Code Generation – III
Instruction Selection with GlobalISel

Nikhil Hegde

Compiler Optimizations in LLVM

Lecture series @ QUALCOMM Inc.

Recap: Compiler phases so far..

$$\begin{tikzcd}
\boxed{\text{AST}} \arrow[r, Rightarrow] & \boxed{\text{IR}} \arrow[r, Rightarrow] & \boxed{\text{Code generation for a target machine}}
\end{tikzcd}$$

Analysis, Pattern matching	Machine independent analysis, optimizations	Machine dependent analysis and optimizations

Recap: Instruction Selection with SelectionDAG

1. Build Initial DAG
2. Optimize Initial DAG
3. Legalize SelectionDAG Types
4. Optimize Selection DAG
5. Legalize SelectionDAG Operations
6. Optimize SelectionDAG
7. Select Instructions from SelectionDAG (also called DAG to DAG)
8. Selection DAG scheduling and Formation

Last two classes: what each step does and how is it done

This class: Instruction Selection with GlobalISel

Global Instruction Selection / GlobalISel

- Framework for Instruction Selection
- Intended to be a replacement for SelectionDAG
- Built with code reuse in mind
- Allows for target specific testing using the Machine IR, a representation used by GlobalISel

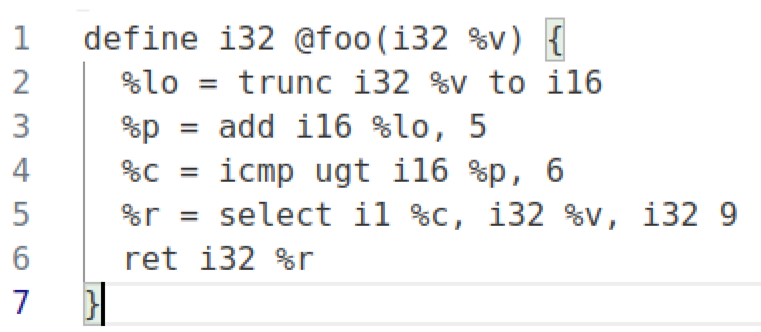
Motivation for GlobalISel

- Selection DAG has its own intermediate representation, the DAG format. Preparation and manipulation of DAG has a compile time cost.
- SelectionDAG operates on basic-block and may loose global optimization opportunities.
- The Type legalizer and Operation legalizer in SelectionDAG are dependent on each other. Hence, why not merge these modules?
- SelectionDAG allows for little reuse of code across phases.

Recap: Legalize Type

Before

```
Optimized lowered selection DAG: %bb.0 'foo:'
SelectionDAG has 15 nodes:
  t0: ch,glue = EntryToken
  t2: i32,ch = CopyFromReg t0, Register:i32 %0
              t3: i16 = truncate t2
            t5: i16 = add t3, Constant:i16<5>
          t8: i1 = setcc t5, Constant:i16<6>, setugt:ch
        t10: i32 = select t8, t2, Constant:i32<9>
  t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
  t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```



```
define i32 @foo(i32 %v) {
  %lo = trunc i32 %v to i16
  %p = add i16 %lo, 5
  %c = icmp ugt i16 %p, 6
  %r = select i1 %c, i32 %v, i32 9
  ret i32 %r
}
```

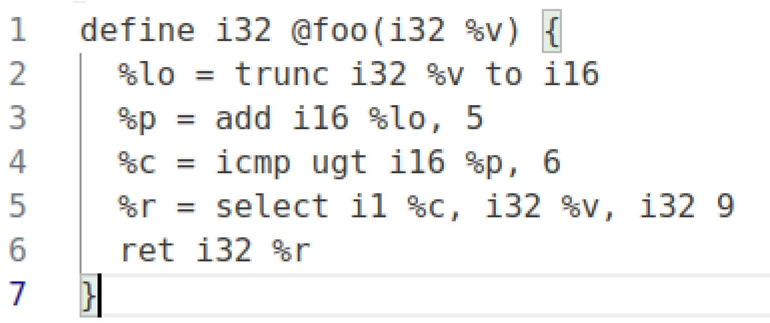
After

```
Type-legalized selection DAG: %bb.0 'foo:'
SelectionDAG has 17 nodes:
  t0: ch,glue = EntryToken
  t2: i32,ch = CopyFromReg t0, Register:i32 %0
              t3: i16 = truncate t2
            t5: i16 = add t3, Constant:i16<5>
            t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
          t18: i8 = and t15, Constant:i8<1>
        t10: i32 = select t18, t2, Constant:i32<9>
  t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
  t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

i1 is not supported natively. Hence, it is promoted to i8 and a mask (0x1) is used to obtain the result.

Recap:Legalize Operation

```
1.  Type-legalized selection DAG: %bb.0 'foo:'
2.  SelectionDAG has 17 nodes:
3.     t0: ch,glue = EntryToken
4.     t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                   t3: i16 = truncate t2
6.                t5: i16 = add t3, Constant:i16<5>
7.             t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
8.          t18: i8 = and t15, Constant:i8<1>
9.       t10: i32 = select t18, t2, Constant:i32<9>
10.    t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
11.    t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```



```
define i32 @foo(i32 %v) {
  %lo = trunc i32 %v to i16
  %p = add i16 %lo, 5
  %c = icmp ugt i16 %p, 6
  %r = select i1 %c, i32 %v, i32 9
  ret i32 %r
}
```

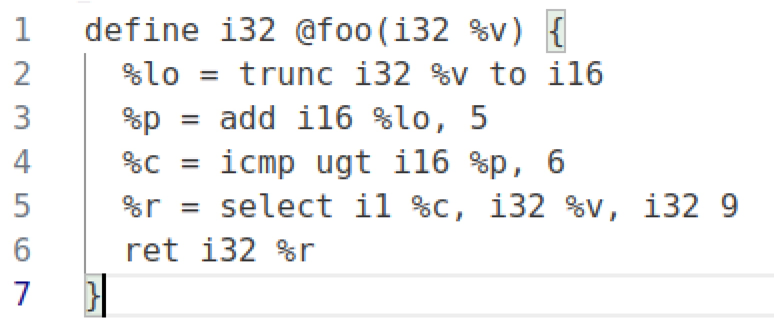
Before: x>6? t2 : 9

```
1.  Legalized selection DAG: %bb.0 'foo:'
2.  SelectionDAG has 15 nodes:
3.     t0: ch,glue = EntryToken
4.     t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                     t3: i16 = truncate t2
6.                 t5: i16 = add t3, Constant:i16<5>
7.             t20: i16,i32 = X86ISD::SUB t5, Constant:i16<7>
8.          t23: i32 = X86ISD::CMOV Constant:i32<9>, t2, TargetConstant:i8<3>, t20:1
9.     t13: ch,glue = CopyToReg t0, Register:i32 $eax, t23
10.    t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

After: combination of SUB and conditional MOV instructions to avoid branching.
Equivalent to: x-7<0? 9 : t2

Recap:Legalize Operation (RISCV32)

```
Optimized lowered selection DAG: %bb.0 'foo:'
SelectionDAG has 14 nodes:
  t0: ch,glue = EntryToken
  t2: i32,ch = CopyFromReg t0, Register:i32 %0
          t3: i16 = truncate t2
        t5: i16 = add t3, Constant:i16<5>
      t8: i1 = setcc t5, Constant:i16<6>, setugt:ch
    t10: i32 = select t8, t2, Constant:i32<9>
  t12: ch,glue = CopyToReg t0, Register:i32 $x10, t10
  t13: ch = RISCVISD::RET_GLUE t12, Register:i32 $x10, t12:1
```



```
define i32 @foo(i32 %v) {
  %lo = trunc i32 %v to i16
  %p = add i16 %lo, 5
  %c = icmp ugt i16 %p, 6
  %r = select i1 %c, i32 %v, i32 9
  ret i32 %r
}
```

Before: operations on i16; t3+5

```
Type-legalized selection DAG: %bb.0 'foo:'
SelectionDAG has 17 nodes:
  t0: ch,glue = EntryToken
  t2: i32,ch = CopyFromReg t0, Register:i32 %0
          t16: i32 = add t2, Constant:i32<5>
        t22: i32 = and t16, Constant:i32<65535>
      t17: i32 = setcc t22, Constant:i32<6>, setugt:ch
    t20: i32 = and t17, Constant:i32<1>
  t10: i32 = select t20, t2, Constant:i32<9>
  t12: ch,glue = CopyToReg t0, Register:i32 $x10, t10
  t13: ch = RISCVISD::RET_GLUE t12, Register:i32 $x10, t12:1
```

After: operations on i16 not supported on RISCV32. So, the operands of add are promoted to i32 and a mask is applied to the result to get the expected precision; (t2+5)&0xFFFF

More Motivation for GlobalISel

- Streamline SelectionDAG legalizer actions : Type legalization and Operation legalization are intertwined.

```
else if (Subtarget.is64Bit()) {
    setOperationAction({ISD::SDIV, ISD::UDIV, ISD::UREM},
                       {MVT::i8, MVT::i16, MVT::i32}, Custom);
  }
```

  - Can specify illegal types as the second argument of setOperationAction.
  - When type legalizer sees illegal types, it consults the operation legalizer to check if there is a custom lowering defined for the operation with illegal type operands.

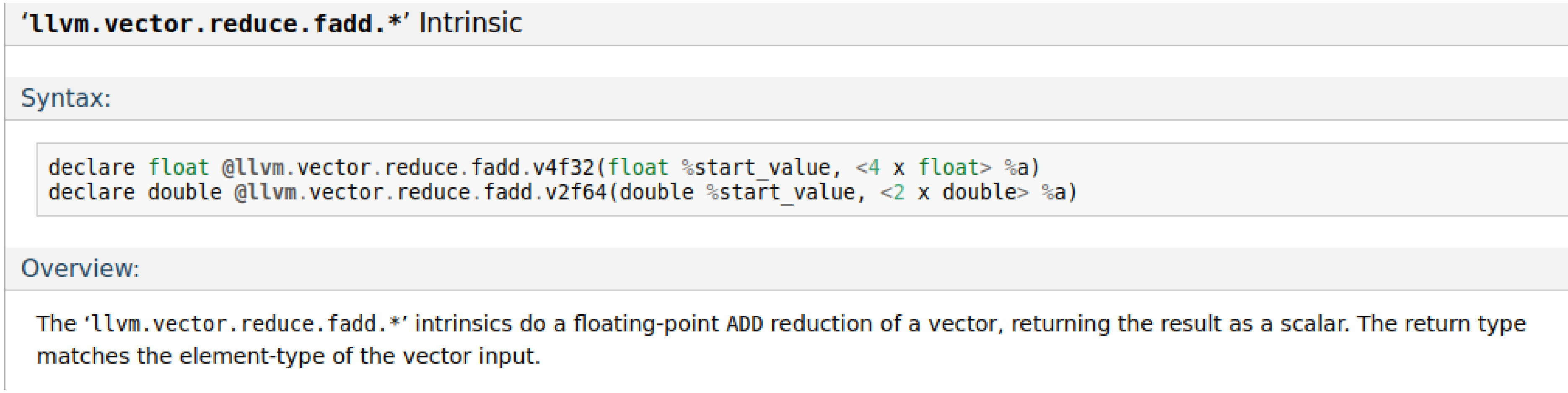
    E.g. ADDW instruction in RISCV64 takes two 32bit operands, widens to 64 bit by sign-extending and adds. If type legalizer had not consulted op legalizer, we could not have chosen ADDW, and the the original ADD operation on 32 bit operands would have triggered type legalizer to perform two sign-extension operations (one for each operand)
  - Does the second argument refer to operands or results of an operation? Not clear.

More Motivation for GlobalISel

- Streamline SelectionDAG legalizer actions
  - E.g. Vector reduction and documentation
  - What should be the type of second argument for vector reduce?

setOperationAction(ISD::VECREDUCE_FADD, **???**, Custom);

declare float @llvm.vector.reduce.fadd.v4f32(float %start_value, <4 x float> %v)



'`llvm.vector.reduce.fadd.*`' Intrinsic

Syntax:

```
declare float @llvm.vector.reduce.fadd.v4f32(float %start_value, <4 x float> %a)
declare double @llvm.vector.reduce.fadd.v2f64(double %start_value, <2 x double> %a)
```

Overview:

The '`llvm.vector.reduce.fadd.*`' intrinsics do a floating-point ADD reduction of a vector, returning the result as a scalar. The return type matches the element-type of the vector input.

SelectionDAG legalizer uses one of the operand types to check against the second argument of setOperationAction for determining the operation's legality. Which operand types to pick? it's predefined (in tblgen file or some implementation) and leads to incorrect interpretation.

Ambiguity resolution in GlobalISel Legalizer

```
getActionDefinitionsBuilder({G_ADD, G_SUB, G_AND, G_OR,
G_XOR})
    .legalFor({s32, sXLen})
    .legalIf(typeIsLegalIntOrFPVec(0, IntOrFPVecTys, ST))
    .widenScalarToNextPow2(0)
    .clampScalar(0, s32, sXLen);
getActionDefinitionsBuilder({G_ADD, G_SUB})
.legalFor({XLenLLT})
.customFor({s32})
.clampScalar(0, XLenLLT, XLenLLT);
```

- `G_ADD` and `G_SUB` are generic opcodes. Target specific ones could be ADDWrr, addl, SUBWrr.
- `XLenLLT`(List of Legal Low-level Type). e.g., s64 on x86
- `clampScalar(OperandIndex, MinType, MaxType)` : If the scalar size at operand index 0 is *smaller than MinType,* widen it. If it's *bigger than MaxType,* narrow it.

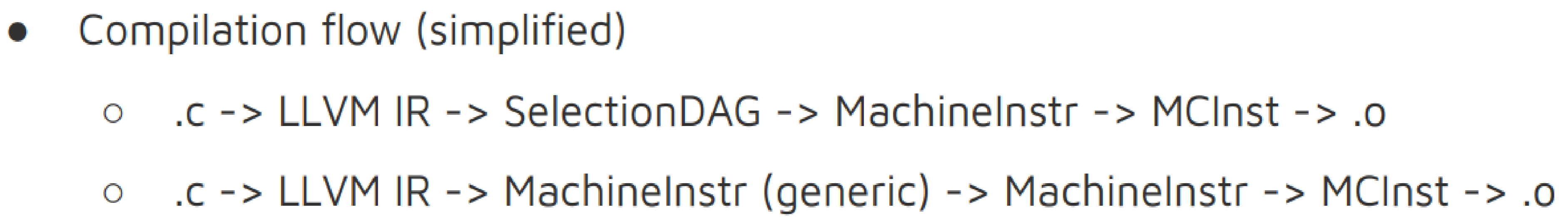
GlobalISel Pipeline

1. IR Translator (LLVM IR to Machine IR/MIR - **g**eneric **MIR**)
2. Legalizer (legalize ops and types in MIR)
3. Regbank select (choose target registers as per the grouping)
4. Instruction select (choose target instructions corresponding to gMIR instructions)

These are the core passes. Implementing these (for a custom target) requires implementing interfaces:

1. `CallLowering`
2. `LegalizerInfo`
3. `RegBankInfo`
4. `InstructionSelector`

GlobalISel Pipeline vs Selection DAG pipeline



- Compilation flow (simplified)
  - .c -> LLVM IR -> SelectionDAG -> MachineInstr -> MCInst -> .o
  - .c -> LLVM IR -> MachineInstr (generic) -> MachineInstr -> MCInst -> .o

courtesy: https://llvm.org/devmtg/2023-05/slides/Tutorial-May11/01-Bradbury-GlobalISelTutorial.pdf

LLVM IR to Generic Machine IR

- Registers
- Virtual registers with constraints -
  - Register banks
  - Class of registers
  - Data type
- Opcodes

```
define float @foo(float %a, float %b){
entry:
 %div = fdiv float %a, %b
 ret float %div
}
```

```
%0:_(s32) = COPY $xmm0
%1:_(s32) = COPY $xmm1
%2:_(s32) = G_FDIV %0, %1
$xmm0 = COPY %2(s32) RET 0,
implicit $xmm0
```

Demo