Using LLVM and MLIR

Quinn Pham
Adapted from slides by Braedy Kuzma, Caio Salvador Rohwedder, and Nelson Amaral

Building

Your Machine

2.11. Installing MLIR

In the VCalc assignment and your final project you will be working with MLIR and LLVM. Due to the complex nature (and size) of MLIR we did not want to include it as a subproject. In fact, you may even want to defer the installation until you're about to start your assignment. Here are the steps to get MLIR up and running.

1. Checkout LLVM to your machine

```
$ cd $HOME
$ git clone https://github.com/llvm/llvm-project.git
$ cd llvm-project
$ git checkout llvmorg-16.0.6
```

2. Build MLIR (more details are available here)

```
$ mkdir build
$ cd build
$ cmake -G Ninja ../llvm \
    -DLLVM_ENABLE_PROJECTS=mlir \
    -DLLVM_BUILD_EXAMPLES=ON \
    -DLLVM_TARGETS_TO_BUILD="Native" \
    -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_ENABLE_ASSERTIONS=ON
$ ninja check-all -j<number of threads>
```

3. Add these configuration lines to your `~/.bashrc` file so that you can use the MLIR tools and so that `cmake` will find your build.

```
export MLIR_INS="$HOME/llvm-project/build/"
export MLIR_DIR="$MLIR_INS/lib/cmake/mlir/" # Don't change me.
export PATH="$MLIR_INS/bin:$PATH" # Don't change me
```

https://cmput415.github.io/415-docs/setup/index.html

Lab Machines

5.1. Using Prebuilt Resources

Set up on the CSC machines is a lot simpler because all of the resources are managed for you. Therefore, all you need to do is to add the provided definitions to your `~/.bashrc`.

```
# C415 Predefinitions
source "/cshome/cmput415/415-resources/415env.sh"
```

This should enable you to build manually using the command line. You should log out and back in so that the changes can take effect.

https://cmput415.github.io/415-docs/setup/index.html

MLIR

MLIR Language Reference

MLIR (Multi-Level IR) is a compiler intermediate representation with similarities to traditional three-address SSA representations (like LLVM IR or SIL), but which introduces notions from polyhedral loop optimization as first-class concepts. This hybrid design is optimized to represent, analyze, and transform high level dataflow graphs as well as target-specific code generated for high performance data parallel systems. Beyond its representational capabilities, its single continuous design provides a framework to lower from dataflow graphs to high-performance target-specific code.

This document defines and describes the key concepts in MLIR, and is intended to be a dry reference document - the rationale documentation, glossary, and other content are hosted elsewhere.

MLIR is designed to be used in three different forms: a human-readable textual form suitable for debugging, an in-memory form suitable for programmatic transformations and analysis, and a compact serialized form suitable for storage and transport. The different forms all describe the same semantic content. This document describes the human-readable textual form.

https://mlir.llvm.org/docs/LangRef/

MLIR Language Reference

MLIR (Multi-Level IR) is a compiler intermediate representation with similarities to traditional three-address SSA representations (like LLVM IR or SIL), but which introduces notions from polyhedral loop optimization as first-class concepts. This hybrid design is optimized to represent, analyze, and transform high level dataflow graphs as well as target-specific code generated for high performance data parallel systems. Beyond its representational capabilities, its single continuous design provides a framework to lower from dataflow graphs to high-performance target-specific code.

This document defines and describes the key concepts in MLIR, and is intended to be a dry reference document - the rationale documentation, glossary, and other content are hosted elsewhere.

MLIR is designed to be used in three different forms: a human-readable textual form suitable for debugging, an in-memory form suitable for programmatic transformations and analysis, and a compact serialized form suitable for storage and transport. The different forms all describe the same semantic content. This document describes the human-readable textual form.

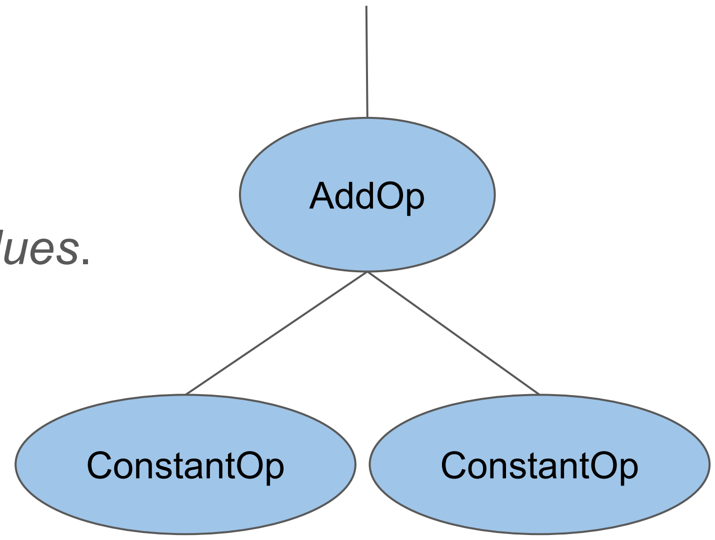
https://mlir.llvm.org/docs/LangRef/

High-level Structure of MLIR

https://mlir.llvm.org/docs/LangRef/#high-level-structure

High-level Structure of MLIR

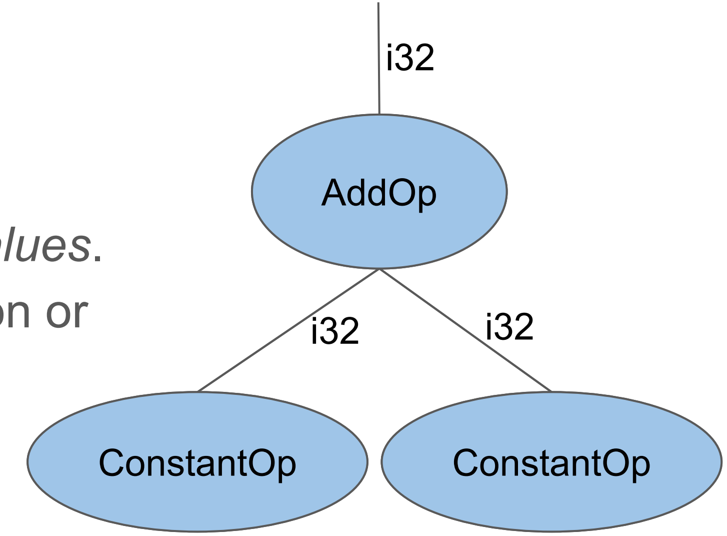
- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.



https://mlir.llvm.org/docs/LangRef/#high-level-structure

High-level Structure of MLIR

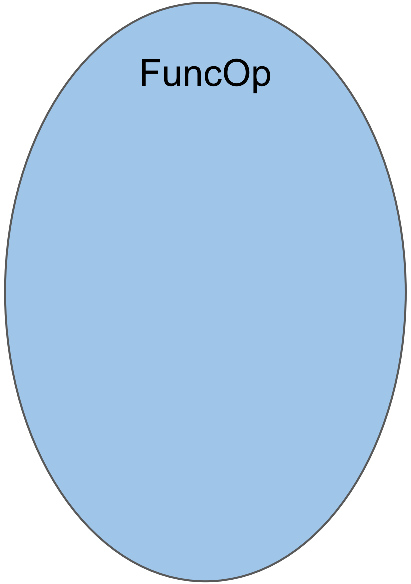
- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.
- Each Value is the result of exactly one Operation or Block Argument, and has a *Value Type*.



https://mlir.llvm.org/docs/LangRef/#high-level-structure

High-level Structure of MLIR

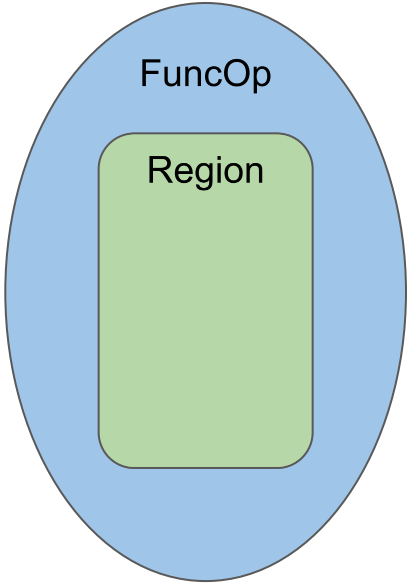
- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.
- Each Value is the result of exactly one Operation or Block Argument, and has a *Value Type*.



https://mlir.llvm.org/docs/LangRef/#high-level-structure

High-level Structure of MLIR

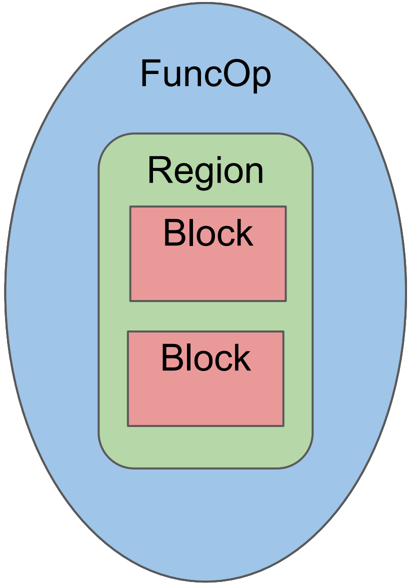
- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.
- Each Value is the result of exactly one Operation or Block Argument, and has a *Value Type*.
- Operations contain Regions.



https://mlir.llvm.org/docs/LangRef/#high-level-structure

High-level Structure of MLIR

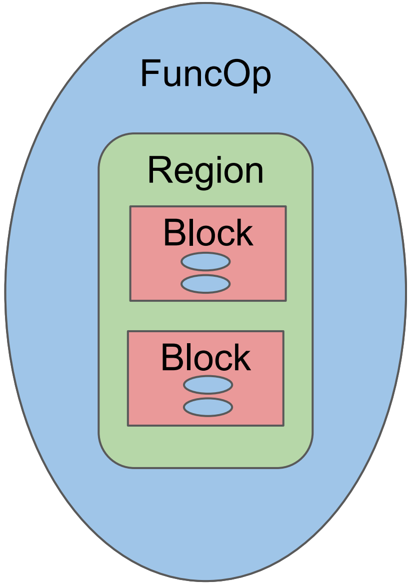
- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.
- Each Value is the result of exactly one Operation or Block Argument, and has a *Value Type*.
- Operations contain Regions.
- Regions contain Blocks.



https://mlir.llvm.org/docs/LangRef/#high-level-structure

High-level Structure of MLIR

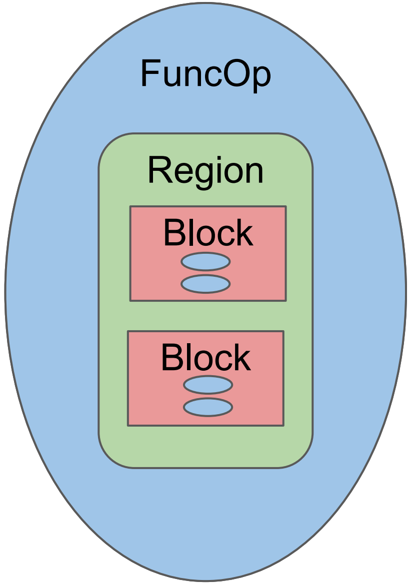
- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.
- Each Value is the result of exactly one Operation or Block Argument, and has a *Value Type*.
- Operations contain Regions.
- Regions contain Blocks.
- Blocks contain operations.



https://mlir.llvm.org/docs/LangRef/#high-level-structure

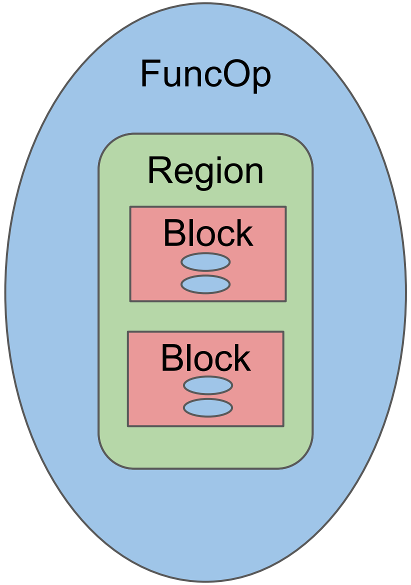
High-level Structure of MLIR

- MLIR is based on a graph-like data structure of nodes, called *Operations*, and edges, called *Values*.
- Each Value is the result of exactly one Operation or Block Argument, and has a *Value Type*.
- Operations contain Regions.
- Regions contain Blocks.
- Blocks contain operations.
- Operations are ordered within their containing block and Blocks are ordered in their containing region.



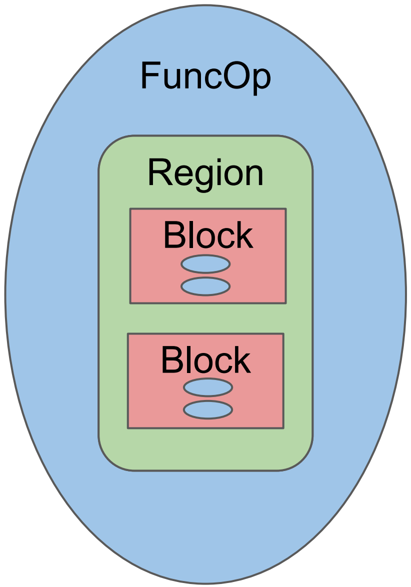
https://mlir.llvm.org/docs/LangRef/#high-level-structure

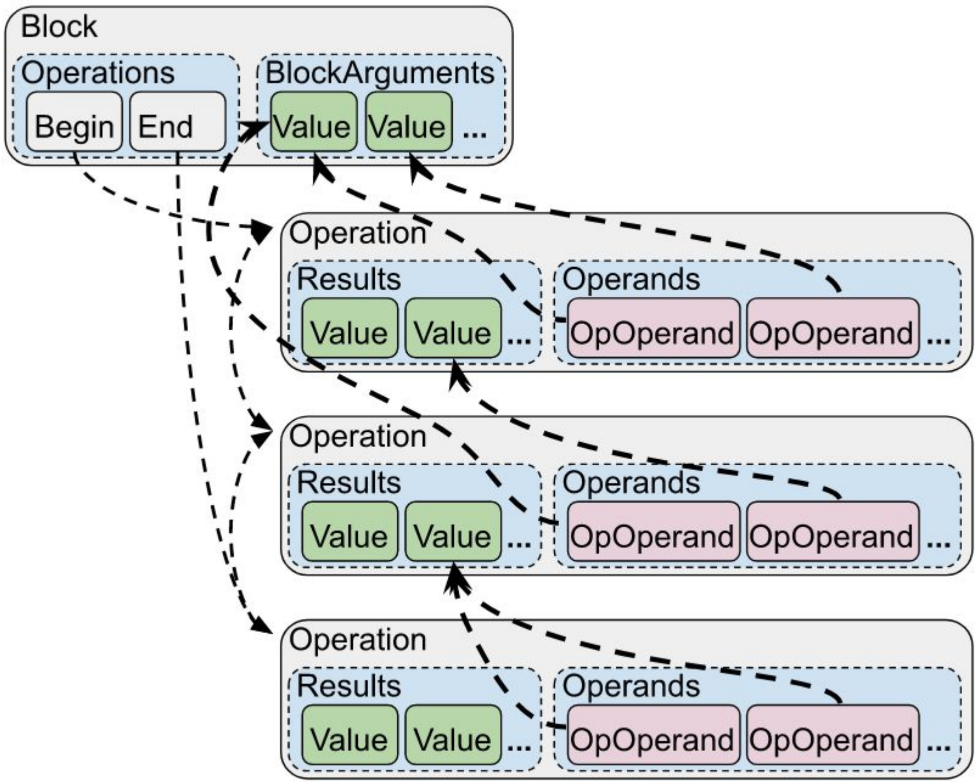
Operations in a Block



https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/

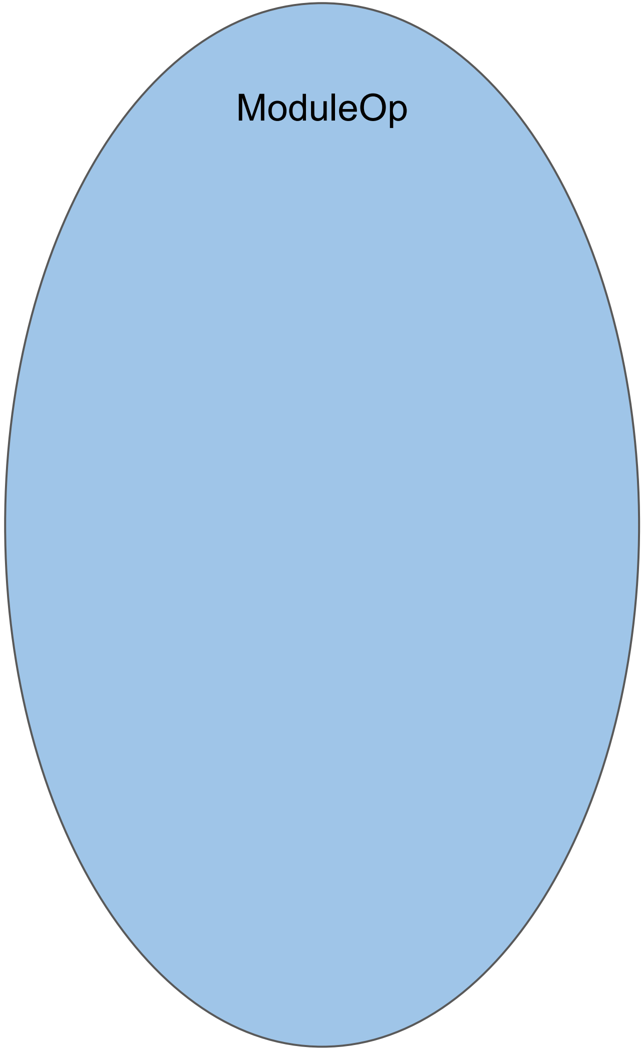
Operations in a Block



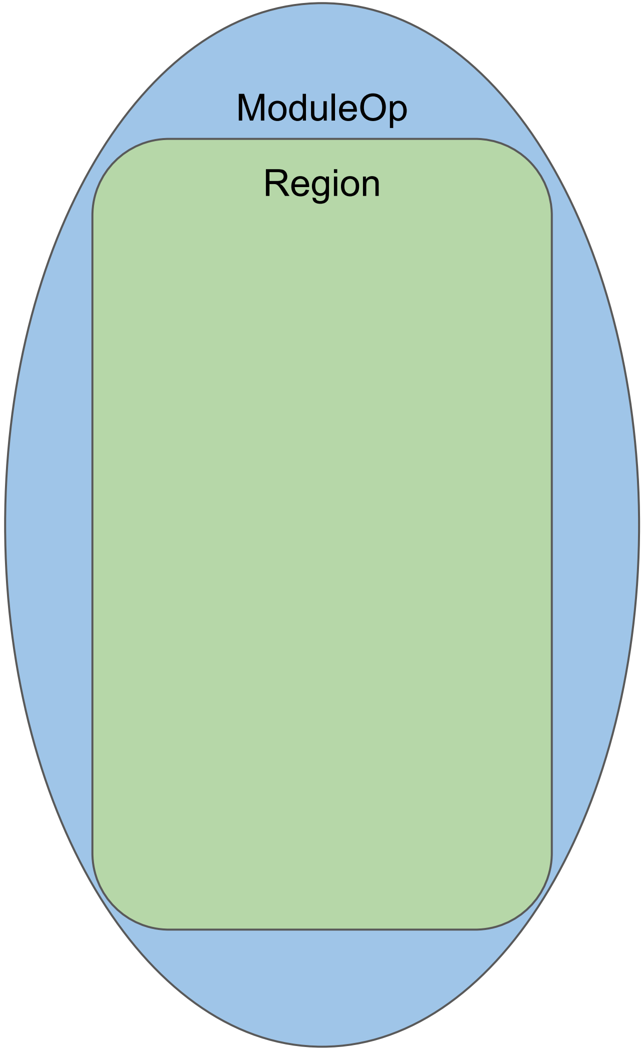


https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/

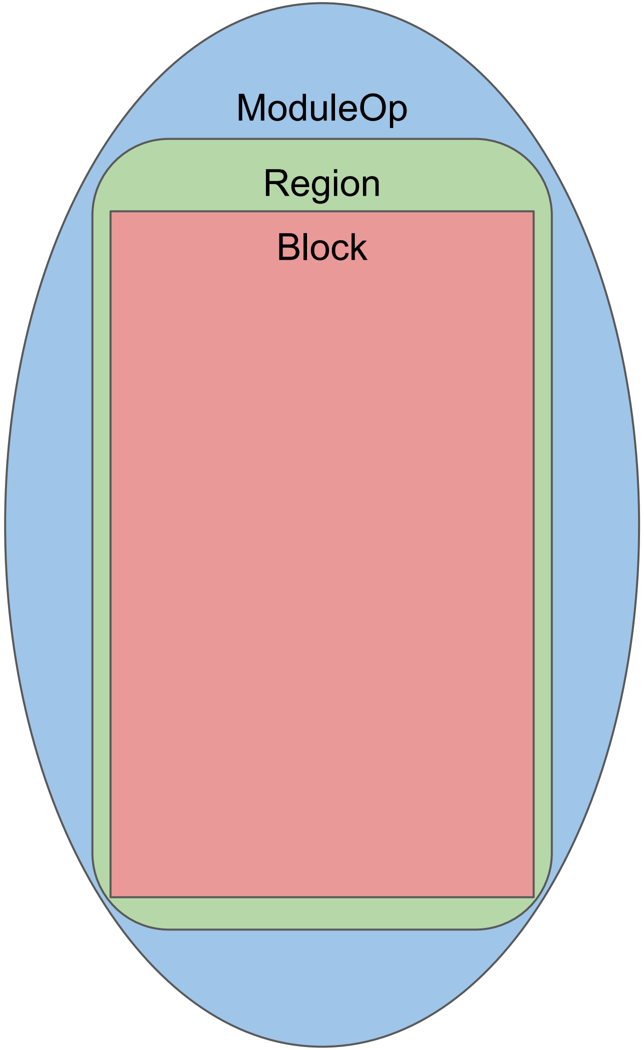
ModuleOp



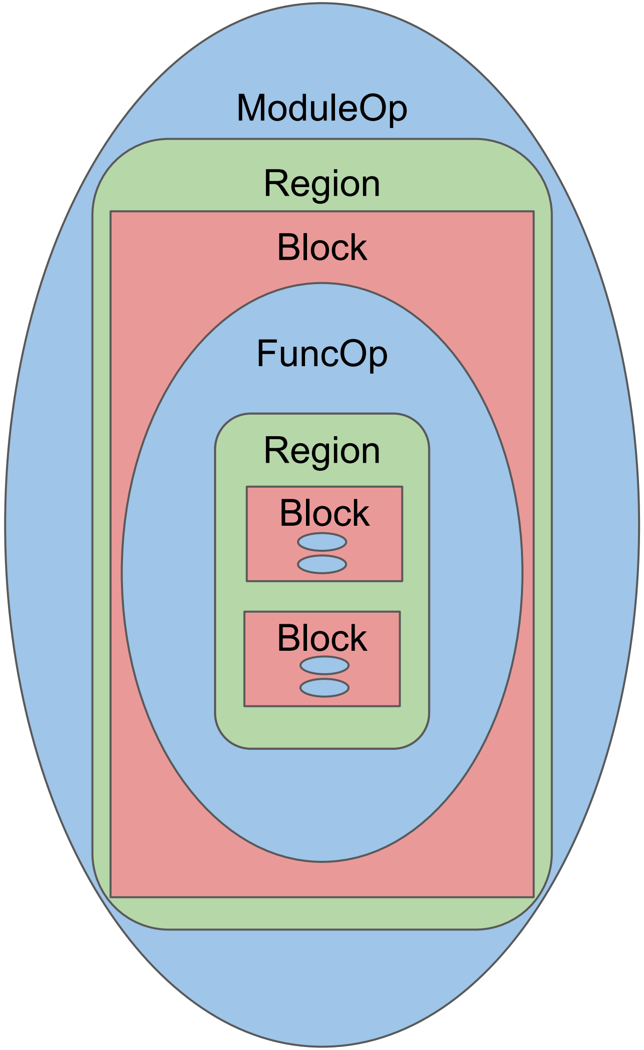
ModuleOp



ModuleOp



ModuleOp



LLVM Dialect MLIR

Dialects

This section contains documentation for core and contributed dialects available from the MLIR repository. The description for each dialect includes content automatically generated from the dialect's Definition Specification (ODS).

Dialects Docs

- 'acc' Dialect
- 'affine' Dialect
- 'amdgpu' Dialect
- 'amx' Dialect
- 'arith' Dialect
- 'arm_neon' Dialect
- 'arm_sve' Dialect
- 'ArmSME' Dialect
- 'async' Dialect
- 'bufferization' Dialect
- 'cf' Dialect
- 'complex' Dialect
- 'dlti' Dialect
- 'emitc' Dialect
- 'func' Dialect
- 'gpu' Dialect
- 'index' Dialect
- 'irdl' Dialect
- 'linalg' Dialect
- 'llvm' Dialect
- 'math' Dialect

https://mlir.llvm.org/docs/Dialects/

Dialects

This section contains documentation for core and contributed dialects available from the MLIR repository. The description for each dialect includes content automatically generated from the dialect's Definition Specification (ODS).

Dialects Docs

- 'acc' Dialect
- 'affine' Dialect
- 'amdgpu' Dialect
- 'amx' Dialect
- 'arith' Dialect
- 'arm_neon' Dialect
- 'arm_sve' Dialect
- 'ArmSME' Dialect
- 'async' Dialect
- 'bufferization' Dialect
- 'cf' Dialect
- 'complex' Dialect
- 'dlti' Dialect
- 'emitc' Dialect
- 'func' Dialect
- 'gpu' Dialect
- 'index' Dialect
- 'irdl' Dialect
- 'linalg' Dialect
- 'llvm' Dialect
- 'math' Dialect

https://mlir.llvm.org/docs/Dialects/

'llvm' Dialect

This dialect maps LLVM IR into MLIR by defining the corresponding operations and types. LLVM IR metadata is usually represented as MLIR attributes, which offer additional structure verification.

We use "LLVM IR" to designate the intermediate representation of LLVM and "LLVM *dialect*" or "LLVM IR *dialect*" to refer to this MLIR dialect.

Unless explicitly stated otherwise, the semantics of the LLVM dialect operations must correspond to the semantics of LLVM IR instructions and any divergence is considered a bug. The dialect also contains auxiliary operations that smoothen the differences in the IR structure, e.g., MLIR does not have `phi` operations and LLVM IR does not have a `constant` operation. These auxiliary operations are systematically prefixed with `mlir`, e.g. `llvm.mlir.constant` where `llvm.` is the dialect namespace prefix.

https://mlir.llvm.org/docs/Dialects/LLVM/

`llvm.add` (LLVM::AddOp) ¶

Syntax:

```
operation ::= `llvm.add` $lhs `,` $rhs custom<LLVMOpAttrs>(attr-dict) `:`
type($res)
```

Traits: AlwaysSpeculatableImplTrait, Commutative, SameOperandsAndResultType

Interfaces: ConditionallySpeculatable, InferTypeOpInterface, NoMemoryEffect (MemoryEffectOpInterface)

Effects: MemoryEffects::Effect{}

Operands: ¶

Operand	Description
`lhs`	integer or LLVM dialect-compatible vector of integer
`rhs`	integer or LLVM dialect-compatible vector of integer

Results: ¶

Result	Description
`res`	integer or LLVM dialect-compatible vector of integer

https://mlir.llvm.org/docs/Dialects/LLVM/

LLVM COMPILER INFRASTRUCTURE

LLVM Language Reference Manual

Documentation

- Getting Started/Tutorials
- User Guides
- Reference

Getting Involved

- Contributing to LLVM
- Submitting Bug Reports
- Mailing Lists
- IRC
- Meetups and Social Events

Additional Links

- FAQ
- Glossary
- Publications
- Github Repository

This Page

Show Source

Quick search

Go

https://llvm.org/docs/LangRef.html

- Instruction Reference
  - Terminator Instructions
    - ‘ret’ Instruction
    - ‘br’ Instruction
    - ‘switch’ Instruction
    - ‘indirectbr’ Instruction
    - ‘invoke’ Instruction
    - ‘callbr’ Instruction
    - ‘resume’ Instruction
    - ‘catchswitch’ Instruction
    - ‘catchret’ Instruction
    - ‘cleanupret’ Instruction
    - ‘unreachable’ Instruction
  - Unary Operations
    - ‘fneg’ Instruction
  - Binary Operations
    - ‘add’ Instruction
    - ‘fadd’ Instruction
    - ‘sub’ Instruction
    - ‘fsub’ Instruction
    - ‘mul’ Instruction
    - ‘fmul’ Instruction
    - ‘udiv’ Instruction
    - ‘sdiv’ Instruction
    - ‘fdiv’ Instruction
    - ‘urem’ Instruction
    - ‘srem’ Instruction
    - ‘frem’ Instruction

https://llvm.org/docs/LangRef.html

‘**add**’ Instruction ¶

Syntax:

```
<result> = add <ty> <op1>, <op2>          ; yields ty:result
<result> = add nuw <ty> <op1>, <op2>      ; yields ty:result
<result> = add nsw <ty> <op1>, <op2>      ; yields ty:result
<result> = add nuw nsw <ty> <op1>, <op2>  ; yields ty:result
```

Overview:

The ‘`add`’ instruction returns the sum of its two operands.

Arguments:

The two arguments to the ‘`add`’ instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the integer sum of the two operands.

If the sum has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n is the bit width of the result.

Because LLVM integers use a two’s complement representation, this instruction is appropriate for both signed and unsigned integers.

`nuw` and `nsw` stand for “No Unsigned Wrap” and “No Signed Wrap”, respectively. If the `nuw` and/or `nsw` keywords are present, the result value of the `add` is a poison value if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = add i32 4, %var          ; yields i32:result = 4 + %var
```

https://llvm.org/docs/LangRef.html#add-instruction

Builtin Dialect

The builtin dialect contains a core set of Attributes, Operations, and Types that have wide applicability across a very large number of domains and abstractions. Many of the components of this dialect are also instrumental in the implementation of the core IR. As such, this dialect is implicitly loaded in every `MLIRContext`, and available directly to all users of MLIR.

Given the far-reaching nature of this dialect and the fact that MLIR is extensible by design, any potential additions are heavily scrutinized.

- Attributes
  - AffineMapAttr
  - ArrayAttr
  - DenseArrayAttr
  - DenseIntOrFPElementsAttr
  - DenseResourceElementsAttr
  - DenseStringElementsAttr
  - DictionaryAttr
  - FloatAttr
  - IntegerAttr

https://mlir.llvm.org/docs/Dialects/Builtin/

Example

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```
module {

}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {









  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```mlir
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32















  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```mlir
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32











  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32










  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```mlir
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:

  ^bb2:









  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
     llvm.return %0 : i32
  ^bb2:









  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
     llvm.return %0 : i32
  ^bb2:
     %5 = llvm.srem %arg1, %2  : i32
     %6 = llvm.icmp "ne" %5, %0 : i32
     llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
     llvm.return %0 : i32
  ^bb4:

  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

LLVM Dialect MLIR

```mlir
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
     llvm.return %0 : i32
  ^bb2:
     %5 = llvm.srem %arg1, %2  : i32
     %6 = llvm.icmp "ne" %5, %0 : i32
     llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
     llvm.return %0 : i32
  ^bb4:
     llvm.return %1 : i32
  }
}
```

C

```c
int areBothEven(int a, int b) {
  if (a % 2 != 0)
     return 0;
  if (b % 2 != 0)
     return 0;
  return 1;
}
```

Handwritten translation.
As far as I know, there
is no complete reliable C
front end for MLIR.

LLVM Dialect MLIR

```mlir
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
     llvm.return %0 : i32
  ^bb2:
     %5 = llvm.srem %arg1, %2  : i32
     %6 = llvm.icmp "ne" %5, %0 : i32
     llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
     llvm.return %0 : i32
  ^bb4:
     llvm.return %1 : i32
  }
}
```

LLVM IR

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {









}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
     llvm.return %0 : i32
  ^bb2:
     %5 = llvm.srem %arg1, %2  : i32
     %6 = llvm.icmp "ne" %5, %0 : i32
     llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
     llvm.return %0 : i32
  ^bb4:
     llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2









}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2
  %4 = icmp ne i32 %3, 0










}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6
5:

6:






}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
     %0 = llvm.mlir.constant(0 : i32) : i32
     %1 = llvm.mlir.constant(1 : i32) : i32
     %2 = llvm.mlir.constant(2 : i32) : i32
     %3 = llvm.srem %arg0, %2  : i32
     %4 = llvm.icmp "ne" %3, %0 : i32
     llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
     llvm.return %0 : i32
  ^bb2:
     %5 = llvm.srem %arg1, %2  : i32
     %6 = llvm.icmp "ne" %5, %0 : i32
     llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
     llvm.return %0 : i32
  ^bb4:
     llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6
5:
  ret i32 0
6:








}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6
5:
  ret i32 0
6:
  %7 = srem i32 %1, 2
  %8 = icmp ne i32 %7, 0
  br i1 %8, label %9, label %10
9:
  ret i32 0
10:

}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6
5:
  ret i32 0
6:
  %7 = srem i32 %1, 2
  %8 = icmp ne i32 %7, 0
  br i1 %8, label %9, label %10
9:
  ret i32 0
10:
  ret i32 1
}
```

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

LLVM IR

```
define i32 @areBothEven(i32 %0, i32 %1) {
  %3 = srem i32 %0, 2
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %6
5:
  ret i32 0
6:
  %7 = srem i32 %1, 2
  %8 = icmp ne i32 %7, 0
  br i1 %8, label %9, label %10
9:
  ret i32 0
10:
  ret i32 1
}
```

Translated using the mlir-translate tool.

LLVM Dialect MLIR

```
module {
  llvm.func @areBothEven(%arg0: i32, %arg1: i32) -> i32 {
      %0 = llvm.mlir.constant(0 : i32) : i32
      %1 = llvm.mlir.constant(1 : i32) : i32
      %2 = llvm.mlir.constant(2 : i32) : i32
      %3 = llvm.srem %arg0, %2  : i32
      %4 = llvm.icmp "ne" %3, %0 : i32
      llvm.cond_br %4, ^bb1, ^bb2
  ^bb1:
      llvm.return %0 : i32
  ^bb2:
      %5 = llvm.srem %arg1, %2  : i32
      %6 = llvm.icmp "ne" %5, %0 : i32
      llvm.cond_br %6, ^bb3, ^bb4
  ^bb3:
      llvm.return %0 : i32
  ^bb4:
      llvm.return %1 : i32
  }
}
```

Tools

LLVM Tools

clang - C front end

  .c -> .ll | .bc

llvm-dis - disassembler

  .bc -> .ll

llvm-as - assembler

  .ll -> .bc

opt - optimizer and analyzer

  .ll | .bc -> .bc

lli - interpreter

  interpret .ll or .bc

llc - compiler

  .ll | .bc -> .o

MLIR Tools

mlir-opt - optimizer and lowerer

optimize mlir

lower mlir to lower level dialect

mlir-translate - translation tool

mlir -> external representation

external representation -> mlir

Spells

Spells

Spells

```
clang inFile.c -S -emit-llvm -o outFile.ll
```

```
  See what LLVM IR clang generates.
```

Spells

`clang inFile.c -S -emit-llvm -o outFile.ll`

See what LLVM IR clang generates.

`lli -dlopen=”pathToSharedLibrary” program.ll`

Execute LLVM IR program with a dynamic library.

Spells

`clang inFile.c -S -emit-llvm -o outFile.ll`

`See what LLVM IR clang generates.`

`lli -dlopen=”pathToSharedLibrary” program.ll`

`Execute LLVM IR program with a dynamic library.`

`mlir-translate inFile.llvm.mlir --mlir-to-llvmir -o outFile.ll`

`Translate LLVM dialect MLIR to LLVM IR.`

More Demo