A Beginner's Guide to SelectionDAG

Justin M. Fargnoli Alex E. MacLean

NVIDIA

2024 LLVM Developers' Meeting

Why talk about SelectionDAG?

SelectionDAG is a framework for instruction selection.

SelectionDAG is important:

- Many targets use SelectionDAG (X86, NVPTX, MIPS, Hexagon, . . .).
- Decisions made within SelectionDAG have a significant impact on final code quality.
- Support for new target instructions or intrinsics requires SelectionDAG changes.

SelectionDAG can be confusing:

- Layer of abstraction between generic and target specific portions can make it hard to follow.

Who is this talk for?

- You feel comfortable working in the LLVM codebase.
- You've heard of SelectionDAG.
- But, you don't quite get it.

Table of Contents

NVIDIA

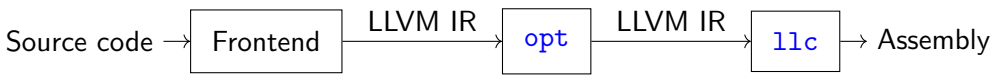
Caveats

- We are not the authors of this framework nor are we experts.
- We have primarily worked on the NVPTX target.
- We have:
  - $\sim$ 2 years of experience with LLVM
  - $\sim$ 2 year of experience with NVPTX
- GlobalISel is an alternative to SelectionDAG, not covered in this presentation.

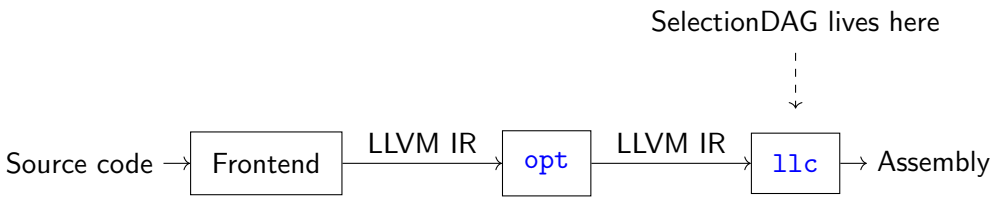
NVIDIA

Table of Contents

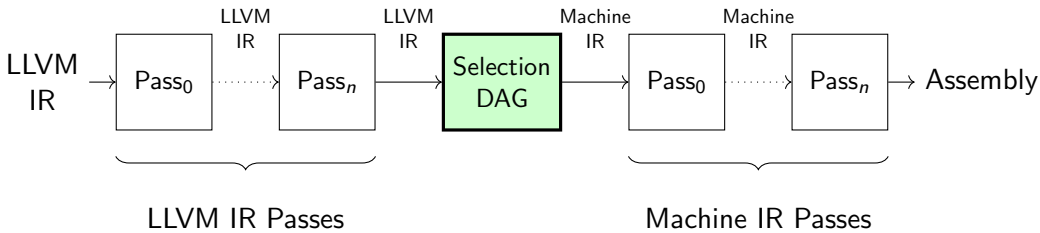
Compilation Flow



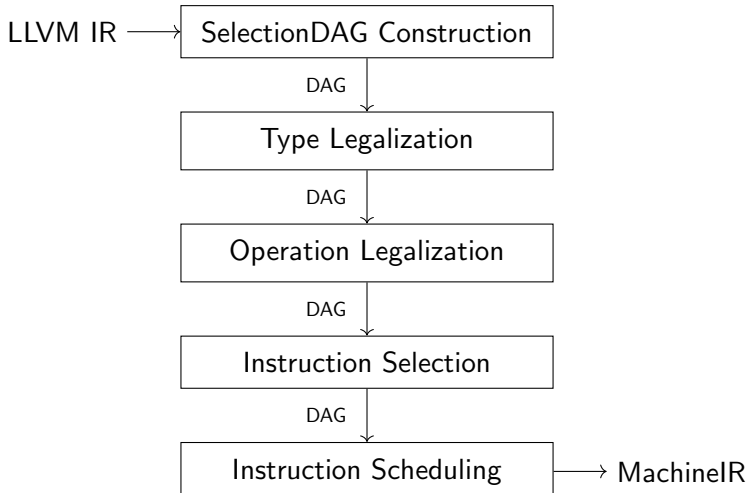
Compilation Flow



llc Compilation Flow



SelectionDAG Compilation Flow



SelectionDAG Compilation Flow

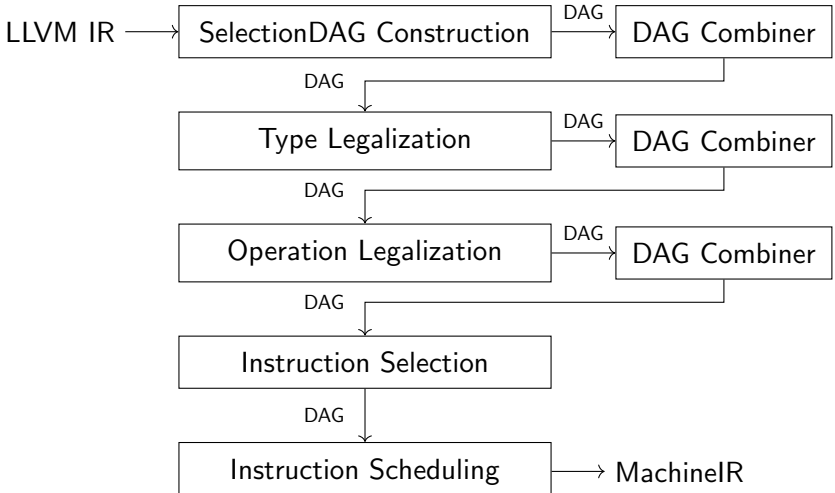
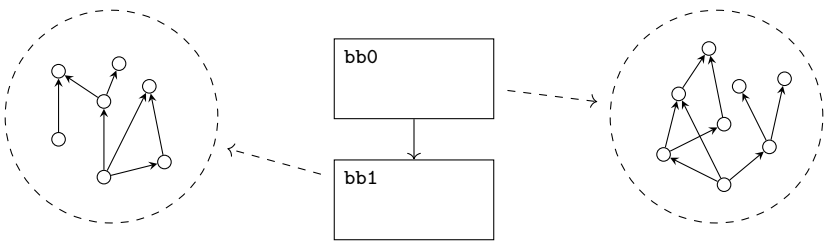


Table of Contents

SelectionDAG - The Data Structure

- SelectionDAGs are an alternate representation of the program
- Each SelectionDAG represents a single basic block



SelectionDAG Types - MVT

Definition

Machine Value Type (MVT) A union of the types that are supported by each target that uses SelectionDAG.

- Any given target will only support a subset of these types
- MVT examples include:
  - Integers: $\{\text{i1}, \text{i32}, \text{i128}, \dots\}$
  - Floats: $\{\text{f16}, \text{bf16}, \text{f80}, \dots\}$
  - Vectors: $\{\text{v1i1}, \text{v2i32}, \text{v128bf16}, \dots\}$
  - Other things: $\{\text{Other}, \text{Glue}, \dots\}$

SelectionDAG Types - EVT

Definition

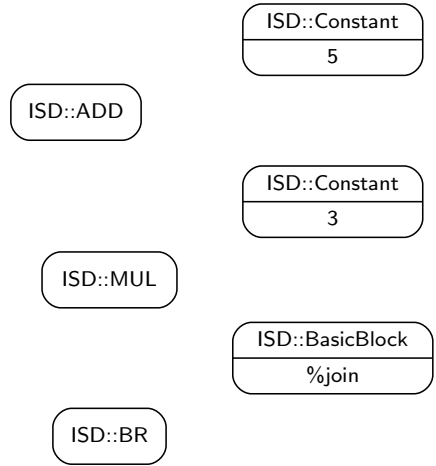
Extended Value Type (EVT) A union of the MVT types and all integer, float and vectors types that LLVM IR supports

- Does not include all LLVM IR types, struct and array types are not in the set
- EVT examples include:
  - All MVT types: $\{\text{i1}, \text{i32}, \text{i128}, \text{f16}, \text{bf16}, \text{f80}, \text{v1i1}, \text{v2i32}, \text{v128bf16}, \text{Other}, \text{Glue}, \dots\}$
  - Integer and vector lengths not natively supported on any architectures: $\{\text{i3}, \text{v100i32}, \text{v99f32}, \text{v99i99}, \dots\}$

SDNode

- These are the building blocks of all SelectionDAGs
- Each node has:
  - Opcode which defines the type
  - Potentially many other fields such as constant value or flags

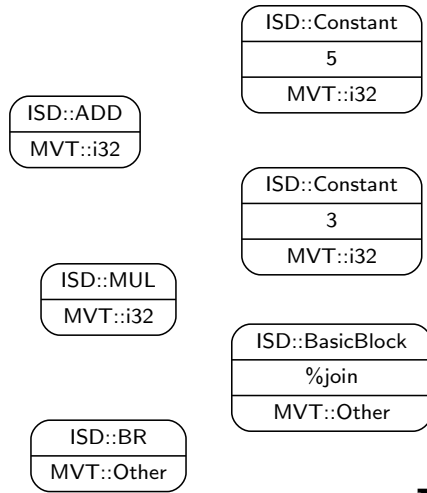
```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```



SDValue

- Represents the "output" of an SDNode
- Has an associated EVT
- SDValues have:
  - SDNode that defines this value
  - Index into the list of results from that node

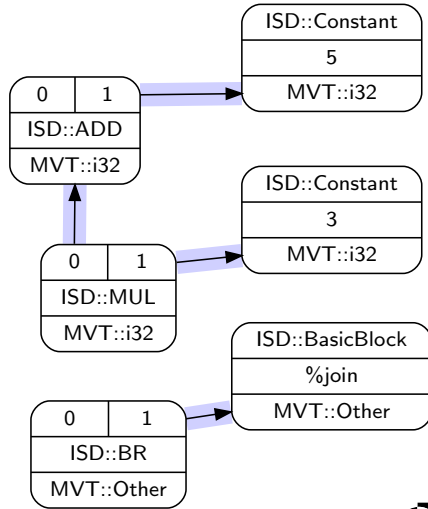
```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```



SDUse

- Represents the "input" to an SDNode
- SDUses have:
  - SDValue that is being used
  - SDNode that is the user
  - Operand index in the user node operand list[1]

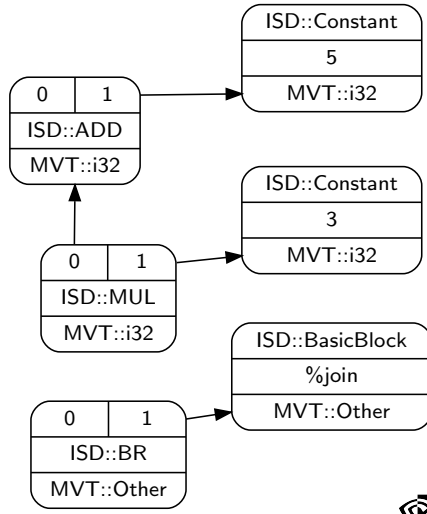
```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```



[1]This is a good mental model, actually implemented as a linked list

SDNode - Revisited

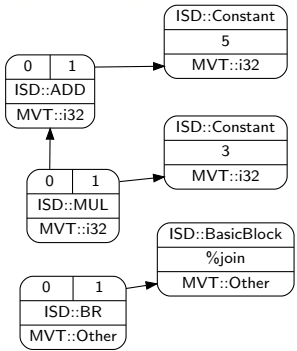
- These are the building blocks of all SelectionDAGs
- Each node has:
  - Opcode which defines the type
  - **1 or more results represented by SDValues**
  - **0 or more operands represented by SDUses**
  - Potentially many other fields such as constant value or flags



Cross-Block Data-Flow

How can we represent %a and %z SSA registers which are live across multiple blocks?

```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```

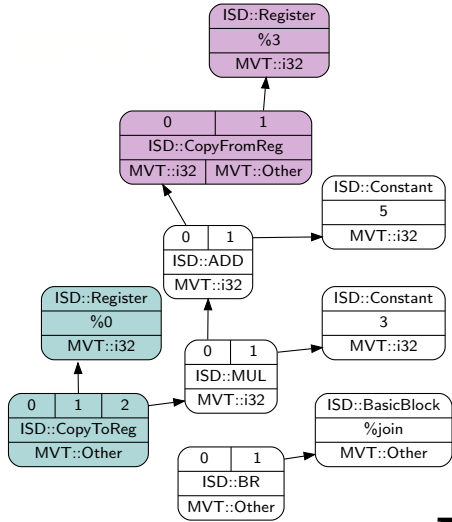


Cross-Block Data-Flow

How can we represent %a and %z SSA registers which are live across multiple blocks?

- CopyFromReg - SSA register used here defined elsewhere
- CopyToReg - SSA register used elsewhere defined here

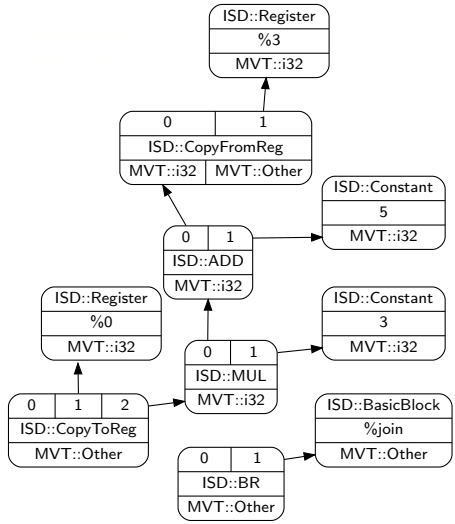
```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```



Scheduling Dependencies

How to ensure that the branch instruction is scheduled after the rest of the block?

```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```

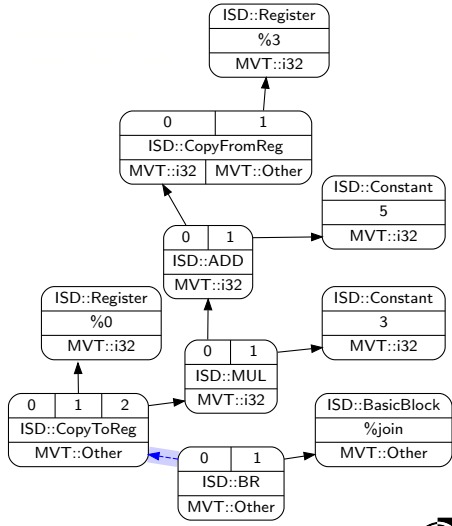


Scheduling Dependencies

How to ensure that the branch instruction is scheduled after the rest of the block?

- Chain values represent non-data dependencies
- As with all other SDUses, the user must be scheduled after the use

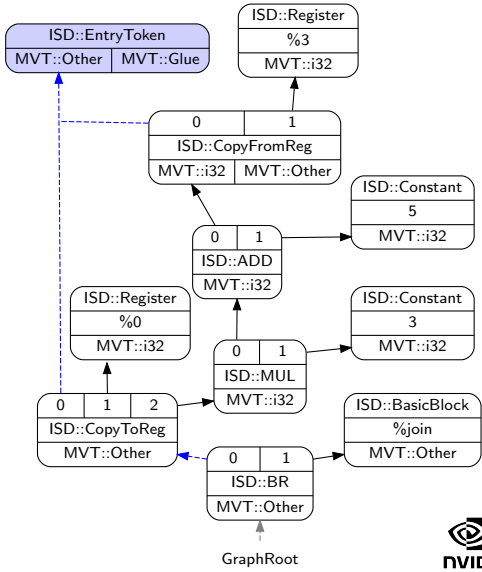
```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```



Beginning and Ending the SelectionDAG

- An Entry token node is added to every block. It represents the dependency on entering the block.
- Each DAG also has a root, usually the terminator instruction

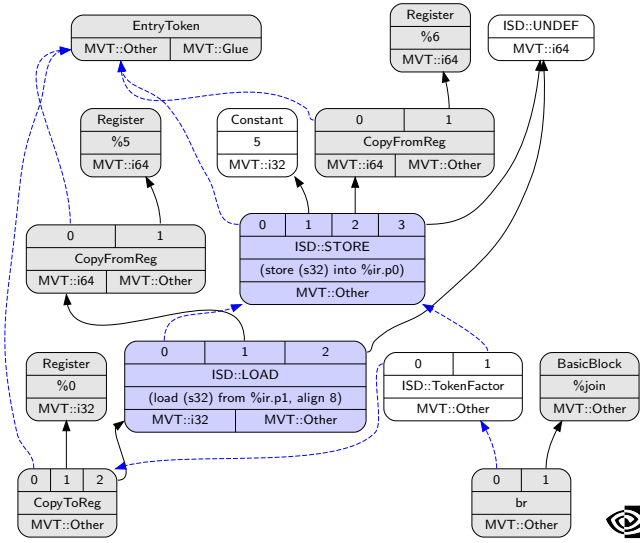
```
then:
%y = add i32 %a, 5
%z = mul i32 %y, 3
br label %join
```



Another Example: Loads and Stores

- Chained together to represent memory dependence
- MemSDNodes contain lots of info about memory interaction

```
then:
store i32 5, ptr %p0
%1 = load i32, ptr %p1, align 8
br label %join
```



Another Example: Loads and Stores

- Token Factor joins multiple chains, allowing node to use multiple dependencies.
- Convenience node will not be joined.

```
then:
store i32 5, ptr %p0
%1 = load i32, ptr %p1, align 8
br label %join
```

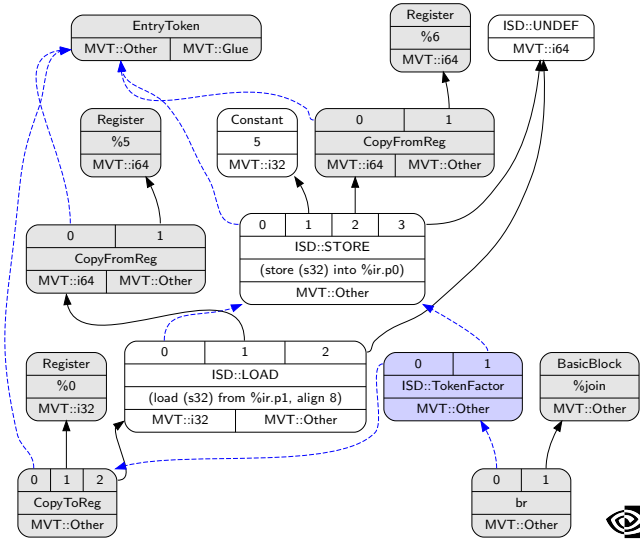
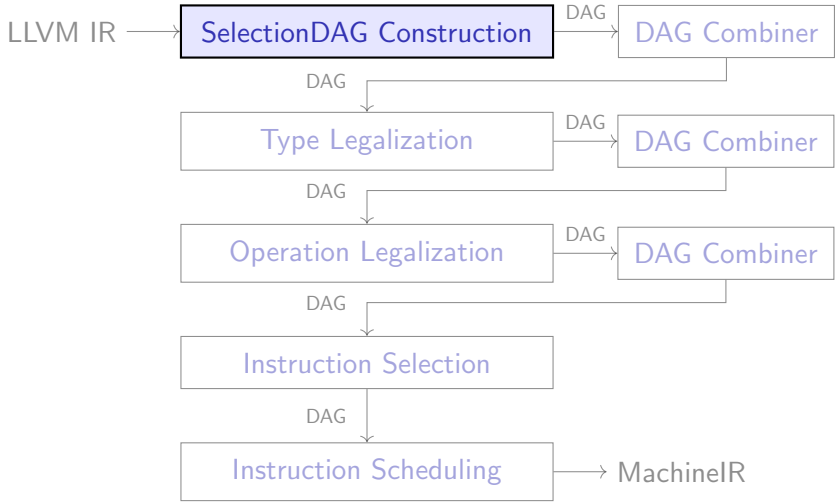


Table of Contents

Phases of SelectionDAG



Building the SelectionDAG

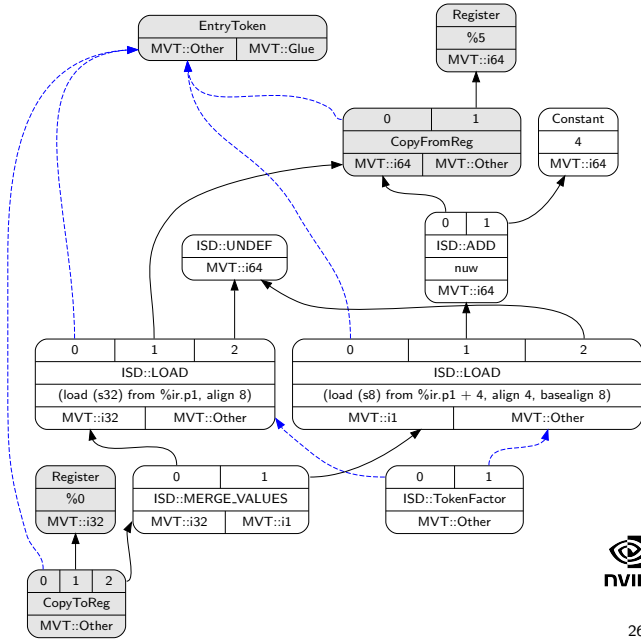
- Build a representation of each basic block in the function.
- 1:1 correspondence between LLVM IR instructions and SelectionDAG nodes (with a few exceptions).
- Struct types are not supported in SelectionDAG. Instructions using them must be expanded.
- Target hooks are required for tricky instructions.

NVIDIA

Example: Struct Splitting

- Struct operations are lowered element-wise.

```
then:
%l = load {i32, i1}, ptr %p1, align 8
%v = extractvalue {i32, i1} %l, 0
br label %join (not pictured)
```



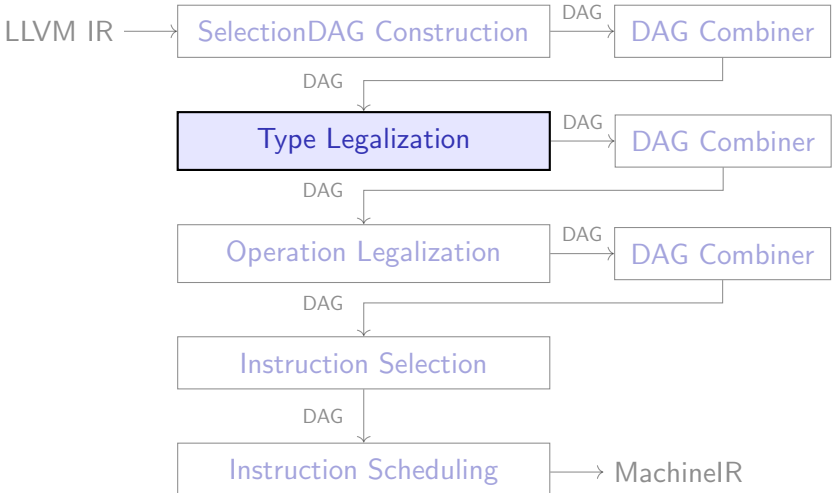
Target Specific APIs

Calling conventions are too target-specific for a generic DAG representation to be feasible. Each target must implement custom DAG building with the following APIs:

- `SDValue TargetLowering::LowerCall(...)`
- `SDValue TargetLowering::LowerFormalArgs(...)`
- `SDValue TargetLowering::LowerReturn(...)`

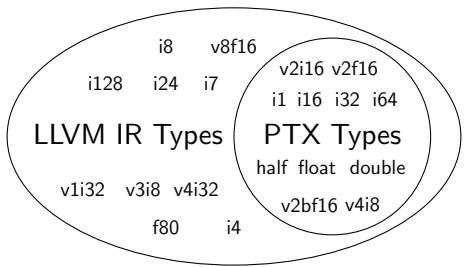
NVIDIA

Phases of SelectionDAG



Type Legalization

- Targets only support a subset of the types that LLVM IR supports.
- Goal: Lower illegal types to legal types.
- Legal ≔ Supported by instruction selection.



How do targets control this?

- `addRegisterClass(MVT)`
  - Communicate to SelectionDAG that the `MVT` is legal for the target.
- Given this information, SelectionDAG handles unsupported types for us!

Behind the Scenes

- SelectionDAG constructs a table that maps every type to an action[2] that will legalize the type.

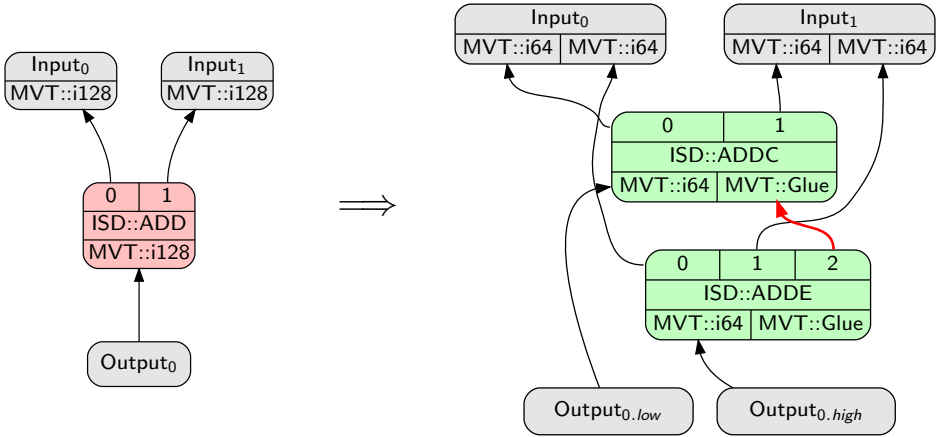
`TypeLegal`	The target natively supports this type.
`TypePromoteInteger`	Replace this integer with a larger one.
`TypeExpandInteger`	Split this integer into two of half the size.
`TypeSoftenFloat`	Convert this float to a same size integer type.
`TypePromoteFloat`	Replace this float with a larger one.
`TypeSoftPromoteHalf`	Soften half to i16 and use float to do arithmetic.
`TypeScalarizeVector`	Replace this one-element vector with its element.
`TypeSplitVector`	Split this vector into two of half the size.
`TypeWidenVector`	This vector should be widened into a larger vector.

Table: Supported `LegalizeTypeAction`s

[2]The legalization action is performed in Legalize*Types.cpp.

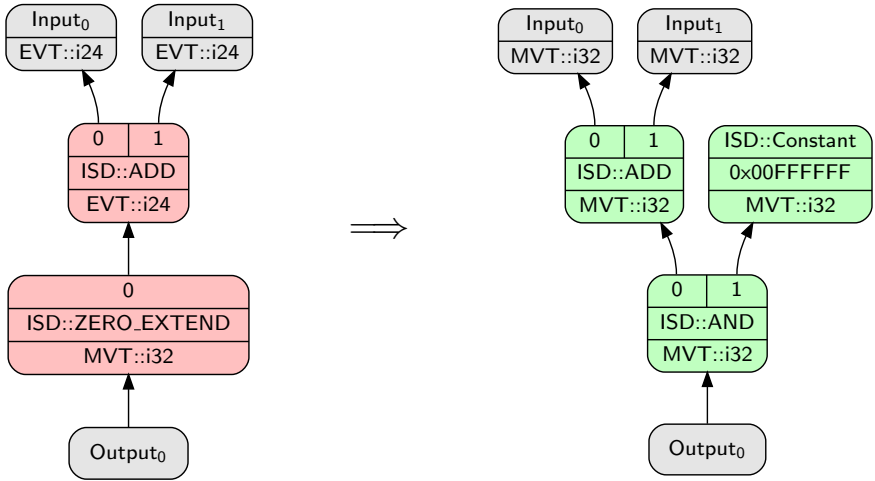
TypeExpandInteger

Type legalization can produce new nodes. It's doesn't just modify the types.



TypePromoteInteger

SelectionDAG handles the possibility of overflow by masking off the high bits.



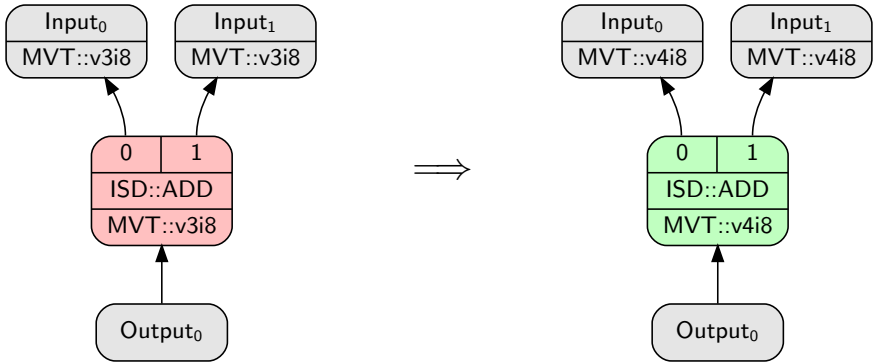
Special Handling for Vectors

- Targets can *optionally* instruct SelectionDAG on how to legalize a vector type.
- `LegalizeTypeAction getPreferredVectorAction(MVT);`
  - By overriding this method, a target tells SelectionDAG what `LegalizeTypeAction` should be performed to legalize the `MVT`.

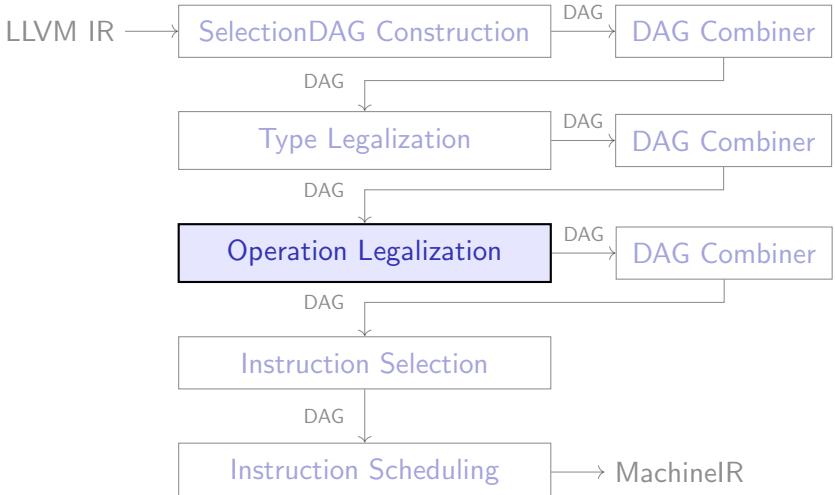
NVIDIA

TypeWidenVector

[Hexagon] Prefer a larger vector that is supported by the target.



Phases of SelectionDAG



Operation Legalization

- SelectionDAG supports 400+ opcodes.
- Targets do *not* support all:
  - Opcodes
  - Combinations of Opcodes $\times$ Legal Types
- Goal: Lower operations supported by SelectionDAG to operations that are legal for the target.

How do targets control this?

- `setOperationAction(Opcode, MVT, LegalizeAction)`
  - Communicate to SelectionDAG how the target will supports the `Opcode` x `MVT`.
- The semantics of the `MVT` parameter aren't well defined.
  - Sometimes it's the operand type.
  - Other times it's the return type.
  - When in doubt, read the code!

NVIDIA

Behind the Scenes

- SelectionDAG constructs a table that maps every `Opcode` × Legal `MVT` to an action[3] that will legalize that `Opcode` × Legal `MVT`.
- `setOperationAction(Opcode, MVT, LegalizeAction)` is how we override the default values in the table.

`Legal`	The target natively supports this operation.
`Promote`	This operation should be executed in a larger type.
`Expand`	Emulate this operation using other operations.
`Custom`	Use the `LowerOperation()` hook to implement custom lowering.

Table: Supported `LegalizeAction`s

[3]The legalization action is performed in LegalizeDAG.cpp.

Promote

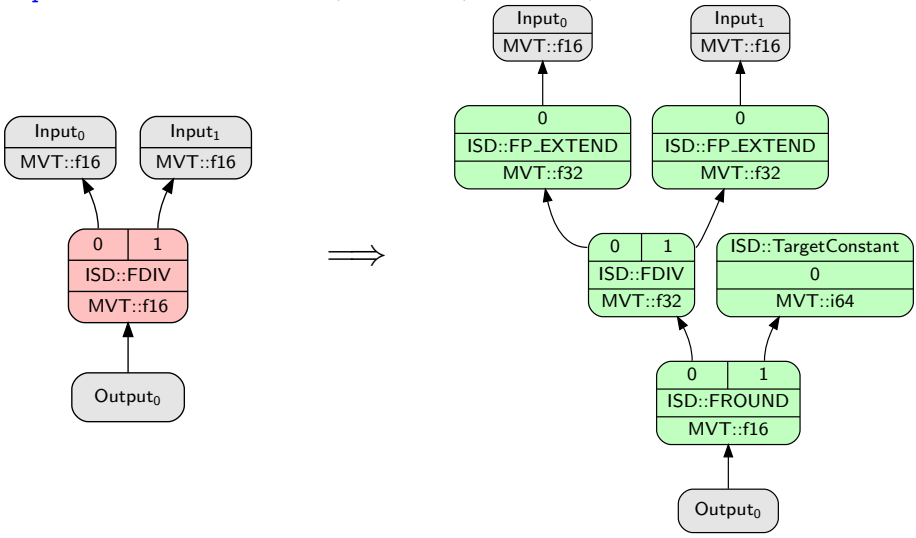
[NVPTX] `ISD::FDIV`

Type	Type supported?	`ISD::FDIV` supported?
`MVT::f32`	✓	✓
`MVT::f16`	✓	✗

- Problem: `ISD::FDIV` × `MVT::f16` is not supported.
- Solution: Execute `ISD::FDIV` on a larger type.
  - `setOperationAction(ISD::FDIV, MVT::f16, Promote);`

Promote

[NVPTX] `setOperationAction(ISD::FDIV, MVT::f16, Promote);`



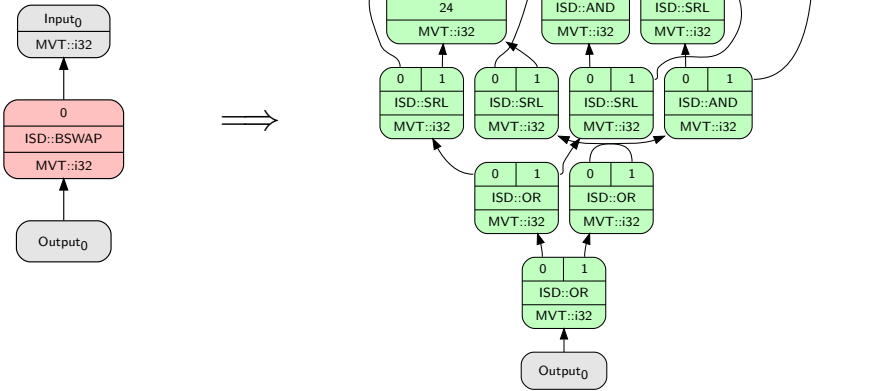
Expand

[MIPS] `ISD::BSWAP`

- Problem: `ISD::BSWAP` $\times$ `MVT::i32` is not supported.
- Solution: Emulate `ISD::BSWAP` with operations that are legal for the target.
  - `setOperationAction(ISD::BSWAP, MVT::i32, Expand);`
- Note: `Expand` does not specify *how* to emulate the operation.

Expand

[MIPS] `setOperationAction(ISD::BSWAP, MVT::i32, Expand);`



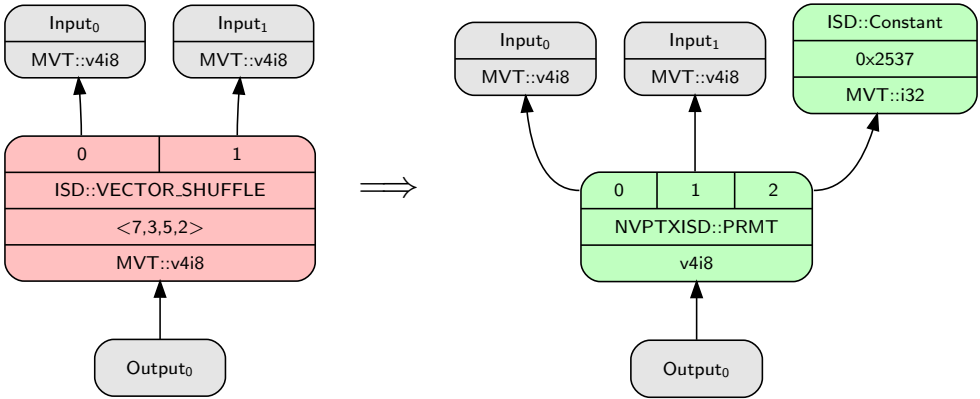
Custom Operation Legalization

- Call `setOperationAction(Opcode, MVT, Custom)` to tell SelectionDAG that we want to implement a custom lowering for `Opcode` × `MVT`.
- When SelectionDAG encounters `Opcode` × `MVT`, it will call `LowerOperation()`.
- Targets override the `LowerOperation()` function to implement the custom lowering.

NVIDIA

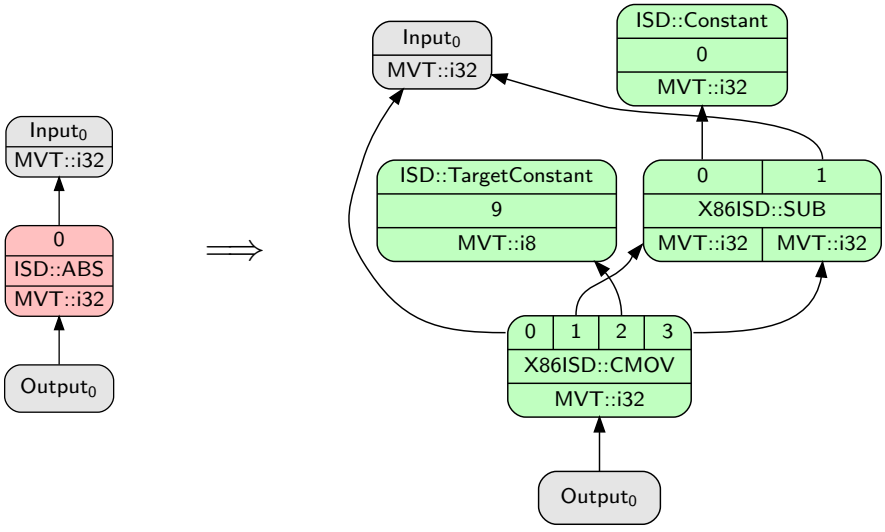
Custom

[NVPTX] `setOperationAction(ISD::VECTOR_SUFFLE, MVT::v4i8, Custom)`

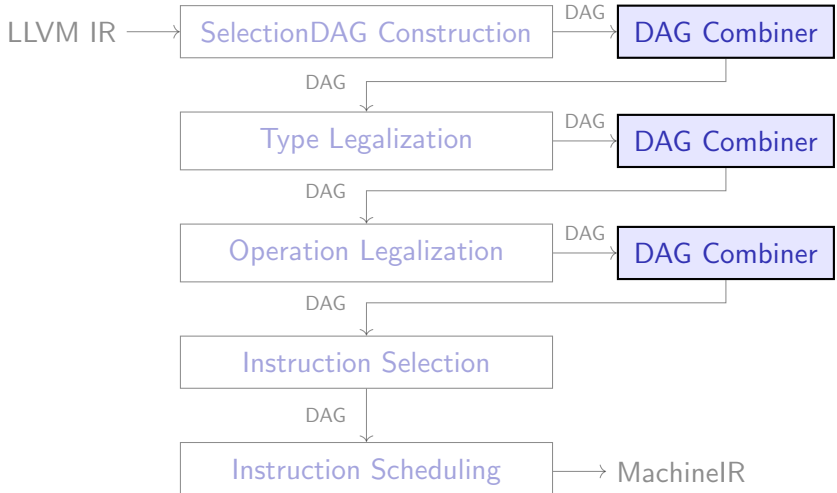


Custom

[X86] `setOperationAction(ISD::ABS, MVT::i32, Custom)`



Phases of SelectionDAG



Why optimize the SelectionDAG?

Why optimize the SelectionDAG? Haven't all the peephole optimizations already been done in LLVM IR?

Why optimize the SelectionDAG?

Why optimize the SelectionDAG? Haven't all the peephole optimizations already been done in LLVM IR?

- Clean up inefficiencies that were introduced while lowering into SelectionDAG and legalizing the DAG.
- Perform peephole that generate unique operations provided by the ISA.

DAGCombiner

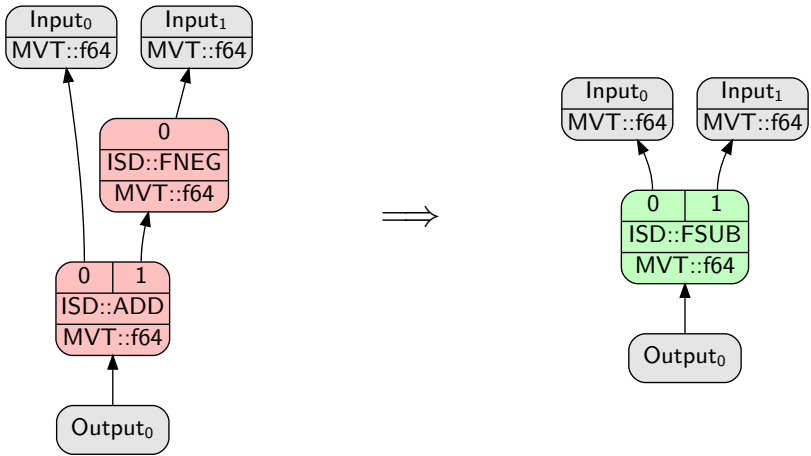
- Performs peepholes for all targets that use SelectionDAG.
  - "InstCombine for SelectionDAG"
- It calls `TargetLoweringInfo` (`TLI`) to understand which optimizations are profitable for a target
- Example from DAGCombiner:

```
// fold (fadd A, (fneg B)) -> (fsub A, B)
if (SDValue NegN1 = TLI.getCheaperNegatedExpression(N1, DAG))
    return DAG.getNode(ISD::FSUB, VT, N0, NegN1);
```

NVIDIA

DAGCombiner

Perform transformation if `TLI.getCheaperNegatedExpression()` is `true`

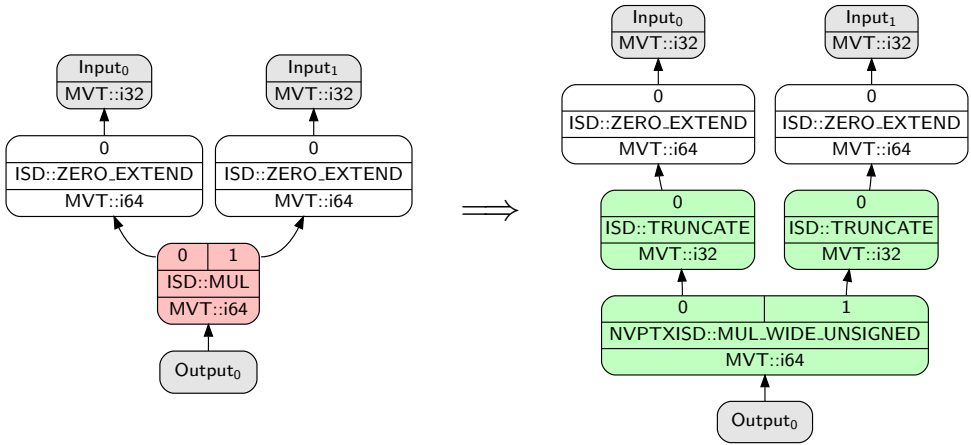


Custom DAG Combines

- `setTargetDAGCombine(Opcode)` tells SelectionDAG that we want to implement a custom DAG combine for `Opcode`.
- Then, when SelectionDAG encounters `Opcode`, it calls `PerformDAGCombine()`.
- Targets override `PerformDAGCombine()` to implement their DAG combines.

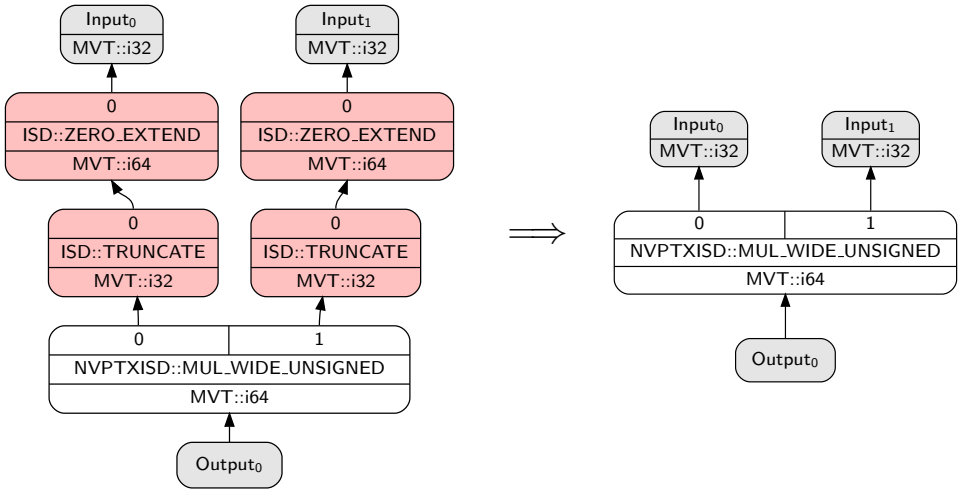
Custom DAG Combine

[NVPTX] Reduce register pressure using wide multiply.



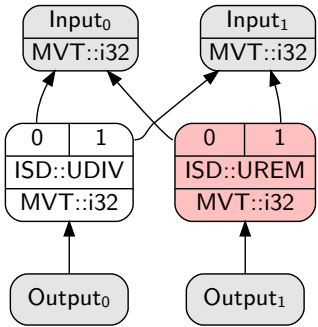
Custom DAG Combine

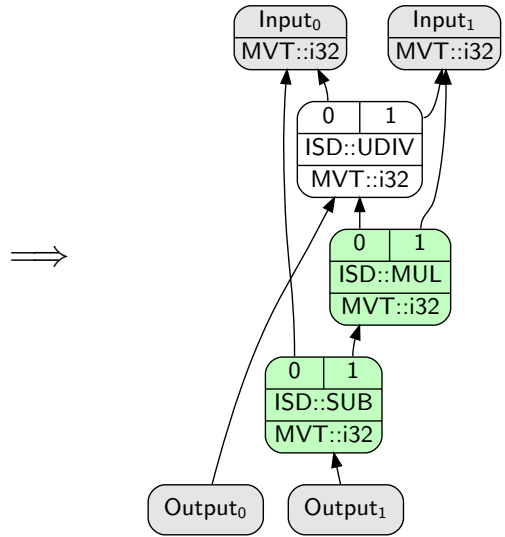
[NVPTX] Like InstCombine, DAGCombiner runs iteratively.



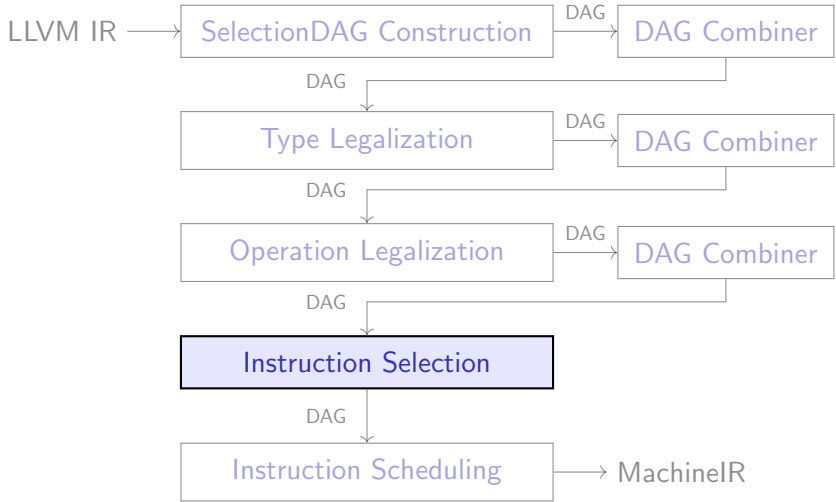
Custom DAG Combine

[NVPTX] Remove the expensive `ISD::UREM` instruction.





Phases of SelectionDAG



Instruction Selection

- Replace most generic `SDNodes` with machine nodes.
- Machine nodes have `MachineInstruction` Opcodes.
- Too target-specific for a generic implementation. Each target must override `Select`.
- Lots of matching code is required.

```
/// Main hook for targets to transform nodes into machine nodes.
virtual void Select(SDNode *N) = 0;
```

TableGen to the Rescue

- Allows succinct specification of instructions and generic DAG patterns they correspond to.
- TableGen will automatically generate all matching code.
- Targets may still implement custom logic outside of TableGen as needed.

NVIDIA

TableGen Instruction Selection Patterns

We need to define 3 items to allow TableGen to build maching code:

- **SDPatternOperator(s)** - Describes what we are looking for in the DAG
- **Instruction** - Representation of the machine instruction we will emit
- **Pattern** - Mapping between DAG and instruction(s)

TableGen: Matching the SelectionDAG

```
def SDTIntBinOp : SDTypeProfile<
    1,                                  Produces 1 value
    2,                                  Takes 2 operands
    [SDTCisSameAs<0, 1>,                Result type (0) is same as first operand (1)
     SDTCisSameAs<0, 2>,                Result type (0) is same as second operand (2)
     SDTCisInt<0>]>;                    Produced value (0) is an integer MVT
```

```
def add : SDNode<
    "ISD::ADD",                         Opcode is ISD::ADD
    SDTIntBinOp,                        SelectionDAG type is SDTIntBinOp
    [SDNPCommutative,                   Properties include commutative, associative
     SDNPAssociative]>;
```

TableGen: Defining a Machine Instruction

- TableGen lets us populate the instruction record with lots of useful information. For ISel, all we're concerned with are the inputs and outputs.

```
def ADDi32rr :  NVPTXInst<
    (outs Int32Regs:$dst),
    (ins Int32Regs:$a, Int32Regs:$b),
    "add.s32 \t$dst, $a, $b;">;
```

OpCode name of this instruction
Types of registers set by this instruction
Types of registers read by this instruction
Assembly string for printing

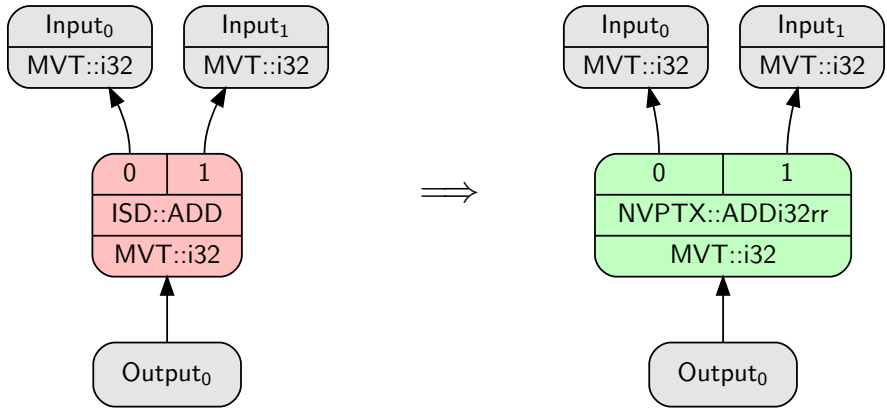
TableGen: Defining a Rewriting Pattern

```
def add : SDNode<...>
```

```
def ADDi32rr : NVPTXInst<...>
```

```
def : Pattern<
    (set Int32Regs:$dst,
      (add (i32 Int32Regs:$a), (i32 Int32Regs:$b))),
    (ADDi32rr Int32Regs:$dst,
              Int32Regs:$a, Int32Regs:$b)>;
```

TableGen Instruction Selection



Select Implementation Design Pattern

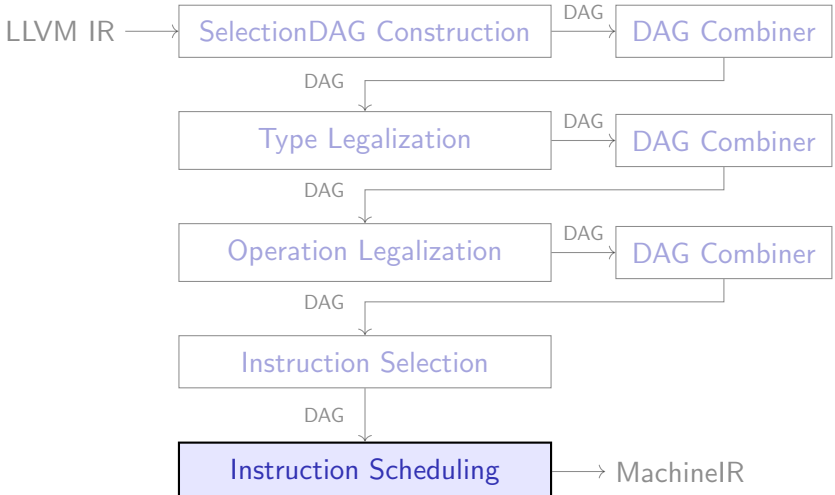
- Targets may insert custom logic prior to TableGen based matching for very complex cases.
- `SelectCode` will execute the TableGen-generated matcher.

```
void NVPTXDAGToDAGISel::Select(SDNode *N) {
    // Custom logic here
    SelectCode(N); // TableGen based selection
}
```

Selection Algorithm Overview

- DAG traversed in topological order "bottom up" ensures operator always selected before any operands.
- TableGen patterns prioritized using heuristics:
  1. Prefer more complex match patterns.
  2. Prefer lower emitted instruction count.
  3. Prefer larger match pattern size.
  4. Prefer later source order.

Phases of SelectionDAG



Instruction Scheduling

- Input - DAG of machine nodes.
- Output - A linear sequence of machine nodes.
- We aren't going to focus on this, because we don't have experience with it.
- Instead, check out:
  - Scheduling Model in LLVM - Part I (Blog)
  - Writing Great Machine Schedulers (2017 LLVM Developers' Meeting)

NVIDIA

Table of Contents

Example: Supporting the PTX mad instruction

- mad := Multiply two values and add a third value.
- Goal: Emit the PTX mad instruction via a peephole optimization.
- Lower latency and register pressure than a mul+add.

```
define i32 @foo(i32, i32, i32) {
  %mul = mul i32 %0, %1
  %add = add i32 %mul, %2
  ret i32 %add
}
```



```
.visible .func (...)  foo(...)  {
  ld.param.u32 %r1, [foo_param_0];
  ld.param.u32 %r2, [foo_param_1];
  ld.param.u32 %r3, [foo_param_2];
  mad.lo.s32   %r4, %r1, %r2, %r3;
  st.param.b32 [func_retval0+0], %r4;
  ret;
}
```

Scoping Things Out

- Observations: We don't need to modify type or operation legalization.
  - `mad` only supports existing legal types.
  - The nodes we're looking for, `ISD::ADD` and `ISD::MUL` are legal for all the types that the PTX `mad` instruction supports.

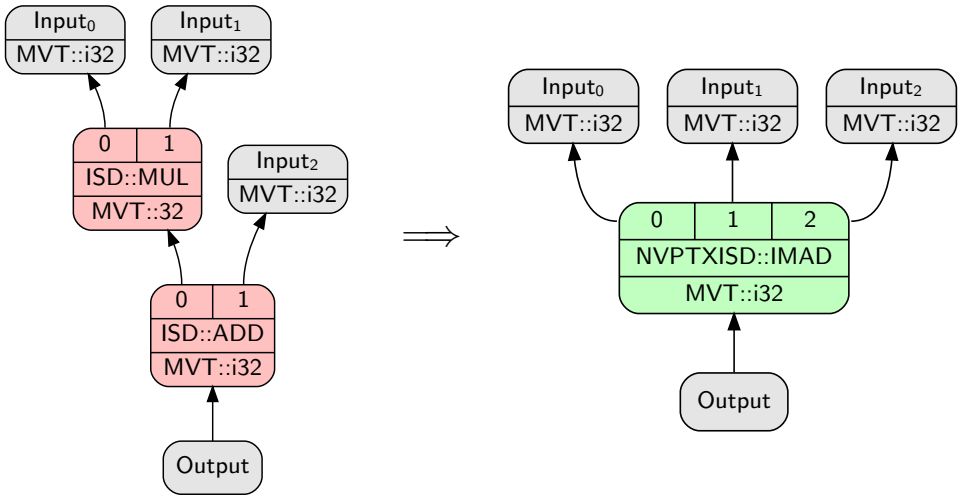
Scoping Things Out

- Observations: We don't need to modify type or operation legalization.
  - `mad` only supports existing legal types.
  - The nodes we're looking for, `ISD::ADD` and `ISD::MUL` are legal for all the types that the PTX `mad` instruction supports.
- We need a target-specific DAG combine to generate a `NVPTXISD::IMAD` SDNode.
- We need instruction selection logic to lower the `NVPTXISD::IMAD` SDNode to a machine node.

Tasks

1. **DAG Combine - Target-specific folding to generate** `NVPTXISD::IMAD`**.**
2. Instruction Selection - Lower `NVPTXISD::IMAD` into a machine node.

DAG Combine


Input0
MVT::i32
Input1
MVT::i32
0
1
ISD::MUL
MVT::32
Input2
MVT::i32
0
1
ISD::ADD
MVT::i32
Output
Input0
MVT::i32
Input1
MVT::i32
Input2
MVT::i32
0
1
2
NVPTXISD::IMAD
MVT::i32
Output

DAG Combine

Declare our target-specific DAG combine.

1.
```
NVPTXTargetLowering::NVPTXTargetLowering() {
  ...
  setTargetDAGCombine(ISD::ADD);
  ...
}
```

DAG Combine

Override `PerformDAGCombine()`

1.
```
NVPTXTargetLowering::NVPTXTargetLowering() {
  setTargetDAGCombine(ISD::ADD);
```

2.
```
SDValue NVPTXTargetLowering::PerformDAGCombine(SDNode *N) {
  switch (N->getOpcode()) {
    case ISD::ADD: return PerformADDCombine(N);
    ...
  }
}
```

DAG Combine

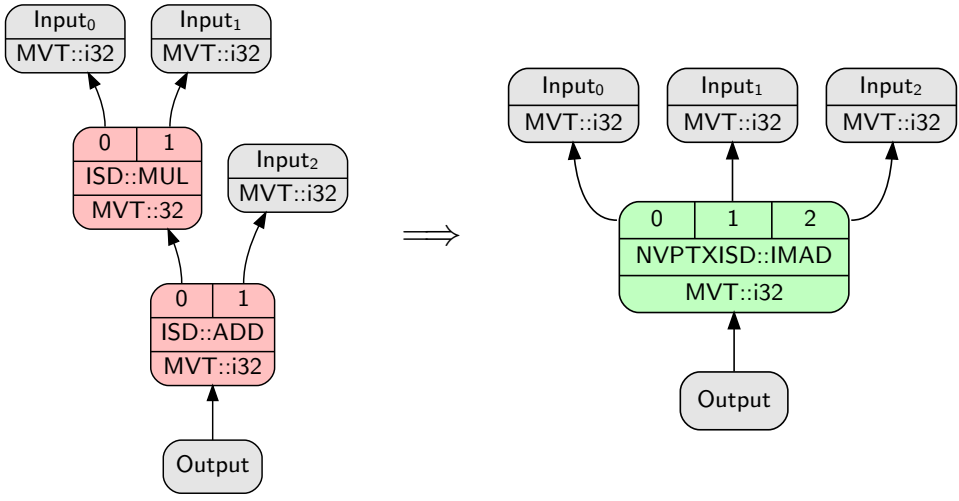
Implement the target-specific DAG combine.

1.
```
NVPTXTargetLowering::NVPTXTargetLowering() {
  setTargetDAGCombine(ISD::ADD);
```

2.
```
SDValue NVPTXTargetLowering::PerformDAGCombine(SDNode *N) {
  switch (N->getOpcode()) {
    case ISD::ADD: return PerformADDCombine(N);
```

3.
```
SDValue PerformADDCombine(SDNode *N) {
  if (N->getOperand(0).getOpcode() != ISD::MUL) return SDValue();
  if (N->getValueType()            != MVT::i32) return SDValue();
  return DAG.getNode(NVPTXISD::IMAD, N->getValueType(),
                     N->getOperand(0).getOperand(0),
                     N->getOperand(0).getOperand(1),
                     N->getOperand(1));
```

PerformADDCombine() - Input/Output DAGs



Tasks

1. DAG Combine - Target-specific folding to generate `NVPTXISD::IMAD`.
2. **Instruction Selection - Lower `NVPTXISD::IMAD` into a machine node.**

NVIDIA

Declare the target-specific SDNode & machine node.

```
def SDTIMAD : SDTypeProfile<1, 3,
    [SDTCisSameAs<0, 1>,
     SDTCisSameAs<0, 2>,
     SDTCisSameAs<0, 3>,
     SDTCisInt<0>]>;

def imad : SDNode<"NVPTXISD::IMAD", SDTIMAD>;
```

```
def MAD32rrr : NVPTXInst<
    (outs Int32Regs:$dst),
    (ins  Int32Regs:$a,
          Int32Regs:$b,
          Int32Regs:$c),
    "mad.lo.s32 \t$dst, $a, $b;">;
```

Select the machine node.

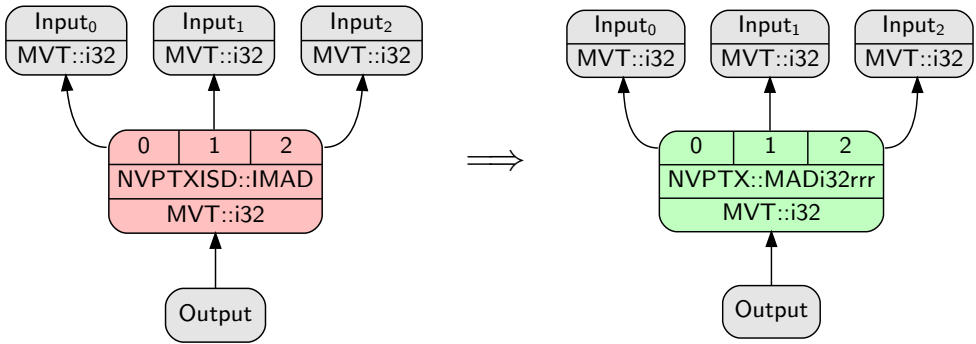
```
def imad : SDNode<...>
```

```
def MAD32rrr : NVPTXInst<...>
```

```
def : Pattern<
    (set Int32Regs:$dst,
      (imad (i32 Int32Regs:$a),
            (i32 Int32Regs:$b),
            (i32 Int32Regs:$c))),
    (MAD32rrr Int32Regs:$dst,
              Int32Regs:$a,
              Int32Regs:$b,
              Int32Regs:$c)>;
```

NVIDIA

mad Instruction Selection - Input/Output DAGs



Support the PTX mad instruction

- Goal: Emit the PTX mad instruction via a peephole optimization. ✓

```
define i32 @foo(i32, i32, i32) {
  %mul = mul i32 %0, %1
  %add = add i32 %mul, %2
  ret i32 %add
}
```



```
.visible .func (...)  foo(...)  {
  ld.param.u32 %r1, [foo_param_0];
  ld.param.u32 %r2, [foo_param_1];
  ld.param.u32 %r3, [foo_param_2];
  mad.lo.s32   %r4, %r1, %r2, %r3;
  st.param.b32 [func_retval0+0], %r4;
  ret;
}
```

Table of Contents

Other Resources

- The LLVM Target-Independent Code Generator (LLVM Docs)
- Instruction Selector (LLVM Docs)
- Legalizations in LLVM Backend (Blog)
- Building an LLVM Backend (2014 LLVM Developers' Meeting)
- CodeGen Overview and Focus on SelectionDAGs (2008 LLVM Developers' Meeting)

NVIDIA

Notes for the Road

- There are links throughout the slides to the relevant sections of the codebase.
  - justinfargnoli.github.io/slides.pdf
- `-debug-only=isel,legalize-types,legalizedag,dagcombine`
- Thank you to Akshay Deodhar, Princeton Ferro, Max Gutierrez, Drew Kersnar, Vladislav Malyshenko, and Kevin McAfee for your help in preparing this presentation.