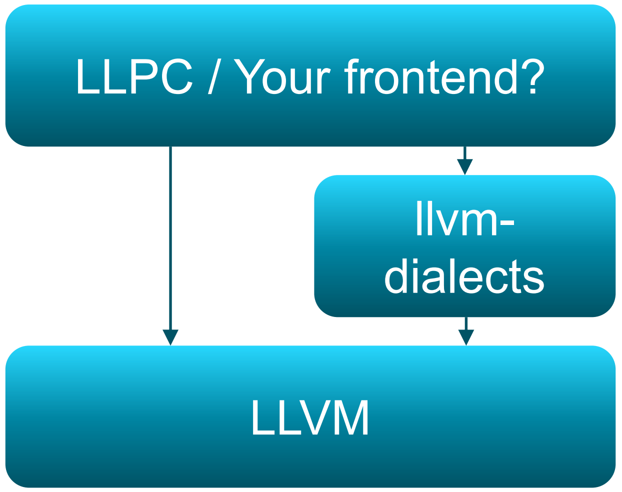
llvm-dialects: bringing dialects to the LLVM IR substrate

Nicolai Hähnle

together we advance_

What is llvm-dialects?

- Helper library (+ tablegen tool) that sits between LLVM and LLPC (or potentially other frontends)
- Enables defining “dialects” with some of the niceties provided by MLIR, but on the unmodified LLVM substrate
- Very new project and very much a work in progress
- LLVM-compatible licensing (Apache 2.0)
- https://github.com/GPUOpen-Drivers/llvm-dialects



AMD
together we advance_

Dialect

Instruction set
Types
Semantics

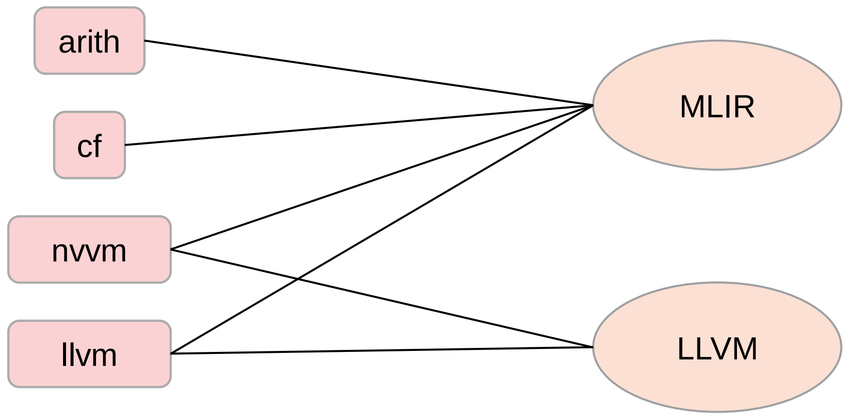
E.g.: LangRef.rst, mlir/Dialect.td

Substrate

Set of C++ classes to represent and manipulate code in one or more dialects

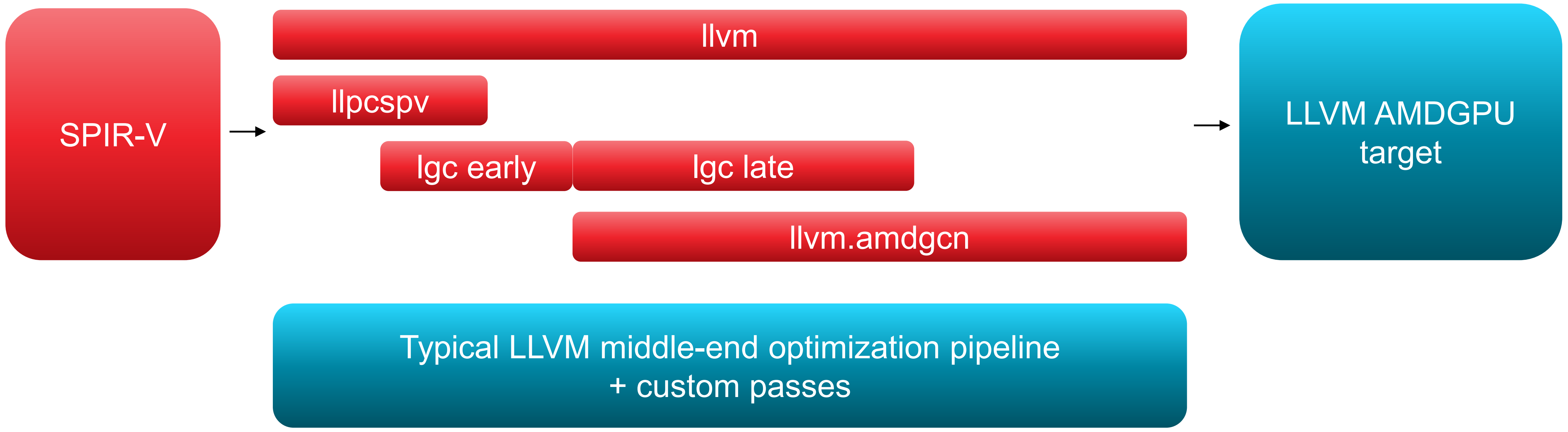
E.g.: llvm::Instruction, llvm::Value, llvm::BasicBlock, …

N:M relationship



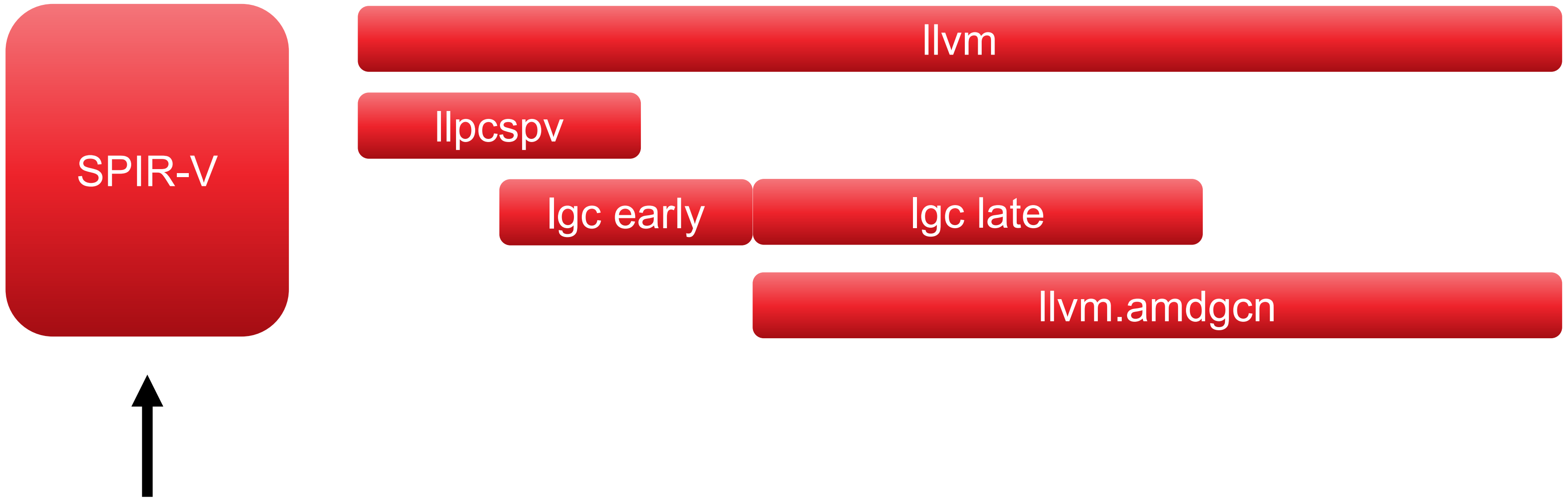
Problem Statement

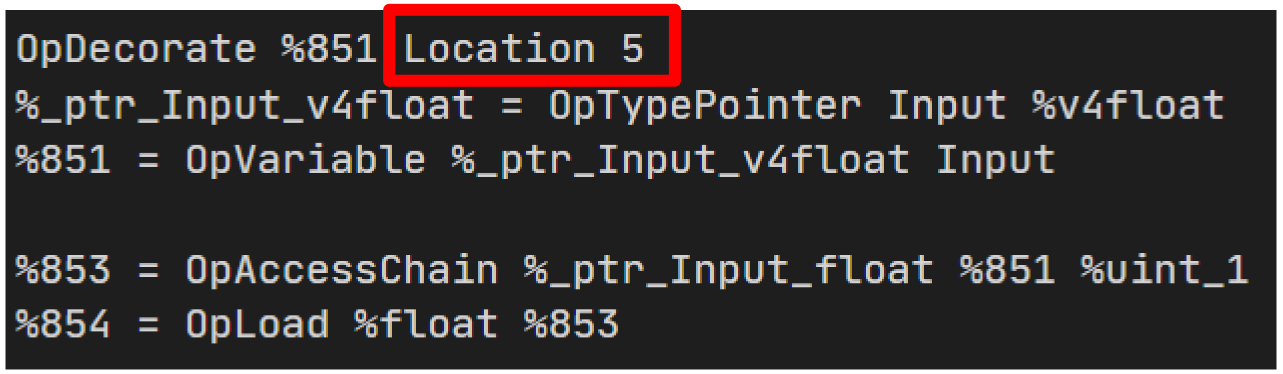
- LLPC compiles Vulkan SPIR-V “graphics” shaders (vertex, fragment, graphics-flavored compute, …)
- Uses LLVM IR augmented with “custom operations”



AMD
together we advance_

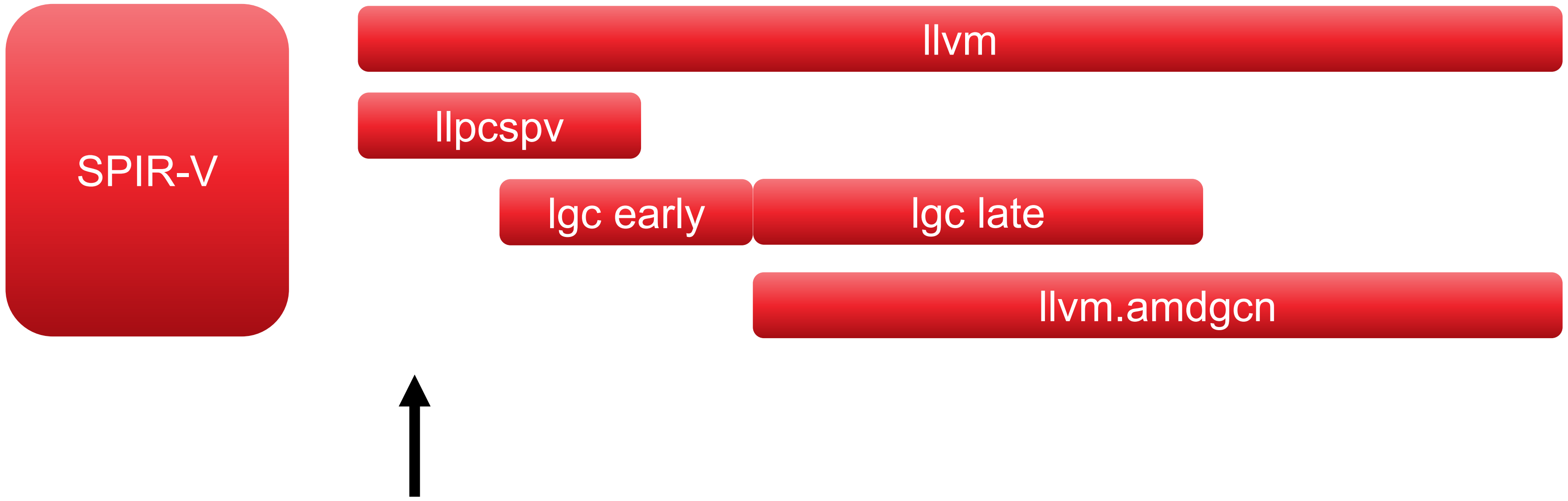
Example: Reading a Fragment Shader Input





```
OpDecorate %851 Location 5
%_ptr_Input_v4float = OpTypePointer Input %v4float
%851 = OpVariable %_ptr_Input_v4float Input

%853 = OpAccessChain %_ptr_Input_float %851 %uint_1
%854 = OpLoad %float %853
```

Example: Reading a Fragment Shader Input

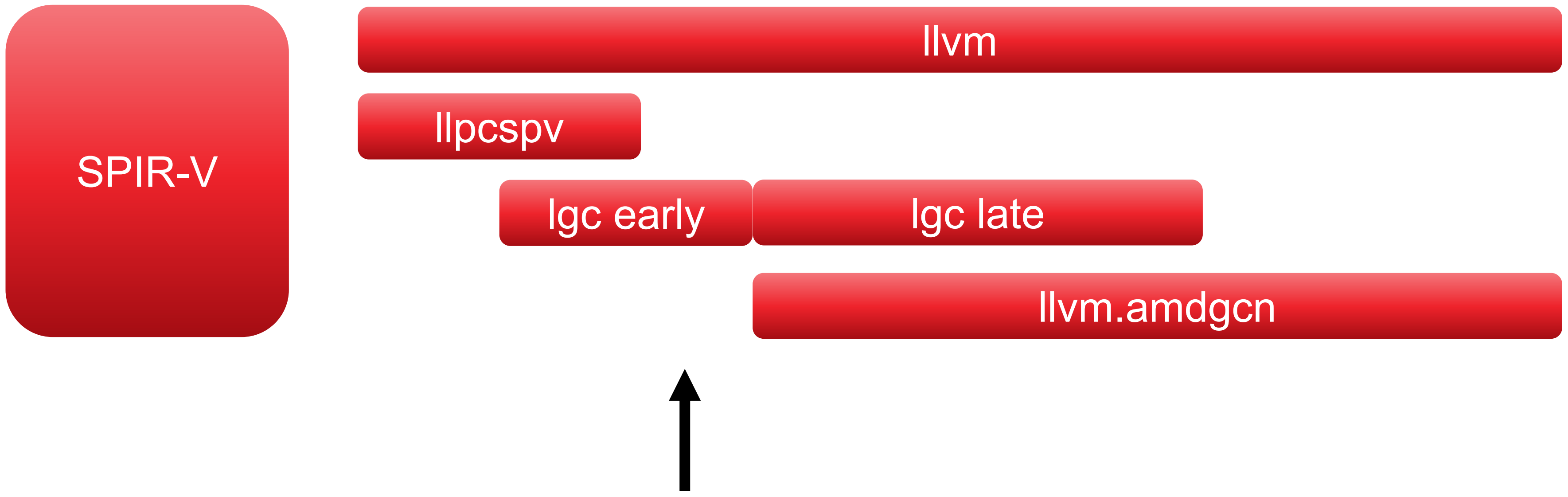


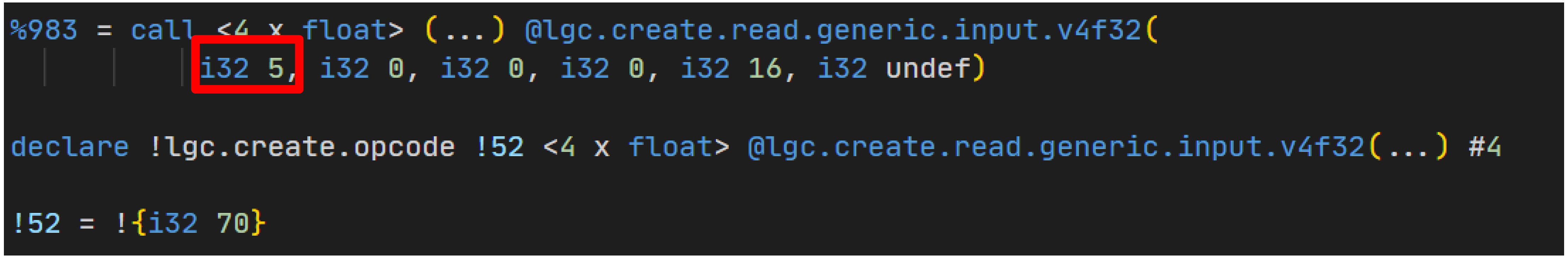
```
@1 = external addrspace(64) global <4 x float>, !spirv.InOut !2

%798 = load float, ptr addrspace(64) getelementptr
            (<4 x float>, ptr addrspace(64) @1, i32 0, i32 1), align 4

!2 = !{{ i64, i64 } { i64 16908293, i64 0 }}
```

0x1020005

AMD
together we advance_

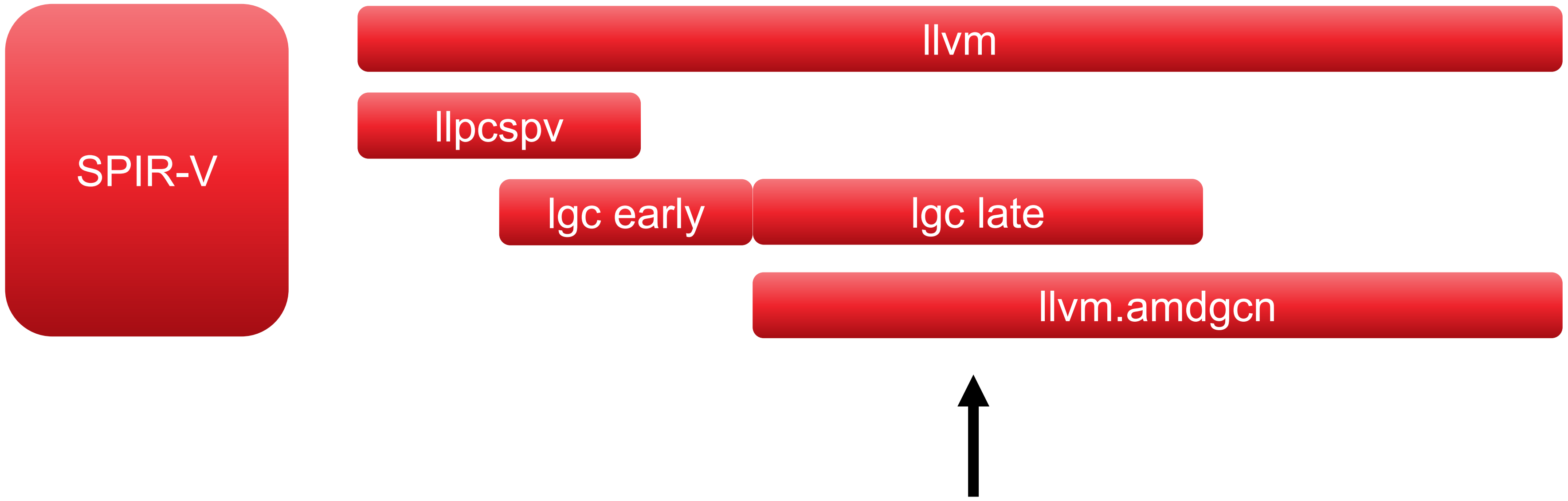
Example: Reading a Fragment Shader Input

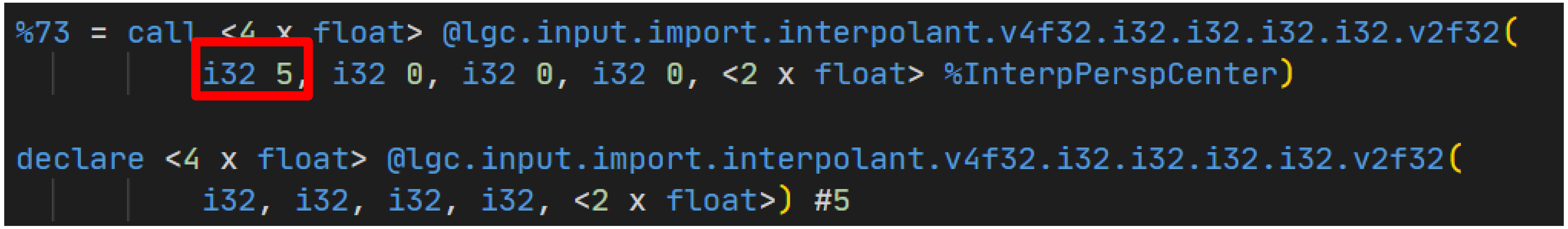




```
%983 = call <4 x float> (...) @lgc.create.read.generic.input.v4f32(
                 i32 5, i32 0, i32 0, i32 0, i32 16, i32 undef)

declare !lgc.create.opcode !52 <4 x float> @lgc.create.read.generic.input.v4f32(...) #4

!52 = !{i32 70}
```

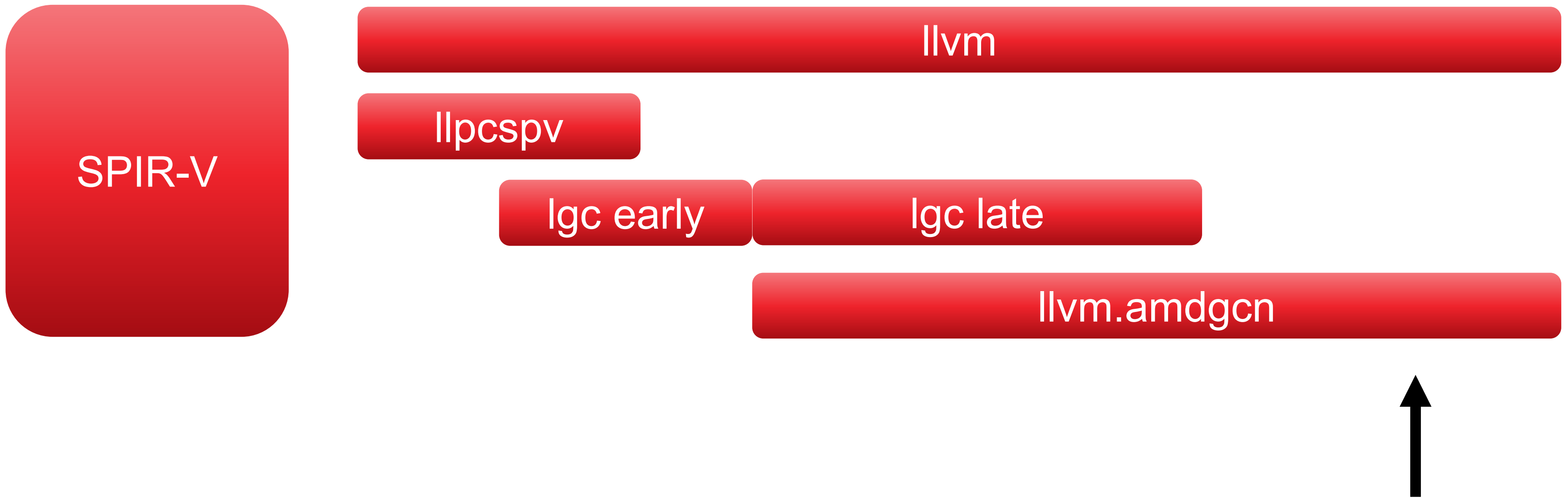
Example: Reading a Fragment Shader Input

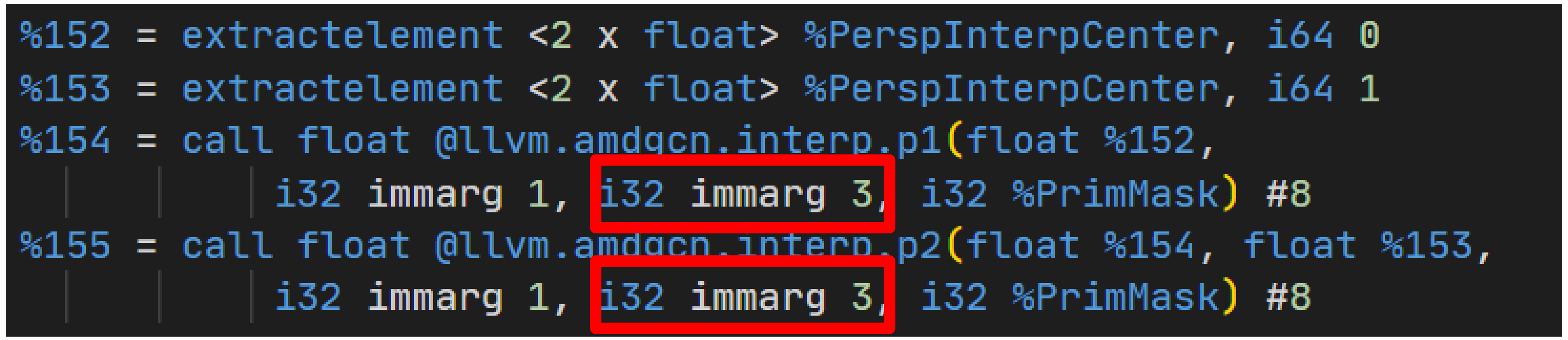




```
%73 = call <4 x float> @lgc.input.import.interpolant.v4f32.i32.i32.i32.i32.v2f32(
            i32 5, i32 0, i32 0, i32 0, <2 x float> %InterpPerspCenter)

declare <4 x float> @lgc.input.import.interpolant.v4f32.i32.i32.i32.i32.v2f32(
            i32, i32, i32, i32, <2 x float>) #5
```

Example: Reading a Fragment Shader Input





```
%152 = extractelement <2 x float> %PerspInterpCenter, i64 0
%153 = extractelement <2 x float> %PerspInterpCenter, i64 1
%154 = call float @llvm.amdgcn.interp.p1(float %152,
                i32 immarg 1, i32 immarg 3, i32 %PrimMask) #8
%155 = call float @llvm.amdgcn.interp.p2(float %154, float %153,
                i32 immarg 1, i32 immarg 3, i32 %PrimMask) #8
```

AMD
together we advance_

Working with Custom Operations

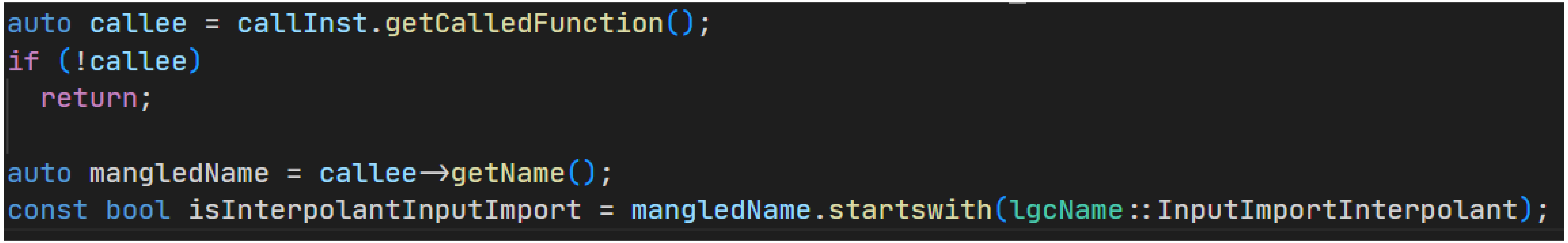
- Creating a custom operation

```
namespace lgcName {
const static char InputImportInterpolant[] = "lgc.input.import.interpolant.";
}
```

```
std::string callName = lgcName::InputImportInterpolant;
SmallVector<Value *, 5> args({
    getInt32(location),
    locationOffset,
    elemIdx,
    getInt32(InOutInfo::InterpModeCustom),
    vertexIndex,
});
addTypeMangling(resultTy, args, callName);
return CreateNamedCall(callName, resultTy, args, {Attribute::ReadOnly, Attribute::WillReturn});
```

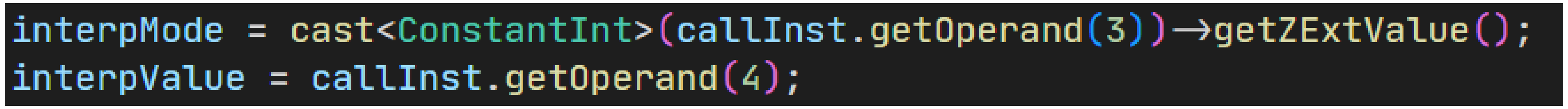
Working with Custom Operations, part 2

- Testing for a custom operation type



```
auto callee = callInst.getCalledFunction();
if (!callee)
  return;

auto mangledName = callee->getName();
const bool isInterpolantInputImport = mangledName.startswith(lgcName::InputImportInterpolant);
```

- Accessing operands



```
interpMode = cast<ConstantInt>(callInst.getOperand(3))->getZExtValue();
interpValue = callInst.getOperand(4);
```

Observations

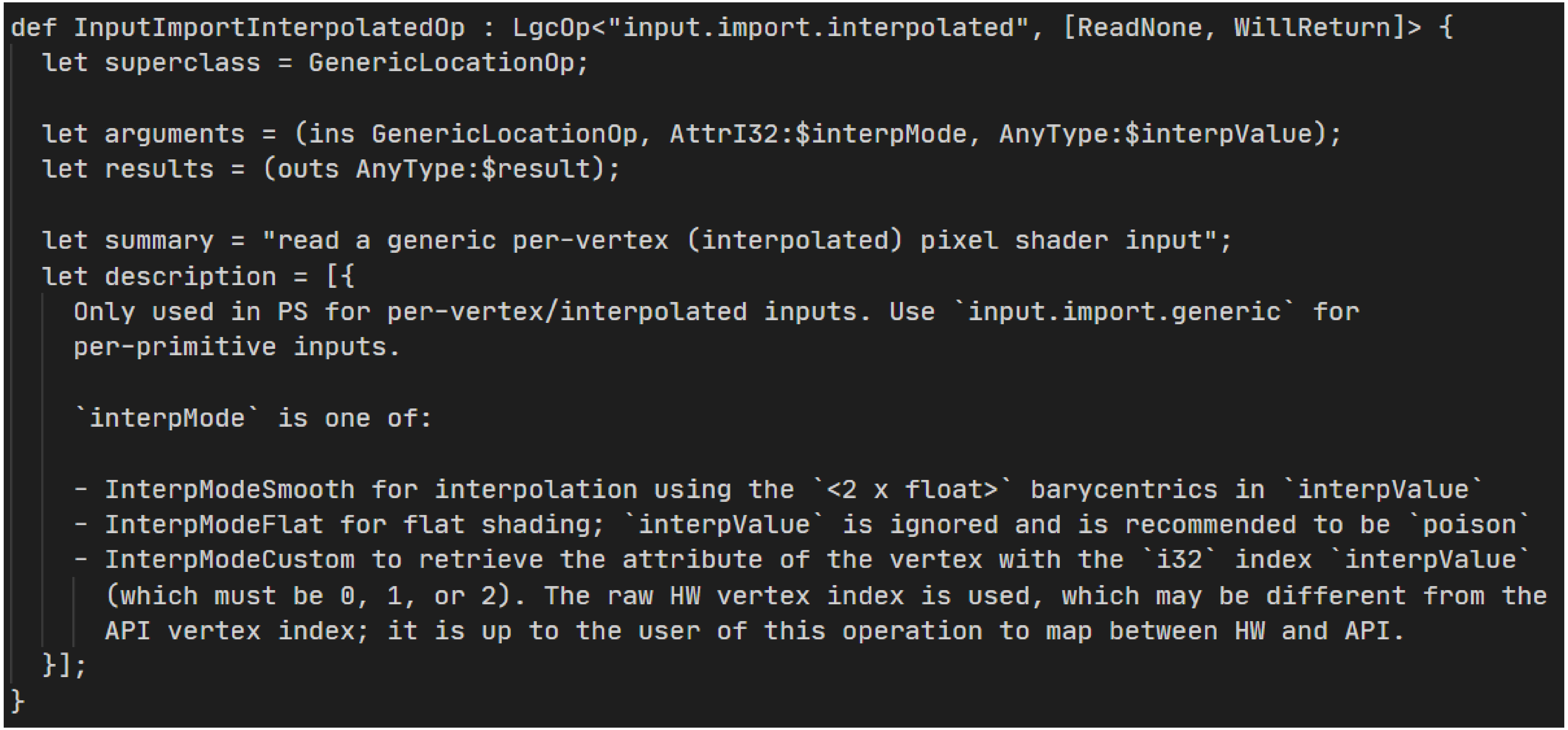
- Declare custom operations as functions that are never defined (never get a body)
- Largely indistinguishable from existing LLVM intrinsics in textual IR
- Also have custom address spaces and metadata
- Weirdness caused by compile-time concerns
  - Incomprehensible bit-packed values in metadata
  - Opcode via !lgc.create.opcode metadata in some cases; string comparison of function names in others
  - No option is truly competitive with IntrinsicID
- Would benefit from MLIR-style operation definitions and attributes, but the compiler pipeline is very much tied to the LLVM substrate

The Gap to MLIR in Detail

- Systematic way of defining custom instructions
- Usability of custom operations in C++
  - getOperand(magic_number) instead of named getters
  - isa<>/cast<>/dyn_cast<> unavailable
  - Hand-written visitors over and over again
- IR verifier integration
- Readability of textual IR
- Compile-time costs
- …

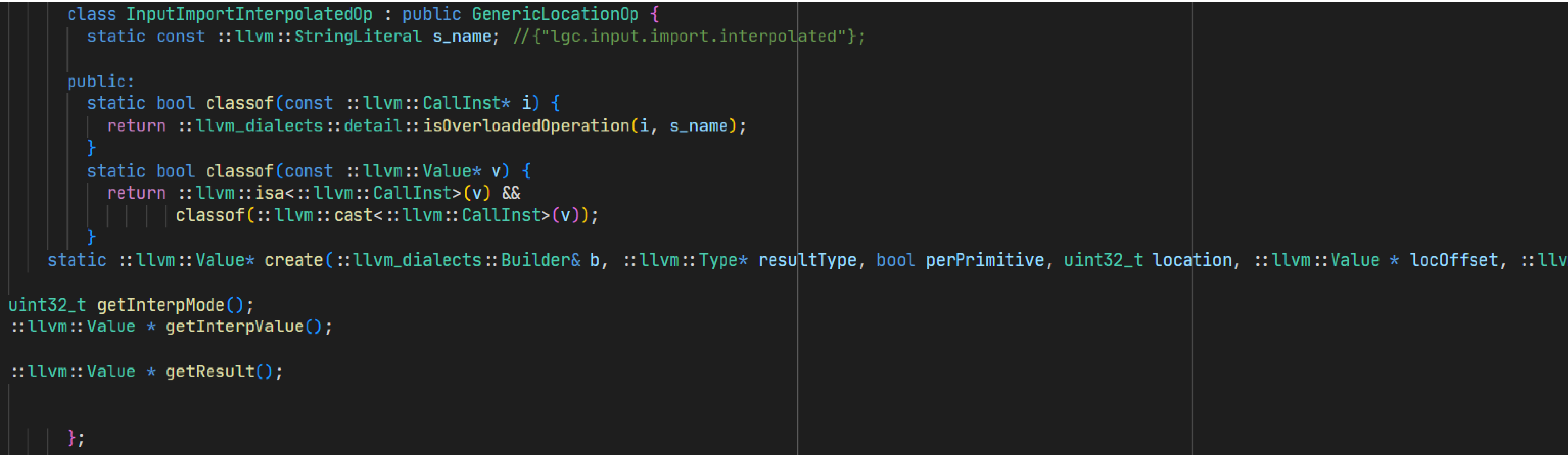
Address these first

llvm-dialects: Operation Definition

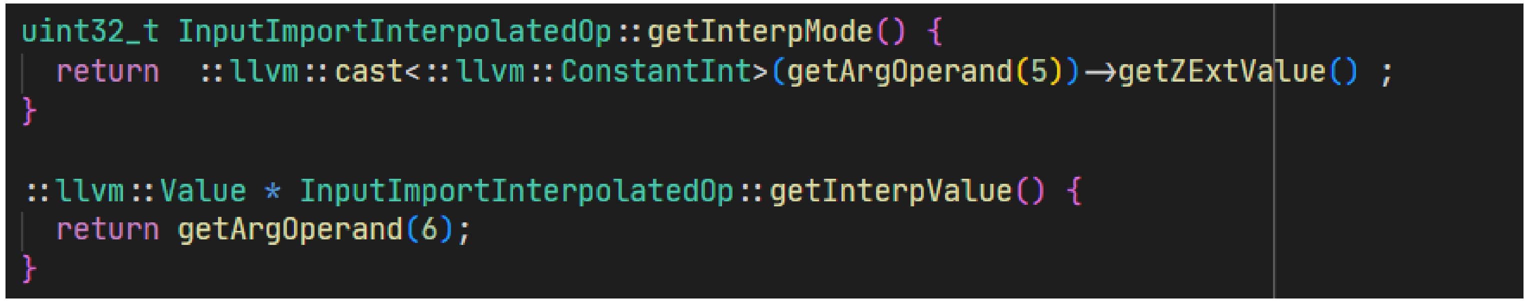


```
def InputImportInterpolatedOp : LgcOp<"input.import.interpolated", [ReadNone, WillReturn]> {
  let superclass = GenericLocationOp;

  let arguments = (ins GenericLocationOp, AttrI32:$interpMode, AnyType:$interpValue);
  let results = (outs AnyType:$result);

  let summary = "read a generic per-vertex (interpolated) pixel shader input";
  let description = [{
    Only used in PS for per-vertex/interpolated inputs. Use `input.import.generic` for
    per-primitive inputs.

    `interpMode` is one of:

    - InterpModeSmooth for interpolation using the `<2 x float>` barycentrics in `interpValue`
    - InterpModeFlat for flat shading; `interpValue` is ignored and is recommended to be `poison`
    - InterpModeCustom to retrieve the attribute of the vertex with the `i32` index `interpValue`
      (which must be 0, 1, or 2). The raw HW vertex index is used, which may be different from the
      API vertex index; it is up to the user of this operation to map between HW and API.
  }];
}
```

llvm-dialects-tblgen: Generated Code



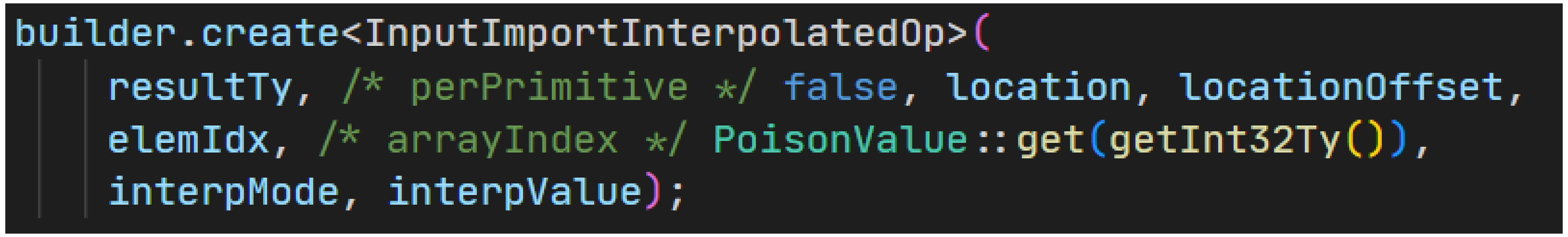
```
   class InputImportInterpolatedOp : public GenericLocationOp {
    static const ::llvm::StringLiteral s_name; //{"lgc.input.import.interpolated"};

   public:
    static bool classof(const ::llvm::CallInst* i) {
     return ::llvm_dialects::detail::isOverloadedOperation(i, s_name);
    }
    static bool classof(const ::llvm::Value* v) {
     return ::llvm::isa<::llvm::CallInst>(v) &&
         classof(::llvm::cast<::llvm::CallInst>(v));
    }
  static ::llvm::Value* create(::llvm_dialects::Builder& b, ::llvm::Type* resultType, bool perPrimitive, uint32_t location, ::llvm::Value * locOffset, ::llv

uint32_t getInterpMode();
::llvm::Value * getInterpValue();

::llvm::Value * getResult();


   };
```



```
uint32_t InputImportInterpolatedOp::getInterpMode() {
 return  ::llvm::cast<::llvm::ConstantInt>(getArgOperand(5))->getZExtValue() ;
}

::llvm::Value * InputImportInterpolatedOp::getInterpValue() {
 return getArgOperand(6);
}
```

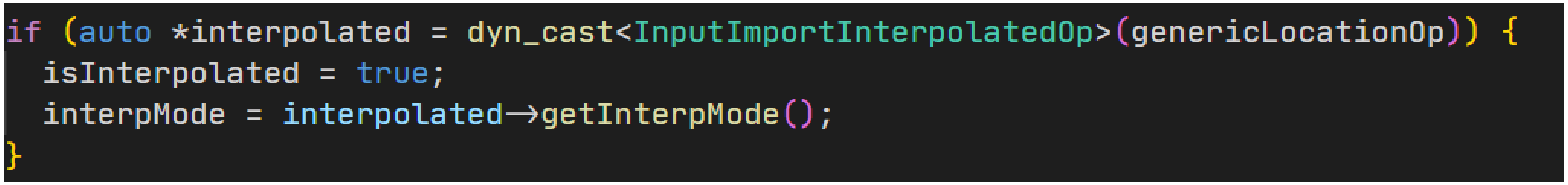
llvm-dialects: Working with Custom Operations

- Creating a custom operation via a generic method in our extended builder class



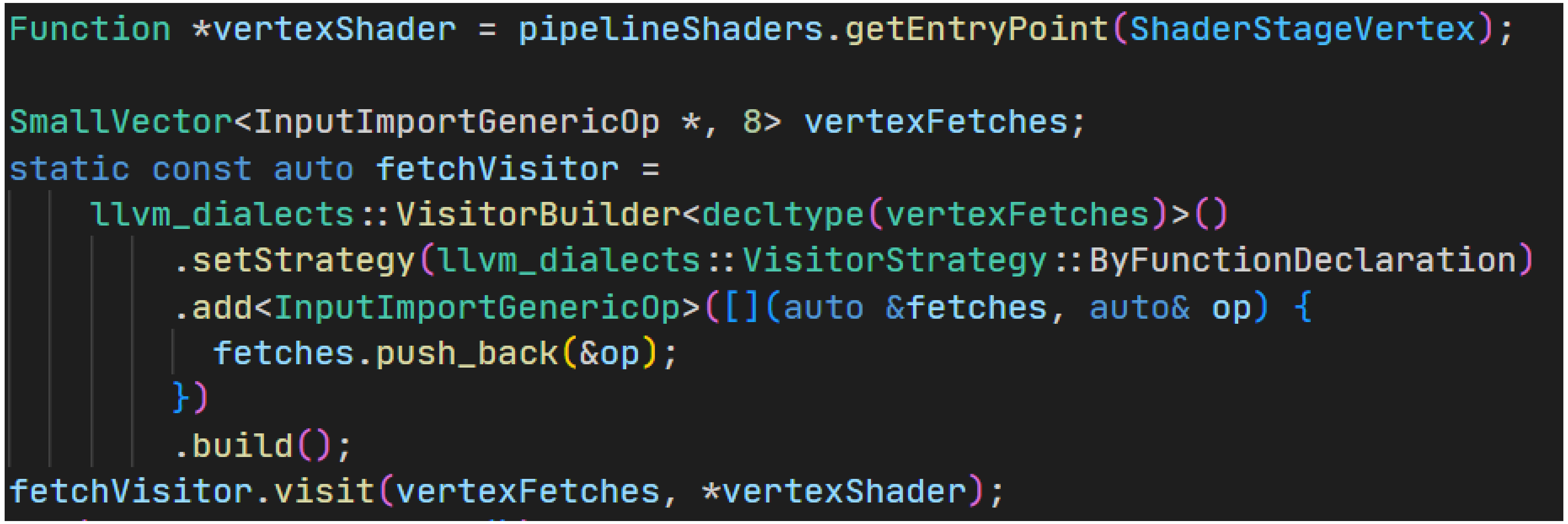
```
builder.create<InputImportInterpolatedOp>(
        resultTy, /* perPrimitive */ false, location, locationOffset,
        elemIdx, /* arrayIndex */ PoisonValue::get(getInt32Ty()),
        interpMode, interpValue);
```

- Testing for custom operation types
- Accessing operands



```
if (auto *interpolated = dyn_cast<InputImportInterpolatedOp>(genericLocationOp)) {
  isInterpolated = true;
  interpMode = interpolated->getInterpMode();
}
```

llvm-dialects: Visitor pattern



```
Function *vertexShader = pipelineShaders.getEntryPoint(ShaderStageVertex);

SmallVector<InputImportGenericOp *, 8> vertexFetches;
static const auto fetchVisitor =
    llvm_dialects::VisitorBuilder<decltype(vertexFetches)>()
        .setStrategy(llvm_dialects::VisitorStrategy::ByFunctionDeclaration)
        .add<InputImportGenericOp>([](auto &fetches, auto& op) {
          fetches.push_back(&op);
        })
        .build();
fetchVisitor.visit(vertexFetches, *vertexShader);
```

- Inspired by llvm::TypeSwitch, with two differences:
  - Visitor object is built once, allowing us to potentially amortize more expensive pre-computations
  - Iteration / visitation is integrated, allowing users to choose between standard iteration over basic blocks and instructions and iteration over the users of function declarations

Limitations of an External Library

- Textual IR
  - Custom operations will always look like function calls
  - Metadata will always be difficult read
- Compile-time cost of type testing
  - No dedicated opcode
  - Not even an IntrinsicID
  - Easiest and costliest option is to use string comparisons
  - Opcode as a function operand (like DXIL) is the fastest option but makes IR harder for humans to read and write
  - Metadata on function declaration (like !lgc.create.opcodes) seems like a reasonable trade-off at runtime, but inconvenient for serialization (opcode stability!)
- Deep changes to the substrate
  - Regions
  - Multiple function return values
  - Multiple defined values

Summary

- llvm-dialects is a liberally licensed helper library (+ tablegen tool) that allows you to define “dialects” with some of the key niceties provided by MLIR, but in LLVM IR
- Already useful today, but still in early development with some rough edges
- We plan to continue developing the library for our (LLPC) use case
- We would be happy for you to join us!
- We will eventually want to post an RFC for upstream inclusion
  - Limitations in what can be done in an external library
  - Aim for true integration into llvm-project/llvm/lib/IR
  - There is no planned timeline
- https://github.com/GPUOpen-Drivers/llvm-dialects

Thank you!

COPYRIGHT AND DISCLAIMER

AMD
together we advance_