DirectX HLSL Compiler

Microsoft

Where We're Legalizing, We Don't Need Validators

Generating valid DXIL for the DirectX Backend

Farzon Lotfi
Principal Engineer
HLSL Modernization Project
10/28/2025

Outline

- Terms
- Current Flow
- Reimagining the LLVM-to-DXIL Pipeline
- Data Transformations: Preparing for DXIL
- The Custom Legalization Pass
- Optimization bypass for custom types
- Future Work

Terms

- DXIL – DirectX Intermediary Language A derivative of LLVM IR 3.7
- DXC – The Current production HLSL to DirectX backend compiler
- DXV – DXIL Validator- A tool that checks if a DXIL module meets the spec and add a validation hash

Current Flow: What is validation for

- To confirm generation of legal DXIL that drivers can compile
- To preserve compliance with DirectX API versions used by older and current GPUs.

Current Flow: When do we validate

In today's DXIL pipeline validation happens *after* code generation.

- This forces rules to be placed in the optimization pipeline in DXC.

Because validator (DXV) development happens in the same repository as DXC staying in sync is easy per release.

Current Flow: Why this is fragile?

- Errors surface too late (validator failures at the end of compilation).
- Legalization rules are forced into the DXC frontend which can impact SPIRV codegen.
- Newer DXC versions can generate invalid DXIL for older validators
- Maintaining compliance is a game of whack-a-mole.

What the validator (dxv) does ✅

- Checks that it follows all the structural and semantic rules defined by the DXIL specification
- Reports any violations — such as:
  - Illegal instruction types,
  - Invalid resource bindings or signatures,
  - Type mismatches,
  - Shader model version misuse,
  - Missing metadata,
  - Out-of-bounds indices or undefined behavior.
- Adds a validator hash for valid DXIL

What the validator (dxv) doesn’t do ❌

- ❌ Does not repair invalid IR or attempt to make corrections.
- ❌ Does not recompile or optimize the shader.
- ❌ Does not rewrite the DXIL module beyond possibly updating the validation hash.

Reimagining the LLVM-to-DXIL Pipeline

Core Idea:

- Don’t generate breaking constructs.
- Keep specific backend legalization requirements in backend passes
- Keep DXV just around for the validation hash and maintaining codgen parity with DXC.
  - LLVM IR → [Transformations & Legalizations] → DXIL BitcodeWriter → DXV → GPU

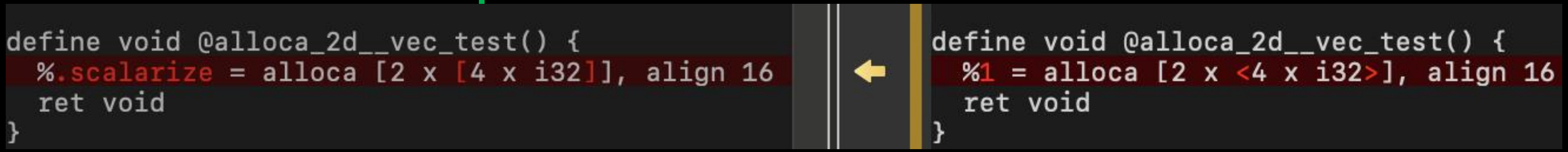
Goals

- Output is inherently DXIL-legal
- Predictable DXIL generation.
- Reuse of LLVM’s existing infrastructure for portability.
- No hidden rules.
  - Every DXV rule gets encoded as a transformation (Ie validation moves from checks to changes/legalizations).

Legalization Types: Data transformations

Scalarization:

DXIL 6.8 and older prefers scalar opcodes but most LLVM IR is vectorized. We extended the existing LLVM Scalarizer pass to ensure all vector operations expand predictably into legal scalar forms. We also do a custom data scalarization pass.

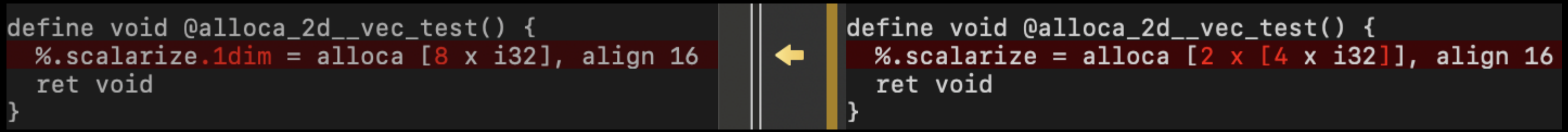


```
define void @alloca_2d__vec_test() {
  %.scalarize = alloca [2 x [4 x i32]], align 16
  ret void
}
```

```
define void @alloca_2d__vec_test() {
  %1 = alloca [2 x <4 x i32>], align 16
  ret void
}
```

Array flattening:

DXIL also restricts how arrays and aggregates are represented. We flatten vectors, nested and multidimensional arrays into linearized memory layouts.



```
define void @alloca_2d__vec_test() {
  %.scalarize.1dim = alloca [8 x i32], align 16
  ret void
}
```

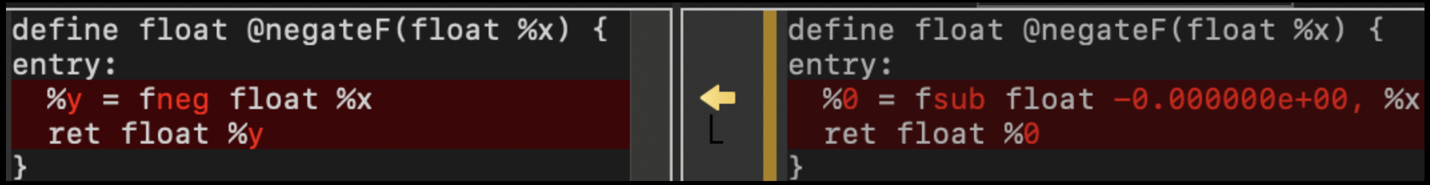
```
define void @alloca_2d__vec_test() {
  %.scalarize = alloca [2 x [4 x i32]], align 16
  ret void
}
```

Legalization: IR Transformations

- Custom Legalization Pass inspired by GlobalIsel Legalizer.
- Can’t use GlobalIsel b\c we need to maintain LLVMIR for Bitcode serialization.
- This pass rewrites any remaining non-DXIL-compliant constructs effectively acting as an in-compiler validator.
- This gives us a path to convert validation rules across generations of validators into transformations.

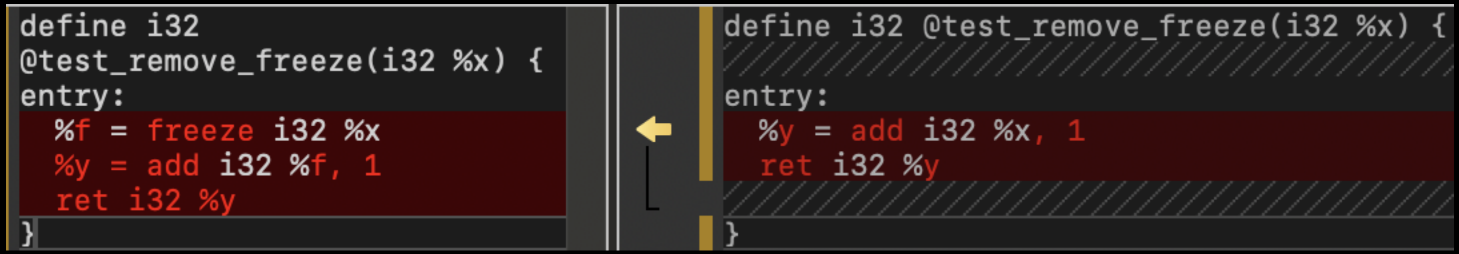
Legalization: Instruction Transformations

- Replace new instructions with ones that existed in 3.7



```
define float @negateF(float %x) {
entry:
  %y = fneg float %x
  ret float %y
}
```

```
define float @negateF(float %x) {
entry:
  %0 = fsub float -0.000000e+00, %x
  ret float %0
}
```

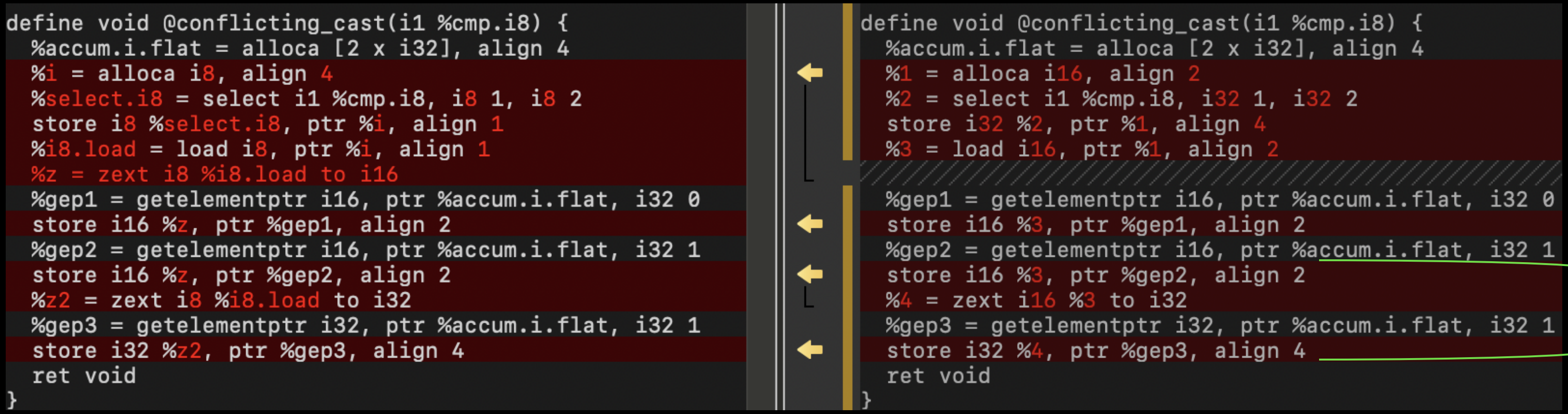


```
define i32
@test_remove_freeze(i32 %x) {
entry:
  %f = freeze i32 %x
  %y = add i32 %f, 1
  ret i32 %y
}
```

```
define i32 @test_remove_freeze(i32 %x) {
entry:
  %y = add i32 %x, 1
  ret i32 %y
}
```

Legalization: Type Transformations

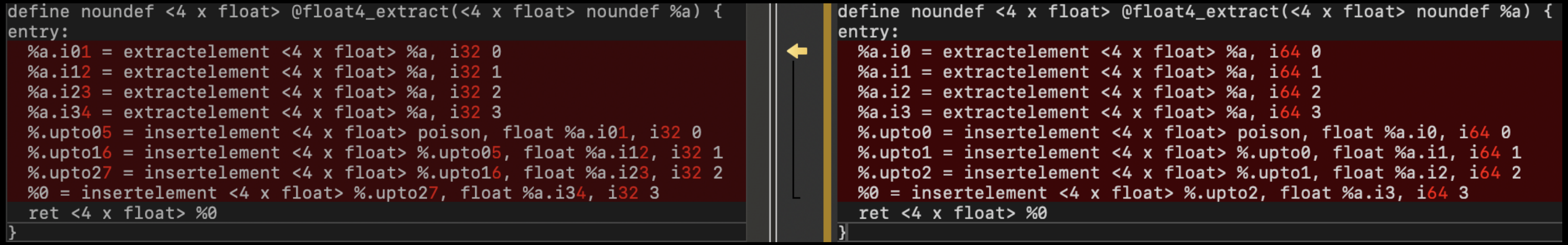
- Replace i8s and i64 indexes with legal types



```
define void @conflicting_cast(i1 %cmp.i8) {
  %accum.i.flat = alloca [2 x i32], align 4
  %i = alloca i8, align 4
  %select.i8 = select i1 %cmp.i8, i8 1, i8 2
  store i8 %select.i8, ptr %i, align 1
  %i8.load = load i8, ptr %i, align 1
  %z = zext i8 %i8.load to i16
  %gep1 = getelementptr i16, ptr %accum.i.flat, i32 0
  store i16 %z, ptr %gep1, align 2
  %gep2 = getelementptr i16, ptr %accum.i.flat, i32 1
  store i16 %z, ptr %gep2, align 2
  %z2 = zext i8 %i8.load to i32
  %gep3 = getelementptr i32, ptr %accum.i.flat, i32 1
  store i32 %z2, ptr %gep3, align 4
  ret void
}
```

```
define void @conflicting_cast(i1 %cmp.i8) {
  %accum.i.flat = alloca [2 x i32], align 4
  %1 = alloca i16, align 2
  %2 = select i1 %cmp.i8, i32 1, i32 2
  store i32 %2, ptr %1, align 4
  %3 = load i16, ptr %1, align 2
  %gep1 = getelementptr i16, ptr %accum.i.flat, i32 0
  store i16 %3, ptr %gep1, align 2
  %gep2 = getelementptr i16, ptr %accum.i.flat, i32 1
  store i16 %3, ptr %gep2, align 2
  %4 = zext i16 %3 to i32
  %gep3 = getelementptr i32, ptr %accum.i.flat, i32 1
  store i32 %4, ptr %gep3, align 4
  ret void
}
```

Stores let us know how to update the alloca use chain.

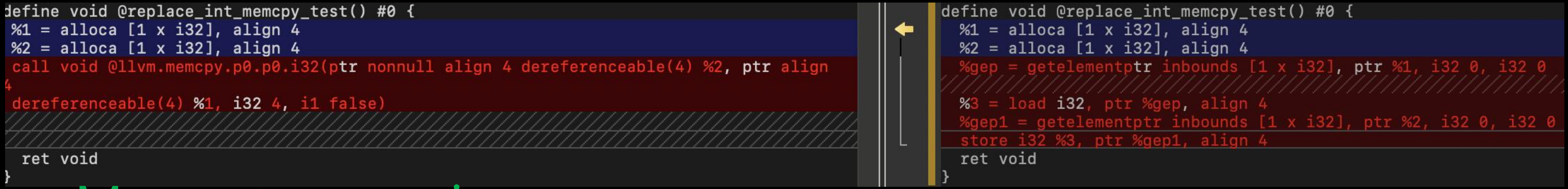


```
define noundef <4 x float> @float4_extract(<4 x float> noundef %a) {
entry:
  %a.i01 = extractelement <4 x float> %a, i32 0
  %a.i12 = extractelement <4 x float> %a, i32 1
  %a.i23 = extractelement <4 x float> %a, i32 2
  %a.i34 = extractelement <4 x float> %a, i32 3
  %.upto05 = insertelement <4 x float> poison, float %a.i01, i32 0
  %.upto16 = insertelement <4 x float> %.upto05, float %a.i12, i32 1
  %.upto27 = insertelement <4 x float> %.upto16, float %a.i23, i32 2
  %0 = insertelement <4 x float> %.upto27, float %a.i34, i32 3
  ret <4 x float> %0
}
```

```
define noundef <4 x float> @float4_extract(<4 x float> noundef %a) {
entry:
  %a.i0 = extractelement <4 x float> %a, i64 0
  %a.i1 = extractelement <4 x float> %a, i64 1
  %a.i2 = extractelement <4 x float> %a, i64 2
  %a.i3 = extractelement <4 x float> %a, i64 3
  %.upto0 = insertelement <4 x float> poison, float %a.i0, i64 0
  %.upto1 = insertelement <4 x float> %.upto0, float %a.i1, i64 1
  %.upto2 = insertelement <4 x float> %.upto1, float %a.i2, i64 2
  %0 = insertelement <4 x float> %.upto2, float %a.i3, i64 3
  ret <4 x float> %0
}
```

Legalization: Intrinsic Expansions

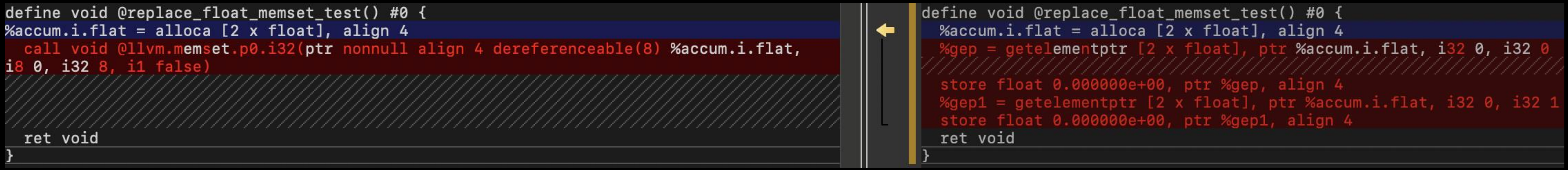
- Memcpy expansion



```
define void @replace_int_memcpy_test() #0 {
 %1 = alloca [1 x i32], align 4
 %2 = alloca [1 x i32], align 4
 call void @llvm.memcpy.p0.p0.i32(ptr nonnull align 4 dereferenceable(4) %2, ptr align
4
 dereferenceable(4) %1, i32 4, i1 false)

  ret void
}
```

```
define void @replace_int_memcpy_test() #0 {
  %1 = alloca [1 x i32], align 4
  %2 = alloca [1 x i32], align 4
  %gep = getelementptr inbounds [1 x i32], ptr %1, i32 0, i32 0

  %3 = load i32, ptr %gep, align 4
  %gep1 = getelementptr inbounds [1 x i32], ptr %2, i32 0, i32 0
  store i32 %3, ptr %gep1, align 4
  ret void
}
```

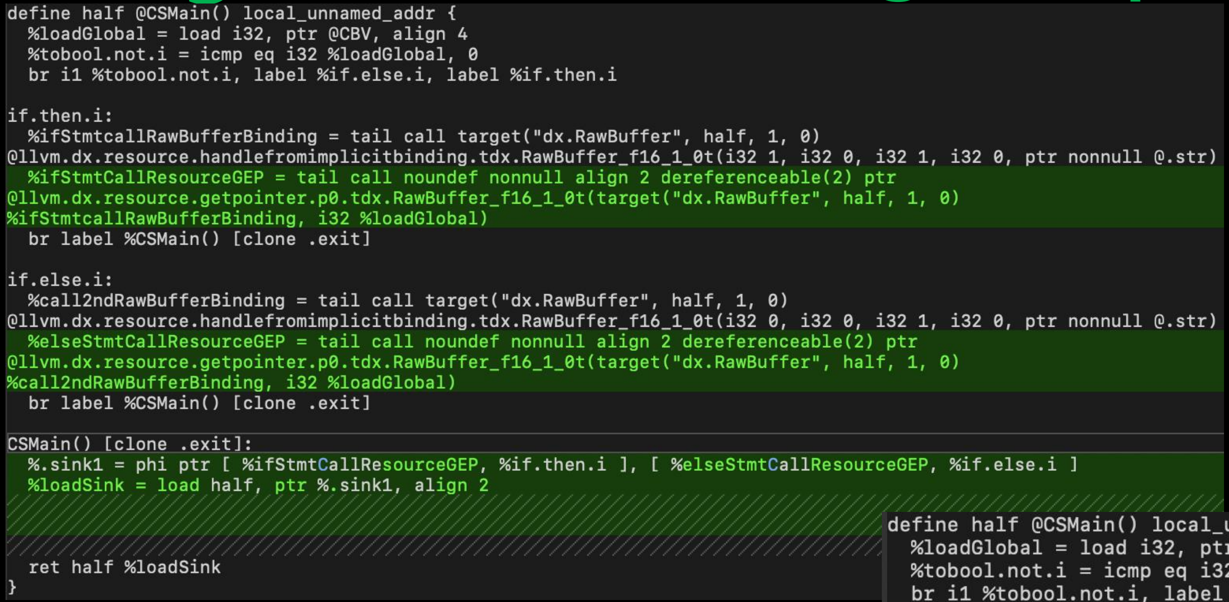
- Memset expansion



```
define void @replace_float_memset_test() #0 {
%accum.i.flat = alloca [2 x float], align 4
  call void @llvm.memset.p0.i32(ptr nonnull align 4 dereferenceable(8) %accum.i.flat,
i8 0, i32 8, i1 false)

  ret void
}
```

```
define void @replace_float_memset_test() #0 {
  %accum.i.flat = alloca [2 x float], align 4
  %gep = getelementptr [2 x float], ptr %accum.i.flat, i32 0, i32 0

  store float 0.000000e+00, ptr %gep, align 4
  %gep1 = getelementptr [2 x float], ptr %accum.i.flat, i32 0, i32 1
  store float 0.000000e+00, ptr %gep1, align 4
  ret void
}
```

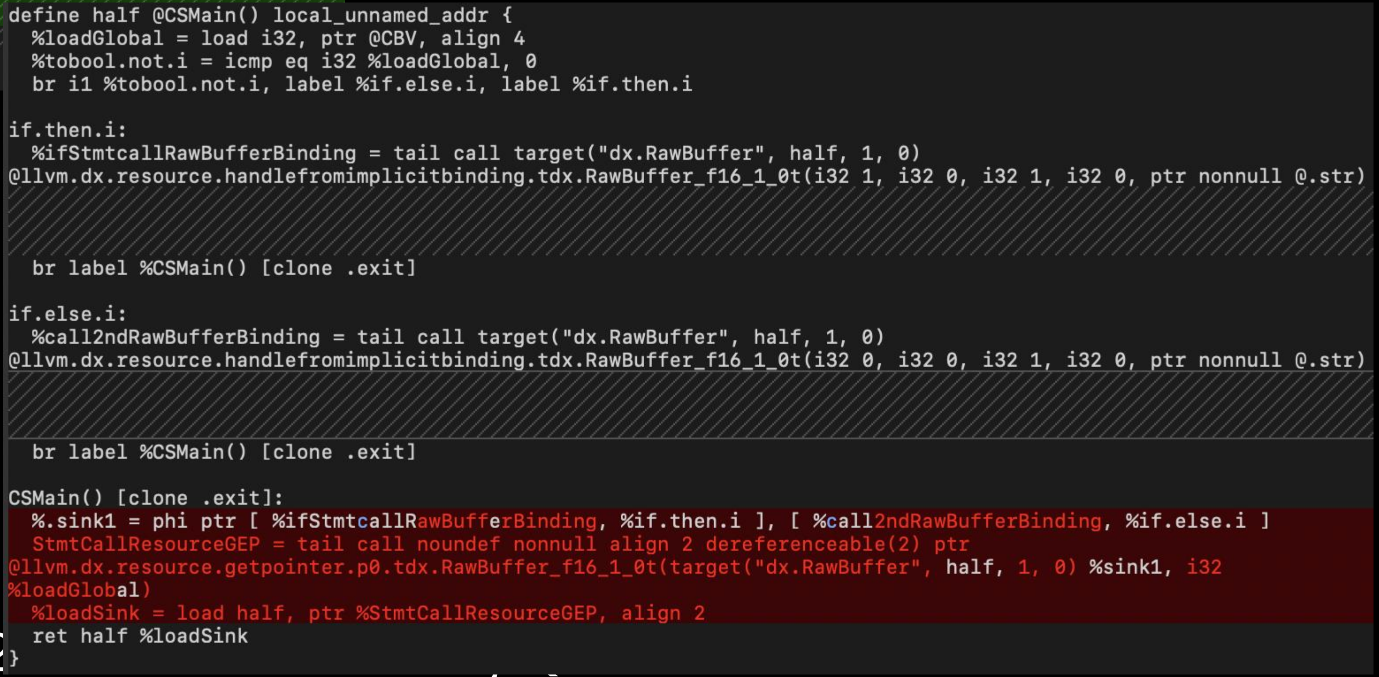
Legalization: Turning off optimizations for extension types



```
define half @CSMain() local_unnamed_addr {
  %loadGlobal = load i32, ptr @CBV, align 4
  %tobool.not.i = icmp eq i32 %loadGlobal, 0
  br i1 %tobool.not.i, label %if.else.i, label %if.then.i

if.then.i:
  %ifStmtcallRawBufferBinding = tail call target("dx.RawBuffer", half, 1, 0)
@llvm.dx.resource.handlefromimplicitbinding.tdx.RawBuffer_f16_1_0t(i32 1, i32 0, i32 1, i32 0, ptr nonnull @.str)
  %ifStmtCallResourceGEP = tail call noundef nonnull align 2 dereferenceable(2) ptr
@llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0)
%ifStmtcallRawBufferBinding, i32 %loadGlobal)
  br label %CSMain() [clone .exit]

if.else.i:
  %call2ndRawBufferBinding = tail call target("dx.RawBuffer", half, 1, 0)
@llvm.dx.resource.handlefromimplicitbinding.tdx.RawBuffer_f16_1_0t(i32 0, i32 0, i32 1, i32 0, ptr nonnull @.str)
  %elseStmtCallResourceGEP = tail call noundef nonnull align 2 dereferenceable(2) ptr
@llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0)
%call2ndRawBufferBinding, i32 %loadGlobal)
  br label %CSMain() [clone .exit]

CSMain() [clone .exit]:
  %.sink1 = phi ptr [ %ifStmtCallResourceGEP, %if.then.i ], [ %elseStmtCallResourceGEP, %if.else.i ]
  %loadSink = load half, ptr %.sink1, align 2

  ret half %loadSink
}
```

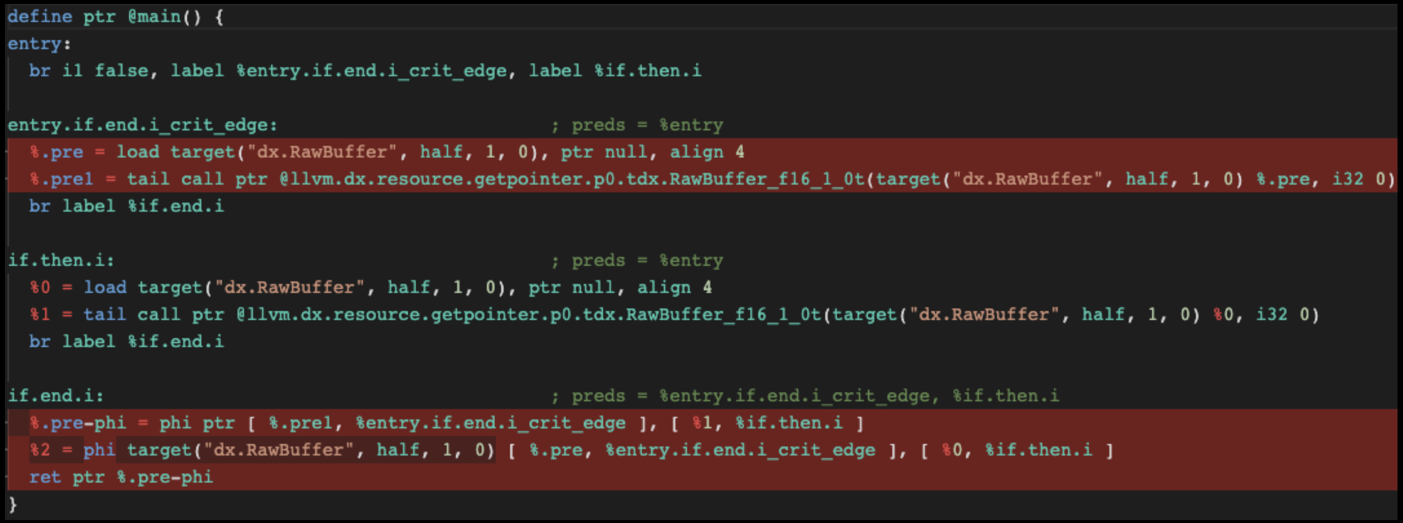
- Treat DX resource target-ext types (like dx.RawBuffer) as *token-like* so optimizations can't rewrite them through phi/select or appear in non-intrinsic function signatures.
- This prevents simplifyCFG from modification.

All that is going on here is dx.rawbuffer is maintained in individual basic blocks.



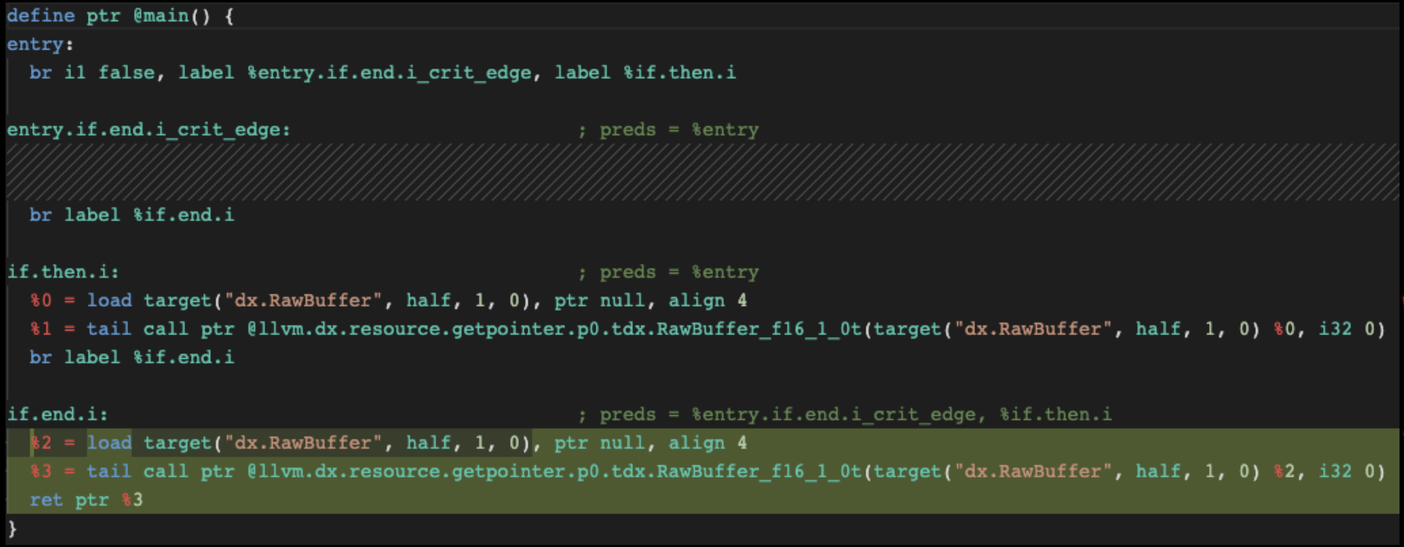
```
define half @CSMain() local_unnamed_addr {
  %loadGlobal = load i32, ptr @CBV, align 4
  %tobool.not.i = icmp eq i32 %loadGlobal, 0
  br i1 %tobool.not.i, label %if.else.i, label %if.then.i

if.then.i:
  %ifStmtcallRawBufferBinding = tail call target("dx.RawBuffer", half, 1, 0)
@llvm.dx.resource.handlefromimplicitbinding.tdx.RawBuffer_f16_1_0t(i32 1, i32 0, i32 1, i32 0, ptr nonnull @.str)

  br label %CSMain() [clone .exit]

if.else.i:
  %call2ndRawBufferBinding = tail call target("dx.RawBuffer", half, 1, 0)
@llvm.dx.resource.handlefromimplicitbinding.tdx.RawBuffer_f16_1_0t(i32 0, i32 0, i32 1, i32 0, ptr nonnull @.str)

  br label %CSMain() [clone .exit]

CSMain() [clone .exit]:
  %.sink1 = phi ptr [ %ifStmtcallRawBufferBinding, %if.then.i ], [ %call2ndRawBufferBinding, %if.else.i ]
  StmtCallResourceGEP = tail call noundef nonnull align 2 dereferenceable(2) ptr
@llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0) %sink1, i32
%loadGlobal)
  %loadSink = load half, ptr %StmtCallResourceGEP, align 2
  ret half %loadSink
}
```

Legalization: Turning off optimizations for extension types



```
define ptr @main() {
entry:
  br i1 false, label %entry.if.end.i_crit_edge, label %if.then.i

entry.if.end.i_crit_edge:                         ; preds = %entry
  %.pre = load target("dx.RawBuffer", half, 1, 0), ptr null, align 4
  %.pre1 = tail call ptr @llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0) %.pre, i32 0)
  br label %if.end.i

if.then.i:                                        ; preds = %entry
  %0 = load target("dx.RawBuffer", half, 1, 0), ptr null, align 4
  %1 = tail call ptr @llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0) %0, i32 0)
  br label %if.end.i

if.end.i:                                         ; preds = %entry.if.end.i_crit_edge, %if.then.i
  %.pre-phi = phi ptr [ %.pre1, %entry.if.end.i_crit_edge ], [ %1, %if.then.i ]
  %2 = phi target("dx.RawBuffer", half, 1, 0) [ %.pre, %entry.if.end.i_crit_edge ], [ %0, %if.then.i ]
  ret ptr %.pre-phi
}
```

- Don’t let GVN pass generate phi for token like extension types.
- Second GVN case



```
define ptr @main() {
entry:
  br i1 false, label %entry.if.end.i_crit_edge, label %if.then.i

entry.if.end.i_crit_edge:                         ; preds = %entry


  br label %if.end.i

if.then.i:                                        ; preds = %entry
  %0 = load target("dx.RawBuffer", half, 1, 0), ptr null, align 4
  %1 = tail call ptr @llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0) %0, i32 0)
  br label %if.end.i

if.end.i:                                         ; preds = %entry.if.end.i_crit_edge, %if.then.i
  %2 = load target("dx.RawBuffer", half, 1, 0), ptr null, align 4
  %3 = tail call ptr @llvm.dx.resource.getpointer.p0.tdx.RawBuffer_f16_1_0t(target("dx.RawBuffer", half, 1, 0) %2, i32 0)
  ret ptr %3
}
```

Future work

- This won’t replace the validator overnight but gives us a means for the new compiler to be responsive to current and older validators.
- Our work here is incomplete. The plan is to continue to encode validation rules into the DirectX backend so that All HLSL compiled with clang-dxc generates valid DXIL.

Thank you!