Moving LLVM's code generator to MLIR framework

Vinay/Ranjith/Prashantha

Compiler Tree Technologies

Agenda

- Motivation
- Our approach
- LLVM representation
- MIR dialect details
- Various Lowerings
- Codegen details
- Our approach for codegen
- Examples

Motivation

- MLIR(Multi-Level IR) is generic enough to represent IRs at any abstraction
- IRs at different abstraction levels can co-exist together in MLIR.
- Maturing at fast pace to subsume the optimization capability of LLVM OPT phase.
    - LLVM Dialect : Bridge between LLVM and MLIR
    - Optimizations can be ported easily (similar data structures)

Motivation

- Optimizations like vectorization bring out the need for vector dialects like AVX-512, NEON.
  - There are so many of them MMX, SSE, SVE, MVE, AMX, etc
  - Needs maintenance -- Similar to clang/lib/Headers
  - Readily available in .td files
- To circumvent “lost in translation” problem, there is a need for direct translation from MLIR to Codegen MIR.
  - Example: Patterns generated in LLVM Vectorizer are changed during lowering (DAG optimizations, etc) without the bigger context

Our Approach

- Prototyped MIR dialect in MLIR to represent the Target specific assembly.
- Using existing ***llvm/lib/MC/**** data structures and the **existing *.td files**
- Ported the required **GlobalIsel** passes to MLIR
- Ported few passes of X86 Target from **llvm/lib/Target/X86**
- we could generate **X86 assembly from subset of std dialect operations** via LLVM dialect.

Our Approach

- MIR Dialect be used along with other dialects (like, std, affine, LLVM, etc)
- Multiple Targets can co-exist together via different Dialects: X86Dialect, NVPTXDialect
- Enables access to Target vector instructions defined in LLVM .td files
- Generates assembly for basic programs.

How are Machine Instructions represented in LLVM?

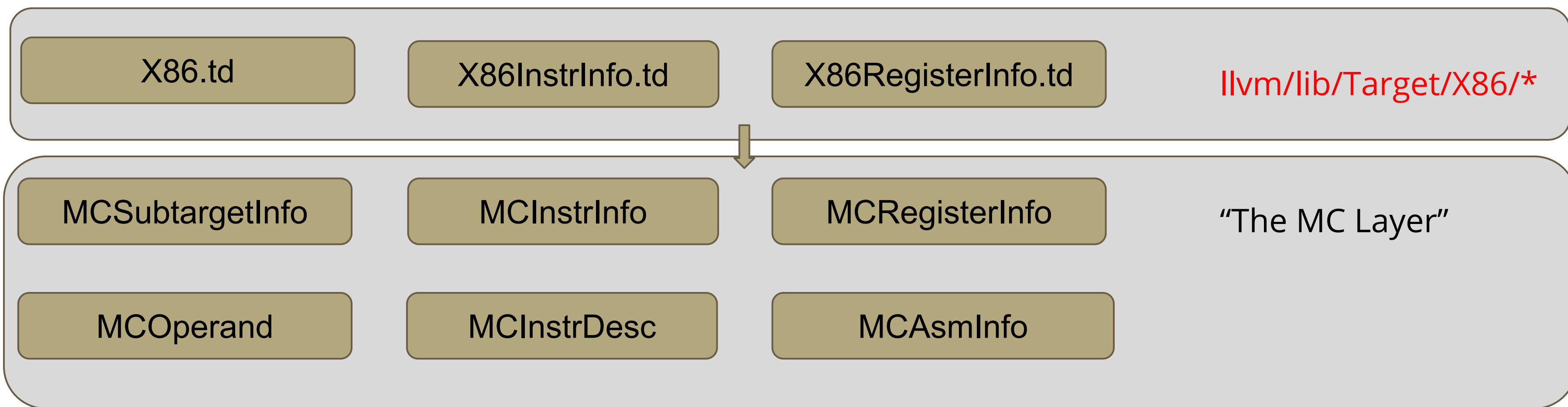
X86.td

X86InstrInfo.td

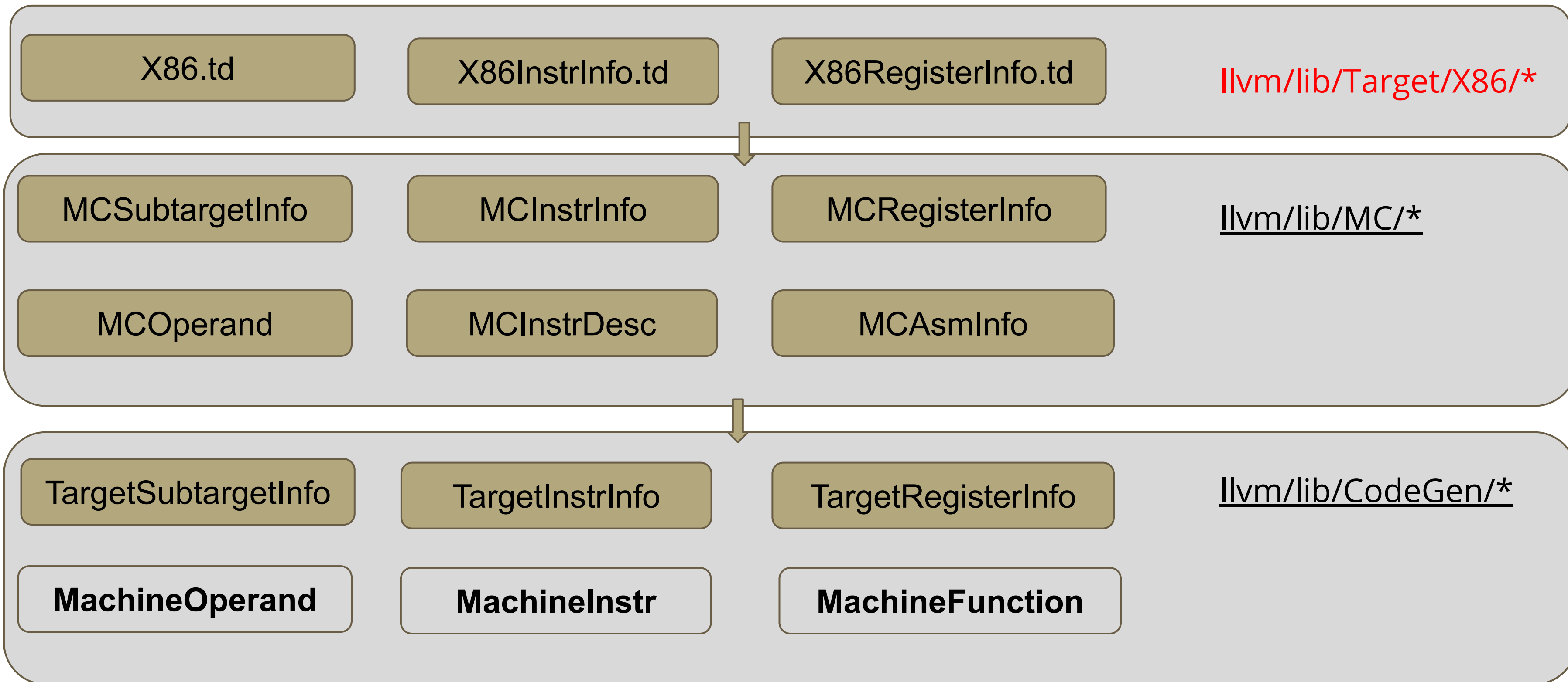
X86RegisterInfo.td

llvm/lib/Target/X86/*

- Contains all the specification of the target in TableGen program



- Target independent data structures
- Contains all the information
- No reference to MIR data structures
- Very low level details : section, labels, etc.



- Target independent
- Has dependency on MIR
- Base class for classes defined in Tablegen generated files

X86.td
X86InstrInfo.td
X86RegisterInfo.td
llvm/lib/Target/X86/*
MCSubtargetInfo
MCInstrInfo
MCRegisterInfo
llvm/lib/MC/*
MCOperand
MCInstrDesc
MCAsmInfo
TargetSubtargetInfo
TargetInstrInfo
TargetRegisterInfo
llvm/lib/CodeGen/*
MachineOperand
MachineInstr
MachineFunction
X86Subtarget
X86InstrInfo
X86RegisterInfo
llvm/lib/Target/X86/*

Machine IR (MIR)

- Target agnostic Machine Intermediate representation
- Originally Intended for testing the code generation passes in LLVM.
- Designed to support both an SSA representation for machine code, as well as a register allocated, non-SSA form.
- IR structure
    - **MachineInstr**: opcode with list of operands
    - **MachineOperand**: Register, Constant, GlobalValue, Memory, etc
    - **MachineBasicBlock**: List of MachineInstr
    - **MachineFunction** :
        - List of basic blocks
        - One-to-one correspondence with llvm::Function
        - Contains MachineFrameInfo, etc
- **Has dependency on LLVM IR**

How to move LLVM Target to MLIR?

Approach 1: Tool to generate MLIR Dialect for each Target

- **Input:** Load all the target related data structures using llvm/MC APIs
- **Output:** Generate corresponding MLIR operations
- **Conversion**
  - MachineInstruction -> mlir::Operation
  - Register classes and Types -> mlir::Type
- **Issue:**
  - **There are too many instructions (~15K for X86 target)**
  - Running above tool for X86 target generated **30 lakh lines of .inc files** in MLIR

- This approach doesn't scale well
- Quickest way to get access for the smaller targets

Approach 2: Build Dialect around the MC layer

- Build target independent Dialect / Dialect interface
- Extend it for different targets
- Common set of operations: accept Instruction type as enum
- Multiple targets (dialects) can be loaded together
- **Re-uses the existing .td files from LLVM**
  - No extra maintenance for target dialects

MIR Dialect

- A new target independent MLIR dialect built as wrapper around "MC Layer"
- Generic mlir::Operation wich extends corresponding traits
  - **MachineOp**, **MachineCallOp**, **MachineTerminatorOp,etc**
- "**MachineInstr**" OpInterface is extended by all the operations
- Custom types for Machine Operands
- Registers types are wrapper around llvm/MC/MCRegister.h
- mlir::Operation contains Opcode Attribute
- Some of the APIs from llvm::MachineInstr are ported to MachineInstr interface.
- We tried to retain the interfaces similar to llvm/lib/CodeGen/* wherever possible.

Various MIR Dialect Lowerings

1. Generate assembly by porting backend passes (Harder path!)

Various MIR Dialect Lowerings

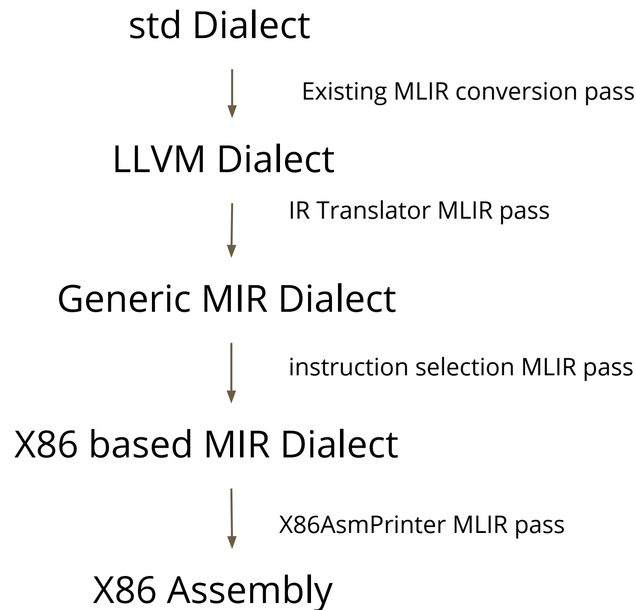
1. Generate assembly by porting backend passes (Harder path!)
2. Generating corresponding LLVM intrinsic
    - Doesn’t work for all instructions
    - May not be accurate (LLVM may optimize)

Various MIR Dialect Lowerings

1. Generate assembly by porting backend passes (Harder path!)
2. Generating corresponding LLVM intrinsic
    - Doesn’t work for all instructions
    - May not be accurate (LLVM may optimize)
3. Generating LLVM inline assembly via LLVM Dialect
    - No support for inline assembly in LLVM IR dialect

Lowering to Assembly

MIR dialect Code Generation (std --> X86 example)



Existing LLVM Code Generator phases

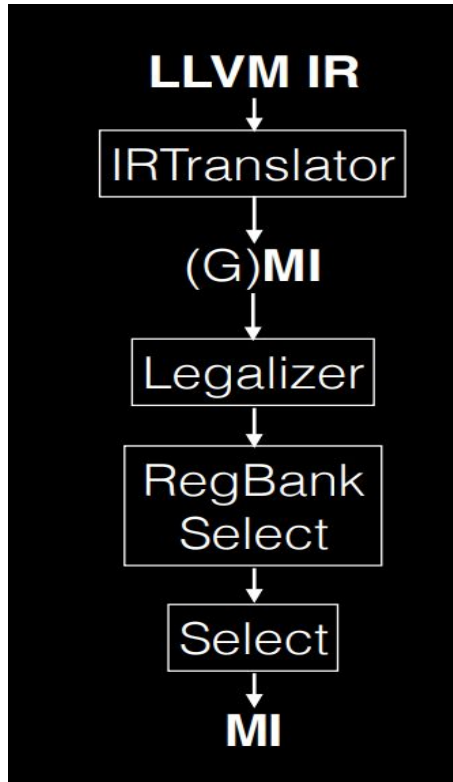
- Instruction selection (LLVM IR to Machine IR)
- Instruction scheduling
- SSA based Machine IR optimizations
- Register Allocation
- Late Machine Code Optimizations
- Code emission

Existing LLVM Code Generator phases

- Instruction selection (LLVM IR to Machine IR)
- Instruction scheduling
- SSA based Machine IR optimizations
- Register Allocation
- Late Machine Code Optimizations
- Code emission

Except Instruction selection rest of the phases can work on Machine IR (MIR)

LLVM GlobalISel Phases:



Converts LLVM IR to Generic MIR

Generic MIR (independent of any target)

Legalizing the Generic MIR for the given target

Selecting the register classes for virtual registers, etc

Instruction Selection: Converts Generic MIR to Target specific MIR

Reference: https://llvm.org/devmtg/2015-10/slides/Colombet-GlobalInstructionSelection.pdf

Porting llvm/lib/CodeGen/*

- We have ported the following GlobelIsel Passes:
  - IRTranslator
    - Works for basic translation
  - InstructionSelect
    - Including X86InstructionSelector and generic InstructionSelector classes
- We are yet to port:
  - RegBankSelect
    - register classes are hard-coded for now
  - Legalizer
- AsmPrinter and related utilities
- Porting Builders for compatibility : MachineInstrBuilder , MachineIRBuilder

Porting X86 Target

- Instruction printing and Selection related classes.
    - X86AsmPrinter
    - X86MCInstrLower
    - X86InstructionSelector

Example: std dialect to Generic MIR dialect conversion

```
module  {
  func @main() -> i32 {
    %c10_i32 = constant 10 : i32
    return %c10_i32 : i32
  }
}
```

```
module attributes {llvm.data_layout = ".."}  {
  func @main() -> i32 {
    %0 = mir.G_CONSTANT i32 10
    mir.RET %0
  }
}
```

Example: Generic MIR to X86 specific MIR conversion

```
module attributes {llvm.data_layout = ".."}  {
  func @main() -> i32 {
    %0 = mir.G_CONSTANT i32 10
    mir.RET %0
  }
}
```

```
module  {
  func @main() -> i32 {
    %0 = x86.MOV32ri 10
    x86.RETQ %0
    "mir.return"() : () -> ()
  }
}
```

Challenges in porting CodeGen(1): LLVM dependency

- LLVM backend relies heavily on LLVM IR data structures
  - Example: GlobalValue
- MIR isn't a complete representation
- We have created the following dummy LLVM data structures to ease the porting
  - MachineMemOperand, MachineFunction, MachineOperand, etc.
- Dummy LLVM module and functions are used to overcome the LLVM IR usage.

Challenges in porting CodeGen (2)

- Porting SelectionDAG and related passes
  - Currently using GlobalIsel
  - Includes custom lowering in llvm/lib/Target/*
  - Too much code!
- Each target related classes should be added manually

Using Target instructions as MLIR dialect operations

```
module  {
 func @add_10(%0: i32 ) -> i32 {
  %c10_i32 = constant 10 : i32
   %1 = x86.cast_to_reg %0 : x86.reg<EAX>
   %2 = x86.MOV32ri 10 :  x86.reg<GR32>
   %3 = x86.ADD32rr %1, %2
   %4 = x86.cast_to_std %3
  return %4 : i32
 }
}
```

Using Target instructions as MLIR dialect operations

```
module {
 func @add_10(%0: i32 ) -> i32 {
   %c10_i32 = constant 10 : i32
    %1 = x86.cast_to_reg %0 : x86.reg<EAX>
    %2 = x86.MOV32ri 10 :  x86.reg<GR32>
    %3 = x86.ADD32rr %1, %2
    %4 = x86.cast_to_std %3
   return %4 : i32
 }
}
```

Cast to Physical or Virtual Register Type

Using Target instructions as MLIR dialect operations

```
module  {
 func @add_10(%0: i32 ) -> i32 {
   %c10_i32 = constant 10 : i32
    %1 = x86.cast_to_reg %0 : x86.reg<EAX>
    %2 = x86.MOV32ri 10 :  x86.reg<GR32>
    %3 = x86.ADD32rr %1, %2
    %4 = x86.cast_to_std %3
   return %4 : i32
 }
}
```

X86 instructions

Using Target instructions as MLIR dialect operations

```
module  {
  func @add_10(%0: i32 ) -> i32 {
    %c10_i32 = constant 10 : i32
     %1 = x86.cast_to_reg %0 : x86.reg<EAX>
     %2 = x86.MOV32ri 10 :  x86.reg<GR32>
     %3 = x86.ADD32rr %1, %2
     %4 = x86.cast_to_std %3
    return %4 : i32
  }
}
```

→ Convert back to std types

Use Vector instructions

```
affine.parallel %i0 = 0 to 10 {
  affine.parallel %i1 = 0 to 1000 step 2 {
    ....
    %10 = x86.VFMADD132SD %4, %5, %8 :
              x86.Reg<VR128>
    ...
  }
}
```

- Can be used in place of AVX-512 and Neon Dialects
- Different lowerings : intrinsics / assembly / inline assembly, etc

Other interesting possibilities!

```
{
  ....
  %10 = x86.VFMADD132SD %4, %5, %8 :
              x86.Reg<VR128>
},
{
  %11 = NVPTX.ADDRi %10,
  ....
 }
```

- Compile multiple targets together!
    - Share SSA values!
- Target Dialect conversions!
    - Let's say, X86 ⇔ AMDGPU or X86 ⇔ AArch64

Next steps

- Try to emit LLVM intrinsics for various Instructions (Example, vector operations)
- Porting missing passes like Scheduler, Register Allocator, Frame Lowering ,etc...
- Completely port the X86 target
- Porting of lit tests from llc
- Remove LLVM IR dependency!!

Thank You