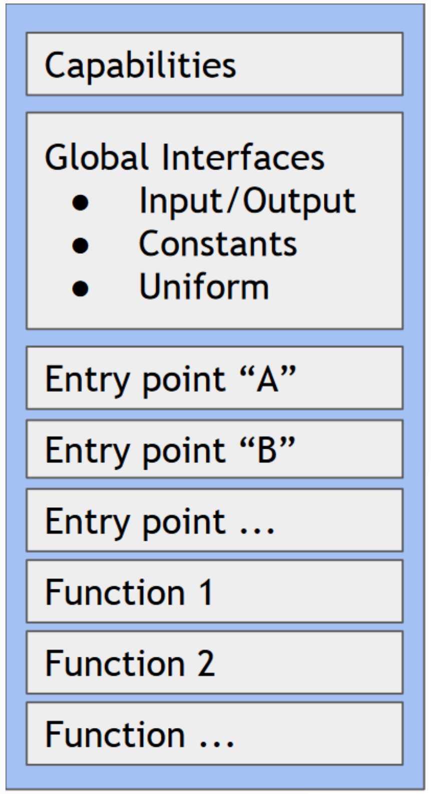
SPIR-V Module overview

```
// GLOBAL VARIABLES
         %26 = OpVariable %_ptr_Output__struct_5 Output
%gl_VertexIndex = OpVariable %_ptr_Input_int Input %int_0
         %45 = OpVariable %_ptr_Output_v2float Output

// ENTRY POINT
         %28 = OpFunction %void None %15
         %29 = OpLabel

// LOCAL VARIABLES
         %34 = OpVariable %_ptr_Function_float Function
         %35 = OpVariable %_ptr_Function_float Function %float_n1
         %41 = OpVariable %_ptr_Function_float Function
         %42 = OpVariable %_ptr_Function_float Function %float_n1
         %46 = OpVariable %_ptr_Function_float Function %float_1
         %52 = OpVariable %_ptr_Function_float Function %float_1
```



SPIR-V Module overview

```
// FUNCTION BODY
        %30 = OpLoad %int %gl_VertexIndex
        %31 = OpBitwiseAnd %int %30 %int_1
        %32 = OpShiftLeftLogical %int %31 %int_2
        %33 = OpConvertSToF %float %32
        %36 = OpLoad %float %35
        %37 = OpFAdd %float %36 %33
              OpStore %34 %33
        %38 = OpBitwiseAnd %int %30 %int_2
        %39 = OpShiftLeftLogical %int %38 %int_1
        %40 = OpConvertSToF %float %39
        %43 = OpLoad %float %42
        %44 = OpFAdd %float %43 %40
              OpStore %41 %40
        %47 = OpLoad %float %46
        %48 = OpFAdd %float %37 %47
        %49 = OpFMul %float %48 %float_0_5
        %50 = OpLoad %v2float %45
        %51 = OpCompositeInsert %v2float %49 %50 0
              OpStore %45 %51
        %53 = OpLoad %float %52
        %54 = OpFAdd %float %44 %53
        %55 = OpFMul %float %54 %float_0_5
        %56 = OpCompositeInsert %v2float %55 %51 1
              OpStore %45 %56
        %57 = OpCompositeConstruct %v4float %37 %44 %float_0 %float_1
        %58 = OpAccessChain %_ptr_Output_v4float %26 %uint_0
              OpStore %58 %57
              OpReturn
              OpFunctionEnd
```



Translating C++ to SPIR-V

SPEAR needs to capture the programs semantics

- All variables and constants created in the program
- Variable types → need structural reflection
- Any operation on variables (global or member)
- Control-flow: If, While, For, function calls

Translating C++ to SPIR-V

We can make some simplifying assumptions and only support:

- One EntryPoint “SPIRVProgram” per module
- A subset of “Shader” SPIR-V’s capabilities (No OpenCL compute)
- GLSL 4.50 ext instructions (almost all are implemented)
- The Logical memory model (also no pointers for now)

Translating C++ to SPIR-V

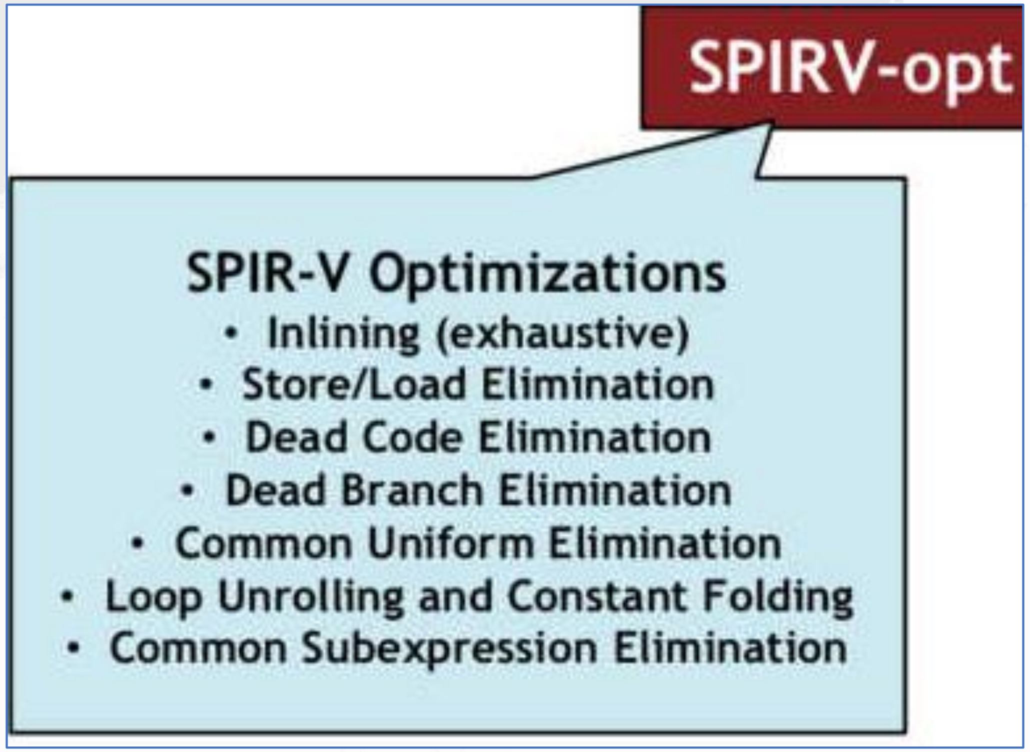
Take shortcuts:

- Everything is inlined, we can skip function calls ☺
- SPIR can be in SSA-Form (Result IDs always are)
  - Would require Phi-Nodes, more work
  - SPEAR uses Load/Store model instead (just like GLSLLang)
- Don't do custom optimization passes, just use **spirv-opt**

Working SPIR-V

SPIR-V Tools: (everything you need for SPIR-V manipulation)

https://github.com/KhronosGroup/SPIRV-Tools

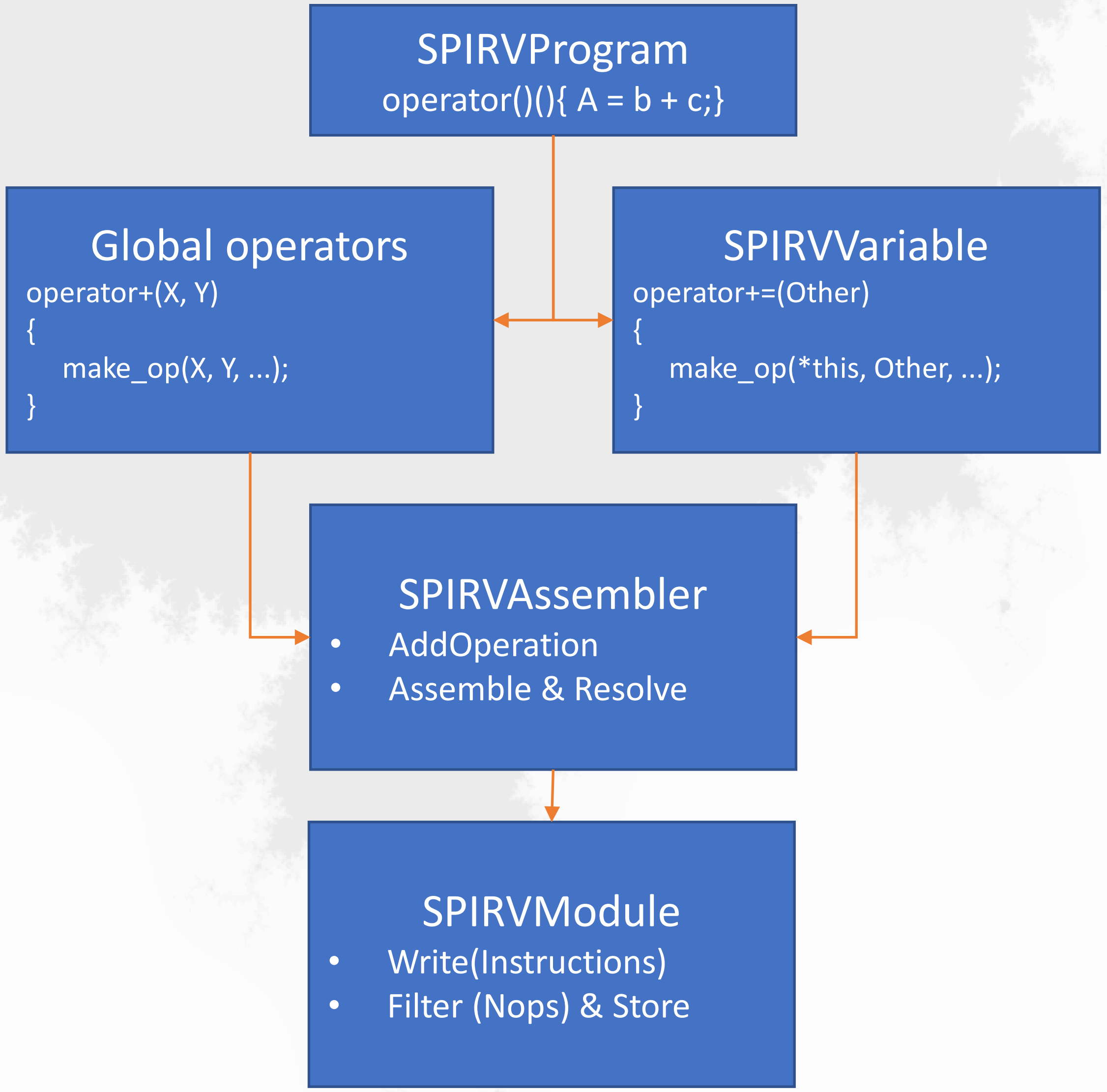


SPIRV-opt

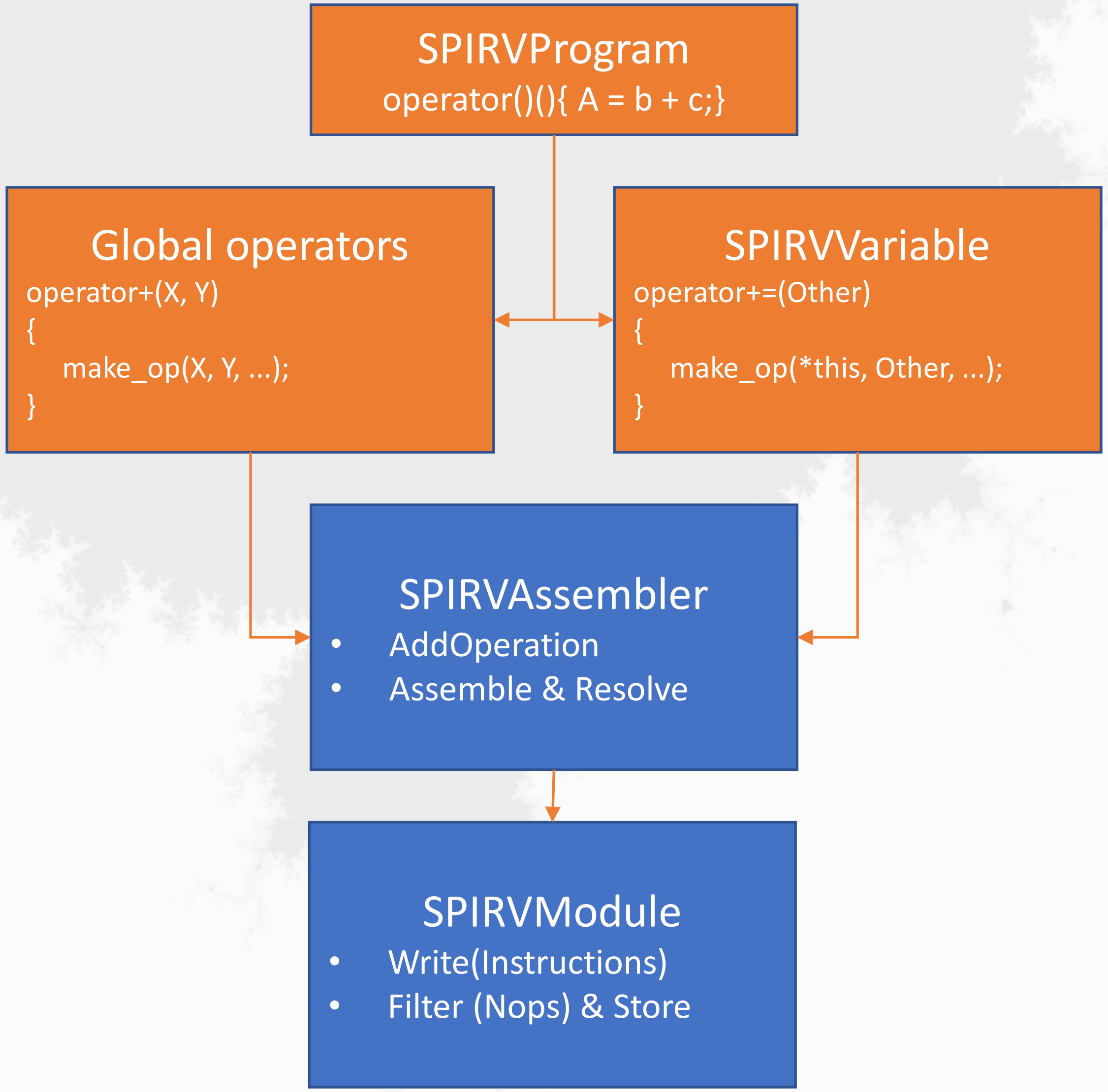
SPIR-V Optimizations

- Inlining (exhaustive)
- Store/Load Elimination
- Dead Code Elimination
- Dead Branch Elimination
- Common Uniform Elimination
- Loop Unrolling and Constant Folding
- Common Subexpression Elimination

SPEAR Framework



SPEAR Framework



```
operator+(X, Y)
{
    make_op(X, Y, ...);
}
```

```
operator+=(Other)
{
    make_op(*this, Other, ...);
}
```

- AddOperation
- Assemble & Resolve

- Write(Instructions)
- Filter (Nops) & Store

Translation via operator overloading

C++ does not allow overloading of primitive operators:

```
inline float operator*(const float& l, const float& r)
{
    // TODO: Emit SPIR-V opcode here
    return l * r;
}
```

→ We have to find a different solution for global operators

Translation via operator overloading

Solution: wrap types into another class:

```cpp
template <typename T, bool Assemble, spv::StorageClass Class>
struct var_t : public var_decoration<Assemble>
{
    T Value;
```

Host representation

```cpp
    // generates OpVar
    template <class... Ts>
    var_t(const Ts& ... _args);

    template <spv::StorageClass C1>
    var_t& operator+=(const var_t<T, Assemble, C1>& _Other);
};
```

Translation via operator overloading

Now we can overload global operators:

```
template <class T, bool Assemble, spv::StorageClass C1, spv::StorageClass C2>
inline var_t<T, Assemble, spv::StorageClassFunction>
operator+(const var_t<T, Assemble, C1>& l, const var_t<T, Assemble, C2>& r)
{
    // TODO: Emit SPIR-V opcode here
    return l.Value + r.Value;
}
```

SPIRVVariables – var_t<>

Store relevant SPIR-V meta information in the wrapper

GPU representation

```
struct var_decoration<true>
{
    SPIRVType Type;
    uint32_t uVarId = HUNDEFINED32; // result id OpVar or OpAccessChain
    uint32_t uResultId = HUNDEFINED32; // result of arithmetic instructions or OpLoad
    uint32_t uLastStoreId = HUNDEFINED32;
    spv::StorageClass kStorageClass = spv::StorageClassMax;
    uint32_t uTypeId = HUNDEFINED32;
    uint32_t uDescriptorSet = HUNDEFINED32; // res input
    uint32_t uBinding = HUNDEFINED32; // local to res input
    uint32_t uLocation = HUNDEFINED32; // res output

    void Store();
    uint32_t Load();
}
```

SPIRVProgram context class

Problem: variables can be instantiated before `operator()()` is executed:

```cpp
template <bool Assemble = true>
class Mandelbrot : public FragmentProgram<Assemble>
{
public:
    Mandelbrot() : FragmentProgram<Assemble>("Mandelbrot"){};

    RenderTarget OutputColor; // We have to start recording instructions
                              // even before the function body is executed
    inline void operator()()
    {
        ...
    }
};
```

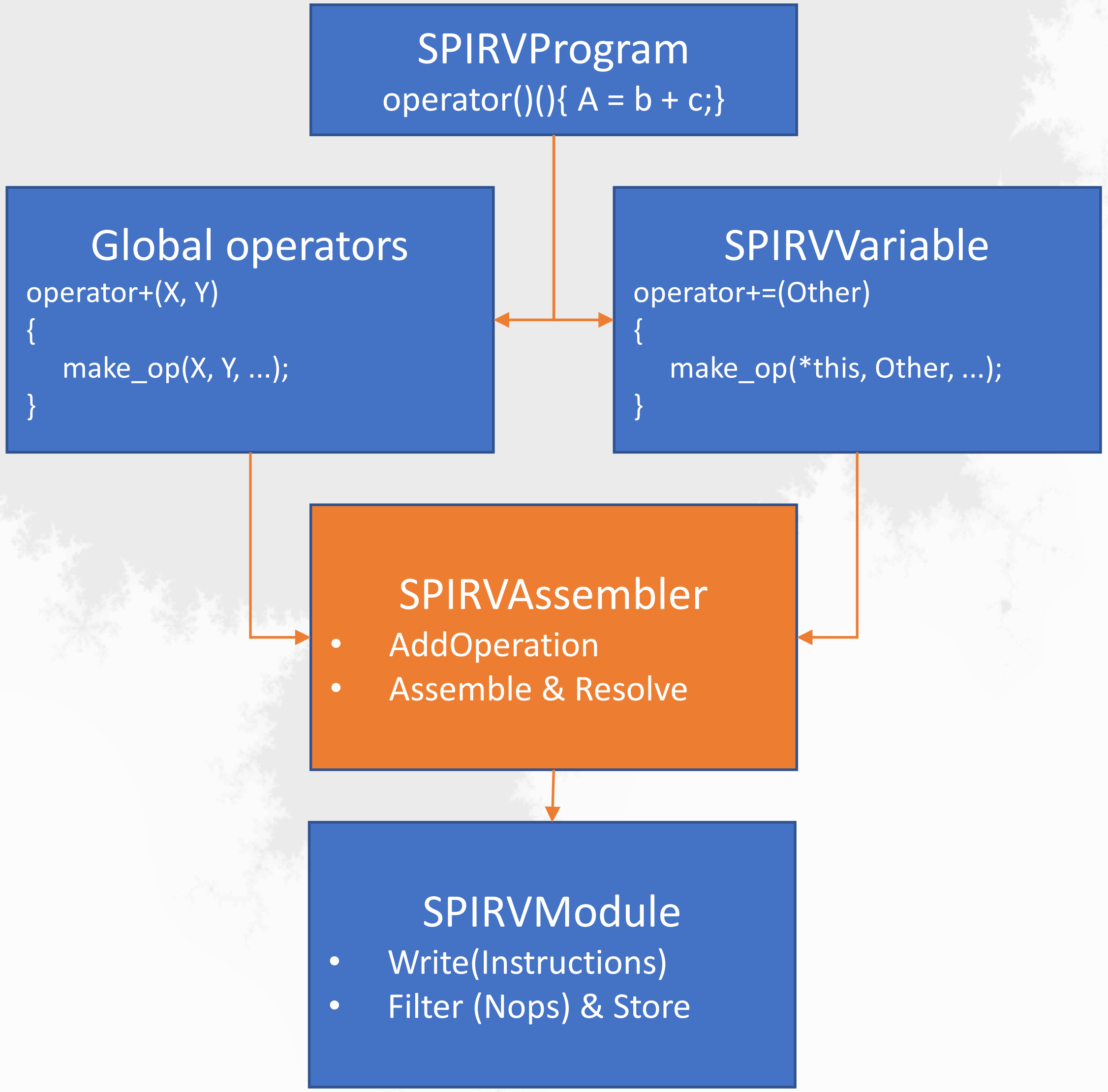
User shader code

Simple Assemble procedure

Problem: variables can be instantiated before `operator()()` is executed:

```
template<class TProg, class ...Ts>
inline SPIRVModule SPIRVAssembler::AssembleSimple(Ts&& ..._args)
{
    InitializeProgram<TProg>(std::forward<Ts>(_args)...); // constructs TProg
    RecordInstructions<TProg>(); // calls operator()() of TProg

    if constexpr(std::is_base_of_v<SPIRVProgram<true>, TProg>)
        return Assemble();
    else
        return SPIRVModule(0);
}
```

Back to SPIR-V



Back to SPIR-V

Most SPIR-V operations produce a result-ID

- Consumed by other operations via intermediate operands
- IDs definition must precede its usage
- IDs need to be in SSA-Form

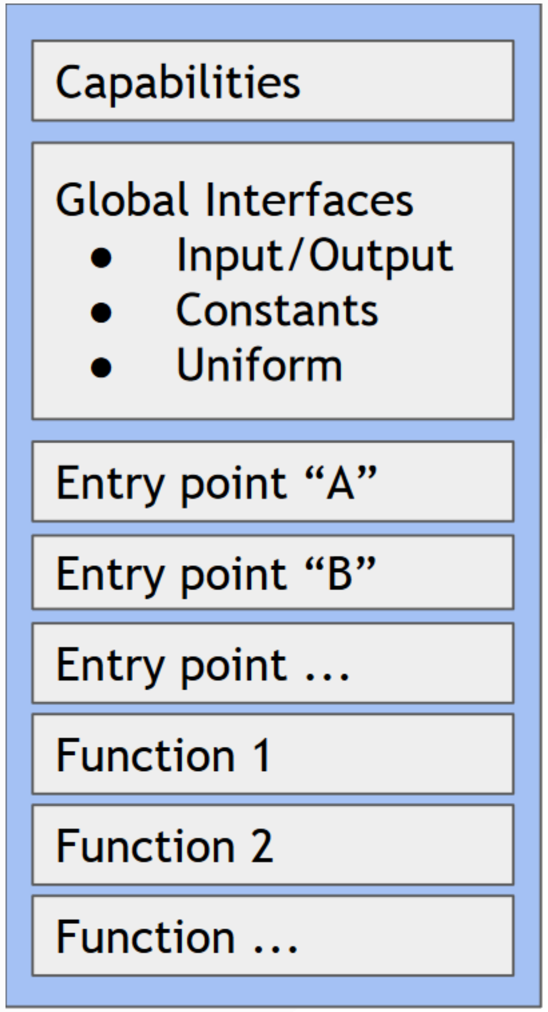
```
%58 = OpFOrdLessThan %bool %56 %57
%60 = OpLoad %v2float %53
%61 = OpDot %float %60 %60
%62 = OpLoad %float %59
%63 = OpFOrdLessThan %bool %61 %62
%64 = OpLogicalAnd %bool %63 %58
      OpBranchConditional %64 %65 %83
%65 = OpLabel
```

Types and Constants

- Types and Constants form hierarchies
- Their definitions should be unique
- They must be defined before any EntryPoint

```
      %float = OpTypeFloat 32
    %v4float = OpTypeVector %float 4
%_ptr_Output_v4float = OpTypePointer Output %v4float
```

```
%float_0 = OpConstant %float 0
     %12 = OpConstantComposite %v4float %float 0 %float 0 %float 0 %float 0
```



SPIR-V Types

```
std::vector<SPIRVType> m_SubTypes; // struct members etc
spv::Op m_kBaseType = spv::OpNop;
uint32_t m_uDimension = 0u; // OpTypeArray, dimension, bits
bool m_bSign = true;

// for image:
bool m_bArray = false;
bool m_bMultiSampled = false;
ETexDepthType m_kTexDepthType = kTexDepthType_Unspecified;
ETexSamplerAccess m_kSamplerAccess = kTexSamplerAccess_Runtime;
```

SPIR-V Types translate to operations

```
const spv::Op kType = _Type.GetType();
SPIRVOperation OpType(kType);

for (const SPIRVType& Type : _Type.GetSubTypes())
{
    SubTypes.push_back(AddType(Type));
}
```

```
// create operands
switch (kType)
{
case spv::OpTypeVoid:
case spv::OpTypeBool:
case spv::OpTypeSampler:
    break; // nothing to do
case spv::OpTypeInt:
    // bitwidth
    OpType.AddLiteral(_Type.GetDimension());
    OpType.AddLiteral(uint32_t(_Type.GetSign()));
    break;
case spv::OpTypeStruct:
    OpType.AddTypes(SubTypes);
    break;
…
```

Unique Types and Constants

Use Hash-Maps to store Types and Constants:

```
const uint32_t uInstrId = AddOperation(std::move(OpType));
m_TypeIds.insert({ uHash, uInstrId });
```

```
size_t uHash = hlx::Hash(
    m_kBaseType, m_uDimension, m_bSign, m_bArray,
    m_bMultiSampled, m_kTexDepthType, m_kSamplerAccess);

for (const SPIRVType& subtype : m_SubTypes)
{
    uHash = hlx::CombineHashes(uHash, subtype.GetHash());
}
```

Creating SPIR-V Types from C++ Types

Simple and primitive types can be trivially reflected:

```
Type = SPIRVType::FromType<T>();
uTypeId = GlobalAssembler.AddType(Type);
```

```
template<>
SPIRVType SPIRVType::FromBaseType<bool>() { return SPIRVType::Bool(); }

template<>
SPIRVType SPIRVType::FromBaseType<float>() {return SPIRVType::Primitive<float>();}

template<>
SPIRVType SPIRVType::FromBaseType<int32_t>() {return SPIRVType::Primitive<int32_t>();}
```

Creating SPIR-V Types from C++ Types

- Nested types require sophisticated C++ reflection capabilities
- C++17 does not have those yet:

```
struct PointLight
{
    SPVStruct  → typedef Tracy::TSPVStructTag SPVStructTag

    float3 vColor;
    f32 fRange;

    float3 vPosition;
    f32 fDecayStart;
};
```

Creating SPIR-V Types from C++ Types

→ First count the number of constructible toplevel entries

```
struct filler { template< typename type > operator type && (); };
template< typename aggregate, typename index_sequence = std::index_sequence<>,typename = void >
struct aggregate_arity : index_sequence{};

template< typename aggregate, std::size_t ...indices>
struct aggregate_arity < aggregate, std::index_sequence< indices... >,
std::void_t< decltype(aggregate{ (indices, std::declval< filler >())..., std::declval< filler >() })
> >: aggregate_arity< aggregate, std::index_sequence< indices..., sizeof...(indices) > >{};

template <class T>
constexpr size_t aggregate_arity =
aggregate_arity<std::remove_reference_t<std::remove_cv_t<T>>>::size();
```

Creating SPIR-V Types from C++ Types

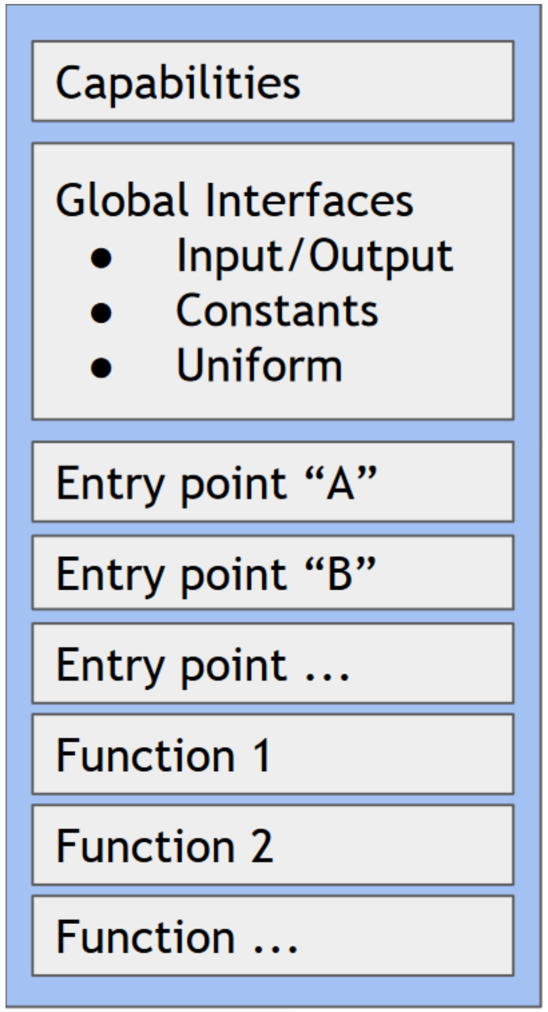
→ Then use that to index into a structured binding:

```
template< std::size_t index, typename type >
constexpr decltype(auto) get(type& value) noexcept {
    constexpr std::size_t arity = aggregate_arity<type>;

    if constexpr (arity == 2) {
        auto &[p0, p1] = value;
        if constexpr (index == 0) {
            return (p0);
        } else if constexpr (index == 1) {
            return (p1);
        } else {
            return;
        }
    }
}
```

Types and Constants

- Because new variables can be defined anywhere in the program we need to sort those operations above any EntryPoint
- The same is true for other classes of operations:
  - DebugInfo: Source, Name, MemberName, String...
  - Decorates: MemberDecorates, `DecorationRowMajor` etc
  - Preamble Capabilities, Extensions, MemoryModel, Execution Model, EntryPoint

→ Reoder pass needed after recording all operations



SPIR-V Operations

Intermediate representation for SPIR-V instructions:

```
spv::Op m_kOpCode = spv::OpNop;
uint32_t m_uInstrId = HUNDEFINED32;
uint32_t m_uResultId = HUNDEFINED32;
std::vector<SPIRVOperand> m_Operands;
uint32_t m_uResultTypeId = HUNDEFINED32;
bool m_bUsed = true;
bool m_bTranslated = false;
```

SPIR-V Operations

Recording instructions:

```
template<class ...Ts>
inline uint32_t SPIRVAssembler::EmplaceOperation(Ts&& ..._args)
{
    m_Operations.emplace_back(std::forward<Ts>(_args)...).m_uInstrId = m_uInstrId;
    return m_uInstrId++;
}
```

Creating instructions from operators

Back to operator overloading:

```
template <class T, bool Assemble, spv::StorageClass C1, spv::StorageClass C2>
inline var_t<T, Assemble, spv::StorageClassFunction>
operator+(const var_t<T, Assemble, C1>& l, const var_t<T, Assemble, C2>& r)
{
    return make_op2(l, r,
        [](const T& v1, const T& v2)-> T {return v1 + v2; },
        kOpTypeBase_Result, spv::OpFAdd, spv::OpIAdd);
}
```

Host computation

Instuction Variants

Creating instructions from operators

Finally create a temporary variable end emit SPIR-V instructions:

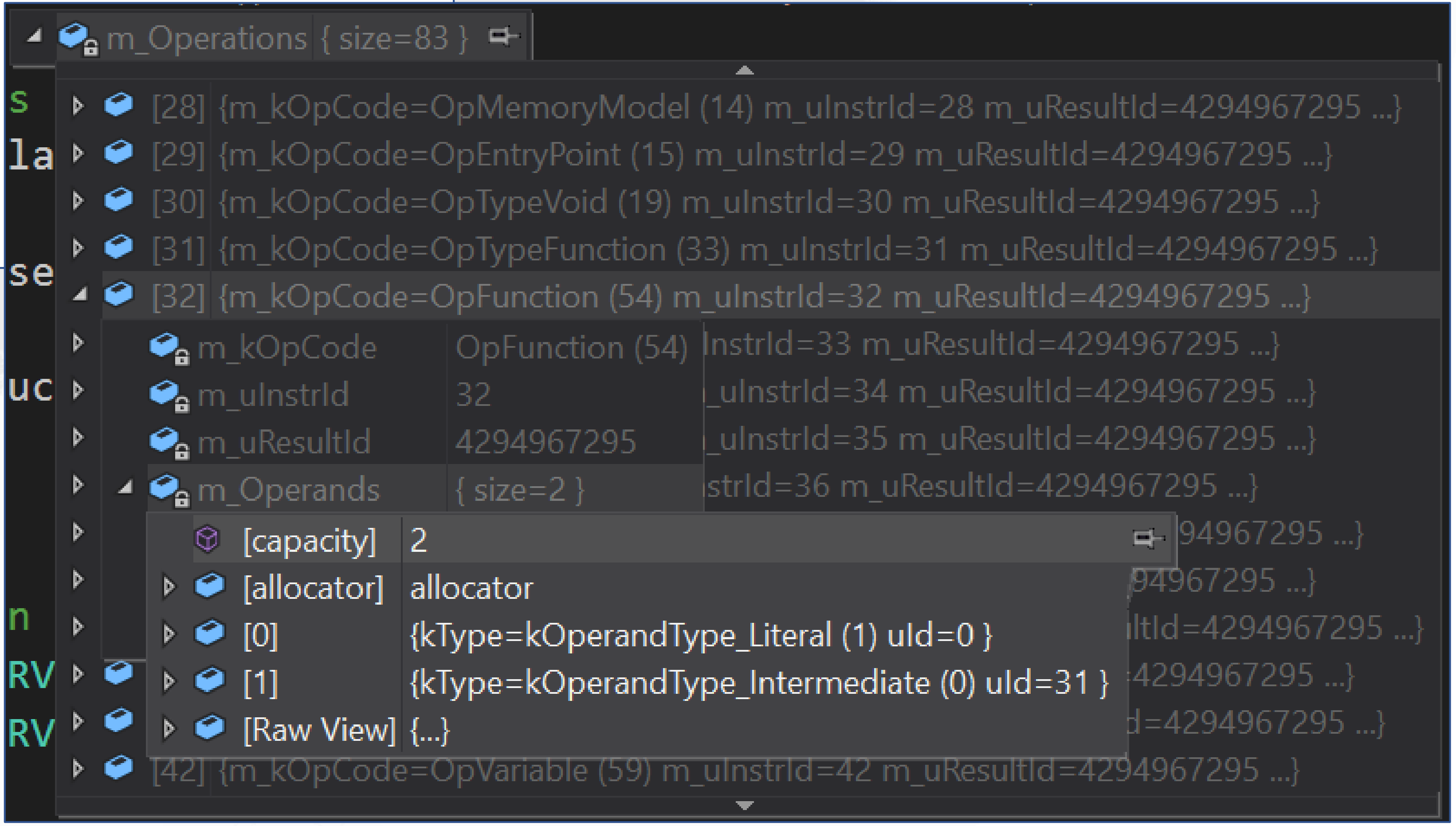
```
auto var = var_t<T, Assemble, spv::StorageClassFunction>(TIntermediate());

if constexpr(Assemble == false)
    var.Value = _OpFunc(l.Value, r.Value);
else {  // Assemble
    var.uResultId = GlobalAssembler.EmplaceOperation(
                OpTypeDeciderEx<T, U, V>(_kOpTypeBase, _Ops...),
                var.uTypeId, // result type
                SPIRVOperand::Intermediate(l.Load()), // operand1
                SPIRVOperand::Intermediate(r.Load())  // operand2
            );
}

return var;
```

Host computation (GLM)

```
var.uResultId = m_uInstId++;
```

Instruction recording

```
complex z(0.f, 0.f);
While(z.Conjugate() < 4.f && i < max)
{
  z = z * z + c;
  ++i;
});
```



```
◢ m_Operations { size=83 }
  s   ▷ [28] {m_kOpCode=OpMemoryModel (14) m_uInstrId=28 m_uResultId=4294967295 ...}
  la  ▷ [29] {m_kOpCode=OpEntryPoint (15) m_uInstrId=29 m_uResultId=4294967295 ...}
      ▷ [30] {m_kOpCode=OpTypeVoid (19) m_uInstrId=30 m_uResultId=4294967295 ...}
      ▷ [31] {m_kOpCode=OpTypeFunction (33) m_uInstrId=31 m_uResultId=4294967295 ...}
  se  ◢ [32] {m_kOpCode=OpFunction (54) m_uInstrId=32 m_uResultId=4294967295 ...}
      ▷     m_kOpCode     OpFunction (54)   InstrId=33 m_uResultId=4294967295 ...}
  uc  ▷     m_uInstrId    32                _uInstrId=34 m_uResultId=4294967295 ...}
      ▷     m_uResultId   4294967295        _uInstrId=35 m_uResultId=4294967295 ...}
      ▷   ◢ m_Operands    { size=2 }        strId=36 m_uResultId=4294967295 ...}
      ▷         [capacity]    2                                                   94967295 ...}
      ▷       ▷ [allocator]   allocator                                           94967295 ...}
  n   ▷       ▷ [0]           {kType=kOperandType_Literal (1) uId=0 }             ltId=4294967295 ...}
  RV  ▷       ▷ [1]           {kType=kOperandType_Intermediate (0) uId=31 }       4294967295 ...}
  RV  ▷       ▷ [Raw View]    {...}                                               d=4294967295 ...}
      ▷ [42] {m_kOpCode=OpVariable (59) m_uInstrId=42 m_uResultId=4294967295 ...}
```

SPEAR Framework

SPIRVProgram
- EntryPoint
- Execution Mode & Model
- Extensions
- Capabilities

SPIRVAssembler
- Map<Hash, ID> TypeIDs
- Map<Hash, ID> ConstantIDs
- Vector<SPIRVOperation> Ops

SPIRVVariable
- VarID & TypeID
- ResultID & LastStoredID
- Name & StorageClass

SPIRVModule
- Vector<VarInfo> (Type, Name, Location, Class...)
- Vector<uint32_t> Instructions

SPIRVAssembler Resolve

Two step resolve and translate pass:

- Assign unassigned result IDs to operations (not to confuse with InstrIDs) depending on the instruction category (preamble etc)
- Translate operands (lookup InstrID and get corresponding resultID) and generate the uint32_t words for the opcode and resolved operands
  - Mark the SPIRVOperation as translated

→ Store the data in simple SPIRVInstruction class (semi-final instruction stream)

SPIRVAssembler Resolve

Assign result ID:

```
void SPIRVAssembler::AssignId(SPIRVOperation& _Op)
{
    uint32_t uResultId = SPIRVInstruction::kInvalidId;

    if (_Op.m_bUsed) // dont resolve unused ops
    {
        if (CreatesResultId(_Op.GetOpCode()))
            uResultId = m_uResultId++;
    }

    _Op.m_uResultId = uResultId;
}
```

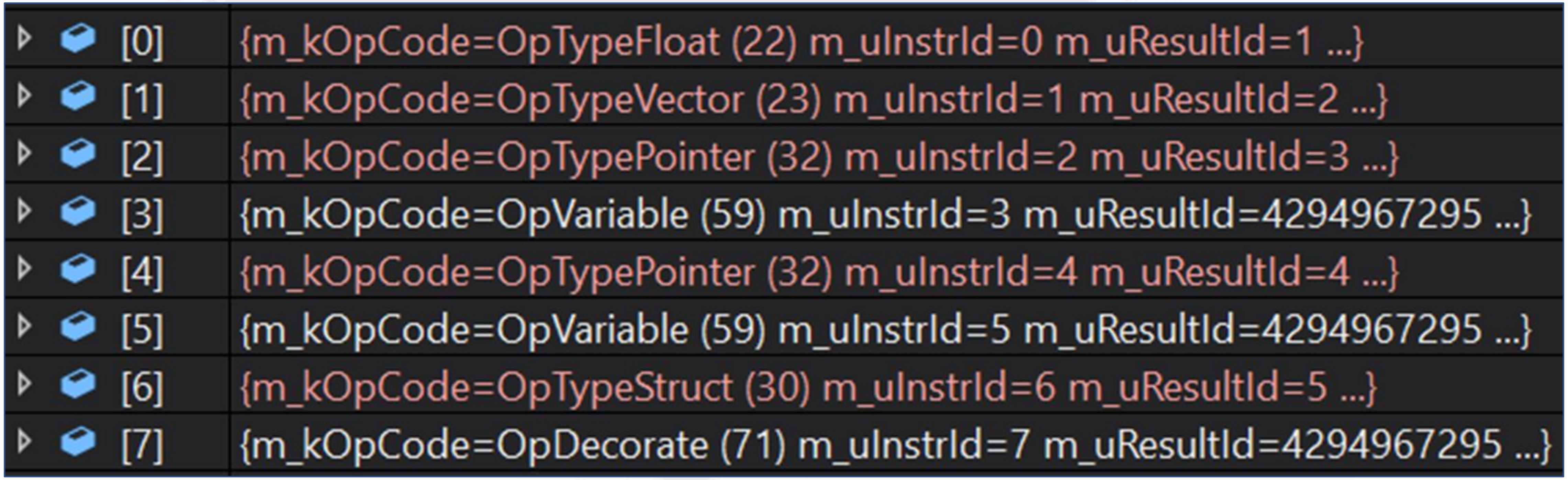
SPIRVAssembler Resolve & Reorder

Assign result IDs to match with preamble ordering:

```
ForEachOp([this](SPIRVOperation& Op)
{
    AssignId(Op);
}, is_type_or_const_op);

ForEachOp([this](SPIRVOperation& Op)
{
    AssignId(Op);
}, is_decorate_op);

ForEachOpEx([this](SPIRVOperation& Op)
{
    AssignId(Op);
}, [](const SPIRVOperation& Op) {return Op.m_uResultId == HUNDEFINED32; });

…
```

SPIRVAssembler Resolve & Reorder

Assign result IDs to match with preamble ordering:



```
[0]   {m_kOpCode=OpTypeFloat (22) m_uInstrId=0 m_uResultId=1 ...}
[1]   {m_kOpCode=OpTypeVector (23) m_uInstrId=1 m_uResultId=2 ...}
[2]   {m_kOpCode=OpTypePointer (32) m_uInstrId=2 m_uResultId=3 ...}
[3]   {m_kOpCode=OpVariable (59) m_uInstrId=3 m_uResultId=4294967295 ...}
[4]   {m_kOpCode=OpTypePointer (32) m_uInstrId=4 m_uResultId=4 ...}
[5]   {m_kOpCode=OpVariable (59) m_uInstrId=5 m_uResultId=4294967295 ...}
[6]   {m_kOpCode=OpTypeStruct (30) m_uInstrId=6 m_uResultId=5 ...}
[7]   {m_kOpCode=OpDecorate (71) m_uInstrId=7 m_uResultId=4294967295 ...}
```

SPIRVAssembler Resolve

Translate SPIRVOperation to SPIRVInstruction:

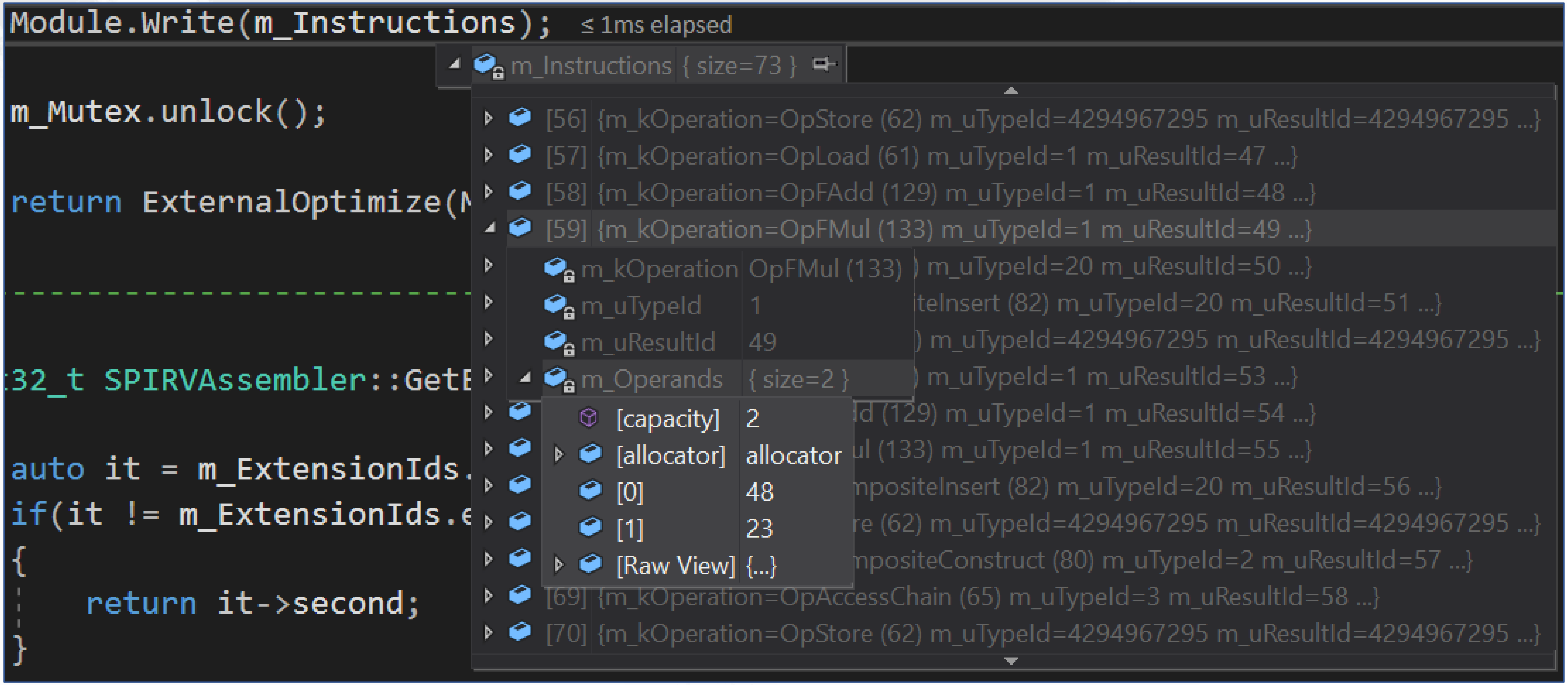
```
SPIRVInstruction SPIRVAssembler::Translate(SPIRVOperation& _Op) const {
    std::vector<uint32_t> Operands; uint32_t uTypeId = SPIRVInstruction::kInvalidId;

    if (_Op.GetResultType() != SPIRVInstruction::kInvalidId)
        uTypeId = m_Operations[_Op.GetResultType()].m_uResultId;

    for (const SPIRVOperand& Operand : _Op.GetOperands()){
        if (Operand.kType == kOperandType_Intermediate)
            Operands.push_back(m_Operations[Operand.uId].m_uResultId);
        else if (Operand.kType == kOperandType_Literal)
            Operands.push_back(Operand.uId);
    }

    _Op.m_bTranslated = true;
    return SPIRVInstruction(_Op.GetOpCode(), uTypeId, _Op.m_uResultId, Operands);
}
```

SPIRVAssembler Resolve & Reorder

Translate operations with final result IDs:

```
ForEachOp([TranslateOp](SPIRVOperation& Op) {TranslateOp(Op);}, is_name_op);
ForEachOp([TranslateOp](SPIRVOperation& Op) {TranslateOp(Op);}, is_decorate_op);
ForEachOp([TranslateOp](SPIRVOperation& Op) {TranslateOp(Op);}, is_type_or_const_op);

const auto TranslateOp = [](SPIRVOperation& _Op)
{
    if (_Op.GetUsed() && _Op.GetTranslated() == false)
    {
        AddInstruction(Translate(_Op));
    }
};
```

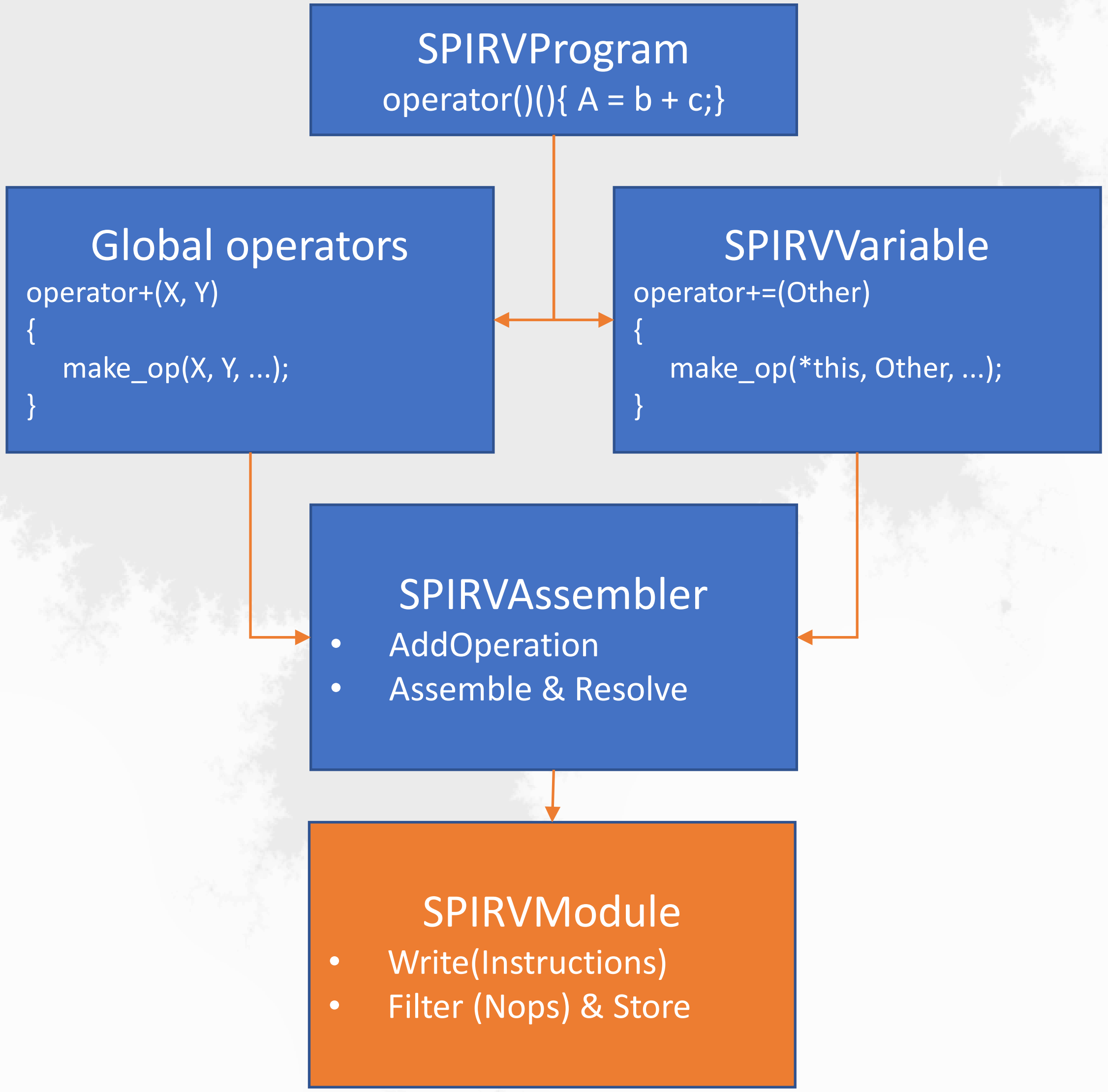
SPIRVAssembler Resolve & Reorder



```
Module.Write(m_Instructions);  ≤ 1ms elapsed

m_Mutex.unlock();

return ExternalOptimize(M

:32_t SPIRVAssembler::GetE

auto it = m_ExtensionIds.
if(it != m_ExtensionIds.e
{
    return it->second;
}
```

```
m_Instructions { size=73 }
[56] {m_kOperation=OpStore (62) m_uTypeId=4294967295 m_uResultId=4294967295 ...}
[57] {m_kOperation=OpLoad (61) m_uTypeId=1 m_uResultId=47 ...}
[58] {m_kOperation=OpFAdd (129) m_uTypeId=1 m_uResultId=48 ...}
[59] {m_kOperation=OpFMul (133) m_uTypeId=1 m_uResultId=49 ...}
    m_kOperation   OpFMul (133)   ) m_uTypeId=20 m_uResultId=50 ...}
    m_uTypeId      1              iteInsert (82) m_uTypeId=20 m_uResultId=51 ...}
    m_uResultId    49             ) m_uTypeId=4294967295 m_uResultId=4294967295 ...}
    m_Operands     { size=2 }     ) m_uTypeId=1 m_uResultId=53 ...}
        [capacity]   2            dd (129) m_uTypeId=1 m_uResultId=54 ...}
        [allocator]  allocator    ul (133) m_uTypeId=1 m_uResultId=55 ...}
        [0]          48           npositeInsert (82) m_uTypeId=20 m_uResultId=56 ...}
        [1]          23           re (62) m_uTypeId=4294967295 m_uResultId=4294967295 ...}
        [Raw View]   {...}        npositeConstruct (80) m_uTypeId=2 m_uResultId=57 ...}
[69] {m_kOperation=OpAccessChain (65) m_uTypeId=3 m_uResultId=58 ...}
[70] {m_kOperation=OpStore (62) m_uTypeId=4294967295 m_uResultId=4294967295 ...}
```

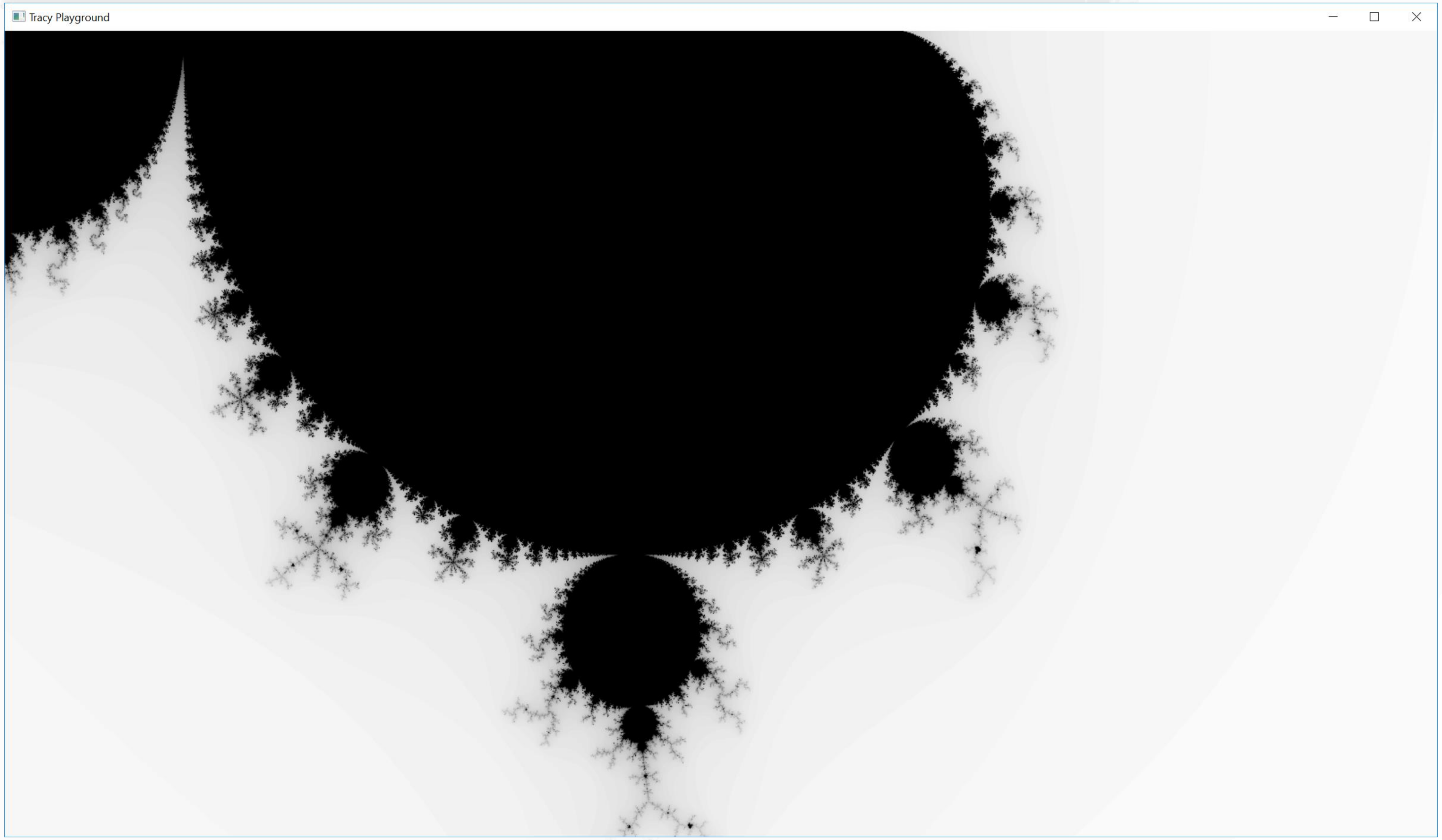
SPEAR Framework



Write the binary module

```
// write header
Put(spv::MagicNumber);
Put(m_uSPVVersion);
Put(uGenerator); // tracy
Put(m_uBounds); // Bounds
Put(uSchema);

// write instructions
for (const SPIRVInstruction& Instr : _Instructions) {
    Put(Instr.GetOpCode());

    if(Instr.GetTypeId() != SPIRVInstruction::kInvalidId)
        Put(Instr.GetTypeId());

    if (Instr.GetResultId() != SPIRVInstruction::kInvalidId)
        Put(Instr.GetResultId());

    for (const uint32_t& uOperand : Instr.GetOperands())
        Put(uOperand);
}
```

Done: C++ to SPIR-V in 50 Slides and 10k LoC



ControlFlow

- Use lambda function to capture expressions
- Condition lambda must return a var_t<bool,...>
- In Execution mode: just call function operator directly
- In Assemble mode: emit branching instructions

```
WhileFunc([=](){return z.Conjugate() < 4.f && i < max; }, [=]()
{
     z = z * z + c;
    ++i;
});
```

```
#define While(_cond) WhileFunc([ExprCaptureRule](){return _cond;}, [ExprCaptureRule]()
```

ControlFlow

In Execution mode just call function operator directly:

```
if constexpr(VarT::AssembleMode == false)
{
    while (_CondFunc().Value)
    {
        _LoopBody();
    }
}
```

ControlFlow

In Assemble mode emit branching instructions:

Condition lambda

Loop body lambda

```
// merge branch label
SPIRVOperation* pOpBranch = nullptr;
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranch), &pOpBranch); // close previous
block
const uint32_t uLoopMergeId = GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel));
pOpBranch->AddIntermediate(uLoopMergeId);

// loop merge
SPIRVOperation* pOpLoopMerge = nullptr;
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLoopMerge), &pOpLoopMerge);

// condition branch label
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranch), &pOpBranch);
const uint32_t uConditionLabelId =
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel));
pOpBranch->AddIntermediate(uConditionLabelId);

// tranlate condition var
GlobalAssembler.EnterScope();
const auto CondVar = _CondFunc();
GlobalAssembler.LeaveScope();

// branch conditional %cond %loopbody %exit
SPIRVOperation* pOpBranchCond = nullptr;
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranchConditional), &pOpBranchCond);
const uint32_t uLoopBodyId = GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel));
pOpBranchCond->AddIntermediate(CondVar.Load());
pOpBranchCond->AddIntermediate(uLoopBodyId);

GlobalAssembler.EnterScope();
_LoopBody();
GlobalAssembler.LeaveScope();

// close block
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranch), &pOpBranch);
const uint32_t uBlockExit = GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel));
pOpBranch->AddIntermediate(uBlockExit);

// exit branch label
GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranch), &pOpBranch);
const uint32_t uExitId = GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel));
pOpBranch->AddIntermediate(uLoopMergeId);

pOpLoopMerge->AddIntermediate(uExitId); // merge block
pOpLoopMerge->AddIntermediate(uBlockExit); // continue
pOpLoopMerge->AddLiteral((uint32_t)_kLoopControl);

pOpBranchCond->AddIntermediate(uExitId); // structured merge
```

ControlFlow

Code generated by conditon

Code generated by loop body

```
%46 = OpLabel
      OpLoopMerge %73 %72 None
      OpBranch %47
%47 = OpLabel
%48 = OpLoad %float %31
%49 = OpLoad %float %32
%50 = OpFOrdLessThan %bool %48 %49
%51 = OpLoad %v2float %45
%52 = OpDot %float %51 %51
%53 = OpFOrdLessThan %bool %52 %float_4
%54 = OpLogicalAnd %bool %53 %50
      OpBranchConditional %54 %55 %73
%55 = OpLabel
%57 = OpLoad %v2float %45
%58 = OpCompositeExtract %float %57 0
%59 = OpCompositeExtract %float %57 1
%60 = OpFMul %float %59 %59
%61 = OpFMul %float %58 %58
%62 = OpFSub %float %61 %60
%63 = OpLoad %v2float %56
%64 = OpCompositeInsert %v2float %62 %63 0
      OpStore %56 %64
%65 = OpFMul %float %59 %58
%66 = OpFMul %float %58 %59
%67 = OpFAdd %float %66 %65
%68 = OpCompositeInsert %v2float %67 %64 1
      OpStore %56 %68
%69 = OpFAdd %v2float %68 %44
      OpStore %45 %69
      OpStore %56 %68
%70 = OpLoad %float %31
%71 = OpFAdd %float %70 %float_1
      OpStore %31 %71
      OpBranch %72
%72 = OpLabel
      OpBranch %46
%73 = OpLabel
```

ControlFlow

```
%46 = OpLabel
      OpLoopMerge %73 %72 None
      OpBranch %47
%47 = OpLabel
%48 = OpLoad %float %31
%49 = OpLoad %float %32
%50 = OpFOrdLessThan %bool %48 %49
%51 = OpLoad %v2float %45
%52 = OpDot %float %51 %51
%53 = OpFOrdLessThan %bool %52 %float_4
%54 = OpLogicalAnd %bool %53 %50
      OpBranchConditional %54 %55 %73
%55 = OpLabel
%57 = OpLoad %v2float %45
%58 = OpCompositeExtract %float %57 0
%59 = OpCompositeExtract %float %57 1
%60 = OpFMul %float %59 %59
%61 = OpFMul %float %58 %58
%62 = OpFSub %float %61 %60
%63 = OpLoad %v2float %56
%64 = OpCompositeInsert %v2float %62 %63 0
      OpStore %56 %64
%65 = OpFMul %float %59 %58
%66 = OpFMul %float %58 %59
%67 = OpFAdd %float %66 %65
%68 = OpCompositeInsert %v2float %67 %64 1
      OpStore %56 %68
%69 = OpFAdd %v2float %68 %44
      OpStore %45 %69
      OpStore %56 %68
%70 = OpLoad %float %31
%71 = OpFAdd %float %70 %float_1
      OpStore %31 %71
      OpBranch %72
%72 = OpLabel
      OpBranch %46
%73 = OpLabel
```

ControlFlow

```
%46 = OpLabel
      OpLoopMerge %73 %72 None
      OpBranch %47
%47 = OpLabel
%48 = OpLoad %float %31
%49 = OpLoad %float %32
%50 = OpFOrdLessThan %bool %48 %49
%51 = OpLoad %v2float %45
%52 = OpDot %float %51 %51
%53 = OpFOrdLessThan %bool %52 %float_4
%54 = OpLogicalAnd %bool %53 %50
      OpBranchConditional %54 %55 %73
%55 = OpLabel
%57 = OpLoad %v2float %45
%58 = OpCompositeExtract %float %57 0
%59 = OpCompositeExtract %float %57 1
%60 = OpFMul %float %59 %59
%61 = OpFMul %float %58 %58
%62 = OpFSub %float %61 %60
%63 = OpLoad %v2float %56
%64 = OpCompositeInsert %v2float %62 %63 0
      OpStore %56 %64
%65 = OpFMul %float %59 %58
%66 = OpFMul %float %58 %59
%67 = OpFAdd %float %66 %65
%68 = OpCompositeInsert %v2float %67 %64 1
      OpStore %56 %68
%69 = OpFAdd %v2float %68 %44
      OpStore %45 %69
      OpStore %56 %68
%70 = OpLoad %float %31
%71 = OpFAdd %float %70 %float_1
      OpStore %31 %71
      OpBranch %72
%72 = OpLabel
      OpBranch %46
%73 = OpLabel
```

Vendor Instruction Extensions

```
enum EGCNShader {kGCNShader_TimeAMD = 3, ...};

template <bool Assemble, class CPUClockType = std::chrono::system_clock>
inline var_t<int64_t, Assemble, spv::StorageClassFunction> Time()
{
    return make_ext_op0<Assemble>([]() -> int64_t {
        return static_cast<int64_t>(CPUClockType::now().time_since_epoch().count());
    }, "SPV_AMD_gcn_shader", kGCNShader_TimeAMD);
}
```

```
auto seededRND = SomeRNDFunc(ExtAMD::GCNShader::Time());
```

Variadic Template Arguments

```
// variadic min
template <class T, bool Assemble, spv::StorageClass C1, spv::StorageClass C2, class ...Ts>
inline var_t<T, Assemble, spv::StorageClassFunction> Min(
const var_t<T, Assemble, C1>& X, const var_t<T, Assemble, C2>& Y, const Ts& ..._Args)
{
    return Min(Min(X, Y), _Args...);
}
```

```
auto min = Min(fD, fD2, fDmax);
```

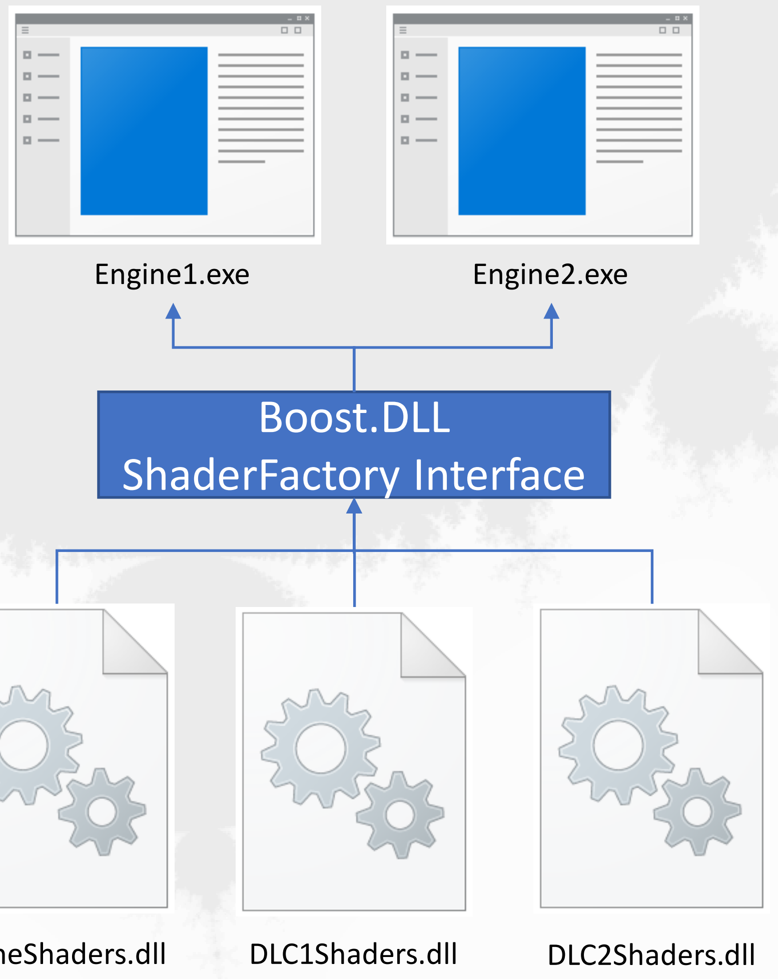
Constexpr If

```
if constexpr(std::is_same_v<Light, DirectionalLight>)
    vL = -_Light->vDirection;
else {
    vL = _Light->vPosition.xyz - _vPosWS;
    fPointToLightDist = Length(vL);
    vL = vL / fPointToLightDist;

    fAttenuation = CalculateAttenuation(fPointToLightDist,
                                              _Light->fRange, _Light->fDecayStart);

    if constexpr(std::is_same_v<Light, SpotLight>)
        fAttenuation *= CalculateSpotCone(_Light, vL);
}
```

Benefits of the SPEAR Framework

- Modern C++ features like templating & auto type deduction
- Polymorphism, software design patterns, modularity and reusability
- Adapting cutting edge GPU features using extension
- C++ Profiling and Debugging Tools
- Quite fast for compiling many shader permutations
  - Makes it possible to compile during runtime for small shaders

Benefits of the SPEAR Framework



Restrictions of the SPEAR Framework

- A LOT!
- Framework is a Proof-Of-Concept
- It’s a “Template-Hell”
- Nothing is unit tested... or sometimes even tested at all!
- I don’t have time to maintain it ☹

→ Please don’t use it in anything productive!

Restrictions of the SPEAR Framework

- Variable types are rather long outside the SPIRVProgram context
- Ugly macros for If, While, For etc…
- Mixing regular C++ variables and var_t<> has undesired sideeffects
- C++ variables are interpreted as constants
- Vector components can not be extracted by reference
- Recursion is not supported (will blow up instruction recording)

Restrictions of the SPEAR Framework

- Return statements will lead to dead code (not translated)
- Missing keywords switch, continue and break
- Ternary conditional `operator?` can not be overloaded in C++
  - Use Select function instead!

Personal Info

- @SiNGUL4RiTY
- f.wahlster@tum.de
- https://github.com/razor8